

31st International Conference on Types for Proofs and Programs

TYPES 2025, June 9–13, 2025, University of Strathclyde,
Glasgow, Scotland

Edited by

Fredrik Nordvall Forsberg
James McKinna



Editors

Fredrik Nordvall Forsberg 

University of Strathclyde, Glasgow, Scotland
fredrik.nordvall-forsberg@strath.ac.uk

James McKinna 

Heriot-Watt University, Edinburgh, Scotland
J.McKinna@hw.ac.uk

ACM Classification 2012

Theory of computation → Categorical semantics; Theory of computation → Constructive mathematics;
Theory of computation → Denotational semantics; Theory of computation → Logic and verification;
Theory of computation → Type theory; Theory of computation → Quantum computation theory

ISBN 978-3-95977-441-3

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-441-3>.

Publication date

July, 2026

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists all publications of this volume in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2025.0

ISBN 978-3-95977-441-3

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

LIPICs – Leibniz International Proceedings in Informatics

LIPICs is a series of high-quality conference proceedings across all fields in informatics. LIPICs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Christel Baier (TU Dresden, DE)
- Roberto Di Cosmo (Inria and Université Paris Cité, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Holger Hermanns (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)
- Daniel Král' (Leipzig University, DE and Max Planck Institute for Mathematics in the Sciences, Leipzig, DE – Chair)
- Sławomir Lasota (University of Warsaw, PL)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Chih-Hao Luke Ong (Nanyang Technological University, SG)
- Eva Rotenberg (IT University of Copenhagen, DK)
- Pierre Senellart (ENS, Université PSL, Paris, France)
- Alexandra Silva (Cornell University, Ithaca, US)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Fredrik Nordvall Forsberg and James McKinna</i>	0:vii
List of Authors	
.....	0:ix
Papers	
Choice Principles and Hypercompletion in HoTT	
<i>Owen Milner</i>	1:1–1:14
Lean4Lean: Verifying a Typechecker for Lean, in Lean	
<i>Mario Carneiro</i>	2:1–2:23
Symmetries in Sorting	
<i>Vikraman Choudhury and Wind Wong</i>	3:1–3:23
HoTT Operads	
<i>Brandon Hewer and Graham Hutton</i>	4:1–4:24
A Graded Modal Type Theory for Pulse Schedules	
<i>Robin Adams, Jean-Philippe Bernardy, Lorenzo Perticone, and Jeremy Pope</i>	5:1–5:22
Formalisation and Extension of Lagois Connections for Secure Information Flow	
<i>Casper Ståhl, Levs Gondelman, René Rydhof Hansen, and Danny Bøgsted Poulsen</i>	6:1–6:22
Towards Fuzzy Constructive Type Theories	
<i>Besik Dundua, Furio Honsell, Temur Kutsia, Marina Lenisa, and Luigi Liquori</i> ..	7:1–7:19
The Rezk Completion for Elementary Topoi	
<i>Kobe Wullaert and Niels van der Weide</i>	8:1–8:23
Kleisli Categories with Display Maps	
<i>Moana Jubert</i>	9:1–9:18
Type-Theoretic Replacement and Univalent Completion: Applications and Interpretations	
<i>Evan Cavallo and Thierry Coquand</i>	10:1–10:27
Multi-Clocked Guarded Recursion Beyond ω	
<i>Rasmus Ejlers Møgelberg</i>	11:1–11:22
Nominal Type Theory by Nullary Internal Parametricity	
<i>Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese</i>	12:1–12:23
A Data Type of Intrinsically Plane Graphs in Agda	
<i>Malin Altenmüller and Conor Titania Mc Bride</i>	13:1–13:24
Functional Representability in Local Set Theories	
<i>Enrique Ruiz Hernández and Pedro Solórzano</i>	14:1–14:21
Mendler Dialgebras and Recursion Schemes of Mixed Variance	
<i>Stephan Alexander Spahn</i>	15:1–15:24

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ Preface

The TYPES meetings are a forum to present new and ongoing work in all aspects of type theory and its applications, especially in formalized and computer assisted reasoning and computer programming. This volume constitutes the post-proceedings of the 31st International Conference on Types for Proofs and Programs, TYPES 2025, that was held at the University of Strathclyde, Glasgow, Scotland, from 9 to 13 June 2025.

The meetings from 1990 to 2008 were annual workshops corresponding to five consecutive EU-funded networking projects. Since 2009, TYPES has been run as an independent conference series. Previous TYPES meetings were organised by Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal near Nijmegen (2002), Torino (2003), Jouy-en-Josas near Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Torino (2008), Aussois (2009), Warsaw (2010), Bergen (2011), Toulouse (2013), Paris (2014), Tallinn (2015), Novi Sad (2016), Budapest (2017), Braga (2018), Oslo (2019), Turin (2020), Leiden (2021), Nantes (2022), Valencia (2023), and Copenhagen (2024). The 2020 and 2021 editions were virtual, because of the SARS-CoV-2 pandemics.

The TYPES areas of interest include, but are not limited to: foundations of type theory; applications of type theory (e.g. linguistics or concurrency); constructive mathematics; dependently typed programming; industrial uses of type theory technology; meta-theoretic studies of type systems; proof assistants and proof technology; automation in computer-assisted reasoning; links between type theory and functional programming; formalizing mathematics using type theory; homotopy type theory and univalent mathematics.

The TYPES conferences are all based on contributed talks based on short abstracts; reporting work in progress and work presented or published elsewhere. A post-proceedings volume is prepared after the conference, whose papers must represent unpublished work. Submitted papers to the post-proceedings are subject to a full peer-review process.

The conference program of TYPES 2025 consisted of 82 contributed talks (of which 40 were 20 minutes long, and 42 were 15 minutes long), and four invited talks by Ingo Blechschmidt (University of Antwerp), Sonia Marin (University of Birmingham), Chris Martens (Northeastern University), and Christian Sattler (Chalmers University of Technology and University of Gothenburg). The programme also contained a co-located Women in EuroProofNet event dedicated to gender balance in our community, with Elaine Pimentel (University College London) as invited speaker. The conference was a successful event with 151 registered participants. All the details of the conference can be found at the conference website <https://msp.cis.strath.ac.uk/types2025/>.

For the post-proceedings, 22 papers were initially submitted, out of which 15 were accepted. We thank all the authors and reviewers for their hard work to make this possible.

Fredrik Nordvall Forsberg and James McKinna, June 2026.

■ List of Authors

Robin Adams  (5)
Chalmers University of Technology,
Gothenburg, Sweden;
University of Gothenburg, Sweden

Malin Altenmüller  (13)
University of Edinburgh, UK

Jean-Philippe Bernardy  (5)
Chalmers University of Technology,
Gothenburg, Sweden;
University of Gothenburg, Sweden

Mario Carneiro  (2)
Chalmers University of Technology &
University of Gothenburg, Sweden

Evan Cavallo  (10)
Department of Computer Science and
Engineering, University of Gothenburg and
Chalmers University of Technology,
Gothenburg, Sweden

Vikraman Choudhury  (3)
University of Strathclyde, Glasgow, UK;
Università di Bologna, Italy

Thierry Coquand  (10)
Department of Computer Science and
Engineering, University of Gothenburg and
Chalmers University of Technology,
Gothenburg, Sweden

Dominique Devriese  (12)
DistriNet, KU Leuven, Belgium

Besik Dundua  (7)
Giorgi Nikoladze Scientific-Research Center,
Kutaisi International University, Georgia;
Vekua Institute of Applied Mathematics,
Tbilisi State University, Georgia

Levs Gondelman  (6)
Aalborg University CPH,
Copenhagen, Denmark

René Rydhof Hansen  (6)
Aalborg University, Denmark

Brandon Hewer  (4)
School of Computer Science,
University of Nottingham, UK

Furio Honsell  (7)
Dipartimento di Scienze Matematiche,
Informatiche e Fisiche, Udine University, Italy

Graham Hutton  (4)
School of Computer Science,
University of Nottingham, UK

Moana Jubert  (9)
Inria Paris, Université Paris-Cité, IRIF, CNRS,
Paris, France

Temur Kutsia  (7)
Research Institute for Symbolic Computation,
Johannes Kepler University Linz, Austria

Marina Lenisa  (7)
Dipartimento di Scienze Matematiche,
Informatiche e Fisiche, Udine University, Italy

Luigi Liquori  (7)
Centre Inria de l'Université Côte d'Azur,
Sophia Antipolis, France

Conor Titania Mc Bride  (13)
University of Strathclyde, Glasgow, UK;
Quantinuum, Cambridge, UK

Owen Milner  (1)
Department of Philosophy, Carnegie Mellon
University, Pittsburgh, PA, USA

Rasmus Ejlers Møgelberg  (11)
IT University of Copenhagen, Denmark

Andreas Nuyts  (12)
DistriNet, KU Leuven, Belgium

Lorenzo Perticone  (5)
Chalmers University of Technology,
Gothenburg, Sweden;
University of Gothenburg, Sweden

Jeremy Pope  (5)
Chalmers University of Technology,
Gothenburg, Sweden;
University of Gothenburg, Sweden

Danny Bøgsted Poulsen  (6)
Aalborg University, Denmark

Enrique Ruiz Hernández  (14)
Centro de Investigación en Teoría de Categorías
y sus Aplicaciones, A.C, Mexico

Pedro Solórzano  (14)
Instituto de Matemáticas, Universidad Nacional
Autónoma de México, Oaxaca de Juárez, Mexico

Stephan Alexander Spahn  (15)
Department of Computer Science,
Tallinn University of Technology, Estonia

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Casper Ståhl  (6)
Aalborg University, Denmark

Niels van der Weide  (8)
Radboud University,
Nijmegen, The Netherlands

Antoine Van Muylder  (12)
DistriNet, KU Leuven, Belgium

Wind Wong  (3)
Vrije Universiteit Amsterdam,
The Netherlands

Kobe Wullaert  (8)
Delft University of Technology,
The Netherlands

Choice Principles and Hypercompletion in HoTT

Owen Milner 

Department of Philosophy, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Building on work of Anel and Barton [1], we will introduce the construction of a hypercompletion modality [8] in HoTT under the assumption that countable choice holds. Central results have been formalized in Cubical Agda [5] and will be referenced where appropriate.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification

Keywords and phrases HoTT, hypercompletion, modalities

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.1

Funding This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-21-1-0009, PI Steve Awodey.

Acknowledgements I want to thank Mathieu Anel, Steve Awodey, Reid Barton, and Hugo Moeneclaey for helpful discussions.

1 Introduction

We work in homotopy type theory [9]. We make ready use of the elimination rule (J -rule) for intensional identity types, the univalence axiom, and higher inductive types. Modalities were studied intensively in the setting of HoTT by Rijke, Shulman and Spitters [8]. We will briefly revisit some of their terminology and key theorems in the preliminaries section, alongside a few other established HoTT results and definitions that we will need going forward. After that we will spend some time on the concepts of countable products and countable inverse limits in HoTT. Some of that material has been formalized by the author. The two sections subsequent to that contain the main results of this work and are fully formalized. There is a description of the choice principles identified by Anel and Barton [1], their relationship to Postnikov towers, and a proof that a certain operation is a modality which presents the hypercomplete types. We conclude with an application to the setting of synthetic Stone duality, unformalized as of the time of writing.

The question whether a hypercompletion operation can be defined in HoTT was posed in [8]. The work presented here (esp. the formalized portion) is a partial resolution to that question in the case where we are allowed to assume further classical axioms (variants of countable or dependent choice).

2 Preliminaries

We follow the notation of (e.g.) [10]. So if P is a type-family over A , then the product type is $(a : A) \rightarrow P(a)$, and the sum type is $(a : A) \times P(a)$. We will write $\mathcal{U}$ for the universe of all (small) types, $\mathcal{U}^{\leq -1}$ for the type of propositions and $\mathcal{U}^{\leq n}$ for the type of n -types. We will write $|\bullet|_n$ for the universal map $X \rightarrow \|X\|_n$ for any type X .

► **Theorem 1** (Fundamental theorem of identity types). *If A is pointed by a and $P : A \rightarrow \mathcal{U}$ is a family of types such that $P(a)$ is pointed, and $(x : A) \times P(x)$ is contractible with the point of $P(a)$ as the center of contraction, then $P(x) \simeq (x = a)$ naturally in x .*

Proof. Theorem 11.2.2 in [7]. ◀



© Owen Milner;

licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 1; pp. 1:1–1:14

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

► **Definition 2** (Pullback). If $f : A \rightarrow B \leftarrow C : g$ is a cospan of maps, then the pullback $A \times_B C$ is the type $(a : A) \times (c : C) \times (f(a) = g(c))$.

The following definitions, propositions and examples come from [8].

► **Definition 3** (Modality). Suppose $\bigcirc : \mathcal{U} \rightarrow \mathcal{U}$ is an operation on the universe with a unit $\eta : (X : \mathcal{U}) \rightarrow X \rightarrow \bigcirc X$, we say that the pair $(\bigcirc, \eta)$ is a (uniquely eliminating) modality if the map

$$\lambda f. f \circ \eta_X : ((x : \bigcirc X) \rightarrow \bigcirc P(x)) \rightarrow ((x : X) \rightarrow \bigcirc P(\eta_X(x)))$$

is an equivalence for all $X : \mathcal{U}$ and $P : \bigcirc X \rightarrow \mathcal{U}$.

► **Definition 4** (Nullification). If $(\bigcirc, \eta)$ is a modality, and X is a type, we say that X is nullified by the modality if $\bigcirc X$ is contractible.

► **Definition 5**. If $(\bigcirc, \eta)$ is a modality and $f : A \rightarrow B$ is a map, we say that f is nullified by the modality if all its fibers are nullified in the sense of the previous definition.

► **Definition 6** (Σ -closed reflective subuniverse). A Σ -closed reflective subuniverse is a triple $(M, \bigcirc, \eta)$ where $\bigcirc$ and η are an operation and a unit as before, and M is a function taking types to propositions, such that

1. $M(\bigcirc X)$ holds for all types X
2. For any type X and any type B such that $M(B)$, the map

$$\lambda f. f \circ \eta_X : (\bigcirc X \rightarrow B) \rightarrow (X \rightarrow B)$$

is an equivalence

3. If X is any type, and $P : X \rightarrow \mathcal{U}$, and $M(X)$, and $M(P(x))$ for all $x : X$, then

$$M((x : X) \times P(x))$$

► **Proposition 7** (Functoriality of $\bigcirc$). If $(M, \bigcirc, \eta)$ is a reflective subuniverse, then there is an operation $(A, B : \mathcal{U}) \rightarrow (A \rightarrow B) \rightarrow (\bigcirc A \rightarrow \bigcirc B)$ which for convenience we shall label $\bigcirc$ also. This is functorial in the sense that

- $\bigcirc(\text{id}_A) = \text{id}_{\bigcirc A}$, and
- $\bigcirc(g \circ f) = \bigcirc g \circ \bigcirc f$

Proof. Lemma 1.21 in [8]. ◀

► **Proposition 8**. If $(M, \bigcirc, \eta)$ and $(M', \bigcirc', \eta')$ are two Σ -closed reflective subuniverses, then they are equal in the type of reflective closed subuniverses if and only if (the proposition) $(M(X) \Leftrightarrow M'(X))$ is inhabited for all types X .

Proof. Theorem 1.18 in [8]. ◀

► **Definition 9**. A (pullback-)stable orthogonal factorization system is a pair of predicates

$$L, R : (A, B : \mathcal{U}) \rightarrow (A \rightarrow B) \rightarrow \mathcal{U}^{\leq -1}$$

such that

- For any $f : A \rightarrow B$, and $g : B \rightarrow C$, if $L(f)$ and $L(g)$, then $L(g \circ f)$, and likewise for R .
- If $f : A \rightarrow B$ is an isomorphism, then $L(f)$ and $R(f)$.

■ For all $f : A \rightarrow B$, the type

$$(C : \mathcal{U}) \times (g : A \rightarrow C) \times (h : C \rightarrow B) \times (f = hg) \times L(g) \times R(h)$$

of factorizations of the map into an L -map and an R -map is contractible, and such factorization diagrams are preserved by pullbacks.

► **Proposition 10.** *If (L, R) is a stable orthogonal factorization system, then R is precisely the type of maps which satisfy right-lifting with respect to all maps in L and vice versa.*

Proof. Lemma 1.46 in [8]. ◀

► **Theorem 11** (Modalities and reflective subuniverses). *The type of modalities is equivalent to the types of*

1. Σ -closed reflective subuniverses
2. Stable orthogonal factorization systems

where, in particular

1. M is the proposition that $\eta(X)$ is an equivalence, if this holds we say that X is a modal type.
2. The maps L are those which are nullified by the modality. The maps in R are those with modal fibers; the units η always lie in L .

Hence, in particular, a modality is uniquely determined by the subuniverse of types such that η is an equivalence.

Proof. Using Theorems 1.13, 1.16, 1.34 and 1.54 in [8]. ◀

► **Example 12** (n -Truncation). Fix a natural number $n : \mathbb{N}$, the pair $(\lambda X. \|X\|_n, \lambda X. |\bullet|_n)$ is a modality.

The corresponding subuniverse is the type of n -truncated types, and the left maps in the corresponding factorization system are the n -connected maps.

► **Definition 13** (Left-exact modality). *If $(\bigcirc, \eta)$ is a modality, and $B \rightarrow A \leftarrow C$ is a cospan of types, then there is a canonical map*

$$(\bigcirc B \times_{\bigcirc A} \bigcirc C) \rightarrow \bigcirc (B \times_A C)$$

We say that $(\bigcirc, \eta)$ is a left-exact modality if this map is an equivalence for all spans.

One consequence is that the maps which are nullified by the modality coincide with the maps whose image under $\bigcirc$ is an equivalence.

► **Proposition 14.** *The 0-truncation modality is not left-exact. This proposition generalizes to all n .*

Proof. Consider the pullback square:

$$\begin{array}{ccc} \mathbb{Z} & \longrightarrow & \mathbf{1} \\ \downarrow & \lrcorner & \downarrow \\ \mathbf{1} & \longrightarrow & S^1 \end{array}$$

$\|\mathbb{Z}\|_0 = \mathbb{Z}$, but $\|S^1\|_0 = \mathbf{1}$, so we would be asking for an equivalence between $\mathbf{1}$ and $\mathbb{Z}$.

More generally we can take a pullback square:

$$\begin{array}{ccc} K(\mathbb{Z}, n) & \longrightarrow & \mathbf{1} \\ \downarrow & \lrcorner & \downarrow \\ \mathbf{1} & \longrightarrow & K(\mathbb{Z}, n+1) \end{array}$$

And similar reasoning applies. ◀

Before returning to modalities, we'll list some more useful facts about connected maps.

► **Proposition 15.** *If*

$$\begin{array}{ccc} A & \xrightarrow{u} & X \\ \downarrow f & & \downarrow g \\ B & \xrightarrow{v} & Y \end{array}$$

commutes, and the right-most map is n -truncated, and the left-most map is n -connected, then the type of fillers for the square is contractible. In fact, this characterizes n -connected maps.

Proof. Refer to Proposition 10, Theorem 11 and Example 12. ◀

► **Proposition 16.** *If A is n -connected and $P(a)$ is n -connected for each $a : A$, then $(a : A) \times P(a)$ is also n -connected.*

Proof. Follows immediately from Theorem 7.3.9 in [9]. ◀

► **Proposition 17.** *If $f : A \rightarrow B$ is n -connected, then $\|f\|_n$ is an equivalence. The converse is not necessarily true. However, $f : \|A\|_n \rightarrow \|B\|_n$ is an equivalence if and only if f is n -connected.*

Proof. For an example of a map such that $\|f\|_0$ is an equivalence, but the map is not 0-connected, see the proof of Proposition 14. The map $\mathbf{1} \rightarrow S^1$ is as specified. ◀

► **Proposition 18.** *Suppose $f = g \circ h$ and f and h are n -connected, then so is g .*

Proof. Immediate using the orthogonal factorization system. ◀

► **Proposition 19.** *If A is $(n+1)$ -connected, then $x =_A y$ is n -connected for all $x, y : A$. If f is $(n+1)$ -connected, then all $\text{ap}(f)$ are n -connected.*

As a partial converse: if A is merely inhabited and $x =_A y$ are all n -connected, then A is $(n+1)$ -connected. If f is surjective (i.e. (-1) -connected) and all $\text{ap}(f)$ are n -connected, then f is $(n+1)$ -connected.

Proof. The first statement follows from Theorem 7.3.12 in [9].

According to (amongst others) Leclerc [3, p. 8], the converse is due to Coquand. If A satisfies the hypothesis of the converse, then $\|A\|_{n+1}$ is a proposition, using the theorem cited above. Then $\|A\|_{n+1} \simeq \|A\|_{-1}$ and the latter type is contractible by assumption. The corresponding statement for maps follows readily. ◀

► **Proposition 20.** *Suppose $f = g \circ h$ and f and g are $(n+1)$ -connected, then h is n -connected.*

Proof. Follows from Proposition 19. ◀

► **Corollary 21.** *If $\|f\|_{n+1}$ is an equivalence, then f is n -connected.*

Back to modalities, drawing still on [8, esp. Section 3].

► **Example 22** (Open modality). If P is a proposition, then there is a modality whose operation is

$$\lambda X. (P \rightarrow X)$$

This modality is left-exact.

► **Definition 23** (Topological modality). A modality is called *topological* if there is some (type-indexed) family of propositions P such that the modal types X (hence the whole modality) are characterized by the lifting properties for the squares

$$\begin{array}{ccc} P & \longrightarrow & X \\ \downarrow & & \downarrow \\ \mathbf{1} & \xlongequal{\quad} & \mathbf{1} \end{array}$$

Clearly each P is nullified by the modal operator.

Clearly the open modality is topological.

► **Proposition 24.** Topological modalities are lex.

Proof. Corollary 3.12 in [8]. ◀

► **Definition 25.** A map f is called ∞ -connected if it is n -connected for every n .

► **Proposition 26.** If A, B are types, $P : A \rightarrow \mathcal{U}$, and $Q : B \rightarrow \mathcal{U}$, and there are maps $f : A \rightarrow B$, and $g_a : P(a) \rightarrow Q(f(a))$ for all $a : A$, each of which is ∞ -connected, then $\lambda(a, p). (f(a), g_a(p)) : (a : A) \times P(a) \rightarrow (b : B) \times Q(b)$ is ∞ -connected.

Proof. The fiber is $(a : A) \times (p : P(a)) \times (q : f(a) = b) \times (g_a(p) =^q p')$. This is equivalent to $(a : A) \times (q : f(a) = b) \times (p : P(a)) \times (g_a(p) =^q p')$ and that this is connected follows from the hypothesis. ◀

► **Definition 27.** A type X is called *hypercomplete* if it has right lifting against all ∞ -connected maps.

► **Proposition 28.** Having left lifting against all maps with hypercomplete fibers characterizes ∞ -connected maps.

Proof. We are trying to prove the bi-implication:

$$f \text{ is } \infty\text{-connected} \Leftrightarrow f \text{ has left lifting against all maps with hypercomplete fibers}$$

The left-to-right direction is immediate.

For the right-to-left direction, observe that the fibers of the map

$$\|f\|_{n+1} : \|X\|_{n+1} \rightarrow \|Y\|_{n+1}$$

will be $(n+1)$ -truncated, hence hypercomplete. So the type of fillers for the square:

$$\begin{array}{ccc} X & \longrightarrow & \|X\|_{n+1} \\ \downarrow f & & \downarrow \|f\|_{n+1} \\ Y & \longrightarrow & \|Y\|_{n+1} \end{array}$$

is contractible, so there is some $u : Y \rightarrow \|X\|_{n+1}$, this map can be factorized like so:

$$\begin{array}{ccc} Y & \xrightarrow{u} & \|X\|_{n+1} \\ & \searrow & \nearrow u' \\ & \|Y\|_{n+1} & \end{array}$$

and it is straightforward to show that the map u' is an inverse for $\|f\|_{n+1}$. So the theorem follows using Corollary 21. $\blacktriangleleft$

► **Definition 29** (Hypercompletion Modality). *A hypercompletion modality if it exists is a modality whose modal types are precisely the hypercomplete types.*

In the corresponding orthogonal factorization system, the left maps will be precisely the ∞ -connected maps. The right maps will be those with hypercomplete fibers.

By Proposition 8 the hypercompletion is unique if it exists. Obviously there is some analogy between hypercompletion and truncation, but there are important differences too.

► **Proposition 30** (Hypercompletion is lex). *If the hypercompletion modality exists, then the units must be ∞ -connected. This implies that a hypercompletion modality must be left exact.*

Proof. Suppose f is nullified by the modal operator, then it is ∞ -connected, in particular η is nullified by the modal operator. By Proposition 28.

The canonical map $A \times_B C \rightarrow \bigcirc A \times_{\bigcirc B} \bigcirc C$ is given by $\lambda(x, y, p).(\eta(x), \eta(y), \text{ap}_\eta(p))$, and ap_η is ∞ -connected because η is by Proposition 19, as are the other two components. A triple of ∞ -connected maps is ∞ -connected by Proposition 26. There is nothing else to show. $\blacktriangleleft$

► **Definition 31.** *A modality is called cotopological if all propositions it nullifies were already contractible. This implies, the only modality that is both topological and cotopological is trivial.*

► **Proposition 32** (Hypercompletion is cotopological). *Any proposition nullified by hypercompletion is trivial.*

3 Countable limits

► **Definition 33.** *A (countable) family is a function $\mathbb{N} \rightarrow \mathcal{U}$.*

► **Definition 34.** *A (countable) tower is a family A together with a dependent family of functions, $a : (n : \mathbb{N}) \rightarrow A_{n+1} \rightarrow A_n$.*

► **Definition 35.** *The product of a family is the type of dependent functions*

$$\prod A := (n : \mathbb{N}) \rightarrow A_n$$

► **Definition 36.** *The limit of a tower is the type*

$$\lim(A, a) := \left(x : \prod A \right) \times ((n : \mathbb{N}) \rightarrow a_n(x(n+1)) = x(n))$$

Abusing notation a bit, if $x : \lim(A, a)$, we will often write x for the first component and q_x for the second component. We might suppress the map in the tower and just write $\lim A$ when it's clear from context.

► **Proposition 37.** *If (A_{n+k}, a_{n+k}) is the tower obtained from A by shifting k places up, then $\lim(A_{n+k}, a_{n+k}) = \lim(A, a)$.*

Proof. There is an obvious map $\lim(A, a) \rightarrow \lim(A_{n+k}, a_{n+k})$, which has contractible fibers as extensions of a term in the latter limit are essentially unique by definition. ◀

► **Definition 38.** *If X is a type, there is a constant tower given by X and the identity. Its limit is X .*

► **Proposition 39.** *If (A, a) is a tower, X is a type, $f : A_0 \rightarrow X$, then there is another tower A' s.t. $A'_{n+1} = A_n$, and $A'_0 = X$. The limit of the new tower is $\lim(A, a)$ still.*

► **Definition 40.** *A map of families is a family of maps.*

► **Definition 41.** *Tower maps or ladders are families of maps $f_n : A_n \rightarrow B_n$ such that all the squares commute.*

$$\begin{array}{ccc}
 \vdots & & \vdots \\
 \downarrow & & \downarrow \\
 A_{n+1} & \xrightarrow{f_{n+1}} & B_{n+1} \\
 \downarrow & & \downarrow \\
 A_n & \xrightarrow{f_n} & B_n \\
 \downarrow & & \downarrow \\
 \vdots & & \vdots
 \end{array}$$

Write $\text{hom}((A, a), (B, b))$ for the type of such ladders.

Abusing notation a bit, if $f : \text{hom}((A, a), (B, b))$ we will often write f for the first component and p_f for the second component.

► **Proposition 42.** *If $(A, a), (B, b)$ are two towers then*

$$((A, a) = (B, b)) \simeq ((f : \text{hom}((A, a), (B, b))) \times (n : \mathbb{N}) \rightarrow \text{isEquiv}(f_n))$$

Proof. We will apply the fundamental theorem of identity types (Theorem 1).

The family $P := \lambda(B, b).((f : \text{hom}((A, a), (B, b))) \times (n : \mathbb{N}) \rightarrow \text{isEquiv}(f_n))$ is such that:

1. $P(A, a)$ is pointed, by the ladder of identity maps on the A_n .
2. We have the following chain of natural equivalences:

$$\begin{aligned}
 & (B : \mathbb{N} \rightarrow \mathcal{U}) \times (b : (n : \mathbb{N}) \rightarrow B_{n+1} \rightarrow B_n) \\
 & \quad \times ((f, e) : \mathbb{N} \rightarrow (A_n \simeq B_n)) \times (p_f : (n : \mathbb{N}) \rightarrow f_n \circ a_n = b_n \circ f_{n+1}) \\
 & \simeq ((B : \mathbb{N} \rightarrow \mathcal{U}) \times (f : (n : \mathbb{N}) \rightarrow B_n \simeq A_n)) \\
 & \quad \times ((b : (n : \mathbb{N}) \rightarrow B_{n+1} \rightarrow B_n) \times ((n : \mathbb{N}) \rightarrow b_n = f_n a_n f_{n+1}^{-1}))
 \end{aligned}$$

which is contractible.

So we may apply the fundamental theorem of identity types and the proposition follows. ◀

► **Proposition 43.** *Tower maps induce maps between limits which are equivalences if their respective components are.*

Proof. If $x = (x, q_x)$ is a term of type $\lim A$, then $\lambda n. f_n(x(n))$ is a term of type $\prod B$, and for each $n : \mathbb{N}$, $p_f(n) \cdot \text{ap}_{f_n}(q_x(n)) : b_n(f_{n+1}(x(n+1))) =_{B_n} f_n(x(n))$, so $\lambda n. f_n(x(n))$ forms a term of type $\lim B$.

By Proposition 42, there is an equivalence between the identity type of the type of towers, and the type of ladders of equivalences between the towers. So, we can appeal to the J-rule and the observation that the above construction on the identity ladder yields the identity map on the limit. $\blacktriangleleft$

► **Proposition 44.** *If X is a type and for each $x : X$, we have a tower $P(x)$, then there is a tower whose types are*

$$(x : X) \rightarrow P_n(x)$$

and whose arrows come from composition $(x : X) \rightarrow P_{n+1}(x) \rightarrow P_n(x)$

There is an equivalence of types

$$((x : X) \rightarrow \lim P(x)) \simeq \lim((x : X) \rightarrow P_n(x))$$

given by postcomposition with the projection maps from $\lim P(x)$.

Proof. This is a consequence of the so-called “type theoretic axiom of choice” not to be confused with the axioms of choice we discuss below

$$\begin{aligned} (x : X) \rightarrow \lim P(x) &\simeq (x : X) \rightarrow \left((x' : \prod P(x)) \times a_n(x'(n+1)) = x'(n) \right) \\ &\simeq (x' : (x : X) \rightarrow \prod P(x)) \\ &\quad \times ((x : X) \rightarrow (n : \mathbb{N}) \rightarrow a_n(x'(x)(n+1)) = x'(x)(n)) \\ &\simeq (x' : \prod((x : X) \rightarrow P(x))) \\ &\quad \times ((n : \mathbb{N}) \rightarrow a_n \circ x'(n+1) = x'(n)) \\ &\simeq \lim((x : X) \rightarrow P(x)) \end{aligned}$$

If f is a function in $(x : X) \rightarrow \lim P(x)$, then the equivalence on the second line sends it to $(\lambda x. f(x), \lambda x. q_{f(x)})$, which is composed with the projections through the equivalence on the third line as claimed. $\blacktriangleleft$

► **Proposition 45.** *A lifting problem*

$$\begin{array}{ccc} Y & \longrightarrow & \lim A \\ \downarrow & & \\ Z & & \end{array}$$

has a solution if and only if each of the lifting problems

$$\begin{array}{ccc} Y & \longrightarrow & A_n \\ \downarrow & & \\ Z & & \end{array}$$

has a solution.

Proof. Immediate from Proposition 44. This has been formalized in [5] (link). $\blacktriangleleft$

► **Proposition 46.** *If $x, y : \prod A$, then $(x = y) \simeq \prod (x_n = y_n)$.*

Proof. This is just function extensionality [9]. ◀

► **Proposition 47.** *If $x, y : \lim A$, then there is a tower whose types are $x_n =_{A_n} y_n$, and whose maps are $\lambda(p : x_{n+1} = y_{n+1}). q_x(n)^{-1} \cdot \text{ap}_{a_n}(p) \cdot q_y(n)$, and we have:*

$$(x = y) \simeq \lim(x_n = y_n)$$

Proof. By well-known facts about the identity types of sum types, we have the following:

$$(x = y) \simeq (p : x = y) \times q_x =^p q_y$$

which by function extensionality becomes

$$(p : x = y) \times ((n : \mathbb{N}) \rightarrow q_x(n) =^p q_y(n))$$

Now using the J-rule, we can show that for any $(x, y : \lim A)$, and $p : x = y$

$$(q_x(n) =^p q_y(n)) \simeq (q_x(n)^{-1} \cdot \text{ap}_{a_n}(p_{n+1}) \cdot q_y(n) = p_n)$$

which implies that our original type is in fact equivalent to:

$$(p : \prod x_n = y_n)((n : \mathbb{N}) \rightarrow q_x(n)^{-1} \cdot \text{ap}_{a_n}(p_{n+1}) \cdot q_y(n) = p_n)$$

Which is precisely the limit of the tower defined before, which completes the proof. ◀

► **Proposition 48.** *If $A_n \rightarrow B_n \leftarrow C_n$ is a family of cospans, then $\prod A \rightarrow \prod B \leftarrow \prod C$ is a cospan of products, and $\prod A \times_{\prod B} \prod C \simeq \prod (A_n \times_{B_n} C_n)$.*

► **Proposition 49.** *If $(A, a), (B, b), (C, c)$ are three families of maps, and there are ladders $f : A \rightarrow B$ and $g : C \rightarrow B$, then there is a tower $A_n \times_{B_n} C_n$, and the limit of this tower is equivalent to the pullback $\lim A \times_{\lim B} \lim C$*

Proof. If $(A, a), (B, b), (C, c)$ are three towers, and $f_n : A_n \rightarrow B_n$ and $g_n : C_n \rightarrow B_n$ are two tower-maps, then there are maps $A_{n+1} \times_{B_{n+1}} C_{n+1} \rightarrow A_n \times_{B_n} C_n$ given by $\lambda(x, y, p).(a_n(x), c_n(y), p_f(n)^{-1} \cdot \text{ap}_{b_n}(p) \cdot p_g(n))$. We will write H for the function in the third argument, for convenience.

Now,

$$\begin{aligned} \lim A \times_{\lim B} \lim C &\simeq (x : \lim A) \times (y : \lim C) \times f(x) = g(y) \\ &\simeq (x : \lim A) \times (y : \lim C) \times \lim(f_n(x_n) = g_n(y_n)) \end{aligned}$$

Using Proposition 47, then the proposition follows by the type-theoretic axiom of choice. Observe that the term $\lambda n.f_n(x(n))$ of type $\lim B$ has for its second component the term

$$\begin{aligned}
& p_f(n) \cdot \text{ap}_{f_n}(q_x(n)) : b_n(f_{n+1}(x(n+1))) = f_n(x(n)), \text{ similarly for } \lambda n. g_n(y(n)) \\
& (x : \lim A) \times (y : \lim C) \times \lim(f_n(x_n) = g_n(y_n)) \\
& \simeq (x : \prod A) \times (q_x : (n : \mathbb{N}) \rightarrow a(x(n+1)) = x(n)) \times \\
& \quad (y : \prod C) \times (q_y : (n : \mathbb{N}) \rightarrow c(y(n+1)) = y(n)) \times \\
& \quad (p : \prod f_n(x(n)) = g_n(y(n))) \times \\
& \quad ((n : \mathbb{N}) \rightarrow \text{ap}_{f_n}(q_x(n))^{-1} \cdot p_f(n)^{-1} \cdot p(n+1) \cdot p_g(n) \cdot \text{ap}_{g_n}(q_y(n)) = p(n)) \\
& \simeq (x : \prod A) \times (y : \prod C) \times (p : \prod f_n(x(n)) = g_n(y(n))) \times \\
& \quad ((n : \mathbb{N}) \rightarrow (q_x : a(x(n+1)) = x(n)) \times (q_y : c(y(n+1)) = y(n)) \times \\
& \quad \quad (H(p(n+1)) = \text{ap}_f(q_x, \text{ap}_g(q_y)) p(n))) \\
& \simeq (x : \prod A) \times (y : \prod C) \times (p : \prod f_n = g_n) \times \\
& \quad ((n : \mathbb{N}) \rightarrow (a(x(n+1)), c(y(n+1)), H(p(n+1))) = (x(n), y(n), p(n))) \\
& \simeq \lim(\lambda n. A_n \times_{B_n} C_n, \lambda n. \lambda(x, y, p). (a(x), c(y), H(p)))
\end{aligned}$$

Using the fact about pullbacks of products above, since the map of the triples is exactly as given here. $\blacktriangleleft$

4 Choice principles

The following definitions and propositions appear in [1]. They have been formalized [5].

► **Definition 50.** *Anel-Barton choice principles*

We say that countable choice of dimension $\leq d$ holds if whenever A, B are families of types, and $f : A_n \rightarrow B_n$ is a family of $(k+d)$ -connected maps between them, then the map $\prod f : \prod A \rightarrow \prod B$ is k -connected.

We say that dependent choice of dimension $\leq d$ holds if whenever $(A, a), (B, b)$ are towers, and $f : A_n \rightarrow B_n$ is a ladder of $(k+d)$ -connected maps between them, then the map $\lim f : \lim A \rightarrow \lim B$ is k -connected.

► **Proposition 51.** *Countable choice of dimension $\leq d$ holds if and only if, whenever A is a family of $(k+d)$ -connected types, then $\prod A$ is k -connected.*

Proof. The fiber of f is the product of the fibers of the f_n by Proposition 48. In the other direction consider the constant family on **1**. $\blacktriangleleft$

► **Proposition 52.** *Dependent choice of dimension $\leq d$ holds if and only if, whenever (A, a) is a family of $(k+d)$ -connected types, then $\lim(A, a)$ is k -connected.*

Proof. The fiber of f is the limit of the fibers of f_n using Proposition 49. In other direction consider constant family on **1**. $\blacktriangleleft$

► **Proposition 53.** *Dependent choice of dimension $\leq d$ implies countable choice of the same dimension*

Proof. $\prod A_n$ is the limit of $A_1 \times \cdots \times A_k$, the latter are $d+n$ connected if each of the A_i are. $\blacktriangleleft$

► **Proposition 54.** *Countable choice of dimension $\leq d$ implies dependent choice of dimension $\leq d+1$.*

Proof. Suppose the hypothesis holds, and (A, a) is a tower whose types are all $n + d + 1$ connected. Then $\prod A$ is $n + 1$ connected. For $x : \prod A$, each identity type $a_n(x(n+1)) = x(n)$ is $n + d$ connected, because of Proposition 19, so $(n : \mathbb{N}) \rightarrow a_n(x(n+1)) = x(n)$ is n -connected by countable choice. It follows that the limit, being a sum of n -connected types over an $n + 1$ connected type is itself n -connected. There is nothing else to show. $\blacktriangleleft$

5 Postnikov effectiveness and hypercompletion

The Postnikov decomposition of a space was introduced by Postnikov [6]. It was subsequently generalized to define a certain class of diagrams in an arbitrary higher topos [4]. The definition can be internalized into HoTT like so:

► **Definition 55** (Postnikov tower). *A tower (A, a) is called a Postnikov tower if each A_n is n -truncated and each a_n is n -connected.*

► **Definition 56** (The Postnikov tower of a type). *If X is a type, the Postnikov tower of X is the tower whose types are $\|X\|_n$ and whose morphisms come from the universal property of truncations.*

► **Definition 57.** *We write $\eta_X : X \rightarrow \lim \|X\|_n$ for the unique morphism from X to the limit of the Postnikov tower of X such that $\lambda x. \eta_X(x)_n = |\bullet|_n$ for all n .*

► **Definition 58** (Postnikov effectiveness). *If (A, a) is a Postnikov tower, then by the universal property, there exist maps $\|\lim(A, a)\|_n \rightarrow A_n$, these are maps ϵ_n such that $\epsilon_n \circ |\bullet|_n = \lambda x. x_n$ the projection map. In fact these maps form a ladder, and we say Postnikov effectiveness holds if they are all equivalences.*

► **Proposition 59** (Anel-Barton [1]). *Anel-Barton dependent choice implies Postnikov effectiveness.*

Proof. Throughout we write $p_n : \lim(A, a) \rightarrow A_n$ for the projection map in an arbitrary tower.

We fix a natural number $d : \mathbb{N}$, and assume that Anel-Barton dependent choice of dimension $\leq d$ holds, we also fix a Postnikov tower (A, a) . We observe that we have a ladder of maps $f_n : \|\lim A_k\|_n \rightarrow A_n$, which is such that $f_n \circ |\bullet|_n^{\lim A_k} = p_n : \lim A_k \rightarrow A_n$.

Observe, it suffices to show that each p_n is n -connected, since then f_n factors an n -connected map through an n -connected map, implying that f_n itself is n -connected, which, since its source and target are both n -truncated, implies that f_n is an equivalence.

So, fix n , and find m such that $m \geq n + d$. Consider the restricted diagram A_{k+m} , and the constant diagram A_m . There is a ladder of maps between these diagrams $q_{n'} : A_{n'+m} \rightarrow A_m$, where $q_{n'} = a_m \circ \dots \circ a_{n'+m-1}$. Each $q_{n'}$ is at least $m \geq n + d$ connected, so the map $\lim q_k : \lim A_{k+m} \rightarrow \lim_k A_m$ is at least n -connected. But $\lim A_{k+m}$ is none other than $\lim A_k$, $\lim_k A_m$ is none other than A_m , and $\lim q_k$ is none other than p_m . Hence, p_m is at least n -connected.

Finally, we observe that $p_n = a_n \circ \dots \circ a_m \circ p_m$, and each of these maps are at least n -connected, so p_n is at least n -connected. Which completes the proof. $\blacktriangleleft$

► **Proposition 60.** *If Postnikov effectiveness holds then, by definition, there is a family of equivalences $\epsilon_n : \|\lim \|X\|_k\|_n \rightarrow \|X\|_n$ for any space X . As an equivalence, ϵ_n has an inverse ϵ_n^{-1} , and this inverse is none other than $\|\eta_X\|_n$.*

Proof. We write p_n for the projection map, as in the previous proof.

We have the following equations:

$$|\bullet|_n^X = p_n \circ \eta_X \quad p_n = \epsilon_n \circ |\bullet|_n^{\lim \|X\|_k}$$

by definition.

The second equation implies that

$$|\bullet|_n^{\lim \|X\|_k} = \epsilon_n^{-1} \circ p_n$$

Whence

$$\begin{aligned} \epsilon_n^{-1} \circ |\bullet|_n^X &= \epsilon_n^{-1} \circ p_n \circ \eta_X \\ &= |\bullet|_n^{\lim \|X\|_k} \circ \eta_X \end{aligned}$$

Which implies

$$\epsilon_n^{-1} = \|\eta_X\|_n$$

by the universal property of the n -truncation. ◀

► **Proposition 61.** *Postnikov effectiveness implies that the operation sending a type to the limit of its Postnikov tower is a modality. Which we call the Postnikov limit modality.*

Proof. We want to check that

$$\lambda f.f \circ \eta_X : ((x : \lim \|X\|_k) \rightarrow \lim \|P(x)\|_n) \rightarrow ((x : X) \rightarrow \lim \|P(\eta_X(x))\|_n)$$

is an equivalence for all $P : \lim \|X\|_k \rightarrow \mathcal{U}$, for all X .

Since limits commute with products (Proposition 44), we see it is enough to check that

$$\lambda f.f \circ \eta_X : ((x : \lim \|X\|_k) \rightarrow \|P(x)\|_n) \rightarrow ((x : X) \rightarrow \|P(\eta_X(x))\|_n)$$

for all n .

Since the type of n -types is an $(n+1)$ -type, the map $\lambda x.\|P(x)\|_n : \lim \|X\|_k \rightarrow \mathcal{U}^{\leq n}$ factors through $|\bullet|_{n+1} : \lim \|X\|_k \rightarrow \|\lim \|X\|_k\|_{n+1}$, say by $\lambda x.Q(x) : \|\lim \|X\|_k\|_{n+1} \rightarrow \mathcal{U}^{\leq n}$, and since n -types are, in particular, $(n+1)$ -truncated, we see that the universal property implies it suffices to check that

$$\lambda f.f \circ \|\eta_X\|_{n+1} : ((x : \|\lim \|X\|_k\|_{n+1}) \rightarrow Q(x)) \rightarrow ((x : \|X\|_{n+1}) \rightarrow Q(\|\eta_X\|_{n+1}(x)))$$

is an equivalence for all $Q : \|\lim \|X\|_k\|_{n+1} \rightarrow \mathcal{U}^{\leq n}$

But this is trivial because $\|\eta_X\|_{n+1}$ is an equivalence by Proposition 60. ◀

► **Proposition 62** (Postnikov limit modality is hypercompletion). *Assuming Postnikov effectiveness, the modal types for the Postnikov limit modality are precisely the hypercomplete types.*

Proof. Assuming Postnikov effectiveness, the map $X \rightarrow \lim \|X\|_k$ is ∞ -connected. This we can see from the equation:

$$|\bullet|_{n+1}^{\lim \|X\|_k} \circ \eta_X = |\bullet|_{n+1}^X \circ p_{n+1}$$

Which shows that η factors an $(n+1)$ -connected map through an $(n+1)$ -connected map, so must itself be n -connected by Proposition 20.

If A_n is a Postnikov tower, then $\lim A_k$ is hypercomplete, this follows from the behavior of limits and lifting problems, see Proposition 45.

So, assuming Postnikov effectiveness, it follows that if X is hypercomplete, then η must be an equivalence – as it is an ∞ -connected map between hypercomplete types. And, if η is an equivalence, then X is hypercomplete by univalence (as it is equivalent to a hypercomplete type). ◀

Thus, we have a hypercompletion modality under the assumption that (e.g.) countable choice holds.

6 SSD dependent choice

In the paper [2], a framework is developed for reasoning about the category of “light condensed anima”. The setting goes by the name of synthetic Stone duality and has applications to condensed mathematics. They introduce a choice principle which we state below. We can prove Postnikov effectiveness from this choice principle, hence derive a characterization of the hypercompletion operation in their setting. The results in this section were arrived at in conversation with Hugo Moeneclaey and have not (yet) been formalized.

► **Definition 63.** *SSD dependent choice*

If (A, a) is a family of types and a_n are all (-1) -connected, then $p_0 : \lim A \rightarrow A_0$ is (-1) -connected.

► **Proposition 64.** *SSD choice implies that if (A, a) is a family of types, and the a_n are all n -connected, then $p_0 : \lim A \rightarrow A_0$ is n -connected.*

Proof. Using Proposition 19 it suffices to show that p_0 is (-1) -connected, and, for each $x, y : \lim A$, $\text{ap}_{p_0} : (x = y) \rightarrow (x_0 = y_0)$ is $(n - 1)$ -connected. The first is true by definition of SSD dependent choice.

Using Proposition 47, we know that $(x = y)$ is a limit of the diagram

$$((x_n = y_n), \lambda p. q_x(n)^{-1} \cdot \text{ap}_{a_n}(p) \cdot q_y(n))$$

and ap_{p_0} is the projection map. Using induction, it suffices to assume that the result of the proposition holds for $(n - 1)$, and then it follows immediately that ap_{p_0} is $(n - 1)$ -connected.

There is nothing else to show. ◀

► **Proposition 65.** *SSD dependent choice implies dependent choice of dimension ≤ 1 .*

Proof. If (A, a) is a tower of $n + 1$ -connected types, then

$$\dots \longrightarrow A_1 \longrightarrow A_0 \longrightarrow \mathbf{1}$$

is a tower whose limit is just the same as $\lim(A, a)$, and whose morphisms are all now n -connected as they factor the $n + 1$ -connected maps into the unit like so: $A_{n+1} \rightarrow A_n \rightarrow \mathbf{1}$. So the map $\lim(A, a) \rightarrow \mathbf{1}$ is n -connected as well by SSD dependent choice. ◀

► **Proposition 66.** *Dependent choice of dimension ≤ 0 implies SSD dependent choice.*

Proof. Consider the ladder between the diagram (A_{n+1}, a_{n+1}) , and the constant diagram A_0 . By hypothesis all these maps are (-1) -connected, so the map of limits, which is precisely the projection is (-1) -connected. ◀

► **Proposition 67.** *SSD dependent choice implies countable choice of dimension ≤ 0 .*

Proof. As before, $\prod A$ is limit of $\mathbf{1} \times A_1 \times \cdots \times A_k$, the projection maps to $\mathbf{1}$ are all of course n -connected if all the A_k are. So the map from the limit is also n -connected by foregoing. ◀

► **Proposition 68.** *SSD choice implies Postnikov effectiveness.*

Proof. Suppose (A, a) is a Postnikov tower.

By results in Section 3, $\lim A$ is also the limit of the towers, (A_{n+k}, a_{n+k}) . These maps, a_{n+k} , are all at least k -connected, whence it follows that $p_k : \lim A \rightarrow A_k$ is k -connected, and the remainder of the proof proceeds like in the other case. ◀

► **Corollary 69.** *In the setting of synthetic Stone duality, the hypercompletion operation exists and is given by $\lambda X. \lim \|X\|$.*

7 Conclusion

We have introduced some choice principles, derived from the work of Anel-Barton [1], and the synthetic Stone duality framework [2] and described their relationship to the form of the hypercompletion operator. Along the way we proved a number of useful propositions about countable inverse limits.

References

- 1 M. Anel and R. Barton. Choice axioms and postnikov completeness, 2024. [arXiv:2403.19772](https://arxiv.org/abs/2403.19772).
- 2 F. Cherubini, T. Coquand, F. Geerligs, and H. Moeneclaey. A foundation for synthetic stone duality. In R. E. Møgelberg and B. van den Berg, editors, *30th International Conference on Types for Proofs and Programs (TYPES 2024)*, 2025.
- 3 L. Leclerc. Cohomology in homotopy type theory. Internship Report, 2022.
- 4 J. Lurie. *Higher Topos Theory*. Princeton University Press, 2009.
- 5 O. Milner. Github Repository. URL: <https://github.com/owen-milner/choicepostnikov>.
- 6 M. M. Postnikov. Investigations in the homotopy theory of continuous mappings. *American Mathematical Society Translations*, 7, 1957.
- 7 E. Rijke. *Introduction to Homotopy Type Theory*. Cambridge University Press, 2026.
- 8 E. Rijke, M. Shulman, and B. Spitters. Modalities in homotopy type theory. *Logical Methods in Computer Science*, 16(1), 2020. doi:10.23638/LMCS-16(1:2)2020.
- 9 The Univalent Foundations Program. Homotopy type theory: Univalent foundations for mathematics, 2013. URL: <https://homotopytypetheory.org/book/>.
- 10 F. van Doorn. *On the formalization of higher inductive types and synthetic homotopy theory*. PhD thesis, Carnegie Mellon University, 2018.

Lean4Lean: Verifying a Typechecker for Lean, in Lean

Mario Carneiro  

Chalmers University of Technology & University of Gothenburg, Sweden

Abstract

In this paper we present a new “external checker” for the Lean theorem prover, written in Lean itself. This is the first¹ complete typechecker for Lean 4 other than the reference implementation in C++ used by Lean itself, and our new checker is competitive with the original, running between 20% and 50% slower and usable to verify all of Lean’s **Mathlib** library, taking an additional step in Lean’s aim to self-host the full elaborator and compiler. Moreover, because the checker is written in a language which admits formal verification, it is possible to state and prove properties about the kernel itself, and we report on progress to formalize the Lean type theory abstractly and prove some theorems about it. Finally, we combine these to get a proof of correctness of parts of the kernel. We plan to use this project to help justify any future changes to the kernel and type theory and ensure unsoundness does not sneak in through either the abstract theory or implementation bugs. The verification is already paying off, as one soundness bug has been spotted and fixed as a result of this work.

2012 ACM Subject Classification Mathematics of computing → Mathematical software performance; Software and its engineering → Formal methods; Social and professional topics → Systems analysis and design

Keywords and phrases Lean, proof assistant, external typechecker, implementation, metatheory, type theory, proof theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.2

Related Version *Full Version*: <https://arxiv.org/abs/2403.14064>

Supplementary Material *Software (Formalization & Implementation)*: <https://github.com/digama0/lean4lean/tree/types2025>

archived at `swh:1:dir:02a8cfda4f3f7a47d8e446a4173279d0374d218e`

Acknowledgements I would like to thank Jeremy Avigad for their support and encouragement, Yannick Forster and the anonymous reviewers for reading early drafts of this work, and Matthieu Sozeau and the rest of the MetaRocq team for collaborating and taking interest in this work. Work supported by the Hoskinson Center for Formal Mathematics at Carnegie Mellon.

1 Introduction

LEAN [12] is a theorem prover based on the Calculus of Inductive Constructions (CIC) [23], quite similar to its older brother ROCQ² [31], but as the name suggests, one of the ways it sought to differentiate itself was the desire to make the kernel “leaner,” relying on simpler primitives while retaining most of the power of the system. This was in particular informed by periodic news of unsoundnesses in Rocq³ due to complications of the many interacting features in the kernel. So from the beginning, Lean advertised its simpler foundations and multiple external checkers [25, 14] as reasons that it would not suffer from the same troubles.

¹ Although more checkers such as **nanoda** have appeared in the meantime.

² Formerly known as Coq.

³ <https://github.com/rocq-prover/rocq/blob/master/dev/doc/critical-bugs.md>



© Mario Carneiro;

licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 2; pp. 2:1–2:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These hopes were broadly upheld: for the entire release history of Lean 3, there were no soundness bugs reported against the kernel (written in C++), and our prior work *Lean’s Type Theory* [7] (henceforth LTT) showed the consistency of Lean with respect to ZFC plus the existence of n inaccessible cardinals for each $n < \omega$, so the soundness story seemed strong.

But times move on, and Lean 4 now exists as an (almost) ground-up rewrite of Lean with most components now written in Lean itself. The kernel was one of the few components that was not rewritten, but it was extended with various features for performance reasons like bignum arithmetic, nested inductive types, and primitive projections, and unfortunately some soundness bugs crept in during the process and Lean’s record is no longer spotless. Another thing that was lost during the port was the external checkers: the whole metaprogramming infrastructure was redesigned so old external checkers are no longer applicable to Lean 4. While most of the new features can be treated as mere abbreviations, nested inductive types and η for structures significantly impact the theory, and as a result the soundness proof in LTT [7] is no longer directly applicable. However, the work by Ullrich [32] covers some of these modifications.

We interpret this as a cautionary tale: It is not sufficient to see a prior system and hope to do better through sheer force of will. We are all human, and mistakes in both the theory and the implementation happen as a matter of course. The way to do better is not to rewrite, but to set up a process that structurally ensures that mistakes are either prevented or corrected. For a proof assistant, there are three main ways to do this: (1) *testing*, (2) *independent reimplementations*, and (3) *formal verification*. Lean has a fairly robust test suite, but testing helps more in the high-frequency low-impact domain; soundness bugs are rare and generally occur in areas that are insufficiently tested because the developer did not even consider the error condition.

The main contribution of this paper lies in (2): an independent reimplementations. To restore the soundness story of Lean to a satisfactory state, we believe it is necessary to bring back the external checkers. And what better language to do so than Lean itself? There are reasons not to use Lean for a truly *independent* verifier (and more external checkers are coming, in Scala⁴ and Rust⁵), but there are also advantages to having a kernel written in Lean:

- The Lean elaborator is already written in Lean as a form of “whitebox automation”: users are able to make use of elaborator APIs to write their own tactics with just as much flexibility as the core language itself. Similarly, having a “whitebox kernel” makes it easier for Lean users to query the internal state and understand how Lean reduces or typechecks terms.
- Unlike Rust and Scala, Lean is designed for both programming and proving, so a kernel written in Lean can be formally verified. And we have good reasons for wanting the kernel to satisfy certain properties.

In this paper, we will present **Lean4Lean**,⁶ an external verifier for Lean 4, written in Lean. The verifier itself is more than a prototype: it is fully capable of checking the entirety of **Lean** (the Lean compiler package), **Batteries** (the extended standard library), and **Mathlib** (the Lean mathematics library).⁷ It is not as fast as the original kernel written in C++ because the Lean compiler still has some ways to go to compete with C++ compilers, but

⁴ <https://github.com/gebner/trepplein>

⁵ https://github.com/ammkrn/nanoda_lib

⁶ <https://github.com/digama0/lean4lean>

⁷ **Lean4Lean** can also check its own source code, although we did not benchmark this configuration specifically.

the overhead is still reasonable in exchange for the additional guarantees it can provide, and it is suitable as a complementary step in Lean projects that wish to validate correctness in a variety of ways.

The second part of this project, which is much larger and on which we report partial progress, is the specification of Lean’s metatheory and verification of the `Lean4Lean` kernel with respect to that theory, covering mistake prevention measure (3). This is similar to the `MetaRocq` project [28, 29], and amounts to a formalization of `LTT` [7]. Our principal contributions here are a definition of the typing judgment, some regularity theorems we can prove and others we can state, and the verification of some representative functions from the typechecker. The remainder is left as future work.

In this paper, we will avoid making value judgments about Lean or Rocq and their respective theories. These systems came to be what they are now as a result of complex historical factors, and we seek only to validate the behavior of Lean as it exists, even though it lacks some type theoretic properties that would have made the job easier if true. One possible outcome of this project is a recommendation for how to improve the typing rules, but this comes at a high cost to existing users so we do not take it lightly and we have no such recommendations currently.

Throughout the paper, code identifiers (both in listings and in inline prose) are hyperlinked to their definitions in the `Lean4Lean` and Lean repositories; we encourage readers to click through for additional context.

The paper is organized as follows: Section 2 defines the main typing judgment and describes some of its properties. Section 3 gives the core data structures of the typechecker and how they relate to the typing rules from Section 2. Section 4 sets the stage for the verification of the typechecker, and how it was used to find a soundness bug in the kernel. Section 5 explains how the global structure of the environment is put together from individual typing judgments. Section 6 explains the complexities associated with supporting inductive types, treated as an extension of the base `MLTT` theory. Section 7 gives some performance results regarding the complete verifier, and Section 8 compares this project to `MetaRocq`, a Rocq formalization of Rocq metatheory and a verified kernel.

2 Base theory

2.1 Expressions

At the heart of the theory is the `VExpr`⁸ type, which represents expressions of type theory. This is a minimalistic version of the type actually used by the kernel in section 3.1, to ease the burden of proving theoretical properties.⁹

```
inductive VExpr where
| bvar (deBruijnIndex : Nat)
| sort (u : VLevel)
| const (declName : Name) (us : List VLevel)
| app (fn arg : VExpr)
| lam (binderType body : VExpr)
| forallE (binderType body : VExpr)
```

⁸ The prefix “V” for “verified” disambiguates it from the `Expr` type used by the kernel itself. This is the more abstracted version of expressions used for specification purposes.

⁹ `MetaRocq` makes a similar but smaller separation: its expression type drops n-ary applications (replaced with binary) and casts (used to control evaluation strategies in Rocq), but otherwise mirrors the type used for metaprogramming. Our `VExpr/Expr` split goes further; see section 3.1 for the full list of differences.

- `bvar n` represents the n th variable in the context, counting from the inside out. As we will see, Lean itself uses “locally nameless” representation, with a combination of de Bruijn variables and named free variables, but here we only have pure de Bruijn variables.
- `sort u` represents a universe, written as `Sort u` (or `Prop` and `Type u` which are syntax for `Sort 0` and `Sort (u+1)` respectively).
- `const c us` represents a reference to global constant c in the environment, instantiated with universes us . (Constants in Lean can be universe-polymorphic, and they are instantiated with concrete universes at each use site.) Inductive constructors, recursors, and definitions are all represented using this constructor.
- `app f a` is function application, written $f a$ or $f(a)$.
- `lam A e` is a lambda-abstraction $\lambda x : A. e$. Because we are using de Bruijn variables, x is not represented; instead e is typechecked in an extended context where variable 0 has type A and the other variables are shifted up by 1.
- `forallE A B`¹⁰ is a dependent Π -type $\Pi x : A. B$, also written as $\forall x : A. B$ because for `Prop` this is the same as the “for all” quantifier. It uses the same binding structure as λ .

The `sort u` and `const c us` constructors depend on another type `VLevel`, which is the grammar of *level expressions*:

```
inductive VLevel where
| zero : VLevel
| succ : VLevel → VLevel
| max : VLevel → VLevel → VLevel
| imax : VLevel → VLevel → VLevel
-- imax a b means (if b = 0 then 0 else max a b)
| param : Nat → VLevel
```

These come up when typechecking expressions, for example the type of `sort u` is `sort (succ u)`. The one notable case here is `param i`, which represents the i th universe variable, where $i < n$ where n is the number of universe parameters to the current declaration.

This definition corresponds to the following BNF-style grammar from LTT [7], with the main change being that it is more precise about how variables and binding are represented. (For presentational reasons, we will use named variables in the informal version, but the formalization uses `lift` and `subst` functions when moving expressions between contexts.)

$$\begin{aligned} \ell &::= 0 \mid S \ell \mid \max(\ell, \ell) \mid \text{imax}(\ell, \ell) \mid u \\ e &::= x \mid U_\ell \mid c_\ell \mid e e \mid \lambda x : e. e \mid \forall x : e. e \\ \Gamma &::= \cdot \mid \Gamma, x : e \end{aligned}$$

2.2 The global environment

The typing relation is defined in a context of previously defined constants, called `VEnv`, with the definition:

```
structure VConstant where (uvars : Nat) (type : VExpr)
structure VDefEq where (uvars : Nat) (lhs rhs type : VExpr)
structure VEnv where
  constants : Name → Option (Option VConstant)
  defeqs : VDefEq → Prop
```

¹⁰The “E” suffix is used because `forall` is a keyword; we could use `forall` anyway but it would require escaping in several places.

This is a minimalistic environment type, essentially exactly what is needed to satisfy the requirements of the typing judgment. (The kernel’s environment type is more complicated and will be discussed in Section 5.)

- **constants** maps a name to a constant, if defined, where a constant is given by its universe arity and its type. It returns **none** if a constant by that name does not exist, and **some none** if the name has been “blocked”, meaning that there is no constant there but we still want to make it illegal to make a definition with that name.¹¹ This corresponds to the $\bar{u}.(c_{\bar{u}} : \alpha)$ declaration appearing in T-CONST below.
- **defeqs** is a set of (anonymous) **VDefEqs**, containing two expressions to be made definitionally equal and their type. This corresponds to the $\bar{u}.(e \equiv e' : \alpha)$ declaration appearing in T-EXTRA below.

2.3 Typing and definitional equality

For the typing rules, we use a slight variation of the typing judgments in LTT [7]; see Figure 1. Some remarks:

- LTT [7] uses two mutually inductive judgments $\Gamma \vdash e : \alpha$ and $\Gamma \vdash e_1 \equiv e_2$ (due to T-DEFEQ and T-PROOF-IRREL), and $\Gamma \vdash e_1 \equiv e_2 : \alpha$ was a defined notion for $\Gamma \vdash e_1 \equiv e_2 \wedge \Gamma \vdash e_1 : \alpha \wedge \Gamma \vdash e_2 : \alpha$. In this version, we have only one all-in-one relation $\Gamma \vdash e_1 \equiv e_2 : \alpha$, and we define $\Gamma \vdash e : \alpha$ to mean $\Gamma \vdash e \equiv e : \alpha$ and define $\Gamma \vdash e_1 \equiv e_2$ as $\exists \alpha. (\Gamma \vdash e_1 \equiv e_2 : \alpha)$.
We conjecture the two formulations to be equivalent, but this version seems to be easier to prove basic structural properties about (see section 2.4). Moreover Lean does not have good support for mutual inductive predicates, so keeping it single-inductive makes induction proofs easier.¹²
- We also define $\Gamma \vdash \alpha$ **type** to mean $\exists \ell. (\Gamma \vdash \alpha : U_\ell)$ (incl. propositions $\Gamma \vdash p : U_0$).
- The rules T-SORT, T-CONST, T-EXTRA make use of a judgment $n \vdash \ell$ **ok**, which simply asserts that every **param** i in ℓ satisfies $i < n$. The $\ell \equiv \ell'$ predicate asserts that ℓ and ℓ' are extensionally equivalent, i.e. for all assignments $v : \mathbb{N} \rightarrow \mathbb{N}$ of natural number values to the universe variables, $\llbracket \ell \rrbracket_v = \llbracket \ell' \rrbracket_v$.
- The T-EXTRA rule says that we can add arbitrary definitional equalities from the environment. This is how we will add the two supported “extensions” to the theory, for inductive types and quotients, without the details of these extensions complicating reasoning about the core theory. But we can’t actually support *arbitrary* definitional equalities – instead each theorem about the typing judgment has its own assumptions about what extensions are allowed. We should be upfront that the specific constraints on T-EXTRA required to handle inductive declarations are not yet pinned down: the typechecker accepts inductives in practice (see Section 6), but our metatheory does not yet close the loop on what extensions a valid inductive specification introduces. This is a known gap, and the largest remaining piece of future work on the theoretical side.

¹¹This is how we model **unsafe** declarations when typechecking safe declarations: they “don’t exist” for most purposes, but it is nevertheless not allowed to shadow them with another safe declaration.

¹²One could also encode the mutual structure as a single indexed inductive, but that doesn’t substantially reduce the complexity – the intricacy here is intrinsic to the typing/def.eq. interaction, not the encoding.

$$\begin{array}{c}
\boxed{\Gamma \ni x : \alpha} \qquad \text{L-ZERO} \quad \frac{}{\Gamma, x : \alpha \ni x : \alpha} \qquad \text{L-SUCC} \quad \frac{\Gamma \ni x : \alpha}{\Gamma, x' : \alpha' \ni x : \alpha} \\
\\
\boxed{\vdash \Gamma \text{ ok}} \qquad \text{C-EMP} \quad \vdash \cdot \text{ ok} \qquad \text{C-CONS} \quad \frac{\vdash \Gamma \text{ ok} \quad \Gamma \vdash \alpha : \mathbb{U}_\ell}{\vdash \Gamma, x : \alpha \text{ ok}} \\
\\
\boxed{\Gamma \vdash_{E,n} e \equiv e' : \alpha} \qquad (\Gamma \vdash e : \alpha) \triangleq (\Gamma \vdash e \equiv e : \alpha) \qquad \text{T-BVAR} \quad \frac{\Gamma \ni x : \alpha}{\Gamma \vdash x : \alpha} \qquad \text{T-SYMM} \quad \frac{\Gamma \vdash e \equiv e' : \alpha}{\Gamma \vdash e' \equiv e : \alpha} \\
\\
\text{T-TRANS} \quad \frac{\Gamma \vdash e_1 \equiv e_2 : \alpha \quad \Gamma \vdash e_2 \equiv e_3 : \alpha}{\Gamma \vdash e_1 \equiv e_3 : \alpha} \qquad \text{T-SORT} \quad \frac{n \vdash \ell, \ell' \text{ ok} \quad \ell \equiv \ell'}{\Gamma \vdash \mathbb{U}_\ell \equiv \mathbb{U}_{\ell'} : \mathbb{U}_{S_\ell}} \\
\\
\text{T-CONST} \quad \frac{\bar{u}.(c_{\bar{u}} : \alpha) \in E \quad \forall i, n \vdash \ell_i, \ell'_i \text{ ok} \wedge \ell_i \equiv \ell'_i}{\Gamma \vdash c_{\bar{\ell}} \equiv c_{\bar{\ell}'} : \alpha[\bar{u} \mapsto \bar{\ell}]} \qquad \text{T-LAM} \quad \frac{\Gamma \vdash \alpha \equiv \alpha' : \mathbb{U}_\ell \quad \Gamma, x : \alpha \vdash e \equiv e' : \beta}{\Gamma \vdash (\lambda x : \alpha. e) \equiv (\lambda x : \alpha'. e') : \forall x : \alpha. \beta} \\
\\
\text{T-ALL} \quad \frac{\Gamma \vdash \alpha \equiv \alpha' : \mathbb{U}_{\ell_1} \quad \Gamma, x : \alpha \vdash \beta \equiv \beta' : \mathbb{U}_{\ell_2}}{\Gamma \vdash (\forall x : \alpha. \beta) \equiv (\forall x : \alpha'. \beta') : \mathbb{U}_{\max(\ell_1, \ell_2)}} \qquad \text{T-APP} \quad \frac{\Gamma \vdash e_1 \equiv e'_1 : \forall x : \alpha. \beta \quad \Gamma \vdash e_2 \equiv e'_2 : \alpha}{\Gamma \vdash e_1 e_2 \equiv e'_1 e'_2 : \beta[x \mapsto e_2]} \\
\\
\text{T-DEFEQ} \quad \frac{\Gamma \vdash \alpha \equiv \beta : \mathbb{U}_\ell \quad \Gamma \vdash e \equiv e' : \alpha}{\Gamma \vdash e \equiv e' : \beta} \qquad \text{T-BETA} \quad \frac{\Gamma, x : \alpha \vdash e : \beta \quad \Gamma \vdash e' : \alpha}{\Gamma \vdash (\lambda x : \alpha. e) e' \equiv e[x \mapsto e'] : \beta[x \mapsto e']} \\
\\
\text{T-ETA} \quad \frac{\Gamma \vdash e : \forall y : \alpha. \beta}{\Gamma \vdash (\lambda x : \alpha. e x) \equiv e : \forall y : \alpha. \beta} \qquad \text{T-PROOF-IRREL} \quad \frac{\Gamma \vdash p : \mathbb{U}_0 \quad \Gamma \vdash h : p \quad \Gamma \vdash h' : p}{\Gamma \vdash h \equiv h' : p} \\
\\
\text{T-EXTRA} \quad \frac{\bar{u}.(e \equiv e' : \alpha) \in E \quad \forall i, n \vdash \ell_i \text{ ok}}{\Gamma \vdash e[\bar{u} \mapsto \bar{\ell}] \equiv e'[\bar{u} \mapsto \bar{\ell}] : \alpha[\bar{u} \mapsto \bar{\ell}]}
\end{array}$$

■ **Figure 1** The rules for the judgment $\Gamma \vdash e \equiv e' : \alpha$, with parameters E (the global environment) and n (the number of universe parameters in context), suppressed in the notation. This is `VEnv.IsDefEq` in the formalization.

2.4 Properties of the typing judgment

► **Lemma 1** (basic properties).

1. (*IsDefEq.hasType*) If $\Gamma \vdash e_1 \equiv e_2 : \alpha$, then $\Gamma \vdash e_1 : \alpha$ and $\Gamma \vdash e_2 : \alpha$.
2. (*IsDefEq.closedN'*) If $\Gamma \vdash e \equiv e' : \alpha$ and Γ is well-scoped, then e , e' , and α are well-scoped¹³ (all free variables have indices less than $|\Gamma|$).
3. (*IsDefEq.weakenN*, weakening) If $\Gamma, \Gamma' \vdash e \equiv e' : \alpha$, then $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha$.

¹³The formalization calls this property **ClosedN**, generalizing the standard sense of “closed” (no free variables) to a context of size N ; the strict sense is recovered when $N = 0$.

4. (*IsDefEq.instL*) If $\Gamma \vdash_{E,n} e \equiv e' : \alpha$ and $\forall i. n' \vdash \ell_i \text{ ok}$, then $\Gamma[\bar{u} \mapsto \bar{\ell}] \vdash_{E,n'} e[\bar{u} \mapsto \bar{\ell}] \equiv e'[\bar{u} \mapsto \bar{\ell}] : \alpha[\bar{u} \mapsto \bar{\ell}]$.
5. (*IsDefEq.instN*) If $\Gamma, x : \beta \vdash e_1 \equiv e_2 : \alpha$ and $\Gamma \vdash e_0 : \beta$, then $\Gamma \vdash e_1[x \mapsto e_0] \equiv e_2[x \mapsto e_0] : \alpha[x \mapsto e_0]$.

Proof sketch.

1. Immediate from T-SYMM and T-TRANS, recalling that $\Gamma \vdash e : \alpha$ means $\Gamma \vdash e \equiv e : \alpha$.
2. This is a straightforward proof by induction on $\Gamma \vdash e \equiv e' : \alpha$, except that one gets stuck at T-CONST because (assuming the environment is well-typed, see Section 5) we know that $\vdash c_{\bar{\ell}} : \alpha$ and need that α is well-scoped. So in fact this is a double induction, first over the environment E and then over $\Gamma \vdash e \equiv e' : \alpha$. We also need lemmas about $e[x \mapsto e']$ and $e[\bar{u} \mapsto \bar{\ell}]$ preserving well-scopedness, but these are also direct by induction on e .
3. By induction. We use Lemma 1.2 in the T-CONST case, to show that the α in $\Gamma, \Gamma' \vdash c_{\bar{\ell}} \equiv c_{\bar{\ell}'} : \alpha$ can be used in $\Gamma, \Delta, \Gamma' \vdash c_{\bar{\ell}} \equiv c_{\bar{\ell}'} : \alpha$ without renaming any bound variables because α is well-scoped.
4. By induction.
5. By induction on the first hypothesis. This uses Lemma 1.3 to lift $\Gamma \vdash e_0 : \beta$ under binders. $\blacktriangleleft$

► **Lemma 2** (*IsDefEq.defeqDF_1*). If $\Gamma, x : \alpha \vdash e_1 \equiv e_2 : \beta$ and $\Gamma \vdash \alpha \equiv \alpha' : \mathcal{U}_{\ell}$, then $\Gamma, x : \alpha' \vdash e_1 \equiv e_2 : \beta$.

Proof. We have $\Gamma, x : \alpha' \vdash x : \alpha$ by T-BVAR, T-DEFEQ and weakening, and $\Gamma, x' : \alpha', x : \alpha \vdash e_1 \equiv e_2 : \beta$ by weakening, so $\Gamma, x : \alpha' \vdash e_1[x \mapsto x] \equiv e_2[x \mapsto x] : \beta[x \mapsto x]$ by Lemma 1.5. $\blacktriangleleft$

► **Lemma 3** (*HasType.forallE_inv*). If $\Gamma \vdash (\forall x : \alpha. \beta) : \gamma$ then $\Gamma \vdash \alpha \text{ type}$ and $\Gamma, x : \alpha \vdash \beta \text{ type}$ (and therefore also $\Gamma \vdash (\forall x : \alpha. \beta) \text{ type}$).

Proof sketch. By induction on $\Gamma \vdash e_1 \equiv e_2 : \gamma$ (recall $\Gamma \vdash e : \gamma$ means $\Gamma \vdash e \equiv e : \gamma$), assuming that one of e_1 or e_2 is $\forall x : \alpha. \beta$. We make repeated use of Lemma 1.1 to extract typing premises from def.eq. hypotheses. Most cases are trivial or inapplicable. Of those that remain:

- T-ALL: If e_1 is $\forall x : \alpha. \beta$ we are done; if e_2 is $\forall x : \alpha. \beta$ then we obtain $\Gamma, x : \alpha' \vdash \beta \text{ type}$ and need Lemma 2 to get $\Gamma, x : \alpha \vdash \beta \text{ type}$.
- T-BETA: For this to apply, it must be that we have $(\lambda y : \delta. e) e' \equiv e[y \mapsto e']$ where $e[y \mapsto e']$ is $\forall x : \alpha. \beta$. So either $e = y$ and $e' = \forall x : \alpha. \beta$, in which case the inductive hypothesis for e' applies, or $e = \forall x : \alpha'. \beta'$ with $\alpha'[y \mapsto e'] = \alpha$ and $\beta'[y \mapsto e'] = \beta$ in which case $\Gamma, y : \delta \vdash \alpha' \text{ type}$ and $\Gamma, y : \delta, x : \alpha' \vdash \beta' \text{ type}$ by the inductive hypothesis for e , and Lemma 1.5 applies.
- T-EXTRA: It could be that $e[\bar{u} \mapsto \bar{\ell}]$ is $\forall x : \alpha. \beta$, but this can only happen if $e = \forall x : \alpha'. \beta'$ with $\alpha'[\bar{u} \mapsto \bar{\ell}] = \alpha$ and $\beta'[\bar{u} \mapsto \bar{\ell}] = \beta$, so by induction hypothesis (for the environment) $\vdash \alpha' \text{ type}$ and $x : \alpha' \vdash \beta' \text{ type}$, and we conclude using level substitution and weakening. $\blacktriangleleft$

► **Lemma 4** (*HasType.sort_inv*). If $\Gamma \vdash_{E,n} \mathcal{U}_{\ell} : \gamma$ then $n \vdash \ell \text{ ok}$ (and therefore also $\Gamma \vdash \mathcal{U}_{\ell} : \mathcal{U}_{S_{\ell}}$).

Proof sketch. Similar to Lemma 3. $\blacktriangleleft$

► **Theorem 5** (*IsDefEq.isType*). If $\vdash \Gamma \text{ ok}$ and $\Gamma \vdash e : \alpha$ then $\Gamma \vdash \alpha \text{ type}$.

Proof sketch. By induction, using weakening and substitution lemmas. The nontrivial cases are:

- T-APP: We have $\Gamma \vdash (\forall x : \alpha. \beta)$ type from the IH and $\Gamma \vdash e_2 : \alpha$ by assumption, and from Lemma 3 we get $\Gamma, x : \alpha \vdash \beta$ type, so $\Gamma \vdash \beta[x \mapsto e_2]$ type by the substitution lemma.
- T-ALL: We have $\Gamma \vdash U_{\ell_1}$ type and $\Gamma, x : \alpha \vdash U_{\ell_2}$ type from the IH, so by Lemma 4 we have $n \vdash \ell_1, \ell_2$ ok, therefore $\Gamma \vdash U_{\max(\ell_1, \ell_2)}$ type. ◀

► **Lemma 6** (*IsDefEq.instDF*). *If $\Gamma, x : \alpha \vdash f \equiv f' : \beta$ and $\Gamma \vdash a \equiv a' : \alpha$ then $\Gamma \vdash f[x \mapsto a] \equiv f'[x \mapsto a'] : \beta[x \mapsto a]$.*

Proof sketch. This is a generalization of Lemma 1.5, but we need Theorem 5 to prove that α and β are well-typed to apply the rules below.

First we show the claim assuming $\Gamma \vdash \beta[x \mapsto a] \equiv \beta[x \mapsto a'] : U_\ell$. In this case we have

$$\begin{aligned} f[x \mapsto a] &\equiv (\lambda x : \alpha. f) a \\ &\equiv (\lambda x : \alpha. f') a' \\ &\equiv f'[x \mapsto a'] : \beta[x \mapsto a], \end{aligned}$$

where we use T-BETA twice, and use the assumption $\beta[x \mapsto a] \equiv \beta[x \mapsto a']$ to justify the last step because T-BETA gives the equality at the type $\beta[x \mapsto a']$ instead.

To finish, we apply the lemma twice, once with f and β so that it suffices to show $\beta[x \mapsto a] \equiv \beta[x \mapsto a'] : U_\ell$ and then again with β in place of f and U_ℓ in place of β so that it suffices to show $U_\ell[x \mapsto a] \equiv U_\ell[x \mapsto a'] : U_{S\ell}$, which is true by reflexivity. ◀

2.5 Conjectured properties of the typing judgment

Unfortunately, there are more structural properties of the typing judgment we need beyond the results in the previous section. Many of these are close relatives of each other and can be proved from other conjectures in this section.

► **Conjecture 7** (*IsDefEq.uniq*). *If $\Gamma \vdash e : \alpha$ and $\Gamma \vdash e : \beta$, then $\Gamma \vdash \alpha \equiv \beta : U_\ell$ for some ℓ .*

Unique typing has a major simplifying effect on the theory. Here are some consequences (with proofs omitted, but they are simple algebraic consequences of earlier lemmas):

► **Corollary 8** (Consequences of unique typing).

1. (*IsDefEqU.trans*) If $\Gamma \vdash e_1 \equiv e_2$ and $\Gamma \vdash e_2 \equiv e_3$, then $\Gamma \vdash e_1 \equiv e_3$. (This is T-TRANS but without requiring the types to match.)
2. (*isDefEqU.iff*) $\Gamma \vdash e \equiv e' : \alpha$ if and only if $\Gamma \vdash e : \alpha$, $\Gamma \vdash e' : \alpha$, and $\Gamma \vdash e \equiv e'$. (Hence this formulation implies the one in LTT [7].)
3. (*IsDefEqU.defeqDF*) If $\Gamma \vdash e \equiv e' : \alpha$ and $\Gamma \vdash \alpha \equiv \beta$, then $\Gamma \vdash e \equiv e' : \beta$. (This is T-DEFEQ but with an untyped definitional equality.)

This group of theorems is proved mutually with Conjecture 7 in LTT [7]:

► **Conjecture 9** (Definitional inversion).

1. (*IsDefEqU.sort_inv*) If $\Gamma \vdash U_\ell \equiv U_{\ell'}$ then $\ell \equiv \ell'$.
2. (*IsDefEqU.forallE_inv*) If $\Gamma \vdash (\forall x : \alpha. \beta) \equiv (\forall x : \alpha'. \beta')$ then $\Gamma \vdash \alpha \equiv \alpha'$ and $\Gamma, x : \alpha \vdash \beta \equiv \beta'$.¹⁴
3. (*IsDefEqU.sort_forallE_inv*) $\Gamma \vdash U_\ell \not\equiv (\forall x : \alpha. \beta)$.

¹⁴This theorem is sometimes called “injectivity of $\forall$ ” but this name is slightly misleading because $\forall$ is not injective as a function on types, which is to say if we use propositional equality instead of definitional equality then the theorem fails.

The reason Conjecture 7 and Conjecture 9 have been downgraded from theorems in LTT [7] to conjectures here is because the proof has an error in one of the technical lemmas, and it remains to be seen if it is possible to salvage the proof. More precisely, the proof constructs a stratification $\vdash_i$ of the typing judgment in order to break the mutual induction between typing and definitional equality, but this stratification does not and cannot respect substitution; that is, if $\Gamma, x : \beta \vdash_i e : \alpha$ and $\Gamma \vdash_j e' : \beta$, then the proof requires $\Gamma \vdash_{\max(i,j)} e[x \mapsto e'] : \alpha$ but only $\Gamma \vdash_{i+j} e[x \mapsto e'] : \alpha$ holds.

We still believe the conjectures are true, but the induction needs more work. Lennon-Bertrand [21] explores proof approaches that could apply here, based on similar work for MetaRocq. Another route, taken by Siles and Herbelin [26] for general PTSs, uses a heavily annotated syntax in which terms carry their types explicitly, making unique typing essentially syntactic; the difficulty then relocates to elaborating the kernel’s unannotated syntax into the annotated form, which is the typing problem in disguise. Soundness is unaffected, since alternative routes to the model avoid unique typing (also in LTT [7]), but these conjectures are needed in some form to prove correctness of the typechecker (see Section 3).

Another class of conjectured theorems concerns the “invertibility” of weakening:¹⁵

► **Conjecture 10** (Strengthening). *If e, e' do not mention any variables in Δ , then $\Gamma, \Delta, \Gamma' \vdash e \equiv e'$ if and only if $\Gamma, \Gamma' \vdash e \equiv e'$.*

► **Corollary 11** (Consequences of strengthening).

1. (*IsDefEq.skips*) *If $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha$ and e, e' do not mention the variables in Δ , then there exists α' not mentioning Δ such that $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha'$.*
2. (*IsDefEq.weakN_iff*) *If e, e', α do not mention any variables in Δ , then $\Gamma, \Delta, \Gamma' \vdash e \equiv e' : \alpha$ if and only if $\Gamma, \Gamma' \vdash e \equiv e' : \alpha$.*
3. (*OnCtx.weakN_inv*) *If $\vdash \Gamma, \Delta, \Gamma' \text{ ok}$ and Γ' does not mention the variables in Δ then $\vdash \Gamma, \Gamma' \text{ ok}$.*

3 The typechecker

3.1 Relating VExpr to Expr

The actual Lean kernel does not use any of the machinery from the previous section directly. Instead, it works with another type, `Expr`:

```
inductive Expr where
| bvar (deBruijnIndex : Nat)
| fvar (fvarId : FVarId)
| mvar (mvarId : MVarId)
| sort (u : Level)
| const (declName : Name) (us : List Level)
| app (fn arg : Expr)
| lam (binderName : Name) (type body : Expr) (info : BinderInfo)
| forallE (binderName : Name) (type body : Expr) (info : BinderInfo)
| letE (declName : Name) (type value body : Expr) (nonDep : Bool)
| lit : Literal → Expr
| mdata (data : MData) (expr : Expr)
| proj (typeName : Name) (idx : Nat) (struct : Expr)
```

¹⁵This is not a trivial theorem, and MetaRocq found a bug in Rocq here; see section 3.5 of [27].

Comparing this with `VExpr` reveals some differences:

- `mvar n` is for metavariables in the elaborator. They are banned in kernel terms.
- `fvar n` is a “free variable”, which is identified by a name rather than an index into the context. Although these are also banned, the kernel creates `fvar n` expressions during typechecking, and converting between locally nameless and pure de Bruijn is one of the major differences we will have to deal with in the verification of the kernel.
- The `Level` type (not shown) also differs from `VLevel` similarly.
- The `lam` and `forallE` constructors contain additional elaborator metadata.
- The `letE` construct is used for `let x := e1; e2` expressions. We handle these by simply expanding them to `e2[x ↦ e1]`.
- The `lit` constructor is for literals, `Literal := Nat ⊕ String`. This is one of the new Lean 4 kernel features: naturals are bignums, and functions like `Nat.mul` are overridden with native implementations rather than using the inductive structure of `Nat` (effectively, unary numerals). For modeling in `VExpr` we just perform this (exponential-size) unfolding into `Nat.zero` and `Nat.succ`; similarly `String` is modeled as `List Char`, though at runtime it is a UTF-8 byte string.
- The `mdata d e` constructor is equivalent to `e` for modeling purposes (this is used for associating metadata to expressions).
- `proj c i e` is a projection out of a `structure` or structure-like inductive type (see Section 6). This is the hardest constructor to desugar: although it equals an application of the recursor, we need the type of `e` to build the term, so the desugaring is type-aware and only well-defined up to definitional equality.

Note that `Expr` was not defined as part of this project; it is imported directly from the Lean elaborator. In fact the C++ kernel reaches it via FFI, so `Lean4Lean` and the C++ kernel share this data structure.

As a result of these considerations, we use an inductive type to encapsulate the translation from `Expr` to `VExpr`:

```
inductive TrExpr (env : VEnv) (Us : List Name) : VLCtx → Expr → VExpr → Prop
```

Here $E; \bar{u}; \Delta \vdash e \triangleright e'$ is shorthand for `TrExpr E  $\bar{u}$   $\Delta$  e e'`, asserting that in local context Δ , with names $\bar{u}$ for the local universe parameters and E for the global environment (see Section 5), $e' : \text{VExpr}$ is a translation of $e : \text{Expr}$. It also asserts that Δ is well-typed and e' is well-typed in it, so $\exists e'. E; \bar{u}; \Delta \vdash e \triangleright e'$ asserts that e is a well-typed `Expr` – our target specification for correctness of the typechecker.

The type `VLCtx` appearing here is new, and it contains the information we need to translate between the `Expr` and `VExpr` types:

```
inductive VLocalDecl where
| vlam (type : VExpr)      -- x : type
| vlet (type value : VExpr) -- x : type := value

def VLCtx := List (Option FVarId × VLocalDecl)
```

The idea is that `vlam  $\alpha$` is a regular variable, the only kind we had in Section 2, while `vlet  $\alpha$  v` is used inside a `let x :  $\alpha$  := v; _` term. We can build a local context in the sense of Section 2 by simply dropping the `vlet  $\alpha$  v` terms:

```
def toCtx : VLCtx → List VExpr
| [] => []
| (_, vlam  $\alpha$ ) ::  $\Delta$  =>  $\alpha$  :: toCtx  $\Delta$ 
| (_, vlet _ _) ::  $\Delta$  => toCtx  $\Delta$ 
```

We have to be careful to reindex the `Expr.bvar i` indices though, since this includes both let- and lambda-variables while `VExpr.bvar i` only counts the lambda-variables, with the let-variables expanded to terms.

The `Option FVarId` in the context represents the “name” of the variables. Lean follows the “locally nameless” discipline, in which most operations act only on “closed terms” (where closed means that there are no `bvar i` variables outside a binder), so any time it needs to process a term under a binder it first *instantiates* the term, replacing `bvar (i + 1)` with `bvar i` and `bvar 0` with `fvar a`, where $a : \text{FVarId}$ is a (globally) fresh variable name. We store these names in Δ to relate them to de Bruijn indices. The name generator uses a global counter, with the invariant that every `FVarId` in the state is below it.

It is an `Option` because when typing an *open* term we must handle variables not yet assigned `FVarIds`. Lean does not process such terms directly, but they appear as subterms of ones it does, so we need both cases for a compositional specification.

3.2 The typechecker implementation

The kernel itself is defined completely separately from `VExpr`, and is a close mirror of the C++ code, modulo translation into functional style. Since the two implementations do the same things in the same order, most bugs or peculiarities we discovered while formalizing `Lean4Lean` are replicated in the C++ kernel, giving us hope that verifying `Lean4Lean` will also lend strong evidence for the correctness of the C++ code.

3.2.1 Untangling recursion via RecM

The kernel consists of a large mutually-recursive set of functions. Because this is difficult for Lean to handle, and it also causes issues for proving theorems about the functions, it is desirable to break the dependencies. It turns out that just 4 kernel functions are enough to cut the mutual recursion: if each is split into two nodes, one receiving all incoming edges and one sending all outgoing edges, the call graph becomes acyclic (except for self-loops corresponding to normal recursion), so the remaining functions can be defined in dependency order. Conveniently, all four are also important high-level entry points:

1. `isDefEqCore s t` returns `true` if $\Gamma \vdash s \equiv t$ is provable. (This is normally used via `isDefEq s t`, which also caches this result, but some callers use `isDefEqCore` directly.)
2. `whnf e` returns the weak head normal form (WHNF) of e . From a modeling perspective, the main important property is that if it returns e' and e is well-typed then $\Gamma \vdash e \equiv e'$ is provable.
3. `whnfCore e cheapRec cheapProj` has the same specification as `whnf e`, but it lacks some early exit paths and does not unfold definitions. `cheapRec` and `cheapProj` are flags affecting which kinds of terms are unfolded. The reason `whnf` and `whnfCore` show up separately in this list is because they are both used in other functions (i.e. some functions call `whnfCore` directly because e.g. the fast path isn’t applicable).
4. `inferType e inferOnly` infers or typechecks a term. That is, if `inferType e false` returns α then $\Gamma \vdash e : \alpha$ holds, and if `inferType e true` returns α then $\Gamma \vdash e : \alpha$ holds assuming $\Gamma \vdash e : \alpha'$ for some α' . (The latter mode is also known as *retyping* in Rocq.)

We use this to define the `RecM` monad:

```
structure Methods where
  isDefEqCore : Expr → Expr → M Bool
  whnfCore (e : Expr) (cheapRec := false) (cheapProj := false) : M Expr
  whnf (e : Expr) : M Expr
  inferType (e : Expr) (inferOnly : Bool) : M Expr

abbrev RecM := ReaderT Methods M
```

Here `ReaderT` is the reader monad transformer, which augments the underlying `TypeChecker.M` monad with an immutable, implicitly-threaded `Methods` argument that any computation can read. `TypeChecker.M` itself contains the actual state of the typechecker.

But this method only kicks the problem up one step: how do we construct a `Methods` if every function needs another `Methods`? Ideally we would prove the kernel terminates, since dependent type theory (DTT) is meant to be terminating. However:

- It is unlikely that we can prove termination of a typechecker for Lean in Lean, because although the soundness proof in LTT [7] does not depend on termination, MetaRocq’s does [29], and generally termination measures for DTT require large cardinals of comparable strength to the proof theory. We are up against Gödel’s incompleteness theorem, so anything that would imply the unconditional soundness of Lean won’t be directly provable.
- Besides this, the Lean type theory is known not to terminate. Coquand and Abel [1] constructed a counterexample to strong normalization using reduction of proofs, and this can be shown to impact definitional equality checks even for regular types:

```
/-- Coquand-Abel construction of nontermination in proofs -/
def True' := ∀ p : Prop, p → p
def om : True' := fun A a =>
  @cast (True' → True') A -- cast : α = β → α → β (reduces on α = α)
  (propext ⟨fun _ => a, fun _ => id⟩) -- propext : (p ↔ q) → p = q
  (fun z => z (True' → True') id z)
def Om : True' := om (True' → True') id om
#reduce Om -- whnf nontermination

/-- nontermination outside proofs: -/
inductive Foo : Prop | mk : True' → Foo
def foo : Foo := Om _ (Foo.mk fun _ => id)
example : foo.recOn (fun _ => 1) = 1 := by rfl -- isDefEq nontermination
```

Essentially, this is a combination of impredicativity, proof irrelevance, and subsingleton elimination.

- The kernel does not loop forever in many cases because this is a bad user experience – it has timeouts and depth limits. Not all parts of the kernel have such limits, but it does give us a reasonable design principle which fortuitously solves our termination problem.

So we use what is arguably¹⁶ the standard solution for defining partial functions in a language like Lean or Rocq: use a fuel parameter, a natural number which counts the number of nested recursive calls to one of the `Methods`, and throw a `deepRecursion` error if we run

¹⁶ Alternatives like the Bove-Capretta method [6] (take a termination proof as an argument) don’t fit here: the kernel API is fixed to match Lean’s, and we want it to time out on extreme cases anyway.

out of fuel. Currently, this limit is a fixed constant (1000), which turns out to be sufficient for checking all of Mathlib, but this could be made configurable.¹⁷

3.2.2 Inferring types

The kernel’s typechecking algorithm is fairly straightforward. It is not bidirectional; instead it propagates the type of expressions forward via the `inferType` function, whose `inferOnly` flag is true when we may assume the expression is well-typed. For example, here is how $\lambda x : \alpha. e$ expressions are typed:

```
def inferLambda (e : Expr) (inferOnly : Bool) :
  RecM Expr := loop #[] e where
  loop fvars : Expr → RecM Expr
  | .lam name dom body bi => do
    let d := dom.instantiateRev fvars
    if !inferOnly then
      let sort ← inferType d inferOnly
      _ ← ensureSortCore sort d
    withLocalDecl name d bi fun fv => do
      loop (fvars.push fv) body
  | e => do
    let e' := e.instantiateRev fvars
    let r ← inferType e' inferOnly
    let r := r.cheapBetaReduce
    return (← getLCtx).mkForall fvars r
```

Observe that this does more than simply checking T-LAM:

- Since it is using locally-nameless representation, whenever it needs to enter a binder it has to create a fresh `fvar` and replace `bvar 0`. The `withLocalDecl` function does this, and changes the local context to include the new `fvar`, but instantiation is deferred here.
- It actually consumes a *sequence* of lambdas, using the tail-recursive function `loop`. This is a performance optimization, since instantiation is $O(n)$ and so repeatedly instantiating the body in each step of the loop would cause $O(n^2)$ work. The actual instantiation is the two calls to `instantiateRev`.
- When `inferOnly` is true, meaning that it can assume the term is well-formed, it does not need to typecheck the type of the lambda.
- At the end of the loop, it uses `cheapBetaReduce` to reduce the result type in the common case that it has the form $(\lambda \bar{x}. x_i) \bar{e}$, or $(\lambda \bar{x}. c) \bar{e}$ (where c does not depend on $\bar{x}$).

We will discuss this function further in Section 4.

3.2.3 WHNF and reduction

The `whnf` function performs reductions at the “head” of an expression until a constructor is exposed. In so doing it implements a lazy, call-by-name evaluation strategy.

- A variable, sort, axiom, lambda, forall, or literal is in WHNF.
- On an application $f a$, reduce f to WHNF first. If it reduces to $\lambda x. e$, then β -reduce to $e[a/x]$ and continue.

¹⁷This is not a limit on the depth of expressions exactly, these use structural recursion and hence need no fuel for the termination argument; instead fuel is only consumed when making a recursive call that is not otherwise decreasing, for example when reducing definitions to WHNF. The code that is most likely to hit depth limits is in proofs by reflection, but these are comparatively rare, in part because the kernel algorithm for this is not very efficient.

- On a let-expression `let x := a; e`, reduce to $e[a/x]$ and continue.
- On the recursor of an inductive type, reduce its major argument first. If it reduces to a constructor, then β/ι -reduce (e.g. $\text{rec}_{\mathbb{N}} C z s (\text{suc } n) \rightsquigarrow s n (\text{rec}_{\mathbb{N}} C z s n)$) and continue.
- On a definition, reduce to its body and continue.

Beyond these basics, there are a few additional Lean specifics:

- There is a list of 15 built-in functions on $\mathbb{N}$ like `+`, `-`, `×`, `div`, `mod`, `bitAnd`, etc., which are evaluated in call-by-value order. For example:
 - If we are reducing $a + b$ to WHNF, then first reduce a and b . If these reduce to literals $[n]$, $[m]$, then compute $[n + m]$ and stop. (Compare: the default behavior on $a + 1$ would reduce to $\text{suc } (a + 0)$ without reducing a .)
- When encountering `reduceBool c` where $c : \text{Bool}$ is a constant definition, c is compiled to interpreter bytecode and executed. The result (`true` or `false`) is reified as an expression of type `Bool` as the result.

Supporting built-in functions on $\mathbb{N}$ in Lean4Lean was not difficult, but doing so soundly required implementing checks to ensure that e.g. when a definition named `Nat.add` is added to the environment, it satisfies the definitional equalities $a + 0 \equiv a$ and $a + \text{suc } n \equiv \text{suc } (a + n)$. This allows us to prove by induction that this definition will compute on literals such that $[n] + [m] \equiv [n + m]$.¹⁸

On the other hand, supporting native evaluation using `reduceBool` is not nearly so easy. In theory, with a formalization of the IR (intermediate representation) semantics, it may be possible to prove the soundness of IR compilation and interpretation. However, IR can also call external functions from C, and with `implemented_by` one can completely decouple the compiler behavior from the kernel’s version of the implementation, so the general feature is simply “unsound by design.” Given that Mathlib and many other mathematics projects avoid `reduceBool` completely for these reasons, we do not lose much by not supporting it.

3.2.4 Definitional equality checking

This is the least “canonical” part of a DTT typechecker. The worst-case complexity is galactically large, so the heuristics are critical for keeping typechecking in the same ballpark as the original check by the human author (we only aim to check Lean projects in the wild, not adversarial inputs). The algorithm for checking $s \stackrel{?}{=} t$ in Lean goes roughly as follows:

- If s and t are pointer-equal, or already known equal via a union-find cache of `def.eq` subterms, return `true`.
- If s and t are both lambdas, sorts, forall, or literals, then compare subterms.
- Validate `true` $\equiv t$ if $t \rightsquigarrow \text{true}$.¹⁹
- Reduce $s \rightsquigarrow s'$ and $t \rightsquigarrow t'$ with `cheapProj := true`.
- If $s' : \alpha$ and $t' : \beta$ and $\alpha \equiv \beta : \mathcal{U}_0$ then return `true` (T-PROOF-IRREL).
- If $s' = f \bar{a}$ and $t' = g \bar{b}$ where f, g are definitions: if $f = g$ and $a_i \equiv b_i$ for all i , return `true`; otherwise unfold the side with lower height²⁰ and continue.
- If $s' = t'$ are variables, or $s' = c_{\bar{\ell}}$ and $t' = c_{\bar{\ell}'}$ with $\bar{\ell} \equiv \bar{\ell}'$, or $s' = u.i$ and $t' = v.i$ with $u \equiv v$, return `true`.

¹⁸The original Lean kernel does not perform these checks. This is technically unsound, but to exploit it one would need to replace the prelude, and this is not a supported configuration.

¹⁹This is an optimization for proof by reflection, where a goal $t = \text{true}$ is proved using `refl true : true = true`.

²⁰ $\text{height}(f) = 1 + \max_{c \preceq \delta(f)} \text{height}(c)$, where $\delta(f)$ is the body of definition f .

- Reduce $s' \rightsquigarrow s''$ and $t' \rightsquigarrow t''$ again, this time with `cheapProj := false` (which can fire projection reductions skipped in the first pass).
- If $s'' = f \bar{a}$ and $t'' = g \bar{b}$ and $f \equiv g$ and $a_i \equiv b_i$ for all i , return true.
- Try the η -laws (in each case possibly swapping s and t): η -expansion ($t'' = \lambda \bar{x}. u$ with $s'' \bar{x} \equiv u$), structure η ($t'' = (\bar{u}_i)$ with $s''.i \equiv u_i$ for all i), and unit η ($s'' : \text{Unit}$ or another unit-like structure); also unfold string literals.
- Otherwise, fail. (Because of various incompletenesses in this algorithm, this doesn't actually imply $s \neq t$.)

4 Verifying the algorithm

We would now like to connect the results of Section 2 and Section 3 by showing that the monadic typechecker functions satisfy their respective specifications. The typechecker monad M is a standard reader/state/exception monad stack:

```
abbrev M := ReaderT Context <| StateT State <| Except Exception
```

Because we give the semantics of `Expr` using `VExpr`, and this translation is only unique up to def.eq. (definitional equality), we will maintain a shadow state `VContext` and `VState` which have `Expr × VExpr` pairs everywhere so that we can project out both the original `Context` as well as the corresponding `VExpr` for each well-formed expression.

We use a lightweight monadic program logic based on Hoare logic, but “one-sided”: for the precondition, we pass the exact context and state (and additional hypotheses become regular preconditions on the theorem), and for the postcondition we have a predicate over possible next state and return values.

```
def M.WF (c : VContext) (vs : VState) (x : M α) (Q : α → VState → Prop) :=
  vs.WF c → ∀ a s', x c.toContext vs.toState = .ok (a, s') →
    ∃ vs', vs'.toState = s' ∧ vs ≤ vs' ∧ vs'.WF c ∧ Q a vs'
```

Here $c : VContext$ and $vs : VState$ are the shadow context and state, and we write $\hat{c} := c.toContext$ and $\hat{vs} := vs.toState$ for their projections to the underlying `Context` and `State` actually threaded through the M monad. In words, the definition says: assuming (c, vs) are well-formed, if running the monadic computation $x : M \alpha$ on $(\hat{c}, \hat{vs})$ evaluates to some $a : \alpha$ in state s' , then there exists vs' with $\hat{vs}' = s'$ such that $vs \leq vs'$ (a partial order on states, which intuitively says that vs' extends vs forward in time), (c, vs') is well-formed, and $Q(a, vs')$ holds. The $\leq$ relation is used mainly to handle the global counter for creating fresh `fvars`: the well-formedness invariant ensures the current counter value is greater than any `fvar` in the state or context, and $vs \leq vs'$ enforces that the counter only ever grows, keeping the state and context well-formed.

We can use this to define the specification of the `Methods`:

```
structure Methods.WF (m : Methods) where
  isDefEqCore : c.TrExpr e1 e1' → c.TrExpr e2 e2' →
    (m.isDefEqCore e1 e2).WF c s fun b _ => b → c.IsDefEqU e1' e2'
  whnfCore : c.TrExpr e e' →
    (m.whnfCore e cr cp).WF c s fun e1 _ => c.FVarsBelow e e1 ∧ c.TrExpr e1 e'
  whnf : c.TrExpr e e' →
    (m.whnf e).WF c s fun e1 _ => c.FVarsBelow e e1 ∧ c.TrExpr e1 e'
  inferType : e.FVarsIn (· ∈ c.vlctx.fvars) →
    (inferOnly = true → ∃ e', c.TrExpr e e') →
    (m.inferType e inferOnly).WF c s fun ty _ => ∃ e' ty', c.TrTyping e ty e' ty'
```

- `isDefEqCore` (and `isDefEq`) ensures that if the inputs are well-typed expressions $e_1 \triangleright e'_1$ and $e_2 \triangleright e'_2$, and it returns `true`, then $e'_1 \equiv e'_2$.
- `whnf` and `whnfCore` ensure that given a well-formed expression e , if they return e_1 then $e \equiv e_1$. We can succinctly say this by assuming $e \triangleright e'$ and asserting $e_1 \triangleright e'$ as well, since $\triangleright$ is closed under `def.eq`. (There is also a technical condition $\text{FV}(e) \supseteq \text{FV}(e_1)$.)
- `inferType` does not assume that it has a well-formed input, but it does need to know that the global counter is fresh for any fvars in the input. If `inferOnly` is true, it also assumes e is well-formed. It asserts that $e \triangleright e'$ and $\tau \triangleright \tau'$ are well-formed, and $e' : \tau'$ – the `VExpr` witnesses are needed because typing is defined at the `VExpr` level, and the same trick as for `whnf` above gives `Expr`-side well-typedness of e and τ from a single statement.

We then define `RecM.WF` with a similar signature to `M.WF` to mean that it is valid when called with a well-formed `Methods`.

The specification of `inferLambda` from section 3.2.2 should now be unsurprising:

```
theorem inferLambda.WF
  (h1 : e.FVarsIn s.ngen.Reserves)
  (hinf : inferOnly = true → ∃ e', c.TrExpr e e') :
  (inferLambda e inferOnly).WF c s fun ty _ =>
    ∃ e' ty', c.TrExpr e e' ∧ c.TrExpr ty ty' ∧ c.HasType e' ty'
```

The theorem is proved compositionally over the code. At the loop, we need an inductive invariant which reveals some of the complications of having a sequence of fvars that have not yet been applied to some of the inputs:

```
theorem inferLambda.loop.WF
  {c : VContext} {e₀ : Expr}
  {m} [mwf : c.MLCWF m] {n} (hn : n ≤ m.length)
  (hdrop : m.dropN n hn = c.mlctx)
  (harr : arr.toList.reverse = (m.fvarRevList n hn).map .fvar)
  (he₀ : e₀ = m.mkLambda n hn ei)
  (hei : e.instantiateList ((m.fvarRevList n hn).map .fvar) = ei)
  (hbelow : ∀ P, IsFVarUpSet P c.vlctx → FVarsIn P e₀ →
    IsFVarUpSet (AllAbove c.vlctx P) m.vlctx ∧ FVarsIn (AllAbove c.vlctx P) ei ∧
    ∀ ty, FVarsIn (AllAbove c.vlctx P) ty →
    FVarsIn (AllAbove c.vlctx P) (m.mkForall n hn ty))
  (hr : e.FVarsIn (· ∈ m.vlctx.fvars))
  (hinf : inferOnly = true → ∃ e', (c.withMLC m).TrExprS ei e') :
  (inferLambda.loop inferOnly arr e).WF (c.withMLC m) s fun ty _ =>
    ∃ e' ty', c.TrTyping e₀ ty e' ty'
```

Here $m : \text{MLCtx}$ is the aforementioned local context consisting of `Expr` $\times$ `VExpr` pairs, which we use to locally extend the local context while leaving the rest of the `VContext` alone. Most of the hypotheses are gluing equations: m extends `c.mlctx` by n binders, `arr` holds the fresh fvars for them, e_i is e with those bvars instantiated, and e_0 is the reconstructed input lambda. The more interesting machinery is `hbelow`, which is a good entry point into the verification toolkit we use throughout:

- An `IsFVarUpSet` predicate P over a context Δ is a set of fvars closed under the dependency graph of Δ : if $v \in P$ and v has a type or value (in Δ) containing v' , then $v' \in P$ as well.
- `AllAbove` Δ P constructs an fvar set on a larger context Δ' as $P \cup (\Delta' \setminus \Delta)$, including all extension variables $x \in \Delta' \setminus \Delta$. Asserting this is an up-set therefore asserts that all extension variables have types definable from variables in P .

With this, `hbelow` says: any up-set property of e_0 over `c.vlctx` lifts (via `AllAbove`) to one over the extended `m.vlctx` that is also preserved by e_i and by wrapping a result type in `m.mkForall`. The intuition is cleanest in the strongest case, where P is the fvar set of e_0 itself: the loop is decomposing e_0 into e_i and a stack of binders, so both e_i and the binder types in m are subterms of e_0 and trivially share its fvars. The general $\forall P$ form makes the proofs more compositional. The remaining `mwf`, `hr`, and `hinf` mirror the preconditions of `inferLambda.WF`, lifted to the extended context. Crucially, the conclusion gives a type for e_0 , the original input, at the original context c , so it works with the tail recursive structure of the function.

4.1 Verification status

The main body of the typechecker has been fully verified along the lines above. That is, the four functions from section 3.2.1 are proved to satisfy the specifications from Section 4, and so by induction on recursion depth the functions all satisfy their specifications when lifted out of the `RecM` monad as well, with theorems such as `checkType.WF`. The areas that are still unfinished include:

- Anything related to inductive types (Section 6) or primitive projections;
- Unique typing and injectivity of type constructors (section 2.5);
- Verification of an experimental complete level equality algorithm following [15];
- Environment modifications (Section 5).

Of these, the most significant is the last one, since the kernel cannot be used without the `addDecl` function. But this at least shows that the kernel can act as a typechecker given a fixed environment.

4.2 Bit tricks and soundness bugs

However, in the course of performing the verification we have already run into some interesting issues, and at least one soundness bug (which has since been fixed in response to our report).

Recall that `inferLambda` calls `cheapBetaReduce`, so we need to prove that `cheapBetaReduce` (which morally performs only β -reduction) preserves `def.eq`. The `cheapBetaReduce` function in turn uses `hasLooseBVars` to determine whether the function is a constant function (since $\lambda \bar{x}. c$ is a constant function if c has no bvars after stripping all the binders without instantiating them). `hasLooseBVars e` is defined as `looseBVarRange e > 0`, so we are reduced to verifying `looseBVarRange`, which returns the largest unbound bvar.

One would expect this to be defined recursively over `Expr`, with $\text{LBR}(\text{bvar } i) = i + 1$, $\text{LBR}(\text{lam } \tau \ e) = \max(\text{LBR}(\tau), \text{LBR}(e) - 1)$, and so on. However, it is instead implemented by masking off 20 bits from a 64-bit data field `e.data` which is stored with every expression. So this naturally leads to the question: what happens on overflow? The calculation of `e.data` had an overflow check, but it was expressed in Lean using the `panic!` function, which signals an error on `stdout` but then continues execution with the default value of the type, which is 0 for `UInt64`, which also happens to be the worst possible answer in this case, since it effectively turns on all of the optimizations, including the one in question about `hasLooseBVars`. Concretely (using a variant of the bug for level expressions), $f^{2^{24}}(u)$ where $f(\ell) = \max(0, \ell)$ is a level expression which normalizes to u , but because the u is so deeply nested in the term it triggers the overflow and so Lean believes it to not have a variable in it, from which one can derive $f^{2^{24}}(u) \equiv f^{2^{24}}(v)$ and thus $U_0 \equiv U_1$.

This is a nice example of a bug that was found entirely theoretically, by failing to prove a theorem and working backwards to an actual exploitable soundness bug.

Unfortunately, the fix does not address all of the problems we identified. The way it was addressed was to make the `Expr.data` function opaque, which at most masks the issue; and also by moving the panic to C++ code, where it can “panic harder” and crash the program, so that the strange consequences of returning 0 are not observed. Using opaques to replace a low level definition with a model is a reasonable option, but currently we are still relying on an unsound assumption (`looseBVarRange_eq`), because the definition of `looseBVarRange` is still visible and so one can prove that e.g. `looseBVarRange e < 2^20` unconditionally, which means that `looseBVarRange` cannot be the “correct” definition, which means the theorem isn’t provable even though all counterexamples are unreachable (but Lean’s logic doesn’t know that).

5 Adding to the global environment

As with `VExpr` vs `Expr`, the kernel does not directly use the type `VEnv` from section 2.2; it uses `Environment` instead to store declarations. This type is complex and contains many details irrelevant to the kernel, but luckily we only really care about a few operations on it:

- `empty : Environment` – constructs an environment with no constants
- `add : Environment → ConstantInfo → Environment` – adds a `ConstantInfo` declaration to the environment
- `find? : Environment → Name → Option ConstantInfo` – retrieves a declaration by name

In effect, the environment is a fancy hashmap indexing `ConstantInfo` declarations by name. Not all extensions of a `VEnv` are legal: as defined there is no constraint on definitional equalities, but Lean only allows them in a few cases – definition unfolding, recursor reduction, and quotient reduction – captured in `Environment` through `ConstantInfo` additions (definitions, theorems, and so on).

So to express environment well-formedness we require it be built up recursively from nothing by adding any kind of well-typed declaration `VDecl`. The declarations are:

- `block n` blocks constant `n`, setting `env.constants n = some none` and preventing a re-declaration as mentioned above.
- `axiom { name := c, uvars := n, type :=  $\alpha$  }` adds $c : \alpha$ as an axiom to the constant map.
- `opaque { c, n,  $\alpha$ , value := e }` checks that $e : \alpha$, then adds $c : \alpha$ to the constant map.
- `def { c, n,  $\alpha$ , e }` does the same as `opaque` but also adds $c \equiv e : \alpha$ to `defeqs`.
- `example { uvars := n, type :=  $\alpha$ , value := e }` checks that $e : \alpha$ and leaves the environment as-is.
- `induct (d : VInductDecl)` adds all the constants and `def.eq`’s from an inductive declaration (see Section 6).
- `quot` adds the quotient axioms and definitional equality. This is a type operator α/R where $R : \alpha \rightarrow \alpha \rightarrow \text{Prop}$ with `mk` : $\alpha \rightarrow \alpha/R$ and `lift` : $(f : \alpha \rightarrow \beta) \rightarrow (\forall xy. R\ x\ y \rightarrow f\ x = f\ y) \rightarrow (\alpha/R \rightarrow \beta)$, with the property that $(\text{lift } f\ h\ (\text{mk } a) \equiv f\ a : \beta)$. (One can construct every part of this in Lean except for the definitional equality.)

This enumeration is similar to the `Declaration` type, the actual front end to the kernel, but it lacks `unsafe` declarations and `mutualDefnDecl` (which, despite the name, is not for mutual definitions but rather unsafe declarations with unchecked self-referential definitions). We deliberately do not model `unsafe` declarations: as certain checks are disabled, they no longer follow the theory, and in any case they play no role in checking theorems.

6 Inductive types

Lean’s implementation of inductive types is based on Dybjer [13]. It differs from Rocq in that rather than primitive `fix` and `match` expressions, we have constants called “recursors” generated from the inductive specification. This significantly simplifies `VExpr`, which has no special support for anything inductive-related beyond the generic `def.eq. hook T-EXTRA`, and means we need no complex guard checker – something MetaRocq axiomatizes [29] and a past source of soundness bugs. Instead the complexity moves to recursor generation.

For single inductives, the algorithm is described in full in LTT [7]. In Lean 3, nested and mutual inductives were simulated using single inductives so the kernel only had to deal with the single inductive case, but for performance reasons Lean 4 moved the checking to the kernel. A mutual block such as `Even` and `Odd` over `Nat`, with `Even.succ : Odd n → Even (n+1)` and `Odd.succ' : Even n → Odd (n+1)` generates the expected constructors together with one recursor per type, `Even.rec` and `Odd.rec`. Each recursor takes a motive and minor premises for *both* types in the block, so that eliminating an `Even n` can recurse into its nested `Odd` subterms. The associated ι -reduction definitional equalities fire when a recursor meets a constructor, dispatching to the matching minor premise and recursing through the sibling recursor; e.g. `@Even.rec .. (Even.succ e)` reduces to `@Fs n e (@Odd.rec .. e)`.

For nested inductives, the checking procedure effectively amounts to running the simulation procedure from Lean 3 to get a mutual inductive type and performing well-formedness checks there, then removing all references to the unfolded type in the recursor.

Currently, `Lean4Lean` contains a complete implementation of inductive types, including nested and mutual inductives and all the typechecking effects of this, but not much work has been done on the theoretical side, defining what an inductive specification should generate.

6.1 Eta for structures

Another new kernel feature in Lean 4 is primitive projections and η for structures: if s is a **structure** (a one-constructor non-recursive **inductive** type), then $s \equiv \text{mk}(s.1, \dots, s.n)$ where $s.i$ is the i th projection (`Expr.proj`). This already held propositionally (by induction on s), but is now definitional, so the kernel must unfold structures as needed. η for unit in particular is known to be problematic [17, 19], but Lean’s kernel does not implement a complete decision procedure anyway; we believe it poses no soundness issue and the main results should still hold even with this extension.

7 Results

`Lean4Lean` contains a command-line frontend for validating already-compiled Lean projects. It reads `.olean` files, essentially serialized (unordered) lists of `ConstantInfo` objects, and topologically sorts the definitions and passes them to the `Lean4Lean` kernel, a function

```
def addDecl (env : Environment) (decl : Declaration) (check := true) :
  Except KernelException Environment
```

that is a drop-in replacement for `Environment.addDeclCore`, an opaque Lean function implemented by FFI (foreign-function interface) to the corresponding C++ kernel function.

This is the same interface (and to some extent, the same code) as `lean4checker`,²¹ which does the same but calls Lean’s `addDecl` instead, using the C++ kernel as an external verifier. Using the Lean kernel as an external verifier for itself may seem odd, but it catches malicious

²¹<https://github.com/leanprover/lean4checker>

	lean4checker	lean4lean	ratio
Lean	37.01 s	44.61 s	1.21
Batteries	32.49 s	45.74 s	1.40
Mathlib + Batteries + Lean	44.54 min	58.79 min	1.32

■ **Figure 2** Comparison of the original C++ implementation of the Lean kernel (`lean4checker`) with `lean4lean` (Lean) on major Lean packages. Tests were performed on a 12 core 12th Gen Intel i7-1255U @ 2.1 GHz, single-threaded, on rev. 526c94c of `mathlib4`.

(or confused) code that uses metaprogramming to bypass the kernel and add constants to the environment without typechecking. For our purposes it is a handy benchmark, since we perform the same operations with a different kernel. Running `lake env lean4lean --fresh Mathlib` in `mathlib4` will check the `Mathlib` module and all of its dependencies.²²

The results are summarized in Figure 2. The new implementation is around 30% slower than the C++ implementation, which we attribute mainly to the Lean compiler and data representation; nevertheless it is practically usable on large libraries, a strong sign. We do not expect it to get faster without compiler improvements, but we intend to keep the speed from regressing as verification grows; the external verification makes this possible in principle, though more tooling may be needed to verify the implementation as-is in practice.

8 Related work

Self-verification of ITP systems has been done in Rocq [4], HOL Light [16], Isabelle [22], and Metamath Zero [8], and also overlaps with bootstrapping theorem provers such as Milawa [11] and Candle [2, 18]. But the most similar work is unquestionably MetaRocq [28, 29, 30], which is a project to develop a verified typechecker for Rocq, in Rocq. It is significantly more complete than the previous effort [4], but is not yet capable of verifying the Rocq standard library due to lack of eta-conversion and template polymorphism; eta-conversion is work in progress [19], and universe polymorphism being reorganised in Rocq’s kernel [24]. With Lean4Lean we took the approach of quickly getting to feature parity with the real Lean kernel and proving correctness later, so we ended up with a feature-complete typechecker but we are still lacking some theorems. Our work also overlaps to some extent with Template-Rocq [3, 5, 33], a certified metaprogramming framework for Rocq developed in the MetaRocq repo. A parallel effort for Agda, Agda Core [9, 10], is currently in development.

As was mentioned, MetaRocq also has to deal with many of the same issues proving properties of the typing judgment, and one may hope for proofs from MetaRocq to be useful in Lean4Lean and vice-versa. Unfortunately, there are some important details that differ:

- In MetaRocq, the conversion relation is a partial order rather than an equivalence relation, because of universe cumulativity, so Conjecture 7 doesn’t hold as stated. Nevertheless, there are theorems regarding “principal types” which can be used to play the same role.
- Unlike in Lean4Lean, the conversion relation is *untyped*: there is no mutual induction between typing and definitional equality, which massively simplifies matters. In Lean this is unfortunately not an option because of the T-PROOF-IRREL rule, which is expressed in a different way in Rocq.
- Lean’s typechecker is known to not be complete, owing to some of the undecidability results in LTT [7], but MetaRocq [30, 20] has a proof of completeness.

²²Single-threaded; per-module parallel checking is possible but harder to benchmark.

9 Conclusion

We have presented **Lean4Lean**, a reimplementaion of the Lean 4 typechecker written in Lean itself and closely following the kernel. Despite its simplicity, **Lean4Lean** can validate large-scale developments such as **Mathlib** with competitive performance.

More importantly, **Lean4Lean** serves as a basis for formal reasoning about Lean’s metatheory. By internalizing the type system, we can verify properties of the checker and use these results to increase confidence in the production kernel. Already, this process has revealed subtle issues and motivated fixes upstream.

Looking ahead, we plan to formalize more of LTT [7], including in particular the proof of relative consistency given a large cardinal assumption. By continuing the verification effort, we hope to discover more potential bugs in the production kernel. In the long term, we hope that this work will lead to a fully verified, self-hosted Lean kernel.

References

- 1 Andreas Abel and Thierry Coquand. Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality. *Logical Methods in Computer Science*, 16(2), June 2020. doi:10.23638/LMCS-16(2:14)2020.
- 2 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A Verified Implementation of HOL Light. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, Haifa, Israel, August 7-10, 2022*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2022.3.
- 3 Abhishek Anand, Simon Boulrier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 20–39, Oxford, United Kingdom, July 2018. Springer. doi:10.1007/978-3-319-94821-8_2.
- 4 Bruno Barras and Benjamin Werner. Coq in Coq. *Available on the WWW*, 1997.
- 5 Simon Pierre Boulrier. *Extending type theory with syntactic models. (Etendre la théorie des types à l’aide de modèles syntaxiques)*. Theses, Ecole nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire, France, November 2018. URL: <https://tel.archives-ouvertes.fr/tel-02007839>.
- 6 Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005. doi:10.1017/S0960129505004822.
- 7 Mario Carneiro. The Type Theory of Lean. Master’s thesis, Carnegie Mellon University, 2019. URL: <https://github.com/digama0/lean-type-theory/releases/tag/v1.0>.
- 8 Mario Carneiro. Metamath Zero: Designing a Theorem Prover Prover. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 71–88, Berlin, Heidelberg, 2020. Springer. doi:10.1007/978-3-030-53518-6_5.
- 9 Jesper Cockx. Agda Core. <https://github.com/jespercockx/agda-core>, 2024.
- 10 Jesper Cockx. Agda Core: The Dream and the Reality. Talk at IFIP WG 2.8, <https://jesper.cx/files/IFIP28-2024.html>, April 2024.
- 11 Jared Curran Davis. *A self-verifying theorem prover*. PhD thesis, University of Texas, 2009.
- 12 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on*

- Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, Springer, 2015. doi:10.1007/978-3-319-21401-6_26.
- 13 Peter Dybjer. Inductive Families. *Formal Aspects of Computing*, 6(4):440–465, 1994. doi:10.1007/BF01211308.
 - 14 Gabriel Ebner. trepplein: a Lean 3 Type Checker. <https://github.com/gebner/trepplein>, 2022.
 - 15 Yoan Gérard. A Canonical Form for Universe Levels in Impredicative Type Theory. In Stefano Guerrini and Barbara König, editors, *34th EACSL Annual Conference on Computer Science Logic, CSL 2026, Paris, France, February 23-28, 2026*, volume 363 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2026. doi:10.4230/LIPIcs.CSL.2026.39.
 - 16 John Harrison. Towards Self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer. doi:10.1007/11814771_17.
 - 17 András Kovács. Eta conversion for the unit type (is still not that simple). In *WITS 2025 - Workshop on the Implementation of Type Systems*, 2025. URL: <https://popl25.sigplan.org/details/wits-2025-papers/4/>.
 - 18 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192, New York, NY, USA, January 2014. ACM. doi:10.1145/2535838.2535841.
 - 19 Meven Lennon-Bertrand. À bas l’ η — Coq’s troublesome η -conversion. In *Proceedings of the First Workshop on the Implementation of Type Systems (WITS)*, 2022. URL: <https://www.meven.ac/documents/22-WITS-abstract.pdf>.
 - 20 Meven Lennon-Bertrand. *Bidirectional Typing for the Calculus of Inductive Constructions. (Typage Bidirectionnel pour le Calcul des Constructions Inductives)*. Theses, University of Nantes, France, June 2022. URL: <https://tel.archives-ouvertes.fr/tel-03848595>.
 - 21 Meven Lennon-Bertrand. What Does It Take to Certify a Conversion Checker? In Maribel Fernández, editor, *10th International Conference on Formal Structures for Computation and Deduction, FSCD 2025, Birmingham, UK, July 14-20, 2025*, volume 337 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:23, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2025.27.
 - 22 Tobias Nipkow and Simon Roßkopf. Isabelle’s Metalogic: Formalization and Proof Checker. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 93–110, Berlin, Heidelberg, 2021. Springer. doi:10.1007/978-3-030-79876-5_6.
 - 23 Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015. URL: <https://inria.hal.science/hal-01094195>.
 - 24 Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédro, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. All Your Base Are Belong to Us: Sort Polymorphism for Proof Assistants. *Proceedings of the ACM on Programming Languages*, 9(POPL):2253–2281, January 2025. Conference talk, Session: POPL Research Papers, Fri 24 Jan 2025 17:40–18:00, Proof Assistants track. doi:10.1145/3704912.
 - 25 Daniel Selsam. tc: Reference Type Checker for Lean 3. <https://github.com/dselsam/tc>, 2017.

- 26 Vincent Siles and Hugo Herbelin. Pure Type System conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, 2012. doi:10.1017/S0956796812000044.
- 27 Matthieu Sozeau. Touring the MetaCoq Project (Invited Paper). In Elaine Pimentel and Enrico Tassi, editors, *Proceedings of the Sixteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2021, Pittsburgh, USA, 16th July 2021*, volume 337 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 13–29, Pittsburgh, United States, July 2021. doi:10.4204/EPTCS.337.2.
- 28 Matthieu Sozeau, Abhishek Anand, Simon Boulrier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The Meta-Coq Project. *Journal of Automated Reasoning*, 64(5):947–999, February 2020. doi:10.1007/s10817-019-09540-0.
- 29 Matthieu Sozeau, Simon Boulrier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):8:1–8:28, December 2020. doi:10.1145/3371076.
- 30 Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *Journal of the ACM*, 72(1):8:1–8:74, January 2025. doi:10.1145/3706056.
- 31 The Coq Development Team. The Coq Proof Assistant, April 2020. doi:10.5281/zenodo.3744225.
- 32 Sebastian Andreas Ullrich. *An Extensible Theorem Proving Frontend*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2023. doi:10.5445/IR/1000161074.
- 33 Théo Winterhalter. *Formalisation and Meta-Theory of Type Theory. (Formalisation et Méta-Théorie de la Théorie des Types)*. PhD thesis, University of Nantes, France, 2020. URL: <https://tel.archives-ouvertes.fr/tel-05425836>.

Symmetries in Sorting

Vikraman Choudhury   

University of Strathclyde, Glasgow, UK

Università di Bologna, Italy

Wind Wong   

Vrije Universiteit Amsterdam, The Netherlands

Abstract

Sorting algorithms are fundamental to computer science, and their correctness criteria are well understood as rearranging elements of a list according to a specified total order on the underlying set of elements. As mathematical functions, they are functions on lists that perform combinatorial operations on the representation of the input list. In this paper, we study sorting algorithms conceptually as abstract sorting functions.

There is a canonical surjection from the free monoid on a set (lists of elements) to the free commutative monoid on the same set (multisets of elements). We show that sorting functions determine a section (right inverse) to this surjection satisfying two axioms, that do not presuppose a total order on the underlying set. Then, we establish an equivalence between (decidable) total orders on the underlying set and correct sorting functions.

The first part of the paper develops concepts from universal algebra from the point of view of functorial signatures, and gives constructions of free monoids and free commutative monoids in (univalent) type theory. Using these constructions, the second part of the paper develops the axiomatisation of sorting functions. The paper uses informal mathematical language, and comes with an accompanying formalisation in Cubical Agda.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Type theory; Theory of computation $\rightarrow$ Logic and verification; Theory of computation $\rightarrow$ Constructive mathematics

Keywords and phrases universal algebra, type theory, homotopy type theory, cubical Agda, constructive mathematics, univalent mathematics, sorting, combinatorics, formalisation

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.3

Related Version *Extended Version*: [arXiv:2512.07349](https://arxiv.org/abs/2512.07349) [11]

Supplementary Material *Software (Formalization)*: <https://github.com/windtf/agda-symmetries> [10]; archived at [swh:1:dir:5e86a19fe276e62519af96e6bf3c8ec42f8fe4f2](https://swh.1:dir:5e86a19fe276e62519af96e6bf3c8ec42f8fe4f2)

Funding *Vikraman Choudhury*: Supported by the EU Marie-Sklodowska-Curie action “ReGraDe-CS”, grant № 101106046.

Wind Wong: Supported by the European Research Council Starting Grant for the project “SecuStack”, funded under the European Union’s Horizon Europe grant № 101115046.

Acknowledgements This project was carried out as part of Wind Wong’s undergraduate thesis at University of Glasgow, supervised by Simon Gay. We are grateful to Simon for his guidance and support. We are also grateful to Fredrik Nordvall Forsberg, and the reviewers of the HoTT/UF 2024 meeting, for their helpful comments and suggestions.

1 Introduction

Consider a puzzle about sorting, inspired by Dijkstra’s Dutch National Flag problem [15, Ch.14]. Suppose there are balls of three colours, corresponding to the colours of the Dutch flag: red, white, and blue. Given an unordered list (bag) of such balls, how many ways can



© Vikraman Choudhury and Wind Wong;

licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 3; pp. 3:1–3:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

you sort them into the Dutch flag?

[red, red, blue, white, blue, red, white, blue]

Obviously there is only one way, decided by the order the colours appear in the Dutch flag: red < white < blue.

[red, red, red, white, white, blue, blue, blue]

What if we are avid enjoyers of vexillology who also want to consider other flags? How many ways can we sort our unordered list? To answer this, we distinguish between a sorting *algorithm* (a computational procedure) and a sorting *function* (a mathematical mapping). Since we sort the elements of a carrier set A directly (so stability and metadata are not concerns), a sorting function must produce a unique, deterministic output list for each input bag. Thus, “how many ways to sort” means how many extensionally distinct sorting functions can be defined. Because there are $3! = 6$ permutations of {red, white, blue}, there are 6 possible orderings, corresponding to 6 categories of tricolour flags (see Wikipedia):



For a carrier set with 3 elements, the 6 possible total orderings yield exactly 6 *extensionally correct* sorting functions. Intuitively, a *correct* sorting function takes a bag and returns a unique, sorted list of the same elements. Extensional correctness therefore requires the output list to be a permutation of the input elements, and sorted according to a fixed total order. Formally, there is a bijection between total orderings on a carrier set A and correct sorting functions on lists over A . Our main contribution is showing how a sorting function can be correctly axiomatised and formalised from this point of view.

Outline and Contributions

The paper is organised as follows:

- In § 2, we describe a formalisation of universal algebra developed from the point of view of functorial signatures, the definition and universal property of free algebras, and equational systems over signatures. These are elementary constructions but necessary groundwork for us, and we give an informal exposition in type theory.
- In § 3, we give constructions of free monoids, and proofs of their universal property. Following this, in § 4, we add commutativity to each representation of free monoids, and extend the proofs of universal property from free monoids to free commutative monoids. These constructions are standard, and we formalise by using our general framework.
- In § 5, we build on the previous constructions and study sorting functions. The main result connects total orders, sorting, and commutativity, by proving an equivalence between decidable total orders on a carrier set A , and correct sorting functions defined using free monoids and free commutative monoids over A .
- Finally, § 6 discusses aspects of the formalisation, related and future work.

These sections are largely independent. Readers interested in universal algebra can start at § 2; those focused on free (commutative) monoids can skip to §§ 3 and 4; and those interested primarily in sorting can proceed directly to § 5. All results are formalised in Cubical Agda, available online at [10]. We use informal mathematical language in the paper to keep it accessible. An extended version with additional details is on arXiv [11].

No prior exposure to Homotopy Type Theory (HoTT) or Univalence is required. We assume only basic familiarity with dependent type theory and functional programming (such as Agda, Coq, or Haskell). While the formalisation is written in Cubical Agda and uses higher inductive types and univalence for quotient constructions, the paper's text is presented in informal mathematical language.

2 Universal Algebra

We develop some basic notions from universal algebra and equational logic [6], which gives us a vocabulary and framework to express our results in. The point of view we take is the standard category-theoretic approach to universal algebra [32], which is an alternative perspective to the Lawvere theory or abstract clone point of view [23, 19]. We keep a running example of monoids in mind, while explaining and defining the abstract concepts.

2.1 Algebras

► **Definition 1** (⚙️ Signature). A signature, denoted σ , is a (dependent) pair consisting of a set of operations, $\text{op} : \text{Set}$, and an arity function for each operation, $\text{ar} : \text{op} \rightarrow \text{Set}$.

► **Example 2.** A premonoid is a set with an identity element (a nullary operation), and a binary multiplication operation, with signature $\sigma_{\text{Mon}} := (\text{Fin}_2, \lambda\{0 \mapsto \text{Fin}_0; 1 \mapsto \text{Fin}_2\})$. Informally, we denote the two operations as a tuple $(e, \bullet)$.

Every signature σ induces a signature functor F_σ on Set .

► **Definition 3** (⚙️ Signature functor).

$$X \mapsto (o : \text{op}) \times X^{\text{ar}(o)} \quad X \xrightarrow{f} Y \mapsto (o : \text{op}) \times X^{\text{ar}(o)} \xrightarrow{(o, - \circ f)} (o : \text{op}) \times Y^{\text{ar}(o)}$$

► **Example 4.** The signature functor for monoids, $F_{\sigma_{\text{Mon}}}$, assigns to a carrier set X , the set of inputs for each operation. Expanding the dependent product on Fin_2 , we obtain a coproduct of sets: $F_{\sigma_{\text{Mon}}}(X) \simeq X^{\text{Fin}_0} + X^{\text{Fin}_2} \simeq \mathbf{1} + X \times X$.

A σ -structure is given by a carrier set, with functions interpreting each operation symbol. The signature functor applied to a carrier set gives the inputs to each operation, and the output is simply a map back to the carrier set. These two pieces of data constitute an algebra for the F_σ functor. We write $\mathfrak{X}$ for a σ -structure with carrier set X , following the model-theoretic notational convention.

► **Definition 5** (⚙️ Structure). A σ -structure $\mathfrak{X}$ is an F_σ -algebra, that is, a pair consisting of a carrier set X , and an algebra map $\alpha_X : F_\sigma(X) \rightarrow X$.

► **Example 6.** Concretely, an $F_{\sigma_{\text{Mon}}}$ -algebra has the type

$$\alpha_X : F_{\sigma_{\text{Mon}}}(X) \rightarrow X \quad \simeq \quad (\mathbf{1} + X \times X) \rightarrow X \quad \simeq \quad (\mathbf{1} \rightarrow X) \times (X \times X \rightarrow X)$$

which is the pair of functions interpreting the two operations. Natural numbers $\mathbb{N}$ with $(0, +)$ or $(1, \times)$ are examples of monoid structures.

► **Definition 7** (⚙️ σ -Homomorphism). A homomorphism between σ -structures $\mathfrak{X}$ and $\mathfrak{Y}$ is a morphism of F_σ -algebras, that is, a map $f : X \rightarrow Y$ making the following diagram commute:

$$\begin{array}{ccc} F_\sigma(X) & \xrightarrow{\alpha_X} & X \\ F_\sigma(f) \downarrow & & \downarrow f \\ F_\sigma(Y) & \xrightarrow{\alpha_Y} & Y \end{array}$$

► **Example 8.** Given two monoid structures $\mathfrak{X}$ and $\mathfrak{Y}$, the top half denotes:

$1 + (X \times X) \xrightarrow{\alpha_X} X \xrightarrow{f} Y$, which applies f to the output of each operation, and the bottom half denotes the map: $1 + (X \times X) \xrightarrow{F_{\sigma\text{Mon}}(f)} 1 + (Y \times Y) \xrightarrow{\alpha_Y} Y$. In other words, a homomorphism between $\mathfrak{X}$ and $\mathfrak{Y}$ is a map f on the carrier sets that commutes with the interpretation of the monoid operations, or simply, preserves the monoid structure.

For a fixed signature σ , the category of F_σ -algebras and their morphisms form a category of algebras, written $F_\sigma\text{-Alg}$, or simply, $\sigma\text{-Alg}$, given by the obvious definitions of identity and composition for the underlying functions.

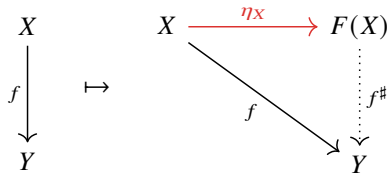
2.2 Free Algebras

The concrete category $\sigma\text{-Alg}$ of structured sets and structure-preserving maps admits a forgetful functor $U: \sigma\text{-Alg} \rightarrow \text{Set}$. In our notation, $U(\mathfrak{X})$ is simply X , a fact we exploit to simplify our notation and formalisation. The left adjoint to this forgetful functor is the free algebra construction, also known as the term algebra or the *absolutely free* algebra without equations [27]. We rephrase this in more concrete terms.

- **Definition 9** (⚙️ Free Algebras). *A free σ -algebra construction consists of the following:*
- a set $F(X)$, for every set X ,
 - a σ -structure on $F(X)$, written as $\mathfrak{F}(X)$,
 - a universal map $\eta_X: X \rightarrow F(X)$, for every X , such that,
 - for any σ -algebra $\mathfrak{Y}$, the operation assigning to each homomorphism $f: \mathfrak{F}(X) \rightarrow \mathfrak{Y}$, the map $f \circ \eta_X: X \rightarrow Y$ (or, pre-composition with η_X), is an equivalence.

In other words, we ask for a bijection between the set of homomorphisms out of the free algebra to any other algebra, and the set of functions from the carrier set of the free algebra to the carrier set of the other algebra. There should be no more data in homomorphisms out of the free algebra than there is in functions out of the carrier set, which is the property of *freeness*. The inverse operation to post-composing with η_X extends a function to a homomorphism.

- **Definition 10** (⚙️ Universal extension). *The universal extension of a function $f: X \rightarrow Y$ to a homomorphism out of the free σ -algebra on X is written as $f^\#: \mathfrak{F}(X) \rightarrow \mathfrak{Y}$. It satisfies the identities: $f^\# \circ \eta_X = f$, $\eta_X^\# = \text{id}_{\mathfrak{F}(X)}$, and $(g^\# \circ f)^\# = g^\# \circ f^\#$. The universal morphism η_X is highlighted in red.*



Free algebra constructions are canonically equivalent.

- **Proposition 11** (⚙️). *If $\mathfrak{F}(X)$ and $\mathfrak{G}(X)$ are free σ -algebras on X , then $\mathfrak{F}(X) \simeq \mathfrak{G}(X)$ in $\sigma\text{-Alg}$, naturally in X , via the canonical extensions.*

We have discussed specifications of free algebras, but not actually given a construction. In type theory, *free* constructions are often given by inductive types, where the constructors are the pieces of data that freely generate the structure, and the type-theoretic induction principle enforces the category-theoretic universal property.

► **Definition 12** (⚙️ Construction of Free σ -algebras). *The free σ -algebra on a type X is given by the inductive type $\text{Tree}(X)$, generated by two constructors:*

$$\begin{aligned} \text{leaf} &: X \rightarrow \text{Tree}(X) \\ \text{node} &: F_\sigma(\text{Tree}(X)) \rightarrow \text{Tree}(X) \end{aligned}$$

The constructors `leaf` and `node` are the generators for the universal map, and the algebra map, respectively. Together, they describe the type of abstract syntax trees for terms in the signature σ – the leaves are the free variables, and the nodes are the branching operations of the tree, marked by the operations in σ .

► **Proposition 13** (⚙️). *$(\text{Tree}(X), \text{leaf})$ is the free σ -algebra on X .*

2.3 Equations

The algebraic framework described so far only captures operations, not equations. These algebras are *lawless* (or *wild* or *absolutely free*) – $F_{\sigma_{\text{Mon}}}$ -algebras are premonoids rather than monoids, and $\mathfrak{F}_{\sigma_{\text{Mon}}}$ -algebras are free premonoids, not free monoids, since they are missing the unit and associativity laws. To impose equations on the generated abstract syntax trees, we adopt the point of view of equational logic.

► **Definition 14** (⚙️ Term Arity). *A term arity, denoted ε , is a (dependent) pair consisting of: a set of names for equations, $\text{eq} : \text{Set}$, and an arity of free variables for each equation, $\text{fv} : \text{eq} \rightarrow \text{Set}$.*

This will be used to define a *system of equations* (Definition 17).

► **Example 15.** The term arity for monoids ε_{Mon} is: $(\text{Fin}_3, \lambda\{0 \mapsto \text{Fin}_1; 1 \mapsto \text{Fin}_1; 2 \mapsto \text{Fin}_3\})$. The three equations are the left and right unit laws, and the associativity law – a 3-element set of names $\{\text{unitl}, \text{unitr}, \text{assoc}\}$. The two unit laws use one free variable, and the associativity law uses three free variables.

Similar to the term signature functor in Definition 3, this produces a *term arity functor* on Set .

► **Definition 16** (⚙️ Term Arity Functor).

$$X \mapsto (e : \text{eq}) \times X^{\text{fv}(e)} \quad X \xrightarrow{f} Y \mapsto (e : \text{eq}) \times X^{\text{fv}(e)} \xrightarrow{(e, - \circ f)} (e : \text{eq}) \times Y^{\text{fv}(e)}$$

To build equations out of this, we use the σ -operations and construct trees for the left and right-hand sides of each equation using the free variables available.

► **Definition 17** (⚙️ System of Equations). *A system of equations over a signature (σ, ε) is a pair of natural transformations:*

$$\ell, \nu : F_\varepsilon \Rightarrow \mathfrak{F}_\sigma.$$

For any set (of variables) V , this gives a pair of functions $\ell_V, \nu_V : F_\varepsilon(V) \rightarrow \mathfrak{F}_\sigma(V)$, and naturality ensures correctness of renaming.

Finally, we say how a given σ -structure $\mathfrak{X}$ satisfies the system of equations $T_{(\sigma, \varepsilon)}$. We need to assign a value to each free variable in the equation in the carrier set, using a valuation function $\rho : V \rightarrow X$. Given such an assignment, we evaluate the left and right trees of the equation, by extending ρ with Definition 10, that is by construction, a homomorphism from $\mathfrak{F}(V)$ to $\mathfrak{X}$. To satisfy an equation, these two evaluations should agree.

► **Definition 18** (⚙️ Satisfaction $\mathfrak{X} \models T$). A σ -structure $\mathfrak{X}$ satisfies the system of equations $T_{(\sigma, \varepsilon)}$ if for every set V , and every assignment $\rho: V \rightarrow X$, $\rho^\#$ is a (co)fork:

$$F_\varepsilon(V) \begin{array}{c} \xrightarrow{\ell_V} \\ \xrightarrow{r_V} \end{array} \mathfrak{F}(V) \xrightarrow{\rho^\#} \mathfrak{X}$$

Given a signature (σ, ε) with a system of equations $T_{(\sigma, \varepsilon)}$, the σ -algebras satisfying $T_{(\sigma, \varepsilon)}$, or varieties, form a full subcategory of $\sigma\text{-Alg}$. While free algebras exist for any signature σ as in Definition 12, constructions of free varieties for arbitrary systems of equations require non-constructive principles [7, § 7, pg.142], in particular, the arity sets need to be projective – we do not pursue this matter further. We have a rudimentary framework for universal algebra and equational logic, which gives us enough tools to develop the next sections.

3 Constructions of Free Monoids

In this section, we consider various constructions of free monoids in type theory, with proofs of their universal property. Since each construction satisfies the same categorical universal property, by Proposition 11, these are canonically equivalent (hence equal, by univalence) as types (and as monoids), allowing us to transport proofs between them. There are countably many such representations of free monoids, allowing us to choose the most convenient one for a given task. Using the universal property allows us to define and prove our programs correct in one go, which we use in § 5.

3.1 Lists

Cons-lists are simple inductive datatypes, well-known to functional programmers, and are a common representation of free monoids in programming languages. More generally, they correspond to list objects [12, 25] and algebraically-free monoids [21] in category theory.

► **Definition 19** (⚙️ Lists). $\text{List}(A)$ is generated by the following constructors:

$$\begin{aligned} \text{nil} &: \text{List}(A) \\ - :: - &: A \rightarrow \text{List}(A) \rightarrow \text{List}(A) \end{aligned}$$

The (universal) generators map is the singleton: $\eta_A(a) := [a] \equiv a :: \text{nil}$, the identity element is the empty list nil , and the monoid multiplication $\bullet$ is given by list concatenation.

► **Proposition 20** (⚙️ Universal property for List). $(\text{List}(A), \eta_A)$ is the free monoid on A .

3.2 Arrays

An alternate (non-inductive) representation of the free monoid on a carrier set, or alphabet A , is A^* , the set of all finite strings or sequences of characters *drawn* from A , as in [16]. We call this an *array*, which is a pair of a natural number n , denoting the length of the array, and a lookup (or index) function $\text{Fin}_n \rightarrow A$, mapping indices to elements of A . In type theory, this is also understood as an extension of a container [1], where the type of shapes is a $n: \mathbb{N}$, and the type of positions for each shape is Fin_n , where the array elements are stored.

► **Definition 21** (⚙️ Arrays).

$$\text{Array}(A) := (n: \mathbb{N}) \times (\text{Fin}_n \rightarrow A)$$

For example, $(3, \lambda\{0 \mapsto 3, 1 \mapsto 1, 2 \mapsto 2\})$ represents the same list as $[3, 1, 2]$. The (universal) generators map is the singleton: $\eta_A(a) = (1, \lambda\{0 \mapsto a\})$, the identity element is $(0, \lambda\{\})$ and the monoid operation $\bullet$ is given by array concatenation.

► **Definition 22** (⚙️ Concatenation). *The concatenation operation $\bullet$, is defined below, where $\oplus: (\text{Fin}_n \rightarrow A) \rightarrow (\text{Fin}_m \rightarrow A) \rightarrow (\text{Fin}_{n+m} \rightarrow A)$ combines the two lookup functions.*

$$(n, f) \bullet (m, g) = (n + m, f \oplus g) \quad (f \oplus g)(k) = \begin{cases} f(k) & \text{if } k < n \\ g(k - n) & \text{otherwise} \end{cases}$$

We need two auxiliary lemmas about array concatenation to do induction on arrays.

► **Lemma 23** (⚙️ Array cons). *Any array $(S(n), f)$ is equal to $\eta_A(f(0)) \bullet (n, f \circ S)$.*

► **Lemma 24** (⚙️ Array split). *For any array $(S(n), f)$ and (m, g) ,*

$$(n + m, (f \oplus g) \circ S) = (n, f \circ S) \bullet (m, g) .$$

► **Definition 25** (⚙️ Universal extension). *Given a monoid $\mathfrak{X}$, and a map $f: A \rightarrow X$, we define $f^\#: \text{Array}(A) \rightarrow X$, by induction on the length of the array:*

$$\begin{aligned} f^\#(0, g) &= e \\ f^\#(S(n), g) &= f(g(0)) \bullet f^\#(n, g \circ S) \end{aligned}$$

► **Proposition 26** (⚙️ Universal property for Array). *$(\text{Array}(A), \eta_A)$ is the free monoid on A .*

Proof sketch. We need to show that $(-)^{\#}$ is an inverse to $(-) \circ \eta_A$. $f^{\#} \circ \eta_A = f$ for all set functions $f: A \rightarrow X$ holds trivially. To show $(f \circ \eta_A)^{\#} = f$ for all homomorphisms $f: \text{Array}(A) \rightarrow \mathfrak{X}$, we need to show $\forall xs. (f \circ \eta_A)^{\#}(xs) = f(xs)$. We use Lemmas 23 and 24 to do induction on xs , similar to doing induction on the inductive datatype $\text{List}(A)$. ◀

An alternative proof of the universal property for Array can be given by directly constructing an equivalence (of types, and monoid structures) between $\text{Array}(A)$ and $\text{List}(A)$ (using `tabulate` and `lookup`), and then applying univalence and transport (see formalisation).

4 Constructions of Free Commutative Monoids

The next step is to add commutativity to the monoid multiplication in each construction of free monoids. Informally, adding commutativity to free monoids turns “ordered lists” to “unordered lists”, where the ordering is the one imposed by the position or index of the elements in the list. This is crucial to our goal of studying sorting, as we will study sorting as a function mapping back unordered lists to ordered lists, later in § 5.3.

It is known that finite multisets are (free) commutative monoids, under the operation of multiset union: $xs \cup ys = ys \cup xs$. The order is “forgotten” in the sense that it doesn’t matter how two multisets are union-ed together, such as $\{a, a, b, c\}$ and $\{b, a, c, a\}$ are equal as finite multisets (justifying the bag notation). Formally, $\{x, y, \dots\}$ denotes $\eta_A(x) \bullet \eta_A(y) \bullet \dots : \mathcal{M}(A)$.

This is unlike free monoids, where $[a, a, b, c] \neq [b, a, c, a]$ (justifying the list notation). Formally, $[x, y, \dots]$ denotes $\eta_A(x) \bullet \eta_A(y) \bullet \dots : \mathcal{L}(A)$, and $x :: xs$ denotes $\eta_A(x) \bullet xs : \mathcal{L}(A)$.

4.1 Free monoids quotiented by permutation relations

Instead of constructing free commutative monoids directly, the first construction we study is to take *any* free monoid and quotient by *symmetries*. The constructions in § 4.2 and § 4.4 are specific instances of this recipe.

From the perspective of universal algebra and equational logic developed in § 2, we work in the signature of monoids, but extend the equational theory of monoids with commutativity. If $(\mathfrak{F}(A), \eta)$ is a free monoid construction satisfying its universal property, then we quotient $F(A)$ by an *appropriate* symmetry relation $\approx$. This is precisely a *permutation relation*.

► **Definition 27** (⚙️ Permutation relation). *A binary relation on a free monoid $(F(A), e, \bullet)$ is a permutation relation iff it is:*

- *reflexive, symmetric, transitive (an equivalence),*
- *a congruence w.r.t $\bullet$: $\forall a, b, c, d, a \approx b \rightarrow c \approx d \rightarrow a \bullet c \approx b \bullet d$,*
- *commutative: $\forall a, b, a \bullet b \approx b \bullet a$, and*
- *respects $(-)^{\#}$: $\forall a, b, f, a \approx b \rightarrow f^{\#}(a) = f^{\#}(b)$.*

Note that being a permutation relation is a proposition. We write $F(A) // \approx$ for the quotient of $F(A)$ by $\approx$, with the inclusion map $q: F(A) \rightarrow F(A) // \approx$. The generators map into $F(A) // \approx$ is given by $q \circ \eta_A$, the identity element is $\hat{e} = q(e)$, and the multiplication operation $\bullet$ is lifted to $\hat{\bullet}$ in the quotient by congruence.

► **Proposition 28** (⚙️). *$(F(A) // \approx, \hat{e}, \hat{\bullet})$ is a commutative monoid.*

The monoid extension operation of $F(A)$, denoted $(-)^{\#}$, lifts to an extension operation for commutative monoids, for $F(A) // \approx$, which we denote by $(-)^{\hat{\#}}$.

► **Definition 29** (⚙️). *Given a commutative monoid $\mathfrak{X}$ and a map $f: A \rightarrow X$, we define the extension operation as follows: we first obtain $f^{\#}: \mathfrak{F}(A) \rightarrow \mathfrak{X}$ by the universal property of F , then lift it to $f^{\hat{\#}}: F(A) // \approx \rightarrow \mathfrak{X}$, by using that $\approx$ respects $(-)^{\#}$, as in Definition 27.*

► **Proposition 30** (⚙️ Universal property for $F(A) // \approx$). *$(F(A) // \approx, \eta_A: A \xrightarrow{\eta_A} \mathfrak{F}(A) \xrightarrow{q} F(A) // \approx)$ with the extension operation $(-)^{\hat{\#}}$ forms the free commutative monoid on A .*

4.2 Lists quotiented by permutation relations

A specific instance of this construction is List quotiented by a concrete permutation relation to get commutativity. Of course, there are many permutation relations in the literature, we consider a simple one which swaps any two adjacent elements anywhere in the middle of the list. This PList construction has also been considered in [20], who prove that PList is equivalent to the free commutative monoid (constructed as a HIT). We give a direct proof of its universal property using our generalisation.

► **Definition 31** (⚙️ Perm). *The inductive family Perm is generated by two constructors and defines a permutation relation on List:*

perm-refl: $\forall xs, \text{Perm } xs \ xs$

perm-swap: $\forall x, y, xs, ys, zs, \text{Perm } (xs \bullet x :: y :: ys) \ zs \rightarrow \text{Perm } (xs \bullet y :: x :: ys) \ zs$

where $\bullet$ is the usual list concatenation operation. PList(A) is the quotient of List(A) by Perm:

PList(A) := List(A) // Perm .

By § 4.1, to obtain that $\text{PList}(A)$ is the free commutative monoid on A , it suffices to prove the property of $\text{Perm}(A)$ being a permutation relation.

► **Proposition 32** (⚙️ Perm is a permutation relation). *The relation Perm is a permutation relation on $\text{List}(A)$.*

► **Proposition 33** (⚙️). *Let $\mathfrak{X}$ be a commutative monoid, and $f: A \rightarrow X$. For $x, y: A$ and $xs, ys: \text{List}(A)$, $f^\#(xs \bullet x :: y :: ys) = f^\#(xs \bullet y :: x :: ys)$. Hence, Perm respects $(-)^\#$.*

PList being a quotient of List makes it easy to lift functions and properties defined on List along the quotient map. The inductive nature of Perm makes it easy to define algorithms that inductively rearrange elements in a list, such as insertion sort. This property also makes it possible to normalize inductively constructed list elements in PList to their simplest forms, e.g. $q([x]) \bullet q([y, z])$ normalizes to $q([x, y, z])$ by definition, saving the efforts of defining auxiliary lemmas to prove their equality.

This inductive nature, however, makes it difficult to define efficient operations on PList . Consider a divide-and-conquer algorithm such as merge sort, which involves partitioning a PList of length $n + m$ into two smaller PList of length n and length m . We must iterate all n elements before we can make such a partition, thus making PList unintuitive to work with when we want to reason with operations that involve arbitrary partitioning.

4.3 Swap Lists

Informally, quotients are defined by generating all the points, then quotienting out into equivalence classes by the congruence relation. Alternately, HITs use generators (points) and higher generators (paths) (and higher higher generators and so on...). We can also define free commutative monoids using HITs where adjacent swaps generate all symmetries, for example swap-lists taken from [8], and in the Cubical library [29].

► **Definition 34** (⚙️ SList). *The higher inductive type $\text{SList}(A)$ is generated by:*

```

nil: SList(A)
- :: -: A → SList(A) → SList(A)
swap: ∀x, y, xs, x :: y :: xs = y :: x :: xs
trunc: ∀x, y, (p, q: x = y) → p = q

```

We refer the reader to [8] or the Cubical Agda library [29] for further details. Much like List , SList is inductively defined, therefore making it intuitive to reason with when defining inductive operations or proofs on SList , but difficult to reason with when defining operations that involve arbitrary partitioning, for reasons similar to those given in § 4.2.

4.4 Bag

Alternatively, we can also quotient the Array type by symmetries to get commutativity. This construction has been considered in [4] and [24], partially considered in [8], and in [20], who give a similar construction, where only the index function is quotiented, instead of the entire array. [14] also considered Bag as a setoid relation on List in an intensional MLTT setting. [20] prove that their version of Bag is the free commutative monoid by equivalence to the other HIT constructions. We give a direct proof of its universal property instead, using our general recipe.

► **Definition 35** (⚙ Bag). *Bags are defined as arrays quotiented by bag equivalence $\approx$:*

$$(n, f) \approx (m, g) := (\sigma: \text{Fin}_n \xrightarrow{\sim} \text{Fin}_m) \times (f = g \circ \sigma)$$

$$\text{Bag}(A) := \text{Array}(A) //_{\approx}$$

Note that by a pigeonhole argument, σ may only be constructed when $n = m$. In other words, we are quotienting by an automorphism on the indices, and its action on the elements. We have already shown Array to be the free monoid in § 3.2. By Proposition 30, to obtain that $\text{Bag}(A)$ is the free commutative monoid on A , it suffices to prove the property of $\approx$ being a permutation relation.

► **Proposition 36** (⚙ Bag equivalence is a permutation relation). *The relation $\approx$ is a permutation relation on $\text{Array}(A)$.*

► **Proposition 37** (⚙). *$\approx$ is congruent with respect to $\bullet$.*

Proof. Given $(n, f) \approx (m, g)$ by σ and $(u, p) \approx (v, q)$ by ϕ , we want to show $(n, f) \bullet (u, p) \approx (m, g) \bullet (v, q)$ by some τ . We construct τ as follows:

$$\tau := \text{Fin}_{n+u} \xrightarrow{\sim} \text{Fin}_n + \text{Fin}_u \xrightarrow{\sigma+\phi} \text{Fin}_m + \text{Fin}_v \xrightarrow{\sim} \text{Fin}_{m+v}$$

which operationally performs:

$$\begin{array}{c} \{0, 1, \dots, n-1, n, n+1, \dots, n+u-1\} \\ \downarrow \sigma+\phi \\ \{\sigma(0), \sigma(1), \dots, \sigma(n-1), \phi(0), \phi(1), \dots, \phi(u-1)\} \end{array} \quad \blacktriangleleft$$

► **Proposition 38** (⚙). *$\approx$ respects commutativity.*

Proof. We want to show for any arrays (n, f) and (m, g) , $(n, f) \bullet (m, g) \approx (m, g) \bullet (n, f)$ by some ϕ . We use formal combinators (see [9]) to define ϕ :

$$\phi := \text{Fin}_{n+m} \xrightarrow{\sim} \text{Fin}_n + \text{Fin}_m \xrightarrow{\text{swap}_+} \text{Fin}_m + \text{Fin}_n \xrightarrow{\sim} \text{Fin}_{m+n}$$

which operationally performs:

$$\begin{array}{c} \{0, 1, \dots, n-1, n, n+1, \dots, n+m-1\} \\ \downarrow \phi \\ \{n, n+1, \dots, n+m-1, 0, 1, \dots, n-1\} \end{array} \quad \blacktriangleleft$$

To show that $f^\#$ is invariant under permutation: for all $\phi: \text{Fin}_n \xrightarrow{\sim} \text{Fin}_n$, $f^\#(n, i) = f^\#(n, i \circ \phi)$, we need some formal combinators for *punching in* and *punching out* indices. These operations are borrowed from [26] and developed further in [9] for studying permutation codes.

► **Lemma 39** (⚙). *Given $\phi: \text{Fin}_{S(n)} \xrightarrow{\sim} \text{Fin}_{S(n)}$, there is a permutation $\tau: \text{Fin}_{S(n)} \xrightarrow{\sim} \text{Fin}_{S(n)}$ such that $\tau(0) = 0$, and $f^\#(S(n), i \circ \phi) = f^\#(S(n), i \circ \tau)$.*

Proof. Let k be $\phi^{-1}(0)$, and $k + j = S(n)$, we construct τ :

$$\begin{array}{ccc} \tau := \text{Fin}_{S(n)} \xrightarrow{\phi} \text{Fin}_{S(n)} \xrightarrow{\sim} \text{Fin}_{k+j} \xrightarrow{\sim} \text{Fin}_k + \text{Fin}_j \xrightarrow{\text{swap}_+} \text{Fin}_j + \text{Fin}_k \xrightarrow{\sim} \text{Fin}_{j+k} \xrightarrow{\sim} \text{Fin}_{S(n)} & & \\ \{0, 1, 2, \dots, k, k+1, k+2, \dots\} & & \{0, 1, 2, \dots, k, k+1, k+2, \dots\} \\ \downarrow \phi & & \downarrow \tau \\ \{x, y, z, \dots, 0, u, v, \dots\} & & \{0, u, v, \dots, x, y, z, \dots\} \end{array}$$

It is trivial to show $f^\#(S(n), i \circ \phi) = f^\#(S(n), i \circ \tau)$, since the only operation on indices in τ is swap_+ . It suffices to show $(S(n), i \circ \phi)$ can be decomposed into two arrays such that $(S(n), i \circ \phi) = (k, g) \bullet (j, h)$ for some g and h . Since the image of $f^\#$ is a commutative monoid, and $f^\#$ is a homomorphism, $f^\#((k, g) \bullet (j, h)) = f^\#(k, g) \bullet f^\#(j, h) = f^\#(j, h) \bullet f^\#(k, g) = f^\#((j, h) \bullet (k, g))$, thereby proving $f^\#(S(n), i \circ \phi) = f^\#(S(n), i \circ \tau)$. $\blacktriangleleft$

► **Lemma 40** (⚙️). *Given $\tau: \text{Fin}_{S(n)} \xrightarrow{\sim} \text{Fin}_{S(n)}$ where $\tau(0) = 0$, there is a $\psi: \text{Fin}_n \xrightarrow{\sim} \text{Fin}_n$ such that $\tau \circ S = S \circ \psi$.*

Proof. We construct ψ as $\psi(x) = \tau(S(x)) - 1$. Since τ maps only 0 to 0 by assumption, $\forall x. \tau(S(x)) > 0$, therefore the (-1) is well defined. This corresponds to the special case for $k = 0$ of the standard punch-in and punch-out operations from the Agda standard library [13], as used for Lehmer codes in [9].

$$\begin{array}{ccc} \{0, 1, 2, 3, \dots\} & & \{0, 1, 2, \dots\} \\ \downarrow \tau & & \downarrow \psi \\ \{0, x, y, z, \dots\} & & \{x-1, y-1, z-1, \dots\} \end{array} \quad \blacktriangleleft$$

► **Theorem 41** (⚙️ Permutation invariance). *For all $\phi: \text{Fin}_n \xrightarrow{\sim} \text{Fin}_n$, $f^\#(n, i) = f^\#(n, i \circ \phi)$.*

Proof. By induction on n .

- At $n = 0$, $f^\#(0, i) = f^\#(0, i \circ \phi) = e$.
- At $n = S(m)$,

$$\begin{aligned} & f^\#(S(m), i \circ \phi) \\ &= f^\#(S(m), i \circ \tau) && \text{by Lemma 39} \\ &= f(i(\tau(0))) \bullet f^\#(m, i \circ \tau \circ S) && \text{by definition of } (-)^\# \\ &= f(i(0)) \bullet f^\#(m, i \circ \tau \circ S) && \text{by construction of } \tau \\ &= f(i(0)) \bullet f^\#(m, i \circ S \circ \psi) && \text{by Lemma 40} \\ &= f(i(0)) \bullet f^\#(m, i \circ S) && \text{induction} \\ &= f^\#(S(m), i) && \text{by definition of } (-)^\# \end{aligned} \quad \blacktriangleleft$$

Unlike `PList` and `SList`, `Bag` and its underlying construction `Array` are not inductively defined, making it difficult to do induction on them. For example, in the proof of Proposition 26, both Lemmas 23 and 24 are needed to do induction on `Array`, as opposed to `List` and its quotients, where we can do induction simply by pattern matching. Much like `PList`, when defining functions on `Bag`, we need to show they respect $\approx$, i.e. $as \approx bs \rightarrow f(as) = f(bs)$. Notably, this is much more difficult than `PList` or `SList` – with `PList` and `SList` we only need to consider swapping adjacent elements, while with `Bag` we need to consider all possible permutations. For example, in the proof of Theorem 41, we need to first construct a τ which satisfies $\tau(0) = 0$ and prove $f^\#(n, i \circ \sigma) = f^\#(n, i \circ \tau)$ before we can apply induction.

5 Sorting Functions

We will now put to work the universal properties of our types of (ordered) lists and unordered lists, to define operations on them systematically, which are mathematically sound, and then reason about them. First, we explore definitions of various operations on both free monoids and free commutative monoids. By univalence (and the structure identity principle [2, 30]),

everything henceforth holds for any presentation of free monoids and free commutative monoids. We use $\mathcal{F}(A)$ to denote the free monoid or free commutative monoid on A , $\mathcal{L}(A)$ to exclusively denote the free monoid construction, and $\mathcal{M}(A)$ to exclusively denote the free commutative monoid construction.

For example `length` is a common operation defined inductively for `List`, but usually, properties about `length`, such as, $\text{length}(xs \bullet ys) = \text{length}(xs) + \text{length}(ys)$, are proven separately after defining it. In our framework of free algebras, where the $(-)^{\#}$ operation is a correct-by-construction homomorphism, we can define operations like `length` directly by universal extension, which also gives us a proof that they are homomorphisms for free. A further application of the universal property is to prove that two different types (such as lists and arrays) are equal, by showing they both satisfy the same universal property as in Proposition 11, which is desirable especially when proving a direct equivalence between the two types turns out to be a difficult exercise in combinatorics.

5.1 Prelude

Any presentation of free monoids or free commutative monoids has a $\text{length}: \mathcal{F}(A) \rightarrow \mathbb{N}$ function, where $\mathbb{N}$ carries the additive monoid structure $(0, +)$, which is also a commutative monoid structure since addition is commutative.

► **Definition 42** (⚙️ `length`). *The length homomorphism is defined as $(\lambda x. 1)^{\#}: \mathcal{F}(A) \rightarrow \mathbb{N}$.*

Further, any presentation of free monoids or free commutative monoids has an element membership predicate $- \in -: A \rightarrow \mathcal{F}(A) \rightarrow \mathbf{hProp}$, for any set A . Here, we use the fact that $\mathbf{hProp}$ forms a (commutative) monoid under disjunction and falsehood $(\perp, \vee)$.

► **Definition 43** (⚙️ `Membership  $\in$`). *The membership predicate on a set A for any element $x: A$ is $x \in - := \mathfrak{L}_A(x)^{\#}: \mathcal{F}(A) \rightarrow \mathbf{hProp}$, where we define $\mathfrak{L}_A(x) := \lambda y. x = y: A \rightarrow \mathbf{hProp}$.*

$\mathfrak{L}$ is formally the Yoneda map under the “types are groupoids” correspondence, where $x: A$ is being sent to its representable in the Hom-groupoid (formed by the identity type), of type $\mathbf{hProp}$. Note that the proofs of (commutative) monoid laws for $\mathbf{hProp}$ use equality, which requires the use of univalence (or at least, propositional extensionality). By construction, this membership predicate satisfies its homomorphic properties, which are colloquially the properties of inductively defined de Bruijn indices.

We note that $\mathbf{hProp}$ is actually one type level higher than A . To make the type levels explicit, A is of type level ℓ , and since $\mathbf{hProp}_{\ell}$ is the type of all types $X: \mathbf{Set}_{\ell}$ that are mere propositions, $\mathbf{hProp}_{\ell}$ has type level $\ell + 1$. We do not assume any propositional resizing axioms [31], and instead use Agda’s universe polymorphism in our formalisation to accommodate this.

Any presentation of free (commutative) monoids $\mathcal{F}(A)$ also supports the `Any` and `All` predicates, which allow lifting a predicate $A \rightarrow \mathbf{hProp}$ (on A), to *any* or *all* elements of $xs: \mathcal{F}(A)$, respectively. We note that $\mathbf{hProp}$ forms a (commutative) monoid in two different ways: $(\perp, \vee)$ and $(\top, \wedge)$ (disjunction and conjunction), which are the two different ways of getting `Any` and `All`, respectively, by extension.

► **Definition 44** (`Any` and `All`).

$$\text{Any}(P) := P^{\#}: \mathcal{F}(A) \rightarrow (\mathbf{hProp}, \perp, \vee) \quad \text{All}(P) := P^{\#}: \mathcal{F}(A) \rightarrow (\mathbf{hProp}, \top, \wedge)$$

Note that Cubical Agda has problems with indexing over HITs [28, 3, § 8], hence it is preferable to program with our universal properties, such as when defining `Any` and `All`, because the indexed-inductive definitions of these predicates get stuck on `transp` terms.

There is a **head** function on lists, which is a function that returns the first element of a non-empty list. Formally, this is a monoid homomorphism from $\mathcal{L}(A)$ to $1 + A$.

► **Definition 45** (⚙️ **head**). *The head homomorphism is defined as $\text{head} := \text{inr}^\# : \mathcal{L}(A) \rightarrow 1 + A$, where the monoid structure on $1 + A$ has unit $e := \text{inl}(\star) : 1 + A$, and multiplication picks the leftmost element that is defined.*

$$\begin{aligned} \text{inl}(\star) \oplus b &:= b \\ \text{inr}(a) \oplus b &:= \text{inr}(a) \end{aligned}$$

This monoid operation $\oplus$ is not commutative, since swapping the input arguments to $\oplus$ would return the leftmost or rightmost element. To make it commutative would require a canonical way to pick between a choice of two elements – this leads us to the next section.

5.2 Total orders

First, we recall the axioms of a total order or linear order $\leq$ on a set A .

► **Definition 46** (⚙️ **Total order**). *A total order on a set A is a relation $\leq : A \rightarrow A \rightarrow \text{hProp}$ that satisfies:*

- *reflexivity: $x \leq x$,*
- *transitivity: if $x \leq y$ and $y \leq z$, then $x \leq z$,*
- *antisymmetry: if $x \leq y$ and $y \leq x$, then $x = y$,*
- *totality: $\forall x, y$, either $x \leq y$ or $y \leq x$.*

A decidable total order requires the $\leq$ relation to be decidable:

- *decidable totality: $\forall x, y$, we have $x \leq y + \neg(x \leq y)$.*

Note that *either-or* means a (truncated) logical disjunction. In the context of this paper, we want to make a distinction between “decidable total order” and “total order”. The decidability axiom strengthens the totality axiom, where we have either $x \leq y$ or $y \leq x$ merely as a proposition, but decidability allows us to produce a witness if $x \leq y$ is true.

► **Proposition 47** (⚙️). *In a decidable total order, it holds that $\forall x, y, (x \leq y) + (y \leq x)$. Further, this makes A discrete, that is $\forall x, y, (x = y) + (x \neq y)$.*

Proof. We decide if $x \leq y$ and $y \leq x$, and then by cases:

- if $x \leq y$ and $y \leq x$: by antisymmetry, $x = y$.
- if $\neg(x \leq y)$ and $y \leq x$: assuming $x = y$ leads to a contradiction, hence $x \neq y$.
- if $x \leq y$ and $\neg(y \leq x)$: similar to the previous case.
- if $\neg(x \leq y)$ and $\neg(y \leq x)$: by totality, either $x \leq y$ or $y \leq x$, which leads to a contradiction. ◀

We also recall the axioms of a *strict* total order $<$ on A .

► **Definition 48** (⚙️ **Strict total order**). *A strict total order on a set A is a relation $< : A \rightarrow A \rightarrow \text{hProp}$ that satisfies:*

- *irreflexivity: $\neg(x < x)$,*
- *transitivity: if $x < y$ and $y < z$, then $x < z$,*
- *asymmetry: if $x < y$, then $\neg(y < x)$,*
- *cotransitivity: $\forall x, y, z$, if $x < z$, then either $x < y$ or $y < z$.*
- *connectedness: $\forall x, y$, if $\neg(x < y)$ and $\neg(y < x)$, then $x = y$.*

A decidable strict total order requires the $<$ relation to be decidable:

- *decidability: $\forall x, y$, we have $x < y + \neg(x < y)$.*

► **Proposition 49** (⚙️). *In a decidable strict total order, it holds that $\forall x, y, (x < y) + (y < x)$. Further, this makes A discrete, that is $\forall x, y, (x = y) + (x \neq y)$.*

Proof. We decide if $x < y$ and $y < x$, and by cases:

- if $\neg(x < y)$ and $\neg(y < x)$: by connectedness, $x = y$.
- if $x < y$ or $y < x$: by irreflexivity, $x \neq y$. ◀

We can show that giving a decidable *strict* total order structure on A is equivalent to giving a decidable total order structure on A . In practice, however, this equivalence is thoroughly awful to work with in a proof assistant, leading to horrible goals when formalising proofs that shift between these representations.

► **Proposition 50** (⚙️). *The set of decidable strict total orders on A and the set of decidable total orders on A are equivalent.*

Proof sketch. Given a decidable total order $\leq$ on A , by Proposition 47, we map it to a decidable strict total order $<$ on A by $x < y := (x \leq y) \times (x \neq y)$. Vice versa, by Proposition 49, given a decidable strict total order $<$ on A , we map it to a decidable total order $\leq$ on A by $x \leq y := (x < y) + (x = y)$. These maps are checked to be inverses of each other. ◀

An equivalent way of defining a total order is using a binary meet operation, without starting from an ordering relation.

► **Definition 51** (⚙️ Meet semi-lattice). *A meet semi-lattice is a set A with a binary operation $\sqcap : A \rightarrow A \rightarrow A$ that is:*

- *idempotent:* $x \sqcap x = x$,
- *associative:* $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$,
- *commutative:* $x \sqcap y = y \sqcap x$.

A total meet semi-lattice further satisfies:

- *total:* $\forall x, y$, either $x \sqcap y = x$ or $x \sqcap y = y$.

A decidable total meet semi-lattice strengthens this to:

- *decidable totality:* $\forall x, y, (x \sqcap y = x) + (x \sqcap y = y)$.

► **Proposition 52** (⚙️). *If A has decidable equality, all total meet semi-lattice structures on A are decidable.*

Proof. We decide if $x \sqcap y = x$ and $x \sqcap y = y$ to compute whether $x \sqcap y = x$ or $x \sqcap y = y$. By totality, one of them must hold. ◀

► **Proposition 53** (⚙️). *A total order structure on a set A is equivalent to a total meet semi-lattice structure on A . Further, a decidable total order on A induces a decidable total meet semi-lattice structure on A .*

Proof sketch. Given a (mere) total order $\leq$ on a set A , we define $x \sqcap y := \text{if } x \leq y \text{ then } x \text{ else } y$. Crucially, this map is *locally-constant*, allowing us to eliminate from an $\mathbf{hProp}$ to an $\mathbf{hSet}$. Meets satisfy the universal property of products, that is, $c \leq a \sqcap b \Leftrightarrow c \leq a \wedge c \leq b$, and the axioms follow by calculation using $\mathcal{J}$ -arguments. Conversely, given a meet semi-lattice, we define $x \leq y := x \sqcap y = x$, which defines an $\mathbf{hProp}$ -valued total ordering relation. If the total order is decidable, by Proposition 47 and Proposition 52, the mapped meet operation is decidable. The converse does not hold however, as a decidable total meet semi-lattice structure does not imply discreteness of A . ◀

$$\mathcal{L}(A) \begin{array}{c} \xrightarrow{\mathbf{q}} \\ \xleftarrow{\mathbf{s}} \end{array} \mathcal{M}(A)$$

■ **Figure 1** Relationship of $\mathcal{L}(A)$ and $\mathcal{M}(A)$.

Finally, tying this back to Definition 45, we have the following for free commutative monoids.

► **Definition 54** (⚙️ head). *Assume a total order $\leq$ on a set A . We define a commutative monoid structure on $1 + A$, with unit $e := \text{inl}(\star) : 1 + A$, and multiplication defined as:*

$$\begin{aligned} \text{inl}(\star) \oplus b &:= b \\ \text{inr}(a) \oplus \text{inl}(\star) &:= \text{inr}(a) \\ \text{inr}(a) \oplus \text{inr}(b) &:= \text{inr}(a \sqcap b) . \end{aligned}$$

This gives a homomorphism $\text{head} := \text{inr}^\# : \mathcal{M}(A) \rightarrow 1 + A$, which picks out the least element of the free commutative monoid.

5.3 Sorting functions

The free commutative monoid is also a monoid, hence, there is a canonical monoid homomorphism $\mathbf{q} : \mathcal{L}(A) \rightarrow \mathcal{M}(A)$, which is given by $\eta_A^\#$, the extension of the unit $\eta_A : A \rightarrow \mathcal{M}(A)$. Since $\mathcal{M}(A)$ is (upto equivalence), a quotient of $\mathcal{L}(A)$ by symmetries (or a permutation relation), it is a surjection (in particular, a regular epimorphism in **Set** as constructed in type theory), the canonical inclusion into the quotient. Concretely, $\mathbf{q}$ simply includes the elements of $\mathcal{L}(A)$ into equivalence classes of lists in $\mathcal{M}(A)$, which “forgets” the order that was imposed by the indexing of the list.

Classically, assuming the Axiom of Choice would allow us to construct a section (right-inverse, in **Set**) to the surjection $\mathbf{q}$, that is, a function $\mathbf{s} : \mathcal{M}(A) \rightarrow \mathcal{L}(A)$ such that $\forall x. \mathbf{q}(\mathbf{s}(x)) = x$. Or in informal terms, given the surjective inclusion into the quotient, a section (uniformly) picks out a canonical representative for each equivalence class. The core question we want to study is the existence of $\mathbf{s}$ in a constructive setting, or equivalently, whether the order factored out by the symmetry quotient can be constructively recovered. Viewing the quotienting relation as a permutation relation (from § 4.1), a section $\mathbf{s}$ to $\mathbf{q}$ has to pick out canonical representatives of equivalence classes generated by permutations. Using **SList** as an example, $\mathbf{s}(x :: y :: xs) = \mathbf{s}(y :: x :: xs)$ for any $x, y : A$ and $xs : \mathbf{SList}(A)$, by **swap**. Since $\forall xs. \mathbf{q}(\mathbf{s}(xs)) = xs$, $\mathbf{s}$ must preserve all the elements of xs . It cannot be a trivial function such as $\lambda xs. []$ – it must produce a permutation of the elements of xs . But to place these elements side-by-side in the list, $\mathbf{s}$ must somehow impose an order on A (invariant under permutation), turning unordered lists of A into ordered lists of A . Axiom of Choice (AC) giving us a section $\mathbf{s}$ to $\mathbf{q}$ “for free” is analagous to how AC implies the well-ordering principle, which states every set can be well-ordered. Thus, if AC was assumed, we could easily recover an order on A from the section $\mathbf{s}$. Instead we study how to constructively define such a section, and in fact, that is exactly the extensional view of a sorting algorithm, and the implications of its existence is that A can be ordered, or sorted.

5.3.1 Section from Order

► **Proposition 55** (⚙️). *Assume a decidable total order on A . There is a sort function $\mathbf{s} : \mathcal{M}(A) \rightarrow \mathcal{L}(A)$ which constructs a section to $\mathbf{q} : \mathcal{L}(A) \twoheadrightarrow \mathcal{M}(A)$*

Proof sketch. We may construct such a sort function by implementing any sorting algorithm. In our formalisation we chose insertion sort, because it can be defined easily using the inductive structure of $\text{SList}(A)$ and $\text{List}(A)$. To implement other sorting algorithms like mergesort, other representations such as **Bag** and **Array** would be preferable, as explained in § 4.4. To see how this proposition holds: $\mathbf{q}(\mathbf{s}(xs))$ first orders an unordered list xs by $\mathbf{s}$, then forgets the order again by $\mathbf{q}$ – imposing and then forgetting an order on xs simply *permutes* its elements, which proves $\mathbf{q} \circ \mathbf{s} = \text{id}$. ◀

5.3.2 Order from Section

The previous section allowed us to construct a section, but an arbitrary section may not be a sorting function. To show a section is indeed a sort function, we need to show the section imposes some total order on A which it sorts by. Indeed, just by the virtue of $\mathbf{s}$ being a section, we can *almost* construct a total-order on the carrier set.

► **Definition 56** (⚙️ *least*). Given a section $\mathbf{s}$, we define a relation $\leq_{\mathbf{s}}$ parametrised by $\mathbf{s}$:

$$x \leq_{\mathbf{s}} y := \text{head}(\mathbf{s}(\wr x, y)) = \text{inr}(x) .$$

Recall that the head homomorphism $\text{head} := \text{inr}^\# : \mathcal{L}(A) \rightarrow 1 + A$ (as in Definition 45) is defined by equipping $1 + A$ with a monoid structure with unit $e := \text{inl}(\star)$ and multiplication defined as $\text{inl}(\star) \oplus b := b$ and $\text{inr}(a) \oplus b := \text{inr}(a)$, which picks out the first (leftmost) element of a list. Here, we take the two-element bag $\wr x, y$, “sort” it by $\mathbf{s}$, and test if the **head** element is x . Note, this is equivalent to $x \leq_{\mathbf{s}} y := \mathbf{s} \wr x, y = [x, y]$, because $\mathbf{s}$ preserves length, and the second element is forced to be y .

► **Proposition 57** (⚙️). $\leq_{\mathbf{s}}$ is reflexive, antisymmetric, and total.

Proof. For all x , $\text{least}(\wr x, x)$ must be $\text{inr}(x)$, therefore $x \leq_{\mathbf{s}} x$, giving reflexivity. For all x and y , given $x \leq_{\mathbf{s}} y$ and $y \leq_{\mathbf{s}} x$, we have $\text{least}(\wr x, y) = \text{inr}(x)$ and $\text{least}(\wr y, x) = \text{inr}(y)$. Since $\wr x, y = \wr y, x$, $\text{least}(\wr x, y) = \text{least}(\wr y, x)$, therefore we have $x = y$, giving antisymmetry. For all x and y , $\text{least}(\wr x, y)$ is merely either $\text{inr}(x)$ or $\text{inr}(y)$, therefore we have merely either $x \leq_{\mathbf{s}} y$ or $y \leq_{\mathbf{s}} x$, giving totality. ◀

A crucial observation is that $\mathbf{s}$ correctly orders 2-element bags, but it does not necessarily sort bags with 3 or more elements.

► **Proposition 58** (⚙️). $\leq_{\mathbf{s}}$ is not necessarily transitive.

Proof. A sorting section $\mathbf{s}$ would indeed give a $\leq_{\mathbf{s}}$ that satisfies transitivity. However, we can turn a sorting section into a non-sorting misbehaving section that gives a non-transitive $\leq_{\mathbf{s}}$. Take the sorting section $\text{sort} : \text{SList}(\mathbb{N}) \rightarrow \text{List}(\mathbb{N})$ which sorts $\text{SList}(\mathbb{N})$ ascendingly, we can use sort to construct a counterexample $\mathbf{s}$ as follows:

$$\mathbf{s}(xs) := \begin{cases} [3, 1] & \text{if } xs = \wr 1, 3 \\ \text{sort}(xs) & \text{otherwise} \end{cases}$$

Indeed, we have $1 \leq_{\mathbf{s}} 2$ and $2 \leq_{\mathbf{s}} 3$ as expected, but $\text{least}(\wr 1, 3) = \text{inr}(3)$, therefore $1 \not\leq_{\mathbf{s}} 3$, thus violating transitivity. ◀

To make sure that $\mathbf{s}$ is a well-behaved section that recovers a total order on A , we will enforce additional constraints on the *image* of $\mathbf{s}$.

► **Definition 59** (⚙️ $- \in \text{im}(\mathfrak{s})$). The fiber of $\mathfrak{s}$ at a point in the codomain $xs: \mathcal{L}(A)$ is given by $\text{fib}_{\mathfrak{s}}(xs) := (ys: \mathcal{M}(A)) \times (\mathfrak{s}(ys) = xs)$. The image of $\mathfrak{s}$ is given by $\text{im}(\mathfrak{s}) := (xs: \mathcal{L}(A)) \times \|\text{fib}_{\mathfrak{s}}(xs)\|_{-1}$. Simplifying, we say that $xs: \mathcal{L}(A)$ is “in the image of $\mathfrak{s}$ ”, or, $xs \in \text{im}(\mathfrak{s})$, if there merely exists a $ys: \mathcal{M}(A)$ such that $\mathfrak{s}(ys) = xs$.

If $\mathfrak{s}$ were a sort function, the image of $\mathfrak{s}$ would be the set of $\mathfrak{s}$ –“sorted” lists, therefore $xs \in \text{im}(\mathfrak{s})$ would imply xs is a correctly $\mathfrak{s}$ –“sorted” list. First, we note that the 2-element case is correct.

► **Proposition 60** (⚙️). $x \leq_{\mathfrak{s}} y \iff [x, y] \in \text{im}(\mathfrak{s})$.

Then, we state the first axiom on $\mathfrak{s}$.

► **Definition 61** (⚙️ isHeadLeast). A section $\mathfrak{s}$ satisfies isHeadLeast iff for all x, y, xs :

$$y \in x :: xs \wedge x :: xs \in \text{im}(\mathfrak{s}) \rightarrow [x, y] \in \text{im}(\mathfrak{s}) .$$

There are two different membership symbols in this axiom, where the first membership is list membership from Definition 43, and the second membership is image membership from Definition 59. The $\in$ symbol is intentionally overloaded to make the axiom look like a logical “cut” rule, that pushes the (least) x element to the head of the list. Informally, the head of an $\mathfrak{s}$ –“sorted” list is always the least element of the list.

► **Proposition 62** (⚙️). If A has a total order $\leq$, the insertion sort function defined using $\leq$ satisfies isHeadLeast .

► **Proposition 63** (⚙️). If $\mathfrak{s}$ satisfies isHeadLeast , $\leq_{\mathfrak{s}}$ is transitive.

Proof. Given $x \leq_{\mathfrak{s}} y$ and $y \leq_{\mathfrak{s}} z$, we want to show $x \leq_{\mathfrak{s}} z$. Consider the 3-element bag $\{x, y, z\}: \mathcal{M}(A)$. Let u be $\text{least}(\{x, y, z\})$, by Definition 61 and Proposition 60, we have $u \leq_{\mathfrak{s}} x \wedge u \leq_{\mathfrak{s}} y \wedge u \leq_{\mathfrak{s}} z$. Since $u \in \{x, y, z\}$, u must be one of the elements. If $u = x$ we have $x \leq_{\mathfrak{s}} z$. If $u = y$ we have $y \leq_{\mathfrak{s}} x$, and since $x \leq_{\mathfrak{s}} y$ and $y \leq_{\mathfrak{s}} z$ by assumption, we have $x = y$ by antisymmetry, and then we have $x \leq_{\mathfrak{s}} z$ by substitution. Finally, if $u = z$, we have $z \leq_{\mathfrak{s}} y$, and since $y \leq_{\mathfrak{s}} z$ and $x \leq_{\mathfrak{s}} y$ by assumption, we have $z = y$ by antisymmetry, and then we have $x \leq_{\mathfrak{s}} z$ by substitution. ◀

5.3.3 Embedding orders into sections

Following from Propositions 57 and 63, and Proposition 62, we have shown that a section $\mathfrak{s}$ that satisfies isHeadLeast produces a total order $x \leq_{\mathfrak{s}} y := \text{least}(\{x, y\}) = \text{inr}(x)$, and a total order $\leq$ on the carrier set produces a section satisfying isHeadLeast , constructed by sorting with $\leq$. This constitutes an embedding of decidable total orders into sections satisfying isHeadLeast .

► **Proposition 64** (⚙️). Assume A has a decidable total order $\leq$, we can construct a section $\mathfrak{s}$ that satisfies isHeadLeast , such that $\leq_{\mathfrak{s}}$ constructed from $\mathfrak{s}$ is equivalent to $\leq$.

Proof. By the insertion sort algorithm parameterized by $\leq$, it holds that $[x, y] \in \text{im}(\mathfrak{s})$ iff $x \leq y$. By Proposition 60, we have $x \leq_{\mathfrak{s}} y$ iff $x \leq y$. We now have a total order $x \leq_{\mathfrak{s}} y$ that is equivalent to $x \leq y$. ◀

5.3.4 Equivalence of order and sections

We want to improve the embedding to an isomorphism, and it remains to show that we can turn a section satisfying `isHeadLeast` to a total order $\leq_s$, then reconstruct the *same* section from $\leq_s$. Unfortunately, `isHeadLeast` is not enough to guarantee this.

► **Proposition 65** (⚙️). *There is no equivalence between sections satisfying the `isHeadLeast` axiom and total orders on A .*

Proof. We consider the set $\mathbb{N}$ of natural numbers, and construct a $\mathfrak{s}$ that satisfies `isHeadLeast` but is not a sort function. Let the correct insertion sort function be `sort`: $\mathcal{M}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N})$. We define `reverseTail` which reverses only the tail of a list, and a section $\mathfrak{s}$ that sorts correctly using `sort` then reverses the tail:

$$\begin{aligned} \text{reverseTail}([]) &:= [] \\ \text{reverseTail}(x :: xs) &:= x :: \text{reverse}(xs) \\ \mathfrak{s}(xs) &:= \text{reverseTail}(\text{sort}(xs)) \end{aligned}$$

For example:

$$\begin{aligned} \mathfrak{s}(\uparrow 2, 3, 1, 4 \downarrow) &= [1, 4, 3, 2] \\ \mathfrak{s}(\uparrow 2, 3, 1 \downarrow) &= [1, 3, 2] \\ \mathfrak{s}(\uparrow 2, 3 \downarrow) &= [2, 3] \end{aligned}$$

Note that both `sort` and $\mathfrak{s}$ satisfy `isHeadLeast`, but $\mathfrak{s}$ only sorts 2-element lists correctly, and $\mathfrak{s} \neq \text{sort}$. By Proposition 64 we can use both `sort` and $\mathfrak{s}$ to reconstruct the same $\leq$ on $\mathbb{N}$. Hence, two different sections satisfying `isHeadLeast` map to the same total order. ◀

Therefore, we need to introduce a second axiom of sorting.

► **Definition 66** (⚙️ `isTailSorted`). *A section $\mathfrak{s}$ satisfies `isTailSorted` iff for all x, xs ,*

$$x :: xs \in \text{im}(\mathfrak{s}) \rightarrow xs \in \text{im}(\mathfrak{s})$$

This says that $\mathfrak{s}$ -“sorted” lists are downwards-closed under `cons`-ing, that is, the tail of an $\mathfrak{s}$ -“sorted” list is also $\mathfrak{s}$ -“sorted”. To prove the correctness of our axioms, first we need to show that a section $\mathfrak{s}$ satisfying `isHeadLeast` and `isTailSorted` is equal to insertion sort parameterized by the $\leq_s$ constructed from $\mathfrak{s}$. In fact, the axioms we have introduced are equivalent to the standard inductive characterisation of sorted lists, found in textbooks, such as in [5].

► **Definition 67** (⚙️ `isSorted`). *Given a total order $\leq$ on A , the predicate `isSorted≤` on $\mathcal{L}(A)$ is generated by the following constructors:*

$$\begin{aligned} \text{sorted-nil} &: \text{isSorted}_{\leq}([]) \\ \text{sorted-}\eta &: \forall x. \text{isSorted}_{\leq}([x]) \\ \text{sorted-cons} &: \forall x, y, zs. x \leq y \rightarrow \text{isSorted}_{\leq}(y :: zs) \rightarrow \text{isSorted}_{\leq}(x :: y :: zs) \end{aligned}$$

Note that `isSorted≤`(xs) is a proposition for every xs , and forces the list xs to be permuted in a unique way. This predicate is equivalent to the standard `Sorted` relation in the Agda standard library [13], which is defined pairwise using the more general `Linked` relation (i.e., `Sorted(xs) := Linked(- ≤ -, xs)`). Indeed, many of these definitions are traditional constructions that are completely independent of univalence or cubical features.

► **Lemma 68** (⚙️). *Given a total order $\leq$, for any lists $xs, ys: \mathcal{L}(A)$, if $q(xs) = q(ys)$, $\text{isSorted}_{\leq}(xs)$, and $\text{isSorted}_{\leq}(ys)$, then $xs = ys$.*

Insertion sort by $\leq$ always produces lists that satisfy $\text{isSorted}_{\leq}$. Sections that also produce lists satisfying $\text{isSorted}_{\leq}$ are equal to insertion sort by function extensionality.

► **Proposition 69** (⚙️). *Given a total order $\leq$, if a section $\mathfrak{s}$ always produces a sorted list, that is, $\forall xs. \text{isSorted}_{\leq}(\mathfrak{s}(xs))$, $\mathfrak{s}$ is equal to insertion sort by $\leq$.*

Finally, this gives us the correctness property of our axioms.

► **Proposition 70** (⚙️). *Given a section $\mathfrak{s}$ that satisfies isHeadLeast and isTailSorted , and $\leq_{\mathfrak{s}}$ the order derived from $\mathfrak{s}$, then for all $xs: \mathcal{M}(A)$, it holds that $\text{isSorted}_{\leq_{\mathfrak{s}}}(\mathfrak{s}(xs))$. Equivalently, for all lists $xs: \mathcal{L}(A)$, it holds that $xs \in \text{im}(\mathfrak{s})$ iff $\text{isSorted}_{\leq_{\mathfrak{s}}}(xs)$.*

Proof. We inspect the length of $xs: \mathcal{M}(A)$. For lengths 0 and 1, this holds trivially. Otherwise, we proceed by induction: given a $xs: \mathcal{M}(A)$ of length ≥ 2 , let $\mathfrak{s}(xs) = x :: y :: ys$. We need to show $x \leq_{\mathfrak{s}} y \wedge \text{isSorted}_{\leq_{\mathfrak{s}}}(y :: ys)$ to construct $\text{isSorted}_{\leq_{\mathfrak{s}}}(x :: y :: ys)$. By isHeadLeast , we have $x \leq_{\mathfrak{s}} y$, and by isTailSorted , we inductively prove $\text{isSorted}_{\leq_{\mathfrak{s}}}(y :: ys)$. ◀

► **Lemma 71** (⚙️). *Given a decidable total order $\leq$ on A , we can construct a section $t_{\leq}$ satisfying the axioms isHeadLeast and isTailSorted , such that, for the order $\leq_{\mathfrak{s}}$ derived from $\mathfrak{s}$, we have $t_{\leq} = \mathfrak{s}$.*

Proof. From $\mathfrak{s}$ we can construct a decidable total order $\leq_{\mathfrak{s}}$ since $\mathfrak{s}$ satisfies isHeadLeast and A has decidable equality by assumption. We construct $t_{\leq_{\mathfrak{s}}}$ as insertion sort parameterized by $\leq_{\mathfrak{s}}$ constructed from $\mathfrak{s}$. By Proposition 69 and Proposition 70, $s = t_{\leq_{\mathfrak{s}}}$. ◀

We can now state and prove our main theorem.

► **Definition 72** (⚙️ Sorting function). *A sorting function is a section $\mathfrak{s}: \mathcal{M}(A) \rightarrow \mathcal{L}(A)$ to the canonical surjection $q: \mathcal{L}(A) \twoheadrightarrow \mathcal{M}(A)$ satisfying two axioms:*

- $\text{isHeadLeast}: \forall x, y, xs, y \in x :: xs \wedge x :: xs \in \text{im}(\mathfrak{s}) \rightarrow [x, y] \in \text{im}(\mathfrak{s})$,
- $\text{isTailSorted}: \forall x, xs, x :: xs \in \text{im}(\mathfrak{s}) \rightarrow xs \in \text{im}(\mathfrak{s})$.

► **Theorem 73** (⚙️). *Let $\text{DecTotOrd}(A)$ be the set of decidable total orders on A , $\text{Sort}(A)$ be the set of sorting functions with carrier set A , and $\text{Discrete}(A)$ be a predicate which states A has decidable equality. There is a map $o2s: \text{DecTotOrd}(A) \rightarrow \text{Sort}(A) \times \text{Discrete}(A)$, which is an equivalence.*

Proof. $o2s$ is constructed by parameterizing insertion sort with $\leq$. By Proposition 47, A is Discrete . The inverse $s2o(s)$ is constructed by Definition 56, which produces a total order by Propositions 57 and 63, and a decidable total order by $\text{Discrete}(A)$. By Proposition 64 we have $s2o \circ o2s = \text{id}$, and by Lemma 71 we have $o2s \circ s2o = \text{id}$, giving an isomorphism, hence an equivalence. ◀

► **Corollary 74** (⚙️). *Let $\text{DecSTotOrd}(A)$ be the set of decidable strict total orders on A , There is a map $t2s: \text{DecSTotOrd}(A) \rightarrow \text{Sort}(A) \times \text{Discrete}(A)$, which is an equivalence.*

Proof. By Proposition 47, we have an equivalence $\text{DecSTotOrd}(A) \simeq \text{DecTotOrd}(A)$, and the result follows from Theorem 73 by transitivity of equivalences. ◀

► **Corollary 75** (⚙️). *Assume A has decidable equality. Let $\text{TMSLat}(A)$ be the set of total meet semilattices on A , There is a map $l2s: \text{TMSLat}(A) \rightarrow \text{Sort}(A)$, which is an equivalence.*

Proof. By assumption and Proposition 52, every total meet semilattice on A is decidable. Thus, by Proposition 53, we have an equivalence $\text{TMSLat}(A) \simeq \text{DecTotOrd}(A)$, and the result follows from Theorem 73 by transitivity of equivalences. $\blacktriangleleft$

The sorting axioms we have come up with are abstract properties of functions. As a sanity check, we can verify that the colloquial correctness specification of a sorting function (starting from a total order) matches our axioms. We use the correctness criterion developed in [3].

► **Proposition 76 (⚙️).** *Assume a decidable total order $\leq$ on A . A sorting algorithm is a map $\text{sort}: \mathcal{L}(A) \rightarrow \mathcal{OL}(A)$, that turns lists into ordered lists, where $\mathcal{OL}(A)$ is defined as $(xs: \mathcal{L}(A)) \times \text{isSorted}_{\leq}(xs)$, such that:*

$$\begin{array}{ccc} \mathcal{L}(A) & \xrightarrow{\text{sort}} & \mathcal{OL}(A) \\ & \searrow q & \swarrow q \circ \pi_1 \\ & \mathcal{M}(A) & \end{array}$$

Sorting functions (as in Definition 72) give sorting algorithms.

Proof. We construct our section $\mathfrak{s}: \mathcal{M}(A) \rightarrow \mathcal{L}(A)$, and define $\text{sort} := \mathfrak{s} \circ q$, which produces ordered lists by Proposition 70. $\blacktriangleleft$

6 Discussion

We conclude by discussing some high-level observations, related work, and future directions.

Formalisation. The paper uses informal type theoretic language, and is accessible without understanding any details of the formalisation. However, the formalisation is done in Cubical Agda, which has a few differences and shortcomings, that cause certain details to be more verbose than they are in informal type theory.

For simplicity we omitted type levels in the paper, but our formalisation has many verbose uses of universe levels due to Agda’s universe polymorphism. Similarly, h-levels were restricted to sets in the paper, but the formalisation is parameterised in many places for any h-level (to facilitate future generalisations). The free algebra framework currently only works with sets. Due to issues of regularity, certain computations only hold propositionally, and the formalisation requires proving auxiliary β and η computation rules in some places.

Universal Algebra. One of the contributions of our work is a rudimentary framework for universal algebra in a more categorical style, which lends itself to an elegant formalisation in type theory. This framework may be improved and generalised from sets to groupoids, using a system of coherences on top of a system of equations, to talk about 2-algebraic structures on groupoids, such as monoidal groupoids, symmetric monoidal groupoids, and so on.

Free commutative monoids. The construction of finite multisets and free commutative monoids in type theory has a long history, and various authors have different approaches to it. We refer the reader to the discussions in [8, 20] for a detailed survey of these constructions. Our work, in particular, was motivated by the colloquial observation that: “there is no way to represent free commutative monoids using inductive types”. From the categorical point of view, note that the free monoid functor on Set is polynomial, but the free commutative monoid endofunctor is not polynomial, since it only weakly preserves pullbacks. Various

authors have given clever encodings of free commutative monoids using inductive types by adding assumptions on the carrier set – in particular, the assumption of total ordering on the carrier set leads to the construction of “fresh-lists”, by [22], which forces the canonical *sorted* ordering on the elements of the finite multiset.

It is worth noting that in programming practice, it is usually the case that all user-defined types have some sort of total order enforced on them, either because they’re finite, or because they can be canonically enumerated. Under these assumptions, the construction of fresh lists is a very reasonable way to represent free commutative monoids, or finite multisets.

The free monoid and free commutative monoid constructions can be categorified to free monoidal categories and free symmetric monoidal categories, respectively. In type theory, these types can be groupoidified to free monoidal groupoids and free symmetric monoidal groupoids, using the groupoid structure of identity types. This is a natural next step, and its connections to assumptions about total orders on the type of objects would be an important direction to explore.

Correctness of Sorting. Sorting is a classic problem in computer science – the programming point of view of sorting and its correctness has been studied by various authors. The simplest view of sorting is a function $\text{sort}: \mathcal{L}(\mathbb{N}) \rightarrow \mathcal{L}(\mathbb{N})$, which permutes the list and outputs an ordered list, which is studied in [5]. Fundamentally, this is a very extrinsic view of program verification, which is commonly used in program verification and proof assistants, and further, a special case of a more general sorting algorithm.

Henglein [17] studies sorting functions abstractly, without requiring a total order on the underlying set. He considers sorting functions as functions on sequences (lists), and recovers the order by applying a “sorting function” on an n -element list, and looking up the position of the elements to be compared. Unlike our approach, he does not factorize sorting functions through free commutative monoids. Henglein also studies sorting on preorders where antisymmetry may not hold, whereas we have only considered total orders in this paper. We are able to give a more refined axiomatisation of sorting because we consider permutations explicitly, and work in a constructive setting (using explicit assumptions about decidability), and which is an improvement over previous work.

Another intrinsic view of correctness of sorting has been studied in [18] using bialgebras, which was further expanded in [3] with an accompanying formalisation in Cubical Agda, that matches our point of view, as explained in Proposition 76. However, their work is not just about extensional correctness of sorting, but also deriving various sorting algorithms starting from bialgebraic semantics and distributive laws. Our work is complementary, in that we are not concerned with the computational content of sorting, but rather the abstract properties of sorting functions, which are independent of a given ordering. It is unclear whether the abstract property of sorting functions can be combined with the intrinsic complexity of sorting algorithms – and that is a direction for future work.

Although we only talk about sorting lists and bags, the same ideas can be applied to other inductive types (polynomial functors) such as trees. We speculate that this could lead to some interesting connections with sorting (binary) trees, and constructions of (binary) search trees, from classical computer science.

References

- 1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of Containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 23–38, Berlin, Heidelberg, 2003. Springer. doi:10.1007/3-540-36576-1_2.

- 2 Peter Aczel. On Voevodsky's Univalence Axiom. *Third European Set Theory Conference*, July 2011. URL: https://www.cs.man.ac.uk/~petera/Recent-Slides/Edinburgh-2011-slides_pap.pdf.
- 3 G. C. (Cass) Alexandru. Intrinsically correct sorting using bialgebraic semantics. Master's thesis, Radboud University, 2023. URL: <https://www.ru.nl/icis/education/master-thesis/vm/theses-archive/>.
- 4 Thorsten Altenkirch, Thomas Anberree, and Nuo Li. Definable Quotients in Type Theory. Manuscript, 2011. URL: <http://www.cs.nott.ac.uk/~psztxa/publ/defquotients.pdf>.
- 5 Andrew W. Appel. *Verified Functional Algorithms*, volume 3. Princeton University, August 2023. URL: <https://softwarefoundations.cis.upenn.edu/vfa-current/index.html>.
- 6 Garrett Birkhoff. On the Structure of Abstract Algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, October 1935. URL: <https://www.cambridge.org/core/journals/mathematical-proceedings-of-the-cambridge-philosophical-society/article/on-the-structure-of-abstract-algebras/D142B3886A3B7A218D8DF8E6DDA2B5B1>, doi:10.1017/S0305004100013463.
- 7 Andreas Blass. Words, free algebras, and coequalizers. *Fundamenta Mathematicae*, 117(2):117–160, 1983. URL: <http://www.impan.pl/get/doi/10.4064/fm-117-2-117-160>, doi:10.4064/fm-117-2-117-160.
- 8 Vikraman Choudhury and Marcelo Fiore. Free Commutative Monoids in Homotopy Type Theory. *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 1 - Proceedings of...:10492, February 2023. URL: <https://entics.episciences.org/10492>, doi:10.46298/entics.10492.
- 9 Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. Symmetries in reversible programming: From symmetric rig groupoids to reversible programming languages. *Proceedings of the ACM on Programming Languages*, 6(POPL):6:1–6:32, January 2022. doi:10.1145/3498667.
- 10 Vikraman Choudhury and Wind Wong. Artifact for Symmetries in Sorting. Software, sw-hId: swh:1:dir:5e86a19fe276e62519af96e6bf3c8ec42f8fe4f2 (visited on 2026-07-20). URL: <https://github.com/windtf/agda-symmetries>, doi:10.4230/artifacts.26218.
- 11 Vikraman Choudhury and Wind Wong. Symmetries in Sorting, December 2025. URL: <http://arxiv.org/abs/2512.07349>, arXiv:2512.07349, doi:10.48550/arXiv.2512.07349.
- 12 J. R. B. Cockett. List-arithmetic distributive categories: LocoI. *Journal of Pure and Applied Algebra*, 66(1):1–29, October 1990. URL: <https://www.sciencedirect.com/science/article/pii/002240499090121W>, doi:10.1016/0022-4049(90)90121-W.
- 13 Matthew L. Daggitt, Guillaume Allais, James McKinna, Andreas Abel, Nathan van Doorn, James Wood, Ulf Norell, Donnacha Oisín Kidney, Sergei Meshveliani, Sandro Stucki, Jacques Carette, Alex Rice, Jason Z. S. Hu, Li-yao Xia, Shu-Hung You, Reed Mullanix, and Wen Kokke. The Agda standard library: version 2.0. *Journal of Open Source Software*, 10(116):9241, December 2025. URL: <https://joss.theoj.org/papers/10.21105/joss.09241>, doi:10.21105/joss.09241.
- 14 Nils Anders Danielsson. Bag Equivalence via a Proof-Relevant Membership Relation. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 149–165, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-32347-8_11.
- 15 Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, USA, 1st edition, September 1997.
- 16 Eduardo J Dubuc. Free monoids. *Journal of Algebra*, 29(2):208–228, May 1974. URL: <https://www.sciencedirect.com/science/article/pii/0021869374900957>, doi:10.1016/0021-8693(74)90095-7.
- 17 Fritz Henglein. What is a Sorting Function? *The Journal of Logic and Algebraic Programming*, 78(7):552–572, August 2009. URL: <https://linkinghub.elsevier.com/retrieve/pii/S1567832608001094>, doi:10.1016/j.jlap.2008.12.003.

- 18 Ralf Hinze, Daniel W.H. James, Thomas Harper, Nicolas Wu, and José Pedro Magalhães. Sorting with bialgebras and distributive laws. In *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming*, WGP '12, pages 69–80, New York, NY, USA, September 2012. Association for Computing Machinery. URL: <https://dl.acm.org/doi/10.1145/2364394.2364405>, doi:10.1145/2364394.2364405.
- 19 Martin Hyland and John Power. The Category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads. *Electronic Notes in Theoretical Computer Science*, 172:437–458, April 2007. URL: <https://linkinghub.elsevier.com/retrieve/pii/S1571066107000874>, doi:10.1016/j.entcs.2007.02.019.
- 20 Philipp Joram and Niccolò Veltri. Constructive Final Semantics of Finite Bags. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ITP.2023.20*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2023. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.20>, doi:10.4230/LIPIcs.ITP.2023.20.
- 21 G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. *Bulletin of the Australian Mathematical Society*, 22(1):1–83, August 1980. URL: <https://www.cambridge.org/core/journals/bulletin-of-the-australian-mathematical-society/article/unified-treatment-of-transfinite-constructions-for-free-algebras-free-monoids-colimits-associated-sheaves-and-so-on/FE2E25E4959E4D8B4DE721718E7F55EE>. doi:10.1017/S0004972700006353.
- 22 Clemens Kupke, Fredrik Nordvall Forsberg, and Sean Watters. A Fresh Look at Commutativity: Free Algebraic Structures via Fresh Lists. In Chung-Kil Hur, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 135–154, Singapore, 2023. Springer Nature. doi:10.1007/978-981-99-8311-7_7.
- 23 F. William Lawvere. FUNCTORIAL SEMANTICS OF ALGEBRAIC THEORIES. *Proceedings of the National Academy of Sciences*, 50(5):869–872, November 1963. URL: <https://pnas.org/doi/full/10.1073/pnas.50.5.869>, doi:10.1073/pnas.50.5.869.
- 24 Nuo Li. Quotient types in type theory, July 2015. URL: <http://eprints.nottingham.ac.uk/28941/>.
- 25 Maria Emilia Maietti. Joyal’s arithmetic universe as list-arithmetic pretopos. *Theory and Applications of Categories*, 24:39–83, 2010. URL: <http://www.tac.mta.ca/tac/volumes/24/3/24-03abs.html>, doi:10.70930/tac/qee78enb.
- 26 Martin Mozler. Cubical Agda: Simple Application of Fin: Lehmer Codes. Agda Github Community, 2021. URL: <https://github.com/agda/cubical/blob/a1d2bb38c0794f3cb00610cd6061cf9b5410518d/Cubical/Data/Fin/LehmerCode.agda>.
- 27 J Reiterman and V Trnková. Free structures. *Category Theory at Work*, pages 277–288, 1991.
- 28 The Agda Community. Proper support for inductive families in Cubical Agda · Issue #3733 · agda/agda, April 2019. URL: <https://github.com/agda/agda/issues/3733>.
- 29 The Agda Community. Cubical Agda Library, July 2025. URL: <https://github.com/agda/cubical>.
- 30 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations Program, Institute for Advanced Study, 2013. URL: <https://homotopytypetheory.org/book>.
- 31 Vladimir Voevodsky. Resizing Rules - their use and semantic justification, 2011. URL: https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2011_Bergen.pdf.
- 32 Robert Frank Carslaw Walters. A categorical approach to universal algebra, 1970. URL: <https://openresearch-repository.anu.edu.au/handle/1885/133321>, doi:10.25911/5D723A9AE331A.

HoTT Operads

Brandon Hewer   

School of Computer Science, University of Nottingham, UK

Graham Hutton   

School of Computer Science, University of Nottingham, UK

Abstract

Internalising classes of datatypes has been a longstanding pursuit in type theory. For example, containers capture strictly positive types, while combinatorial species capture finitely labelled structures. To date, however, there has been no similar attempt to internalise classes of *operations* on datatypes. In this paper we show how the theory of operads, which extend species with a well-behaved notion of composition, provides a natural approach to internalising finitary operations. We present an internalisation of a generalised notion of operad in homotopy type theory, which provides a generic framework for capturing and reasoning about operations with particular algebraic properties. All our results are formalised in Cubical Agda.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Constructive mathematics

Keywords and phrases operads, homotopy type theory, proof assistants

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.4

Supplementary Material *Software (Formalisation)*: <https://github.com/brandonhewer/Operads-HoTT>

Funding This work was partially funded by EPSRC grant EP/Y010744/1, *Semantics-Directed Compiler Construction: From Formal Semantics to Certified Compilers*.

Acknowledgements The authors thank the reviewers for their constructive feedback and suggestions that have helped to improve this article.

1 Introduction

Type theorists have a longstanding interest in studying classes of datatypes. For example, containers [1] provide an internal theory of datatypes that captures strictly positive types [2], while combinatorial species capture finitely labelled structures [26]. Whereas containers represent datatypes by their “shapes” and “positions”, species describe the construction of a structure from a finite set of labels. One of the key applications of both containers and species is generic programming in a dependent type theory. In this way, both theories give rise to an internalised calculus of datatypes.

It is also natural to consider a calculus of *operations over datatypes*. Surprisingly, this idea has been much less explored in type theory. However, category theory provides a well-known tool for describing such algebraic structures. In particular, the notion of an *operad* generalises the concept of a category to a “multicategory” (with one object), where the source of a morphism is a finite sequence of objects. The theory of operads can be viewed as an extension of the theory of combinatorial species, in which an operad is simply a species together with a well-behaved notion of composition.

In this paper we present an internalised calculus of composable operations by realising the categorical notion of an operad in a suitable intensional type theory. The key contribution of the paper is the presentation of a generalised notion of operad in homotopy type theory (HoTT) that allows for different equational theories to be captured by varying the groupoid on which operations are indexed. More concretely, we make the following contributions:



© Brandon Hewer and Graham Hutton;

licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 4; pp. 4:1–4:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- Internalising the notion of planar (section 4) and symmetric (section 5) operads in HoTT, and providing practical examples of each form;
- Introducing a generalised notion of operad whose operations are indexed by an arbitrary groupoid (section 6), and proving that such operads form a category (section 7);
- Proving that every such operad induces a canonical monad (section 8), which provides an *operadic* style of constructing programs from a small collection of composable terms;
- Showing how the free operad can be constructed as a higher inductive family (section 9).

All our results are formalised in Cubical Agda [4] version 0.9. The paper is aimed at a general type theory audience who are interested in investigating operadic structures, and we provide an introduction to the required additional concepts from cubical type theory in Section 2.

2 Background and Metatheory

In this paper, ideas and formalisations are presented in the metatheory of Cubical Agda [23]. We begin by reviewing *path types*, a key concept in HoTT. For any type A , we can construct the type $x \equiv y$ of *paths* between terms $x, y : A$. The underlying definition of a path type varies between different models of HoTT. For example, in the CCHM model of cubical type theory [12], on which Cubical Agda is based, a path type on elements of A corresponds to a continuous function from the real interval $[0, 1]$ to A .

Readers unfamiliar with HoTT can think of the path type $x \equiv y$ as behaving identically to the inductively defined identity type. In particular, the eliminator for path types is given by the J -rule, which states that for any term $x : A$ and family of types $M : (y : A) \rightarrow x \equiv y \rightarrow \text{Type}$, if we have a proof $t : M \ x \ \mathbf{refl}$ then for all $y : A$ and $p : x \equiv y$ we can construct a term $J_{M,t} \ x \ y \ p : M \ y \ p$. Intuitively, the J -rule states that if the end-point y of the path p can vary, then we can substitute p for reflexivity at x .

Cubical Agda also provides a primitive construction for heterogeneous path types which are presented in the form `PathP` $(\lambda i \rightarrow T) \ a \ b$ for terms $a : A$, $b : B$ and $T : \text{Type}$. The first argument to `PathP` is a continuous function out of the built-in interval type `I` for which the endpoints `i0` `i1` : `I` are mapped to a and b respectively.

3 Basic Idea

In this section we introduce the notion of an operation and a collection of operators using a simple example, and describe what it means for a collection of operations to be operadic.

Informally, we can think of an *operation* as a map from elements of a structured collection to an element of the same collection. For example, addition of natural numbers and disjoint union of finite sets are examples of *binary* operations, whose domain is given by the binary product of the underlying collection. More generally, we will consider any domain that can be expressed as a strictly positive endofunctor on the underlying collection, e.g. finite and countable products. That is, we can understand an operation on a type A to simply be given by a strictly positive endofunctor $F : \text{Type} \rightarrow \text{Type}$, together with an F -algebra $f : F \ A \rightarrow A$. For example, consider the following simple expression language in Agda:

```
data Expr : Type where
  val : ℕ → Expr
  add : Expr → Expr → Expr
```

The constructors `val` and `add` can be seen as operations on `Expr`: `val` is an algebra for the constant functor choosing `ℕ`, while `add` is a (curried) algebra for the binary diagonal functor.

While the above definition of an operation is sufficient to discuss properties of operations in isolation, it is not sufficient to describe how operations interact, and in particular compose. For example, consider the collection of operations constructed as finite compositions of `val` and `add`, together with the identity function `id : Expr → Expr`. Importantly, the inclusion of `id` allows us to introduce variables, e.g. in the term $\lambda x y z \rightarrow \text{add } (\text{add } x \text{ (val } 1)) \text{ (add } y z)$. Operations constructed in this way are *finite* and *linear*, where *linear* means that each variable bound by the surrounding λ appears exactly once in the body of the operation.

To represent the desired collection of operations on `Expr`, we first observe that the type $\sum [n \in \mathbb{N}] ((\text{Fin } n \rightarrow \text{Expr}) \rightarrow \text{Expr})$, where `Fin :  $\mathbb{N} \rightarrow \text{Type}$` is the family of totally-ordered finite sets, corresponds to all (non-symmetric) finitary operations on `Expr`. However, our goal is to capture only those operations constructible as compositions of `val`, `add` and `id`. In order to do this, we might search for a subtype of all finitary operations which precisely corresponds to such a collection. Alternatively, it is perhaps easier and more natural to give an abstract representation of this collection, together with an interpretation as concrete operations. Indeed, we can do this by first defining the following inductive family:

```
data IExpr :  $\mathbb{N} \rightarrow \text{Type}$  where
  id↑ : IExpr 1
  val↑ :  $\mathbb{N} \rightarrow \text{IExpr } 0$ 
  add↑ : IExpr m → IExpr n → IExpr (m + n)
```

For every $n : \mathbb{N}$, we can think of `IExpr n` as representing the n -ary operations that are constructible as finite compositions of `id`, `val` and `add`. In particular, `id↑` and `val↑` correspond directly to `id` and `val`, while `add↑` describes how the outputs of an m -ary and n -ary operation can be plugged into the inputs of `add` to construct an $(m + n)$ -ary operation. To define a function $\llbracket \_ \rrbracket$ that interprets abstract terms of `IExpr n` as concrete n -ary functions, we use projections $\pi^1 : (\text{Fin } (m + n) \rightarrow \text{Expr}) \rightarrow \text{Fin } m \rightarrow \text{Expr}$ and $\pi^2 : (\text{Fin } (m + n) \rightarrow \text{Expr}) \rightarrow \text{Fin } n \rightarrow \text{Expr}$, which project the first m and last n elements from an $(m + n)$ -fold product:

```
 $\llbracket \_ \rrbracket : \text{IExpr } n \rightarrow (\text{Fin } n \rightarrow \text{Expr}) \rightarrow \text{Expr}$ 
 $\llbracket \text{id}\uparrow \rrbracket es = es \text{ zero}$ 
 $\llbracket \text{val}\uparrow n \rrbracket es = \text{val } n$ 
 $\llbracket \text{add}\uparrow e_1 e_2 \rrbracket es = \text{add } (\llbracket e_1 \rrbracket (\pi^1 es)) (\llbracket e_2 \rrbracket (\pi^2 es))$ 
```

While we do not give a proof here, it can be shown that $\llbracket \_ \rrbracket$ is injective, and as the codomain $(\text{Fin } n \rightarrow \text{Expr}) \rightarrow \text{Expr}$ is an h-set, it is also an embedding. Consequently, we can understand `IExpr n` to be a subtype of $(\text{Fin } n \rightarrow \text{Expr}) \rightarrow \text{Expr}$, and indeed up to equivalence it is precisely the type of operations that are constructible as finite compositions of `add`, `val` and `id`.

It is natural to identify a reasonable condition for when a countable family such as `IExpr :  $\mathbb{N} \rightarrow \text{Type}$` can be considered an abstract collection of finitary operations without appealing to a specific interpretation function such as $\llbracket \_ \rrbracket$. Importantly, this condition should capture the compositional structure of the operations. The minimal such condition we will require is for the family to be equipped with an identity operation and closed under an n -ary composition map that respects identity and associativity laws. To give a definition of an n -ary composition map, we first require a family of functions `sum n : (Fin n →  $\mathbb{N}$ ) →  $\mathbb{N}$` for calculating the sum of n natural numbers, which can be defined by induction on n in the obvious way. For every $n : \mathbb{N}$ and n -ary product of natural numbers $ns : \text{Fin } n \rightarrow \mathbb{N}$, we can now construct a family of functions `comp n ns : IExpr n → ((i : Fin n) → IExpr (ns i)) → IExpr (sum n ns)`:


```

comp 1 ns id↑ es = es zero
comp 0 ns (val↑ k) es = val↑ k
comp .(m + n) ns (add↑ e1 e2) es =
  let es1 = comp m (π1 ns) e1 (Π1 es)
    es2 = comp m (π2 ns) e2 (Π2 es)
  in subst IExpr (π1+π2 ns) (add↑ es1 es2)

```

In the above definition, we use the n -ary dependent projection functions Π^1 and Π^2 , together with the path $\pi^1 + \pi^2 ns : \text{sum } m (\pi^1 ns) + \text{sum } n (\pi^2 ns) \equiv \text{sum } (m + n) ns$, whose constructions can be found in our Cubical Agda formalisation.

Intuitively, our composition map plugs the outputs of n operations into the inputs of an n -ary operation. Importantly, we can show that `comp` preserves associativity and identity laws, where the identity operation is given by `id↑`. The left and right identity laws assert that for any n -ary operation, plugging its output into the input of the identity operation, or plugging the identity’s output into each of its n inputs, leaves the operation unchanged. Associativity states that composing a finitely branching tree of operations “bottom-up” is the same as composing “top-down”. We formally characterise these laws in Section 4.

We can also observe that a similar compositional structure to that described by `comp` exists for the family of all finitary operations $(\text{Fin } n \rightarrow \text{Expr}) \rightarrow \text{Expr}$, where the composition map is simply given by n -ary function composition, and the unit operation is given by the term $\lambda es \rightarrow es \text{ zero} : (\text{Fin } 1 \rightarrow \text{Expr}) \rightarrow \text{Expr}$. The interpretation function $\llbracket \_ \rrbracket : \text{IExpr } n \rightarrow (\text{Fin } n \rightarrow \text{Expr}) \rightarrow \text{Expr}$ can then be understood as a homomorphism between such structures, respecting both the composition map and the identity operation. This common compositional structure on collections of operations, is precisely the structure described by planar operads.

4 Planar Operads

A collection of finitary operations with a distinguished unit, closed under an n -ary composition map that respects unitality and associativity, is known as a *planar operad*. For example, the compositional structure for `IExpr` in the previous section describes a planar operad. In particular, planar here simply means that our operadic composition map has a total order on input operations. Consequently, the composition map of a planar operad does not necessarily respect reordering of the input operations. In this section we formalise planar operads in Cubical Agda, and provide another example and important properties of our definition.

We begin by recalling that a countable family of types $K : \mathbb{N} \rightarrow \text{Type}$ can be understood as an *arity-indexed family of types of operations*: for each $n : \mathbb{N}$, an element of $K \ n$ represents an abstract n -ary operation, without yet committing to its action on data. The operadic structure on K then specifies how such operations compose. To avoid higher coherences, we consider only families of h-sets, i.e. $K : \mathbb{N} \rightarrow \text{hSet}$ where `hSet` is the universe of h-sets. In Cubical Agda, `hSet` is not a built-in universe but is expressed as a dependent sum $\sum [A \in \text{Type}] \text{isSet } A$, where `isSet` A is the proposition that A is an h-set. However, given a family of sets $K : \mathbb{N} \rightarrow \text{hSet}$, we will write $K \ n : \text{Type}$ for the first projection, and make it clear when we use the proof that this type is an h-set.

► **Definition 1** (Planar operad). *Given any family of h-sets $K : \mathbb{N} \rightarrow \text{hSet}$, we can define the planar operads on K as a record type `record NonSymmOperad` $K : \text{Type}$, i.e. a finitely iterated sigma type with named projections. In particular, a planar operad $O : \text{NonSymmOperad } K$ has a distinguished identity operation `id` $O : K \ 1$ and a family of composition maps*

$$\text{comp } O \ n \ ns : K \ n \rightarrow ((i : \text{Fin } n) \rightarrow K \ (ns \ i)) \rightarrow K \ (\text{sum } n \ ns),$$

for every $n : \mathbb{N}$ and $ns : \text{Fin } n \rightarrow \mathbb{N}$. The composition map tells us how to plug the outputs of n operations, each with its own number of inputs ns , into the inputs of an operation with n inputs. Furthermore, we require paths witnessing that this composition map respects the identity and associativity laws. In particular, the left identity law is witnessed by a path

$$\text{idl } O \ n \ k : \text{comp } O \ 1 \ (\lambda i \rightarrow n) \ (\text{id } O) \ (\lambda i \rightarrow k) \equiv k,$$

for every $n : \mathbb{N}$ and $k : K \ n$. The path $\text{idl } O \ n \ k$ witnesses that plugging the output of k into the identity is equal to k . Right identity is then witnessed by a heterogeneous path

$$\text{idr } O \ n \ k : \text{PathP } (\lambda i \rightarrow K \ (\text{sum-idr } n \ i)) \ (\text{comp } O \ n \ (\lambda i \rightarrow 1) \ k \ (\lambda i \rightarrow \text{id } O)) \ k,$$

where $\text{sum-idr } n : \text{sum } n \ (\lambda i \rightarrow 1) \equiv n$ witnesses that the sum of n ones is n . Intuitively, $\text{idr } O \ n \ k$ witnesses that the operation plugging the output of the identity into each of the n inputs of k is equal to k . Finally, in order to state the operad associativity law, we first introduce the following family of equivalences,

$$\text{sum}\Sigma \ ns : \text{Fin } (\text{sum } n \ ns) \rightarrow \Sigma [i \in \text{Fin } n] \text{Fin } (ns \ i)$$

for all $n : \mathbb{N}$, $ns : \text{Fin } n \rightarrow \mathbb{N}$. This family witnesses that totally ordered finite sets are closed under dependent sums, and can be used to define the associativity condition for sum ,

$$\begin{aligned} \text{sum-assoc } n \ ns \ nss : \text{sum } n \ (\lambda i \rightarrow \text{sum } (ns \ i) \ (nss \ i)) \\ \equiv \text{sum } (\text{sum } n \ ns) \ (\text{uncurry } nss \circ \text{sum}\Sigma \ ns) \end{aligned}$$

for all $nss : (i : \text{Fin } n) \rightarrow \text{Fin } (ns \ i) \rightarrow \mathbb{N}$. The operad associativity law, $\text{assoc } O \ n \ ns \ nss$, can then be expressed as a heterogeneous path over $\text{cong } K \ (\text{sum-assoc } n \ ns \ nss)$ between top-down composition

$$\text{comp } O \ n \ (\lambda i \rightarrow \text{sum } (ns \ i) \ (nss \ i)) \ (\lambda i \rightarrow \text{comp } O \ (ns \ i) \ (nss \ i) \ (ks \ i) \ (kss \ i))$$

$$\begin{aligned} \text{comp } O \ n \ (\text{sum } n \ ns) \ (\text{uncurry } nss \circ \text{sum}\Sigma \ ns) \\ (\text{comp } O \ n \ ns \ k \ ks) \ (\text{uncurry } kss \circ \text{sum}\Sigma \ ns) \end{aligned}$$

Beyond the LExp operad of Section 3, a second example is given by *partial lists* with “holes”.

► **Example 2** (Partial-list planar operad). We define the family of partial lists with “holes” as the following inductive family:

```
data PartialList (A : Type) : ℕ → Type where
  [] : PartialList A 0
  _::_ : A → PartialList A n → PartialList A n
  poke : PartialList A n → PartialList A (suc n)
```

Intuitively, the family $\text{PartialList } A$ corresponds to partial lists indexed by their number of holes. The constructors $[]$ and $_::_$ are as usual for lists, while poke introduces a hole at the start of a list. Partial lists have a concatenation operation $_{++} : \text{PartialList } A \ m \rightarrow \text{PartialList } A \ n \rightarrow \text{PartialList } A \ (m + n)$, defined by:

```
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
poke xs ++ ys = poke (xs ++ ys)
```

4:6 HoTT Operads

Given any h-set $A : \text{Type}$, the countable family of h-sets $\text{PartialList } A$ comes equipped with a planar operadic structure

$$O : \text{NonSymmOperad } (\text{PartialList } A)$$

with n -ary composition map defined as follows:

$$\begin{aligned} \text{comp } O \ 0 \ ns \ xs \ xss &= xs \\ \text{comp } O \ (\text{suc } n) \ ns \ (x :: xs) \ xss &= x :: \text{comp } O \ (\text{suc } n) \ ns \ xs \ xss \\ \text{comp } O \ 1 \ ns \ (\text{poke } xs) \ xss &= \text{subst } (\text{PartialList } A) \ (+\text{-zero } (ns \ \text{zero})) \ (xss \ \text{zero} ++ xs) \\ \text{comp } O \ (\text{suc } (\text{suc } n)) \ ns \ (\text{poke } xs) \ xss &= xss \ \text{zero} ++ \text{comp } O \ (\text{suc } n) \ (ns \circ \text{suc}) \ xs \ (xss \circ \text{suc}) \end{aligned}$$

The identity operation for the compositional structure on partial lists is then simply $\text{id } O = \text{poke } []$. The identity and associativity laws are proved in our Cubical Agda formalisation.

We might ask if the operations characterised by the operadic structure on $\text{PartialList } A$ correspond to a collection of concrete n -ary operations. Indeed, there is an interpretation $\text{fill} : \text{PartialList } A \ n \rightarrow (\text{Fin } n \rightarrow \text{List } A) \rightarrow \text{List } A$ which fills n holes with n lists.

► **Proposition 3** (fill is a planar operad homomorphism). *Similarly to $[\llbracket \_ \rrbracket] : \text{IExpr } n \rightarrow (\text{Fin } n \rightarrow \text{Expr}) \rightarrow \text{Expr}$ in Section 3, fill is a homomorphism between planar operads and respects the compositional structure and identity operation. The proof can be found in our Cubical Agda formalisation, and we introduce operad morphisms and algebras in Section 7.*

5 Symmetric Operads

In the previous section we introduced planar operads in HoTT. These generalise the usual notion of operad, whose operations also come equipped with actions of symmetric groups compatible with the composition map. In this section, we show how our notion of a planar operad can be specialised to the symmetric case. Naively, this means extending our definition of a planar operad $O : \text{NonSymmOperad } K$ with a field

$$\text{permute } O \ n : \text{Fin } n \simeq \text{Fin } n \rightarrow K \ n \simeq K \ n,$$

for every $n : \mathbb{N}$, where $A \simeq B$ is the type of equivalences between the h-sets A and B . An equivalence $\sigma : A \simeq B$ has projections $\text{fun } \sigma : A \rightarrow B$, $\text{inv } \sigma : B \rightarrow A$, $\text{linv} : (a : A) \rightarrow \text{inv } \sigma \ (\text{fun } \sigma \ a) \equiv a$ and $\text{rinv} : (b : B) \rightarrow \text{fun } \sigma \ (\text{inv } \sigma \ b) \equiv b$. Moreover, $\text{permute } O \ n$ must be a group homomorphism between the automorphism groups $\text{Fin } n \simeq \text{Fin } n$ and $K \ n \simeq K \ n$, but we omit the evident fields to witness preservation of identity, composition and symmetry. Furthermore, for all $ns : \text{Fin } n \rightarrow \mathbb{N}$, $k : K \ n$, $ks : (i : \text{Fin } n) \rightarrow K \ (ns \ i)$ and $\sigma : \text{Fin } n \simeq \text{Fin } n$, we require a field witnessing that $\text{permute } O \ n$ is compatible with the operad composition map. That is, this field asserts the operadic *equivariance* law, which states that the symmetric-group action commutes with composition. We retain the name symm to mirror our formalisation:

$$\begin{aligned} \text{symm } O \ n \ ns \ k \ ks \ \sigma &: \text{PathP } (\lambda i \rightarrow K \ (\text{sum-permute } n \ ns \ \sigma \ i)) \\ &(\text{comp } O \ n \ ns \ (\text{fun } (\text{permute } O \ n \ \sigma) \ k) \ ks) \\ &(\text{comp } O \ n \ ns \ k \ (ks \circ \text{fun } \sigma)) \end{aligned}$$

This naive presentation of symmetric operads can be internalised in standard Martin-Löf type theory, but requires an explicit construction of the permutation homomorphism and proof of compatibility with the operad composition map. However, this approach can be

unwieldy in practice. This problem arises because a countable family of h-sets $K : \mathbb{N} \rightarrow \text{Type}$ does not always come equipped with actions of symmetric groups. Intuitively, this means that the composition map of a planar operad can freely make use of the ordering of operations. This is an expected consequence of indexing operations by the natural numbers, which are in equivalence with the type of finite sets equipped with a total order, i.e. the type $\sum [A \in \text{Type}] \sum [n \in \mathbb{N}] A \simeq \text{Fin } n$. This can readily be seen by recognising that for any $n : \mathbb{N}$, the type $\sum [A \in \text{Type}] A \simeq \text{Fin } n$ is, by univalence, equivalent to the singleton type $\sum [A \in \text{Type}] A \equiv \text{Fin } n$ centred at $\text{Fin } n$, and is therefore contractible.

Alternatively, we can exploit additional principles in HoTT to consider operadic structures on collections of operations indexed over the groupoid of finite sets and bijections, i.e. finite sets that are not always totally-ordered. A common presentation of this groupoid, Bishop-finite sets [26, 24], internalises finiteness as a predicate on types.

► **Definition 4** (Bishop-finite sets). *A type $A : \text{Type}$ is Bishop-finite when*

$$\text{isFinite } A = \sum [n \in \mathbb{N}] \parallel A \simeq \text{Fin } n \parallel$$

is inhabited, where $\parallel X \parallel$ denotes the propositional truncation of X . The universe of Bishop-finite sets is $\text{FinSet} = \sum [A \in \text{Type}] \text{isFinite } A$, with interpretation map $\text{El} : \text{FinSet} \rightarrow \text{Type}$ given by the first projection.

For any type A , there is a well-known equivalence between the type $\text{isFinite } A$ and the h-proposition $\parallel \sum [n \in \mathbb{N}] A \simeq \text{Fin } n \parallel$. Intuitively, this equivalence witnesses that every finite set has a unique cardinality, and therefore this notion of finiteness is an h-proposition.

Although our definition of FinSet is a large type, it can equivalently be defined as a small type by taking an appropriate groupoid quotient of the naturals. Accordingly, we will ignore size issues related to FinSet throughout this paper. Importantly, El is an embedding, which follows from the proof that the finiteness of a type is an h-proposition. That is, for any finite sets $A B : \text{FinSet}$, the functorial action of El on paths is an equivalence between the path space of codes $A \equiv B$ and the path space of the underlying types $\text{El } A \equiv \text{El } B$. A Tarski universe with this property is also known as a univalent universe [11, 9, 10]. Since FinSet is a subtype of Type whose predicate isFinite is an h-proposition, FinSet is also a *univalent subuniverse* of the ambient universe: its codes inherit the path structure of the underlying types. We recall that the property that a type A is an h-set, i.e. $(x y : A) (p q : x \equiv y) \rightarrow p \equiv q$, is itself an h-proposition. Therefore, given $(A, n, p) : \text{FinSet}$, it follows from the truncated equivalence between A and $\text{Fin } n$ witnessed by p that, since $\text{Fin } n$ is an h-set, so is A . That is, for every finite set $A : \text{FinSet}$ the underlying type $\text{El } A$ is an h-set. Therefore, given any two finite sets $A B : \text{FinSet}$, by univalence the type of paths $A \equiv B$ is equivalent to the type of equivalences $\text{El } A \simeq \text{El } B$. We express the forward direction of this equivalence by:

$$\text{un} : (A B : \text{FinSet}) \rightarrow \text{El } A \simeq \text{El } B \rightarrow A \equiv B$$

The universe FinSet is closed under common type formers including dependent sums and products. In particular, for every finite set $A : \text{FinSet}$ and family of finite sets $B : \text{El } A \rightarrow \text{FinSet}$, we have $\hat{\sum} A B : \text{FinSet}$ and $\hat{\prod} A B : \text{FinSet}$, together with definitional equalities $\text{El} (\hat{\sum} A B) = \sum [a \in \text{El } A] \text{El} (B a)$ and $\text{El} (\hat{\prod} A B) = (a : \text{El } A) \rightarrow \text{El} (B a)$. Finiteness proofs for $\text{isFinite} (\sum [a \in \text{El } A] \text{El} (B a))$ and $\text{isFinite} ((a : \text{El } A) \rightarrow \text{El} (B a))$ can be found in our Cubical Agda formalisation, and in the UniMath Coq library [24]. Our formalisation also shows that the universe of Bishop-finite sets is closed under isomorphism. That is, for any two finite sets $A B : \text{FinSet}$ we can construct $A \hat{=} B : \text{FinSet}$ with a definitional equality $\text{El} (A \hat{=} B) = \text{El } A \simeq \text{El } B$ and a proof of $\text{isFinite} (\text{El } A \simeq \text{El } B)$.

From this notion of Bishop-finite sets internalised in HoTT, we can define a collection of finitary operations as a **FinSet**-indexed family of h-sets. Indeed, such families are also known as *combinatorial species* [20, 26], or equivalently, *symmetric finitary containers* [16, 19]: finitary containers whose position type carries a symmetric-group action induced by the indexing groupoid **FinSet**. Every such family $K : \mathbf{FinSet} \rightarrow \mathbf{Type}$ comes equipped with a canonical right action of symmetric groups of the form $\mathbf{El} A \simeq \mathbf{El} A$, or equivalently $A \equiv A$, for $A : \mathbf{FinSet}$. Importantly, $\mathbf{El} A \simeq \mathbf{El} A$ and $A \equiv A$ are equivalent not just as sets, but also as groups. In particular, for every symmetric group element $\sigma : A \equiv A$, this action is given by the automorphism witnessed by $\mathbf{subst} K \sigma : K A \rightarrow K A$ and $\mathbf{subst} K (\mathbf{sym} \sigma) : K A \rightarrow K A$. We note that as K is a family of h-sets, the group of paths $K A \equiv K A$ is equivalent to the group of automorphisms $K A \simeq K A$. It follows from the laws of substitution over paths that this construction from $A \equiv A$ to $K A \equiv K A$ is a group homomorphism. That is, up to a path, substitution over the identity is equal to the identity and substitution over the composite of paths is equal to the composition of substitutions. The proof of the symmetry law follows from the identity and composition laws, and the symmetry law on paths: $p \cdot \mathbf{sym} p \equiv \mathbf{refl}$.

So far, we have seen how **FinSet**-indexed families have a canonical notion of right action of symmetric groups, in contrast to that of $\mathbb{N}$ -indexed families. However, we haven't described what it means for a family of types indexed over **FinSet** to have an operadic structure. To do so, we begin by considering the definition of planar operads and replacing the finite summation function **sum** with the dependent sum type former of finite sets $\hat{\Sigma}$. The intuitive connection is that **sum** is the dependent sum type former for the Tarski universe with codes given by $\mathbb{N}$ and interpretation map $\mathbf{Fin} : \mathbb{N} \rightarrow \mathbf{Type}$, i.e. the universe of totally-ordered finite sets. For example, given a collection of finitary operations $K : \mathbf{FinSet} \rightarrow \mathbf{Type}$, a finite set $A : \mathbf{FinSet}$, and family of finite sets $B : \mathbf{El} A \rightarrow \mathbf{FinSet}$, the composition map of an operad $O : \mathbf{Operad} K$ is given by a field

$$\mathbf{comp} O A B : K A \rightarrow ((a : \mathbf{El} A) \rightarrow K (B a)) \rightarrow K (\hat{\Sigma} A B)$$

Moreover, given the singleton finite set $\top : \mathbf{FinSet}$ and a family of finite sets $C : (a : \mathbf{El} A) \rightarrow \mathbf{El} (B a) \rightarrow \mathbf{FinSet}$, the right identity and associativity laws are now given as heterogeneous paths over the following canonical paths:

$$\begin{aligned} \hat{\Sigma}\text{-idr} A &: \hat{\Sigma} A (\lambda a \rightarrow \top) \equiv A \\ \hat{\Sigma}\text{-assoc} A B C &: \hat{\Sigma} A (\lambda a \rightarrow \hat{\Sigma} (B a) (C a)) \equiv \hat{\Sigma} (\hat{\Sigma} A B) (\lambda (a, b) \rightarrow C a b) \end{aligned}$$

We can construct these paths by applying **un** to the corresponding isomorphisms in the meta-theory, i.e. for all $A : \mathbf{Type}$, $B : A \rightarrow \mathbf{Type}$ and $C : (a : A) \rightarrow B a \rightarrow \mathbf{Type}$, we construct $\hat{\Sigma}\text{-idr}$ and $\hat{\Sigma}\text{-assoc}$ by applying **un** to the following canonical isomorphisms:

$$\begin{aligned} \Sigma\text{-idr} &: A \times \top \simeq A \\ \Sigma\text{-assoc} &: \sum [a \in A] \sum [b \in B a] C a b \simeq \sum [(a, b) \in \sum [a \in A] B a] C a b \end{aligned}$$

Furthermore, given any operation $k : K A$ it is also now necessary to witness the left identity as a heterogeneous path as follows:

$$\mathbf{idl} O A k : \mathbf{PathP} (\lambda i \rightarrow K (\hat{\Sigma}\text{-idl} A i)) (\mathbf{comp} O A (\lambda a \rightarrow \top) k (\lambda a \rightarrow \mathbf{id} O)) k,$$

where the path $\hat{\Sigma}\text{-idl} A : \hat{\Sigma} \top (\lambda t \rightarrow A) \equiv A$ is constructed by the application of **un** $(\hat{\Sigma} \top (\lambda t \rightarrow A)) A$ to the canonical isomorphism that exists between $\top \times \mathbf{El} A$ and $\mathbf{El} A$.

To complete our definition of a symmetric operad, we might consider adding a field witnessing that our previously described right actions of symmetric groups on the collection of operations K is compatible with the composition map. That is, we might introduce an analogue to the `symm` field for `FinSet`-indexed families. In particular, for every finite set $A : \text{FinSet}$, family of finite sets $B : \text{El } A \rightarrow \text{FinSet}$, operation $k : K \ A$, family of operations $ks : (a : \text{El } A) \rightarrow K \ (\text{El } (B \ a))$ and symmetric group element $\sigma : A \equiv A$, we could introduce the following path as a field:

```

symm O A B k ks σ : let p i a = transp (λ j → El (σ (i ∨ j))) i a in
  PathP (λ i → K (∑ (σ i) (λ a → B (p i a))))
    (comp O A (B ∘ subst El σ) k (ks ∘ subst El σ))
    (comp O A B (subst K σ k) ks)

```

However, it turns out that this form of `symm`, where the right action of a symmetric group is defined by path substitution, can already be constructed in Cubical Agda as follows:

```

symm O A B k ks σ i =
  comp O (σ i) (λ a → B (transp (λ j → El (σ (i ∨ j))) i a))
    (transp (λ j → K (σ (i ∧ j))) (∼ i) k)
    (λ a → ks (transp (λ j → El (σ (i ∨ j))) i a))

```

That is, by switching from `ℕ` to `FinSet` indexed families, and changing from `sum` to the dependent sum type former $\sum$, our notion of an operad restricts to the classical definition which includes compatibility with the right actions of symmetric groups.

► **Definition 5** (Symmetric operad on a `FinSet`-indexed family). *Given a family of h -sets $K : \text{FinSet} \rightarrow h\text{Set}$, a symmetric operad on K is a term of the record type `Operad` K whose fields are obtained from Definition 1 by replacing `sum` by $\sum$, `Fin n` by `El A`, and the indexing `ℕ` by `FinSet`. The record retains the same five fields `id`, `comp`, `idl`, `idr` and `assoc`; the equivariance law is derivable from these fields together with the substitution actions of `FinSet`, and so does not appear as a separate field.*

We can similarly translate our notion of planar operad morphism to use `FinSet` and $\sum$, and we don't require a field witnessing preservation of the actions of symmetric groups. In particular, for any two families $K \ L : \text{FinSet} \rightarrow \text{Type}$, family of maps $(A : \text{FinSet}) \rightarrow K \ A \rightarrow L \ A$, operation $k : K \ A$, and symmetric group element $\sigma : A \equiv A$, a proof of $f \ A \ (\text{subst } K \ \sigma \ k) \equiv \text{subst } L \ \sigma \ (f \ A \ k)$ follows directly from substitution commuting with morphisms in slice categories.

► **Example 6** (Symmetric expression operad `SymExpr`). We recall the example of the planar operadic structure on the type family `LErr` : `ℕ` $\rightarrow$ `Type`, and introduce a near-identical `FinSet`-indexed analogue.

```

data SymExpr : FinSet → Type₁ where
  id↑ : SymExpr ⊤
  val↑ : ℕ → SymExpr ⊥
  add↑ : SymExpr A → SymExpr B → SymExpr (A ⊔ B)

```

In this example, the type former `⊔` : `FinSet` $\rightarrow$ `FinSet` $\rightarrow$ `FinSet` corresponds to coproducts in the universe of finite sets. That is, `El (A ⊔ B) = El A ⊔ El B`, and `⊥` is the empty type.

The proof that finite sets are closed under finite coproducts can again be found both in our Cubical Agda formalisation and in the UniMath library [24]. Notably, `SymExpr` is a family of large types because it is indexed over `FinSet` and `add↑` is parameterised over a large type. In Section 6 we address this with a small type of unordered finite sets.

To define the operad composition map for the family `SymExpr`, for all types $A, B : \mathbf{Type}$ and families $C : A \uplus B \rightarrow \mathbf{Type}$, we will make use of the following canonical isomorphism:

$$\begin{aligned} \uplus\text{-distr } A B C &: (\sum [a \in A] C (\text{inl } a)) \uplus (\sum [b \in B] C (\text{inr } b)) \\ &\simeq (\sum [x \in A \uplus B] C x) \end{aligned}$$

where $\uplus\text{-distr } A B C$ is constructed in the evident way. Moreover, for all families $A : \top \rightarrow \mathbf{Type}$ and $B : \perp \rightarrow \mathbf{Type}$, we will use the following two canonical isomorphisms:

$$\sum\text{-idl } A : A \text{ tt} \simeq \sum [x \in \top] A x \quad \sum\text{-}\perp B : \perp \simeq \sum [x \in \perp] B x$$

We can then construct the composition map for an operad $O : \mathbf{Operad} \text{ SymExpr}$ as follows:

```
comp O .⊤ B id↑ es = subst SymExpr (un (El (B tt)) (sum [x ∈ ⊤] B x) (sum-idl B)) (es tt)
comp O .⊥ B (val↑ n) es = subst SymExpr (un ⊥ (sum [x ∈ ⊥] B x) (sum-⊥ B)) (val↑ n)
comp O .(A1 ⊕ A2) B (add↑ e1 e2) es =
  let es1 = comp O A1 (B ∘ inl) e1 (es ∘ inl)
      es2 = comp O A2 (B ∘ inr) e2 (es ∘ inr)
  in subst SymExpr (un _ _ (⊕-distr A1 A2 B)) (add↑ es1 es2)
```

Proofs that this construction respects the necessary identity and associativity laws are given in our Cubical Agda formalisation.

6 Generalised Operad Universes

We have seen how symmetric and planar operads can be defined in HoTT by varying the Tarski universe that indexes operations. For the planar case, we considered operations indexed by the universe of totally-ordered finite sets $(\mathbb{N}, \mathbf{Fin})$, while for the symmetric case we considered the more general groupoid of finite sets and bijections $(\mathbf{FinSet}, \mathbf{El})$. To describe how an operadic composition map acts on indices, we required both universes to be closed under dependent sums and have a unit element. In particular, for the planar case we required the finite summation map `sum` together with the unit `1`, while for the symmetric case we used closure of finite sets under dependent sums $\sum$ together with the singleton finite set $\top$. As the symmetric and planar operad definitions are otherwise uniform, this suggests a generalisation for collections of operations indexed by any universe with sufficient closure properties. In this section, we introduce an internal notion of such universes in HoTT, which allows us to work with a generalised notion of operads that supports many important operadic constructions being defined without the need to separate details for the symmetric and planar versions.

We begin by recalling that a Tarski type universe comprises a type of codes $U : \mathbf{Type}$ and an interpretation family $E : U \rightarrow \mathbf{Type}$. For example, the groupoid of finite sets and bijections forms a universe with $U = \mathbf{FinSet}$ and $E = \mathbf{El}$. In Agda, such universes can be defined as a record type $\mathcal{U} : \mathbf{Universe}$ with a field for the type of codes `Code  $\mathcal{U}$` $: \mathbf{Type}$, and for every code $A : \mathbf{Code} \mathcal{U}$ a field $\llbracket \mathcal{U} \ni A \rrbracket : \mathbf{Type}$ corresponding to the underlying type. To ensure universes are closed under dependent sums, we first introduce the requirement that $\mathcal{U}$ must come equipped with the type former

$$\hat{\Sigma} : (A : \mathbf{Code} \mathcal{U}) \rightarrow (\llbracket \mathcal{U} \ni A \rrbracket \rightarrow \mathbf{Code} \mathcal{U}) \rightarrow \mathbf{Code} \mathcal{U}$$

such that for every $A : \text{Code } \mathcal{U}$ and $B : [\mathcal{U} \ni A] \rightarrow \text{Code } \mathcal{U}$, we have an equivalence:

$$[\hat{\Sigma}] A B : [\mathcal{U} \ni (\hat{\Sigma} A B)] \simeq \sum [a \in [\mathcal{U} \ni A]] [\mathcal{U} \ni B a].$$

In addition to closure under dependent sums, we require a field $\hat{\top} : \text{Code } \mathcal{U}$ corresponding to the code for the unit type, together with an equivalence $[\mathcal{U} \ni \hat{\top}] \simeq \top$. However, these properties are not sufficient to show $\mathcal{U}$ is closed under dependent sums and a unit object. For example, for every $A : \text{Code } \mathcal{U}$, $B : [\mathcal{U} \ni A] \rightarrow \text{Code } \mathcal{U}$ and $C : (a : [\mathcal{U} \ni A]) \rightarrow [\mathcal{U} \ni B a] \rightarrow \text{Code } \mathcal{U}$, these fields are insufficient to construct a proof of associativity:

$$\hat{\Sigma}\text{-assoc } A B C : \hat{\Sigma} A (\lambda a \rightarrow \hat{\Sigma} (B a) (C a)) \equiv \hat{\Sigma} (\hat{\Sigma} A B) (\text{uncurry } C \circ [\hat{\Sigma}] A B)$$

To address this, it suffices to require a map from the path space between any two underlying types in a universe to the path space between their codes. Concretely, this means that for any two codes $A B : \text{Code } \mathcal{U}$, we require the following extra field:

$$\text{Inj } A B : [\mathcal{U} \ni A] \simeq [\mathcal{U} \ni B] \rightarrow A \equiv B$$

However, the **Inj** field alone is insufficient to map the higher path structure between underlying types in a universe to the corresponding higher path structure between their codes. In particular, we recall that this notion of a generalised operad universe will be used to index a family of h-sets corresponding to the operations of an operad. That is, given a universe $\mathcal{U} : \text{Universe}$, a $\mathcal{U}$ -operad has operations given by a family of h-sets $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$. Importantly, the type of h-sets is an h-groupoid, and hence the higher path structure of codes beyond that of the 1-groupoid structure, i.e. paths between paths, is trivially respected by K .

Hence, it is sufficient to include a coherency condition witnessing that **Inj** preserves this 1-groupoid structure. For all codes $A B C : \text{Code } \mathcal{U}$ and equivalences $e_1 : [\mathcal{U} \ni A] \simeq [\mathcal{U} \ni B]$ and $e_2 : [\mathcal{U} \ni B] \simeq [\mathcal{U} \ni C]$, we can achieve this by adding the field:

$$\text{InjComp } A B C e_1 e_2 : \text{Inj } A C (e_1 \cdot e_2) \equiv \text{Inj } A B e_1 \cdot \text{Inj } B C e_2$$

Intuitively, **InjComp** asserts that **Inj** must preserve composition of equivalences, mapping composition of equivalences to the composition of paths. Moreover, this condition is sufficient to witness that up to a path **Inj** maps identity equivalences to reflection paths.

► **Proposition 7 (InjRefl).** *For every code $A : \text{Code } \mathcal{U}$ there is a path*

$$\text{InjRefl } A : \text{Inj } A A (\text{id} \simeq [\mathcal{U} \ni A]) \equiv \text{refl},$$

where $\text{id} \simeq [\mathcal{U} \ni A] : [\mathcal{U} \ni A] \simeq [\mathcal{U} \ni A]$ is the identity equivalence on $[\mathcal{U} \ni A]$.

Proof. To construct the path **InjRefl**, we first apply **InjComp** $A A A$ to the identity equivalence (twice) on $[\mathcal{U} \ni A]$ and precompose with the functorial action of **Inj** $A A$ on paths applied to the path witnessing that the identity equivalence composed with itself is the identity equivalence. This construction gives us a path between **Inj** $A A (\text{id} \simeq [\mathcal{U} \ni A])$ and the same path composed with itself. It follows from the groupoid structure of paths that for any type A , term $x : A$, and path $p : x \equiv x$, if there is a path of type $p \cdot p \equiv p$ then we can construct a path of type $p \equiv \text{refl}$. We note that this is a purely path-algebraic fact about idempotent loops, obtained by left-cancelling p against itself, and does not require A to be a set. Hence, by applying this rule we can construct a path between **Inj** $A A (\text{id} \simeq [\mathcal{U} \ni A])$ and **refl**. ◀

► **Proposition 8 (InjSec).** *For all codes $A B : \text{Code } \mathcal{U}$ and paths $p : A \equiv B$, there is a path*

$$\text{InjSec } A B p : \text{Inj } A B (\text{pathToEquiv } (\text{cong } [\mathcal{U} \ni \_] p)) \equiv p.$$

Proof. For any types $X Y : \text{Type}$, if we let $\text{pathToEquiv } X Y : X \equiv Y \rightarrow X \simeq Y$ be the canonical map from paths to equivalences then by applying the J -elimination rule to the composition of the path witnessing that pathToEquiv maps reflection to the identity equivalence and the path $\text{InjRefl } A$, we can construct the above family of paths. $\blacktriangleleft$

In particular, InjSec witnesses that Inj has all sections given by the composition of the functorial action of the interpretation map $\llbracket \mathcal{U} \ni \_ \rrbracket$ on paths and the canonical map from paths to equivalences. Intuitively, this condition asserts that the path spaces of the representing codes of a universe precisely reflect the path spaces of the underlying types, and no more.

To conclude our description of generalised operad universes, we first recall that an operadic structure on a collection of operations includes heterogeneous paths that witness that the composition map respects identity and associativity laws. In particular, for generalised operad universes these paths will necessarily be heterogeneous over Inj applied to the following canonical equivalences, which correspond to left/right identity and associativity,

$$\begin{aligned} \hat{\Sigma} \text{Idl} \simeq \mathcal{U} A : \llbracket \mathcal{U} \ni \hat{\Sigma} \hat{\top} (\lambda t \rightarrow A) \rrbracket &\simeq \llbracket \mathcal{U} \ni A \rrbracket \\ \hat{\Sigma} \text{Idr} \simeq \mathcal{U} A : \llbracket \mathcal{U} \ni \hat{\Sigma} A (\lambda a \rightarrow \hat{\top}) \rrbracket &\simeq \llbracket \mathcal{U} \ni A \rrbracket \\ \hat{\Sigma} \text{Assoc} \simeq \mathcal{U} A B C : \llbracket \mathcal{U} \ni \hat{\Sigma} A (\lambda a \rightarrow \hat{\Sigma} (B a) (C a)) \rrbracket &\simeq \\ \llbracket \mathcal{U} \ni \hat{\Sigma} (\hat{\Sigma} A B) (\text{uncurry } C \circ \text{fun } (\llbracket \hat{\Sigma} \rrbracket A B)) \rrbracket & \end{aligned}$$

for all $A : \text{Code } \mathcal{U}$, $B : \llbracket \mathcal{U} \ni A \rrbracket \rightarrow \text{Code } \mathcal{U}$ and $C : (a : \llbracket \mathcal{U} \ni A \rrbracket) \rightarrow \llbracket \mathcal{U} \ni B a \rrbracket \rightarrow \text{Code } \mathcal{U}$. These are precisely the equivalences constructed from the underlying canonical equivalences on the dependent sum construction in the meta-theory composed with equivalences constructed from application of $\llbracket \hat{\Sigma} \rrbracket$. To construct $\mathcal{U}$ -operadic structures, we require each of the above equivalences to be represented by the paths they are sent to by Inj . That is, we require paths between the above equivalences and the application of the map that sends each $X Y : \text{Code } \mathcal{U}$ and $e : \llbracket \mathcal{U} \ni A \rrbracket \simeq \llbracket \mathcal{U} \ni A \rrbracket$ to $\text{pathToEquiv } (\text{cong } \llbracket \mathcal{U} \ni \_ \rrbracket) (\text{Inj } X Y e)$ to the same equivalences. Equivalently, given the inverse map to pathToEquiv following from univalence, i.e. $\text{ua} : X \simeq Y \rightarrow X \equiv Y$, we can capture the required conditions with the following fields:

$$\begin{aligned} \llbracket \hat{\Sigma} \text{Idl} \rrbracket \mathcal{U} A : \text{ua } (\hat{\Sigma} \text{Idl} \simeq \mathcal{U} A) &\equiv \text{cong } \llbracket \mathcal{U} \ni \_ \rrbracket (\text{Inj } \_ \_ (\hat{\Sigma} \text{Idl} \simeq \mathcal{U} A)) \\ \llbracket \hat{\Sigma} \text{Idr} \rrbracket \mathcal{U} A : \text{ua } (\hat{\Sigma} \text{Idr} \simeq \mathcal{U} A) &\equiv \text{cong } \llbracket \mathcal{U} \ni \_ \rrbracket (\text{Inj } \_ \_ (\hat{\Sigma} \text{Idr} \simeq \mathcal{U} A)) \\ \llbracket \hat{\Sigma} \text{Assoc} \rrbracket \mathcal{U} A B C : \text{ua } (\hat{\Sigma} \text{Assoc} \simeq \mathcal{U} A B C) &\equiv \text{cong } \llbracket \mathcal{U} \ni \_ \rrbracket (\text{Inj } \_ \_ (\hat{\Sigma} \text{Assoc} \simeq \mathcal{U} A B C)) \end{aligned}$$

Collecting the fields together, we obtain the following record type.

► **Definition 9** (Generalised operad universe). *A generalised operad universe is a record $\mathcal{U} : \text{Universe}$ comprising a type of codes $\text{Code } \mathcal{U}$ and an interpretation family $\llbracket \mathcal{U} \ni \_ \rrbracket$; a code $\hat{\top}$ for the unit together with an equivalence $\llbracket \hat{\top} \rrbracket$; a code former $\hat{\Sigma}$ for dependent sums together with an equivalence $\llbracket \hat{\Sigma} \rrbracket$; a representation map Inj satisfying the 1-coherence InjComp ; and three path-level closure laws $\llbracket \hat{\Sigma} \text{Idl} \rrbracket$, $\llbracket \hat{\Sigma} \text{Idr} \rrbracket$ and $\llbracket \hat{\Sigma} \text{Assoc} \rrbracket$, as described above.*

Both the universe of totally-ordered finite sets and the groupoid of finite sets and bijections satisfy all the above criteria. Another example of a universe for which all the above terms can be constructed, is the universe Type itself. Similarly, for any homotopy-level $n \geq -1$, i.e. propositions or higher, the universe of n -types also satisfies these criteria.

We note that our construction does not require that the type of codes $\text{Code } \mathcal{U}$ is a 1-type. Since the operations of an operad will form a family $K : \text{Code } \mathcal{U} \rightarrow \text{hSet}$, and hSet is itself a 1-type, every such family factors through the groupoid truncation $\llbracket \text{Code } \mathcal{U} \rrbracket_1$. Equivalently, families of h-sets over $\text{Code } \mathcal{U}$ are equivalent to families of h-sets over its 1-truncation, so our

theory of operads never inspects any higher path structure of $\text{Code } \mathcal{U}$ beyond its 1-groupoid truncation. Accordingly, it suffices to require that Inj preserves 1-groupoid structure as witnessed by InjComp , together with a proof of closure under dependent sums and a unit type, alongside the 1-coherences $\llbracket \hat{\Sigma} \text{Idl} \rrbracket$, $\llbracket \hat{\Sigma} \text{Idr} \rrbracket$ and $\llbracket \hat{\Sigma} \text{Assoc} \rrbracket$. The resulting notion of a generalised operad universe is closely related to the univalent universes of Christensen [11], and in particular the univalent universe of finite types studied by Carette and Sabry [9] and Choudhury, Karwowski and Sabry [10].

From this characterisation of universes closed under both dependent sums and a unit type, our internalisation of operads can be generalised by parametrising over these universes.

► **Definition 10** ($\mathcal{U}$ -operad). *Given a generalised operad universe $\mathcal{U}$ and a family of h-sets $K : \text{Code } \mathcal{U} \rightarrow \text{hSet}$, a $\mathcal{U}$ -operad on K is a term of a parameterised record type $\text{Operad } \mathcal{U} K : \text{Type}$ whose fields are obtained from Definition 1 by replacing finite summation sum and dependent sum over finite sets by the universe-supplied $\hat{\Sigma}$, and replacing the indexing $\mathbb{N}$ or FinSet by $\text{Code } \mathcal{U}$.*

Given such an operadic structure $O : \text{Operad } \mathcal{U} K$, its fields are almost identical to those detailed in Sections 4 and 5, with finite summation or dependent sum over finite sets replaced by $\hat{\Sigma}$. For example, for every $A : \text{Code } \mathcal{U}$ and $B : \llbracket \mathcal{U} \ni A \rrbracket \rightarrow \text{Code } \mathcal{U}$ the composition map for the operad O is given by the following field:

$$\text{comp } O A B : K A \rightarrow ((a : \llbracket \mathcal{U} \ni A \rrbracket) \rightarrow K (B a)) \rightarrow K (\hat{\Sigma} A B)$$

Similarly, the unit operation is given by a field $\text{id } O : K \hat{\top}$. The families of heterogeneous paths witnessing the left/right unit laws and the associativity law can be defined by appropriately applying Inj to the respective laws on dependent sums. For example, for every $A : \text{Code } \mathcal{U}$ and $k : K A$ the corresponding left unit law for O is witnessed by the following field:

$$\begin{aligned} \text{idl } O A k : \text{PathP} \\ (\lambda i \rightarrow K (\text{Inj } (\hat{\Sigma} \hat{\top} (\lambda u \rightarrow A)) A (\hat{\Sigma} \text{Idl } \llbracket \mathcal{U} \ni A \rrbracket) i)) \\ (\text{comp } O \hat{\top} (\lambda u \rightarrow A) (\text{id } O) (\lambda u \rightarrow k)) k \end{aligned}$$

To conclude our discussion on generalised operads, we describe the groupoid structure on $\mathcal{U}$ -operads, which arises from the path space on the type of $\mathcal{U}$ -operads. In HoTT, given any type A and family of h-propositions $P : A \rightarrow \text{Type}$, the sum type $\sum [a \in A] P a$ can be considered a *subtype* of A , as witnessed by the first projection $\pi_1 : \sum [a \in A] P a \rightarrow A$ being an embedding. Concretely, this means that the functorial action of the first projection map on paths, i.e. $\text{cong } \pi_1 : x \equiv y \rightarrow \pi_1 x \equiv \pi_1 y$, is an equivalence. As such, the path space of a subtype is characterised by the path space of its underlying type. Given a collection of operations $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$, where K is a family of h-sets and is equipped with an operad structure $O : \text{Operad } \mathcal{U} K$, the types of each of the paths $\text{idl } O n k$, $\text{idr } O n k$ and $\text{assoc } O n ns nss k ks kss$ are then h-propositions. Hence, an operad can be understood as a subtype of a record type with only an identity id and composition map comp . That is, the operad laws are not important in characterising the path spaces between operads, and we make use of this fact to simplify the following definition of these path spaces.

Given any two collections of operations $K L : \text{Code } \mathcal{U} \rightarrow \text{Type}$, where K and L are families of h-sets with operad structures $O^K : \text{Operad } \mathcal{U} K$ and $O^L : \text{Operad } \mathcal{U} L$, and given any family of paths $K \equiv L : (A : \text{Code } \mathcal{U}) \rightarrow K A \equiv L A$, the heterogeneous path space between O^K and O^L is equivalent to a pair of heterogeneous paths ranging over $K \equiv L$ between the identities and composition maps of O^K and O^L . Importantly, it follows from K and L being families of h-sets that this pair is an h-proposition.

► **Proposition 11** (Groupoid structure on the type of $\mathcal{U}$ -operads). *For any universe $\mathcal{U}$ and any two $\mathcal{U}$ -operads (K, O^K) and (L, O^L) , the heterogeneous path space between O^K and O^L over a family of paths $K \equiv L$ is an h -proposition. Consequently, the type $\sum [K \in (\text{Code } \mathcal{U} \rightarrow \text{hSet})] \text{Operad } \mathcal{U} K$ of all $\mathcal{U}$ -operads carries a 1-groupoid structure induced by the group structure on path spaces.*

We conclude with some remarks on the utility of our generalised operad construction. In particular, by developing a generalised formulation of operads, we can often introduce further operadic ideas by parameterising over a generalised operad universe. This allows us to define concepts such as operad morphisms, as well as algebras and monads over an operad, each of which are uniform over the universe, without specialising for the planar and symmetric cases. Indeed, as we show in Section 9, even the free operad construction can be parameterised over the universe of codes. This highlights one of the key benefits of studying operads in HoTT, namely that the functoriality of type families with respect to paths enables a generalised notion of an operad that supports uniform definitions for many constructions. For example, as shown in our formalisation, the *associative*, *commutative* and endomorphism operads can all be defined uniformly over generalised operad universes, such that specialising to totally ordered finite sets or Bishop-finite sets gives the expected planar and symmetric versions.

7 Category of Operads

To work with operads in HoTT, we require both structure-preserving maps between them and concrete interpretations of the abstract collections of operations that they represent. These correspond, respectively, to operad homomorphisms and operad algebras. A concrete example of both is the `fill` function from Section 4, which can be understood as a morphism from the `PartialList` operad to the *endomorphism operad* on lists. In this section, we begin by presenting the categorical structure of operads within HoTT. We then introduce the endomorphism operad on a given type, whereby morphisms in the category of operads into the endomorphism operad constitute the notion of an operad algebra.

In addition to the groupoid structure described in the previous section, the (large) type $\sum [K \in (\text{Code } \mathcal{U} \rightarrow \text{hSet})] \text{Operad } \mathcal{U} K$ has a more general category structure where a morphism between operads is any family of maps between their operations that preserves composition and identity.

► **Definition 12** (Operad morphism and the category of $\mathcal{U}$ -operads). *Given operads (K, O^K) and (L, O^L) , an operad morphism $f : O^K \Rightarrow O^L$ is a term of a record type with a projection $\langle f \rangle : (A : \text{Code } \mathcal{U}) \rightarrow K A \rightarrow L A$ and proofs that f respects operad identity and composition. The category of $\mathcal{U}$ -operads has $\mathcal{U}$ -operads as objects and operad morphisms as arrows.*

As with the path space between operads, this type is a subtype of the type of slice morphisms $(A : \text{Code } \mathcal{U}) \rightarrow K A \rightarrow L A$, with the identity and composition laws as h -propositional fields. This yields the following characterisation of equality.

► **Proposition 13** (Equality of operad morphisms). *For any two operad morphisms $f, g : O^K \Rightarrow O^L$, the function `cong` $\langle \_ \rangle : f \equiv g \rightarrow \langle f \rangle \equiv \langle g \rangle$ is an equivalence. That is, operad morphisms are equal if they agree on operations.*

The composition of operad morphisms is given by composing their underlying slice morphisms. In particular, given a third operad (J, O^J) and operad morphisms $f : O^J \Rightarrow O^K$ and $g : O^K \Rightarrow O^L$ we can construct the operations of their composition $\langle g \bullet f \rangle : O^J \Rightarrow O^L$ as $\langle g \bullet f \rangle A k = \langle g \rangle A (\langle f \rangle A k)$. The composition of slice morphisms respects both

operad identity and composition structure. The identity $\text{id}^{op} O : O \Rightarrow O$ on $O : \text{Operad } \mathcal{U} K$ is defined by $\langle O \rangle A k = k$, with both the proof of identity and composition preservation holding definitionally. Associativity and unitality of operad morphism composition then follow from the corresponding laws on slice morphisms via Proposition 13.

We have already seen two examples of planar operad morphisms in Section 4, namely $\llbracket \_ \rrbracket : \text{IExpr } n \rightarrow (\text{Fin } n \rightarrow \text{Expr}) \rightarrow \text{Expr}$ and $\text{fill} : \text{PartialList } A n \rightarrow (\text{Fin } n \rightarrow \text{List } A) \rightarrow \text{List } A$. Both are operad morphisms into an *endomorphism operad*, an operad whose operations are “ n -ary like” functions on an h-set and whose composition map is function composition.

► **Definition 14** (Endomorphism operad $\text{Endo } \mathcal{U} X$). *For any h-set $X : \text{Type}$, the collection of operations $\text{EndoOps} : \text{Code } \mathcal{U} \rightarrow \text{Type}$ given by $\text{EndoOps } A = (\llbracket \mathcal{U} \ni A \rrbracket \rightarrow X) \rightarrow X$ forms a $\mathcal{U}$ -operad structure $\text{Endo } \mathcal{U} X : \text{Operad } \mathcal{U} \text{EndoOps}$. We call this the endomorphism $\mathcal{U}$ -operad on X and operad composition is simply given by function composition.*

Notably, when specialising to the universe of totally-ordered finite sets, EndoOps corresponds to the family of n -ary functions on X . Given that X is an h-set, EndoOps is also a family of h-sets. The unit operation $\text{id} (\text{Endo } \mathcal{U} X) : (\llbracket \mathcal{U} \ni \hat{\top} \rrbracket \rightarrow X) \rightarrow X$ is given by:

$$\text{id} (\text{Endo } \mathcal{U} X) f = f (\text{inv} (\llbracket \hat{\top} \rrbracket \mathcal{U}) \text{tt})$$

Intuitively, the unit operation extracts the only element from a unitary collection $f : \llbracket \mathcal{U} \ni \hat{\top} \rrbracket \rightarrow X$. For every $A : \text{Code } \mathcal{U}$, $B : \llbracket \mathcal{U} \ni A \rrbracket \rightarrow \text{Code } \mathcal{U}$ together with an output operation $f : (\llbracket \mathcal{U} \ni A \rrbracket \rightarrow X) \rightarrow X$ and input operations $fs : (a : \llbracket \mathcal{U} \ni A \rrbracket) \rightarrow (\llbracket \mathcal{U} \ni B a \rrbracket \rightarrow X) \rightarrow X$, we define the composition map of the endomorphism operad on X as follows:

$$\text{comp} (\text{Endo } \mathcal{U} X) A B f fs xs = f (\lambda a \rightarrow fs a (\lambda b \rightarrow xs (\text{inv} (\llbracket \hat{\Sigma} \rrbracket \mathcal{U} A B) (a, b))))$$

To construct proofs of the identity and associativity laws, we first explain how the univalence map from equivalences to paths, i.e. $\text{ua} : X \simeq Y \rightarrow X \equiv Y$, acts when appearing twice to the left of the function arrow. Concretely, for any types $X_1 X_2 Y Z : \text{Type}$, equivalence $e : X_1 \simeq X_2$ and function $f : (X_1 \rightarrow Y) \rightarrow Z$, the action of $\text{ua } e : X_1 \equiv X_2$ on f is witnessed by a heterogeneous path,

$$\begin{aligned} \text{ua} \rightarrow f e : \text{PathP } (\lambda i \rightarrow (\text{ua } e i \rightarrow Y) \rightarrow Z) f (\lambda ys \rightarrow f (ys \circ \text{fun } e)) \\ \text{ua} \rightarrow f e i ys = f (ys \circ \text{ua-gluePt } e i) \end{aligned}$$

where $\text{ua-gluePt } e i : X_1 \rightarrow \text{ua } e i$ is a path from the identity function to $\text{fun } e$ and is a standard result. While we do not give the explicit constructions here, the proofs that the composition map of the endomorphism operad respects the identity and associativity laws proceed by application of $\text{ua} \rightarrow$ to the equivalences $\hat{\Sigma} \text{Idl} \simeq A$, $\hat{\Sigma} \text{Idr} \simeq A$ and $\hat{\Sigma} \text{Assoc} \simeq A B C$ followed by substitution along the paths $\llbracket \hat{\Sigma} \text{Idl} \rrbracket$, $\llbracket \hat{\Sigma} \text{Idr} \rrbracket$ and $\llbracket \hat{\Sigma} \text{Assoc} \rrbracket$ respectively.

8 Monad over an Operad

When using operadic collections of operations, it is often useful to dependently build operations from data. Indeed, this is the behaviour of the canonical monad that arises over any operad. In particular, an element of this monad is an operation together with data stored at each input. Concretely, for any universe $\mathcal{U} : \text{Universe}$ and collection of operations $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$, where K is a family of h-sets, the monad over any $\mathcal{U}$ -operad can be internalised in Cubical Agda by first introducing the following record type.

► **Definition 15** (The carrier $\text{OpM } O \ X$). For an operad $O : \text{Operad } \mathcal{U} \ K$ and a type $X : \text{Type}$, the type $\text{OpM } O \ X$ has terms given by the following record type:

```
record OpM (O : Operad U K) (X : Type) : Type where
  constructor _▷_▷_
  field
    Index : Code U
    Op : K Index
    Data : [U ⊃ Index] → X
```

When the type of codes for the universe $\mathcal{U}$ is an h-groupoid, then for any operad $O : \text{Operad } \mathcal{U} \ K$ the family $\text{OpM } O$ restricts to an endofunctor on the category of h-sets and gives rise to a monad on h-sets in the usual sense, with the monadic laws holding as paths. Restricted to h-groupoids, the same construction extends to a 2-monad on the 2-category of h-groupoids, functions and paths, whose 2-cell coherence is provided in our Cubical Agda formalisation. Throughout the rest of this section, we refer to either structure simply as the monad over the operad O . The unit map for this monad corresponds to storing an element at the output of the unit operation, and can be defined as follows:

```
return : X → OpM O X
return x = ↑ U ▷ unit O ▷ λ t → x
```

The composition map for this monad describes how given an output operation whose leaf data corresponds to input operations with their own data, we can use the operadic composition map to compose the output and input operations while retaining the data attached to the inputs. We define a dependent variation of this map whereby for every term $ox : \text{OpM } O \ X$ and function $([U ⊃ \text{Index } ox] \rightarrow X \rightarrow \text{OpM } O \ Y)$, we construct a term $\text{compM } ox \ f : \text{OpM } O \ Y$ as follows, where $\text{next } ox \ f = \lambda i \rightarrow f \ i \ (\text{Data } ox \ i)$:

```
Index (compM ox f) = ∑ U (Index o next ox f)
Op (compM ox f) = comp O (Index ox) (Index o next ox f) (Op ox) (Op o next ox f)
Data (compM ox f) k
  = let (i, j) = fun ([∑ U (Index ox) (Index o next ox f)]) k
    in Data (f i (Data ox i)) j
```

From this definition of compM , we can define the usual monad multiplication map, $\text{join} : \text{OpM } O \ (\text{OpM } O \ X) \rightarrow \text{OpM } O \ X$, by simply mapping each $o : \text{OpM } O \ (\text{OpM } O \ X)$ to $\text{compM } o \ (\lambda i \ x \rightarrow x)$. The functorial map $\_<\$>\_ : (X \rightarrow Y) \rightarrow \text{OpM } O \ X \rightarrow \text{OpM } O \ Y$ can also be defined using compM , however, this results in an index of $\sum U (\text{Index } o) (\lambda i \rightarrow \uparrow U)$, while the obvious direct definition has the simpler index $\text{Index } o$. Moreover, by using the more direct definition of the functorial map, the usual functor laws hold definitionally, as required. The proofs that return and join are natural transformations similarly hold by definition.

To describe the monadic structure of $\text{OpM } O$, we also require proofs of the coherence laws. We begin by giving an equivalent characterisation of the path space on $\text{OpM } O \ X$ for any $X : \text{Type}$, which we use to construct the coherence witnesses. In particular, for every $x \ y : \text{OpM } O \ X$, the paths between x and y are trivially equivalent to the triples of a path $p : \text{Index } x \equiv \text{Index } y$ on the indices, together with a heterogeneous path $\text{PathP } (\lambda i \rightarrow K (p \ i)) (\text{Op } x) (\text{Op } y)$ on the operations and a heterogeneous path $\text{PathP } (\lambda i \rightarrow [U ⊃ p \ i] \rightarrow X) (\text{Data } x) (\text{Data } y)$ on the data. Moreover, for constructing the heterogeneous paths between the Data fields, we note that given any types $X \ Y : \text{Type}$, functions $f : X \rightarrow Y$, $g : Y \rightarrow X$ and equivalence $e : X \simeq Y$, there is an equivalence between the heterogeneous path space $\text{PathP } (\lambda i \rightarrow \text{ua } e \ i \rightarrow X) f \ g$ and families of paths $(x : X) \rightarrow f \ x \equiv g \ (\text{fun } e \ x)$.

► **Theorem 16** (*OpM* O is a monad on $\mathbf{h}\text{-sets}$). For any operad $O : \mathbf{Operad} \mathcal{U} K$, the assignment $X \mapsto \mathbf{OpM} O X$ with unit *return*, multiplication *join* and functorial map $\langle \$ \rangle$ is a monad on the category of $\mathbf{h}\text{-sets}$, with the monad laws witnessed by paths *join-return*₁, *join-return*₂ and *join-assoc*.

Proof. We proceed by giving an overview of the path constructions witnessing the monadic laws; the details can be found in our formalisation. The required paths on indices are $\hat{\Sigma}\text{Idl}$, $\hat{\Sigma}\text{Idr}$ and $\hat{\Sigma}\text{Assoc}$, and the corresponding heterogeneous paths on operations are then obtained from the operad laws *idl* O , *idr* O and *assoc* O . To construct the heterogeneous paths on data, we first substitute in the family $(\lambda p \rightarrow \text{PathP } (\lambda i \rightarrow p i \rightarrow X))$ over the path interpretations $\llbracket \hat{\Sigma}\text{Idl} \rrbracket$, $\llbracket \hat{\Sigma}\text{Idr} \rrbracket$ and $\llbracket \hat{\Sigma}\text{Assoc} \rrbracket$. From our earlier characterisation of heterogeneous paths of the form $\text{PathP } (\lambda i \rightarrow \text{ua } e i \rightarrow X) f g$, it then suffices to construct families of paths $(x : X) \rightarrow f x \equiv g (\text{fun } e x)$ for each monadic law. The identity laws hold definitionally, while associativity is proved in our formalisation. ◀

For any operad $O : \mathbf{Operad} \mathcal{U} K$, the algebras over the monad $\mathbf{OpM} O$, or simply *the algebras over* O , describe how the abstract collection of operations represented by O can be interpreted as concrete operations on a carrier type.

► **Definition 17** (*Operad algebra*). An algebra over an operad $O : \mathbf{Operad} \mathcal{U} K$ is a pair of a carrier $\mathbf{h}\text{-set}$ $A : \mathbf{Type}$ together with an operad morphism $\alpha : O \Rightarrow \mathbf{Endo} \mathcal{U} A$ from O to the endomorphism operad on A (Definition 14).

Examples of operad algebras include both the function $\llbracket \_ \rrbracket : \mathbf{IExpr} n \rightarrow (\mathbf{Fin} n \rightarrow \mathbf{Expr}) \rightarrow \mathbf{Expr}$ with carrier $\mathbf{Expr}$ and the function $\text{fill} : \mathbf{PartialList} A n \rightarrow (\mathbf{Fin} n \rightarrow \mathbf{List} A) \rightarrow \mathbf{List} A$ with carrier $\mathbf{List} A$. This notion of an operad algebra gives rise to an interpretation function $\text{runAlg} : (O \Rightarrow \mathbf{Endo} \mathcal{U} A) \rightarrow \mathbf{OpM} O A \rightarrow A$ defined simply as $\text{runAlg } \alpha o = \langle \alpha \rangle (\text{Index } o) (\text{Op } o) (\text{Data } o)$.

► **Example 18** (*select* and *mapWhere* via the partial-list operad). To conclude our discussion of monads over an operad, we provide a concrete example of using the monad over the partial list operad. We begin by letting $\hat{\mathbb{N}} : \mathbf{Universe}$ be the generalised operad universe of totally ordered finite sets, i.e. $\text{Code } \hat{\mathbb{N}} = \mathbb{N}$ and $\llbracket \hat{\mathbb{N}} \ni n \rrbracket = \mathbf{Fin} n$, and $\mathbf{PList} A : \mathbf{Operad} \hat{\mathbb{N}} (\mathbf{PartialList} A)$ be the partial list operad on an $\mathbf{h}\text{-set}$ A . We then observe that we can define monadic liftings of the $\_::\_$ and *poke* constructors. In particular, for every $\mathbf{h}\text{-groupoid}$ $X : \mathbf{Type}$ we construct the monadic lifting of $\_::\_$ as follows:

```
consM : A → OpM (PList A) X → OpM (PList A) X
consM a o = Index o ▷ (a :: Op o) ▷ Data o
```

We similarly define the monadic lifting of *poke* by:

```
pokeM : X → OpM (PList A) X → OpM (PList A) X
pokeM x o = suc (Index o) ▷ poke (Op o) ▷ λ { zero → x ; (suc i) → Data o i }
```

Using these two functions we can define a function that extracts elements from a list that satisfy a predicate, and preserve the original list with holes in place of these elements:

```
select : (A → Bool) → List A → OpM (PList A) A
select p [] = 0 ▷ [] ▷ λ ()
select p (a :: as) = if p a then pokeM a (select p as) else consM a (select p as)
```

Intuitively, `select` highlights certain elements of a list, allowing us to modify them while leaving the rest intact. Indeed, if we let $\text{fill} \Rightarrow : \mathbf{PList} A \Rightarrow \mathbf{Endo} \mathcal{U} (\mathbf{List} A)$ be the operad morphism with $\langle \text{fill} \Rightarrow \rangle = \text{fill}$, then for every $p : A \rightarrow \mathbf{Bool}$ and $xs : \mathbf{List} A$ we can construct a path of type $\text{runAlg fill} \Rightarrow (\llbracket \_ \rrbracket <\$> \text{select } p \, xs) \equiv xs$, where $\llbracket \_ \rrbracket$ constructs a singleton list. That is, selecting elements and then putting them back unchanged gives the original list.

Building on `select`, we obtain a function that applies a given transformation to every element satisfying a predicate while leaving the rest of the list unchanged:

```
mapWhere : (A → Bool) → (A → A) → List A → List A
mapWhere p f xs = runAlg fill ⇒ ((f ∘  $\llbracket \_ \rrbracket$ ) <$> select p xs)
```

More generally, the monad over an operad gives a structured way to build effectful traversals that respect a chosen compositional structure on the underlying datatype: in this case, the partial-list operad structure on $\mathbf{List} A$.

9 Free Operad

We have seen how a generalised form of operads that encompasses both planar and symmetric operads can be presented in HoTT. Using this framework, we now formalise one of the key operadic constructions, namely the *free* operad on a collection of operations. In particular, this is the left adjoint construction to the forgetful functor from operads to their underlying operations. Intuitively, the operations of free operads correspond to trees with nodes labelled by elements of a collection of operations depending on their arity, where composition is given by grafting of trees. Free operads allow us to construct operation trees independently from the choice of how to compose the underlying operations. In this section, we will give an account of the free construction for generalised operads in HoTT, together with some important properties and examples.

We begin by introducing the notion of a free planar operad on a countable family of h-sets $K : \mathbb{N} \rightarrow \mathbf{Type}$, and then show how the construction can be generalised. The operations of the free planar operad on K are characterised by planar, finitely-branching trees in which nodes with n -input vertices are labelled by an element of $K \, n$. We also require a unit tree with a single input and output vertex that acts as a left and right unit for tree composition.

For readers with experience of *rose trees*, the structure of the free planar operad may seem familiar. To recall, rose trees are planar finitely-branching trees, and are typically labelled with elements of a parameterised type. We can define rose trees as follows:

```
data RoseTree (A : Type) : Type where
  leaf : RoseTree A
  node : (n : ℕ) → A → (Fin n → RoseTree A) → RoseTree A
```

We can define a function $\text{count} : \mathbf{RoseTree} A \rightarrow \mathbb{N}$ that counts leaves in a rose tree. The fibers of this function, i.e. the countable family of types mapping $n : \mathbb{N}$ to the type $\sum [t \in \mathbf{RoseTree} A] \text{count } t \equiv n$ of rose trees with n leaves, has an operad structure given by tree composition with unit `leaf`. When A is the unit type, the induced operad is known as the *planar tree operad*. A generalisation of this construction involves labelling nodes with elements from a countable family of h-sets $K : \mathbb{N} \rightarrow \mathbf{Type}$, where a node with n -inputs is labelled with an element of $K \, n : \mathbf{Type}$.

► **Definition 19** (Free planar operad on a countable family). *For any countable family $K : \mathbb{N} \rightarrow \text{Type}$, the underlying operations of the free planar operad on K are given by the following inductive family:*

```
data FreePOps (K : ℕ → Type) : ℕ → Type where
  unit : FreePOps K 1
  comp : (ns : Fin n → ℕ) → K n → ((∀ i → FreePOps K (ns i)) → FreePOps K (Σ n ns))
```

This definition might at first appear rather different from that of rose trees. However, this is a consequence of indexing our inductive definition over the number of leaves rather than defining a separate count function whose fibers are equivalent to $\text{FreePOps } K$.

As expected, our inductive construction for the operations of the free planar operad comes equipped with a planar operadic structure with the identity operation given by $\text{unit} : \text{FreePOps } K \ 1$. While we do not give an account of the full operadic structure on FreePOps here, it follows as a specific case of the more general free operad construction. In particular, given a universe $\mathcal{U} : \text{Universe}$, a first attempt at defining the operations of the free $\mathcal{U}$ -operad is the following translation of FreePOps to the generalised case:

```
data FreeOps (K : Code U → Type) : Code U → Type where
  leaf : FreeOps K (↑ U)
  node : (A : Code U) (B : [U ⊃ A] → Code U) →
    K A → (∀ a → FreeOps K (B a)) → FreeOps K (Σ̂ U A B)
```

However, this definition of FreeOps is not a sufficient characterisation of the free operad as it is not always a family of h-sets. This is not immediately evident as although node parametrises over a type of codes $\text{Code } \mathcal{U}$ that may not be an h-set, i.e. FinSet , these codes are then constrained by the output index of the node constructor. Indeed, it is possible to show that the assumption that $\text{FreeOps } K \ A$ is an h-set for every family of h-sets $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$ and every code $A : \text{Code } \mathcal{U}$ leads to a contradiction. In particular, for any universe $\mathcal{U}$ with an element $\hat{1} : \text{Code } \mathcal{U}$ such that the interpretation of $\hat{1}$ is the empty type, there is a retraction from $\text{FreeOps } K \ \hat{1}$ to $\text{Code } \mathcal{U}$. In turn, because retractions preserve h-levels, when $\text{Code } \mathcal{U}$ is not an h-set as in the case of FinSet , the assumption that $\text{FreeOps } K \ \hat{1}$ is an h-set yields a contradiction. More generally, for any $n \geq 0$, if $\text{Code } \mathcal{U}$ is an n -type then $\text{FreeOps } K \ A$ is at most an n -type. The set-truncation path constructor we introduce below is therefore only required when $\text{Code } \mathcal{U}$ has higher path structure than an h-set.

As a consequence of FreeOps not being a family of h-sets, it is necessary to add a set-truncation path constructor

```
set : (A : Code U) → isSet (FreeOps K A)
```

to our inductive definition. Notably, this is not the same as defining a new family of types which maps each $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$ and $A : \text{Code } \mathcal{U}$ to $\| \text{FreeOps } K \ A \|_0$, i.e. the set or 0-truncation of $\text{FreeOps } K \ A$. Moreover, in the case where the type of codes for the considered universe is an h-set, such as for the universe of totally-ordered finite sets, then the versions of the FreeOps with and without the set path constructor are equivalent.

By presenting the operations of the free operad on K as the higher-inductive family $\text{FreeOps } K$, the operad composition map requires explicit substitutions along paths on codes. In practice, this can lead to “transport hell”, whereby operations built by composition are nested substitutions that cannot be unfolded in further computations. This is a notable issue when we are simply interested in computing along the tree structure. In the presence of (small) induction-recursion, this can be addressed by an alternative encoding that separates the shape of a tree from the indexing of its leaves.

Our alternative encoding uses the known equivalence between inductive families and small induction-recursion [17]. However, this equivalence has not been extended to higher-inductive families and higher small induction-recursion. To highlight this issue, we first consider a naive translation of only the data constructors of the higher inductive family $\text{FreeOps } K$ to a small inductive-recursive definition.

► **Definition 20** (Inductive-recursive shape FreeOpsIR and its decoder CodeOp). *For every family $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$, the type $\text{FreeOpsIR } K : \text{Type}$ and the recursive function $\text{CodeOp } K : \text{FreeOpsIR } K \rightarrow \text{Code } \mathcal{U}$ are defined mutually as follows:*

```
data FreeOpsIR K : Type where
  leaf : FreeOpsIR K
  node : (A : Code U) → K A → (∥ U ⊃ A ∥ → FreeOpsIR K) → FreeOpsIR K

CodeOp K leaf = ↑ U
CodeOp K (node A k ts) = ∑ U A (CodeOp ∘ ts)
```

In particular, there is an equivalence between fibers of $\text{CodeOp } K$ and the inductive family $\text{FreeOps } K$ without its set path constructor. However, as the type of codes $\text{Code } \mathcal{U}$ may not be an h-set, the fibers of $\text{CodeOp } K$ cannot always be shown to be h-sets and are hence not equivalent to the higher-inductive family $\text{FreeOps } K$.

In the absence of the axiom of choice, it is not sufficient to simply consider the set truncated fibers over $\text{CodeOp } K$, i.e. the family mapping each code $A : \text{Code } \mathcal{U}$ to the 0-truncated type $\| \sum [t \in \text{FreeOpsIR } K] \text{CodeOp } K t \equiv A \|_0$. Instead, we require the addition of an appropriate higher path constructor to $\text{FreeOpsIR } K$, together with the action of $\text{CodeOp } K$ on this constructor, to ensure that the fibers of $\text{CodeOp } K$ are h-sets. To do this, we begin by noting that for any types $X, Y : \text{Type}$ and function $f : X \rightarrow Y$, the proposition that the fibers of f are all h-sets is equivalent to the proposition that the functorial action of f on paths is an embedding, i.e. for all terms $x_1, x_2 : X$ and paths $p, q : x_1 \equiv x_2$, the function $\text{cong} (\text{cong } f) : p \equiv q \rightarrow \text{cong } f p \equiv \text{cong } f q$ is an equivalence. Indeed, this embedding is precisely what we will aim to establish by adding a path constructor to $\text{FreeOpsIR } K$.

To construct this embedding, it suffices to show that $\text{cong} (\text{cong} (\text{CodeOp } K)) : p \equiv q \rightarrow \text{cong} (\text{CodeOp } K) p \equiv \text{cong} (\text{CodeOp } K) q$ is an isomorphism. We can introduce an equivalent isomorphism into FreeOpsIR using a path constructor whose type is almost identical to that of $\text{cong} (\text{cong} (\text{CodeOp } K))$. However, to ensure FreeOpsIR only appears strictly positively, we first flip the resulting square along its diagonal. Concretely, we introduce a path constructor that for each $t, u : \text{FreeOpsIR}$ and $p, q : t \equiv u$, constructs the following path:

```
set p q : PathP (λ i → CodeOp K (p i) ≡ CodeOp K (q i)) refl refl → p ≡ q
```

Interestingly, it is possible to show that the $\text{set } p, q$ path constructor is both a section and retraction of $\text{cong} (\text{cong} (\text{CodeOp } K))$, which yields the following.

► **Proposition 21** (Fibers of $\text{CodeOp } K$ are h-sets). *For every universe $\mathcal{U}$ and family $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$, the functorial action of $\text{CodeOp } K$ on paths is an embedding. Equivalently, for every code $A : \text{Code } \mathcal{U}$ the fiber $\text{fiber} (\text{CodeOp } K) A$ is an h-set.*

Proof. This is a generalisation of *univalent inductive-recursive types*. A univalent inductive-recursive definition is one in which the mutually defined recursive function is forced to have propositional fibers by adding a path constructor that makes its functorial action on paths an embedding; the resulting type behaves like a Tarski universe whose decoding map reflects

path structure. Our setting generalises this from propositional fibers to set-level fibers, which is what is required when the codomain $\text{Code } \mathcal{U}$ is itself a groupoid rather than a set. The details of this proof can be found in our Cubical Agda formalisation. $\blacktriangleleft$

Notably, we can always eliminate terms of $\text{FreeOpsIR } K$ into any h-set whereby the path constructor set is necessarily respected. Importantly, this includes eliminating into the fibers of the function $\text{CodeOp } K$ which is necessary to define the composition map for the free operad. To construct this, we first introduce the following two required families of paths:

$$\begin{aligned} \hat{\Sigma} \text{IdID } \mathcal{U} \, X : X \, (\text{inv } (\llbracket \hat{\top} \rrbracket \mathcal{U}) \, \text{tt}) &\equiv \hat{\Sigma} \mathcal{U} \, (\hat{\top} \mathcal{U}) \, X \\ \hat{\Sigma} \text{AssocD } \mathcal{U} \, A \, B \, C : \\ \hat{\Sigma} \mathcal{U} \, A \, (\lambda a \rightarrow \hat{\Sigma} \mathcal{U} \, (B \, a) \, \lambda b \rightarrow C \, (\text{inv } (\llbracket \hat{\Sigma} \rrbracket \mathcal{U} \, A \, B) \, (a \, , \, b))) &\equiv \hat{\Sigma} \mathcal{U} \, (\hat{\Sigma} \mathcal{U} \, A \, B) \, C \end{aligned}$$

where $X : \llbracket \mathcal{U} \ni \hat{\top} \rrbracket \rightarrow \text{Code } \mathcal{U}$, $A : \text{Code } \mathcal{U}$, $B : \llbracket \mathcal{U} \ni A \rrbracket \rightarrow \text{Code } \mathcal{U}$ and $C : \llbracket \mathcal{U} \ni \hat{\Sigma} \mathcal{U} \, A \, B \rrbracket \rightarrow \text{Code } \mathcal{U}$. We do not give the full constructions for these paths here, but they follow from $\hat{\Sigma} \text{Id}$ and $\hat{\Sigma} \text{Assoc}$. We proceed by defining a variant of the operad composition map whose input operations are not indexed. For each tree $t : \text{FreeOpsIR } K$ and family of trees $ts : \llbracket \mathcal{U} \ni \text{CodeOp } K \, t \rrbracket \rightarrow \text{FreeOpsIR } K$ we construct the following term:

$$\begin{aligned} \text{graft } t \, ts &: \text{fiber } (\text{CodeOp } K) \, (\hat{\Sigma} \mathcal{U} \, (\text{CodeOp } K) \, (\text{CodeOp } K \circ ts)) \\ \text{graft leaf } ts &= ts \, (\text{inv } (\llbracket \hat{\top} \rrbracket \mathcal{U}) \, \text{tt}) \, , \, \hat{\Sigma} \text{IdID } \mathcal{U} \, (\text{CodeOp } K \circ ts) \, (\text{inv } (\llbracket \hat{\top} \rrbracket \mathcal{U}) \, \text{tt}) \\ \text{graft (node } A \, k \, ts) \, tss & \\ &= \text{let } iv = \text{inv } (\llbracket \hat{\Sigma} \rrbracket \mathcal{U} \, A \, (\text{CodeOp } K \circ ts)) \\ &\quad us = \text{graft } (ts \, a) \, \lambda b \rightarrow tss \, (iv \, (a \, , \, b)) \\ &\quad ac = \hat{\Sigma} \text{AssocD } \mathcal{U} \, (\text{CodeOp } K \circ ts) \, (\text{CodeOp } K \circ tss) \\ &\quad \text{in node } A \, k \, (\text{fst } \circ us) \, , \, (\lambda i \rightarrow \hat{\Sigma} \mathcal{U} \, A \, \lambda a \rightarrow \text{snd } (us \, a) \, i) \cdot ac \end{aligned}$$

We note that the action of graft on the set path constructor follows from the proof that fibers of $\text{CodeOp } K$ are h-sets. The composition map for the free operad follows from the definition of graft by reinserting the indexing of the input operations. The unit operation is given by the term leaf whereby $\text{CodeOp } K \, \text{leaf} \equiv \hat{\top} \mathcal{U}$ holds definitionally. Finally, while we do not provide the constructions for the paths witnessing the identity and associativity operad laws here, they can be found in our Cubical Agda formalisation [4] and involve significant manipulation of nested substitutions.

► **Definition 22** (Operad on $\text{fiber } (\text{CodeOp } K)$). *Given a $\mathcal{U}$ -species $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$, $\text{FreeOperad } K : \text{Operad } \mathcal{U} \, (\text{fiber } (\text{CodeOp } K))$ is the operad whose composition map $\text{comp } A \, B$ is defined by mapping every $(t, p) : \text{fiber } (\text{CodeOp } K) \, A$ to the transport of graft along p in the type $((a : \llbracket \mathcal{U} \ni A \rrbracket) \rightarrow \text{fiber } (\text{CodeOp } K) \, (B \, a)) \rightarrow \text{fiber } (\text{CodeOp } K) \, (\hat{\Sigma} \mathcal{U} \, A \, B)$.*

We have thus far described the operadic structure on the fibers of $\text{CodeOp } K$ for every $\mathcal{U}$ -species $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$, but have not yet shown it is the *free* operad on K . To do so, we first highlight the categories on which we will define the free adjunction. In particular, the objects of our underlying category are $\mathcal{U}$ -species which can be understood as functors from the h-groupoid structure of $\text{Code } \mathcal{U}$ to the category of h-sets and functions. Concretely, the functorial action of a $\mathcal{U}$ -species $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$ is given by path substitution in K , i.e. a path $p : A \equiv B$ is lifted to $\text{subst } K \, p : K \, A \rightarrow K \, B$. It is a standard result of path substitution that this satisfies the functorial laws. For any two $\mathcal{U}$ -species $K \, L : \text{Code } \mathcal{U} \rightarrow \text{Type}$, their hom is given by the type of natural transformations $(A : \text{Code}) \rightarrow K \, A \rightarrow L \, A$, where naturality follows from substitution commuting with slice morphisms. Finally, the target category for the free construction of a $\mathcal{U}$ -operad is precisely the category of $\mathcal{U}$ -operads from Section 7.

There is a forgetful functor from the category of $\mathcal{U}$ -operads to the category of $\mathcal{U}$ -species which simply forgets the operadic structure and acts similarly on operad morphisms by means of the projection $\langle \_ \rangle : (O^K \Rightarrow O^L) \rightarrow (A : \text{Code}) \rightarrow K A \rightarrow L A$. The free operad on a $\mathcal{U}$ -species $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$ is, up to a path, the operad constructed by the left-adjoint to this forgetful functor. As we will show, the left-adjoint is the functor mapping each $K : \text{Code } \mathcal{U} \rightarrow \text{Type}$ to the collection of operations $\text{fiber}(\text{CodeOp } K) : \text{Code } \mathcal{U} \rightarrow \text{Type}$ equipped with its previously described operadic structure. The functoriality of this construction follows from the universal property of being a left adjoint in the usual way. The first step in establishing that adjunction is to define the following unit natural transformation:

$$\begin{aligned} \eta : (A : \text{Code } \mathcal{U}) &\rightarrow K A \rightarrow \text{fiber}(\text{CodeOp } K) A \\ \eta A k &= \text{node } A k (\lambda a \rightarrow \text{leaf}) , \sum \text{Idr } \mathcal{U} A \end{aligned}$$

Moreover, we require that for every operad $L : \text{Code } \mathcal{U} \rightarrow \text{Type}$, $O : \text{Operad } \mathcal{U} L$, and every morphism of $\mathcal{U}$ -species $f : (A : \text{Code } \mathcal{U}) \rightarrow K A \rightarrow L A$, that we can construct an operad morphism from the operadic structure $\text{CodeOp } K$ to O . In particular, we begin by defining a family of morphisms $\text{interpret } O f : (A : \text{Code } \mathcal{U}) \rightarrow \text{fiber}(\text{CodeOp } K) A \rightarrow L A$ as follows:

$$\begin{aligned} \text{interpret } O f . (\uparrow \mathcal{U}) \text{ leaf} &= \text{id } O \\ \text{interpret } O f . (\sum \mathcal{U} A B) (\text{node } A B k ts) &= \text{comp } O A B (f A k) \lambda a \rightarrow \text{interpret } O f (B a) (ts a) \end{aligned}$$

Note that interpret definitionally respects the operad identity. The proof that it also respects operad composition and thus extends to a morphism between the operads $\text{FreeOperad } K$ and O requires manipulation of nested substitutions and is provided in our library.

► **Theorem 23** (Free–forgetful adjunction). *For every operad $O : \text{Operad } \mathcal{U} L$ and every morphism of $\mathcal{U}$ -species $f : (A : \text{Code } \mathcal{U}) \rightarrow K A \rightarrow L A$, the type*

$$\sum [g \in \text{FreeOperad } K] ((A : \text{Code } \mathcal{U}) (k : K A) \rightarrow \langle g \rangle A (\eta A k) \equiv f A k)$$

is contractible, with the base point given by the function $\text{interpret } O f$ extended to an operad map together with the proof that $\text{interpret } O f A (\eta A k) = f A k$ for all $A : \text{Code } \mathcal{U}$ and $k : K A$. Consequently, the FreeOperad construction is left adjoint to the forgetful functor U from $\mathcal{U}$ -operads to $\mathcal{U}$ -species.

Proof. The detailed construction is provided in our library. The category of $\mathcal{U}$ -species can be understood as a category of *signatures* over the universe of $\mathcal{U}$ -arities: an object specifies, for each code $A : \text{Code } \mathcal{U}$, the type of generating operations of that arity. The adjunction therefore exhibits FreeOperad as the canonical way to turn a signature of generators into an operad whose laws are freely generated. ◀

10 Related Work

Several generic representations of datatypes have been developed in the type theory literature. The idea of separating structure from data first arose in work on shapely types [18]. More recently, containers were developed to capture strictly positive types [1]. The more general presentation of indexed containers was later developed to capture inductive families [5]. Containers are closed under finite products and exponentials [6], and have a notion of derivative [3]. The theory of containers also captures the shapely types. In particular, shapely type constructors are precisely given by the extensions of discretely-finite containers.

A key aspect of our theory of internal operads is the notion of a finite set, and the choice of a suitable definition of finiteness. Such a choice only manifests in constructive mathematics, where various classically equivalent notions of finiteness are distinct. In this

paper we adopt the notion of finiteness used in previous work on internalising the theory of combinatorial species in HoTT [26], namely Bishop-finiteness. However, there has also been work on internalising “enumerated” types and Kuratowski finite sets in the framework of HoTT [13, 15]. Adopting a different notion of finiteness impacts both which collection of operations are definable as a species, and which proofs about operads can be internalised.

Combinatorial species were introduced to unify approaches for analysing generating functions of discrete structures [20]. Intuitively, the idea of a species arose from the idea of building a structure from a finite collection of labels. Our internalisation of operads in HoTT can be seen as an extension of previous work on internalising combinatorial species [26]. In particular, Yorgey internalises combinatorial species to capture specific classes of data types, analogous to shapely types and the theory of containers.

Coloured operads generalise ordinary operads by labelling the inputs and outputs of operations with elements from a set C , and present a well-behaved notion of composition for finitary C -typed operations [25]. Such coloured operads have been formalised and studied in Coq [14]. In this paper, we focus on the single-coloured case and pursue a different direction of generalisation: rather than varying the set of colours, we vary the groupoid indexing the operations of an operad. This approach provides a uniform way to capture different flavours of operads by encoding symmetries directly in the indexing groupoid.

The theory of operads originated in the field of algebraic topology [21, 8] for describing operations on iterated loop-spaces. Another direction generalising operads is the theory of opetopes and opetopic sets [7], which have also been considered in the context of type theory by [22]. The Baez-Dolan slicing construction describes how to construct n -opetopes as an iterated construction making use of the free coloured operad. Generalising our work to the coloured setting would allow us to internalise this construction as an alternative way to define opetopes in HoTT.

11 Conclusion and Further Work

In this paper we showed how operads can be internalised in HoTT, and how this gives rise to a generic calculus of operations. Notably, we showed how discretely finite containers, which represent inductive data types with constructors of finite arity, have a natural operadic interpretation. This offers an operation-centric view of inductive types, complementing the traditional data-centric perspective. Our results open up a new line of work on investigating properties and applications of operads from a type-theoretic viewpoint. From a practical perspective, operad algebras provide semantics for embedded domain-specific languages whose terms are finitary compositions over a chosen signature: the free operad gives the syntax, and operad-algebra morphisms give compositional interpreters. Interesting topics for further work include generalising from species and operads to the “coloured” counterparts, considering how the notion of finiteness that we adopt influences which algebraic structures are expressible as a family over finite sets, and dualising our ideas to co-operads.

References

- 1 Michael Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.
- 2 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science*, 342(1), 2005. doi:10.1016/j.tcs.2005.06.002.
- 3 Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of Containers. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 2003.

- 4 HoTT Operads Cubical Agda Library. <https://tinyurl.com/y5fbrzdb>, 2026.
- 5 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed Containers. *Journal of Functional Programming*, 25, 2015. doi:10.1017/S095679681500009X.
- 6 Thorsten Altenkirch, Paul Levy, and Sam Staton. Higher-Order Containers. In *Conference on Computability in Europe*. Springer, 2010.
- 7 John C Baez and James Dolan. Higher-Dimensional Algebra III: n-Categories and the Algebra of Opetopes. *Advances in Mathematics*, 135(2):145–206, 1998.
- 8 John Michael Boardman and Rainer M Vogt. *Homotopy Invariant Algebraic Structures on Topological Spaces*, volume 347. Springer, 2006.
- 9 Jacques Carette, Chao-Hong Chen, Vikraman Choudhury, and Amr Sabry. From Reversible Programs to Univalent Universes and Back. *Electronic Notes in Theoretical Computer Science*, 336:5–25, 2018. doi:10.1016/j.entcs.2018.03.013.
- 10 Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. Symmetries in Reversible Programming: From Symmetric Rig Groupoids to Reversible Programming Languages. *Proceedings of the ACM on Programming Languages*, 6(POPL), 2022. doi:10.1145/3498667.
- 11 J. Daniel Christensen. A Characterization of Univalent Fibrations, 2022.
- 12 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.TYPES.2015.5.
- 13 Thierry Coquand and Arnaud Spiwack. Constructively Finite? In *Contribuciones Científicas en Honor de Mirian Andrés Gómez*. Universidad de La Rioja, 2010.
- 14 Zachary Flores, Angelo Taranto, Eric Bond, and Yakir Forman. A Formalization of Operads in Coq, 2023. doi:10.48550/arXiv.2303.08894.
- 15 Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. Finite Sets in Homotopy Type Theory. In *International Conference on Certified Programs and Proofs*, 2018.
- 16 Håkon Robbestad Gylterud. Symmetric Containers. Master’s thesis, University of Oslo, 2011., 2011.
- 17 Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small Induction Recursion. In *International Conference on Typed Lambda Calculi and Applications*, pages 156–172. Springer, 2013. doi:10.1007/978-3-642-38946-7_13.
- 18 C. Barry Jay and J. Robin B. Cockett. Shapely Types and Shape Polymorphism. In *European Symposium on Programming*. Springer, 1994.
- 19 Philipp Joram and Niccolò Veltri. Data Types with Symmetries via Action Containers. In *30th International Conference on Types for Proofs and Programs (TYPES 2024)*, pages 6–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.TYPES.2024.6.
- 20 André Joyal. Une Théorie Combinatoire des Séries Formelles. *Advances in Mathematics*, 42(1), 1981.
- 21 J Peter May. *The Geometry of Iterated Loop Spaces*, volume 271. Springer, 2006.
- 22 Pierre-Louis Curien, Cédric Ho Thanh, and Samuel Mimram. Syntactic Approaches to Opetopes, 2019. arXiv:1903.05848.
- 23 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages*, 3(ICFP), 2019. doi:10.1145/3341691.
- 24 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath: A Computer-Checked Library of Univalent Mathematics. Available at <https://github.com/UniMath/UniMath>.
- 25 Donald Yau. *Colored Operads*, volume 170. American Mathematical Society, 2016.
- 26 Brent Abraham Yorgey. *Combinatorial Species and Labelled Structures*. PhD thesis, University of Pennsylvania, 2014.

A Graded Modal Type Theory for Pulse Schedules

Robin Adams  

Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Jean-Philippe Bernardy  

Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Lorenzo Perticone  

Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Jeremy Pope  

Chalmers University of Technology, Gothenburg, Sweden
University of Gothenburg, Sweden

Abstract

The operations to be performed by a quantum computer are almost invariably given in the form of a quantum circuit. In the final stage of compilation, a quantum circuit must be translated into the input signals accepted by the quantum hardware itself. For a quantum computer based on superconducting qubits, this will be a sequence of microwave control pulses to be sent to the various input channels. A *pulse schedule* gives a full specification for which pulse should be applied to which channel at what time. There is as yet no language for these pulse schedules that is very amenable to formal semantics.

In this paper, we propose such a language called GRAMPUS (GRAded Modal type theory for Pulse Schedules). It is a *graded modal type theory*, where the grades represent timing information: a variable $x :^{50} Q_1$ will represent a state of qubit Q_1 that will exist 50 nanoseconds in the future, and a variable $y :^{-75} Q_2$ will represent a state of qubit Q_2 that existed 75 nanoseconds in the past.

We give the syntax for two type theories, one with grades (the *annotated* language) and one without (the *plain* language). We prove some metatheoretic properties, and describe the semantics in terms of category theory. We show that the input signals to a quantum chip forms a model of the annotated language. We also give a syntactic model, prove that it is initial, and hence prove soundness and completeness theorems.

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Theory of computation → Type theory; Theory of computation → Categorical semantics; Theory of computation → Quantum computation theory

Keywords and phrases Quantum computing, superconducting qubits, linear type theory, graded modal type theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.5

Supplementary Material *Software (Source Code)*: <https://codeberg.org/radams78/grampus.git> [2]
archived at `swh:1:dir:bc79e493a8f11d72e5382289e1ef8d1284024082`

Funding This research was funded by the SSF (Swedish Foundation for Strategic Research) project QuantumStack, grant number FUS21-0063.

1 Introduction

The operations to be performed by a quantum computer are almost invariably given in the form of a *quantum circuit* (see e.g. [19, Ch. 4]). This is a well-defined language with universally agreed semantics: there is no ambiguity about the unitaries and measurement operations that a quantum circuit diagram denotes.



© Robin Adams, Jean-Philippe Bernardy, Lorenzo Perticone, and Jeremy Pope;
licensed under Creative Commons License CC-BY 4.0
31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 5; pp. 5:1–5:22



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the final stage of compilation, a quantum circuit must be translated into the input signals accepted by the quantum hardware itself. For a quantum computer based on superconducting qubits, this will be a sequence of microwave control pulses to be sent to the various input channels [13]. A *pulse schedule* gives a full specification for which pulse should be applied to which channel at what time.

There are languages for describing these pulse schedules, such as OpenPulse [16, 3]. However these languages are not very amenable to formal semantics. If we want, for example, a formally verified quantum compiler, then we require a language for specifying pulse schedules that can be given precisely defined semantics.

We propose a type theory called GRAMPUS (GRAded Modal type theory for PUlse Schedules). It is a linear type theory augmented with *grades* [7, 1, 12] (equivalently modalities) which represent relative timing information: a variable $x :^{50} Q_1$ will represent a state of qubit Q_1 that will exist 50 nanoseconds in the future, and a variable $y :^{-75} Q_2$ will represent a state of qubit Q_2 that existed 75 nanoseconds in the past. Such a language can then be compiled to a pulse schedule. The correctness of this compiler can be captured by the following commutation square:

$$\begin{array}{ccc}
 \Gamma \vdash_d t : A & \xrightarrow{[\cdot]} & \Gamma \vdash t : A \\
 \downarrow \llbracket \cdot \rrbracket_p & & \downarrow \llbracket \cdot \rrbracket_u \\
 \text{Time} \rightarrow \text{Channel} \rightarrow \mathbb{R} & \xrightarrow{\text{hardware model}} & \mathbb{C}^{2^n} \multimap \mathbb{C}^{2^n}
 \end{array} \tag{1}$$

In the above, $\Gamma \vdash t : A$ represents a *plain program* that corresponds to a quantum circuit without measurement (and with no timing information). The judgement $\Gamma \vdash_d t : A$ represents a quantum circuit annotated with timings. Such annotations can be erased ($[\cdot]$) to obtain a plain program $\Gamma \vdash t : A$. In the semantics, a pulse schedule maps every channel and every time to a signal: $(\text{Channel} \rightarrow \text{Time} \rightarrow \mathbb{R})$. We denote the space of unitary operators as $\mathbb{C}^{2^n} \multimap \mathbb{C}^{2^n}$.

The plain program can be interpreted to a unitary operator. The hardware model maps a pulse schedule to the unitary operator that it implements. The interpretation function $\llbracket \cdot \rrbracket_u$ maps a program $\Gamma \vdash t : A$ to a unitary operator in $\llbracket \Gamma \rrbracket_u \multimap \llbracket A \rrbracket_u$ (Section 3.3). The interpretation function $\llbracket \cdot \rrbracket_p$ maps an annotated program $\Gamma \vdash_d t : A$ into a global pulse schedule, by combining the pulses associated with each gate occurring in the program (Section 3.4).

If the hardware model is correct then the diagram commutes (Theorem 25), ensuring that a pulse schedule implements the same unitary operator as its corresponding circuit.

2 Syntax of the Programming Languages

A quantum chip consists of a number of *qubits* (quantum bits) with input channels (or drive lines) and output channels (or readout lines). There are certain operations or *quantum gates* that we wish to perform on the qubits. We assume given a number of gates (usually called the *native gates* for the chip) such that we know the signal (or pulse) that must be sent down the appropriate input channel(s) in order to effect that gate.

Our languages are based on the multiplicative fragment of (first-order) intuitionistic linear logic [4]. Namely, we have products $(A \otimes B)$ with unit 1. Additionally, we have a base type q_i for each qubit q_i , representing the state of that qubit. We can also annotate a type with a (relative) timing $(\textcolor{blue}{d}A)$ meaning the data will be available exactly d units of time in the future. In the annotated language, every variable in the context will be annotated with its relative availability. In what follows, the part of the syntax related to annotations is shown in blue. The erasure function $[\cdot]$ removes exactly this part.

We assume given:

- a set $\mathbf{Q}$ of *qubits* (or *qubit labels*)
- a set $\mathbf{G}$ of *gates* (or *gate constants*)
- for every gate $G \in \mathbf{G}$, a tuple $(q_1, \dots, q_n)$ of distinct elements of $\mathbf{Q}$ which the gate acts on, and a non-negative real number d_G , the *duration* of the gate G . We say that a gate G is an *n-ary gate* iff its tuple of qubits has length n . We shall often write $(G, d_G, q_1, \dots, q_n) \in \mathbf{G}$.

The syntax of GRAMPUS is then given by the grammar:

$$\begin{aligned}
 \text{Type } A, B &::= 1 \mid q \mid A \otimes B \mid \boxed{d} A \\
 \text{Context } \Gamma &::= [] \mid \Gamma, x :^d A \\
 \text{Term } s, t &::= x \mid \star \mid \text{let } \star = s \text{ in } t \mid G(t_1, \dots, t_n) \mid (s, t) \\
 &\quad \mid \text{let } (x, y) = s \text{ in } t \mid \text{box } d \ t \mid \text{let } \text{box } d \ x = s \text{ in } t \mid \text{delay}_d(t) \\
 &\quad \mid \text{let } x = s \text{ in } u \\
 \text{Judgement } \mathcal{J} &::= \Gamma \vdash t : A \mid \Gamma \vdash s = t : A
 \end{aligned}$$

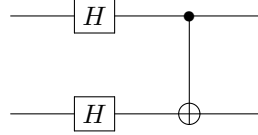
where $q \in \mathbf{Q}$, $d \in \mathbb{R}$ and $G \in \mathbf{G}$ is an n -ary gate.

► **Example 1.** Consider a very simple quantum chip with two qubits q_1 and q_2 , where the native gates are:

- H_1 , a Hadamard gate applied to q_1 with duration 100 nanoseconds;
- H_2 , a Hadamard gate applied to q_2 with duration 110 nanoseconds;
- $CNOT$, a controlled NOT gate applied to q_1 and q_2 with duration 500 nanoseconds

That is, $\mathbf{Q} = \{q_1, q_2\}$ and $\mathbf{G} = \{(H_1, 100, q_1), (H_2, 110, q_2), (CNOT, 500, q_1, q_2)\}$.

We wish to run the following circuit on this chip:



In the plain language, this is represented by the term

$$x : q_1, y : q_2 \vdash CNOT(H_1(x), H_2(y)) : q_1 \otimes q_2$$

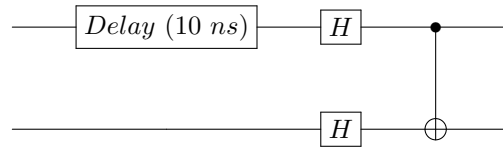
In the annotated language, we can represent this as a pulse schedule where q_1 's timeline starts at time -600 and q_2 's timeline starts at time -610 as follows:

$$x :^{-600} q_1, y :^{-610} q_2 \vdash CNOT(H_1(x), H_2(y)) : q_1 \otimes q_2$$

We can start the two qubits at the same time (-610) by inserting a delay of 10 ns before the Hadamard gate on q_1 :

$$x :^{-610} q_1, y :^{-610} q_2 \vdash CNOT(H_1(\text{delay}_{10}(x)), H_2(y)) : q_1 \otimes q_2$$

representing the following sequence of operations on the chip:

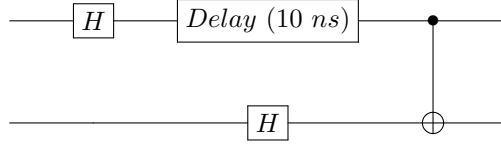


5:4 A Graded Modal Type Theory for Pulse Schedules

Or we can represent the pulse schedule where we apply the Hadamard gate to both qubits starting at time -610 , and insert a delay of 10 nanoseconds after the Hadamard gate on qubit q_1 , as follows:

$$x :^{-610} q_1, y :^{-610} q_2 \vdash CNOT(\text{delay}_{10}(H_1(x)), H_2(y)) : q_1 \otimes q_2$$

representing the following sequence of operations on the chip:



In both of these examples, we start the pulse schedule at a negative time, and then end at time zero. We can shift the second example so it starts at time 0 and ends at time 610 as follows:

$$x :^0 q_1, y :^0 q_2 \vdash \text{box } 610 \ CNOT(H_1(\text{delay}_{10}(x)), H_2(y)) : \boxed{610}(q_1 \otimes q_2)$$

As expected, terms of the language allow manipulating products, but we can also apply an n -ary quantum gate G operating on an n -tuple of the qubits, or apply a delay of any duration to any qubit, which represents sending no input signal for that period of time. (In the ideal quantum computer, applying a delay does not change the qubit's state. In a real quantum computer, the state of an idle qubit quickly decays as it loses energy to, and gains noise from, the environment.)

The typing rules for the language are given in Figure 1. The key point is that if a gate G takes d units of time to perform its operation in hardware, then we annotate the corresponding inputs by $-d$, capturing the fact that to obtain the expected output *now*, the inputs must be provided d units of time in the past. The introduction and elimination rules for $\boxed{d}A$ are setup in a way to correctly represent a time-shift by d , following the pattern of general frameworks for grades or modalities [7, 1, 12]. It is easy to see that erasure preserves typing.

2.1 Judgemental Equality

We have rules establishing that judgemental equality is reflexive, symmetric, transitive, and a congruence for each of the primitive constructors, plus:

Beta Rules

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash (\text{let } * = * \text{ in } t) = t : A}$$

$$\frac{\Gamma \vdash s : A \quad \Delta \vdash t : B \quad x :^d A, y :^d B, \Theta \vdash u : C}{d + \Gamma, d + \Delta, \Theta \vdash (\text{let } (x, y) = (s, t) \text{ in } u) = u[x := s, y := t] : C}$$

$$\frac{\Gamma \vdash s : A \quad x :^e A, \Delta \vdash t : B}{e + \Gamma, \Delta \vdash (\text{let box } d \ x = \text{box } d \ s \text{ in } t) = t[x := s] : B}$$

$$\frac{\Gamma \vdash s : A \quad x :^e A, \Delta \vdash t : B}{e + \Gamma, \Delta \vdash (\text{let } x = s \text{ in } t) = t[x := s] : B}$$

$$\begin{array}{c}
\frac{\Gamma, x : {}^d A, y : {}^e B, \Delta \vdash t : C}{\Gamma, y : {}^e B, x : {}^d A, \Delta \vdash t : C} \text{Swap} \qquad \frac{}{x : {}^0 A \vdash x : A} \text{Var} \qquad \frac{}{\vdash * : 1} \text{Unit-I} \\
\\
\frac{\Gamma \vdash s : 1 \quad \Delta \vdash t : A}{{}^d \Gamma, \Delta \vdash \text{let } * = s \text{ in } t : A} \text{Unit-E} \qquad \frac{\Gamma_1 \vdash t_1 : q_1 \quad \dots \quad \Gamma_n \vdash t_n : q_n}{{}^d \Gamma_1, \dots, {}^d \Gamma_n \vdash G(t_1, \dots, t_n) : q_1 \otimes \dots \otimes q_n} \text{Gate} \\
\\
\frac{\Gamma \vdash s : A \quad \Delta \vdash t : B}{\Gamma, \Delta \vdash (s, t) : A \otimes B} \text{Tens-I} \qquad \frac{\Gamma \vdash s : A \otimes B \quad x : {}^d A, y : {}^d B, \Delta \vdash t : C}{{}^d \Gamma, \Delta \vdash \text{let}(x, y) = s \text{ in } t : C} \text{Tens-E} \\
\\
\frac{\Gamma \vdash t : A}{{}^d \Gamma \vdash \text{box } d \ t : \boxed{d} A} \text{Box-I} \qquad \frac{\Gamma \vdash s : \boxed{d} A \quad x : {}^e A, \Delta \vdash t : B}{{}^e - {}^d \Gamma, \Delta \vdash \text{let box } d \ x = s \text{ in } t : B} \text{Box-E} \\
\\
\frac{\Gamma \vdash t : q \quad d \geq 0}{{}^d \Gamma \vdash \text{delay}_d(t) : q} \text{Delay} \qquad \frac{\Gamma \vdash s : A \quad x : {}^d A, \Delta \vdash t : C}{{}^d \Gamma, \Delta \vdash \text{let } x = s \text{ in } t : C} \text{Let}
\end{array}$$

■ **Figure 1** Typing rules for our languages. In the Gate rule, G is a gate acting on $(q_1, \dots, q_n)$ with duration d . Timing annotations are highlighted in blue. We write ${}^d \Gamma$ for the result of increasing every grade in the context Γ by d .

Eta Rules

$$\begin{array}{c}
\frac{\Gamma \vdash t : I}{\Gamma \vdash (\text{let } * = t \text{ in } *) = t : I} \\
\\
\frac{\Gamma \vdash t : A \otimes B}{\Gamma \vdash (\text{let } (x, y) = t \text{ in } (x, y)) = t : A \otimes B} \\
\\
\frac{\Gamma \vdash t : \boxed{d} A}{\Gamma \vdash (\text{let box } d \ x = t \text{ in box } d \ x) = t : \boxed{d} A}
\end{array}$$

Note we do not have $t = *$ for $t : 1$. (If t represents a process that takes time > 0 then it is not semantically equal to $*$.)

Commuting Conversions

In any linear type theory we also need judgemental equality rules for *commuting conversions*. The full list is given in Appendix 9.

3 Semantics

In this section, we describe the interpretation for our languages. We first introduce algebraic structures capturing what is needed to interpret such languages (models), and describe a general procedure to construct interpretations which we shall employ throughout this section. In the following, we assume fixed a set of qubit labels $\mathbf{Q}$ and a set of gates $\mathbf{G}$, whose elements $(G, d_G, q_1, \dots, q_n)$ encode that gate G acts on qubits $(q_1, \dots, q_n)$ in d_G units of time.

3.1 Categorical model of the plain language

The plain language can be interpreted in arbitrary symmetric monoidal categories, using the ideas from [7].

► **Definition 2** (Categorical model of the plain language). A (categorical) model for the plain language $(\mathcal{C}^\otimes, X_-, f_-)$ is given by the following data:

- A symmetric monoidal category (SMC) $\mathcal{C}^\otimes$,
- For every qubit $q \in \mathbf{Q}$, an object $X_q : \mathcal{C}^\otimes$,
- For every gate $(G, d, q_1, \dots, q_n)$ a morphism $f_G : \bigotimes_{i=1}^n X_{q_i} \rightarrow \bigotimes_{i=1}^n X_{q_i}$

We can hence interpret the plain language in any of its categorical models: the interpretation is constructed as follows.

► **Definition 3** (Interpretation of the plain language). Given a categorical model of the plain language $(\mathcal{C}^\otimes, X_-, f_-)$, we define the following interpretation $\llbracket - \rrbracket_u$ operation:

- For every type A , we define an object $\llbracket A \rrbracket_u \in \mathcal{C}^\otimes$; we set $\llbracket 1 \rrbracket_u := I$, $\llbracket q \rrbracket_u := X_q$ and $\llbracket A \otimes B \rrbracket_u := \llbracket A \rrbracket_u \otimes \llbracket B \rrbracket_u$;
- For every context Γ , we define an object $\llbracket \Gamma \rrbracket_u$; we set $\llbracket [] \rrbracket_u := I$ and $\llbracket x : A, \Gamma \rrbracket_u := \llbracket A \rrbracket_u \otimes \llbracket \Gamma \rrbracket_u$;
- For every derivable judgement $\Gamma \vdash t : A$, we define a morphism $\llbracket \Gamma \vdash t : A \rrbracket_u := \llbracket \Gamma \rrbracket_u \xrightarrow{\llbracket t \rrbracket_u} \llbracket A \rrbracket_u$, where $\llbracket t \rrbracket_u$ is defined by induction on the judgement's derivation: Var and Unit-I correspond to identities, Swap is composition with $\mathcal{C}^\otimes$'s symmetry, Unit-E and Tens-I are the obvious tensor products, Tens-E becomes the following morphism:

$$\llbracket \text{let}(x, y) = s \text{ in } t \rrbracket_u = \llbracket \Gamma \rrbracket_u \otimes \llbracket \Delta \rrbracket_u \xrightarrow{\llbracket s \rrbracket_u \otimes \text{Id}} \llbracket A \rrbracket_u \otimes \llbracket B \rrbracket_u \otimes \llbracket \Delta \rrbracket_u \xrightarrow{\llbracket t \rrbracket_u} \llbracket C \rrbracket_u$$

and Gate becomes the following morphism:

$$\llbracket G(t_1, \dots, t_n) \rrbracket_u = \bigotimes_{i=1}^n \llbracket \Gamma_i \rrbracket_u \xrightarrow{\bigotimes_{i=1}^n \llbracket t_i \rrbracket_u} \bigotimes_{i=1}^n \llbracket q_i \rrbracket_u \xrightarrow{f_G} \bigotimes_{i=1}^n \llbracket q_i \rrbracket_u$$

► **Proposition 4.** *The interpretation defined above preserves judgemental equality: if the judgement $s = t$ is admissible, then $\llbracket s \rrbracket_u = \llbracket t \rrbracket_u$.*

Proof. This is straightforward to prove by induction on the derivation of the equality judgement. ◀

3.2 Categorical model of the annotated language

In analogy to the previous subsection, we can interpret the annotated language in (appropriately structured) SMCs, which we describe here. Most of our definitions are analogous to what we described for the plain language. We additionally introduce a symmetric monoidal endofunctor $\text{Shift}(x)$ for any real number $x \in \mathbb{R}$, to interpret time shifts (through their action on objects, hence on types and contexts) and the box construct (through their action on morphisms, hence on terms). Symmetry and monoidality for such endofunctors corresponds to the grades interacting appropriately with tensor products and unit types. In order to package the endofunctors $\text{Shift}(x)$, we first recall the notion of a symmetric monoidal *module category* [8] (sometimes also called *actegories*).

► **Definition 5** (Symmetric monoidal $\mathcal{B}^\otimes$ -module categories). Given a monoidal category $\mathcal{B}^\otimes = (\mathcal{B}, 1, \otimes, \dots)$, a *symmetric monoidal $\mathcal{B}^\otimes$ -module category* $\mathcal{C}^{\boxtimes, \bullet} = (\mathcal{C}^{\boxtimes}, \bullet)$ ($\mathcal{B}^\otimes$ -SMMC in the future, or just SMMC if no emphasis is put on $\mathcal{B}^\otimes$) is the data of an SMC $\mathcal{C}^{\boxtimes} = (\mathcal{C}, I, \boxtimes, \dots)$ together with a strong monoidal functor $\bullet : \mathcal{B}^\otimes \rightarrow \mathbf{SMC}[\mathcal{C}^{\boxtimes}, \mathcal{C}^{\boxtimes}]^\circ$ into the monoidal (under composition) category of strong symmetric monoidal endofunctors of $\mathcal{C}^{\boxtimes}$.

► **Definition 6** (Categorical model of the annotated language). A categorical model of the annotated language $(\mathcal{C}^{\otimes, \bullet}, X_-, f_-, \text{delay}_-)$ is given by the following (where $\mathbb{R}^+$ denotes the discrete SMC of real numbers under addition):

- An $\mathbb{R}^+$ -SMMC $\mathcal{C}^{\otimes, \bullet}$ (we shall write $\text{Shift}(d) := (d \bullet -)$),
- For every qubit $q \in \mathbf{Q}$ an object $X_q : \mathcal{C}^\otimes$,
- For every gate $(G, d, q_1, \dots, q_n)$, a morphism $f_G : \text{Shift}(-d)(\bigotimes_{i=1}^n X_{q_i}) \rightarrow \bigotimes_{i=1}^n X_{q_i}$,
- For every real number $x \in \mathbb{R}$ and qubit $q \in \mathbf{Q}$, a morphism $\text{delay}_x : \text{Shift}(-x)(X_q) \rightarrow X_q$,

The above definition is worth unpacking: monoidality for the functor $\text{Shift}(-)$ means that, for every real numbers $d_1, d_2 \in \mathbb{R}$ and object $X : \mathcal{C}^\otimes$, we get isomorphisms (natural in d_1, d_2, X , and appropriately coherent)

$$\text{Shift}(0)(X) \simeq X \text{ and } \text{Shift}(d_1 + d_2)(X) \simeq \text{Shift}(d_1)(\text{Shift}(d_2)(X))$$

and for a fixed $d \in \mathbb{R}$, monoidality of $\text{Shift}(d)$ means that, for every $X, Y : \mathcal{C}^\otimes$, we get isomorphisms (natural in d, X, Y and, as before, appropriately coherent)

$$\text{Shift}(d)(I) \simeq I \text{ and } \text{Shift}(d)(X \otimes Y) \simeq \text{Shift}(d)(X) \otimes \text{Shift}(d)(Y)$$

This allows us to interpret the annotated language in any of its categorical models. The interpretation is built in a way analogous to Definition 3; the only differences relate to annotations, Box-I, Box-E and Delay (which are missing from the plain language).

► **Definition 7** (Interpretation for the annotated language). Given a categorical model of the annotated language $(\mathcal{C}^{\otimes, \bullet}, X_-, f_-, \text{delay}_-)$, we define an interpretation $\llbracket - \rrbracket_p$ as follows:

- For types we define $\llbracket - \rrbracket_p$ in the same way as $\llbracket - \rrbracket_u$, plus the clause $\llbracket \boxed{d} A \rrbracket_p := \text{Shift}(d)(\llbracket A \rrbracket_p)$,
- For every context Γ , we define an object $\llbracket \Gamma \rrbracket_p$; we set $\llbracket [] \rrbracket_p := I$ and $\llbracket x :^d A, \Gamma \rrbracket_p := \text{Shift}(d)(\llbracket A \rrbracket_p \otimes \llbracket \Gamma \rrbracket_p)$ (notice that monoidality of $\text{Shift}(d)$ implies $\llbracket d + \Gamma \rrbracket_p \simeq \text{Shift}(d)(\llbracket \Gamma \rrbracket_p)$),
- For judgements, we define $\llbracket - \rrbracket_p$ in the same way as $\llbracket - \rrbracket_u$ (after inserting coherences where appropriate); furthermore, we interpret

$$\llbracket \text{delay}_d(t) \rrbracket_p = \llbracket -d + \Gamma \rrbracket_p \simeq \text{Shift}(-d)(\llbracket \Gamma \rrbracket_p) \xrightarrow{\text{Shift}(-d)(\llbracket t \rrbracket_p)} \text{Shift}(-d)(\llbracket q \rrbracket_p) \xrightarrow{\text{delay}_d} \llbracket q \rrbracket_p$$

$$\llbracket \text{box } d \ t \rrbracket_p = \llbracket d + \Gamma \rrbracket_p \simeq \text{Shift}(d)(\llbracket \Gamma \rrbracket_p) \xrightarrow{\text{Shift}(d)(\llbracket t \rrbracket_p)} \text{Shift}(d)(\llbracket A \rrbracket_p)$$

$$\llbracket \text{let box } d \ x = s \text{ in } t \rrbracket_p = \llbracket (e - d) + \Gamma \rrbracket_p \otimes \llbracket \Delta \rrbracket_p \xrightarrow{\phi \otimes \text{Id}} \text{Shift}(e)(\llbracket A \rrbracket_p \otimes \llbracket \Delta \rrbracket_p) \xrightarrow{\llbracket t \rrbracket_p} \llbracket B \rrbracket_p$$

where the arrow ϕ is defined as

$$\phi = \llbracket (e - d) + \Gamma \rrbracket_p \simeq \text{Shift}(e - d)(\llbracket \Gamma \rrbracket_p) \xrightarrow{\text{Shift}(e - d)(\llbracket s \rrbracket_p)} \text{Shift}(e - d)(\text{Shift}(d)(\llbracket A \rrbracket_p)) \simeq \text{Shift}(e)(\llbracket A \rrbracket_p)$$

For instance, the term generated by Tens-I is interpreted as the following

$$\begin{aligned} \llbracket \text{let}(x, y) = s \text{ in } t \rrbracket_p &= \llbracket d + \Gamma, \Delta \rrbracket_p \\ &\simeq \text{Shift}(d)(\llbracket \Gamma \rrbracket_p) \otimes \llbracket \Delta \rrbracket_p \\ &\xrightarrow{\text{Shift}(d)(\llbracket s \rrbracket_p)} \text{Shift}(d)(\llbracket A \rrbracket_p \otimes \llbracket B \rrbracket_p) \otimes \llbracket \Delta \rrbracket_p \\ &\simeq \text{Shift}(d)(\llbracket A \rrbracket_p \otimes \text{Shift}(d)(\llbracket B \rrbracket_p \otimes \llbracket \Delta \rrbracket_p)) \\ &\xrightarrow{\llbracket t \rrbracket_p} \llbracket C \rrbracket_p \end{aligned}$$

and gates become

$$\begin{aligned}
\llbracket G(t_1, \dots, t_n) \rrbracket_p &= \llbracket -d + \Gamma_1, \dots, -d + \Gamma_n \rrbracket_p \\
&\simeq \text{Shift}(-d) \left(\bigotimes_{i=1}^n \llbracket \Gamma_i \rrbracket_p \right) \\
&\xrightarrow{\text{Shift}(-d) \left(\bigotimes_{i=1}^n \llbracket t_i \rrbracket_p \right)} \text{Shift}(-d) \left(\bigotimes_{i=1}^n \llbracket q_i \rrbracket_p \right) \\
&\xrightarrow{f_G} \bigotimes_{i=1}^n \llbracket q_i \rrbracket_p
\end{aligned}$$

► **Proposition 8.** *The interpretation defined above preserves judgemental equality: if the judgment $s = t$ is admissible, then $\llbracket s \rrbracket_p = \llbracket t \rrbracket_p$.*

Proof. Again, this is straightforward to prove by induction on the derivation of the equality judgement. ◀

Every model of the plain language can be made into a model of the annotated language; this corresponds syntactically to erasing timing annotations (the $\lfloor - \rfloor$ operation defined in Section 1), and semantically to equipping an SMC with the trivial SMMC structure (where objects in $\mathbb{R}^+$ act identically).

► **Remark 9.** Given a model of the plain language $(\mathcal{C}^\otimes, X_-, f_-)$, we can define a model of the annotated language by defining

- Time shifts to be identity functors $\text{Shift}(d) := \text{Id}_{\mathcal{C}^\otimes}$ for every real number d , and
- Delays to be identity morphisms $\text{delay}_d := \text{Id}_{X_q}$.

3.3 Hilbert Spaces and Unitary Operators

Importantly, the category of finite-dimensional Hilbert spaces and unitary operators $\mathbf{FdHilb}_{\mathbf{U}}$ is a categorical model of the plain language (Definition 2):

► **Proposition 10** (Categorical model of Unitary Operators). *Assume that, for every n -ary gate G , we have a unitary operator $f_G : \mathbb{C}^{2^n} \rightarrow \mathbb{C}^{2^n}$. The category $\mathbf{FdHilb}_{\mathbf{U}}$ is symmetric monoidal (i.e. $\mathbf{FdHilb}$ is symmetric monoidal and its identities, associators, unitors and symmetries are unitary operators). This, together with the assignments*

- $X_q := \mathbb{C}^2$ for every qubit $q \in \mathbf{Q}$;
 - f_G as the linear operator corresponding to the quantum gate G .
- makes it into a categorical model of the plain language.*

In consequence, programs in the plain language can be interpreted in the ideal quantum computer – ignoring all aspects related to duration of execution. A program $\Gamma \vdash t : A$ hence corresponds to a unitary operator $\llbracket t \rrbracket \in \llbracket \Gamma \rrbracket \multimap \llbracket A \rrbracket$. In particular, for any gate G operating on n qubits, we assume given a corresponding operator $\llbracket G \rrbracket \in \mathbb{C}^{2^n} \multimap \mathbb{C}^{2^n}$.

3.4 Signals

In a physical quantum chip with superconducting qubits, there are a number of input channels or *drive lines* that can be used to perform operations on the qubits, and *readout lines* for receiving the outcome of measuring a qubit. (For details, see [13].)

The input to a drive line can be modelled as a function $[startTime, endTime] \rightarrow \mathbb{R}$. A complete pulse schedule consists then of such a function for each channel connected to such a drive line: $Channel \rightarrow [startTime, endTime] \rightarrow \mathbb{R}$. However, these do not form the

morphisms of an SMC; we could only form the tensor product of two pulse schedules if their start times and end times agree. To support the SMC structure, we must allow a different start or end time for each channel: $(c : \text{Channel}) \rightarrow [\text{StartTime}(c), \text{EndTime}(c)] \rightarrow \mathbb{R}$.

We can then define an SMC $\mathcal{S}$ whose morphisms are the pulse schedules themselves.

► **Definition 11** (SMC of Signals). Given a set of channels $\mathbf{C}$, we define the SMC $\mathcal{S}$ such that:

- Objects are finite sequences of pairs $\mathbf{Ob}(\mathcal{S}) := (\mathbb{R} \times \mathbf{C})^*$ of a real number (the starting time of the signal) and a channel;
- Given $X, Y : \mathcal{S}$ with $X = [(s_1, c_1), \dots, (s_m, c_m)]$ and $Y = [(t_1, d_1), \dots, (t_n, d_n)]$, define the homset $\mathbf{Hom}_{\mathcal{S}}[X, Y]$ as

$$\{(\sigma, [\phi_1, \dots, \phi_n]) \mid \sigma \in S_n \text{ such that } \forall k, c_k = d_{\sigma(k)} \text{ and } s_k \leq t_{\sigma(k)} \text{ and } \phi_k : [s_k, t_{\sigma(k)}] \rightarrow \mathbb{R}\}$$

if $m = n$ and as the empty set otherwise, where S_n is the symmetric group. One could require some additional regularity conditions on the functions ϕ_k , which we simply omit;

- Identities are hence given by pairs $(e, [\emptyset, \dots, \emptyset])$ where e is the identity permutation, and $\emptyset$ is the empty function;
- Composition is given by composing the permutations and concatenating the pulses:

$$(\sigma_1, [\phi_1, \dots, \phi_n]) \circ (\sigma_2, [\psi_1, \dots, \psi_n]) := (\sigma_1 \circ \sigma_2, [\gamma_1, \dots, \gamma_n])$$

where $\gamma_k : [r_k, t_{(\sigma_1 \circ \sigma_2)(k)}] \rightarrow \mathbb{R}$ is defined as

$$\gamma_k(t) := \begin{cases} \psi_k(t) & \text{if } t \in [r_k, s_{\sigma_2(k)}] \\ \phi_{\sigma_2(k)}(t) & \text{if } t \in [s_{\sigma_2(k)}, t_{(\sigma_1 \circ \sigma_2)(k)}] \end{cases}$$

(intuitively, we keep track of how channels are permuted and apply the pulses sequentially)

- The tensor unit is given by the empty sequence $I := [] : \mathcal{S}$;
- Tensor product of objects is given by list concatenation;
- Tensor product of morphisms is given by the following

$$(\sigma_1, [\phi_1, \dots, \phi_m]) \otimes (\sigma_2, [\psi_1, \dots, \psi_n]) := (\sigma_1 \star \sigma_2, [\phi_1, \dots, \phi_m, \psi_1, \dots, \psi_n])$$

where by $(- \star -) : S_m \times S_n \rightarrow S_{m+n}$ we denote the obvious group homomorphism.

- The unitors and associator are given by identities, and
- The symmetry is given by $(\sigma, [\emptyset, \dots, \emptyset])$ where σ is the appropriate permutation and $\emptyset$ is the empty map.

We can use this category to build a model of the annotated language out of the input signals to the quantum chip, in such a way that the interpretation function $\llbracket \cdot \rrbracket_p$ is exactly the compiler that translates from a GRAMPUS term to the corresponding input signals:

► **Proposition 12** ($\mathcal{S}$ is a categorical model of the annotated language).

Assume given a correspondence between channels and qubits (in the simplest case this could be an injection $\mathbf{Q} \rightarrow \mathbf{C}$) and a pulse representation of each gate in $\mathbf{G}$. Then the SMC $\mathcal{S}$ can be extended to a categorical model of the annotated language as follows:

- For every real number $x \in \mathbb{R}$, the functor $\text{Shift}(x)$ is defined as follows:

$$\text{Shift}(x)[(t_1, c_1), \dots, (t_n, c_n)] := [(t_1 + x, c_1), \dots, (t_n + x, c_n)]$$

$$\text{Shift}(x)(\sigma, [\phi_1, \dots, \phi_n]) := (\sigma, [\phi_1(\cdot - x), \dots, \phi_n(\cdot - x)])$$

- For every real number $x \in \mathbb{R}$, the morphisms delay_x are built from the identity permutation $e \in S_1$ and the pulse δ_0 which is constantly zero¹.

$$\text{delay}_x = (e, [\delta_0])$$

Proof. Functoriality and (symmetric) monoidality for $\text{Shift}(x)$ are straightforward to verify. Furthermore, in this model, the structural natural isomorphisms making $\text{Shift}(-)$ into a monoidal functor are identities, meaning it is *strict* monoidal. ◀

3.5 Syntactic Model and Completeness Theorem

We can build a model for either of our type theories out of the terms of the theory itself. It is generally considered desirable if this term model is initial in the category of models, with the interpretation function $\llbracket \cdot \rrbracket$ being the unique morphism from the term model to an arbitrary model (see e.g. [24]). In this section, we prove that this is the case for both of our languages.

► **Definition 13** (Term category for the annotated Language).

We define the model of the plain language $\mathbf{Syntax} = (\mathbf{Syntax}, X_-, f_-)$ as follows. We begin by defining a function $\mathbf{T}_y$ that sends contexts Γ to types $\mathbf{T}_y(\Gamma)$ by

$$\mathbf{T}_y(\llbracket \rrbracket) := 1; \quad \mathbf{T}_y(x :^d A, \Gamma) := (\llbracket d \rrbracket A) \otimes \mathbf{T}_y(\Gamma)$$

We can then define the underlying category of $\mathbf{Syntax}$ as follows:

- Its objects are contexts,
- Its morphisms $f : \Gamma \rightarrow \Delta$ are derivable judgments $\Gamma \vdash t : \mathbf{T}_y(\Delta)$, quotiented by judgmental equality².
- Identities $\text{id}_\Gamma : \Gamma \rightarrow \Gamma$ are defined by induction on Γ :

$$\text{Id}_\llbracket \rrbracket := *; \quad \text{id}_{x :^d A, \Gamma} := (x, \text{id}_\Gamma)$$

- Composition is given by inductively using the Tens-E and Unit-E rules: given $f : \Gamma \rightarrow \Delta$ and $g : \Delta \rightarrow \Xi$, if $\Delta = x_1 :^{d_1} A_1, \dots, x_n :^{d_n} A_n$,

$$\begin{aligned} g \circ f &:= \text{let } (y_1, z_1) = f && \text{in let box } d_1 \ x_1 = y_1 \text{ in} \\ &\quad \text{let } (y_2, z_2) = z_1 && \text{in let box } d_2 \ x_2 = y_2 \text{ in} \\ &\quad \dots \\ &\quad \text{let } (y_n, z_n) = z_{n-1} && \text{in let box } d_n \ x_n = y_n \text{ in} \\ &\quad \text{let } * = z_n && \text{in } g \end{aligned}$$

for distinct, fresh $y_1, \dots, y_n, z_1, \dots, z_n$. We can give a derivation for the judgment $\Gamma \vdash (g \circ f) : \mathbf{T}_y(\Xi)$ by generalizing the construction:

$$\frac{\Gamma \vdash t : \mathbf{T}_y(\Delta) \quad \Delta, \Xi \vdash s : C}{\Gamma, \Xi \vdash \text{let}_\Delta t \text{ in } s : C}$$

and performing induction on Δ ; the let_Δ statement above is defined in exactly the same way as composition, and generalizes it.

¹ Notice how these morphisms are not invertible: in general, due to phenomena such as decoherence, a qubit left alone will not stay the same. For a more detailed treatment of this, see section 3 of [13].

² In the rest of this definition, we will silently treat representatives of such equivalence classes as the classes themselves.

Verifying that the above definition indeed gives a category structure is routine: unitality and associativity of composition hinges on the usual substitution lemmas. We can further enhance the structure on **Syntax** to that of an SMC in the obvious way.

► **Definition 14** (Monoidal structure on **Syntax**). The category **Syntax** can be made into an SMC (**Syntax**, $I, \otimes, \dots$) by defining the following:

- Its unit object I is given by the unit type $I := 1$;
- Its tensor product of objects is given by context concatenation $\Gamma \otimes \Delta := \Gamma, \Delta$;
- Its tensor product of morphisms is given by inductively applying the Tens-E and Tens-I rules: if $\Gamma_1 \vdash t_1 : \mathbf{Ty}(\Delta_1)$ and $\Gamma_2 \vdash t_2 : \mathbf{Ty}(\Delta_2)$, define

$$\begin{aligned} t_1 \otimes t_2 := & \text{let } (x_1, y_1) = t_1 && \text{in} \\ & \text{let } (x_2, y_2) = y_1 && \text{in} \\ & \dots \\ & \text{let } (x_n, y_n) = y_{n-1} && \text{in} \\ & \text{let } * = y_n && \text{in} \\ & (x_1, \dots, (x_n, t_2) \dots) \end{aligned}$$

- Checking that unitors and associators can just be identities is straightforward;
- The symmetry can be defined with a similar inductive strategy as the tensor product of morphisms (employing the Swap rule appropriately).

Checking the axioms of an SMC is straightforward. Since we shall evidently interpret qubits and gates as themselves (i.e. a qubit $q \in \mathbf{Q}$ is interpreted as the context $x :^0 q$, and similarly for gates), we only need to define the SMMC structure and the delay_d morphisms.

► **Definition 15** (SMMC structure and delay_d for **Syntax**). For any given real number $d \in \mathbb{R}$, define the symmetric monoidal endofunctor $\text{Shift}(d) : \mathbf{Syntax} \rightarrow \mathbf{Syntax}$ as follows:

- On objects Γ , define $\text{Shift}(d)(\Gamma) := d + \Gamma$
- On morphisms $\Gamma \vdash t : \mathbf{Ty}(\Delta)$, define

$$\begin{aligned} \text{Shift}(d)(t) := & \text{let } (x_1, y_1) = t && \text{in} \\ & \text{let } (x_2, y_2) = y_1 && \text{in} \\ & \dots \\ & \text{let } (x_n, y_n) = y_{n-1} && \text{in} \\ & (\text{box } d \ x_1, \dots, (\text{box } d \ x_n, \text{box } d \ y_n) \dots) \end{aligned}$$

Endowing $\text{Shift}(d)$ with the further structure of a symmetric monoidal functor is a simple exercise. Similarly, given a qubit $q \in \mathbf{Q}$ and a real number $d \in \mathbb{R}$, we define $\text{delay}_d : [x :^{-d} q] \rightarrow [x :^0 q]$ as (the obvious derivation for) the judgment

$$[x :^{-d} q] \vdash (\text{box } 0 \ \text{delay}_d(x), *) : q \otimes 1$$

The same constructions as above (Definitions 13 and 14) can be repeated for the plain language, producing an SMC **Syntax_u**. By construction, **Syntax** (resp. **Syntax_u**) is a model of the annotated (resp. plain) language.

► **Remark 16.** Every categorical model of the plain (resp. annotated) language $(\mathcal{C}^\otimes, X_-, f_-)$ (resp. $(\mathcal{C}^\otimes, \bullet, X_-, f_-, \text{Shift}(-), \text{delay}_-)$) admits a symmetric monoidal functor $\mathbf{Syntax}_u \rightarrow \mathcal{C}^\otimes$ preserving qubits and gates (resp. symmetric monoidal $\mathbb{R}^+$ -linear functor³ $\mathbf{Syntax} \rightarrow \mathcal{C}^\otimes$, also appropriately preserving delay_-). This functor is unique up to natural isomorphism that preserves qubits and gates, implying initiality for **Syntax**.

Proof. Such a functor can be constructed by adapting Definition 3 (resp. Definition 7). ◀

► **Theorem 17 (Completeness).** *Suppose $\Gamma \vdash_d s : A$ and $\Gamma \vdash_d t : A$. If $\llbracket s \rrbracket_p = \llbracket t \rrbracket_p$ in every model, then $\Gamma \vdash_d s = t : A$.*

Proof. If the equality holds in every model then it holds in **Syntax**, which means precisely that $\Gamma \vdash_d s = t : A$. ◀

A similar Completeness Theorem can be proved for the plain language.

3.6 The Category Circ

We can give an alternative characterization of the initial model **Syntax** as follows.

► **Definition 18.** The $\mathbb{R}^+$ -SMMC **Circ** is defined in the following way:

- Its objects are finite sequences of pairs (d, q) of a real number and a qubit: $\mathbf{Ob}(\mathbf{Circ}) := (\mathbb{R} \times \mathbf{Q})^*$, hence are exactly contexts whose types are restricted to just qubits,
- Its unit object is the empty sequence,
- Its tensor product is given, on objects, by concatenation,
- Its morphisms are generated (under composition and tensor products) by permutations, delays for every qubit q and real number x

$$\text{delay}_d : [(x, q)] \rightarrow [(x + d, q)]$$

and gates for every $(G, d, q_1, \dots, q_n) \in \mathbf{G}$

$$\phi_G : [(x, q_1), \dots, (x, q_n)] \rightarrow [(x + d, q_1), \dots, (x + d, q_n)]$$

- Unitors and associators are just identities,
- Symmetries are the appropriate permutations,
- For every real number $d \in \mathbb{R}$, the endofunctor $\bullet(d)$ acts
 - On objects, by adding d to the first component of every element of the sequence,

$$\bullet(d)((x_1, q_1), \dots, (x_n, q_n)) := [(x_1 + d, q_1), \dots, (x_n + d, q_n)]$$
 - On morphisms, it is the identity on permutations, maps delays to delays appropriately, and similarly for gates.

Let us call this category circuits (**Circ**) and write its homsets $\Gamma \multimap_G \Delta$. A similar definition gives **Circ_u**, where we omit the delay morphisms and the timing data from the objects.

It is important to notice that **Circ** (resp. **Circ_u**) is *not* the category of contexts over the annotated (plain) theory: it is the free *strict* SMMC (SMC) on the data encoded by $\mathbf{Q}$ and $\mathbf{G}$, while the category of contexts would be the free SMMC on such data. The equivalence between the two is a consequence of a general strictification theorem⁴.

³ I.e. a strong linear functor [8, definition 3.3.2] such that the underlying functor and the lineator are strong symmetric monoidal

⁴ While the case for the unannotated language is well-known ([14] and [15, chapter XI]), in the general case it is easier to prove the equivalence directly.

► **Remark 19.** A strong symmetric monoidal $\mathbb{R}^+$ -linear functor $\mathcal{C}^{\otimes, \bullet} \rightarrow \mathcal{D}^{\otimes, \bullet}$ between models preserves the choices of qubits, gates and delays precisely when it makes the following diagram commute

$$\begin{array}{ccc} & \text{Circ} & \\ \swarrow & & \searrow \\ \mathcal{C}^{\otimes, \bullet} & \xrightarrow{\quad} & \mathcal{D}^{\otimes, \bullet} \end{array}$$

Hence the category of models is equivalent to the coslice category $\text{Circ} \downarrow \mathbb{R}\text{-SMC}$.

In complete analogy, the category of models for the unannotated language is equivalent to the coslice $\text{Circ}_u \downarrow \text{SMC}$.

4 Normal forms

In this section we describe a *normal* representation for GRAMPUS; i.e. one that makes semantically equivalent terms syntactically α -equivalent. At the end of the section, we will deal with the representation for Circ morphisms themselves. But we first discuss how such morphisms should be embedded into the terms of a programming language. We construct such normal forms in three stages:

1. Decomposition of the inputs using eliminators ($\Gamma \vdash_e C$);
2. Application of circuit gates and delays ($\Gamma \multimap_G \Delta$, see Section 3.6);
3. Construction of the output of the desired type by using constructors ($\Gamma \vdash_n C$).

These judgements are defined by the following rules.

$$\begin{array}{c} \frac{\Theta, \Delta \vdash_e t : A}{\Theta, x : \textcolor{blue}{d} 1, \Delta \vdash_e \text{let } * = x \text{ in } t : A} \text{Unit-E} \qquad \frac{}{\vdash_n * : 1} \text{Unit-I} \\[10pt] \frac{\Theta, x : \textcolor{blue}{d} A, y : \textcolor{blue}{d} B, \Delta \vdash_e t : C}{\Theta, z : \textcolor{blue}{d} (A \otimes B), \Delta \vdash_e \text{let}(x, y) = z \text{ in } t : C} \text{Tens-E} \qquad \frac{\Gamma \vdash_n s : A \quad \Delta \vdash_n t : B}{\Gamma, \Delta \vdash_n (s, t) : A \otimes B} \text{Tens-I} \\[10pt] \frac{\Theta, x : \textcolor{blue}{d}^+ A, \Delta \vdash_e t : B}{\Theta, z : \textcolor{blue}{d} \textcolor{blue}{[e]} A, \Delta \vdash_e \text{let box } e \ x = z \text{ in } t : B} \text{Box-E} \qquad \frac{\Gamma \vdash_n t : A}{d + \Gamma \vdash_n \text{box } d \ t : \textcolor{blue}{[d]} A} \text{Box-I} \\[10pt] \frac{\Delta \vdash_n t : B \quad c : \Theta \multimap_G \Delta}{\Theta \vdash_e t \text{ after } c : B} \text{Circ} \qquad \frac{}{x : \textcolor{blue}{0} q \vdash_n x : q} \text{Var} \end{array}$$

In the above, the eliminators only work on the first non-qubit variable in the context (Θ stands for a context with only qubits). Because those are only variables, no further reduction can happen.

► **Remark 20.** Because gates have the same domain and codomain up to permutation and later durations, so does any circuit applied in the Circ rule, and thus variable names can remain the same before and after applying the circuit (even if they can be permuted).

With the exception of the Circ rule, the rules are a subset of GRAMPUS, and thus are given the same semantics. With this in place, we can show that the effect that the semantics of normal-form introduction and elimination rules is trivial.

► **Lemma 21.** *If $\Gamma \vdash_n t : A$, then $\llbracket \Gamma \rrbracket_p = \llbracket A \rrbracket_p$.*

Proof. The proof is a simple induction on the structure of t , relying on the fact that the interpretation of constructors do not use gates or delays. Additionally $id \otimes id = id$ and $\text{Shift}(d) id = id$. ◀

To state a similar result about eliminators, we want to generalise over them. We can do so by defining the relation $\Gamma \multimap_e \Delta$, which is any context transformation performed by an eliminator in the form allowed by the $\vdash_e$ judgement.

► **Definition 22** (Relation $\Gamma \multimap_e \Delta$).

$$* = x : (x :^d 1 \multimap_e []) \quad (x, y) = z : (z :^d (A \otimes B) \multimap_e x :^d A, y :^d B)$$

$$\text{box } d x = z : (z :^d [e] A \multimap_e x :^{d+e} A)$$

Then all normalised eliminators become instances of the following rule:

$$\frac{e : \Gamma \multimap_e \Delta \quad \Theta, \Delta, \Xi \vdash_e t : A}{\Theta, \Gamma, \Xi \vdash_e \text{let } e \text{ in } t : A}$$

and we can state the desired lemma as follows.

► **Lemma 23.** *For any $e : \Gamma \multimap_e \Delta$, we have $\llbracket \text{let } e \text{ in } t : A \rrbracket_p = \llbracket t \rrbracket_p$.*

Proof. By case analysis. For example the case for split is: $\llbracket \text{let } (x, y) = z \text{ in } t \rrbracket_p = \llbracket t \rrbracket_p \circ \text{Shift}(d) \llbracket z \rrbracket = \llbracket t \rrbracket_p$ ◀

► **Theorem 24** (Normalization). *Given a term $\Gamma \vdash t : A$, there is a unique u such that $\Gamma \vdash_e u : A$ and $\llbracket t \rrbracket_{\text{Circ}} = \llbracket u \rrbracket_{\text{Circ}}$.*

Proof.

- Recall that in **Circ**, contexts evaluate to contexts. More precisely $\llbracket \Gamma \rrbracket_{\text{Circ}}$ is a list of (delayed) qubits from Γ , in the same order.
- The circuit to embed (in the exactly single occurrence of rule *Circ*) is given by the semantics into **Circ**; $\llbracket t \rrbracket_{\text{Circ}} : \llbracket \Gamma \rrbracket_{\text{Circ}} \multimap_G \llbracket A \rrbracket_{\text{Circ}}$.
- The matching constructor $\llbracket A \rrbracket_{\text{Circ}} \vdash_n s : A$ is constructed by induction on the type. Furthermore constructor rules are purely type-directed, and exactly one rule applies at any point.
- The elimination sequence which matches Γ to $\llbracket \Gamma \rrbracket_{\text{Circ}}$ is done by induction on the suffix of Γ which starts with a non-elementary qubit type. As for constructors, there is a single rule which can apply at any point, because the first non-elementary type dictates which rule applies. When there are only qubits in the context, rule *Circ* applies on $\llbracket t \rrbracket_{\text{Circ}}$, and this is always applicable because the eliminators previously applied did not change the context semantics (Lemma 23). Hence there is always exactly one eliminator to use.

The fact that this normal form respects semantics is a consequence of Lemmata 21 and 23. ◀

Discussion

We have so far established the existence of unique normal forms, but the computation itself is grouped in a single morphism of the SMC **Circ**. Finally we can consider what method should be used to best represent morphisms in this category. There are many possible choices, because SMCs feature a number of laws relating objects and morphisms.

One route is to map them onto the GRAMPUS syntax, such that for any morphism $h : \Gamma \multimap_G \Delta$, give a term $\Delta, \Xi \vdash t : C$ we can construct a term of type C in context Γ, Ξ . The structural morphisms correspond to context shuffling. The gates and delays must be

embedded in the following way. Let us consider the case of a gate shifted by time e , ie. mapping context $e + \Gamma$ to $e + \Delta$. The corresponding term is:

$$\text{let box } e \, z = \text{box } e \, G(\Gamma) \text{ in let }_{e+\Delta} z = z \text{ in } t.$$

To represent composition, one can always apply one such transformation and then the other.

Another possible choice is to represent Circ morphisms in Einstein notation, commonly used to represent tensor expressions in physics. It is clear that plain tensors are representable this way, but pulse schedules can be represented using Einstein notation just as well, because every free SMC can be represented this way. In Einstein notation, every input *and* output variable is made explicit. Every morphism is annotated with its inputs as subscripts and outputs as superscript. So a circuit composed of a single gate $H : (x : q_1, y : q_2 \multimap_G z :^d q_1, w :^d q_2)$ is written H_{xy}^{zw} . The composition $f \circ g$ is written as a multiplication $F_{inputs}^{intermediate} G_{intermediate}^{outputs}$. Unitors, associators, and exchange morphisms are represented by the appropriate Kronecker deltas. Then the laws of the SMCs correspond to simplifications of such Kronecker deltas. After such simplifications, testing the equality of morphisms in Einstein notation is a simple syntactic test, up to renaming of intermediate variable names. We direct the reader to [5] for details on conversions back and forth between Einstein notation and morphisms in an arbitrary SMC.

In the interests of implementation and formal verification, however, a variable-free representation is used for scheduling. This representation is described in Section 6.3.

5 Hardware Model and Correctness

A *hardware model* specifies the action of a complete pulse schedule (given as a function $\text{Time} \rightarrow \text{Channel} \rightarrow \mathbb{R}$) on the state of a quantum chip. If the chip has n qubits, then the state of the chip is given by a unit vector in $\mathbb{C}^{2^n}$, and inputting a pulse schedule will perform a certain unitary operator $U : \mathbb{C}^{2^n} \multimap \mathbb{C}^{2^n}$ on the state.

In principle, we can calculate this unitary from the Hamiltonian of the chip, which is determined by the chip's circuit design. If there are k input channels $V_1(t), \dots, V_k(t)$ (where $V_i(t)$ is the voltage applied on input channel i at time t), then the Hamiltonian can be given in the form of a linear operator $H(V_1(t), \dots, V_k(t)) : \mathbb{C}^{2^n} \rightarrow \mathbb{C}^{2^n}$.

The evolution of the state of the quantum chip is then given by $|\phi(t)\rangle = U(t)|\phi(0)\rangle$, where $|\phi(t)\rangle \in \mathbb{C}^{2^n}$ is the state at time t and $U(t)$ is the unitary

$$U(t) = \exp \left(-\frac{i}{\hbar} \int_0^t H(V_1(t'), \dots, V_k(t')) dt' \right) .$$

Given a quantum chip whose Hamiltonian is given by the above, we say that the pulse schedule given by $V_1(t'), \dots, V_k(t')$, with start time 0 and end time t , *implements* the unitary $U(t)$.

► **Theorem 25 (Correctness).** *Assume that we have a model of the plain language in Hilbert spaces in which every gate G is assigned the unitary U_G ; and a model of the annotated language in $\mathcal{S}$ such that, for every gate G , the morphism f_G is a pulse schedule that implements U_G . Then the diagram (1) commutes.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash_d t : A$ and relies on the following observations: If pulse schedule p implements U and q implements V then $q \circ p$ implements $V \circ U$. If p implements U and q implements V then $p \otimes q$ implements $U \otimes V$. We always have p and $\text{Shift}(d)(p)$ implement the same unitary. ◀

Therefore, if a scheduling algorithm (a function from the plain language to the annotated language) is a right inverse to erasure, then the pulse schedule generated implements the same unitary as the original circuit.

6 Intermediate Representations

Intermediate representations in the dependently-typed programming language Agda [20] are introduced in order to facilitate the implementation of scheduling; they are isomorphic respectively to the plain and annotated surface languages ($\Gamma \vdash t : A$ and $\Gamma \vdash_d t : A$) up to normalisation, but avoid redexes while being more normalised than a free SMC generated by qubits and gates. The source code is available online;⁵ listings in this section make minor cosmetic changes to improve readability.

6.1 Plain circuits

The plain language is represented through the type family `Circuit`, indexed by two lists of qubits indicating the types of the circuit's input and output.

■ Listing 1 `Circuit`.

```
data Circuit : Qubits → Qubits → Set where
  wire : Circuit a a
  gate : Gate a → Circuit a a
  serial : Circuit a b → Circuit b c → Circuit a c
  parallel :
    par-qubits a1 a2 a3 →
    par-qubits b1 b2 b3 →
    Circuit a1 b1 →
    Circuit a2 b2 →
    Circuit a3 b3
```

The type `par-qubits x y z` used in the `parallel` constructor maps each qubit to a bijection between its occurrences in `x` or `y` and its occurrences in `z`.

It is defined as follows.

```
par-qubits : Qubits → Qubits → Qubits → Set
par-qubits x y z = (q : Qubit) → Bij ((q ∈ x) ∪ (q ∈ y)) (q ∈ z)
```

This is equivalent to permutations from the concatenation of x and y to z , meaning that the `parallel` constructor captures association, unitors, and exchange.

As is the case for the plain language, the `Circuit` datatype does not preclude multiple occurrences of the same qubit, but is well-behaved in that it treats occurrences as distinct individuals (informally, they are never confused with one another, copied, or destroyed; the definition of `par-qubits` plays a significant role in this). As a result, issues of duplication can be safely confined to the observation that an element of type `Circuit a b` is physically realizable exactly when `a` (or equivalently, `b`) is free of duplicates.

6.2 Annotated circuits

In order to describe the annotated language, the notion of a *timing function* is introduced. A timing function on a list x of qubits is a function that takes a qubit q and an element of $q \in x$, i.e. the index of an occurrence of q in x , and produces a time value:

⁵ Repository: <https://codeberg.org/radams78/grampus.git>

■ **Listing 2** Timing.

```
Timing : Qubits → Set
Timing qs = (q : Qubit) → q ∈ qs → Time
```

The annotated language can then be represented in a similar way to the plain language, but with as additional type indices two timing functions f and g ; f sends each input qubit to the time the annotated circuit needs it, and g sends each output qubit to the time the annotated circuit is done with it.

If an annotated circuit were depicted graphically as a schedule of gates, with qubits on the y-axis and time on the x-axis, the two timing functions would represent the schedule's left and right boundaries. Accordingly, they are referred to respectively as the annotated circuit's *left-timing* and *right-timing*. With respect to a typing judgement $\Gamma \vdash_d t : A$, f and g capture annotations in Γ as well as boxes in Γ and A .

The datatype – a possible representation of the category Circ – is then defined as follows:

■ **Listing 3** AC.

```
data AC : (a b : Qubits) → Timing a → Timing b → Set where
  delay : f ≤ g → AC a a f g
  gate :
    (gt : Gate a) →
      constantly t f →
      constantly (t + duration gt) g →
      AC a a f g
  serial : AC a b f g → AC b c g h → AC a c f h
  parallel :
    (pa : par-qubits a1 a2 a3) →
    (pb : par-qubits b1 b2 b3) →
    (par-timing pa f1 f2 f3) →
    (par-timing pb g1 g2 g3) →
    AC a1 b1 f1 g1 →
    AC a2 b2 f2 g2 →
    AC a3 b3 f3 g3
```

The type `constantly t f` asserts that f gives time t for each qubit in its domain, and the relation $\leq$ on timing functions is defined pointwise. For the `gate` constructor it is noted that the uses of `constant` mean the gate uses all its qubits over the same time interval; graphically, its schedule is a rectangle, that is to say its left and right edges are flat rather than ragged.

The type `par-timing` ensures that timing functions agree for each qubit occurrence across the bijection specified by `par-qubits`, motivated both practically by the encoding of timing functions, and also by the interest in distinguishing occurrences of qubits as described above in relation to the definition of `par-qubits`.

Finally, annotations can be erased: each constructor of `AC` is simply replaced by its counterpart in `Circuit` (with `delay` becoming `wire`). This is carried out by the function `erase-timing`, with the following type:

■ **Listing 4** erase.

```
erase : AC a b f g → Circuit a b
```

6.3 As-late-as-possible scheduling

The *as-late-as-possible* scheduling algorithm is implemented through the function `alap`:

■ **Listing 5** As Late As Possible.

```
alap : (g : Timing b) → Circuit a b →  $\Sigma$  (Timing a) ( $\lambda f \rightarrow \text{AC } a \ b \ f \ g$ )
```

It takes a right-timing g and plain circuit as arguments, and returns both an annotated circuit and its left-timing f . It is implemented by structural induction on the circuit, packing the schedule as far to the right as possible and propagating timings from right to left.

The non-trivial case is that of a **gate**: gates operate on all of their qubits over the same time interval, hence their constant left and right timing functions in **AC**, but the provided right timing g is arbitrary (graphically: the gate is a rectangle, while g might be jagged). This is solved by first “flattening” g by inserting a delay with right-timing g and constant left-timing $h = \text{const}(\min(g))$. The gate is then scheduled left of this delay, i.e. with right-timing h . It is observed that the **delay** will have duration zero for at least one qubit occurrence (where g attains its minimum), so we could not have hoped to schedule the gate later, and `alap` is indeed performing as-late-as-possible scheduling as claimed.

While the full definitions of equivalence on **Circuit** and **AC** are not yet complete, a rudimentary notion of equivalence, written $\approx$, is defined on **Circuit**: it is the smallest relation that is reflexive, respected by **serial** and **parallel**, and equates **serial ckt wire** with **ckt**. This form of equivalence is sufficient to show that `alap` (with some decoration) is a valid scheduling algorithm, which is to say a right inverse to erasure:

■ **Listing 6** erase-alap.

```
erase-alap : (g : Timing b) (ckt : Circuit a b) →  
  (erase  $\circ$  proj2  $\circ$  alap g) ckt  $\approx$  ckt
```

From the result at the end of Section 5 we then know that a pulse schedule generated from the annotated output of `alap` implements the same unitary as the original circuit.

7 Related Work

There are a growing number of quantum programming languages in existence now, both imperative languages such as Q# [17], and functional languages such as LIQUI|> [25], Quipper [11] and Proto-Quipper [23]. All of these languages have as their intended semantics *quantum circuits*. They provide the user with high-level tools for describing the construction of quantum algorithms. They thus correspond to what we called the *plain language* in this paper.

There have been a few languages that use type theory in their design, including Proto-Quipper [10], Qwire [21] and Silq [6]. The survey by Nguyen [18] provides a review of the applications of type theory to quantum programming language design. Again, these languages describe quantum circuits without timing information, and so correspond to our plain language.

The pulse schedules are sent in the form of an extremely low-level language such as Q1ASM [22] to the wavefunction generator that creates the signals passed down the input channels of the quantum hardware itself.

There are very few languages that represent a quantum circuit with timing information and explicit delays – the intermediate stage between a quantum circuit (without timing information) and a language such as Q1ASM. The only such language that the authors are aware of is OpenPulse [16, 3], until recently a subset of OpenQASM [9].

We therefore see GRAMPUS as filling a need of a language that represents quantum circuits with timing information, in a form that is suitable for formal semantics and for building a verified quantum compiler.

The design of GRAMPUS was heavily inspired by Granule [12], an implementation of graded modal type theory in Haskell.

8 Conclusion

We have given two linear type theories that can represent quantum circuits with timing information. We have given a model in terms of the input signals to a quantum chip, in such a way that the interpretation function is the algorithm that perform compilation. We have also proved that the term models are the initial models, giving confidence that the syntax and semantics are in close correspondence.

For future work, we want to expand our language GRAMPUS with the other features used in quantum algorithms, in particular measurement, and the ability to perform hybrid algorithms where the result of a measurement affects which gates are subsequently performed.

References

- 1 Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proceedings of the ACM on Programming Languages*, 4(ICFP):90:1–90:28, 2020. doi:10.1145/3408972.
- 2 Robin Adams, Jean-Philippe Bernardy, Lorenzo Perticone, and Jeremy Pope. Graded Modal Type Theory for Pulse Schedules. Software, This research was funded by the SSF (Swedish Foundation for Strategic Research) project QuantumStack, grant number FUS21-0063., sw-hId: swh:1:dir:bc79e493a8f11d72e5382289e1ef8d1284024082 (visited on 2026-07-23). URL: <https://codeberg.org/radams78/grampus.git>, doi:10.4230/artifacts.27329.
- 3 Thomas Alexander, Naoki Kanazawa, Daniel J. Egger, Lauren Capelluto, Christopher J. Wood, Ali Javadi-Abhari, and David C. McKay. Qiskit Pulse: Programming quantum computers through the cloud with pulses. *Quantum Science and Technology*, 5(4):044006, August 2020. doi:10.1088/2058-9565/aba404.
- 4 Nick Benton, Gavin Bierman, Valeria De Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. In Marc Bezem and Jan Friso Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 1993. doi:10.1007/BFb0037099.
- 5 Jean-Philippe Bernardy and Patrik Jansson. Domain-specific tensor languages. *Journal of Functional Programming*, 35:e9, 2025. doi:10.1017/S0956796825000048.
- 6 Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, PLDI 2020, pages 286–300, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386007.
- 7 Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coefficient calculus. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014. doi:10.1007/978-3-642-54833-8_19.

- 8 Matteo Capucci and Bruno Gavranović. Categories for the working amthematician, 2024. [arXiv:2203.16351](#).
- 9 Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heide, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):12:1–12:50, September 2022. doi:10.1145/3505636.
- 10 Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. In Ivan Lanese and Mariusz Rawski, editors, *Reversible Computation - 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings*, volume 12227 of *Lecture Notes in Computer Science*, pages 153–168, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-52482-1_9.
- 11 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A scalable quantum programming language. In Hans-Juergen Boehm and Cormac Flanagan, editors, *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 333–342, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491956.2462177.
- 12 Jack Hughes and Dominic Orchard. Program synthesis from graded types. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 83–112, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-57262-3_4.
- 13 P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver. A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews*, 6(2):021318, June 2019. doi:10.1063/1.5089550.
- 14 Saunders Mac Lane. Natural associativity and commutativity. *Rice Institute Pamphlet - Rice University Studies*, 49:28–46, 1963.
- 15 Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate texts in mathematics. Springer-Verlag, New York, NY, 1971.
- 16 David C. McKay, Thomas Alexander, Luciano Bello, Michael J. Biercuk, Lev Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, and Jay M. Gambetta. Qiskit backend specifications for openqasm and openpulse experiments, 2018. [arXiv:1809.03452](#).
- 17 Microsoft. *Q# Language Specification*, 2020. URL: <https://github.com/microsoft/qsharp-language/tree/main/Specifications/Language#q-language>.
- 18 Van Han Nguyen. A survey of quantum type theory: From linearity to formal verification. *EAI Endorsed Transactions on Context-aware Systems and Applications*, 10, July 2025. doi:10.4108/eetcasa.9669.
- 19 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000. URL: <https://www.cambridge.org/de/academic/subjects/physics/quantum-physics-quantum-information-and-quantum-computation/quantum-computation-and-quantum-information-10th-anniversary-edition?format=HB>.
- 20 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
- 21 Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, POPL '17, pages 846–858, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009894.

- 22 QBLOX BV. Q1ASM user guide. https://docs.qblox.com/en/v2026.04.0/products/qblox_instruments/q1/, 2026.
- 23 Neil J. Ross. Algebraic and logical methods in quantum computation, 2017. [arXiv:1510.02198](#).
- 24 Taichi Uemura. A general framework for the semantics of type theory. *Mathematical Structures in Computer Science*, 33(3):134–179, March 2023. [doi:10.1017/s0960129523000208](#).
- 25 Dave Wecker and Krysta M. Svore. LIQ|U|>: A software design architecture and domain-specific language for quantum computing, 2014. [arXiv:1402.4467](#).

9 Appendix – Commuting Conversions

The full list of commuting conversions is given below. Each one should be given as a rule of deduction with premises that ensure the left-hand side is well-typed (which ensures the right-hand side is well-typed too). We omit these to save space here.

$$\begin{aligned}
& (\text{let } * = s \text{ in let } * = t \text{ in } u) = (\text{let } * = \text{let } * = s \text{ in } t \text{ in } u) \\
& (\text{let } * = s \text{ in let } * = t \text{ in } u) = (\text{let } * = t \text{ in let } * = s \text{ in } u) \\
& (\text{let } * = s \text{ in } G(t_1, \dots, t_n)) = G(t_1, \dots, t_{i-1}, \\
& \quad \text{let } * = s \text{ in } t_i, \\
& \quad t_{i+1}, \dots, t_n) \\
& (\text{let } * = s \text{ in } (t, u)) = (\text{let } * = s \text{ in } t, u) \\
& (\text{let } * = s \text{ in } (t, u)) = (t, \text{let } * = s \text{ in } u) \\
& (\text{let } * = s \text{ in let } (x, y) = t \text{ in } u) = (\text{let } (x, y) = \text{let } * = s \text{ in } t \text{ in } u) \\
& (\text{let } * = s \text{ in let } (x, y) = t \text{ in } u) = (\text{let } (x, y) = t \text{ in let } * = s \text{ in } u) \\
& (\text{let } * = s \text{ in let box } d \ x = t \text{ in } u) = (\text{let box } d \ x = \text{let } * = s \text{ in } t \text{ in } u) \\
& (\text{let } * = s \text{ in let box } d \ x = t \text{ in } u) = (\text{let box } d \ x = t \text{ in let } * = s \text{ in } u) \\
& (\text{let } (x, y) = s \text{ in let } * = t \text{ in } u) = (\text{let } * = \text{let } (x, y) = s \text{ in } t \text{ in } u) \\
& (\text{let } (x, y) = s \text{ in let } * = t \text{ in } u) = (\text{let } * = t \text{ in let } (x, y) = s \text{ in } u) \\
& (\text{let } (x, y) = s \text{ in } G(t_1, \dots, t_n)) = G(t_1, \dots, t_{i-1}, \\
& \quad \text{let } (x, y) = s \text{ in } t_i, \\
& \quad t_{i+1}, \dots, t_n) \\
& (\text{let } (x, y) = s \text{ in } (t, u)) = (\text{let } (x, y) = s \text{ in } t, u) \\
& (\text{let } (x, y) = s \text{ in } (t, u)) = (t, \text{let } (x, y) = s \text{ in } u) \\
& (\text{let } (x, y) = s \text{ in let } (z, w) = t \text{ in } u) = (\text{let } (z, w) = \text{let } (x, y) = s \text{ in } t \text{ in } u) \\
& (\text{let } (x, y) = s \text{ in let } (z, w) = t \text{ in } u) = (\text{let } (z, w) = t \text{ in let } (x, y) = s \text{ in } u) \\
& (\text{let } (x, y) = s \text{ in let box } d \ z = t \text{ in } u) = (\text{let box } d \ z = \text{let } (x, y) = s \text{ in } t \text{ in } u) \\
& (\text{let } (x, y) = s \text{ in let box } d \ z = t \text{ in } u) = (\text{let box } d \ z = t \text{ in let } (x, y) = s \text{ in } u) \\
& (\text{let box } d \ z = s \text{ in let } * = t \text{ in } u) = (\text{let } * = \text{let box } d \ z = s \text{ in } t \text{ in } u)
\end{aligned}$$

$$\begin{aligned}
& (\text{let box } d \ z = s \text{ in let } * = t \text{ in } u) = (\text{let } * = t \text{ in let box } d \ z = s \text{ in } u) \\
& (\text{let box } d \ z = s \text{ in } G(t_1, \dots, t_n)) = G(t_1, \dots, t_{i-1}, \\
& \qquad \qquad \qquad \text{let box } d \ z = s \text{ in } t_i, \\
& \qquad \qquad \qquad t_{i+1}, \dots, t_n) \\
& (\text{let box } d \ z = s \text{ in } (t, u)) = (\text{let box } d \ z = s \text{ in } t, u) \\
& (\text{let box } d \ z = s \text{ in } (t, u)) = (t, \text{let box } d \ z = s \text{ in } u) \\
& (\text{let box } d \ z = s \text{ in let } (x, y) = t \text{ in } u) = (\text{let } (x, y) = \text{let box } d \ z = s \text{ in } t \text{ in } u) \\
& (\text{let box } d \ z = s \text{ in let } (x, y) = t \text{ in } u) = (\text{let } (x, y) = t \text{ in let box } d \ z = s \text{ in } u) \\
& \qquad (\text{let box } d \ z = s \text{ in box } d' \ t) = \text{box } d' \ (\text{let box } d \ z = s \text{ in } t) \\
& (\text{let box } d' \ z = s \text{ in let box } d \ x = t \text{ in } u) = (\text{let box } d \ x = t \text{ in let box } d' \ z = s \text{ in } u)
\end{aligned}$$

Formalisation and Extension of Lagois Connections for Secure Information Flow

Casper Ståhl  

Aalborg University, Denmark

Levs Gondelman  

Aalborg University CPH, Copenhagen, Denmark

René Rydhof Hansen  

Aalborg University, Denmark

Danny Bøgsted Poulsen  

Aalborg University, Denmark

Abstract

Lagois connections have been proposed as a way to formulate and formalise a secure information flow policy between two organisations each with their own information flow policies that must also be enforced. By composing Lagois connections in a chain, it is possible to compose the information flow policies of several organisations. However, chains are not always suitable. For example, if one of the “links” in the chain enforces a coarser policy than the surrounding links, this corresponds to taking low security information as input and reclassifying it as high security, preventing further communication of (formerly low) information at the low security level. This is an instance of the well-known problem of “label creep”.

In this paper we extend the compositionality results of Lagois connections to *forests*, and graphs that “behave” like forests, of Lagois connections. In addition to the extra flexibility gained in the communication topology, it also solves the label creep problem: Intuitively, to bypass the problematic intermediary, we can verify that the subgraph bypassing the problematic intermediary is located in a forest of secure sub-graphs. We show that this is sound with respect to noninterference, by proving that if information in a program flows according to a secure graph then an attacker can never observe a policy violation in the system.

As a further contribution of this paper, all of the above has been formalised in the Rocq proof assistant, including a novel formalisation of a substantial part of the literature of Lagois connections, providing a strong foundation for future work and implementations.

2012 ACM Subject Classification Security and privacy → Formal methods and theory of security; Theory of computation → Type theory

Keywords and phrases Lagois connection, information flow, security

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.6

Supplementary Material

Software (Source code): <https://github.com/CasperStaahl/LC4S-Rocq> [11]

archived at `swh:1:dir:828cd84332eae467073735e4aee31198815adf58`

1 Introduction

The distributed and interconnected nature of modern computing remains a challenge for the traditional approach to information flow security which often relies on a central notion or policy of how information ought to flow¹. In practice, organisations seldom have the resources or access to verify such a property for the totality of an entire system. Therefore, there is a need for compositional methods and theories of security, i.e., how does one combine secure

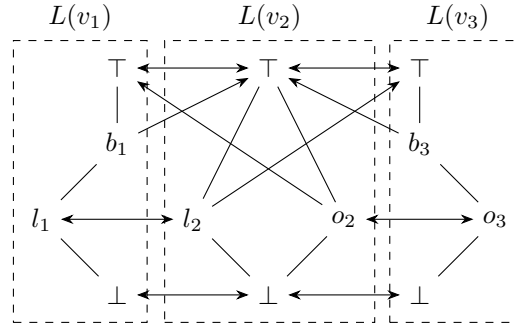
¹ This paper builds on and expands the work of the first author’s MSc thesis [10].



subsystems such that security is consequent in the aggregate system. One such theory is that of the Lagois connection for secure information flow, as originally presented by Bhardwaj and Prasad [2, 3]. But this theory is not without its own challenges, one of which we seek to address in this paper.

To illustrate, imagine a scenario in which two systems v_1 and v_2 wish to communicate securely with regards to information flow. Within systems v_1 and v_2 , the flow of information is governed by the respective security lattices $L(v_1)$ and $L(v_2)$ [4]. Now how do we model and enforce the flow of information when communication between v_1 and v_2 has to be taken into account? As shown by Bhardwaj and Prasad [2, 3] a *Lagois connection* between the two security lattices, $(L(v_1), f, g, L(v_2))$ provides a natural foundation for formalising and studying information flow security in such a context. Most importantly the Lagois connection describes communication that is secure in the sense that an organisation can never observe a violation of its own policy. For example in the case of v_1 one has $p \leq g(f(p))$ for all $p \in L(v_1)$ (analogous for v_2). Thus, the functions f and g capture the flow of information between v_1 and v_2 , while $L(v_1)$ and $L(v_2)$ continue to represent the internal flow of information within each organization.

Generalising the above to n systems $v_1, v_2, \dots, v_n$ who wish to communicate securely, Bhardwaj and Prasad have shown that secure Lagois connections can be *chained* together [2, 3]. To illustrate, consider three systems v_1, v_2 and v_3 with security lattices and Lagois connections as depicted below:



Intuitively we would like for v_1 to be able to share information with label b with members of v_3 with clearance b , and vice versa. But this is not possible because $L(v_2)$ acts as a lossy intermediate with respects to labels, swapping the order of the chain is no help either as $L(v_1)$ or $L(v_3)$ will become the lossy intermediate. This paper seeks to expand the use of Lagois connections exactly to overcome this limitation.

In particular we consider *graphs* of lattices connected by Lagois connections such that lossy intermediates can be bypassed. Not all such graphs are secure and we therefore identify graphs that are secure and ways in which secure graphs may be constructed. This model seems especially well suited for describing secure information flow in distributed systems and thus we contribute to consolidating the theory for such a setting by giving a semantic condition for when the flow of information in a distributed system respects a graph, and by showing that if the flow of information in a distributed system respects a secure graph then the system observes a suitable notion of noninterference. The preceding contributions have all been formalized in the Rocq proof assistant, providing a rigorous foundation for future work. In summary, the contributions of this paper are:

- A Rocq formalisation of Lagois connections and substantial parts of Melton et al. [6],
- A Rocq formalisation of the information flow framework of Bhardwaj and Prasad [2],
- An extension of the composability of Lagois-based information flow policies to more complex topologies, e.g., virtual Lagois forests, also formalised in Rocq.

2 Lagois Connections

In this section we introduce Lagois connection, but first re-visit and define several concepts from order theory, in particular partial orders, monotonic functions and poset systems, that underlie Lagois connections.

► **Definition 1** (Partial order). *A partial order is a tuple $(A, \leq)$ where A is a set and $\leq$ is a relation on A such that*

- $x \leq x$ for all $x \in A$ ($\leq$ is reflexive),
- $x \leq y \implies y \leq z \implies x \leq z$ for all $x, y, z \in A$ ($\leq$ is transitive), and
- $x \leq y \implies y \leq x \implies x = y$ for all $x, y \in A$ ($\leq$ is antisymmetric).

For a partial order $P = (A, \leq)$ we will omit explicitly mentioning A or P when the context is clear. Further, we will overload the symbol $\leq$ when multiple orders are present and the context is clear.

► **Definition 2** (Monotone function). *For two partial orders P and Q , a function $f : P \rightarrow Q$ is monotone whenever $p \leq p'$ implies $f(p) \leq f(p')$.*

► **Definition 3** (Poset system). *A poset system is a tuple (P, f, g, Q) of two partial orders P and Q and monotonic functions $f : P \rightarrow Q$ and $g : Q \rightarrow P$.*

Security with respect to information flow is typically formulated as *information flow policies* formalised in terms of (security-)lattices, as prescribed by Denning [4].

► **Definition 4** (Lattice). *A lattice is tuple $(A, \sqcup, \sqcap)$ where A is a partial order, $-\sqcup- : A \rightarrow A \rightarrow A$ is a join operator and $-\sqcap- : A \rightarrow A \rightarrow A$ is a meet operator such that*

- $x \sqcup y \leq z \iff x \leq z \wedge y \leq z$ for all $x, y, z \in A$, and
- $x \leq y \sqcap z \iff x \leq y \wedge x \leq z$ for all $x, y, z \in A$.

For information flow purposes, the lattices are assumed to be finite and inhabited. Taking Denning's perspective, given a lattice A information is allowed to flow from an object labeled $x \in A$ to an object labeled $y \in A$ if $x \leq y$, i.e., information is only allowed to flow from lower levels to higher levels. This object with label x could for example be a document, and the other object labeled y could be an employee. Typically the objects are program variables in an imperative programming language.

With the preliminaries covered we now introduce the Lagois connection formally.

► **Definition 5** (Lagois connection [6]). *A poset system (P, f, g, Q) is called a Lagois connection whenever:*

$$p \leq (g \circ f)(p) \text{ for all } p \in P, \tag{LC1}$$

$$q \leq (f \circ g)(q) \text{ for all } q \in Q, \tag{LC2}$$

$$f \circ g \circ f = f, \text{ and} \tag{LC3}$$

$$g \circ f \circ g = g. \tag{LC4}$$

For two organisations v and v' , we can view a Lagois connection $(L(v), f, g, L(v'))$ as a secure, reciprocal and converging translation between the labels of the two lattices, that is, we can use $f : L(v) \rightarrow L(v')$ and $g : L(v') \rightarrow L(v)$ to securely translate between $L(v)$ and $L(v')$ [2, 3].² Equivalently we can see a Lagois connection $(L(v), f, g, L(v'))$ as a policy for

² By “reciprocal” we are referring to what is called “precision” in [2, 3].

cross communication between v and v' . In particular, information in v labeled $p \in L(v)$ is allowed to flow as specified by the lattice $L(v)$ but also from p to $q \in L(v')$ whenever $f(p) \leq q$ (analogous for g), and via the transitive closure of the two ordering relations. Properties **LC1** and **LC2** ensure that the connection is secure:

$$p \leq (g \circ f)(p) \text{ for all } p \in P \quad (\text{SC1})$$

$$q \leq (f \circ g)(q) \text{ for all } q \in Q \quad (\text{SC2})$$

Intuitively, translating a label back and forth will not violate the local policy nor will sending a piece of information back and forth ever accidentally declassify it.

When labels are translated back and forth between systems, properties **LC3** and **LC4** ensure *precision* and *convergence*, i.e., that the translation does not give rise to *label creep* and unnecessary over-approximation. This is mainly relevant for enforcement of security, e.g., through type systems or static analysis, but not for the purposes of this paper. We therefore only include them here for completeness and refer to [10, Appendix A] for a more detailed discussion. The properties below, following from **LC3** and **LC4**, ensure precision of the Lagois connection:

$$f(p) = \bigsqcup g^{-1}(p) \text{ for all } p \in g[Q] \quad (\text{PC1})$$

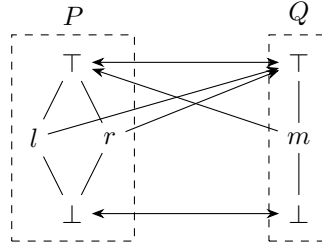
$$g(q) = \bigsqcup f^{-1}(q) \text{ for all } q \in f[P] \quad (\text{PC2})$$

Where $g^{-1}(p)$ denotes the preimage of p with respect to g and $g[Q]$ denotes the image of Q with respect to g . Further, **LC3** and **LC4** ensures that the connection is converging:

$$g(f(p)) = p' \implies g(f(p')) = p' \text{ for all } p, p' \in P \quad (\text{CC1})$$

$$f(g(q)) = q' \implies f(g(q')) = q' \text{ for all } q, q' \in Q \quad (\text{CC2})$$

► **Example 6.** Figure 1 displays a poset system (P, f, g, Q) that is a Lagois connection. P and Q are shown as their Hasse diagram. f is indicated by arrows going from P to Q and vice versa for g .



■ **Figure 1** The Lagois connection (P, f, g, Q) .

Communication rarely occurs between two system in isolation, therefore it makes sense to extend the framework of Lagois connection for secure information flow to an arbitrary (finite) number of systems. Bhardwaj and Prasad propose composition of Lagois connections as a method for such extensions [3] and define composition of Lagois connections as follows:

$$(P, f, g, Q) \circ (Q, \hat{f}, \hat{g}, R) = (P, \hat{f} \circ f, \hat{g} \circ g, R) \quad (1)$$

To formalise and show that security is preserved by composing Lagois connections, Bhardwaj and Prasad present the following theorem by Melton et al. [6]:

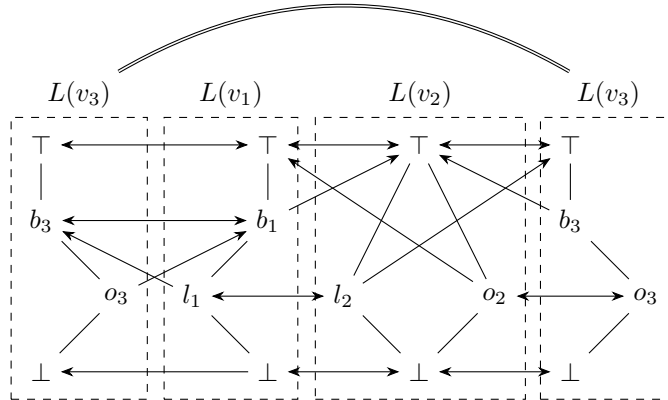
► **Theorem* 7** (Theorem 11 in [6]). *Let (P, f, g, Q) and $(Q, \hat{f}, \hat{g}, R)$ be Lagois connections. Then $(P, \hat{f} \circ f, \hat{g} \circ g, R)$ is a Lagois connection iff $\hat{g} \circ \hat{f} \circ f[P] \subseteq f[P]$ and $f \circ g \circ \hat{g}[R] \subseteq \hat{g}[R]$.*

Slightly disappointing is the fact that not all Lagois connections compose to form Lagois connections. However, the composition of secure connections (as in satisfying **LC1** and **LC2**) is always secure:

► **Proposition* 8.** *Let (P, f, g, Q) and $(Q, \hat{f}, \hat{g}, R)$ satisfy **LC1** and **LC2**. Then $(P, \hat{f} \circ f, g \circ \hat{g}, R)$ satisfies **LC1** and **LC2**.*

Proof. Fix a $p : P$. Have $f(p) \leq (\hat{g} \circ \hat{f} \circ f)(p)$ by **LC1** on $(Q, \hat{f}, \hat{g}, R)$. Then, by monotonicity of g and **LC1** on (P, f, g, Q) we have $p \leq (g \circ f)(p) \leq (g \circ \hat{g} \circ \hat{f} \circ f)(p)$ i.e. **LC1**. The proof of **LC2** is analogous. ◀

More importantly as pointed out in section 1, this way of chaining Lagois connections to connect multiple systems securely also has an unintended side effect, as each translation of a label may introduce imprecision, e.g., by merging labels. Essentially we want to bypass the intermediate lattice when possible. This motivates us to study graphs where each vertex has an associated lattice and when two vertices are connected in the graph their underlying lattices are connected by some Lagois connection. Continuing the example from section 1, Figure 2 shows such a graph. Here $L(v_1), L(v_2)$ and $L(v_3)$ can communicate directly with one another, without introducing imprecision in the label information potentially caused by an intermediate. Is this secure³? In this case the pattern of communication is secure, but generally this is not the case⁴.



■ **Figure 2** A Lagois graph.

3 Lagois Graphs and Security

In this section, we define the notion of *Lagois graph* to formalise the communication between systems with different security policies. We further define and investigate what it means for communication to be secure in such a system defined by a Lagois graph. In particular, we develop several notions of security capturing different intentions and intuitions about secure communication, but conclude by showing they are all logically equivalent. We start by reviewing and formalising essential concepts from graph theory.

³ See Definition 15 for a definition of security.

⁴ See Figure 3 for a graph that is not secure.

► **Definition 9 (Graph).** A graph is a tuple $G = (V, E)$ where V is a finite inhabited set of vertices and E is a edge relation on V such that:

- $(v, v) \notin E$ for all $v \in V$ (E is irreflexive),
- if $(v, v') \in E$ then $(v', v) \in E$ for all $v, v' \in G$ (E is symmetric).

For convenience, the basic graphs we work with are undirected (symmetric) graphs with no self-loops (irreflexive), as defined above. For a graph $G = (V, E)$ we let G denote both the graph and V when this causes no confusion.

► **Definition 10 (Path).** For a graph $G = (V, E)$ a path is either the empty path, denoted ϵ , or a sequence of edges $(v_1, v_2), (v_2, v_3) \dots, (v_n, v_{n+1}) \in E^*$.

We will mainly be working with two operations on paths, reversal $(-)^{-1}$ and concatenation $- \star -$. For a path p we let $|p|$ denotes its length. Certain paths turn out to be useful, namely simple paths and loops.

► **Definition 11 (Simple path).** A simple path is either the empty path ϵ or a path $(v_1, v_2), (v_2, v_3), \dots, (v_n, v_{n+1})$ where each vertex is distinct $i \neq j \implies v_i \neq v_j$.

► **Definition 12 (Loop).** A path $(v_1, v_2), (v_2, v_3) \dots, (v_n, v_{n+1})$ is a loop if $v_1 = v_{n+1}$.

Note that this definition allows for nested loops. Furthermore, the only simple loop from v to v is the empty path ϵ , as otherwise v would appear twice: at the beginning of the sequence and at the end.

We can now formally define the notion of *Lagois graph* as a graph of Lagois connections.

► **Definition 13 (Lagois graph).** A Lagois graph is a tuple (G, L, F) where

- $G = (V, E)$ is a graph,
- L is a function associating each vertex $v \in V$ to a finite inhabited lattice $L(v)$,
- F is a function associating each edge $e = (v, v') \in E$ to a monotone function $F(e) : L(v) \rightarrow L(v')$, and
- for each pair of symmetric edges $e = (v, v')$ and $e' = (v', v)$ in E we have that $(L(v), F(e), F(e'), L(v'))$ is a Lagois connection.

We will denote a Lagois graph (G, L, F) by G when it does not cause confusion. For some Lagois graph $G = (V, E)$ we can view each $v \in V$ as a system, actor, organisation, etc. who wishes to communicate securely with other $v' \in V$. Each system $v \in V$ has an associated security lattice $L(v)$ that specifies its local policy. When two systems $v, v' \in V$ are connected in the graph, i.e., we have $e = (v, v') \in E$ and consequently $e' = (v', v) \in E$, we assume that a policy for secure information exchange has been negotiated forming a Lagois connection $(L(v), F(e), F(e'), L(v'))$. Information is then allowed to flow via a flow relation that is induced by the structure of the graph and its underlying lattices and Lagois connections.

► **Definition 14 (Flow relation).** Each Lagois graph $G = (V, E)$ induces a flow relation for $v, v' \in V$ and $p \in L(v), q \in L(v')$ with judgments $(v, p) \Rightarrow (v', q)$ which is inductively defined as follows:

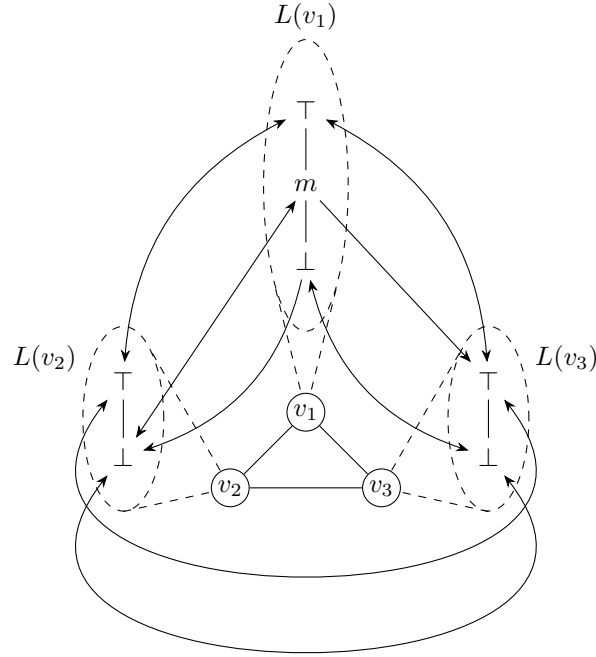
$$\frac{p \leq q}{(v, p) \Rightarrow (v, q)} \quad (\text{flowle}), \quad \frac{e = (v, v') \in E}{(v, p) \Rightarrow (v', F(e)(p))} \quad (\text{flowlc})$$

$$\frac{(v, p) \Rightarrow (v', r) \quad (v', r) \Rightarrow (v'', q)}{(v, p) \Rightarrow (v'', q)} \quad (\text{flowtran}).$$

For $(v, p) \Rightarrow (v', q)$ we drop v, v' when they can be inferred from the context, giving judgments $p \Rightarrow q$.

Rule `flowlc` tells us that information is allowed to flow according to local policies. Rule `flowlc` allows information to flow via the Lagois connections established by the Lagois graph. And rule `flowtran` is the transitive closure.

Figure 3 displays a Lagois graph $G = (V, E)$. The graph contains three elements $v_1, v_2, v_3 \in V$ which are drawn as the innermost three nodes in the diagram. The lattices $L(v_1), L(v_2), L(v_3)$ are displayed around the graph, and the functions inducing the Lagois connections are indicated by arrows. Notice that $\text{flowlc}(v_1, m) \Rightarrow (v_2, \perp) \Rightarrow (v_3, \perp) \Rightarrow (v_1, \perp)$, i.e., $(v_1, m) \Rightarrow (v_1, \perp)$ which *violates* the local policy of v_1 because $m \not\leq \perp$. Worryingly, it is intuitive to see that not all Lagois graphs are secure.



■ **Figure 3** An insecure Lagois graph.

This example gives rise to our first notion of security, similar to the one presented in [8, 9].

► **Definition 15** (Flow secure). *A graph G is said to be flow secure if for all $v \in G$ and $p, p' \in L(v)$ it is the case that $(v, p) \Rightarrow (v, p')$ implies $p \leq p'$.*

Essentially, a graph G is flow secure if for all vertices $v \in G$ all flows that start and end at v respects v 's local policy as specified by $L(v)$. This setup is somewhat similar to that of [8] and we conjecture that the cubic time algorithm (in the number of communicating nodes) presented there for verifying flow security of communication between programs, can be adapted to determine if a graph is flow secure.

We will now develop a second definition of security which is somewhat similar to **SC1** (or **SC2**). This definition of security turns out to be a stepping stone towards further results and as we will show is also logically equivalent to flow security. Before we can define this type of security we need extend the notion of a path inside a Lagois graph. A path $f = (v_1, v_2), (v_2, v_3), \dots, (v_n, v_{n+1})$ in a Lagois graph can similar to edges be treated as a function $\hat{f} : L(v_1) \rightarrow L(v_{n+1})$. $\hat{f}$ is intuitively computed as $\hat{f} = F(e_n) \circ \dots \circ F(e_2) \circ F(e_1)$ for $f = e_1, e_2, \dots, e_n$. Formally it is computed by induction on the sequence, with the base case $\hat{e} = \text{id}$. We can now state the second notion of security.

► **Definition 16** (Loop secure). *Loop security is defined in three levels:*

- A loop $f = (v, \_), \dots, (\_, v)$ is loop secure whenever $p \leq \hat{f}(p)$ for all $p \in L(v)$.
- A vertex $v \in G$ is loop secure if for all loops $f = (v, \_), \dots, (\_, v)$, f is loop secure.
- A Lagois graph G is loop secure if for all $v \in G$, v is loop secure.

As mentioned previously a graph being flow secure is logically equivalent to it being loop secure, as the following lemma and proposition show.

► **Lemma 17.** *If $p \Rightarrow q$ for some $p \in L(v)$, $q \in L(v')$, then there exists a path $f = (v, \_), \dots, (\_, v')$ such that $\hat{f}(p) \leq q$.*

Proof. By induction on the derivation of $p \Rightarrow q$. ◀

► **Proposition 18.** *A graph is flow secure iff it is loop secure.*

Proof.

→ Fix $v \in G$ a path $f = (v, \_) \dots, (\_, v)$ and $p \in L(v)$. Because there is a loop f there is a flow $p \Rightarrow \hat{f}(p)$. Thus by our assumption that flow is secure we have $p \leq \hat{f}(p)$.

← Fix $v \in G$ and $p, p' \in L(v)$ and assume that $p \Rightarrow p'$. By Lemma 17 there exists a path $f = (v, \_), \dots, (\_, v)$ such that $\hat{f}(p) \leq p'$. Thus we have $p \leq \hat{f}(p) \leq p'$ by our assumption that all loops are secure. ◀

For convenience, when it does not cause confusion, we will refer to a graph as *secure* if it is either flow secure or loop secure, as they are logically equivalent.

We can now derive the final notion of security, one that is very close to **SC1** (or **SC2**). This definition of security will again turn out to be logically equivalent to our other definitions of security.

► **Definition 19** (Simply secure). *A Lagois graph G is simply secure whenever for all pairs of vertices $v, v' \in G$ and for all simple paths $f = (v, \_), \dots, (\_, v')$ and $g = (v', \_), \dots, (\_, v)$ it is the case that $f \star g$ is secure.*

This notion might seem superfluous, especially given the results below showing that security and simple security are logically equivalent, but in the next section we will use it to prove that certain graphs are always secure without having to resort to strong induction in the future.

► **Lemma 20.** *For all paths f in a Lagois graph G , $\hat{f}$ is always monotone.*

Proof. By induction on the length of f . ◀

► **Proposition 21.** *A graph is simply secure iff it is secure.*

Proof.

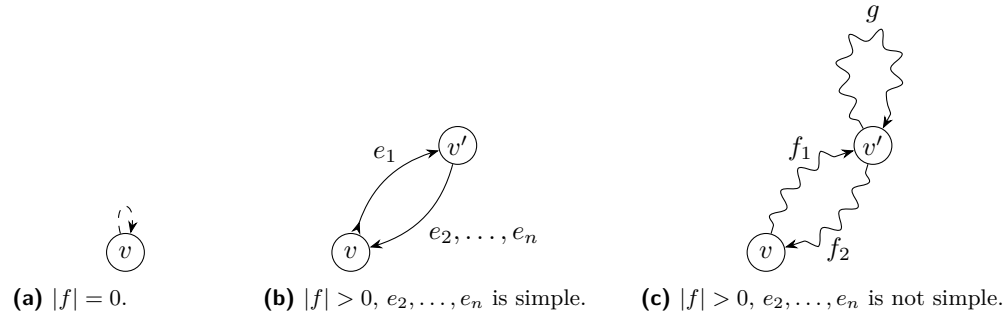
⇒ A secure graph is simply secure because all of its loops are secure.

⇐ Fix a loop $f = e_1, e_2, \dots, e_n$. We proceed by strong induction over the length of f , Figure 4 depicts all three cases.

$|f| = 0$ The empty path ϵ is trivially secure.

$|f| > 0$, $e_2, \dots, e_n$ is **simple** f is secure because the path e_1 is simple, $e_2, \dots, e_n$ is simple and because the graph is simply secure.

$|f| > 0$, $e_2, \dots, e_n$ is **not simple** We have $f = f_1 \star g \star f_2$ for some f_1, g, f_2 such that g and $f_1 \star f_2$ are loops strictly shorter than f . By strong induction we can assume g is secure, from which we can derive $\hat{f}_1(p) \leq \widehat{(f_1 \star g)}(p)$. By strong induction we can assume $f_1 \star f_2$ is secure, and then by monotonicity of paths we have $p \leq \widehat{(f_1 \star f_2)}(p) \leq \widehat{(f_1 \star g \star f_2)}(p)$. ◀



■ **Figure 4** The three different cases in the proof of Proposition 21. The empty paths ϵ are depicted with a little dashed arrow, simple paths are depicted with straight arrow, paths that consists of a single edge are depicted as a straight arrow with a tail, and general paths are depicted with a squiggly arrow.

As shown above, our notion of Lagois graph addresses the problem of label information loss that may occur when communicating through an intermediary with a coarse(er) flow policy, by allowing more general communication patterns than chains. However, this also removes the “automatic” guarantee of security that follows from simple chaining. In the following section, we adress this problem and identify communication patterns that can restore the “automatic” security guarantee as well as provide tools for analysing and arguing about security of yet other patterns.

4 Secure Graphs

In this section we identify a class of secure Lagois graphs that expand and generalise over the basic communication chains. We further prove results showing how Lagois graphs can be built in a way that guarantees security. For this purpose, we will build on the intuition provided by the following lemma.

► **Lemma 22.** $f \star f^{-1}$ is secure for all paths f .

Proof. By induction on the length of f . ◀

Lemma 22 intuitively says that flow of information is secure as long as it returns along the same path it came from. It will be fruitful to explore less strict versions of this principle. However, an object that immediately embodies this principle is that of a forest, thus we reiterate the definition here.

► **Definition 23 (Forest).** A graph G is called a forest if all simple paths between all pairs of vertices are unique.

The first subset of graphs that turns out to be secure is the one containing only virtual Lagois forests.

► **Definition 24 (Virtual Lagois forest).** A Lagois graph G is a virtual forest whenever for all vertices $v, v' \in G$ it is the case that for all simple paths $f = (v, \_), \dots, (\_, v')$ and $g = (v, \_), \dots, (\_, v')$ one has $\hat{f} = \hat{g}$.

Essentially a virtual Lagois forest is a Lagois graph in which for all $v, v' \in G$, all simple paths from v to v' have the same behavior when seen as functions. Said otherwise, if you go from v to v' via a simple path it does not matter which simple path you take in regards

to label translation. Practically speaking, this would correspond to a network in which information with the same initial label would always receive the same label after arriving at some other organization assuming it took a direct path.

► **Theorem 25.** *A virtual Lagois forest is secure.*

Proof. Fix a graph G , by Proposition 21 it is enough to consider all pairs of simple paths $f = (v, \_), \dots, (\_, v')$, $g = (v', \_), \dots, (\_, v)$ for all pairs of points $v, v' \in G$. $f \star f^{-1}$ is secure by Lemma 22 and therefore $f \star g$ is also secure because $\widehat{f^{-1}} = \hat{g}$ by the fact that G is a virtual forest. ◀

It immediately follows as a corollary that all Lagois forests are secure, i.e., a Lagois graph whose underlying graph is a forest is always secure.

► **Corollary 26.** *A Lagois forest is secure.*

Proof. A virtual Lagois forest are secure by Theorem 25 and a Lagois forest is a virtual Lagois forest because $f = g$ implies $\hat{f} = \hat{g}$. ◀

We can now effectively prove a stronger version of Proposition 8 by observing that a chain of composed Lagois connections is essentially a Lagois graph that is a path graph, and that a path graph is a forest which is secure as we have just shown. Further, Corollary 26 is interesting because it tells us that a Lagois graph can be determined to be secure by looking at the structure of the graph alone. Taking another view, security of an entire network can emerge from security ensured entirely between organizations in isolation.

Although (virtual) forests are secure, we have arrived at the same problem that we set out to solve in the first place: A lossy intermediate might still be present. In the case of a forest the lossy intermediate cannot be bypassed as there is only one direct path, and any indirection will intuitively only cause further loss of label information. In the case of the virtual forest it does not matter which path information takes as they all have the same effect: if one path from v to v' incurs some unwanted loss of label information then all paths from v to v' do.

Again employing the intuition from Lemma 22, the idea is to securely bypass intermediates locally in subgraphs and then stitch together these secure graphs in such a manner that information can only return to the subgraph the way it came. This notion of stitching graphs together is depicted in Figure 5, and we make it precise in the following manner.

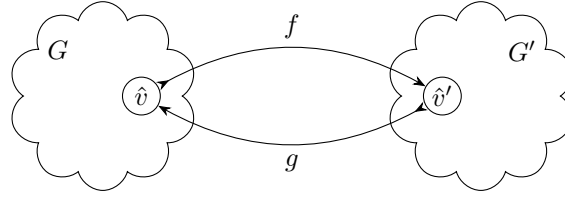
► **Definition 27** (Secure graph composition). *We define the secure composition of disjoint Lagois graphs G and G' with respect to vertices $\bar{v} \in G$ and $\bar{v}' \in G'$ and Lagois connection $(L(\bar{v}), f, g, L(\bar{v}'))$ as*

$$G \stackrel{f}{\underset{g}{\rightleftharpoons}}_{\bar{v}}^{\bar{v}'} G' = ((V, E), L, F) \stackrel{f}{\underset{g}{\rightleftharpoons}}_{\bar{v}}^{\bar{v}'} ((V', E'), L', F') \quad (2)$$

$$:= ((V'', E''), L'', F''), \quad (3)$$

where $V'' = V \cup V'$, $E'' = E \cup E' \cup \{(\bar{v}, \bar{v}'), (\bar{v}', \bar{v})\}$ and

$$L''(v) = \begin{cases} L(v) & \text{if } v \in V \\ L'(v) & \text{if } v \in V', \end{cases} \quad F''(e) = \begin{cases} F(e) & \text{if } e \in E \\ F'(e) & \text{if } e \in E' \\ f & \text{if } e = (\bar{v}, \bar{v}') \\ g & \text{if } e = (\bar{v}', \bar{v}). \end{cases} \quad (4)$$

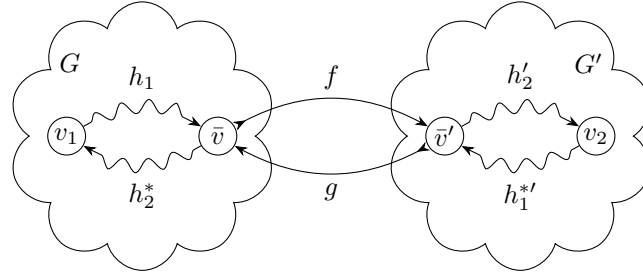


■ **Figure 5** Secure composition of Lagois graphs.

Note that there is no need to define the operation in terms of multiple graphs, as we can chain the operation freely. We can then show that the operation $G \stackrel{f}{\bar{v}} \stackrel{g}{\bar{v}'} G'$ preserves security.

► **Theorem 28.** *If G and G' are disjoint and secure graphs then for all $\bar{v} \in G$, $\bar{v}' \in G'$ and f, g such that $(L(\bar{v}), f, g, L(\bar{v}'))$ is a Lagois connection then $G \stackrel{f}{\bar{v}} \stackrel{g}{\bar{v}'} G'$ is secure.*

Proof. By Proposition 21 it is enough to consider all pairs of simple paths $h = (v_1, \_, \dots, \_, v_2)$, $h^* = (v_2, \_, \dots, \_, v_1)$ for all pairs of vertices $v_1, v_2 \in G \stackrel{f}{\bar{v}} \stackrel{g}{\bar{v}'} G'$. If each vertex resides exclusively within G then h and h^* must also be contained within G , because if either h or h^* were to pass over into G' it must have been through $e_f = (\bar{v}, \bar{v}') \in E''$ (and $e_g = (\bar{v}', \bar{v}) \in E''$ on the way back) contradicting the fact that they are simple (analogous for vertices residing exclusively within G'). If v_1 resides within G and v_2 resides within G' we can decompose $h = h_1 \star e_f \star h'_2$ and $h^* = h_1^* \star e_g \star h_2^*$ such that h_1, h_2^* are contained within G and h'_2, h_1^* are contained within G' , as depicted below.



Fix a $p \in L(v_1)$, now $(\widehat{h_1 \star e_f})(p) \leq (\widehat{h_1 \star e_f \star h'_2 \star h_1^*})(p)$ because $h'_2 \star h_1^*$ is secure by virtue of being a loop contained within G' . Then

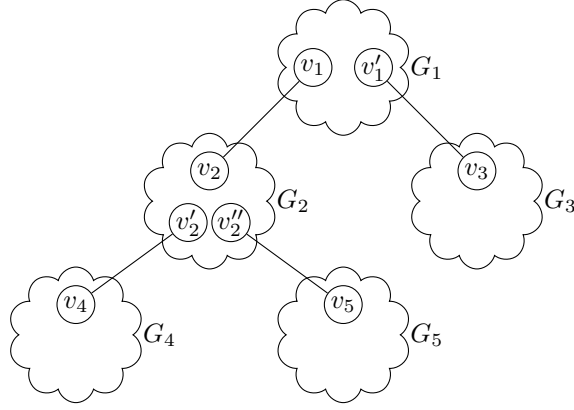
$$\hat{h}_1(p) \leq (\widehat{h_1 \star e_f \star e_g})(p) \leq (\widehat{h_1 \star e_f \star h'_2 \star h_1^* \star e_g})(p)$$

by security of $e_f \star e_g$ and monotonicity of e_g . Finally,

$$\begin{aligned} p &\leq (\widehat{h_1 \star h_2^*})(p) \\ &\leq (\widehat{h_1 \star e_f \star h'_2 \star h_1^* \star e_g \star h_2^*})(p) \\ &= (\widehat{h \star h^*})(p) \end{aligned}$$

because $h_1 \star h_2^*$ is secure by virtue of being contained within G and because h_2^* is monotone (analogous when v_1 resides in G' and v_2 resides in G). ◀

Essentially, Theorem 7 tells us that if a graph G is a forest of secure subgraphs $G_1, G_2, \dots, G_n$ then G is a secure graph, as depicted for some G in Figure 6. Theorem 28 is thus a generalisation of Corollary 26, in this regard it should be easy to see that Corollary 26 can be derived from Theorem 28 instead of Theorem 25.



■ **Figure 6** A forest of secure subgraphs.

5 Soundness (w.r.t. Noninterference)

As mentioned in section 1, the theory presented in the prequel seems well suited for describing secure information flow in distributed systems. Thus in this section we make the following notion precise: If information in some distributed program flows according to a secure Lagois graph G (i.e. the program respects G) then the distributed program observes noninterference. So far, our theory has been detached from any particular model of computation. We consider this a strength and seek to keep it that way. We take a knowledge-based approach, as shown by Askarov and Chong [1], in our description of when a program respects a graph and of noninterference. Unlike [1], the theory presented here is not parametric in the attacker, i.e., we assume that the attacker has perfect recollection.

Fix some Lagois graph G , a set $\mathcal{S}$ of program configurations, and a set of values N , then let the joint state of a distributed system over G be denoted by

$$\Sigma : \mathcal{S} = (v \in G) \rightarrow ((L(v) \rightarrow N) \times \mathcal{S}). \quad (5)$$

Where $\Sigma(v) = (\rho, x)$ is the local state at v and ρ provides the security interface for which an attacker is able to observe the state. Essentially an attacker at v on level $\ell \in L(v)$ will be able to observe $\rho(\ell')$ for all $\ell' \leq \ell$. We let x denote the actual internal configuration of the program at that state. Similarly we will refer to $\Gamma : (v \in G) \rightarrow L(v) \rightarrow N$ as a distributed security interface.

We further assume there exists a *computation relation* $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ modelling the states of execution of a program. A trace is defined to be a sequence $\Sigma s = \Sigma_1, \Sigma_2, \dots, \Sigma_n \in \mathcal{S}^*$ such that $\Sigma_1 \longrightarrow \Sigma_2 \longrightarrow \dots \longrightarrow \Sigma_n$. Appendix A provides an example instance of $(\mathcal{S}, \longrightarrow)$. Take π_i to be the i th projection of a tuple; then for a given distributed state Σ the state observable by some attacker at v on level $\ell \in L(v)$ is denoted by $\Sigma|_\ell^v : L(v) \rightarrow N$ which is defined as

$$\Sigma|_\ell^v(\ell') = \begin{cases} \pi_1(\Sigma(v))(\ell') & \text{if } \ell' \leq \ell, \\ \text{nothing} & \text{otherwise.} \end{cases} \quad (6)$$

Given trace Σs , a partial trace observable by some attacker at v on level $\ell \in L(v)$ is denoted by $\Sigma s|_\ell^v : (L(v) \rightarrow N)^*$ which is defined as

$$()|_\ell^v = () \quad (7)$$

$$(\Sigma \star \Sigma s)|_\ell^v = \begin{cases} \Sigma s|_\ell^v & \text{if } \Sigma s|_\ell^v = \Sigma|_\ell^v \star \sigma s \\ \Sigma|_\ell^v \star \Sigma s|_\ell^v & \text{otherwise.} \end{cases} \quad (8)$$

Essentially $\Sigma s|_\ell^v$ applies $-|_\ell^v$ pointwise to Σs and filters out adjacent duplicates; we justify this choice by appealing to the intuition that an observer should not be able to detect a change if nothing happened from the observer's point of view.

We then say that a distributed state Σ can emit a partial trace $\sigma s \in (L(v) \rightarrow V)^*$ observable by an attacker at v on level $\ell \in L(v)$ denoted $\Sigma \Downarrow_\ell^v \sigma s$ iff there exists a trace Σs such that $\Sigma \star \Sigma s$ is a trace and $(\Sigma \star \Sigma s)|_\ell^v = \sigma s$.

We can now describe when an initial distributed state respects G in relation to how communication takes place: Define for some v and $\ell \in L(v)$ the equivalence relations on distributed security interfaces where all information that may flow into (v, ℓ) is equal:

$$\Gamma \approx_\ell^v \Gamma' \iff \Gamma(v', \ell') = \Gamma'(v', \ell') \text{ for all } (v', \ell') \rightrightarrows (v, \ell) \quad (9)$$

This gives rise to an equivalence class (for each Γ):

$$[\Gamma]_\ell^v = \{\Gamma' \mid \Gamma \approx_\ell^v \Gamma'\}. \quad (10)$$

We define the knowledge of an attacker at v on level $\ell \in L(v)$ that is trying to guess the initial distributed security interface the system after observing σs as

$$\mathbb{K}_\ell^v(\sigma s) = \{\Gamma \mid \exists \Sigma. (\pi_1 \circ \Sigma) = \Gamma \wedge \Sigma \Downarrow_\ell^v \sigma s\}. \quad (11)$$

Now we are ready to define what it means for a distributed state to respect G in terms of information flow:

► **Definition 29.** *We say that Σ respects G whenever for all $v \in G$, $\ell \in L(v)$ and σs such that $\Sigma \Downarrow_\ell^v \sigma s$ it is the case that*

$$[\pi_1 \circ \Sigma]_\ell^v \subseteq \mathbb{K}_\ell^v(\sigma s). \quad (12)$$

Essentially Definition 29 tells us if some Σ respects G then the attacker may only learn in accordance to the flow of G . Said otherwise, an attacker at (v, ℓ) may only learn any information about $\pi_1(\Sigma(v'))(\ell')$ whenever $(v', \ell') \rightrightarrows (v, \ell)$.

The above is a very strong notion of noninterference, but not the one in which we are directly interested. Specifically, we want to ensure that a system at v may never observe a violation of its own policy $L(v)$. We define an equivalence relation over security interfaces at v that will seem the same to an attacker on level $\ell \in L(v)$ as follows

$$\rho \sim_\ell^v \rho' \iff \rho(\ell') = \rho'(\ell') \text{ for all } \ell' \leq \ell \in L(v). \quad (13)$$

As above, this gives rise to an equivalence class

$$[\rho]_\ell^v = \{\rho' \mid \rho \sim_\ell^v \rho'\}. \quad (14)$$

We then define the knowledge of an attacker at v on level $\ell \in L(v)$ that is trying to guess the initial security interface at their location as

$$K_\ell^v(\sigma s) = \{\rho \mid \exists \Sigma. \pi_1(\Sigma(v)) = \rho \wedge \Sigma \Downarrow_\ell^v \sigma s\}. \quad (15)$$

The notion of noninterference we are interested in can then be defined in the following manner:

► **Definition 30.** *We say that Σ observes noninterference whenever for all $v \in G$, $\ell \in L(v)$ and σs such that $\Sigma \Downarrow_\ell^v \sigma s$ it is the case that*

$$[\pi_1(\Sigma(v))]_\ell^v \subseteq K_\ell^v(\sigma s) \quad (16)$$

Thus, Σ observing noninterference intuitively means that an attacker at v on level $\ell \in L(v)$ interested in learning about the initial security interface at that location, i.e., $\pi_1(\Sigma(v)) = \rho$, may only learn about $\rho(\ell')$ for all $\ell' \leq \ell \in L(v)$. Soundness with respect to noninterference can then be stated and proved in the following manner:

► **Theorem 31 (Soundness).** *If Σ respects G and G is secure then Σ observes noninterference.*

Proof. Fix $v \in G$, $\ell \in L(v)$ and σs such that $\Sigma \Downarrow_\ell^v \sigma s$ then for some ρ we need to show

$$(\forall \ell' \leq \ell \in L(v). \pi_1(\Sigma(v))(\ell') = \rho(\ell')) \implies \exists \Sigma'. \pi_1(\Sigma'(v)) = \rho \wedge \Sigma' \Downarrow_\ell^v \sigma s \quad (17)$$

Take $\Gamma = (\pi_1 \circ \Sigma)[v \mapsto \rho]$. We have

$$\forall (v', \ell') \rightrightarrows (v, \ell). \pi_1(\Sigma(v'))(\ell') = \Gamma(v', \ell') \quad (18)$$

by case analysis on v' :

$v' = v$ We have $\pi_1(\Sigma(v'))(\ell') = \Gamma(v', \ell') = \rho(\ell')$ by the assumption stated in Equation 17 and the fact that $(v, \ell') \rightrightarrows (v, \ell) \implies \ell' \leq \ell$ because G is secure.

$v' \neq v$ We have $\Gamma(v', \ell') = (\pi_1 \circ \Sigma)[v \mapsto \rho](v', \ell') = \pi_1(\Sigma(v'))(\ell')$.

By the assumption that Σ respects G and the fact of Equation 18 there exists Σ' such that $\Sigma' \Downarrow_\ell^v \sigma s$ and $(\pi_1 \circ \Sigma') = \Gamma$ which means $\pi_1(\Sigma'(v)) = \Gamma(v) = \rho$. ◀

This result is straightforward, but it does verify that our notion of Σ respecting G is safe with respect to noninterference. We leave exploration of further alternative formalisations for future work.

6 Formalisation

The formalisation covers the developments both in this paper and in the 2019 paper by Bhardwaj and Prasad [2]. In particular all results presented in [2] have been verified independently by us; further, all results not marked with an “*” within this paper have also been formalised and verified. The formalisation provides a solid foundation for automating proofs of soundness of various enforcement mechanisms. These types of proofs are usually unwieldy and tedious yet lend themselves to automation.

The formalisation is written in the *Rocq* proof assistant, and the source is available online⁵. Our work is built on top of the *Mathematical Components* library (and *Hierarchy Builder*), mainly for its order theory module. The formalised theory does not depend on any additional axioms besides those necessitated by Rocq’s type system. The formalisation is split into six Rocq modules, each of which is given an overview below: `lagois`, `abssyn`, `sem`, `typrls`, `lagoisgraph`, and `soundness`.

The `lagois` module contains a formal definition of Lagois connections denoted `Lagois`, defined in terms of a *Hierarchy Builder* structure. Further, it contains formal proofs for results due to Melton et al. [6]. In particular: Proposition 3.3, Proposition 3.7, Proposition 3.8, Theorem 3.9, Proposition 3.11.(1), Proposition 3.11.(2). Generally we follow Bhardwaj and Prasad [2, 3] and limit the formalisation of these proofs to finite lattices with top and bottom elements, as opposed to partial orders. The formal proofs follow the structure of the original proofs by Melton et al. very closely. For some theorems, Melton et al. provide an explicit transpose result. In the formalisation we have left out such results and instead register

⁵ <https://github.com/CasperStaahl/LC4S-Rocq/releases/tag/vpaper>

the transpose of a Lagois connection as an instance of the `Lagois` structure, from which transpose results can be easily derived. The module also contains some miscellaneous lemmas and corollaries: `PC` is a proof of the fact that an instance of `Lagois` satisfies Equation **PC1** and Equation **PC2**. `PC`, `melton_3_7` and `melton_3_7_full` are essentially variations of each other. `PC` follows the original proof by Melton et al., and `melton_3_7`, `melton_3_7_full` are derived from it in an ad hoc fashion. `CC` is a proof that an instance of `Lagois` satisfies Equation **CC1** and Equation **CC2**.

The modules `abssyn`, `sem`, and `typrls` contain the formalisation of the abstract syntax, operational model, and type system (and soundness thereof) from [2, 3], respectively. The formalisation of the abstract syntax and operational model contain some minor and inconsequential differences to its informal counterpart. In particular we limit import-, export- and local variables to be members of the natural numbers, instead of them being members of the mutually disjoint sets Z, Z', X, X', Y and Y' . Further, the syntax and operational model have been rewritten to factor out duplicate grammar- and induction rules.

In `sem` the inductive definitions `commsem` and `phrasesem` correspond to the inductive rules defining judgements $\langle \mu, \mu' \rangle \vdash p \Rightarrow \langle \nu, \nu' \rangle$ in the operational model given in [2, 3]. More specifically rule `Tsem`, `WRsem` and `RDsem` in `commsem` along with rule `Lcommsem` in `phrasesem` correspond to rules `T` (and `T'`), `WR` (and `WR'`) and `RD` (and `RD'`), respectively. Here, `Lcommsem` factors out duplicate rules. Rules `TRLsem` and `TLRsem` in `phrasesem` correspond to rules `TRL` and `TLR`, respectively. The inductive definition `seqsem` corresponds to the induction rules for the judgement $\langle \mu, \mu' \rangle \vdash s \Rightarrow^* \langle \nu, \nu' \rangle$. Specifically `seqsem` and `seqconsssem` correspond to `SEQ0` and `SEQS` respectively. The `SubSemantic Hierarchy Builder` structure defined in `sem` formalises the constraints placed on the atomic transactional operations ranged over by t and t' in [2, 3]. We overload the notation $\_ == [\_] ==> \_$ such that $\mu \vdash t \Rightarrow \nu$ has a formal counterpart $\mu == [t] ==> \nu$, $\langle \mu, \mu' \rangle \vdash p \Rightarrow \langle \nu, \nu' \rangle$ has a formal counterpart $(\mu, \mu') == [p] ==> (\nu, \nu')$ as defined by `phrasesem`, and $\langle \mu, \mu' \rangle \vdash s \Rightarrow^* \langle \nu, \nu' \rangle$ has a formal counterpart $(\mu, \mu') == [s] ==> (\nu, \nu')$ as defined by `seqsem`.

The inductive definitions `CommSType`, `PhraseSType`, `SeqSType` in the module `typrls` define the formal counterpart to the type system presented in [2, 3] with duplicate rules factored out. Specifically, rules `Tt`, `Trd` and `Twr` in `CommSType` correspond to rule `TT` (and `TT'`), `TRD` (and `TRD'`) and `TWR` (and `TWR'`), respectively, with rules `LCT` and `RCT` in `PhraseSType` factoring out duplicate rules. Further, formal rules `TTrl` and `TTlr` in `PhraseSType` correspond to rules `TTRL` and `TTLR'`, respectively. Finally, formal rules `Com0` and `Comp` in `SeqSType` correspond to rules `COM0` and `COMP` respectively. Our formal proof of soundness `SeqSType_sound` for the type system follows the same structure as that of [2, 3], although in some cases a case analysis is replaced by a direct argument incidentally fixing a few missing cases in the analysis. Essentially, the formal proof proceeds by induction on the derivations of `seqsem`, from which a case analysis is conducted on the resulting derivations of `phrasesem` and `commsem`. The lemma `CommsSType_sound` corresponds to the case analysis on members in `commsem`, and the lemma `T_sound` corresponds to the remaining cases where `TTrl` (or `TTlr`) is the bottom-most rule used in the derivation of the transition.

The module `lagoisgraph` pertains to the formalisation of the content presented in sections 3 and 4. Besides the results presented in sections 3 and 4 the module contains several helper lemmas not covered here for the sake of brevity. The `LagoisGraph` module gives the formal definition of a Lagois graph as a `Hierarchy Builder` structure. Notably, the formalisation does not contain an underlying notion of a graph:

```

HB.mixin Record IsLagoisGraph V of Equality V := {
  lattice : V -> {d & porderType d} ;
  edge v v' : bool ;
  label v v' : edge v v' ->
    Lagois.type (projT2 (lattice v)) (projT2 (lattice v')) ;
  edge_irefl v : edge v v = false ;
  edge_sym v v' : edge v v' -> edge v' v ;
  label_sym v v' p p' : (label v v' p).1 =1 (label v' v p').2;
}.
HB.structure Definition LagoisGraph :=
  {T of IsLagoisGraph T & Equality T}.

```

Essentially, a Lagois graph is a type V equipped with a function `lattice` that maps points in V to their respective partial orders. In this regard the formalisation is more general, allowing the codomain of `lattice` to be the type of partial orders instead of the type of lattices. `edge` is the edge relation that is interpreted in the expected manner. `label` is a function labelling edges, i.e. proofs of `edge v v' = true`, with a Lagois connection between the lattices associated to v and v' . `edge_irefl` enforces that there are no immediate self-loops. `edge_sym` enforces that the edge relation is symmetric, and `label_sym` enforces that symmetric edges agree in their associated Lagois connections. Further, we do not require that the underlying type that represents the vertices in the graph is finite, only that it has decidable equality. We use $L(v)$ to denote the underlying poset of a vertex v for some Lagois graph, and denote the presence of an edge between vertices v and v' as $v @> v'$. Further, we denote the type of paths, as defined by `path`, between v and v' in G as $v \sim> v' :> G$ (or $v \sim> v'$) and path concatenation as $\backslash*$. The formal inductive definition `flow` corresponds to the informal definition of the flow relation (Definition 14). The formal definitions `flow_secure` and `flow_secure_graph` correspond to the informal definition of flow secure (Definition 15). Similarly, the formal definitions `loop_secure`, `loop_secure_vertex` and `loop_secure_graph` correspond to the informal definition of loop secure (Definition 16). Lemma 17 and Proposition 18 have formal counterparts `flow2path` and `flow_secure_graph_is_loop_secure`. `simply_secure` is the formalisation of Definition 19, with related Lemma 20 and Proposition 21 formalised as `simply_secure_iff_loop_secure` and `path_nondecreasing`, respectively. The proof `simply_secure_iff_loop_secure` is based on an induction principle over loops in a graph instead of strong induction, the induction principle is defined in two parts as:

```

Fixpoint loop_ind_aux (P : forall v : G, v ~> v -> Prop)
  (bc_1 : forall v : G, P v (ε))
  (bc_2 : forall (v v' : G) (g : v @> v') (h : v' ~> v),
    uniq h -> P v (g \* h))
  (ic : forall (v v' : G) f1 f2 h,
    P v' h -> P v (f1 \* f2) -> P v (f1 \* h \* f2))
  v (f : v ~> v) (ACC : Acc lelooplen (existT _ v f))
  {struct ACC} : P v f :=
  (...)
Definition loop_ind (P : forall v : G, v ~> v :> G -> Prop)
  (bc_1 : forall v : G, P v (ε))
  (bc_2 : forall (v v' : G) (g : v @> v') (h : v' ~> v),
    uniq h -> P v (g \* h))

```

```

(ic : forall (v v' : G) f1 f2 h,
  P v' h -> P v (f1 \* f2) -> P v (f1 \* h \* f2))
v (f : v ~> v) : P v f :=
loop_ind_aux bc_1 bc_2 ic (subloopwf (existT _ v f)).

```

The induction principle is derived as described by [5] and the general pattern used in the proof of Proposition 21. That is, the induction principle is first written as a general recursive function, which is generally not allowed in Rocq. Then we add the assumption that f is accessible through the strict ordering of loops based on their length i.e., **ACC**. This allows us to do structural recursion on **ACC**, which is allowed in Rocq, giving us the function `loop_ind_aux`. We can get rid of the assumption **ACC** by proving that the ordering of loops based on their length is well-founded, i.e., all loops are accessible through the ordering just described. Indeed, given a type A , a relation $R(x, y) := f(x) < f(y)$ with $f : A \rightarrow \mathbb{N}$ is always well-founded. In this particular case, A is the type of loops in a graph and $f(x)$ is the length of loop x . With **ACC** eliminated, we can finally derive the complete induction principle as seen in the function `loop_ind`.

Moving on to the part of the module `lagoisgraph` in which section 4 has been formalised: For Lemma 22 we have the formal counterpart `reverse_path_loop_secure`, the reader should note that $f^{\sim}$ denotes the reverse path of f . The Hierarchy Builder structures `LagoisVForest` and `LagoisForest` correspond respectively to the definitions of virtual forests (Definition 24) and forests (Definition 23), with associated Theorem 25 and Corollary 26 formalised as `VForest_loop_secure` and `Forest_loop_secure`. The secure composition operation (Definition 27) is formally defined in terms of the carrier type:

```

Inductive Bridge (G G' : LagoisGraph.type)
  (v_abut : G) (v'_abut : G')
  (fg : Lagois.type L(v_abut) L(v'_abut)) : Type :=
| lbank : G -> Bridge fg
| rbank : G' -> Bridge fg.

```

Thus for $G \xrightarrow{f}^{\bar{v}'} G'$: G, G' correspond to G, G' ; v_abut, v'_abut correspond to $\bar{v}, \bar{v}'$; and fg correspond to $(L(\bar{v}), f, g, L(\bar{v}'))$. The carrier type, `Bridge G G' v_abut v'_abut fg` (or `Bridge fg` when the preceding arguments can be inferred), is equivalent to $G + G'$. The carrier type is then made a canonical instance of the Lagois graph structure:

```

Context (G G' : LagoisGraph.type) (v_abut : G) (v'_abut : G')
  (fg : Lagois.type L(v_abut) L(v'_abut)).
(...)
HB.instance Definition _ :=
  IsLagoisGraph.Build (Bridge fg)
  Bridge_edge_irefl Bridge_edge_sym Bridge_label_sym.

```

such that `Bridge_edge_irefl`, `Bridge_edge_sym`, `Bridge_label_sym`, and the inferred arguments are defined in the expected manner. Finally, module `lagoisgraph` finishes with theorem `Bridge_secure` asserting formally that secure composition is indeed secure (Theorem 7).

Finally, the module `soundness` covers the formal counterpart to section 5. Variables G, S, N and $\mathcal{S}$ have formal counterparts with the same symbols, however the formal definitions of N and $\mathcal{S}$ are slightly more general. In particular, $N : \text{forall } (v : G) (\ell : L(v)), \text{Type}$ and $\mathcal{S} := \text{forall } (v : G), (\text{forall } \ell : L(v), N \ell) * \mathcal{S}$, that is, N may depend on ℓ . $\rightarrow_{\subseteq} \mathcal{S} \times \mathcal{S}$ has the formal equivalent `R` and `Trace` is the type of traces. `Interface` gives the

type of distributed security interfaces informally ranged over by Γ . Similarly, `LocalInterface` gives the type of local security interfaces informally ranged over by ρ . $\Sigma|_\ell^v$ and $\Sigma s|_\ell^v = \sigma s$ have respective formal counterparts `obs v l Σ : (forall ℓ' : L(v)), option (N ℓ')` and `Obs v l Σ s σ s : Prop`, and `emits v l Σ σ s` formalises the emission predicate $\Sigma \Downarrow_\ell^v \sigma s$. `distr_interface_equiv`, `distr_interface_class`, `local_interface_equiv` and `local_interface_class` are the formal counterparts to $\approx$, $\llbracket \_ \rrbracket$, $\sim$, and $\llbracket \_ \rrbracket$, respectively. The knowledge of an attacker $\mathbb{K}$ and K are denoted with the same symbols in the formalisation. `respects` and `noninterference` respectively formalise Definition 29 and Definition 30. Finally, Theorem 31 has the formal counterpart `respects2noninterference`.

7 Related Work

Lagois connections are originally due to the seminal work of Melton et al. [6] in which results are presented that are analogous to similar results derived from Galois connections.

The idea to use Lagois connections as a framework for secure information flow is due to Bhardwaj and Prasad as originally presented in [2]. Along with the framework several results are presented and adapted from [6] that are relevant for secure information flow. The framework is illustrated for a restricted language and a type system is developed that is sound with respect to noninterference for secure information flow between two organizations.

In [3], Bhardwaj and Prasad extend their prior work. Firstly, to establish and update secure connections based on the Lagois connection framework, again relying on results from [6]. In particular, they cover the extension from two actors to an arbitrary number, as discussed in section 1 and 2. Secondly, they prove that Lagois connections can be used to establish a secure connection with two organizations using the decentralized label model due to Myers [7].

In [8], Nielson et al. present an alternative to the type system of Bhardwaj and Prasad. Most importantly, the type system can deal with an arbitrary number of actors/organizations. Further, actors are not restricted in the way they are connected, i.e., they do not have to be connected in a chain and can move between clusters of interconnected actors provided that they drop all information that is not already public before moving. The programs under consideration are also not limited in the fashion that those considered by Bhardwaj and Prasad are. However, the security of the system is described in terms of a condition similar to our Definition 15; it is thus unclear how strong the noninterference guarantees are.⁶

In [1], Askarov and Chong present an alternative perspective on noninterference in terms of the change in knowledge of an attacker. Their definition is parametric in the attacker, and programs can be secure against some attacker but not others. In particular they identify a collection of relevant attackers, most notably the perfect attacker that remembers everything it sees and the “ i ’th event only” attacker that only observes the i ’th event. They show that a program that is secure against an “ i ’th event only” attacker for all i is also secure against the perfect attacker. They use this fact to develop enforcement mechanisms, both static and dynamic, that are secure against the perfect attacker. The framing in [1] is quite different from ours: It is mainly concerned with only a single program and security of input channels/streams to this program. Further, the flows that are legal can dynamically change and the attacker is only able to observe a single channel as opposed to several variables.

⁶ [9], also by Nielson et al., builds on the concepts of [8]. However, the ability for actors to move between clusters is however not covered.

8 Conclusion and Future Work

As mentioned in section 1 and 2, the main motivation for studying arbitrary graphs of Lagois connections instead of chains was to bypass intermediate lattices incurring label loss. We have mitigated this problem to some degree, but it is inherently the case that the graph must be forest shaped if we want ensure security without peeking into the underlying paths of the Lagois Graph and in the worst case computing and then checking the flow relation. Thus if we want to bypass a lossy intermediate, we cannot determine a Lagois graph to be secure by looking at the graph structure alone, although Theorem 28 tells us that we only have to check the parts of the graph that do not have the structure of a forest.

Our soundness result w.r.t. noninterference is independent of a specific model of computation and enforcement mechanism. A natural next step is thus to fix such a model and define an enforcement mechanism that ensures a given system state Σ respects the graph G over which it is defined. One such example could possibly be an adaptation of the model and enforcement mechanism presented in Nielson et al. [8, 9].

For the soundness result we take a state to be in $\mathcal{S} = (v \in G) \rightarrow ((L(v) \rightarrow N) \times S)$. However it may be convenient for N to depend on the value of $L(v)$, i.e., $\mathcal{S} = (v \in G) \rightarrow ((\ell \in L(v)) \rightarrow N(\ell)) \times S$. For example in the example presented in Appendix A it would be nice to have

$$N(\ell) = \begin{cases} G \rightarrow \text{Stmt} \cup \mathbf{1} & \text{if } \ell = \perp \\ \mathbb{N}^* & \text{otherwise} \end{cases}, \text{ instead of } N = \mathbb{N}^* \cup (G \rightarrow \text{Stmt}) \cup \mathbf{1}$$

as it would allow an attacker to learn that high buffers are actually only buffers, which is intuitively safe as long as they do not learn about the actual contents. Indeed, the soundness result still holds for such $\mathcal{S}$. This is evident by the formalisation in which soundness is defined by- and proved for this more general $\mathcal{S}$ in a similar manner to the proof of soundness already presented herein.

Two concepts that our framework does not account for is dynamism of agents (à la Nielson et al. [8, 9]) and dynamism of policies (à la Askarov and Chong [1]). By *dynamism of agents* we mean the ability of agents to move about in the network and in the process dropping old connections and establishing new ones. For example [9] has dynamism of agents: Agents can move between clusters of agents via a primitive `relocate( $l$ )` command, that relocates agent v to cluster l when executed at agent v . By *dynamism of policies* we mean the ability of policies to change during runtime. Conversely [1] has dynamism of policies: A program can update its information flow policy via a `setPolicy( $\sqsubseteq$ )` primitive for some reflexive relation $\sqsubseteq$. We conjecture that our framework may be expanded to cover both types of dynamism using dependent functions:

$$G \rightarrow_{\mathcal{G}} G' = (v \in G) \rightarrow ((v' \in G) \times (L(v) \rightarrow L(v'))).$$
(19)

For $F \in G \rightarrow_{\mathcal{G}} G'$ and $v \in G$ we would then interpret $F(v)$ as v moving to or becoming $\pi_1(F(v))$ and v 's lattice transforming as specified by $\pi_2(F(v))$. More specifically $\pi_2(F(v))(p) = p'$ would denote label p becoming label p' . We leave this for future work.

Finally, we would like to thank the reviewers for their detailed, constructive, and very helpful feedback.

References

- 1 Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 308–322. IEEE, 2012. doi:10.1109/CSF.2012.31.
- 2 Chandrika Bhardwaj and Sanjiva Prasad. Only connect, securely. In *Formal Techniques for Distributed Objects, Components, and Systems: 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 39*, pages 75–92. Springer, 2019. doi:10.1007/978-3-030-21759-4_5.
- 3 Chandrika Bhardwaj and Sanjiva Prasad. Secure information flow connections. *Journal of Logical and Algebraic Methods in Programming*, 127:100761, 2022. doi:10.1016/J.JLAMP.2022.100761.
- 4 Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. doi:10.1145/360051.360056.
- 5 Xavier Leroy. Well-founded recursion done right. In *CoqPL 2024: The Tenth International Workshop on Coq for Programming Languages*, 2024.
- 6 Austin Melton, Bernd SW Schröder, and George E Strecker. Lagois connections—a counterpart to Galois connections. *Theoretical Computer Science*, 136(1):79–107, 1994.
- 7 Andrew C Myers and Barbara Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, 31(5):129–142, 1997. doi:10.1145/268998.266669.
- 8 Flemming Nielson, René Rydhof Hansen, and Hanne Riis Nielson. Adaptive security policies. In *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part II 9*, pages 280–294. Springer, 2020. doi:10.1007/978-3-030-61470-6_17.
- 9 Flemming Nielson, René Rydhof Hansen, and Hanne Riis Nielson. Benign interaction of security domains. In *Protocols, Strands, and Logic: Essays Dedicated to Joshua Guttman on the Occasion of his 66.66 th Birthday*, pages 312–331. Springer, 2021. doi:10.1007/978-3-030-91631-2_17.
- 10 Casper Ståhl. Extending Lagois connections for secure information flow to n organizations. Master’s thesis, Dept. of Computer Science, Aalborg University, 2025.
- 11 Casper Ståhl, Levs Gondelman, René Rydhof Hansen, and Danny Bøgsted Poulsen. LC4S-Rocq. Software, swbId: swb:1:dir:828cd84332eae467073735e4aee31198815adf58 (visited on 2026-07-23). URL: <https://github.com/CasperStaahl/LC4S-Rocq>, doi:10.4230/artifacts.27330.

A

Example instance of computational model

In this appendix we provide an example instance of $(\mathcal{S}, \longrightarrow)$ in the form of a distributed while language with communication taking place between some buffers. Fix a set of variables Var and let x be a metavariable ranging over this set. We then define a simple while language with statements Stmt and expressions defined Exp inductively as:

$$\begin{aligned}
 e \in \text{Exp} &::= n \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2 \mid e_1 = e_2 \mid \neg e \mid e_1 \bmod e_2 \mid e_1 / e_2 \\
 s \in \text{Stmt} &::= \text{skip} \mid \text{get}_\ell(x) \mid \text{send}_\ell^v(x) \mid x := e \mid \text{if } e \text{ then } s \text{ else } s \mid \text{while } e \text{ do } s \mid s_1; s_2
 \end{aligned}$$

Fix some Lagois graph G , and set $N = \mathbb{N}^* \cup (G \rightarrow \text{Stmt}) \cup \mathbf{1}$, $S = \text{Stmt} \times (\text{Var} \rightarrow \mathbb{N})$ such that

$$\mathcal{S} = (v \in G) \rightarrow ((L(v) \rightarrow \mathbb{N}^* \cup (G \rightarrow \text{Stmt}) \cup \mathbf{1}) \times \text{Stmt} \times (\text{Var} \rightarrow \mathbb{N})).$$

Then for each $\Sigma \in \mathcal{S}$ we interpret $\Sigma(v) = (\rho, s, m)$ for some v as ρ is the current state of the buffers at v at a given level, except at level $\perp$ (the lowest level in $L(v)$) which contains the empty buffer or the initial distributed statement that was running in the system. Let s be the current statement executed at v , $\text{get}_\ell(x)$ denotes that the head of a buffer is retrieved and stored in variable x , and $\text{send}_\ell^v(x)$ indicates that the value of variable x is sent to the buffer associated to ℓ at v . The memory configuration for local variables at v is denoted m and we let $m(e)$ denote the evaluation of expression e with respect to memory $m \in \text{Var} \rightarrow \mathbb{N}$. Let Sig be the set of signals ranged over by metavariable α that is inductively defined as:

$$\alpha ::= n?\ell \mid n!(v, \ell)$$

such that $n?\ell$ is interpreted as n was received in the buffer associated to ℓ , and $n!(v, \ell)$ as n was sent to buffer associated to ℓ at v . We then define $\longrightarrow$ in terms of another relation with judgements $v \vdash (\rho, s, m) \longrightarrow'_\alpha (\rho', s', m')$ such that $v \in G$ and $(\rho, s, m), (\rho', s', m') \in (L(v) \rightarrow \mathbb{N}^* \cup (G \rightarrow \text{Stmt}) \cup \mathbf{1}) \times \text{Stmt} \times (\text{Var} \rightarrow \mathbb{N})$ that is defined inductively as seen below. $v \vdash (\rho, s, m) \longrightarrow'_\alpha (\rho', s', m')$ should intuitively be read as: in one step local state (ρ, s, m) at v can go to state (ρ', s', m') emitting signal α .

$$\begin{array}{c} \frac{m' = m[x \mapsto m(e)]}{v \vdash (\rho, x := e, m) \longrightarrow'_\epsilon (\rho, \text{skip}, m')}, \quad \frac{v \neq v' \quad \ell' \in L(v') \quad m(x) = n}{v \vdash (\rho, \text{send}_{\ell'}^{v'}(x), m) \longrightarrow'_{n!(v', \ell')} (\rho, \text{skip}, m')}, \\ \\ \frac{\ell \in L(v) \quad \rho(\ell) = n \star ns}{v \vdash (\rho, \text{get}_\ell(x), m) \longrightarrow'_\epsilon (\rho[\ell \mapsto ns], \text{skip}, m[x \mapsto n])}, \quad \frac{\ell \in L(v) \quad \rho(\ell) \neq n \star ns}{v \vdash (\rho, \text{get}_\ell(x), m) \longrightarrow'_\epsilon (\rho, \text{skip}, m')}, \\ \\ \frac{\ell \in L(v) \quad \rho(\ell) \in \mathbb{N}^*}{v \vdash (\rho, s, m) \longrightarrow'_{n?\ell} (\rho[\ell \mapsto ns \star n], s, m)}, \quad \frac{\ell \in L(v) \quad \rho(\ell) \notin \mathbb{N}^*}{v \vdash (\rho, s, m) \longrightarrow'_{n?\ell} (\rho[\ell \mapsto n], s, m)}, \\ \\ \frac{m(e) = 0}{v \vdash (\rho, \text{if } e \text{ then } s_1 \text{ else } s_2, m) \longrightarrow'_\epsilon (\rho, s_2, m)}, \quad \frac{v \vdash (\rho, s_1, m) \longrightarrow'_\alpha (\rho', s'_1, m')}{v \vdash (\rho, s_1; s_2, m) \longrightarrow'_\alpha (\rho', s'_1; s_2, m')}, \\ \\ \frac{m(e) > 0}{v \vdash (\rho, \text{if } e \text{ then } s_1 \text{ else } s_2, m) \longrightarrow'_\epsilon (\rho, s_1, m)}, \quad \frac{}{v \vdash (\rho, \text{skip}; s, m) \longrightarrow'_\epsilon (\rho, s, m)}, \\ \\ \frac{}{v \vdash (\rho, \text{while } e \text{ do } s, m) \longrightarrow'_\epsilon (\rho, \text{if } e \text{ then } (s; \text{while } e \text{ do } s) \text{ else skip}, m)} \end{array}$$

Let $\text{INIT}(\Sigma)$, and $\text{IM}(\Sigma)$ denote that the distributed security interface of Σ is initial and intermediate, respectively, which we formally define as follows:

$$\begin{aligned} \text{INIT}(\Sigma) &= \forall v \in G. \pi_1(\Sigma(v))(\perp) \in \mathbf{1} \\ \text{IM}(\Sigma) &= \forall v \in G. \pi_1(\Sigma(v))(\perp) \in (G \rightarrow \text{Stmt}) \end{aligned}$$

Then $\longrightarrow$ can be defined, with judgements $\Sigma \longrightarrow \Sigma'$, as:

$$\begin{array}{c} \frac{\text{INIT}(\Sigma)}{\Sigma \longrightarrow \Sigma[v \mapsto (\pi_1(\Sigma(v))[\perp \mapsto \pi_2 \circ \Sigma], \pi_2(\Sigma(v)), \pi_3(\Sigma(v))) \mid v \in G]}, \\ \\ \frac{v' \vdash \Sigma(v') \longrightarrow'_\epsilon \sigma \quad \text{IM}(\Sigma)}{\Sigma \longrightarrow \Sigma[v' \mapsto \sigma]}, \quad \frac{v_1 \vdash \Sigma(v_1) \longrightarrow_{n!(v_2, \ell)} \sigma_1 \quad v_2 \vdash \Sigma(v_2) \longrightarrow_{n?\ell} \sigma_2 \quad \text{IM}(\Sigma)}{\Sigma \longrightarrow \Sigma[v_1 \mapsto \sigma_1, v_2 \mapsto \sigma_2]}. \end{array}$$

To further concretise the example fix a lattice $\mathcal{L} = \perp \leq l \leq h$ and a Lagois graph $G = (\{v_1, v_2\}, \{(v_1, v_2), (v_2, v_1)\}, L, F)$ such that $L(v_1) = L(v_2) = \mathcal{L}$ and $F((v_1, v_2)) = F((v_2, v_1)) = \text{id}$. Then $\Sigma(v_1) = (\rho_1, s_1, m_2)$ and $\Sigma(v_2) = (\rho_2, s_1, m_2)$ where $\rho_1(\perp) = \rho_2(\perp) = *$, and s_1, s_2 are given by Figure 7, and Σ therefore observes noninterference whenever $42 < \pi_3(\Sigma(v_1))(x)$ because in that case Σ respects G which is secure.

```
while 1 do (
  sendhv2(x);
  x := x + 1
)
```

(a) Statement s_1 .

```
while 1 do (
  geth(x);
  if x = 42 then
    sendlv1(x)
  else
    sendhv1(x)
)
```

(b) Statement s_2 .

■ **Figure 7** Statements s_1 and s_2 .

Towards Fuzzy Constructive Type Theories

Besik Dundua 

Giorgi Nikoladze Scientific-Research Center, Kutaisi International University, Georgia
Vekua Institute of Applied Mathematics, Tbilisi State University, Georgia

Furio Honsell 

Dipartimento di Scienze Matematiche, Informatiche e Fisiche, Udine University, Italy

Temur Kutsia 

Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria

Marina Lenisa 

Dipartimento di Scienze Matematiche, Informatiche e Fisiche, Udine University, Italy

Luigi Liquori 

Centre Inria de l'Université Côte d'Azur, Sophia Antipolis, France

Abstract

In this paper, we explore combinations of Fuzzy Logic and Constructive Higher Order Type Theory. Although Fuzzy Logic is more than 60 years old, and Fuzzy Type Theories have been studied in the classical case of Church's Theory of Types, there is yet no satisfactory understanding of how to deal with judgements of the shape $\Gamma \vdash M :_d A$, where d is a fuzzy degree of confidence. Addressing this issue is important in view of the growing interest in quantitative and non-idempotent type theories. To this end, we introduce *fuzzy type assignment systems* in order to investigate how minimal fuzzy formulæ behave as types, according to some *fuzzy proposition-as-types* paradigm. Moreover, we study a *fuzzy version* of *intersection types*. In both systems assumptions are multisets and d , in $\Gamma \vdash M :_d A$, is a formula of minimal propositional Fuzzy Logic. Evaluating d in a residuated lattice $[0,1]$, endowed with a left-continuous T-norm, we can analyse how the *degree of confidence* propagates from the assumptions to the conclusion, thereby yielding new information on the derivation. The former system sheds light on the connection between minimal Fuzzy Logics and *affine λ -calculus* and permits to show that the tautologies in the **BCK**-logic amount to the simple types which are inhabited by **BCK**-combinators. The latter system keeps track of the number of times a given fuzzy assumption is made. We discuss the standard suite of metatheorems (inversion lemmata and subject-conversion) for the systems, and point to possible uses of the fuzzy formula attached to the membership construct.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Constructive mathematics

Keywords and phrases Fuzzy type theory, constructive type theory, lambda calculus, residuated lattices

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.7

Funding Supported by the Austrian Science Fund (FWF) under project P 35530 and Shota Rustaveli National Science Foundation under project FR-25-5957.

Acknowledgements The authors would like to express their gratitude to the reviewers for their precious comments on the original version of the present paper.

1 Introduction

Fuzzy Logics [16, 12, 14] are more than 60 years old. They are a family of many-valued logics which arise when multiple and linearly ordered *degrees of truth* are considered, namely when the truth value of a propositional formula may be taken to be any real number between 0 and 1. Fuzzy Logics have been extensively used in control systems, in artificial intelligence, and recently in digital health and life-enhancing technologies. Significant work has already



© Besik Dundua, Furio Honsell, Temur Kutsia, Marina Lenisa, and Luigi Liquori;
licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 7; pp. 7:1–7:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

appeared on Fuzzy Type Theory in the classical setting of Church’s Theory of Types [13], and as far as first order theories [6]. However, to the best of our knowledge, there is yet no complete and satisfactory understanding of how the basic idea behind Fuzzy Logic combines with Higher Order *Constructive* Type Theory. We think that this issue needs to be addressed, since decorating a typing judgement with a fuzzy coefficient adds a new expressive dimension to the invariants which types can capture, thus leading to *fuzzy constructive type theory*. This should be particularly relevant in view of the growing interest, in the theoretical computer science community, towards systems supporting a finer analysis of programs based on quantitative and non-contractive reasoning, *e.g.* [10, 5, 4, 11]. One of the characteristics of Fuzzy Logic, which lies at the root of its explanation of the Sorites Paradox [15], is that each application of *modus ponens* further decreases the confidence value of a formula.

In this paper, we introduce and investigate various notions of *fuzzy type assignment systems* for pure λ -calculus, thus providing possible answers to the following questions.

- How do intuitionistic fragments of Fuzzy Logic behave as types in a possible *Fuzzy Proposition-as-Types* and *Proofs as λ -terms* paradigm?
- What is a fuzzy version of *Intersection Type Assignment Systems*?

Answering these two questions should pave the way for addressing more general questions such as: What are “fuzzy twins” of other type assignment systems *à la* Curry? What is the appropriate notion of *Fuzzy Constructive Type Theory* for λ -terms *à la* Church? Moreover, since Fuzzy Logic has in general a non-idempotent semantics, how do fuzzy type assignment systems relate to existing quantitative and non-idempotent non-contractive type theories and program logics?

In particular, in the present paper, we introduce three fuzzy type assignment systems. The first two formal systems $\mathcal{F}$ and $\mathcal{Q}$ deal with judgements of the shape $\Gamma \vdash M : d$, where the type d is a formula of negation-free Fuzzy Logic. They provide answers to the first question. These systems differ in that $\mathcal{F}$ is contractive, namely contexts are taken as sets of assumptions, while $\mathcal{Q}$ is non-contractive, *i.e.* contexts are multisets of assumptions. Evaluating the semantics of the formula d in the lattice $[0,1]$, enriched with a left-continuous T-norm (see Definition 32), permits us to relate the notion of *inhabited* fuzzy formula, in $\mathcal{F}$ and $\mathcal{Q}$, to its degree of truth. The value of d depends on the value of the degree of confidence assigned to the assumptions. If d is a tautology, the value of the formula is clearly 1, but if the confidence degree in the assumption is not 1, its value in general is smaller, thereby providing information on the number of occurrences a variable has been used. This permits us to recover, along a different route (see Corollary 11), the connection between the closed *affine* λ -terms, namely the **BCK**-combinators, and the theory of **BCK**-logic, which has been extensively studied in the context of Fuzzy Logic, see *e.g.* [14]. We recall that **BCK**-logic, namely the sound and complete logic of **BCK**-algebras, is included in all Fuzzy-Logics. Moreover, we show that taking contexts as multisets preserves the interpretation of fuzzy types under β -conversion.

Building on $\mathcal{Q}$, we introduce the *Fuzzy Intersection Type Assignment System* $\mathcal{Q}^\cap$, which is a “fuzzification” of the Intersection Type Assignment System [2, 3], for establishing judgements of the shape $\Gamma \vdash_d M : A$, where d is a formula of Fuzzy Logic and A is an intersection-type. Intuitively we may say that the term M has *fuzzy degree d of membership* to the type A . Incidentally, we remark that the former systems $\mathcal{F}$ and $\mathcal{Q}$ amount to studying the fuzzy membership to a unique reflexive type A , such that $A \sim A \rightarrow A$, in a suitable Intersection Type Theory, see Remark 25. The system $\mathcal{Q}^\cap$ is non-contractive, hence *relevant*, where fuzzy coefficients index the membership relation and contexts are *fuzzy multisets*, rather than sets. All the standard suite of results which hold in the *crisp* version, *i.e.* the

original Intersection Types system, also hold for $\mathcal{Q}^\cap$. The term “crisp”, which belongs to the terminology in Fuzzy Logic, denotes the system where the truth value of propositions is just 0 or 1. Namely, we have: inversion lemmata, subject expansion, and subject reduction. The benefit of this system, besides providing a theory of how the degree of confidence propagates via applications and abstractions, permits us to keep track of the number of occurrences of a variable in a term. We hint to some applications. We chose to focus on Intersection Types because they generalise simple types and, moreover, they can capture the class of strongly normalising terms, as well as many other classes of λ -terms. We are confident that the same “fuzzification” can be similarly carried out for other types assignment systems for λ -terms *à la* Curry, such as higher order impredicative types.

For all notations and notions which are not explicitly spelled out in this paper, we refer to [3], for typed λ -calculi, and to *e.g.* [12, 14], for Fuzzy Logic.

Synopsis. In Section 2 we give basic definitions on Fuzzy Logic and Type Assignment Systems. In Section 3 we introduce the type assignment systems $\mathcal{F}$ and $\mathcal{Q}$ for assigning a formula of Fuzzy Logic, and ultimately a *confidence degree* to a λ -term, by evaluating the fuzzy formula, and study their main properties. Full Intersection Type Theory is considered in Section 4. Final remarks on Constructive Type Theories with a *fuzzy typing relation*, comparison with related work and directions for future work appear in Section 5. The Appendix A contains additional standard results and definitions in Fuzzy Logic.

2 Preliminaries

2.1 Fuzzy Logic

The language of *minimal Fuzzy Logic* is given by the following grammar:

$$d, e, f ::= X \mid d \wedge e \mid d \otimes e \mid d \Rightarrow e \mid 0 \mid 1.$$

The semantics of d , which is denoted by $\llbracket d \rrbracket_\rho^T$, is given by interpreting propositional variables X , according to an assignment ρ , as values in the interval $[0,1]$ endowed with a *left-continuous T-norm*, thus making the interval $[0,1]$ a residuated lattice, see Appendix A. The constants 0 and 1 denote falsity and truth. By abuse of notation, when clear from the context, we will use values in $[0, 1]$ as constants in the language. The T-norm allows us to interpret the connectives, *viz.* $\otimes$ by the T-norm itself, $\Rightarrow$ by its residuum, and $\wedge$ by the minimum. We assume that $\otimes$ has higher priority than $\Rightarrow$, the latter being right-associative, *e.g.* $d \Rightarrow e \Rightarrow f \otimes f'$ means $d \Rightarrow (e \Rightarrow (f \otimes f'))$. The three prominent examples of left-continuous T-norms on $[0,1]$, namely Gödel’s T-norm, Goguen’s product T-norm, and Łukasiewicz’s T-norm, appear in Appendix A, together with some of the main properties of T-norms. We refer to [12, 14] for more details and basic definitions. A T-norm is said to be *idempotent* if $d \otimes d = d$. Gödel’s T-norm is the only idempotent left-continuous T-norm. By abuse of notation, when clear from the context, we denote the norm and its residuum, by $\otimes$ and $\Rightarrow$.

2.2 Lambda terms and Intersection Type Assignment Systems

In this subsection, we introduce the untyped λ -calculus and the set of types that can be assigned to it. We focus on Intersection Type systems [2, 3, 8].

7:4 Towards Fuzzy Constructive Type Theories

► **Definition 1** (λ -terms).

1. The set Λ of (untyped) λ -terms is defined as usual:

$$M, N ::= x \mid \lambda x.M \mid MN$$

where x is a variable, $\lambda x.M$ is an abstraction, and MN is a term application.

2. Affine λ -terms are λ -terms, where each abstracted variable occurs at most once. Linear λ -terms are λ -terms, where each abstracted variable occurs exactly once. Closed terms are called combinators.

We refer to [1, 3] for standard notions and results concerning λ -calculus.

We briefly recall the basic definitions concerning simple and intersection types.

► **Definition 2.**

1. Given a set of constants $\mathbb{A}$ and a distinguished constant $\mathbb{U}$, the set $\mathbb{T}_{\mathbb{A}}$ of intersection types over $\mathbb{A}$ are defined by the grammar:

$$A, B ::= c \mid \mathbb{U} \mid A \rightarrow B \mid A \cap B$$

where $c \in \mathbb{A}$; simple types are those types which do not contain the constant $\mathbb{U}$ and the $\cap$ constructor.

2. A subtyping relation $\leq$ is a binary relation on $\mathbb{T}_{\mathbb{A}}$ closed under the following axioms and rules:

$$A \leq A \quad (\text{Ref}) \qquad A \cap B \leq A \quad (\text{incL}) \qquad B \cap A \leq A \quad (\text{incR})$$

$$B \rightarrow A \cap A' \leq (B \rightarrow A) \cap (B \rightarrow A') \quad (\rightarrow \cap)$$

$$A \leq \mathbb{U} \quad (\text{Top}) \qquad \mathbb{U} \leq A \rightarrow \mathbb{U} \quad (\rightarrow \mathbb{U})$$

$$\frac{A \leq B \quad A \leq B'}{A \leq B \cap B'} \quad (\text{Glb}) \qquad \frac{A \leq B \quad B \leq B'}{A \leq B'} \quad (\text{Trans})$$

$$\frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'} \quad (\rightarrow)$$

3. An intersection type theory (itt) $\mathcal{T}$ is determined by a set of type constants $\mathbb{A}$ and a subtyping relation on the set $\mathbb{T}_{\mathbb{A}}$ which includes those in (ii), i.e. $\mathcal{T} = \langle \mathbb{A}, \leq_{\mathcal{T}} \rangle$.

We assume that $\cap$ has higher priority than $\rightarrow$, the latter being right-associative e.g. $A \rightarrow B \rightarrow C \cap C'$ means $A \rightarrow (B \rightarrow (C \cap C'))$. We write $A \sim B$ as short for $A \leq B$ and $B \leq A$. The above rules permit us to show that $\sim$ is a congruence w.r.t. to the signature of intersection types, given by the $\cap$ and $\rightarrow$ constructors, and that $\mathbb{T}/\sim$ is a meet semilattice. In this paper we will discuss various type assignment systems, here we just recall the general definition of Intersection Type Assignment system [3, 8]. We also recall that a type is said to be *inhabited* when it is the type of a closed λ -term.

► **Definition 3** (Intersection Type Assignment System). The intersection type assignment system induced by an itt $\mathcal{T} = \langle \mathbb{A}, \leq_{\mathcal{T}} \rangle$ is a formal system deriving judgements of the shape $\Gamma \vdash_{\mathcal{T}} M : A$, where $A \in \mathbb{T}_{\mathbb{A}}$ and a basis Γ is a finite mapping from term variables to types in $\mathbb{T}_{\mathbb{A}}$:

$$\Gamma ::= \emptyset \mid \Gamma, x : A.$$

The axioms and rules of the type system are the following:

$$\begin{array}{c}
\frac{}{\Gamma, x : A \vdash x : A} \text{ (Ax)} \qquad \frac{}{\Gamma \vdash M : \mathbb{U}} \text{ (U)} \\
\frac{\Gamma, x : B \vdash M : A}{\Gamma \vdash \lambda x.M : B \rightarrow A} (\rightarrow) \qquad \frac{\Gamma \vdash M : B \rightarrow A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : A} (\rightarrow E) \\
\frac{\Gamma \vdash M : B \quad \Gamma \vdash M : A}{\Gamma \vdash M : B \cap A} (\cap I) \qquad \frac{\Gamma \vdash M : B \quad B \leq_{\tau} A}{\Gamma \vdash M : A} (\leq)
\end{array}$$

3 Fuzzy Propositions-as-types

In this section we define type assignment systems for untyped λ -terms, where types are formulæ in minimal Fuzzy Logic. The systems can be viewed according to two different perspectives. On the one hand, viewing fuzzy formulæ as types, λ -terms can be viewed as proofs for this fragment of Fuzzy Logic. Thus we can characterise the sets of fuzzy formulæ which are inhabited, thereby providing an alternate route (see Corollary 11) to showing that **BCK**-logic is included in all Fuzzy Logics, see [14]. On the other hand, evaluating the semantics of the formulæ appearing as types (by means of an assignment to fuzzy propositional variables, ρ , into $[0,1]$, endowed with a residuated T-norm), these systems can be also viewed as systems for assigning a *confidence degree* to λ -terms. In Remark 25, we will see that this degree can be viewed as the confidence in their membership to a universal type A such that $A \sim A \rightarrow A$.

In particular, we discuss two type assignment systems $\mathcal{F}$ and $\mathcal{Q}$, which differ only in that, the first is *contractive* and hence assumptions are taken to be sets, while the second is *non-contractive* and assumptions are taken to be multisets.

3.1 Minimal Fuzzy Logic - Qualitative Case

We introduce a fuzzy assignment system, *i.e.* a system for deriving judgements of the shape $\Gamma \vdash M : d$, where Γ is a context, M is a λ -term, and d is a formula of minimal Fuzzy Logic.

► **Definition 4** (Fuzzy Assumptions, Fuzzy Qualitative Contexts).

- A fuzzy assumption is a pair of the form (x, d) , usually written as $x : d$, where x is a variable and d is a fuzzy formula. The fuzzy assumption, (x, d) , means that the variable x has confidence degree $\llbracket d \rrbracket_{\rho}^T$, once we evaluate the formula d according to some assignment ρ in $[0,1]$ endowed with a residuated T-norm.
- A fuzzy qualitative context Γ is a finite set of fuzzy assumptions. Note that Γ may contain two or more different assumptions for the same variable, with possibly different confidence degrees.

► **Definition 5** (The type assignment system $\mathcal{F}$). The qualitative Fuzzy Assignment System, $\mathcal{F}$, assigns fuzzy formulas to pure λ -terms according to the following rules:

$$\begin{array}{c}
\frac{}{\Gamma, x : d \vdash_{\mathcal{F}} x : d} \text{ (Ax)} \\
\frac{\Gamma, x : d_1, \dots, x : d_n \vdash_{\mathcal{F}} M : d \quad x \notin \Gamma}{\Gamma \vdash_{\mathcal{F}} \lambda x.M : d_1 \otimes \dots \otimes d_n \Rightarrow d} (\Rightarrow I) \\
\frac{\Gamma \vdash_{\mathcal{F}} M : d_1 \quad \Gamma \vdash_{\mathcal{F}} N : d_2}{\Gamma \vdash_{\mathcal{F}} MN : d_1 \otimes d_2} (\Rightarrow E)
\end{array}$$

► **Remark 6.** As we pointed out already, given an interpretation of fuzzy propositional variables ρ in $[0,1]$, endowed with a residuated T-norm, we can evaluate a fuzzy formula d . Thus $\mathcal{F}$ can be viewed also as a system for assigning degrees of confidence to λ -terms, once we view the judgements of $\mathcal{F}$ as meaning

$$x_1 : \llbracket d_1 \rrbracket_\rho^T, \dots, x_n : \llbracket d_n \rrbracket_\rho^T \vdash_{\mathcal{F}} M : \llbracket d \rrbracket_\rho^T.$$

Hence, when convenient and clear from the context, we may conflate d with its value $\llbracket d \rrbracket_\rho^T \in [0, 1]$, given ρ and T, and blur the distinction between the two interpretations of the judgement in $\mathcal{F}$.

The following comments on the rules of the system $\mathcal{F}$ are in order.

- In Rule ($\Rightarrow$ I), we use the residuum, which corresponds to fuzzy implication.
- Since system $\vdash_{\mathcal{F}}$ satisfies the contraction rule, then we can always assume that any fresh variable x such that $x \notin M$ appears as the subject of a fuzzy assumption in Γ . Thus from $\Gamma \vdash_{\mathcal{F}} M : d$ by Rule ($\Rightarrow$ I) we can conclude $\Gamma \vdash_{\mathcal{F}} \lambda x.M : e \Rightarrow d$, for any e .
- When $[0,1]$ is endowed with a T-norm, we always have that $x \otimes y \leq x$ and $x \leq y \Rightarrow x$, see Proposition 35. Hence when we evaluate the Fuzzy Logic formulæ involved in Rule ($\Rightarrow$ E), by means of any ρ in $[0,1]$, we see that ($\Rightarrow$ E) *decreases* confidence, while ($\Rightarrow$ I) *increases* confidence. Thus, the system exhibits the feature which permits Fuzzy Logic to explain the so-called *Sorites Paradox* [15], namely that each application of *Modus Ponens* decreases the confidence degree. In the context of types this permits to detect the number of applications which have occurred.
- When dealing with multiple assumptions, we use the T-norm, *i.e.* weak conjunction, in the antecedent of $\Rightarrow$. Alternatively, we could have used the $\wedge$ operator corresponding to fuzzy strong conjunction [14], namely the *minimum*. We do not make this choice because it would make the system more difficult to deal with, since the T-norm is well-behaved with respect to its residuum, see Proposition 35.8, and moreover the feature of Fuzzy Logic of potentially registering the number of well-typed applications, mentioned above, would be lost. This makes our system more resource-sensitive. Of course, if we have in mind Gödel-Dummet Logic, there is no difference.

When viewing system $\mathcal{F}$ as a system for assigning confidence degrees (given an assignment ρ in $[0,1]$ endowed with a T-norm) a few more comments are in order:

- it is convenient to assume that ρ does not assign the trivial value 0 to any propositional fuzzy variable;
- it might be convenient to add in Rule ($\Rightarrow$ E) the side condition $\llbracket d_1 \otimes d_2 \rrbracket > 0$, if we want to get rid of trivial judgement where the type evaluates to 0.

The system $\mathcal{F}$ is very permissive and allows all λ -terms to be typed with a minimal fuzzy formula, albeit not a tautology, as shown by the following example.

► **Example 7** (Self-application).

$$\frac{\frac{x : d \vdash_{\mathcal{F}} x : d \quad x : d \vdash_{\mathcal{F}} x : d}{x : d \vdash_{\mathcal{F}} xx : d \otimes d}}{\vdash_{\mathcal{F}} \lambda x.xx : d \Rightarrow d \otimes d.}$$

Clearly for all T-norms which are non-idempotent and for all ρ 's such that $\llbracket d \rrbracket_\rho^T \neq 1$, we have $\llbracket d \Rightarrow d \otimes d \rrbracket_\rho^T \neq 1$.

The following Theorem 8, which has the form of a deduction theorem, and Proposition 9 characterise which proofs, *i.e.* λ -terms, yield valid fuzzy tautologies in system $\mathcal{F}$, when the predicates in the typing judgements are evaluated according to a left-continuous T-norm.

► **Theorem 8.**

1. If M is affine, then for all assignments ρ to propositional fuzzy variables of values in $[0,1]$, endowed with a left-continuous T -norm, we have:

$$x_1 : d_1, \dots, x_n : d_n \vdash_{\mathcal{F}} M : d$$

implies

$$\llbracket d_1 \Rightarrow \dots \Rightarrow d_n \Rightarrow d \rrbracket_{\rho}^T = 1.$$

2. If the T -norm is idempotent, i.e. T is Gödel's T -norm (see Definition 34), then

$$x_1 : d_1^1, \dots, x_1 : d_1^{m_1}, \dots, x_n : d_n^1, \dots, x_n : d_n^{m_n} \vdash_{\mathcal{F}} M : d$$

implies

$$\llbracket d_1^1 \otimes \dots \otimes d_1^{m_1} \Rightarrow \dots \Rightarrow d_n^1 \otimes \dots \otimes d_n^{m_n} \Rightarrow d \rrbracket_{\rho}^T = 1.$$

Proof. If $\Gamma = x_1 : d_1, \dots, x_n : d_n$ we use $\bigotimes \Gamma$ to denote $d_1 \otimes \dots \otimes d_n$.

1. Let Γ_M be the set of the assumptions in Γ for the variables which occur in M . By induction on the derivation in $\mathcal{F}$, we prove that if M is affine and $\Gamma \vdash_{\mathcal{F}} M : d$, then $\llbracket \bigotimes \Gamma_M \Rightarrow d \rrbracket_{\rho}^T = 1$, or equivalently, by Proposition 35 in Appendix A, that $\llbracket \bigotimes \Gamma_M \rrbracket_{\rho}^T \leq \llbracket d \rrbracket_{\rho}^T$.
 (Ax). Immediate, since $\llbracket \bigotimes \Gamma \otimes d \rrbracket_{\rho}^T \leq \llbracket d \rrbracket_{\rho}^T$ by Definition 33.
 ($\Rightarrow$ I). By induction hypothesis and Proposition 35.7.
 ($\Rightarrow$ E). By affinity we know that $\Gamma_M \cup \Gamma_N = \Gamma_{MN}$, hence we apply the monotonicity of the norm to the induction hypotheses.
 Finally using again Propositions 35.6 and 35.7 we get the result.
2. By induction on the derivation in $\mathcal{F}$, we prove that if $\Gamma \vdash_{\mathcal{F}} M : d$, then $\llbracket \bigotimes \Gamma \Rightarrow d \rrbracket_{\rho}^T = 1$, or equivalently by Definition 34 in Appendix A, that $\llbracket \bigotimes \Gamma \rrbracket_{\rho}^T \leq \llbracket d \rrbracket_{\rho}^T$.
 (Ax). Immediate, since $\llbracket \bigotimes \Gamma \otimes d \rrbracket_{\rho}^T \leq \llbracket d \rrbracket_{\rho}^T$ by Definition 33.
 ($\Rightarrow$ I). By induction hypothesis and Proposition 35.7.
 ($\Rightarrow$ E). We use the fact that if $\llbracket d \Rightarrow e_1 \rrbracket_{\rho}^T = 1$ and $\llbracket d \Rightarrow e_2 \rrbracket_{\rho}^T = 1$, then we also have $\llbracket d \Rightarrow e_1 \otimes e_2 \rrbracket_{\rho}^T = 1$ since $e_1 \otimes e_2 = \min\{e_1, e_2\}$. Then we conclude as above. ◀

We now give a sort of converse to Theorem 8, namely that using a non-idempotent norm, non-affine λ -terms do not inhabit tautologies.

► **Proposition 9.** Let M be a λ -term where a variable, say x , occurs more than once in M , then we can derive a judgement $\Gamma \vdash_{\mathcal{F}} M : d$ where, for some T -norm, and assignment ρ , $\llbracket d \rrbracket_{\rho}^T$ does not evaluate to 1, and hence, d is not a fuzzy tautology.

Proof. We take Goguen's *product* T -norm, see Definition 34 in the Appendix. We define a derivation $\mathcal{D} : \Gamma \vdash_{\mathcal{F}} M : d$ as follows. Fuzzy assumptions in $\mathcal{D}$ assign propositional variables to λ -variables. The assignment ρ assigns the value 1 to propositional variables typing λ -variables which are vacuously abstracted and the value $\frac{1}{2}$ to propositional variables typing λ -variables which do occur in M . In the applications of Rule ($\Rightarrow$ I) we have $n = 1$. We can show that for every judgement $\Gamma' \vdash N : d'$ which appears in $\mathcal{D}$:

$$\llbracket d' \rrbracket_{\rho}^T = \frac{2^{\#_{\lambda}(N)}}{2^{\#_{var}(N)}}$$

where $\#_{\lambda}(N)$ denotes the number of non vacuous abstractions in N , and $\#_{var}(N)$ denotes the number of variable occurrences in N . Clearly we have that $\#_{\lambda}(N) \leq \#_{var}(N)$, the two being equal if and only if the term is affine.

The proof is by induction on $\mathcal{D}$. The only interesting case is Rule ($\Rightarrow$ I) when x occurs in M' :

$$\frac{\Gamma, x : X \vdash_{\mathcal{F}} M' : d' \quad x \notin \Gamma}{\Gamma \vdash_{\mathcal{F}} \lambda x.M' : X \Rightarrow d'}$$

By induction hypothesis $\llbracket d' \rrbracket_{\rho}^T = \frac{2^{\#_{\lambda}(M')}}{2^{\#_{\text{var}}(M')}} \llbracket d' \rrbracket_{\rho}^T$ where $\#_{\lambda}(M') < \#_{\text{var}}(M')$ since x is free in M' . If $\#_{\lambda}(M') < \#_{\text{var}}(M') - 1$, then $\llbracket X \Rightarrow d' \rrbracket_{\rho}^T = \frac{2^{\#_{\lambda}(M')}}{2^{\#_{\text{var}}(M')}} \times 2 = \frac{2^{\#_{\lambda}(\lambda x.M')}}{2^{\#_{\text{var}}(\lambda x.M')}} \llbracket d' \rrbracket_{\rho}^T$. If $\#_{\lambda}(M') = \#_{\text{var}}(M') - 1$, then $\llbracket X \Rightarrow d' \rrbracket_{\rho}^T = 1 = \frac{2^{\#_{\lambda}(\lambda x.M')}}{2^{\#_{\text{var}}(\lambda x.M')}} \llbracket d' \rrbracket_{\rho}^T$. Since the term is non-affine there exists at least one judgement in the derivation $\mathcal{D}$ such that for all subsequent judgements $\llbracket d' \rrbracket_{\rho}^T < 1$. $\blacktriangleleft$

We illustrate the above results by the following example, which deals with an affine combinator using Theorem 8.1.

► **Example 10.** Proposition 9 implies that if $\vdash_{\mathcal{F}} \mathbf{S} \equiv \lambda xyz.xz(yz) : d$, then it is not the case that $d = 1$ for every left-continuous T-norm, and assignment ρ . However, for a linear variant of $\mathbf{S}$, namely $\lambda xyzw.xz(yw)$ we have that if $\vdash_{\mathcal{F}} \lambda xyzw.xz(yw) : d$, then the evaluation of d in any left-continuous T-norm and assignment ρ is 1, by Theorem 8.1.

The following corollary provides the natural connection between simple types of affine λ -terms and formulæ in **BCK**-logic, see [14].

► **Corollary 11.** *All simple types inhabited by an affine λ -term are tautologies in all Fuzzy Logics, once we read $\rightarrow$ as $\Rightarrow$.*

Proof. Let $\mathcal{D} : \vdash M : A$ be a derivation of the simple type A for the closed affine λ -term M . We can, inductively, parallel the derivation $\mathcal{D}$ in the the system $\vdash_{\mathcal{F}}$, appropriately replacing arrows. The main difference in the new derivation is that the application of the *modus ponens* rule in the simple types assignment system, namely $\frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$ is replaced by $\frac{\Gamma \vdash M : A' \Rightarrow B' \quad \Gamma \vdash N : A'}{\Gamma \vdash MN : (A' \Rightarrow B') \otimes A'}$, where A', B' are the corresponding types. Proceeding top-down in the derivation consider the first use of the application rule in a branch of the derivation. By Theorem 8.1 we have that $\otimes \Gamma' \Rightarrow (A' \Rightarrow B') \otimes A'$ is a tautology, but since $\llbracket (A' \Rightarrow B') \otimes A' \rrbracket_{\rho}^T \leq \llbracket B' \rrbracket_{\rho}^T$ by Proposition 35.8 we have also that $\llbracket \otimes \Gamma' \Rightarrow B' \rrbracket_{\rho}^T = 1$. Recall that each application of modus ponens decreases the confidence value. Proceeding along the derivation we can then replace all occurrences of $(A' \Rightarrow B') \otimes A'$ by B' still preserving the fact that its value is 1. Hence all simple types inhabited by an affine λ -term are fuzzy tautologies. $\blacktriangleleft$

It is folklore that the *simple* types inhabited by affine λ -terms are the simple types which can be assigned to applications of the combinators $\mathbf{B} \equiv \lambda xyz.x(yz)$, $\mathbf{C} \equiv \lambda xyz.xzy$, and $\mathbf{K} \equiv \lambda xy.x$. These are the tautologies of **BCK**-logic. By the above Corollary 11 therefore, we recover along a different route the fact that all such implicational formulæ hold in the Minimal T-norm Fuzzy Logic MTL, see [14], and hence in all its extensions, which include all left-continuous T-norm Fuzzy Logics.

Similarly from Theorem 8.2, we derive that all minimal intuitionistic tautologies are valid formulæ for the idempotent T-norm, namely Göde's T-norm. These are the purely implicational, *i.e.* minimal, intuitionistic tautologies, which are all valid in Gödel-Dummet Fuzzy Logic [12]. Clearly every minimal intuitionistic tautology is inhabited by a simply typed λ -term. Less obvious is that extending the inhabitants to all λ -terms does not increase the class of inhabited types.

3.2 Minimal Fuzzy Logic - Quantitative Case

In this subsection we modify system $\mathcal{F}$ by taking contexts to be *multisets*, namely we allow them to contain two or more *equal*, or different, fuzzy assumptions, in the sense of Definition 4, for the same variable. In fact, if we interpret the formulæ via a norm and an assignment these are indeed fuzzy multisets. We will see that by doing so all inhabited Fuzzy Logic formulæ necessarily have value 1, in any left-continuous T-norm and assignment ρ .

► **Definition 12** (Fuzzy Quantitative Context). *A fuzzy quantitative context Γ is a finite multiset of fuzzy assumptions.*

The above definition is just the *multiset* version of Definition 4, which, interpreting d 's in $[0,1]$, is the appropriate definition of *fuzzy multiset*.

► **Definition 13** (Quantitative Fuzzy Assignment, $\mathcal{Q}$). *The rules of the quantitative Fuzzy Assignment System, $\mathcal{Q}$, are the following:*

$$\begin{array}{c} \frac{}{x : d \vdash_{\mathcal{Q}} x : d} \text{ (Ax)} \\[10pt] \frac{\Gamma, x : d_1, \dots, x : d_n \vdash_{\mathcal{Q}} M : d \quad x \notin \Gamma}{\Gamma \vdash_{\mathcal{Q}} \lambda x. M : d_1 \otimes \dots \otimes d_n \Rightarrow d} \text{ (}\Rightarrow\text{I)} \\[10pt] \frac{\Gamma \vdash_{\mathcal{Q}} M : d \quad x \notin \Gamma}{\Gamma \vdash_{\mathcal{Q}} \lambda x. M : 1 \Rightarrow d} \text{ (}\Rightarrow\text{0)} \\[10pt] \frac{\Gamma_1 \vdash_{\mathcal{Q}} M : d_1 \quad \Gamma_2 \vdash_{\mathcal{Q}} N : d_2}{\Gamma_1 :: \Gamma_2 \vdash_{\mathcal{Q}} MN : d_1 \otimes d_2} \text{ (}\Rightarrow\text{E)} \end{array}$$

where $::$ denotes multiset union.

Notice that in (Ax), above, there is no extra-context; this enforces the *relevance* of this quantitative system. Therefore, we need to add Rule ($\Rightarrow$ 0) to take care explicitly of *weakening*.

Theorem 15, below, shows that the inhabited fuzzy formulæ in $\mathcal{Q}$ evaluate to a confidence degree equal to 1 for any left-continuous T-norm, and any assignment ρ . First, we need a lemma.

► **Lemma 14** ($\mathcal{Q}$ -Invariance). *Let $\otimes \Gamma$ denote the tensor product of all confidence degrees assigned to variables in the context Γ . If $\Gamma \vdash_{\mathcal{Q}} M : d$, then $\llbracket \otimes \Gamma \rrbracket_{\rho}^T \leq \llbracket d \rrbracket_{\rho}^T$, for any T-norm, and any assignment ρ .*

Proof.

By induction on the derivation in $\mathcal{Q}$:

(Ax). Immediate;

($\Rightarrow$ I) and ($\Rightarrow$ 0). Immediate by induction hypothesis and Definition 33;

($\Rightarrow$ E). By monotonicity of the T-norm, see Definition 32. ◀

► **Theorem 15.**

If the T-norm is left-continuous, then

$$x_1 : d_1^1, \dots, x_1 : d_1^{m_1}, \dots, x_n : d_n^1, \dots, x_n : d_n^{m_n} \vdash_{\mathcal{Q}} M : d$$

implies

$$\llbracket d_1^1 \otimes \dots \otimes d_1^{m_1} \Rightarrow \dots \Rightarrow d_n^1 \otimes \dots \otimes d_n^{m_n} \Rightarrow d \rrbracket_{\rho}^T = 1$$

for all assignments ρ .

Proof. By Lemma 14 we see that if $\Gamma \vdash_Q M : d$, then $\llbracket \otimes \Gamma \rrbracket_\rho^T \leq \llbracket d \rrbracket_\rho^T$, hence using Proposition 35.5 in Appendix A we get the result. $\blacktriangleleft$

It is natural, now, to ask the following question: what are the inhabited types in System $\mathcal{Q}$? The simple answer is given in the following proposition.

► **Proposition 16.** *The class of inhabited types in $\mathcal{Q}$ is the same as that of the types inhabited in $\mathcal{F}$ by affine terms.*

Proof. Taking different variables for different uses of the Axiom (Ax), any derivation in $\mathcal{Q}$ can be turned in a derivation in $\mathcal{F}$ for an affine term. $\blacktriangleleft$

We end this subsection remarking that both in $\mathcal{F}$ and $\mathcal{Q}$ all λ -terms can receive infinitely many degrees of confidence depending on which (quantitative) fuzzy contexts we start from. For instance, using the trivial assignment, all terms can receive degree of confidence 1, even unsolvable terms, and hence also unsolvable terms can play the role of inhabitants. Fuzzy Intersection Type theories allow us to harness the class of derivations, as we will see in Section 4.

3.3 Subject Conversion

For all type assignment systems, it is natural to ask whether types are preserved under β -conversion, namely reduction/expansion, of the term. Since both $\mathcal{F}$ and $\mathcal{Q}$ are sensitive to the number of times an assumption appears in the proof, albeit in different ways, preservation in reduction or expansion of the subject are problematic, both as far as the fuzzy formula itself as well as its evaluation. The following counterexamples and proposition illustrate the situation.

► **Counterexample 17** (Subject Conversion Fails in $\mathcal{F}$ for $\Delta \mathbf{I} \rightarrow_\beta \mathbf{II}$). *We recall that $\Delta \equiv \lambda x.xx$ and $\mathbf{I} \equiv \lambda x.x$. Consider the following reduction $(\lambda x.xx)(\lambda x.x) \rightarrow_\beta (\lambda x.x)(\lambda x.x)$, we have the derivation:*

$$\frac{\frac{\frac{x : d \vdash_{\mathcal{F}} x : d}{x : d \vdash_{\mathcal{F}} xx : d \otimes d}}{\emptyset \vdash_{\mathcal{F}} \lambda x.xx : d \Rightarrow d \otimes d} \quad \frac{x : d \vdash_{\mathcal{F}} x : d}{\emptyset \vdash_{\mathcal{F}} \lambda x.x : d \Rightarrow d}}{\emptyset \vdash_{\mathcal{F}} (\lambda x.xx)(\lambda x.x) : (d \Rightarrow d \otimes d) \otimes (d \Rightarrow d)}$$

and the derivation:

$$\frac{\frac{x : d \vdash_{\mathcal{F}} x : d}{\emptyset \vdash_{\mathcal{F}} \lambda x.x : d \Rightarrow d} \quad \frac{x : d \vdash_{\mathcal{F}} x : d}{\emptyset \vdash_{\mathcal{F}} \lambda x.x : d \Rightarrow d}}{\emptyset \vdash_{\mathcal{F}} (\lambda x.x)(\lambda x.x) : (d \Rightarrow d) \otimes (d \Rightarrow d)}.$$

But in Goguen's product norm, for a ρ such that $\llbracket d \rrbracket_\rho^T \prec 1$ we have $\llbracket d \Rightarrow d \rrbracket_\rho^T = 1$, but $\llbracket (d \Rightarrow d \otimes d) \otimes (d \Rightarrow d) \rrbracket_\rho^T \prec \llbracket (d \Rightarrow d) \otimes (d \Rightarrow d) \rrbracket_\rho^T = 1$, in general. Hence in $\mathcal{F}$ the reduct can have a higher confidence degree than the redex.

► **Counterexample 18** (Subject Conversion fails in $\mathcal{F}$ for $\mathbf{2} \mathbf{2} \rightarrow_\beta \mathbf{4}$). *We recall that $\mathbf{2} \equiv \lambda xy.x(xy)$ and $\mathbf{4} \equiv \lambda xy.x(x(xy))$ are Church encodings of the natural numbers 2 and 4. Consider the following reduction $\mathbf{2} \mathbf{2} \rightarrow_\beta \mathbf{4}$. We have the derivation:*

$$\frac{\mathcal{D} \quad \mathcal{D}}{\emptyset \vdash_{\mathcal{F}} \mathbf{2} \mathbf{2} : (d \Rightarrow d \Rightarrow d \otimes d \otimes d) \otimes (d \Rightarrow d \Rightarrow d \otimes d \otimes d)},$$

where $\mathcal{D}$ denotes the derivation:

$$\frac{\frac{\frac{x : d \vdash_{\mathcal{F}} x : d \quad y : d \vdash_{\mathcal{F}} y : d}{x : d, y : d \vdash_{\mathcal{F}} xy : d \otimes d}}{x : d, y : d \vdash_{\mathcal{F}} x(xy) : d \otimes d \otimes d}}{x : d \vdash_{\mathcal{F}} \lambda y. x(xy) : d \Rightarrow d \otimes d \otimes d}}{\emptyset \vdash_{\mathcal{F}} \lambda xy. x(xy) : d \Rightarrow d \Rightarrow d \otimes d \otimes d,}$$

and also the derivation:

$$\frac{\frac{\frac{\frac{x : d \vdash_{\mathcal{F}} x : d \quad y : d \vdash_{\mathcal{F}} y : d}{x : d, y : d \vdash_{\mathcal{F}} xy : d \otimes d}}{x : d, y : d \vdash_{\mathcal{F}} x(xy) : d \otimes d \otimes d}}{x : d, y : d \vdash_{\mathcal{F}} x(x(xy)) : d \otimes d \otimes d \otimes d}}{x : d \vdash_{\mathcal{F}} \lambda y. x(x(xy)) : d \Rightarrow d \otimes d \otimes d \otimes d,}}{\emptyset \vdash_{\mathcal{F}} \lambda xy. x(x(xy)) : d \Rightarrow d \Rightarrow d \otimes d \otimes d \otimes d,}$$

but in Goguen's product T-norm for a ρ such that $\llbracket d \rrbracket_{\rho}^T \leq 1$ the value of the confidence degree of the reduct is equal to $\llbracket d \rrbracket_{\rho}^T \otimes \llbracket d \rrbracket_{\rho}^T \otimes \llbracket d \rrbracket_{\rho}^T$, which is strictly smaller than the confidence degree of the redex, which is equal to $\llbracket d \rrbracket_{\rho}^T \otimes \llbracket d \rrbracket_{\rho}^T$.

The counterexamples above show that confidence degree of the reduct can be strictly larger or strictly smaller than that of the redex. So we have the following proposition.

► **Proposition 19.** *Subject conversion holds in System $\mathcal{F}$ if and only if the norm is idempotent.*

Proof. The “only if” part follows almost directly from Counterexample 17. Let $\llbracket d \rrbracket_{\rho}^T = a$. Since the T-norm is left continuous we have that $a \Rightarrow a \otimes a = \sup\{c \mid a \otimes c \leq a \otimes a\}$. If this were 1, as it would be necessary for subject reduction, then we would have that $a \leq a \otimes a$, and, since the converse always holds, we would have that $a = a \otimes a$, namely the T-norm would be idempotent.

The “if” part follows from the fact that if the T-norm is idempotent by Theorem 8. 2 we have that the value of the confidence degree inhabited by closed λ -terms is always 1. ◀

If we were to consider only affine terms in $\mathcal{F}$, β -conversion would naturally hold. Subject conversion for $\mathcal{Q}$ is satisfactory only for closed terms as the following proposition and counterexamples show.

► **Proposition 20.** *Subject Reduction and Subject Expansion hold in System $\mathcal{Q}$ for closed λ -terms, but not for open terms.*

Proof. In system $\mathcal{Q}$, the value of the confidence degree of fully abstracted inhabited types is always 1, thus subject conversion holds for closed terms. ◀

► **Counterexample 21** (Subject Conversion Fails in $\mathcal{Q}$ for $\Delta x \rightarrow_{\beta} xx$). *Consider the following reduction $(\lambda x. xx)x \rightarrow_{\beta} xx$, we have the derivation:*

$$\frac{\frac{\frac{x : d \vdash_{\mathcal{Q}} x : d \quad x : d \vdash_{\mathcal{Q}} x : d}{x : d, x : d \vdash_{\mathcal{Q}} xx : d \otimes d}}{\emptyset \vdash_{\mathcal{Q}} \lambda x. xx : d \otimes d \Rightarrow d \otimes d} \quad x : d \vdash_{\mathcal{Q}} x : d}{x : d \vdash_{\mathcal{Q}} (\lambda x. xx)x : (d \otimes d \Rightarrow d \otimes d) \otimes d}$$

and the derivation:

$$\frac{x : d \vdash_{\mathcal{Q}} x : d \quad x : d \vdash_{\mathcal{Q}} x : d}{x : d, x : d \vdash_{\mathcal{Q}} xx : d \otimes d}$$

As we can see the context of the conclusion of the second derivation has expanded.

► **Counterexample 22** (Subject Conversion Fails in $\mathcal{Q}$ for $(\lambda x.y)z \rightarrow_{\beta} y$). We have the derivation:

$$\frac{\frac{y : d \vdash_{\mathcal{Q}} y : d}{y : d \vdash_{\mathcal{Q}} \lambda x.y : 1 \Rightarrow d} \quad z : d \vdash_{\mathcal{Q}} z : d}{y : d, z : d \vdash_{\mathcal{Q}} (\lambda x.y)z : (1 \Rightarrow d) \otimes d}$$

which has a longer context than its reduct, namely $y : d \vdash_{\mathcal{Q}} y : d$.

In conclusion the problems with subject conversion come from the interaction between non-idempotent T-norms and duplication or erasure of variables during β -reduction. The remarks above and Proposition 20 show that subject conversion can still work in fuzzy settings, for example, if we assume Gödel's T-norm for affine terms or in system $\mathcal{Q}$ for closed terms.

4 Fuzzy Intersection Types

We are now in the position of fruitfully studying the combination of intersection types, see Definition 2, with fuzzy type-membership. Namely, we introduce a *fuzzy* type assignment system assigning intersection types to pure λ -terms, decorating appropriately the Intersection Type Assignment System, see *e.g.* [2, 3, 8], with minimal fuzzy formulæ.

► **Definition 23** (Typing assumption, typing context).

1. A typing assumption is a triple of the form (x, A, d) , usually written as $x :_d A$, where x is a variable, A is an intersection type and d is a fuzzy formula, see Section 2.1, or, equivalently, a real number in $[0, 1]$ obtained by its evaluation w.r.t. a T-norm and an assignment ρ . Its intended meaning is that x belongs to the type A with degree of confidence $\llbracket d \rrbracket_{\rho}^T$.
2. A typing context Γ is a finite multiset of typing assumptions with non-zero degrees.

Notice that a context Γ may contain two or more assumptions for the same variable, possibly with the same type, and the same confidence degree. Thus, a typing context can be seen as a *fuzzy multiset*, once we evaluate the fuzzy formulæ appearing in the indices, with respect to a norm and an assignment of values to the propositional fuzzy variables.

► **Definition 24** (Quantitative Fuzzy Intersection Types $\mathcal{Q}^\cap$). *The typing rules of the Fuzzy Intersection Types $\mathcal{Q}^\cap$ induced by the type theory $\langle \mathbb{A}, \leq_T \rangle$ are the following:*

$$\begin{array}{c}
\frac{}{x :_d A \vdash_{\mathcal{Q}^\cap} x :_d A} (Ax) \qquad \frac{}{\vdash_{\mathcal{Q}^\cap} M :_1 \mathbb{U}} (U) \\
\\
\frac{\Gamma, x :_{d_1} A_1, \dots, x :_{d_n} A_n \vdash_{\mathcal{Q}^\cap} M :_d A \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{Q}^\cap} \lambda x. M :_{d_1 \otimes \dots \otimes d_n \Rightarrow d} A_1 \cap \dots \cap A_n \rightarrow A} (\rightarrow I) \\
\\
\frac{\Gamma \vdash_{\mathcal{Q}^\cap} M :_d A \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash_{\mathcal{Q}^\cap} \lambda x. M :_{1 \Rightarrow d} B \rightarrow A} (\rightarrow 0) \\
\\
\frac{\Gamma_1 \vdash_{\mathcal{Q}^\cap} M :_{d_1} A \rightarrow B \quad \Gamma_2 \vdash_{\mathcal{Q}^\cap} N :_{d_2} A}{\Gamma_1 :: \Gamma_2 \vdash_{\mathcal{Q}^\cap} MN :_{d_1 \otimes d_2} B} (\rightarrow E) \\
\\
\frac{\Gamma_1 \vdash_{\mathcal{Q}^\cap} M :_{d_1} A \quad \Gamma_2 \vdash_{\mathcal{Q}^\cap} M :_{d_2} B}{\Gamma_1 :: \Gamma_2 \vdash_{\mathcal{Q}^\cap} M :_{d_1 \otimes d_2} A \cap B} (\cap I) \qquad \frac{\Gamma \vdash_{\mathcal{Q}^\cap} M :_d A \quad A \leq_T B}{\Gamma \vdash_{\mathcal{Q}^\cap} M :_d B} (\leq)
\end{array}$$

where $\Gamma_1 :: \Gamma_2$ denotes multiset union, i.e. multiplicities of occurrences are not erased.

Some comments are in order.

- How does the system $\mathcal{Q}^\cap$ qualify with respect to *contraction* and *idempotency*? It is contraction-free with respect to the fuzzy formula indexing the “:_d” in the judgement $\Gamma \vdash_{\mathcal{Q}^\cap} M :_d A$, but although contexts are multisets, the system is nonetheless *idempotent* as far as type intersections are concerned, e.g. in Rule $(\rightarrow I)$. Hence, since $\otimes$ is not idempotent in general, the judgements store the information on the number of times an assumption is *used*, as if the system were non-contractive, only in a different format. Therefore $\mathcal{Q}^\cap$ has a quantitative nature.
- The crisp Intersection Type Assignment System given in Definition 3 can be obtained from $\vdash_{\mathcal{Q}^\cap}$ by erasing all fuzzy coefficients from the “:_d” symbol, namely $(\Gamma \vdash_{\mathcal{Q}^\cap} M :_d A)^\#$ yields $\Gamma' \vdash_{\mathcal{T}} M : A$ for $\mathcal{T}$ with the same subtyping, where Γ' is the set obtained from Γ erasing multiple assumptions after having erased the confidence degree in each assumption; we denote by $\Gamma^\#$ the crisp context thus obtained.
- Removing the coefficient d in the judgements does not yield the same system that we would get if we take all the d ’s in assumptions to be the constant 1. Indeed all d ’s would evaluate to 1, but the indexes still carry information. This is still a fuzzy system, but just with a trivial value. If, on the other hand, we erase the coefficient, we completely forget information and collapse everything to the usual crisp intersection type system where contexts are needlessly multisets.
- In principle, we could introduce a fuzzy intersection type assignment system building on top of $\mathcal{F}$ rather than $\mathcal{Q}$, but then the value of the fuzzy index would not be preserved by reduction or expansion, unless we would focus on Gödel’s T-norm, or affine λ -terms.
- Notice, incidentally, that when evaluating the confidence degrees w.r.t. a T-norm and an assignment ρ , the fuzzy constructor $\Rightarrow$ behaves w.r.t. $\leq$ contravariantly on domains and covariantly on codomains, as does the intersection types constructor $\rightarrow$, see Definition 2. Namely, if $\llbracket d \rrbracket_\rho^T \leq \llbracket d' \rrbracket_\rho^T$ and $\llbracket e \rrbracket_\rho^T \leq \llbracket e' \rrbracket_\rho^T$, then $\llbracket d' \Rightarrow e \rrbracket_\rho^T \leq \llbracket d \Rightarrow e' \rrbracket_\rho^T$. However $\otimes$ does not behave as $\cap$. For instance Rule $(\rightarrow \cap)$ (see Definition 2) fails in Goguen’s T-norm.
- In the introduction Rules $(\rightarrow I)$ and $(\cap I)$ rather than $d_1 \otimes d_2$, we could consider instead $d_1 \wedge d_2$, which is usually interpreted by the minimum. We adopt $\otimes$ to maintain a uniform propagation of confidence across abstraction and application, thus making no qualitative distinction between uses of the same variable with the same type or uses of the same variable with different types.

The choice between $\otimes$ and $\wedge$ is not immaterial. Using $\wedge$, interpreted as minimum, yields the largest confidence compatible with both premises, namely $\min(d_1, d_2)$. In contrast, $\otimes$ may yield a strictly smaller value and therefore captures quantitatively the repeated use of assumptions. Since $d_1 \otimes d_2 \leq d_1 \wedge d_2$, the system based on $\otimes$ is more sensitive to resource usage, while the system based on $\wedge$ is closer to the set-theoretic interpretation of intersection types. In any case the choice of $\wedge$ would make the theory more involved and we should re-work the theory developed in the earlier sections.

- Rule ($\leq$) is parametrised on a type theory (see Definition 2). In this rule we could have allowed for a weakening of the degree of confidence in the conclusion, but this would break the preservation of the degree of confidence under conversion for closed λ -terms.
- As we did for the earlier systems, according to the needs of the argument, when clear from the context, we will consider the index in $:_d$ either as a formula of minimal negation-free Fuzzy Logic, or as its value w.r.t. to a given T-norm and assignment ρ .
- The confidence degree in Rule (U) appears to be the natural one.

► **Remark 25** (On the relation between system $\mathcal{F}$ and $\mathcal{Q}^\cap$). As we have pointed out at the outset of Section 3 we can read a fuzzy assumption, $x : d$, in $\mathcal{F}$, see Definition 4, as the typing assumption $x :_d c$, in Definition 23, for an intersection type theory, $\mathcal{T}$, with only one constant c such that $c \sim_{\mathcal{T}} c \rightarrow c$. Namely a judgement of the shape $x_1 : d_1, \dots, x_n : d_n \vdash_{\mathcal{Q}} M : d$ can be understood as a judgement $x_1 :_{d_1} c, \dots, x_n :_{d_n} c \vdash_{\mathcal{Q}^\cap} M :_d c$.

Hence, in studying systems $\mathcal{F}$, and $\mathcal{Q}$, we have clarified the behaviour of fuzzy formulæ as types *per se*, thereby illustrating the non-contractive nature of Fuzzy Logic, which becomes significant once we go beyond purely affine λ -terms. In studying $\mathcal{Q}^\cap$ we combine the two systems. The exact shape of the fuzzy minimal formula in the typing judgement is not necessarily preserved, but its evaluation in terms of a T-norm and an appropriate assignment provides interesting invariants under evaluation.

We are now in the position of proving the main theorem.

► **Theorem 26.** *Given a left-continuous T-norm and an assignment ρ , if*

$$\begin{aligned} & x_1 :_{d_{1,1}^1} A_1^1, \dots, x_1 :_{d_{1,k_1}^1} A_1^1, \dots, x_1 :_{d_{m_1,1}^1} A_{m_1}^1, \dots, x_1 :_{d_{m_1,k_{m_1}}^1} A_{m_1}^1, \dots \\ & \dots \\ & \dots, x_n :_{d_{1,1}^n} A_1^n, \dots, x_n :_{d_{1,k_1}^n} A_1^n, \dots, x_n :_{d_{m_n,1}^n} A_{m_n}^n, \dots, x_n :_{d_{m_n,k_{m_n}}^n} A_{m_n}^n \vdash_{\mathcal{Q}^\cap} M :_d A, \end{aligned}$$

then

$$\begin{aligned} & \llbracket d_{1,1}^1 \otimes \dots \otimes d_{1,k_1}^1 \Rightarrow \dots \Rightarrow d_{m_1,1}^1 \otimes \dots \otimes d_{m_1,k_{m_1}}^1 \Rightarrow \dots \\ & \dots \\ & \dots d_{1,1}^n \otimes \dots \otimes d_{1,k_1}^n \Rightarrow \dots \Rightarrow d_{m_n,1}^n \otimes \dots \otimes d_{m_n,k_{m_n}}^n \Rightarrow d_{\rho}^T = 1 \end{aligned}$$

and

$$\Gamma^\# \vdash_{\mathcal{T}} M : A$$

in the same intersection type theory.

Proof. (sketch) By induction on the structure of derivations we prove that the systems $\mathcal{Q}^\cap$ and the intersection types assignment system $\mathcal{T}$ progress in parallel. Theorem 15 ensures that the interpretation of the fuzzy formula indexing the membership relation, once fully abstracted, yields the value 1. ◀

We can now also establish the standard suite of proof theoretical results on intersection types assignment systems.

► **Lemma 27** (Inversion Lemma).

1. If $\Gamma \vdash_{\mathcal{Q}^\cap} x :_d A$ and $A \approx \mathbb{U}$, then

$$\Gamma = \{x :_{d_1} A_1, \dots, x :_{d_n} A_n\}$$
 for some $d_1, \dots, d_n, A_1, \dots, A_n$ such that $d = d_1 \otimes \dots \otimes d_n$ and $A_1 \cap \dots \cap A_n \leq A$;
2. If $\Gamma \vdash_{\mathcal{Q}^\cap} MN :_d A$ and $A \approx \mathbb{U}$, then there exist I, B_i, C_i for $i \in I$, such that $\bigcap_{i \in I} C_i \leq A$ and $\Gamma \vdash_{\mathcal{Q}^\cap} M :_{e_i} B_i \rightarrow C_i$ and $\Gamma \vdash_{\mathcal{Q}^\cap} N :_{f_i} B_i$ for all $i \in I$, and d is $\bigotimes_{i \in I} e_i \otimes f_i$;
3. If $\Gamma \vdash_{\mathcal{Q}^\cap} \lambda x.M :_d A$ and x occurs in M , then there exist I and B_i, C_i for $i \in I$ such that $\bigcap_{i \in I} (B_i \rightarrow C_i) \leq A$ and $\Gamma, x :_{e_i} B_i \vdash_{\mathcal{Q}^\cap} M :_{f_i} C_i$ for all $i \in I$, and d is $\bigotimes_{i \in I} (e_i \Rightarrow f_i)$.
4. If $\Gamma \vdash_{\mathcal{Q}^\cap} \lambda x.M :_d A$ and x does not occur in M , then there exist I and B_i, C_i for $i \in I$ such that $\bigcap_{i \in I} (B_i \rightarrow C_i) \leq A$ and $\Gamma \vdash_{\mathcal{Q}^\cap} M :_{f_i} C_i$ for all $i \in I$, and d is $\bigotimes_{i \in I} (1 \Rightarrow f_i)$.

Proof. (1). A variable can only be typed by Axiom (Ax) and hence the judgements thus obtained can only be premises of the Rules ($\cap I$) and ($\leq$). The coefficient of the conclusion of Rule ($\cap I$) is obtained by applying $\otimes$ to the coefficients of the premises. The coefficient of the conclusion of Rule ($\leq$) is the coefficient of the premise.

(2). An application can only be typed by Rule ($\rightarrow E$). We conclude as in Item (1).

(3). An abstraction where the abstracted variable occurs in the body can only be typed by Rule ($\rightarrow I$). We conclude as in Item (1).

(4). An abstraction where the abstracted variable does not occur in the body can only be typed by Rule ($\rightarrow 0$). We conclude as in Item (1). ◀

► **Theorem 28** (Subject Expansion). If $M \rightarrow_\beta M'$ and $\Gamma \vdash_{\mathcal{Q}^\cap} M' :_d A$ imply $\Gamma' \vdash_{\mathcal{Q}^\cap} M :_e A$.

Proof. Paralleling the proof of Subject Expansion [3, Corollary 14.2.5(ii)], taking into account that the context may change, since the system $\mathcal{Q}^\cap$ is quantitative, *i.e.* it accounts for the number of times a variable is effectively used. ◀

Notice that, in the above Theorem 28, the context Γ' may be contained in the context Γ . To see this, consider the analogue of Counterexample 21, where the typing assumptions have as type predicate the constant c , in an intersection type theory, $\mathcal{T}$, with one constant c such that $c \sim_{\mathcal{T}} c \rightarrow c$.

► **Theorem 29** (Subject Reduction). If the background type theory $\mathcal{T}$ satisfies the condition that $\Gamma \vdash_{\mathcal{Q}^\cap} \lambda x.N :_{d'} B \rightarrow C$ implies $\Gamma, x :_f B \vdash_{\mathcal{Q}^\cap} N :_{f'} C$ where $d' \text{ id } f \Rightarrow f'$, then

$$M \rightarrow M' \text{ and } \Gamma \vdash_{\mathcal{Q}^\cap} M :_d A \text{ imply } \Gamma' \vdash_{\mathcal{Q}^\cap} M' :_e A.$$

Proof. Paralleling the proof of Subject Reduction [3, Proposition 14.2.1(ii)], taking into account that the context may change, since the system $\mathcal{Q}^\cap$ is quantitative, *i.e.* it accounts for the number of times a variable is effectively used. ◀

Notice that, in the above Theorem 29, the context Γ' may contain the context Γ . To see this, consider the analogue of Counterexample 22.

In the above Theorems 28 and 29 one could give a sharper relation on the shapes of the formulæ occurring in the indexes d and e . In the following Subsection 4.1 we elaborate on this issue.

We end this subsection with an example which illustrates the need to consider the possibility of having different contexts in judgements involving converted terms.

► **Example 30** (Self-application Revisited).

We have the following derivations

$$\begin{array}{c}
\frac{x :_d (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_d (A \rightarrow B) \cap A \quad x :_e (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_e (A \rightarrow B) \cap A}{x :_d (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_d (A \rightarrow B) \quad x :_e (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_e A} \\
\frac{x :_d (A \rightarrow B) \cap A, x :_e (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} xx :_{d \otimes e} B}{\vdash_{\mathcal{Q}^\cap} \lambda x. xx :_{d \otimes e \Rightarrow d \otimes e} (A \rightarrow B) \cap A \rightarrow B} \\
\frac{\vdash_{\mathcal{Q}^\cap} \lambda x. xx :_{d \otimes e \Rightarrow d \otimes e} (A \rightarrow B) \cap A \rightarrow B \quad x :_d (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_d (A \rightarrow B) \cap A}{x :_d (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} (\lambda x. xx)x :_{d \otimes (d \otimes e \Rightarrow d \otimes e)} B}
\end{array}$$

but

$$\frac{x :_d (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_d (A \rightarrow B) \cap A \quad x :_e (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_e (A \rightarrow B) \cap A}{x :_d (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_d (A \rightarrow B) \quad x :_e (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} x :_e A} \\
\frac{x :_d (A \rightarrow B) \cap A, x :_e (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} xx :_{d \otimes e} B}{x :_d (A \rightarrow B) \cap A, x :_e (A \rightarrow B) \cap A \vdash_{\mathcal{Q}^\cap} xx :_{d \otimes e} B}$$

4.1 What sort of information can we extract from the fuzzy degree of a judgement?

If we evaluate the confidence degree of a provable judgement $\Gamma \vdash_{\mathcal{Q}^\cap} M :_d A$ in $[0, 1]$, this gives us a measure on the membership of the term M to the type A . But viewing the confidence degree d as a formula of minimal Fuzzy Logic, we also have a “skeleton” of the derivation which established the inhabitation of the type A by M . The confidence degree encodes a lot of information indeed. Here are some intriguing questions and conjectures, which we shall not address here.

- To what extent can the derivation be reconstructed from a provable judgement in $\mathcal{Q}^\cap$? We conjecture that the *inhabitation problem* $\vdash_{\mathcal{Q}^\cap} ? :_d A$ should be decidable. This conjecture is supported by the decidability of inhabitation for non-idempotent intersection types, see [5].
- The shape of the fuzzy formula should provide an estimate on the length of the longest β -reduction of M , if M is typed without $\cup$, that is, if it is strongly normalising.

5 Final remarks

In this paper, we have started an investigation of how Fuzzy Logic fares with Constructive Type Theory in view of the growing interest on quantitative systems in constructive Type Theory. We have proposed a number of fuzzy type assignment systems. These permit us to derive some results on Fuzzy Logic *per se*, through a fuzzy analogue of the formulæ-as-types paradigm, as well as provide a “fuzzification” of intersection type theories for pure λ -calculus. More work needs to be done to answer the open questions left in Section 4.1, as well as to assess the expressive benefits of using $\wedge$ in place of $\otimes$ in Rule $(\cap I)$, this preventing us to treat on a par multiple uses of the same assumption with different types.

We also plan to explore in the future λ -calculi à la Church, such as Constructive Type Theories, where judgements have the shape $\Gamma \vdash_{\mathcal{Q}} M :_d A$, but also have equality judgements, $\Gamma \vdash_{\mathcal{Q}} M_1 = M_2 :_d A$.

A related line of research, which we intend to explore, is how to derive a suitable fuzzy intersection type assignment system starting from a notion of *fuzzy similarity* between types. The natural understanding of fuzzy equality-types in Constructive Type Theory,

given above, namely $\Gamma \vdash_Q M_1 =_d M_2 : A$, is a similarity relation, see Definition 36 in the Appendix. More precisely, given basic sets D_i 's, assume that a *similarity relation*, S_{D_i} , is given for each. Is there a natural way to extend the similarity to higher types, thereby deriving a fuzzy simple type-assignment system? Moreover, can we take $(D, \cdot)$ to be an applicative structure, and use as similarity the inverse of Hausdorff distance, *i.e.* $S_{\mathcal{P}(D)}(A, B) = \min\{\inf_{x \in A} \sup_{y \in B} S_D(x, y), \inf_{x \in B} \sup_{y \in A} S_D(x, y)\}$, to inductively define similarity between logical relations within D ? Appropriate conditions on the applicative behaviour of the points in D need to be enforced, however. This could provide a reading of the results in [11] on λ -quantitative theories, in terms of similarities, rather than of metrics. Yet another line of research is the use of fuzzy sets in some models of untyped λ -calculus such as Engeler's model [9].

Related Work. Some of our results can also be derived from the results on **BCK**-algebras in the theory of Fuzzy Propositional Logics, see *e.g.* [14], but our proofs follow a different route. Vilém Novák and his group have worked considerably on Fuzzy Type theory, but have mainly taken a classical approach [13]. Castelnovo and Miculan [6] consider first order fuzzy theories in general, whereas we focus on higher order structures. A few authors have considered Fuzzy Type Theory specifically. S. Arya, G. Coraglia, P. North, S. O'Connor, H. Riess, A. Tenório have done some preliminary work on Fuzzy Type Theory in the context of 2-categories (see <https://etagreta.github.io/>).

Since, as we already pointed out, Fuzzy Logic has a non-idempotent flavour, our work relates to non-idempotent intersection type theories [10, 7, 5, 4]. The information stored in the type in [5] is included in the information stored in the fuzzy formula that indexes the typing judgement in our system $\mathcal{Q}^\cap$. More work needs to be done to compare the two approaches in depth.

We hope that the present paper will stimulate future discussions on the intriguing combination of Fuzzy Logic with Constructive Type Theory, as well as with other formal systems for reasoning about concurrent processes.

References

- 1 Henk Barendregt. *The Lambda Calculus - its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. URL: https://api.pageplace.de/preview/DT0400.9780080933757_A23543814/preview-9780080933757_A23543814.pdf.
- 2 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983. doi:10.2307/2273659.
- 3 Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2013. doi:10.1017/CB09781139032636.
- 4 Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Inhabitation for non-idempotent intersection types. *Logical Methods in Computer Science*, 14(3), 2018. doi:10.23638/LMCS-14(3:7)2018.
- 5 Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017. doi:10.1093/JIGPAL/JZX018.
- 6 Davide Castelnovo and Marino Miculan. Fuzzy algebraic theories. In Florin Manea and Alex Simpson, editors, *CSL*, volume 216 of *LIPICs*, pages 13:1–13:17. Schloss Dagstuhl, 2022. doi:10.4230/LIPICs.CSL.2022.13.

- 7 Daniel de Carvalho. Execution time of λ -terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018. doi:10.1017/S0960129516000396.
- 8 Mariangiola Dezani-Ciancaglini, Besik Dundua, Paola Giannini, and Furio Honsell. Sensible intersection type theories. In Thorsten Altenkirch, Franco Barbanera, Ferruccio Damiani, and Ugo de'Liguoro, editors, *Logics and Type Theory: essays dedicated to Stefano Berardi on the occasion of his 100000th birthday*, volume 441 of *EPTCS*, pages 126–147, 2026. doi:10.4204/EPTCS.441.8.
- 9 Erwin Engeler. Algebras and combinators. *Algebra Universalis*, 13:389–392, 1981. doi:10.1007/BF02483849.
- 10 Philippa Gardner. Discovering needed reductions using type theory. In Masami Hagiya and John C. Mitchell, editors, *TACS*, volume 789 of *LNCS*, pages 555–574. Springer, 1994. doi:10.1007/3-540-57887-0_115.
- 11 Ugo Dal Lago, Furio Honsell, Marina Lenisa, and Paolo Pistone. On quantitative algebraic higher-order theories. In Amy P. Feltz, editor, *FSCD*, volume 228 of *LIPICs*, pages 4:1–4:18. Schloss Dagstuhl, 2022. doi:10.4230/LIPICs.FSCD.2022.4.
- 12 George Metcalfe. Fundamentals of fuzzy logic, 2005. URL: <https://www.logic.at/tbilisi05/Metcalfe-notes.pdf>.
- 13 Vilém Novák. On fuzzy type theory. *Fuzzy Sets Syst.*, 149(2):235–273, 2005. doi:10.1016/J.FSS.2004.03.027.
- 14 Rostislav Horčík. Petr Cintula, Petr Hájek. Formal systems of fuzzy logic and their fragments. *Annals of Pure and Applied Logic*, 150:49–72, 2007. doi:10.1016/j.apal.2007.07.004.
- 15 Sorites paradox. URL: <https://plato.stanford.edu/entries/sorites-paradox/>.
- 16 Lotfi A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, June 1965. doi:10.1016/S0019-9958(65)90241-X.

A

 Notions in Fuzzy logic

► **Definition 31.** A fuzzy set is a pair (S, μ_S) , where S is a set and $\mu_S : S \rightarrow [0, 1]$ is a function. For all $x \in S$, $\mu_S(x)$ is called the degree of membership of x . We say that:

- x is fully included in (S, μ_S) , if $\mu_S = 1$;
- x is not included in (S, μ_S) , if $\mu_S(x) = 0$;
- x is partially included in (S, μ_S) , if $0 < \mu_S(x) < 1$.

If all $x \in S$ are fully included in S , the fuzzy set S is said to be crisp.

► **Definition 32 (T-norm).** A T-norm, see [12], is a binary function $T : [0, 1] \times [0, 1] \rightarrow [0, 1]$ that satisfies the following properties:

- commutativity: $T(a, b) = T(b, a)$,
- associativity: $T(a, T(b, c)) = T(T(a, b), c)$,
- monotonicity: $T(a, b) \leq T(c, d)$ if $a \leq c$ and $b \leq d$,
- the number 1 is the identity element: $T(a, 1) = a$.

Usually, T-norms are written in infix notation and the symbol $\otimes$ is used to denote them. In the paper, we consider only left-continuous t-norms.

► **Definition 33 (Residuum).** A binary function $\otimes : [0, 1]^2 \rightarrow [0, 1]$ is said to be residuate if and only if there exists a binary function $\Rightarrow : [0, 1]^2 \rightarrow [0, 1]$ called the residuum of $\otimes$, such that $\otimes$ and $\Rightarrow$ form an adjoint pair, i.e.:

$$\forall a, b, c \in [0, 1]. a \otimes b \leq c \Leftrightarrow a \leq b \Rightarrow c.$$

► **Definition 34.** *Prominent examples of left-continuous T-norms are:*

- Gödel's T-norm: $a \otimes b = \min(a, b)$, whose residuum is

$$a \Rightarrow b = \begin{cases} b & \text{if } b \leq a \\ 1 & \text{otherwise;} \end{cases}$$

- Goguen's product T-norm: $a \otimes b = a * b$, whose residuum is

$$a \Rightarrow b = \begin{cases} b/a & \text{if } b \leq a \\ 1 & \text{otherwise;} \end{cases}$$

- Łukasiewicz's T-norm: $a \otimes b = \max(0, a + b - 1)$, whose residuum is

$$a \Rightarrow b = \begin{cases} 1 - a + b & \text{if } b \leq a \\ 1 & \text{otherwise.} \end{cases}$$

► **Proposition 35 (Norms with Residuum).** *Given a left-continuous T-norm $\otimes$, we have the following standard properties, where $a, b, c \dots \in [0, 1]$:*

1. $\otimes$ uniquely determines a residuum defined as follows

$$a \Rightarrow b := \sup\{c \mid a \otimes c \leq b\};$$

2. $\Rightarrow$ is monotone in the second argument and anti-monotone in its first argument;
3. $a \Rightarrow a = a \Rightarrow 1 = 1$, $1 \Rightarrow a = a$;
4. $a \leq b \Rightarrow a$;
5. $a \leq b$ if and only if $a \Rightarrow b = 1$;
6. $a \otimes b \leq a$, and also $a \otimes b \leq \min\{a, b\}$;
7. $(a \otimes b) \Rightarrow c = a \Rightarrow (b \Rightarrow c)$;
8. $(a \otimes (a \Rightarrow b)) \leq b$, and hence $(a \otimes (a \Rightarrow b)) \leq \min\{a, b\}$.

Proof.

- Property (1) follows by the adjunction in Definition 33.
- Properties (2)-(5) follow from previous items and the Definition 33. We illustrate (5). From the adjunction we have that $1 \leq a \Rightarrow b \iff 1 \otimes a \leq b$, but $a = 1 \otimes a$ by (2).
- Property (6) follows from monotonicity of the T-norm and identity.
- Property (7) follows from the adjunction of the T-norm.
- Property (8) follows from the fact that, by adjunction we have $a \otimes (a \Rightarrow b) \leq b \iff (a \Rightarrow b) \leq (a \Rightarrow b)$, and hence $a \otimes (a \Rightarrow b) \leq b$ always holds. ◀

► **Definition 36 (Similarity Relation).** *A similarity relation, S , over a domain D , is a fuzzy subset of $D \times D$, which is reflexive, symmetric, and transitive.*

Namely let $x, y, z \in D$ and let $\mu_S(x, y)$ denote the grade of membership of the ordered pair $(x, y) \in D \times D$. Then S is a similarity relation in D if and only if, for all $x, y, z \in D$,

- $\mu_S(x, x) = 1$ (reflexivity),
- $\mu_S(x, y) = 1$ implies $\mu_S(y, x) = 1$ (symmetry),
- $\mu_S(x, z) \geq \mu_S(x, y) \otimes \mu_S(y, z)$ (transitivity).

The Rezk Completion for Elementary Topoi

Kobe Wullaert   

Delft University of Technology, The Netherlands

Niels van der Weide   

Radboud University, Nijmegen, The Netherlands

Abstract

The development of category theory in univalent foundations and the formalization thereof is an active field of research. Categories in that setting are often assumed to be univalent which means that identities and isomorphisms of objects coincide. One consequence hereof is that equivalences and identities coincide for univalent categories and that structure on univalent categories transfers along equivalences. However, constructions such as the Kleisli category, the Karoubi envelope, and the tripos-to-topos construction, do not necessarily give univalent categories. To deal with that problem, one uses the Rezk completion, which completes a category into a univalent one. However, to use the Rezk completion when considering categories with structure, one also needs to show that the Rezk completion inherits the structure from the original category.

In this work, we present a modular framework for lifting the Rezk completion from categories to categories with structure. We demonstrate the modularity of our framework by lifting the Rezk completion from categories to elementary topoi in manageable steps.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification

Keywords and phrases univalent foundations, univalent categories, Rezk completions, UniMath, formalization, elementary topoi

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.8

Supplementary Material *Software (Source code)*: <https://github.com/UniMath/UniMath/>
archived at `swh:1:dir:a6f8ec337e875b0db0f2f46b97a577a1d7836871`

Funding *Niels van der Weide*: This research was supported by the NWO project “The Power of Equality” OCENW.M20.380, which is financed by the Dutch Research Council (NWO).

Acknowledgements We thank Benedikt Ahrens for valuable feedback on the versions of this paper and for the many helpful discussions. Furthermore, we thank Arnoud van der Leer for reviewing the formalization and for refactoring the material on cartesian closed categories. We gratefully acknowledge the work by the Rocq development team in providing the Rocq proof assistant and surrounding infrastructure, as well as their support in keeping UniMath compatible with Rocq.

1 Introduction

Univalent foundations [23] has proven itself to be an adequate foundation for the development of mathematics and category theory in particular. In univalent foundations, structures adhere to the principle that equality and equivalence of structures coincide, such that equivalent structures are indistinguishable. This principle is known as the univalence principle [5].

For structured sets, such as monoids or groups, posets, or setoids, this principle is usually a consequence of the univalence axiom, which postulates that principle for types. That is, both equivalence (i.e., isomorphism) and equality between such structured sets coincide with equivalence and equality, respectively, on the underlying sets. However, for higher-dimensional structures, such as categories, the story is more subtle [2, 23]. For categories to adhere to this principle, we need an equivalence between the identity type ($\mathcal{A} = \mathcal{C}$) and the type ($\mathcal{A} \simeq \mathcal{C}$) of *adjoint equivalences*. The univalence axiom implies that we have an equivalence



© Kobe Wullaert and Niels van der Weide;

licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 8; pp. 8:1–8:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

between $(\mathcal{A} = \mathcal{C})$ and the type of *isomorphisms* $(\mathcal{A} \cong \mathcal{C})$. Hence, the univalence principle for categories can be restated as having an equivalence $(\mathcal{A} \cong \mathcal{C}) \simeq^{(1)} (\mathcal{A} \simeq \mathcal{C})$. However, (1) does not hold in general, unless when we consider *univalent categories*: categories for which identity and isomorphism for objects are equivalent.

The univalence condition for categories is met by many naturally occurring categories in UF. This condition can furthermore be motivated by looking at the set-theoretic models of UF, such as Voevodsky’s simplicial set model of homotopy type theory [14]. In this model, types are interpreted as simplicial sets, and so the interpretation of categories (in HoTT) gives rise to a bisimplicial set which is a Segal space. Specifically, one simplicial set interprets the objects (and the internal homotopies or identities), and the other interprets the morphisms, and both are connected by source, target, and identity maps of simplicial sets. Under this interpretation, the univalence condition corresponds precisely to the completeness condition for Segal spaces. Observe, however, that we do not obtain all Segal spaces, due to the set condition on the hom-types in HoTT. Hence, (univalent) categories in HoTT correspond to truncated or 1-categorical (complete) Segal spaces, whose higher homotopies on the morphisms are trivial.

Even though univalence for categories is, by the aforementioned reasons, a “natural” condition, different constructions do not produce univalent categories; see Section 2. This is where the *Rezk completion* comes into play.

Abstractly, the Rezk completion refers to the univalent completion of a category [2]. That is, the Rezk completion of a category $\mathcal{C}$ is a univalent category $\mathrm{RC}(\mathcal{C})$, together with a functor $\mathcal{C} \xrightarrow{\eta_{\mathcal{C}}} \mathrm{RC}(\mathcal{C})$, which satisfies the following universal property. For every univalent category $\mathcal{E}$, any functor of type $\mathcal{C} \rightarrow \mathcal{E}$ factors uniquely through $\eta_{\mathcal{C}}$. Ahrens et al. [2] proved that this universal property is equivalent to the requirement that $\eta_{\mathcal{C}}$ is a *weak* equivalence. However, the weakness of this equivalence means that $\eta_{\mathcal{C}}$ need not have an inverse. In particular, structure on $\mathcal{C}$ is not guaranteed to transport across $\eta_{\mathcal{C}}$ to a structure on $\mathrm{RC}(\mathcal{C})$.

In this work, we present a modular approach to lift the Rezk completion from categories to structured categories, such as elementary topoi. In particular, we show that the Rezk completion of a topos inherits the structure of a topos and that the universal property extends to the level of topoi. Furthermore, this work is accompanied by a formalization in the Rocq-library UniMath, and our approach is designed with this formalization in mind. In particular, our approach is modular and is designed to be applicable to a lot of structures a category can have. Furthermore, we expect our approach to be practical for the formalization hereof in any proof assistant based on homotopy type theory.

1.1 Prior Work

As for (1-)categories, higher-dimensional categories are supposed to satisfy an additional univalence condition such that they adhere to the univalence principle. A general theory making this precise has been developed by Ahrens et al. [5] where a signature for finite-dimensional categorical structures has been developed and a notion of equivalence has been formulated for such structures, which for e.g. categories coincides with the notion of equivalence.

Different categorical theories have been developed in univalent foundations, including the theory of monoidal categories [33], enriched categories [25], bicategories [1], and double (bi)categories [28, 20]. In particular, those works formulated the notion of univalence for such categorical structures, which coincide with the one given by Ahrens et al. [5].

The Rezk completion [2] has been lifted to multiple categorical structures, including some of the mentioned above [1, 33, 25]. Furthermore, the Rezk completion has been constructed for a class of structured groupoids [30].

Moreover, univalent category theory, and the Rezk completion in particular, has been used in the study of denotational semantics. For instance, the Rezk completion is used to construct from a category with representable maps of presheaves a univalent category with families [4]. Furthermore, the Rezk completion is used to relate two different constructions of Scott’s representation theorem, which model the untyped λ -calculus [24]. Lastly, various classes of topoi and their internal logic, based on comprehension categories, are constructed in the univalent setting by Van der Weide [27].

1.2 Overview and Contributions

This paper is structured as follows:

1. Section 2 contains background material on univalent categories and the Rezk completion.
2. In Section 3, we discuss the framework and strategy for lifting the Rezk completion from categories to structured categories. We start by briefly recalling some of the basics of displayed bicategories. Then we define the notion of displayed universal arrows as an intermediate step for the modular construction of biadjoints. Afterwards, we discuss how such arrows specialize in the case of lifting the Rezk completion.
3. In Section 4, we apply the framework presented in Section 3 to lift the Rezk completion from categories to various classes of structured categories where we work our way up to W -topoi. More precisely, we consider finitely (co)complete categories (Section 4.1), cartesian closed categories (Section 4.2), elementary topoi (Section 4.3), and W -topoi (Section 4.4).
4. In Sections 5 and 6, we elaborate on the different methods to extend the Rezk completion to categories with structure found in the literature and possible ways in which this work can be extended.

1.3 Formalization and Foundations

This work is accompanied by formalization in the Rocq-library **UniMath** on univalent mathematics [22, 31]. The logic underlying the **UniMath** library is an intensional dependent type theory whose universes satisfy the univalence axiom. We do not rely on the axiom of choice, except in Lemma 39, nor the law of excluded middle. We rely on propositional truncation.

The formalization, and this work, makes heavily use of the already existing material on (bi)category theory in the **UniMath**-library and the accompanied literature [2, 1]. The environments annotated with a  denote that the result is formalized. Upon clicking , the reader is directed towards the HTML documentation. The commit from which the documentation is generated is <https://github.com/UniMath/UniMath/commit/94b49c3>.

2 On the Rezk Completion for Categories

In this section, we recall and elaborate on the Rezk completion for categories. Most of the material in this section is already found in the original paper on univalent categories [2]. First, we recall the definition of univalent categories in Definition 2. Then we recall weak equivalences (Definition 3) and we recall the Rezk completion in terms of weak equivalences (Definition 5). Afterwards, we zoom in on some examples of categorical constructions where the Rezk completion plays a role. Lastly, we discuss a bicategorical formulation of the Rezk completion, which we use in Sections 3 and 4 to lift the Rezk completion from categories to structured categories.

Univalent Categories. In univalent foundations, **categories** $\mathcal{C}$ consist in particular of a type of objects $\mathbf{ob}(\mathcal{C})$ and for every pair of objects $x, y : \mathbf{ob}(\mathcal{C})$ a type of morphisms $\mathcal{C}(x, y)$. Furthermore, a category $\mathcal{C}$ consists of a composition operation on the morphisms $(- \cdot -) : \mathcal{C}(x, y) \rightarrow \mathcal{C}(y, z) \rightarrow \mathcal{C}(x, z)$, identity morphisms $\mathrm{id}_x : \mathcal{C}(x, x)$, and the composition is associative and unital. Lastly, each $\mathcal{C}(x, y)$ is a set, which means that equalities between morphisms form a proposition; recall that a type A is a proposition if each identity type $x = y$ is equivalent to the unit type, for every $x, y : A$.

► **Remark 1.** Composition of morphisms is written in diagrammatic order: the composite of $f : \mathcal{C}(x, y)$ and $g : \mathcal{C}(y, z)$ is denoted by $f \cdot g$.

However, as mentioned in the introduction, such categories do not adhere to the principle that equality and equivalence coincide. That is, the univalence axiom implies that equality and isomorphism for categories coincide [2, Lemma 6.16], but equality and isomorphism do not coincide with equivalence for categories. Those categories for which equivalences do coincide with the aforementioned are precisely the **univalent categories**. These are categories for which identities and isomorphisms between two objects coincide. More precisely, let $\mathcal{C}$ be a category and denote by $(x \cong y)$ the type of isomorphisms from x to y , for $x, y : \mathbf{ob}(\mathcal{C})$. By path induction, we have a function $\mathrm{idtoiso}_{x,y} : (x = y) \rightarrow (x \cong y)$.

► **Definition 2** (🔗 Dfn 3.6. [2]). A category $\mathcal{C}$ is **univalent** if $\mathrm{idtoiso}_{x,y}$ is an equivalence of types, for every $x, y : \mathbf{ob}(\mathcal{C})$.

By the univalence axiom, many categories, such as those of sets, groups, etc., are univalent.

Observe that univalence implies that $\mathbf{ob}(\mathcal{C})$ is a groupoid, i.e., the types $x = y$ are sets.

The Rezk completion. The Rezk completion refers to the univalent completion of a category [2, 23]. That is, if $\mathcal{C}$ is a category, then its Rezk completion is a univalent category $\hat{\mathcal{C}}$ together with a functor $\eta : \mathcal{C} \rightarrow \hat{\mathcal{C}}$ which is initial among the functors from $\mathcal{C}$ to a univalent category, i.e., for every $\mathcal{E}$ univalent, the precomposition functor $(\eta \cdot -) : [\hat{\mathcal{C}}, \mathcal{E}] \rightarrow [\mathcal{C}, \mathcal{E}]$ is an isomorphism of categories, where $[-, -]$ denotes the functor category. In particular, this means that every functor into a univalent category factors uniquely along through η :

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{\eta} & \hat{\mathcal{C}} \\ & \searrow \forall F & \swarrow \exists! H \\ & \mathcal{E} & \end{array}$$

We start by recalling weak equivalences, which allow us to describe the universal property of the Rezk completion.

► **Definition 3** (🔗 Dfn 6.7. [2]). Let $\mathcal{C}, \mathcal{E}$ be categories. A **weak equivalence** is a functor $F : \mathcal{C} \rightarrow \mathcal{E}$ which is fully faithful and is **essentially surjective** (on objects); where **essential surjectivity** (🔗) means that for every object $y : \mathcal{E}$ in the codomain, there merely exists an object $x : \mathcal{C}$ and an isomorphism of type $F x \cong y$.

Observe that essential surjectivity does not imply that for every $y : \mathcal{E}$ one can choose an object $x : \mathcal{C}$ in the preimage, due to the truncation in the mere existence. In particular, there need not to be a function $F^{-1} : \mathbf{ob}(\mathcal{E}) \rightarrow \mathbf{ob}(\mathcal{C})$ such that $F(F^{-1}(y)) \cong y$. Those functors $F : \mathcal{C} \rightarrow \mathcal{E}$ for which for every $y : \mathbf{ob}(\mathcal{E})$ we can choose an object $x : \mathbf{ob}(\mathcal{C})$ such that $F x \cong y$ are said to be **split surjective** (🔗). Furthermore, being split surjective and fully faithful is equivalent to F having a right adjoint whose unit and counit are isomorphisms. Hence, such F 's are called **(adjoint) equivalences**. However, as opposed to weak equivalences, being

an equivalence is generally not a proposition. In particular, a weak equivalence is not an equivalence. In fact, between categories whose type of objects are sets (setcategories), the statement that weak equivalences are split surjective is equivalent to the axiom of choice ([23, Page 309]). However, between univalent categories, weak equivalences, equivalences, and even isomorphisms do coincide ([2, Dfn. 6.9, Lemma 6.10]).

► **Proposition 4** (🔴 Thm 8.4. [2]). *Let $G : \mathcal{C}_0 \rightarrow \mathcal{C}_1$ be a weak equivalence between not-necessarily univalent categories. Then for every univalent category $\mathcal{C}_2$, the precomposition functor $(G \cdot -) : [\mathcal{C}_1, \mathcal{C}_2] \rightarrow [\mathcal{C}_0, \mathcal{C}_2]$ is an equivalence of univalent categories.*

Univalence can be characterized using weak equivalences [2, Thm. 8.9].

The Rezk completion [2] refers to a construction of the univalent completion (see Remark 6). Nevertheless, we only rely on the description of the Rezk completion in terms of weak equivalences. Hence, we will refer to *the Rezk completion* as in the following definition:

► **Definition 5.** *Let $\mathcal{C}$ be a category. A **Rezk completion** for $\mathcal{C}$ consists of a univalent category $\hat{\mathcal{C}}$ and a weak equivalence $\eta : \mathcal{C} \rightarrow \hat{\mathcal{C}}$.*

That a Rezk completion indeed satisfies the universal property of the univalent completion is precisely Proposition 4. Furthermore, for every category, its type of Rezk completions is a proposition. In particular, Rezk completions are unique up to identity, and hence, we can say *the* Rezk completion. Given a category $\mathcal{C}$, we denote by the $\text{RC}(\mathcal{C}) := \hat{\mathcal{C}}$ the univalent category and $\eta_{\mathcal{C}} : \mathcal{C} \rightarrow \text{RC}(\mathcal{C})$ the weak equivalence.

► **Remark 6.** The Rezk completion has two constructions [2, 23]. Ahrens et al. [2] constructed the Rezk completion of a category $\mathcal{C}$ as a full subcategory of the category of presheaves $[\mathcal{C}^{\text{op}}, \text{Set}]$ given by those functors which are merely isomorphic to a representable functor. However, this construction raises the universe level since $[\mathcal{C}^{\text{op}}, \text{Set}]$ has a higher universe level than $\mathcal{C}$. More precisely, recall that there is a hierarchy of universes, let $\mathcal{U}_k$ be the universe at level k , and denote by $\text{Cat}^{(i,j)}$ the type of categories whose type of objects is in $\mathcal{U}^{(i)}$ and where every $\mathcal{C}(x, y)$ is in $\mathcal{U}_j$. Whenever we have $\mathcal{C} : \text{Cat}^{(i,j)}$, we have $\text{RC}(\mathcal{C}) : \text{Cat}^{(i \vee (j+1), i \vee j)}$, where $\vee$ is the least upper bound of universe levels.

We can also consider a second construction, assuming the existence of higher inductive types in the same universe. Here, we can define the type of objects of $\text{RC}(\mathcal{C})$ as a higher inductive type which contains, in particular, a term constructor $i : \text{ob}(\mathcal{C}) \rightarrow \text{ob}(\text{RC}(\mathcal{C}))$ and a path constructor $\prod_{x,y:\mathcal{C}} (x \cong y) \rightarrow (i x = i y)$. For a full definition of this construction, we refer the reader to [23, p. 335-337]. This construction has the advantage that the Rezk completion remains in the same universe.

This work is independent of the precise construction of the Rezk completion, and solely relies on the fact that the Rezk completion can be characterized as a weak equivalence into a univalent category. Hence, we refer the reader to the aforementioned literature [2, 23] for more details on these constructions.

Examples. We now discuss some examples and constructions of categories to illustrate the following aspects concerning the Rezk completion. Example 7 illustrates that the Rezk completion generalizes certain constructions on set-theoretic structures such as groups and posets. Examples 8 and 9 illustrate that working in univalent foundations distinguishes constructions which are classically equivalent. Example 10 is the motivating example of this work, where we have a construction of topoi which usually does not produce a univalent topos. Hence, to construct an alternative univalent construction thereof, the Rezk completion is essential. In particular, Example 10 illustrates that we need sufficient results about the transfer of structure from a category to its Rezk completion. Of course, this observation also plays a role in the other examples.

► **Example 7 (Setcategories).** A setcategory is a category whose type of objects is a set. Often, such categories are not univalent.

1. The walking iso category, i.e., the category with two distinct but isomorphic objects, is not univalent. The Rezk completion hereof is the terminal category. More generally, we can consider a set with a preorder, which is equivalently a setcategory whose hom-sets are propositions. Such a category is univalent if and only if the preorder is anti-symmetric [2, Exa. 3.14] Hence, the Rezk completion of a preorder is its posetal completion.
2. A monoid M is equivalently a one-object setcategory BM . If M contains more than one invertible element, this category is not univalent. If M is an abelian group, the Rezk completion of BM is the Eilenberg-MacLane space of M [16, 7], which in univalent foundations can be constructed via higher inductive types. In particular, whereas the Rezk completion for preordered sets can be understood as picking out a representative of every equivalence class, this example illustrates that non-trivial identities are added. Furthermore, $\eta : BM \rightarrow \text{RC}(BM)$ is an example of a weak equivalence that does not have an inverse.
3. The Rezk completion allows for the construction of quotients of sets. Indeed, let A be a set equipped with an equivalence relation $\simeq$ (i.e., a setoid). Then A can be made into a category whose objects are the elements in A and a morphism of type $a \rightarrow b$ witnesses $a \sim b$. This makes A into a groupoid. The Rezk completion for A is then a groupoid whose objects are equivalence classes of $\sim$. More generally, the Rezk completion can be seen as a generalization of the groupoid quotient [30].
4. Moreover, syntactic categories are often non-univalent setcategories. For example, the category associated to a Lawvere theory is a setcategory whose objects are natural numbers, and a morphism from n to m is a term with n inputs and m outputs definable in the theory. This category is generally not univalent. For example, in the theory of groups, the *swap* morphism $(x_0, x_1) \mapsto (x_1, x_0)$ is a non-trivial isomorphism from 2 to itself. Similarly, the cartesian closed category associated to the simply-typed λ -calculus is a non-univalent setcategory.

► **Example 8 (Kleisli category).** Monads are an important tool in functional programming as they provide the right abstraction to model programs with effects. Intuitively, given a category $\mathcal{C}$ of datatypes and programs, and a monad T which specifies an effect, its Kleisli category contains the same objects (datatypes), but the morphisms are programs where the effect is integrated.

The Kleisli category of T can be constructed via Kleisli morphisms or via free algebras. While the latter construction produces a univalent category, if $\mathcal{C}$ is univalent, the former does not [26, 5]. However, the Rezk completion of the Kleisli morphism construction is precisely the free algebra construction.

► **Example 9 (idempotent completion).** The (split-)idempotent completion is a fundamental construction in many areas. For instance, in category theory, this completion can be used to study equivalences between presheaf categories [6, Thm 6.5.11]. Furthermore, this construction generalizes the Cauchy completion of metric spaces [15]. Moreover, the idempotent completion can be used to construct models of the untyped lambda-calculus [12]. We discuss two constructions of the idempotent completion: the *Cauchy completion* and the *Karoubi envelope*.

To construct the Karoubi envelope of a category $\mathcal{C}$, we use the notion of split idempotents. A morphism $f : \mathcal{C}(x, x)$ is *idempotent* if $f \cdot f = f$, and an idempotent morphism f is *split* if f can be factorized as a composite $x \xrightarrow{\pi} y \xrightarrow{\iota} x$, for some $y : \mathcal{C}$, and such that $\text{id}_y = \iota \cdot \pi$. The objects of the Karoubi envelope are the idempotent morphisms in $\mathcal{C}$. In general, the *Karoubi envelope* is not univalent.

The Cauchy completion is constructed as a full subcategory of the category of presheaves. The Cauchy completion is always univalent, and is precisely the Rezk completion of the Karoubi envelope [24].

► **Example 10** (tripos-to-topos). The tripos-to-topos construction is fundamental in topos theory [19, 13]. One can use it to construct realizability topoi [29] and to give an alternative description of categories of sheaves on a complete Heyting algebra [4]. Intuitively, both triposes and topoi represent models of higher-order intuitionistic logic, but there is a key difference. While one represents formulas as morphisms into the subobject classifier in a topos, triposes offer more flexibility. More specifically, in a tripos we have a functor $\mathbb{P} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{HA}$ that assigns to each set a Heyting algebra of formulas. In essence, the notion of formula in a topos $\mathcal{E}$ is internal to $\mathcal{E}$, but this is not so in a tripos. The tripos-to-topos construction constructs from every tripos $\mathbb{P} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{HA}$ a corresponding topos $\mathbf{PER}(\mathbb{P})$.

In general, the tripos-to-topos construction does not give a univalent category. A counter example for such a non-univalent category is the following. Every complete Heyting algebra $\mathcal{H}$ gives rise to a tripos such that the formulas on a set X are the same as functions $X \rightarrow \mathcal{H}$. If we apply the tripos-to-topos construction, then we get the category of $\mathcal{H}$ -valued sets [11]. If $\mathcal{H}$ is the terminal Heyting algebra, we obtain the category whose objects are sets but where each hom-set is the unit type – which is clearly not univalent.

► **Example 11** (functor categories). Let $\mathcal{A}$ and $\mathcal{C}$ be categories. Univalence is preserved under taking functor categories. That is, if $\mathcal{C}$ is univalent, then the category $[\mathcal{A}, \mathcal{C}]$ of functors is also univalent [2, Thm. 4.15]. However, it is not provable that the Rezk completion of a functor category $[\mathcal{A}, \mathcal{C}]$ is (equivalent to) the functor category $[\mathcal{A}, \mathbf{RC}(\mathcal{C})]$ for every $\mathcal{A}$ and $\mathcal{C}$. That this is not provable follows from choosing $\mathcal{A}$ to be a discrete category on a set X , and $\mathcal{C}$ the 1-object category associated to the group $\mathbb{Z}/2$. Then, from an equivalence between $[\mathcal{A}, \mathbf{RC}(\mathcal{C})]$ and $\mathbf{RC}([\mathcal{A}, \mathcal{C}])$ we can derive the statement:

$$\prod_{X : \mathbf{Set}} \prod_{\mathbb{P} : X \rightarrow \mathbf{Set}} \left(\prod_{x : X} \|\mathbb{P}(x) \simeq \mathbf{2}\| \right) \rightarrow \left\| \prod_{x : X} \mathbb{P}(x) \right\|.$$

This statement, however, is a slight generalization of the World's Simplest Axiom of Choice [8] which can fail in a sheaf topos [9]. It remains an open problem whether the statement $\prod_{[\mathcal{A}, \mathcal{C}]} \mathbf{RC}([\mathcal{A}, \mathcal{C}]) \simeq [\mathcal{A}, \mathbf{RC}(\mathcal{C})]$ is consistent, and is hence independent from UF.

A bicategorical perspective. To lift the Rezk completion from categories to structured categories, we formulate the Rezk completion bicategorically.

Analogous to the way that structured sets, e.g., monoids, form the objects in their respective categories, categories with additional structure form the objects in a bicategory [1]. One advantage of working bicategorically is that we can treat structure uniformly. Furthermore, the universal property of the Rezk completion can be internalized in the bicategory $\mathbf{Cat}$ of categories, functors, and natural transformations, since the Rezk completion makes the bicategory $\mathbf{Cat}_{\text{univ}}$ of univalent categories into a reflective subcategory of $\mathbf{Cat}$. That is, the inclusion of $\mathbf{Cat}_{\text{univ}}$ into $\mathbf{Cat}$ has a left biadjoint (Proposition 13).

► **Definition 12** (👉). Let $R : \mathcal{B}_1 \rightarrow \mathcal{B}_2$ be a pseudofunctor. A **left biadjoint** L for R is given by:

1. a function $L : \mathcal{B}_2 \rightarrow \mathcal{B}_1$;
 2. a family of morphisms $\eta : \prod_{X : \mathcal{B}_2} X \rightarrow R(L(X))$ called **the unit**;
- such that for every $X : \mathcal{B}_2$ and $Y : \mathcal{B}_1$, the functor

$$\eta_X \cdot R(-) : \mathcal{B}_1(L X, Y) \rightarrow \mathcal{B}_2(X, R Y),$$

is an adjoint equivalence of categories.

We write $L \dashv R$ to denote that L is a left biadjoint of R .

Analogous to the way that adjunctions have a characterization in terms of the unit and counit transformations, the same holds for biadjoints. In particular, given a left biadjoint L for R , L can be made into a pseudofunctor of type $\mathcal{B}_2 \rightarrow \mathcal{B}_1$. Indeed, the function L specifies the action of objects and the action on morphisms (resp. 2-cells) is induced by the inverse of $\eta_- \cdot R$. Furthermore, the unit η becomes a pseudonatural transformation from $\text{id}_{\mathcal{B}_2}$ to $L \cdot R$. The counit is pointwise given by $(\eta_{RY} \cdot R)^{-1}(\text{id}_{RY})$, for $Y : \mathcal{B}_1$. Observe however that the triangle and pentagon identities hold up to a modification. In particular, we also obtain a pseudomonad, whose multiplication can be derived from the counit.

In the formalization, we've axiomatized the existence of the following left biadjoint. Nevertheless, a proof of that proposition, relying on the construction via higher inductive types, is formalized at [🔗](#).

► **Proposition 13.** *The inclusion of Cat_{univ} into Cat has a left biadjoint $\text{RC} : \text{Cat} \rightarrow \text{Cat}_{\text{univ}}$. In particular, the action on objects is given by $\mathcal{C} \mapsto \text{RC}(\mathcal{C})$, and the unit is pointwise given by $\eta_{\mathcal{C}}$.*

Observe that the inclusion pseudofunctor is fully faithful. Hence, the counit is pointwise an isomorphism, and by univalence, coming from an identity. This means precisely that the Rezk completion of a univalent category is itself.

3 On the Lifting of Biadjoints

The examples presented in Section 2 illustrate that various constructions for categories often found produce a not-necessarily univalent category. The Rezk completion provides a solution to this problem. However, while structure on a category can be transported along isomorphisms, we need to verify such transportations for weak equivalences by hand. In Sections 3.2 and 4, we elaborate on how the theory of (displayed) bicategories, in combination with Proposition 13, can be used for modularly constructing the Rezk completion for categories with structure.

Now we can describe the Rezk completion and its universal property a left biadjoint. That is, the Rezk completion forms a pseudofunctor $\text{RC} : \text{Cat} \rightarrow \text{Cat}_{\text{univ}}$, which is a left biadjoint to the forgetful pseudofunctor $\iota : \text{Cat}_{\text{univ}} \rightarrow \text{Cat}$. The transportation of structure from categories to their Rezk completions can then be described as follows. Consider a structure on categories and let $\mathcal{B}$ be the bicategory of categories equipped with that structure and whose morphisms preserve the structure. For example, consider categories with finite products. Then $\mathcal{B}$ is the bicategory of finite products and functors that preserve those products. We have an inclusion $\bar{\iota} : \mathcal{B}_{\text{univ}} \hookrightarrow \mathcal{B}$, where $\mathcal{B}$ is the bicategory of univalent categories equipped finite products. By forgetting the products, we have forgetful pseudofunctors $U : \mathcal{B} \rightarrow \text{Cat}$ and $U|_{\text{univ}} : \mathcal{B}_{\text{univ}} \rightarrow \text{Cat}_{\text{univ}}$. Furthermore, we have that the following diagram commutes:

$$\begin{array}{ccc} \mathcal{B}_{\text{univ}} & \xrightarrow{\bar{\iota}} & \mathcal{B} \\ U|_{\text{univ}} \downarrow & & \downarrow U \\ \text{Cat}_{\text{univ}} & \xrightarrow{\iota} & \text{Cat} \end{array}$$

The transportation and preservation of finite products by the Rezk completion then corresponds to having a left biadjoint $\bar{\text{RC}}$ of $\bar{\iota} : \mathcal{B}_{\text{univ}} \hookrightarrow \mathcal{B}$. We say that the Rezk completion **lifts** from categories to categories equipped with that structure if a left biadjoint $\bar{\text{RC}}$ exists and if the following diagram commutes:

$$\begin{array}{ccc}
\mathcal{B}_{\text{univ}} & \xleftarrow{\text{RC}} & \mathcal{B} \\
\downarrow \text{U|}_{\text{univ}} & & \downarrow \text{U} \\
\text{Cat}_{\text{univ}} & \xleftarrow{\text{RC}} & \text{Cat.}
\end{array}$$

In this section, we present a framework and strategy for constructing such lifts, which we use in Section 4 to lift the Rezk completion from categories to elementary topoi. The methodology builds on the theory of displayed bicategories [1]. In Section 3.1, we discuss how we represent structure on categories via displayed bicategories. Furthermore, we introduce displayed universal arrows in Section 3.2, which allows us to lift the Rezk completion from categories to structured categories. In Section 3.3, we observe that the lifting to e.g., elementary topoi often simplifies in at least two ways, which leads to Theorem 25.

3.1 Displayed Bicategories and Structured Categories

In the remainder of this paper, we make heavy use of displayed bicategories, and we start by recalling this notion. Displayed bicategories serve various purposes, and one of these is that we can use them to modularly construct bicategories and to modularly prove their univalence [1, 3]. To understand what displayed bicategories are, let us first recall an example of a displayed category. We can construct the category **Mon** of monoids and monoid homomorphisms by endowing the objects and morphisms of **Set** with extra structure and properties. Specifically, for every set there is a type of monoid structures on it, and for every function between sets with monoid structures on them we have a type expressing that this function is a homomorphism. One must also prove that the identity function is a homomorphism and that homomorphisms are closed under composition. Displayed categories generalize such descriptions of categories. Specifically, in a displayed category $\mathcal{D}$ over a category $\mathcal{C}$, we have a type $\mathcal{D}(x)$ of displayed objects over every object $x : \mathcal{C}$ and a set $\bar{x} \rightarrow_f \bar{y}$ of displayed morphisms for every morphism $f : x \rightarrow y$ and displayed objects $\bar{x} : \mathcal{D}(x)$ and $\bar{y} : \mathcal{D}(y)$. Displayed bicategories generalize displayed categories to the bicategorical setting.

The bicategories we consider have as their 2-cells just the natural transformations – i.e., each type of 2-cells is contractible. That is, we require no compatibility conditions between the natural transformations and the additional structure. This contractibility assumption is often met when considering categories equipped with structures defined via universal properties, such as finitely complete categories, cartesian closed categories, and elementary topoi. This assumption, in turn, offers a lot of simplifications when stating definitions and verifying properties. For example, the definition of a displayed bicategory reduces to the following, and is thus a simplified version as compared to [1].

► **Definition 14** (👉). *Let $\mathcal{B}$ be a bicategory. A **displayed bicategory** (with contractible 2-cells) $\mathcal{D}$ over $\mathcal{B}$ consists of*

1. *for every $x : \mathcal{B}$, a type $\mathcal{D}(x)$; whose terms are called **displayed objects**;*
2. *for every $f : \mathcal{B}(x, y)$, $\bar{x} : \mathcal{D}(x)$, and $\bar{y} : \mathcal{D}(y)$, a type $\bar{x} \rightarrow_f \bar{y}$; whose terms are called **displayed morphisms**;*
3. *for every $x : \mathcal{B}$ and $\bar{x} : \mathcal{D}(x)$, a displayed morphism of type $\bar{x} \rightarrow_{\text{id}_x} \bar{x}$;*
4. *for every $\bar{f} : \bar{x} \rightarrow_f \bar{y}$ and $\bar{g} : \bar{y} \rightarrow_g \bar{z}$, a displayed morphism of type $\bar{x} \rightarrow_{f \cdot g} \bar{z}$.*

There are, of course, many displayed bicategories that do not have contractible 2-cells. For example, we may consider the bicategory of monoidal categories, constructed as a displayed bicategory over **Cat**. In contrast to categories with binary products, the monoidal product does not automatically commute with the natural transformation since it is not given by

8:10 The Rezk Completion for Elementary Topoi

a universal property. Or, more generally, if we consider bicategories of algebras over a pseudofunctor (or pseudomonad), we require the natural transformations to be compatible with the algebras.

► **Remark 15.** From now on, we leave implicit that the displayed bicategories have contractible 2-cells. In particular, the bicategorical laws need not be verified since these are phrased in terms of equality of 2-cells. For instance, the associativity of composition, up to an invertible 2-cells, follows directly from the statement for categories. Moreover, notions such as displayed pseudofunctors and displayed biadjoints – later referred to as displayed universal arrows – in essence always reduce to the data related to the (displayed) objects and the morphisms, but not 2-cells.

► **Definition 16** (🔗). The **total bicategory** of a displayed bicategory $\mathcal{D}$, denoted $\int \mathcal{D}$, is the bicategory whose objects are (dependent) pairs $(x, \bar{x})$, with $x : \mathcal{B}$ and $\bar{x} : \mathcal{D}(x)$. The morphisms are pairs $(f, \bar{f})$ with $f : \mathcal{B}(\bar{x}, \bar{y})$ and $\bar{f} : \bar{x} \rightarrow_f \bar{y}$. The 2-cells are the 2-cells in $\mathcal{B}$.

► **Definition 17** (🔗). Let $\mathcal{D}$ be a displayed bicategory over $\mathcal{B}$. We have a (strict) pseudofunctor $U : \int \mathcal{D} \rightarrow \mathcal{B}$, called the **forgetful pseudofunctor**.

► **Example 18** (🔗 The displayed bicategory of elementary topoi). Elementary topoi are categories with a lot of structure, such as finite limits, exponentials, and a subobject classifier (for a definition of those structures see Section 4). We construct the bicategory of elementary topoi and logical functors as a displayed bicategory over $\mathbf{Cat}$ in multiple steps. First, we define the displayed bicategory of finite limits, whose displayed objects equip a category with chosen finite limits, and whose displayed morphisms say that a functor preserves those limits. Then the bicategory $\mathbf{FinLim}$ of finitely complete categories is the total bicategory of that displayed bicategory. Second, we define analogously the displayed bicategory $\mathbf{FinLim}_\Omega$, resp. $\mathbf{CCC}$, of subobject classifiers and exponentials respectively, whose base bicategory is $\mathbf{FinLim}$. The bicategory $\mathbf{ElTop}$ is then constructed as the total bicategory of the product ([1, Definition 6.6]) of $\mathbf{FinLim}_\Omega$ and $\mathbf{CCC}$ over $\mathbf{FinLim}$.

► **Remark 19** (the $\sum$ construction). The framework presented below is developed to work for displayed bicategories over $\mathbf{Cat}$. However, the displayed bicategory of elementary topoi (Example 18) as constructed there is a displayed bicategory over $\int_{\mathbf{Cat}} \mathcal{D}$. Nonetheless, if $\mathcal{E}$ is a displayed bicategory over $\int_{\mathbf{Cat}} \mathcal{D}$, the total bicategory $\int_{\int_{\mathbf{Cat}} \mathcal{D}} \mathcal{E}$ is equivalently a bicategory $\int_{\mathbf{Cat}} \sum_{\mathcal{D}} \mathcal{E}$ displayed over $\mathbf{Cat}$ by applying the $\sum$ construction for displayed bicategories ([1, Definition 6.6]). Hence, by the displayed bicategory of elementary topoi, we refer to the one over $\mathbf{Cat}$.

► **Example 20** (The displayed bicategory of univalent elementary topoi and the restriction of displayed bicategories). Univalent topoi are those topoi whose underlying category is univalent. Hence, the bicategory $\mathbf{ElTop}_{\text{univ}}$ of univalent topoi can be constructed as a full subcategory of $\mathbf{ElTop}$ (Example 18). One way to construct $\mathbf{ElTop}_{\text{univ}}$ is thus by restricting the displayed bicategory defining $\mathbf{ElTop}$ to $\mathbf{Cat}_{\text{univ}}$. More generally, if $\mathcal{D}$ is a displayed bicategory over $\mathbf{Cat}$, we can restrict $\mathcal{D}$ to a displayed bicategory $\mathcal{D}_{\text{univ}}$ over $\mathbf{Cat}_{\text{univ}}$ where the displayed objects and morphisms are those in $\mathcal{D}$; see 🔗.

Below, we sometimes use the following terminology. Let $\mathcal{D}$ be a displayed bicategory. Then we say a $\mathcal{D}$ -structure on a category $\mathcal{C}$ is a $\mathcal{D}$ -displayed object over $\mathcal{C}$.

3.2 Displayed Biadjoints

Previously, we discussed that structured categories are the objects in a displayed bicategory $\mathcal{D}$ over $\mathbf{Cat}$. Furthermore, $\mathcal{D}$ can be restricted to a displayed bicategory $\mathcal{D}_{\text{univ}}$ over $\mathbf{Cat}_{\text{univ}}$. That is, $\int \mathcal{D}_{\text{univ}}$ is the full subcategory of $\int \mathcal{D}$ whose objects are univalent categories with a $\mathcal{D}$ -structure. Hence, we have the following diagram of bicategories.

$$\begin{array}{ccc} \int \mathcal{D}_{\text{univ}} & & \int \mathcal{D} \\ U \downarrow & \swarrow \text{RC} & \downarrow U \\ \mathbf{Cat}_{\text{univ}} & & \mathbf{Cat} \\ & \searrow \iota & \end{array}$$

We now reduce the problem of lifting the biadjunction $\text{RC} \dashv \iota$ to the level of displayed bicategories. First, we recall the notion of displayed pseudofunctor [1] in Definition 21. Second, we define (left) displayed universal arrows Definition 23, which is specialized to $\text{RC} \dashv \iota$ in Section 3.3.

Observe that we have a pseudofunctor $\bar{\iota}$ from $\int \mathcal{D}_{\text{univ}}$ to $\int \mathcal{D}$ given by the identity. Indeed, $\mathcal{D}_{\text{univ}}$ is $\mathcal{D}$ restricted to a full subcategory and ι is a pseudofunctor. More generally, we have that pseudofunctors $\mathcal{F}$ between total bicategories can often be constructed by a pseudofunctor $\mathcal{G}$ between the base bicategories and a *displayed pseudofunctor* over $\mathcal{G}$:

► **Definition 21** (🔴🔴). Let $F : \mathcal{B}_1 \rightarrow \mathcal{B}_2$ be a pseudofunctor and $\mathcal{D}_i$ a displayed bicategory over $\mathcal{B}_i$, for $i = 1, 2$. A **displayed pseudofunctor** $\hat{F}$ over F consists of:

1. for every $x : \mathcal{B}_1$, a function $\hat{F}_0 : \mathcal{D}_1(x) \rightarrow \mathcal{D}_2(Fx)$;
2. for every $f : \mathcal{B}_1(x, y)$ and $\bar{x} : \mathcal{D}_1(x), \bar{y} : \mathcal{D}_1(y)$, a function $\hat{F}_1 : (x \rightarrow_f y) \rightarrow (\hat{F}_0 \bar{x} \rightarrow_{Ff} \hat{F}_0 \bar{y})$;

The assignment $(x, \bar{x}) \rightarrow (Fx, \hat{F}_0 \bar{x})$ bundles into a pseudofunctor $\int_F \hat{F} : \int \mathcal{D}_1 \rightarrow \int \mathcal{D}_2$, and is referred to as the **total pseudofunctor**.

For the full definition of displayed pseudofunctors, that is, without contractible 2-cells, see [1] or 🔴.

► **Example 22** (Displayed pseudofunctor from $\mathcal{D}_{\text{univ}}$ to $\mathcal{D}$). The inclusion pseudofunctor $\bar{\iota}$ from $\int \mathcal{D}_{\text{univ}}$ to $\int \mathcal{D}$ is a total pseudofunctor over ι , where the displayed pseudofunctor, denoted $\hat{\iota}$, acts trivially on the displayed objects and morphisms.

In Section 4, we construct a left biadjoint to $\bar{\iota}$ for concrete $\mathcal{D}$. Analogous to the construction of pseudofunctors between total bicategories, biadjoints can be constructed in two steps. That is, we first construct a biadjoint between the base bicategories, and then do a construction at the level of the displayed bicategories. The ingredients used to express the construction at the displayed level are bundled into the definition of **displayed universal arrows** (Definition 23).

For the definition of displayed adjoint equivalence [3, Pages 8 and 9] between (1-)categories, we refer the reader to the formalization 🔴. When the base biadjoint is given by RC , and when considering displayed bicategories with contractible 2-cells, the conditions of being a displayed adjoint equivalence become particularly simple (see Section 3.3). Hence, we will not recall this notion.

► **Definition 23** (🔴). Let R be a pseudofunctor from $\mathcal{B}_1$ to $\mathcal{B}_2$ with a left biadjoint (L, η) . Let $\hat{R}$ be a displayed pseudofunctor from $\mathcal{D}_1$ to $\mathcal{D}_2$, over R . A **(left) displayed universal arrow** for $\hat{R}$ (over (R, L, η)) consists of:

1. $\hat{L} : \prod_{x:\mathcal{B}_2} \mathcal{D}_2(x) \rightarrow \mathcal{D}_1(Lx)$, we write $\hat{L}(\bar{x}) := \hat{L}(x, \bar{x})$;

2. a family $\hat{\eta}_{\bar{x}} : \bar{x} \rightarrow_{\eta_x} \hat{R}(\hat{L}(\bar{x}))$ of displayed morphisms, for all $(x, \bar{x})$ in $\int \mathcal{D}_2$; and such that the displayed functor between displayed hom-categories

$$\hat{\eta}_{\bar{x}} \cdot \hat{R}(-) : \mathcal{D}_1(\hat{L} \bar{x}, \bar{y}) \rightarrow \mathcal{D}_2(\bar{x}, \hat{R} \bar{y}),$$

is a displayed adjoint equivalence whose base of displayment is the adjoint equivalence $\eta_x \cdot R(-)$ given by (R, L, η) .

► **Proposition 24** (🔴). *Let $(\hat{R}, \hat{L}, \hat{\eta})$ be a displayed universal arrow for (R, L, η) . Then the total pseudofunctor $\int_R \hat{R} : \int_{\mathcal{B}_1} \mathcal{D}_1 \rightarrow \int_{\mathcal{B}_2} \mathcal{D}_2$, has a left biadjoint given by $(x, \bar{x}) \mapsto (Lx, \hat{L}\bar{x})$ and whose unit is given by $(\eta_x, \hat{\eta}_{\bar{x}})$.*

3.3 Displayed Universal Arrows over the Rezk Completion

In Section 3.2, we defined the notion of displayed universal arrows, which, in particular, provides the right formulation for the modular construction of biadjoints. Often, there are simplifications due to how the bicategories are defined, for instance, in the case of topoi, the structure related to 2-cells becomes irrelevant. Furthermore, RC is completely characterized through weak equivalences, which allows for a further simplification. In this section, we specialize the notion of displayed universal arrows to take into account these considerations. In particular, we conclude that for lifting the Rezk completion to a bicategory such as $\mathbf{ElTop}$, the displayed universal arrows become particularly simple (Theorem 25).

Recall that structure on a category is represented via a displayed bicategory $\mathcal{D}$ over $\mathbf{Cat}$, whose displayed objects witness the structure and whose displayed morphisms witness preservation of the structure. Then the bicategory of such structured categories is the total bicategory $\int_{\mathbf{Cat}} \mathcal{D}$. Furthermore, its full subbicategory on univalent structured categories is $\int_{\mathbf{Cat}_{\text{univ}}} \mathcal{D}_{\text{univ}}$. Moreover, we have the following diagram:

$$\begin{array}{ccc} \int \mathcal{D}_{\text{univ}} & \begin{array}{c} \xleftarrow{\quad ? \quad} \\ \xrightarrow{\quad \int \hat{\iota} \quad} \end{array} & \int \mathcal{D} \\ U \downarrow & & \downarrow U \\ \mathbf{Cat}_{\text{univ}} & \begin{array}{c} \xleftarrow{\quad \text{RC} \quad} \\ \xrightarrow{\quad \iota \quad} \end{array} & \mathbf{Cat} \end{array}$$

Hence, the construction of the Rezk completion for categories with a $\mathcal{D}$ -structure corresponds to the construction of a left biadjoint of the inclusion $\bar{\iota} = \int \hat{\iota}$. If such a left biadjoint exists, we say that the Rezk completion (for categories) *lifts* to categories with structure.

By Proposition 24, the construction of such a left biadjoint amounts to the construction of a displayed universal arrow for RC. That is, for every $\mathcal{C} \in \mathbf{Cat}$, and every displayed object $x : \mathcal{D}(\mathcal{C})$ on $\mathcal{C}$ (i.e., $\mathcal{C}$ has a given $\mathcal{D}$ -structure), we have to construct a $\mathcal{D}$ -structure on $\text{RC}(\mathcal{C})$ and show that $\eta_{\mathcal{C}} : \mathcal{C} \rightarrow \text{RC}(\mathcal{C})$ is $\mathcal{D}$ -structure preserving. How one can construct those in concrete cases is explained in Section 4. Furthermore, we have to show that the displayed precomposition functor $\hat{\eta}_{\bar{x}} \cdot \hat{R}(-)$, see Definition 23, is indeed a displayed adjoint equivalence. In a bicategory such as $\mathbf{ElTop}$, this equivalence condition reduces to the statement that if a structure-preserving functor $F : \mathcal{C} \rightarrow \mathcal{E}$, into a univalent category, factorizes through $\eta_{\mathcal{C}}$, then the functor $\text{RC}(\mathcal{C}) \rightarrow \mathcal{E}$ is also structure-preserving. More generally:

► **Theorem 25** (🔴). *Let $\mathcal{D}$ be a displayed bicategory with contractible 2-cells over $\mathbf{Cat}$ such that for every weak equivalence $G : \mathcal{C}_0 \rightarrow \mathcal{C}_1$, whose codomain is univalent, we have:*

1. *for $\bar{x} : \mathcal{D}(\mathcal{C}_0)$, there is a $\hat{x} : \mathcal{D}(\mathcal{C}_1)$ and a $\hat{G} : \bar{x} \rightarrow_G \hat{x}$;*
 2. *for every univalent category $\mathcal{C}_2$, functors $F : \mathcal{C}_0 \rightarrow \mathcal{C}_2$ and $H : \mathcal{C}_1 \rightarrow \mathcal{C}_2$, and a natural isomorphism $\alpha : G \cdot H \Rightarrow F$, if $\bar{x}_i : \mathcal{D}(\mathcal{C}_i)$ and $\hat{F} : \bar{x}_0 \rightarrow_F \bar{x}_2$, then there is a $\hat{H} : \bar{x}_1 \rightarrow_H \bar{x}_2$.*
- Then the pseudofunctor $\text{RC} : \mathbf{Cat} \rightarrow \mathbf{Cat}_{\text{univ}}$ lifts to a left biadjoint for $\int \mathcal{D}_{\text{univ}} \rightarrow \int \mathcal{D}$.*

Theorem 25 is an immediate consequence of Proposition 24. The first two conditions in the definition of universal arrows (i.e., Items 1 and 2) follow from the first condition in this theorem. Those guarantee that RC creates and preserves the structure given by $\mathcal{D}$, and that the unit preserves that structure. The second condition in Theorem 25 is a vast simplification of the axiom of a displayed universal arrow, which follows due to the local contractibility condition on $\mathcal{D}$.

Theorem 25 thus says that the Rezk completion lifts from categories to some structured categories, if we can show that the Rezk completion $\text{RC}(\mathcal{C})$ inherits the structure on a category $\mathcal{C}$, the weak equivalence $\eta : \mathcal{C} \rightarrow \text{RC}(\mathcal{C})$ becomes a homomorphism for this structure, and that the universal property of the Rezk completion commutes with the structure. To conclude the universal property, we usually have to show that any homomorphism of structured categories – into a univalent structured category – factorizes uniquely through the weak equivalence given by the Rezk completion (i.e., the unit). However, Theorem 25 witnesses that we do not need to verify the uniqueness, as a consequence of the contractible 2-cells. Furthermore, observe that the second condition is phrased in terms of natural isomorphisms rather than equality of 2-cells. By univalence of $\mathcal{C}_2$, these two formulations coincide. Yet, to conclude the third condition for concrete structures, we often rely on the naturality, making this formulation more practical to use.

4 Examples

In this section, we lift the Rezk completion from categories to various categorical structures, resulting in the Rezk completion for $(W\text{-})\text{topoi}$. To construct the lifting, we use Theorem 25.

First, we consider categories equipped with finite limits in Section 4.1. The lifting thereof is concluded in Proposition 36. Dually, we conclude the lifting of finite colimits in Corollary 38. Furthermore, we discuss the lifting of infinite limits in Lemma 39.

Afterwards, we consider elementary topoi. In particular, we lift the Rezk completion to cartesian closed categories and categories with a subobject classifier. The lifting thereof is concluded in Proposition 43 and Theorem 49 respectively. Furthermore, in Section 4.4, we consider the case where a category is equipped with a parameterized natural numbers object, which combined with the previous result, provides the Rezk completion for $W\text{-topoi}$.

► **Remark 26.** In Section 4.1, we discuss the lifting of the Rezk completion from categories to categories with binary products in detail. As will be seen in Section 4.1, the lifting amounts to proving that equivalences between categories preserve, create, and reflect structures. To this end, we will only spell out the details in Section 4.1 and refer the reader to the formalization for the proofs of the other structures.

4.1 Finitely Complete Categories

We now lift the Rezk completion from categories to finitely (co)complete categories. It is well-known that the existence of finite limits follows from the existence of finite products and equalizers. Hence, it is sufficient to lift the Rezk completion for those limits. We discuss the lifting of binary products and refer the reader to the formalization for the lifting of terminal objects and equalizers.

► **Definition 27** (🔗). *Let $\mathcal{D}^\times$ be the Cat -displayed bicategory whose displayed objects over $\mathcal{C}$ witness that $\mathcal{C}$ is equipped with binary products and whose displayed morphisms over a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ witness that F preserves binary products up to isomorphism. The total bicategory, denoted $\text{Cat}^\times$, is the bicategory of categories with binary products and product-preserving functors.*

Hence, the goal of this subsection is to lift $\text{RC} : \text{Cat} \rightarrow \text{Cat}_{\text{univ}}$ to a left biadjoint for $\text{Cat}_{\text{univ}}^{\times} \hookrightarrow \text{Cat}^{\times}$. To lift this, we apply Theorem 25. Concretely, we have to show for every category $\mathcal{C}$ with binary products the following.

1. The Rezk completion $\text{RC}(\mathcal{C})$ of $\mathcal{C}$ is equipped with binary products, and the weak equivalence $\eta_{\mathcal{C}} : \mathcal{C} \rightarrow \text{RC}(\mathcal{C})$ preserves products.
2. For every binary product preserving functor $F : \mathcal{C} \rightarrow \mathcal{E}$ into a univalent category $\mathcal{E}$, its factorization through $\eta_{\mathcal{C}} : \mathcal{C} \rightarrow \text{RC}(\mathcal{C})$ should preserve binary products as well.

Both statements in Item 1 follow once we've proven that the image of a product is preserved by any weak equivalence (Proposition 29). The proof of the latter follows the same strategy as proving that an equivalence or isomorphism preserves binary products, after a careful observation that being a product is a proposition. To conclude that $\text{RC}(\mathcal{C})$ has all binary products, as opposed to only those in the image of $\eta_{\mathcal{C}}$, we rely on univalence of $\text{RC}(\mathcal{C})$ in order to apply the essential surjectivity. This concludes the first item in Theorem 25, and will be discussed in length in the first paragraph: the creation of products.

Item 2 follows from the fact that fully faithful functors, and hence weak equivalences, reflect binary products, and using essential surjectivity of $\eta_{\mathcal{C}}$. This concludes the second item in Theorem 25, and will be discussed in length in the first paragraph: the universality.

► **Remark 28.** In the remainder of this section, we write products for binary products, to simplify notation.

The creation of products. We now show that if $\mathcal{C}$ is a category with products, then $\text{RC}(\mathcal{C})$ has products and those are preserved by $\eta_{\mathcal{C}}$. First, we show that weak equivalences preserve products.

► **Proposition 29** (🔴). *Let $G : \mathcal{C} \rightarrow \mathcal{D}$ be a weak equivalence and $(x_1 \xrightarrow{\pi_1} p \xleftarrow{\pi_2} x_2)$ a product in $\mathcal{C}$. Then $(G x_1 \xleftarrow{G \pi_1} G p \xrightarrow{G \pi_2} G x_2)$ is a product in $\mathcal{D}$.*

Proof. Let $(G x_1 \xrightarrow{g_1} y \xleftarrow{g_2} G x_2)$ be a cone in $\mathcal{D}$. We have to construct a morphism $h : \mathcal{D}(y, G p)$ which is unique w.r.t., to the property that for $k = 1, 2$, the following diagram commutes:

$$\begin{array}{ccc} & y & \\ g_k \swarrow & & \downarrow h \\ G x_k & \xleftarrow{G \pi_k} & G p \end{array} \quad (1)$$

Type-theoretically, we need to show that the following type is contractible:

$$A := \sum h : \mathcal{D}(y, G p), h \cdot G(\pi_1) = g_1 \times h \cdot G(\pi_2) = g_2. \quad (2)$$

To show that A is inhabited, we use essential surjectivity of G to obtain an $x : \mathcal{C}$ and an isomorphism $i : G x \cong y$. Note that we can construct x and i , because being contractible is a proposition.

We claim that h is given by $y \xrightarrow{i^{-1}} G x \xrightarrow{G(\langle f_1, f_2 \rangle)} G p$, where $f_k := G^{-1}(i \cdot g_k)$ and $\langle f_1, f_2 \rangle$ is the unique morphism given by the universal property of p .

That h is an inhabitant of A follows from the following calculation (🔴):

$$\begin{aligned} h \cdot G(\pi_k) &= i^{-1} \cdot G(\langle f_1, f_2 \rangle) \cdot G(\pi_k), && \text{by dfn. of } h, \\ &= i^{-1} \cdot G(\langle f_1, f_2 \rangle \cdot \pi_k), && \text{by functoriality of } G, \\ &= i^{-1} \cdot G(f_k), && \text{by dfn. of } \langle f_1, f_2 \rangle \\ &= i^{-1} \cdot G(G^{-1}(i \cdot g_k)) = g_k, && \text{by dfn. of } f_k. \end{aligned}$$

It remains to verify that the type A is a proposition (🔴). Let $\phi_1 := (h_1, q_1)$ and $\phi_2 := (h_2, q_2)$ be in A . Since A is a subtype of $\mathcal{D}(y, Gp)$, it suffices to show that $h_1 = h_2$. This is equivalent to showing $G^{-1}(i \cdot h_1) = G^{-1}(i \cdot h_2)$ since i is an isomorphism and a fully faithful functor reflects equality of morphisms. The result now follows by the universal property of the product and by assumption we have $h_1 \cdot G(\pi_k) = h_2 \cdot G(\pi_k)$. ◀

Observe that Proposition 29 entails that if $\mathcal{C}$ and $\mathcal{D}$ are equipped with binary products, and if $\mathcal{D}$ is univalent (i.e., $\mathcal{D} = \text{RC}(\mathcal{C})$), then $G(x \times_{\mathcal{C}} y) = G(x) \times_{\mathcal{D}} G(y)$.

Before concluding that $\text{RC}(\mathcal{C})$ is equipped with products if $\mathcal{C}$ is, observe:

► **Lemma 30** (🔴). *In a univalent category, products are unique up to identity, if they exist.*

► **Proposition 31** (🔴). *If $\mathcal{C}$ is equipped with binary products, then $\text{RC}(\mathcal{C})$ is equipped with binary products.*

Proof. Let $y_1, y_2 : \text{RC}(\mathcal{C})$. We have to construct the binary product of y_1 and y_2 . By univalence of $\text{RC}(\mathcal{C})$, the product of y_1 and y_2 is necessarily unique, if it exists. Hence, we can use essential surjectivity of $\eta_{\mathcal{C}}$ to construct x_1 and x_2 such that $\eta_{\mathcal{C}}(x_k) \cong y_k$. Since the product is closed under isomorphism, it suffices to construct the product of $G(x_1)$ and $G(x_2)$, which we have constructed in Proposition 29. ◀

The universality. The final step in lifting RC to categories with binary products is proving that the universal property of the Rezk completion commutes with binary products. Let $\mathcal{E}$ be a univalent category with binary products and $F : \mathcal{C} \rightarrow \mathcal{E}$ a functor preserving binary products up to isomorphism. Suppose F factorizes through $\eta_{\mathcal{C}}$, up to a natural isomorphism α , as depicted in the following diagram:

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{G} & \mathcal{D} \\ & \searrow F & \swarrow H \\ & \mathcal{E} & \end{array} \quad \begin{array}{c} \alpha \\ \swarrow \nearrow \end{array} \quad (3)$$

We have to show that if G is a weak equivalence and if F preserves binary products, then H preserves binary products.

► **Lemma 32** (🔴). *A fully faithful functor reflects products.*

► **Lemma 33** (🔴). *Assume that G is a weak equivalence and that F preserves binary products. Then H preserves binary products.*

Proof. We have to show that if $(y_1 \xrightarrow{\pi_1} p_y \xleftarrow{\pi_2} y_2)$ is a product cone in $\mathcal{D}$, then $(H y_1 \xrightarrow{H \pi_1} H p_y \xleftarrow{H(\pi_2)} H y_2)$ is a product in $\mathcal{E}$.

Since G is essential surjective, we have objects $x_1, x_2, p_x : \mathcal{C}$ such that $G x_1 \cong y_1$, $G x_2 \cong y_2$ and $G p_x \cong p_y$. Note that we can construct such objects and isomorphisms, because being a product is a proposition. Now, since α is a natural isomorphism, we have isomorphisms $H(y_k) \cong F(x_k)$, for $k = 1, 2$, and $H(p_y) \cong F(p_x)$.

By Lemma 32, p_x is a product of x_1 and x_2 . Then by assumption on F , $F(p_x)$ is a product of $F(x_1)$ and $F(x_2)$. The claim now follows since products are closed under isomorphism and because the following diagram commutes:

$$\begin{array}{ccccc} F(x_1) & \xrightarrow{F(\pi_1)} & F(p_x) & \xleftarrow{F(\pi_2)} & F(x_2) \\ \downarrow \cong & & \downarrow \cong & & \downarrow \cong \\ H(y_1) & \xrightarrow{H(\pi_1)} & H(p_y) & \xleftarrow{H(\pi_2)} & H(y_2) \end{array} \quad \blacktriangleleft$$

8:16 The Rezk Completion for Elementary Topoi

Hence, Lemma 33 implies that the action of $\text{RC} : \text{Cat} \rightarrow \text{Cat}_{\text{univ}}$ on morphisms (functors) preserves the preservation of products. This concludes the proof of the assumptions in Theorem 25 instantiated to the displayed bicategory of binary products.

The lifted biadjunction. We now conclude that the Rezk completion for categories lifts to categories with binary products. Furthermore, we conclude the same result for finite limits more generally, or more precisely: terminal objects and equalizers. In the formalization, we also consider pullbacks, because this is used for lifting the subobject classifier.

► **Proposition 34** (🔗). *The inclusion $(\text{Cat}^\times)_{\text{univ}} \hookrightarrow \text{Cat}^\times$ has a left biadjoint.*

Proof. The result follows from Theorem 25 whose assumptions are proven in Lemma 33 and Proposition 31. ◀

► **Definition 35** (🔗). *Let $\mathcal{D}^{\text{lim}}$ be the product of the $\mathcal{D}^\times$ and the displayed bicategories $\mathcal{D}^1$ and $\mathcal{D}^{\text{eq}}$ whose displayed objects over a category $\mathcal{C}$ witness that $\mathcal{C}$ is equipped with a terminal object, resp. equalizers, and whose displayed morphisms over a functor witness the preservation of the terminal object and equalizers respectively. The total bicategory of $\mathcal{D}^{\text{lim}}$, denoted FinLim , is the bicategory of finitely complete categories and finite-limit preserving functors.*

► **Proposition 36** (🔗). *The inclusion $\text{FinLim}_{\text{univ}} \hookrightarrow \text{FinLim}$ has a left biadjoint.*

Colimits. Previously, we’ve constructed the Rezk completion for finitely complete categories. The duality between limits and colimits entail the same result for finitely cocomplete categories, as a consequence of the following lemma:

► **Lemma 37** (🔗). *Let $F : \mathcal{C} \rightarrow \mathcal{D}$ be a functor. Then F is a weak equivalence if and only if $F^{\text{op}} : \mathcal{C}^{\text{op}} \rightarrow \mathcal{D}^{\text{op}}$ is a weak equivalence.*

We define the bicategory FinColim of finitely cocomplete categories similarly to the bicategory of finitely complete categories. That is, as the product of displayed bicategories of binary coproducts and coequalizers 🔗:

► **Corollary 38** (🔗). *The inclusion $\text{FinColim}_{\text{univ}} \hookrightarrow \text{FinColim}$ has a left biadjoint.*

Infinite limits. Whereas the Rezk completion lifts to finitely complete categories, the same result does not need to hold for infinitary limits.

Recall that the axiom of (n, m) -choice denotes the statement [23, Exer. 7.8.]

$$\prod_{X : \text{Set}} \prod_{Y : X \rightarrow \mathcal{U}_n} \left(\prod_{x : X} \|Y(x)\|_m \right) \rightarrow \left\| \prod_{x : X} Y(x) \right\|_m,$$

where $-1 \leq n, m \leq \infty$ and $\mathcal{U}_n$ denotes the type of n -types.

► **Lemma 39.** *Assume the axiom of $(\infty, -1)$ -choice holds. Let $\mathcal{C}$ be a category equipped with limits indexed by a setcategory $\mathcal{A}$. Then $\text{RC}(\mathcal{C})$ has all $\mathcal{A}$ -indexed limits.*

Proof. Let $\mathcal{J} : \mathcal{A} \rightarrow \text{RC}(\mathcal{C})$ be a diagram. We want to conclude that $\mathcal{J}$ has a limit. Let $X := \text{ob}(\mathcal{A})$ and $Y : \mathcal{A} \rightarrow \mathcal{U}_\infty : a \mapsto \sum_{x : \mathcal{C}} \mathcal{J}(a) \cong \eta_{\mathcal{C}}(x)$. Observe that $\mathcal{J}(a) \cong \eta_{\mathcal{C}}(x)$ is a set, but that the homotopy level of $\text{ob}(\mathcal{C})$ might indeed be infinite.

By choice, we have a function:

$$\left(\prod_{a : \mathcal{A}} \left\| \sum_{x : \mathcal{C}} \mathcal{J}(a) \cong \eta_{\mathcal{C}}(x) \right\| \right) \rightarrow \left\| \prod_{a : \mathcal{A}} \sum_{x : \mathcal{C}} \mathcal{J}(a) \cong \eta_{\mathcal{C}}(x) \right\|,$$

Since $\eta_{\mathcal{C}}$ is essentially surjective, the domain thereof is inhabited. Hence, we are left to construct

$$\|\prod_{a:\mathcal{A}} \sum_{x:\mathcal{C}} \mathcal{J}(a) \cong \eta_{\mathcal{C}}(x)\| \rightarrow \lim(\mathcal{J}),$$

where $\lim(\mathcal{J})$ is the type of limits of $\mathcal{J}$. Since $\lim(\mathcal{J})$ is a proposition by univalence of $\mathbf{RC}(\mathcal{C})$, we are given

$$k' : \prod_{a:\mathcal{A}} \sum_{x:\mathcal{C}} \mathcal{J}(a) \cong \eta_{\mathcal{C}}(x).$$

In particular, we have a diagram $\mathcal{K} : \mathcal{A} \rightarrow \mathcal{C}$, which by assumption on $\mathcal{C}$ has a limit cone $\lim(\mathcal{K})$. We claim that $\eta_{\mathcal{C}}(\lim(\mathcal{K}))$ is a limit for $\mathcal{J}$.

First, $\eta_{\mathcal{C}}$ preserves limits. Hence, $\eta_{\mathcal{C}}(\lim(\mathcal{K}))$ is a limit for the diagram $\mathcal{K} \cdot \eta_{\mathcal{C}}$. Since limits are closed under isomorphic diagrams, the result follows if $\mathcal{J} \cong \mathcal{K} \cdot \eta_{\mathcal{C}}$. Such an isomorphism is indeed given by k' .

Observe that the naturality and functoriality conditions need not be verified since we can assume that $\mathcal{J}$ is discrete. $\blacktriangleleft$

4.2 Cartesian Closed Categories

We now lift the Rezk completion to cartesian closed categories. The approach taken therefore is the same as for the lifting of limits. In particular, we rely on the results from Section 4.1.

Recall that an object $x : \mathcal{C}$ in a category with binary products is **exponentiable** if $- \times x : \mathcal{C} \rightarrow \mathcal{C}$ has a right adjoint, which is denoted as $(-)^x : \mathcal{C} \rightarrow \mathcal{C}$. That is, for every $y : \mathcal{C}$ there are given an object $y^x : \mathcal{C}$ and a morphism $\mathbf{ev} := \mathbf{ev}_{x,y} : \mathcal{C}(y^x \times x, y)$ which is universal in the sense that $(y^x, \mathbf{ev})$ is the terminal such pair. The object y^x is referred to as the *exponential* of x with y , and $\mathbf{ev}_{x,y}$ is referred to as the *evaluation morphism*. We say that a category is **cartesian closed** if it has all finite products and all exponentials.

Furthermore, recall that a binary product preserving functor $F : \mathcal{C} \rightarrow \mathcal{E}$ between cartesian closed categories is a **cartesian closed functor** which **preserves exponentials**. That is, for every $x, y : \mathcal{C}$ the unique morphism from $F(y^x)$ to $F(y)^{F(x)}$ is an isomorphism.

Recall from Definition 27 that we denote by $\mathbf{Cat}^{\times}$ the bicategory of categories with binary products.

► **Definition 40** (🔴). *Let $\mathcal{D}^{\text{exp}}$ be the $\mathbf{Cat}^{\times}$ -displayed bicategory whose displayed objects over $\mathcal{C}$ witness that $\mathcal{C}$ is cartesian closed and whose displayed morphisms witness the preservation of exponentials. The total bicategory, denoted $\mathbf{CCC}$, is the bicategory of cartesian closed categories and cartesian closed functors.*

Hence, to lift the Rezk completion to cartesian closed categories, we lift $\mathbf{RC} : \mathbf{Cat}^{\times} \rightarrow \mathbf{Cat}_{\text{univ}}^{\times}$ to a left biadjoint for $\mathbf{CCC}_{\text{univ}} \hookrightarrow \mathbf{CCC}$. Analogous to before, we use Theorem 25 and take the same steps as before.

Recall that the Rezk completion lifts to categories with binary products. In particular, there is a necessarily unique isomorphism $\mu := \mu_{x,y} : \eta(x) \times \eta(y) \cong \eta(x \times y)$, for every $x, y : \mathcal{C}$ and natural in x and y .

► **Lemma 41** (🔴). *Let $\mathcal{C}$ be a cartesian closed category and $G : \mathcal{C} \rightarrow \mathcal{E}$ a weak equivalence. Then the image of an exponential, under G is again an exponential. That is, if $(y^x, \mathbf{ev})$ is an exponential of x with y , then $(G(y^x), G(\mathbf{ev}))$ is an exponential of $G(x)$ with $G(y)$.*

► **Proposition 42** (🔴🔴). *If $\mathcal{C}$ is a cartesian closed category, then $\mathrm{RC}(\mathcal{C})$ is cartesian closed. Furthermore, $\eta_{\mathcal{C}} : \mathcal{C} \rightarrow \mathrm{RC}(\mathcal{C})$ is a cartesian closed functor.*

Proof. Since exponentials are unique up to isomorphism, the preservation of exponentials is equivalent to the statement that the image of an exponential object is again an exponential object as in Lemma 41. ◀

We are now able to conclude that the Rezk completion lifts from categories to cartesian closed categories.

► **Proposition 43** (🔴). *The inclusion $\mathrm{CCC}_{\mathrm{univ}} \hookrightarrow \mathrm{CCC}$ has a left biadjoint.*

Proof. Again, we use Theorem 25 to $\mathcal{D} := \mathcal{D}^{\mathrm{exp}}$. The first two conditions thereof are proven in Proposition 42. The last condition (universality) follows since weak equivalences reflect exponential objects. ◀

4.3 Elementary Topoi

We now lift the Rezk completion to elementary topoi: Cartesian closed categories equipped with equalizers and a subobject classifier. Observe that elementary topoi are necessarily finitely complete. By Propositions 36 and 43, we know that the cartesian closed structure and the equalizers are lifted. Hence, it remains to lift the subobject classifier.

Let $\mathcal{C}$ be a category with a terminal object T . Recall that a **subobject classifier** consists of an object $\Omega : \mathcal{C}$ and a monomorphism $\tau : \mathcal{C}(T, \Omega)$, which furthermore satisfies the following universal property. For every mono $f : \mathcal{C}(x, y)$, there is a unique morphism $\chi_f : \mathcal{C}(y, \Omega)$ such that the following diagram commutes and is a pullback square:

$$\begin{array}{ccc} x & \xrightarrow{!} & T \\ f \downarrow & & \downarrow \tau \\ y & \xrightarrow{\chi_f} & \Omega \end{array}$$

We refer to τ as the *truth map*.

Furthermore, recall that a functor $F : \mathcal{C}_0 \rightarrow \mathcal{C}_1$ between categories with a terminal object, denoted T_i for $i = 0, 1$, and a subobject classifier (Ω_i, τ_i) preserves the subobject classifier if F preserves the terminal object and one of the following two equivalent conditions holds:

1. $\left(F(\Omega_0), T_1 \xrightarrow{!} F(T_0) \xrightarrow{F(\tau_0)} F(\Omega_0) \right)$ is a subobject classifier in $\mathcal{C}_1$;
 2. there is an isomorphism $i : F(\Omega_0) \simeq \Omega_1$ such that $F(\tau_0) \cdot i = !^{-1} \cdot \tau_1$;
- where $!$ denotes the unique morphism given by terminality of $F(T_0)$.

Recall from Definition 35 that we denote by Cat^1 the bicategory of categories with a terminal object.

► **Definition 44** (🔴). *Let $\mathcal{D}^\Omega$ be the Cat^1 -displayed bicategory whose displayed objects over a category with a terminal object $\mathcal{C}$ are subobject classifiers and whose displayed morphisms witness the preservation of the subobject classifier. The total bicategory, denoted Cat^Ω , is the bicategory of categories with a terminal object and a subobject classifier and whose morphisms are functors preserving the terminal object and the subobject classifier.*

Recall that we denote by FinLim the bicategory of finitely complete categories, which is constructed as $\mathrm{FinLim} := \int_{\mathrm{Cat}} \mathcal{D}^{\mathrm{lim}}$. Furthermore, observe that $\mathcal{D}^{\mathrm{exp}}$ and $\mathcal{D}^\Omega$ can be considered as displayed bicategories over FinLim .

► **Definition 45** (🔗). Let $\mathcal{D}^{\text{top}}$ be the product of the FinLim -displayed bicategories $\mathcal{D}^{\text{exp}}$ with $\mathcal{D}^{\Omega}$. The total bicategory of $\mathcal{D}^{\text{top}}$, denoted ElTop , is the bicategory of elementary topoi, logical functors, and natural transformations.

Recall that a morphism $f : \mathcal{C}(x, y)$ is a monomorphism if and only if

$$\begin{array}{ccc} x & \xlongequal{\quad} & x \\ \parallel & & \downarrow f \\ x & \xrightarrow{f} & y \end{array}$$

is a pullback square.

► **Lemma 46** (🔗). Weak equivalences preserve and reflect monomorphisms.

Observe that the reflection of monomorphisms follows from only fully faithfulness, but not essential surjectivity.

Recall that if G is a weak equivalence and T_0 is a terminal object, then $G(T_0)$ is terminal. Lemma 47 follows since weak equivalences preserve pullbacks:

► **Lemma 47** (🔗). Let $G : \mathcal{C} \rightarrow \mathcal{E}$ be a weak equivalence between categories with a terminal object, denoted $T_{\mathcal{C}}$ and $T_{\mathcal{E}}$ respectively. Suppose that $(\Omega, \tau : T_{\mathcal{C}} \rightarrow \Omega)$ is a subobject classifier in $\mathcal{C}$. Then $G(\Omega)$ is a subobject classifier whose truth map is $! \cdot G(\tau) : T_{\mathcal{E}} \rightarrow G(\Omega)$, where $!$ is the unique morphism from $T_{\mathcal{E}}$ to $G(T_{\mathcal{C}})$.

► **Proposition 48** (🔗). The inclusion $\text{Cat}_{\text{univ}}^{\Omega} \hookrightarrow \text{Cat}^{\Omega}$ has a left biadjoint.

We now conclude that $\text{RC} : \text{FinLim} \rightarrow \text{FinLim}_{\text{univ}}$ lifts to a left biadjoint for $\text{ElTop}_{\text{univ}} \rightarrow \text{ElTop}$.

► **Theorem 49** (🔗). The inclusion $(\text{ElTop})_{\text{univ}} \hookrightarrow \text{ElTop}$ has a left biadjoint. In particular:

1. the Rezk completion $\text{RC}(\mathcal{E})$ of an elementary topos $\mathcal{E}$ satisfies the axioms of an elementary topos;
2. the weak equivalence $\eta_{\mathcal{E}} : \mathcal{E} \rightarrow \text{RC}(\mathcal{E})$ is a logical functor;
3. and, $(\text{RC}(\mathcal{E}), \eta_{\mathcal{E}})$ is universal among the univalent elementary topoi.

4.4 W-Topoi

As a final example, we lift the Rezk completion to categories equipped with a (parameterized) natural numbers object. In combination with Proposition 48, we conclude that the Rezk completion lifts from categories to elementary topoi with a natural numbers object. Since elementary topoi with a natural numbers object have W-types [18, Prop. 3.6.], we call such topoi **W-topoi**.

The most common way to interpret, or axiomatize, the object of natural numbers in a category leads to the definition of a *natural numbers object*. Nonetheless, in the absence of exponentials, there is a more appropriate interpretation by strengthening the recursion principle and is known as a *parameterized natural numbers object* 🔗 [17]. We work with the more general version.

A **parameterized natural numbers object** (pNNO) in a category $\mathcal{C}$ with finite products is a tuple $(\mathbb{N}, z, s)$ where $\mathbb{N} : \mathcal{C}$ is an object and $z : \mathcal{C}(T, \mathbb{N})$, $s : \mathcal{C}(\mathbb{N}, \mathbb{N})$ are morphisms which satisfies the following universal property: for every tuple $(t : \mathcal{C}, m : \mathcal{C}, z' : \mathcal{C}(t, m), s' : \mathcal{C}(m, m))$, there exists a unique $f : \mathcal{C}(t \times \mathbb{N}, m)$ such that the following diagram commutes:

$$\begin{array}{ccccc}
t & \xrightarrow{\langle \text{id}_t, ! \cdot z \rangle} & t \times \mathbb{N} & \xleftarrow{\text{id} \times s} & t \times \mathbb{N} \\
& \searrow z' & \downarrow f & & \downarrow f \\
& & m & \xleftarrow{s'} & m
\end{array}$$

Observe that a pNNO is unique up to isomorphism since its universal property is an initiality condition. Alternatively, every pNNO is also a natural numbers object, and natural numbers objects are unique up to isomorphism. Hence, the structure of a pNNO is a proposition when considering univalent categories $\color{red}{\text{👉}}$.

Recall that the image of a terminal object under a weak equivalence is again terminal.

► **Lemma 50** ($\color{red}{\text{👉👉}}$). *Let $G : \mathcal{C}_0 \rightarrow \mathcal{C}_1$ be a weak equivalence where both $\mathcal{C}_0$ and $\mathcal{C}_1$ have finite products. Assume that $\mathbb{N} : \mathcal{C}_0$, $z : \mathcal{C}_0(T_0, \mathbb{N})$, and $s : \mathcal{C}_0(\mathbb{N}, \mathbb{N})$ are given. Then $(\mathbb{N}, z, s)$ is a pNNO if and only if $(G(\mathbb{N}), G(z), G(s))$ is a pNNO.*

Let $\text{Cat}_{\mathbb{N}}$ be the bicategory whose objects are categories equipped with finite products and a pNNO, and whose morphisms are terminal preserving functors $F : \mathcal{C}_0 \rightarrow \mathcal{C}_1$ such that the unique morphism from $\mathbb{N}_1 \rightarrow F(\mathbb{N}_0)$, induced by the universal property of the pNNO, is an isomorphism.

► **Proposition 51** ($\color{red}{\text{👉}}$). *The inclusion $(\text{Cat}_{\mathbb{N}})_{\text{univ}} \hookrightarrow \text{Cat}_{\mathbb{N}}$ has a left biadjoint.*

► **Corollary 52** ($\color{red}{\text{👉}}$). *The left biadjoint RC lifts to elementary topoi with an (parameterized) natural numbers object.*

5 Related Work

We elaborate on the different methods to extend the Rezk completion to categories with structure found in the literature.

Ahrens et al. [4] extended the Rezk completion to categories with families (CwFs). That is, they showed that every weak equivalence into a univalent category (i.e., the Rezk completion) induces a map from the type of CwFs on the domain to the type of CwFs on the codomain. Furthermore, they constructed for every weak equivalence an equivalence between the representable map structures on the involved categories. These results allowed them to construct an equivalence between representable maps of presheaves on a category and the CwFs on its Rezk completion. While our focus has been on displayed bicategories with contractible 2-cells, CwFs and representable maps require propositional 2-cells. Nonetheless, Theorem 25 could be slightly generalized to have propositional 2-cells. In particular, we expect our methodology to also work for these structures.

Veltri and Van der Weide [30] proved that the inclusion of the bicategory of univalent groupoids into the bicategory of all groupoids admits a left biadjoint. Furthermore, it was shown that this biadjoint lifts to a variety of structures on groupoids. However, instead of considering each structure individually, as we have done, they presented a signature for higher inductive types to encode structure on groupoids and they showed that the inclusion of structured univalent groupoids into structured groupoids admits a left biadjoint. In particular, they considered *algebraic* structure, whereas we consider structure defined via a universal property. Furthermore, while we work with biadjoints in terms of equivalences between hom-categories, they worked with biadjoints in terms of the unit-counit description.

Wullaert et al. [33] extended the Rezk completion to *monoidal categories*. In particular, they also proved that the biadjoint given by the Rezk completion lifts from categories to monoidal categories, although not in the exact same words. The approach considered

here differs in the following way. First, we do not rely on the universal property of weak equivalences and the Rezk completion. Instead, we do the lifting in terms of weak equivalences. Second, whereas they relied on displayed categories to lift the equivalence on hom-categories, we now rely on displayed bicategories to formulate displayed biadjunctions/universal arrows.

Instead of lifting via *abstract Rezk completions*, one can also try to use a *concrete implementation* of the Rezk completion, either via the presheaf construction, or via higher inductive types. Wullaert et al. [33] furthermore sketched how the presheaf construction inherits a monoidal structure by considering the Day convolution structure on the category of presheaves. Van der Weide [25] extended both implementations of the Rezk completion to the setting of enriched categories. However, as opposed to the monoidal case, some modifications have been made. First, the notion of fully faithfulness, and hence weak equivalence, has to be adapted to take the enrichment into account. Nonetheless, Van der Weide [25] showed that such weak equivalences imply an equivalence on the *enriched functor categories*, similar to the (non-enriched) weak equivalences. Second, as opposed to the monoidal Rezk completion, he considered enriched presheaves, as opposed to **Set**-valued presheaves. Note that even though the enriched presheaf construction does a priori not correspond to the ordinary presheaf construction, the HIT construction for the enriched Rezk completion shows that the underlying category of the enriched Rezk completion coincides with the *ordinary* Rezk completion.

6 Conclusion

Conclusion. We presented a modular approach for lifting the Rezk completion from categories to structured categories, based on displayed universal arrows (Definition 23). We illustrated this approach, and its modularity, by lifting the Rezk completion from categories to topoi (Corollary 52). In particular, we’ve observed that the lifting of the Rezk completion essentially reduces to the preservation and reflection properties of weak equivalences. Moreover, the approach presented here is independent of an explicit construction, but follows from its characterization in terms of weak equivalences. However, while elementary topoi already illustrate that various structures do lift, Lemma 39 illustrates that for certain structures we need additional assumptions in our logical foundation. Lastly, topoi are an important notion in categorical logic and semantics since they allow for an interpretation of various type formers and logical constructs. Hence, our results imply that the Rezk completion preserves these operations, which means that the Rezk completion can be used for the construction of univalent models.

Future Directions. There are multiple ways in which this work can be extended. First, in [1, Constr. 5.13], it is proven that the presheaf construction for the Rezk completion of (1-)categories induces a similar result when passing to (locally univalent) bicategories. Hence, there is the question of how the results presented here generalize to higher categories and topoi (see e.g., [21, 32]).

Second, our results imply that the tripos-to-topos construction (Example 10) can be extended to a *tripos-to-univalent-topos construction*. Consequently, this can be used for the construction of realizability topoi. Specifically, if we start with a realizability topos, our results imply that its Rezk completion is topos. However, it remains to be seen whether the Rezk completion is itself a realizability topos, i.e., by verifying that it satisfies the assumptions that characterize realizability topoi axiomatically ([10, Thm. 4.6]).

Third, there are more categorical structures one can consider. One such example is *locally cartesian closedness*, used to interpret Π -types. We expect this to follow from the results about cartesian closed categories presented above and the fact that taking slice categories is well-behaved with respect to weak equivalences. Furthermore, the approach presented also works for other categorical structures. For example, abelian categories can be characterized in an *enrichment-free* way using (finitary) universal constructions, whose preservation properties should follow from the above.

References

- 1 Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccolò Veltri, and Niels van der Weide. Bicategories in univalent foundations. *Math. Struct. Comput. Sci.*, 31(10):1232–1269, 2021. doi:10.1017/S0960129522000032.
- 2 Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Math. Struct. Comput. Sci.*, 25(5):1010–1039, 2015. doi:10.1017/S0960129514000486.
- 3 Benedikt Ahrens and Peter LeFanu Lumsdaine. Displayed Categories. *Logical Methods in Computer Science*, Volume 15, Issue 1, March 2019. doi:10.23638/LMCS-15(1:20)2019.
- 4 Benedikt Ahrens, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. Categorical Structures for Type Theory in Univalent Foundations. *Logical Methods in Computer Science*, 14(3), 2018. doi:10.23638/LMCS-14(3:18)2018.
- 5 Benedikt Ahrens, Paige North, Michael Shulman, and Dimitris Tsementzis. The Univalence Principle. *Memoirs of the American Mathematical Society*, 305(1541):vii+176, 2025. doi:10.1090/memo/1541.
- 6 Francis Borceux. *Handbook of Categorical Algebra: Volume 1, Basic category theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1994.
- 7 Ulrik Buchholtz, Floris van Doorn, and Egbert Rijke. Higher Groups in Homotopy Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '18*, pages 205–214, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3209108.3209150.
- 8 Tom de Jong and Martín Hötzel Escardó. Examples and counterexamples of injective types. *arXiv preprint arXiv:2601.12536*, 2026. doi:10.48550/arXiv.2601.12536.
- 9 M. P. Fourman and A. Ščedrov. The “world’s simplest axiom of choice” fails. *manuscripta mathematica*, 38(3):325–332, 1982. doi:10.1007/BF01170929.
- 10 Jonas Frey. Characterizing partitioned assemblies and realizability toposes. *Journal of Pure and Applied Algebra*, 223(5):2000–2014, 2019. doi:10.1016/j.jpaa.2018.08.012.
- 11 Denis Higgs. Injectivity in the Topos of Complete Heyting Algebra Valued Sets. *Canadian Journal of Mathematics*, 36(3):550–568, 1984. doi:10.4153/CJM-1984-034-4.
- 12 J. Martin E. Hyland. Classical lambda calculus in modern dress. *Mathematical Structures in Computer Science*, 27(5):762–781, 2017. doi:10.1017/S0960129515000377.
- 13 John Martin Elliott Hyland, Peter Tennant Johnstone, and Andrew Mawdesley Pitts. Tripos theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 88(2):205–232, 1980. doi:10.1017/S0305004100057534.
- 14 Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after Voevodsky). *Journal of the European Mathematical Society*, 23(6):2071–2126, 2021. doi:10.4171/JEMS/1050.
- 15 F. William Lawvere. Metric spaces, generalized logic, and closed categories. *Rendiconti del Seminario Matematico e Fisico di Milano*, 43(1):135–166, 1973. doi:10.1007/BF02924844.
- 16 Daniel R. Licata and Eric Finster. Eilenberg-MacLane spaces in homotopy type theory. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2603088.2603153.

- 17 Maria Emilia Maietti. Joyal’s arithmetic universe as list-arithmetic pretopos. *Theory and Applications of Categories*, 24, January 2010. doi:10.70930/tac/qee78enb.
- 18 Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1):189–218, 2000. doi:10.1016/S0168-0072(00)00012-9.
- 19 Andrew Mawdesley Pitts. Tripos Theory in Retrospect. *Math. Struct. Comput. Sci.*, 12(3):265–279, 2002. doi:10.1017/S096012950200364X.
- 20 Nima Rasekh, Niels van der Weide, Benedikt Ahrens, and Paige Randall North. Insights from Univalent Foundations: A Case Study Using Double Categories. In Jörg Endrullis and Sylvain Schmitz, editors, *33rd EACSL Annual Conference on Computer Science Logic (CSL 2025)*, volume 326 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 45:1–45:18, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2025.45.
- 21 Ross Street. Two-dimensional sheaf theory. *Journal of Pure and Applied Algebra*, 23(3):251–270, 1982. doi:10.1016/0022-4049(82)90101-3.
- 22 The Rocq Development Team. The Rocq Prover, April 2025. doi:10.5281/zenodo.15149629.
- 23 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- 24 Arnoud van der Leer, Kobe Wullaert, and Benedikt Ahrens. Scott’s Representation Theorem and the Univalent Karoubi Envelope. In Yannick Forster and Chantal Keller, editors, *16th International Conference on Interactive Theorem Proving, ITP 2025, Reykjavik, Iceland, September 28 - October 1, 2025*, volume 352 of *LIPIcs*, pages 33:1–33:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ITP.2025.33.
- 25 Niels van der Weide. Univalent Enriched Categories and the Enriched Rezk Completion. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, volume 299 of *LIPIcs*, pages 4:1–4:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.FSCD.2024.4.
- 26 Niels van der Weide. The Formal Theory of Monads, Univalently. *Logical Methods in Computer Science*, Volume 21, Issue 1, February 2025. doi:10.46298/lmcs-21(1:16)2025.
- 27 Niels van der Weide. The internal languages of univalent categories. In *2025 40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 112–126, 2025. doi:10.1109/LICS65433.2025.00016.
- 28 Niels van der Weide, Nima Rasekh, Benedikt Ahrens, and Paige Randall North. Univalent Double Categories. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024*, pages 246–259, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3636501.3636955.
- 29 Jaap van Oosten. *Realizability: An Introduction to Its Categorical Side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*. Elsevier B. V., Amsterdam, 2008.
- 30 Niccolò Veltri and Niels van der Weide. Constructing Higher Inductive Types as Groupoid Quotients. *Logical Methods in Computer Science*, Volume 17, Issue 2, April 2021. doi:10.23638/LMCS-17(2:8)2021.
- 31 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. Available at <http://unimath.github.io/UniMath/>, 2024. doi:10.5281/zenodo.13828995.
- 32 Mark Weber. Yoneda Structures from 2-toposes. *Applied Categorical Structures*, 15(3):259–323, 2007. doi:10.1007/s10485-007-9079-2.
- 33 Kobe Wullaert, Ralph Matthes, and Benedikt Ahrens. Univalent Monoidal Categories. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France*, volume 269 of *LIPIcs*, pages 15:1–15:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.TYPES.2022.15.

Kleisli Categories with Display Maps

Moana Jubert 

Inria Paris, Université Paris-Cité, IRIF, CNRS, Paris, France

Abstract

In this paper, we show that the Kleisli category $\mathcal{C}_T$ for a monad $T : \mathcal{C} \longrightarrow \mathcal{C}$ is sometimes a structured display map category, meaning that it provides a minimal categorical setting for dependent type theory. If the underlying category $\mathcal{C}$ is itself a (structured) display map category, we prove that under mild conditions the canonical functor $F_T : \mathcal{C} \longrightarrow \mathcal{C}_T$ induces a bijection between the contexts, types, and terms of both $\mathcal{C}$ and $\mathcal{C}_T$, thus establishing a kind of “collapsing” result. Finally, we provide several examples and nonexamples, most notably showing that Δ^{op} equipped with the class of surjective maps is a (structured) display map category.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Type theory; Theory of computation $\rightarrow$ Categorical semantics

Keywords and phrases Kleisli categories, Structured display map categories, Dependent type theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.9

Acknowledgements I owe a great debt of gratitude to the scientific diversity of my laboratory, the Institut de Recherche en Informatique Fondamentale (IRIF).

1 Introduction

This paper is about constructing new categorical models of dependent type theories from well-behaved monads. It emerged from one of the author’s ongoing lines of inquiry into defining a “language of profunctors”. The path presented here proved unpromising for this specific goal, most notably because of the negative result of Example 4.12. In addition, other successful lines of inquiry already exist in the literature, such as *virtual equipment type theory* by Max New and Daniel Licata [24], further developed by Hayato Nasu [23]. As a result, we really discuss profunctors only to explain the reasoning that led to this paper, and its core material may be read independently. We believe nonetheless that the community will find the results we collected along the way useful and interesting.

A profunctor $\mathcal{A} \multimap \mathcal{B}$ from a small category $\mathcal{A}$ to a small category $\mathcal{B}$ consists of the data of a bifunctor $\mathcal{B}^{\text{op}} \times \mathcal{A} \longrightarrow \mathbf{Set}$. Profunctors compose, and they assemble into a bicategory **Prof**. There is a canonical, covariant embedding $\mathbf{Cat} \hookrightarrow \mathbf{Prof}$ which, to some extent, may be read as “decorating” **Cat** with the extra information of profunctors, allowing one to do “more” reasoning on small categories. This way of presenting profunctors motivated, among other things, the abstract notion of *proarrow equipment* [30] for which **Cat** equipped with profunctors is the primary example.

We now say that a profunctor is *representable* if it is the image of a functor under the canonical embedding. Presheaves over a small category $\mathcal{C}$ are special cases of profunctors $* \multimap \mathcal{C}$ from the one-object category $*$ to the category $\mathcal{C}$, and accordingly a presheaf is representable exactly when it is so as a profunctor. Many concepts – if not *all* – may be reformulated in terms of representable presheaves and, by extension, representable profunctors. This prompted us to carefully study the interaction between general and representable profunctors so as to come up with a sort of *calculus of profunctors*¹ – a “language” for which **Prof**, equipped with the class of all representable profunctors, would be the model.

¹ Part of this calculus has already been worked out by Jean Bénabou in [5], where profunctors are instead called *distributors*.



© Moana Jubert;

licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 9; pp. 9:1–9:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

If such a language is possible, then **Prof** should fit into any one of the categorical interpretations² of type theory, modulo some changes. One possibility introduced in [1, §3.1.1] is that of *structured display map categories*, which are simply defined as categories equipped with a class of arrows stable under pullbacks, called *display maps*. This differs from the well-known notion of *display map category* in that we do not ask the class of display maps to be *replete*, *i.e.*, invariant under isomorphisms. It is natural to ask whether the class of representable profunctors could play the role of display maps for **Prof**.

Of course, **Prof** is a bicategory, so we would need to tweak the categorical interpretations in some way. It is worth mentioning the work of Benedikt Ahrens et al. in this direction [2], where they develop the foundations of *comprehension bicategories*. Let us set the dimensional issues aside for a moment to look at **Prof** from afar. If we squint our eyes, **Prof** looks like the Kleisli (bi)category of some (pseudo)monad on **Cat** sending any small category $\mathcal{C}$ to its large category of presheaves $\widehat{\mathcal{C}}$. Representable profunctors are the image of arrows under the canonical embedding $\mathbf{Cat} \hookrightarrow \mathbf{Prof}$ induced by this monad. Then we ask whether the Kleisli (bi)category **Prof** equipped with the arrows in the image of the canonical embedding is a structured display map category. Obviously, such a monad could not possibly exist because of size issues. But *monads need not be endofunctors* [3], and the functor $\mathcal{C} \mapsto \widehat{\mathcal{C}}$ from the category of small categories **Cat** into the category of locally small categories **CAT** is a canonical example of a *relative pseudomonad* [15] for which **Prof** is the associated Kleisli bicategory. Now, if we reformulate our reasoning in 1-categorical terms, given a (true) monad (T, η, μ) on a category $\mathcal{C}$, its associated Kleisli category $\mathcal{C}_T$ always comes with a canonical functor $F_T : \mathcal{C} \rightarrow \mathcal{C}_T$, and we can ask whether the category $\mathcal{C}_T$ equipped with the arrows in the image of F_T (they are said, by analogy, to be *representable*) is a structured display map category. In this paper, we give sufficient conditions to answer this last question positively.

These sufficient conditions are adapted from an article by Dominique Bourn [7], who studied the existence of pullbacks in Kleisli categories in order to understand the relationship between their internal categories and the so-called *T-categories* in the sense of Albert Burroni [8]. Just as in Dominique Bourn’s work, we see that *cartesian monads* (*cf.* Definition 2.5) are exceptionally well-behaved, in that when the base category has all pullbacks, the Kleisli category is always a structured display map category. We go one step further, however, by considering the case where the base category itself forms a structured display map category. This level of precision allows us to consider more examples of Kleisli categories that are structured display map categories for a proper subclass of the representable arrows. As a notable example, we explain in Example 4.8 how the category Δ^{op} (opposite of the simplex category Δ) is a Kleisli category, as well as a (structured) display map category when equipped with the class of surjective maps in Δ .

Our Contributions

We use and extend (Proposition 2.9) the results of Dominique Bourn to see that some Kleisli categories are in fact structured display map categories. If both the base category and its associated Kleisli category are structured display map categories, then we prove that, under mild conditions, there is an adjunction between the two in the 2-category of structured display map categories (Theorem 3.10, Corollary 3.11). We also show that under some conditions the contexts, types, and terms of the two are in bijection (Corollary 3.15). Finally, we provide an opinionated list of examples (*e.g.*, Example 4.8).

² We can mention, for example, *contextual categories* [9], *categories with families* [14], and *comprehension categories* [19]. They are all comparable to one another, as shown in [1].

Outline of This Paper

In Section 2, we recall Kleisli categories and study their pullbacks. We prove results about structured display map categories in Section 3, and, finally, in Section 4, we walk through several examples.

2 Kleisli Categories

Let (T, η, μ) be a monad on an arbitrary category $\mathcal{C}$.

- **Definition 2.1.** We write $\mathcal{C}_T$ for the associated Kleisli category of T where
- objects $X, Y, \dots$ of $\mathcal{C}_T$ are the same as those of $\mathcal{C}$;
 - an arrow of $\mathcal{C}_T$, which we write $\sigma : X \dashrightarrow Y$ with a marking, is the data of an arrow $\sigma : X \longrightarrow TY$ in $\mathcal{C}$.

Identities and composition are defined using the unit η and multiplication μ of T . There is a canonical pair of adjoint functors

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{F_T} & \mathcal{C}_T \\ & \perp & \\ & \xleftarrow{U_T} & \end{array}$$

where F_T is the identity on objects and sends any arrow $f : A \longrightarrow B$ of $\mathcal{C}$ to $\eta_B \circ f : A \dashrightarrow B$, and U_T coincides with T on objects and sends any arrow $\sigma : X \dashrightarrow Y$ of $\mathcal{C}_T$ to the arrow $\mu_Y \circ T\sigma : TX \longrightarrow TY$. Moreover, we have $U_T \circ F_T = T$ by definition.

- **Definition 2.2.** An arrow of $\mathcal{C}_T$ lying in the image of F_T is said to be representable. We draw representable arrows $F_T(f) : A \dashrightarrow B$ without the marking, and write $\text{Repr}(T) := F_T(\text{Arr}(\mathcal{C}))$ for the class of all representable arrows of $\mathcal{C}_T$.

Of course, the representing arrow of a representable arrow need not be unique. It is unique precisely when F_T is a faithful functor, which is equivalent to the unit η being monomorphic, i.e., $\eta \circ f = \eta \circ g$ implies $f = g$.

- **Example 2.3.** Consider the monad $\mathcal{P} : \mathbf{Set} \longrightarrow \mathbf{Set}$ on $\mathbf{Set}$ sending a set A to its set of subsets $\mathcal{P}(A)$. Its associated Kleisli category is called **Rel**, and the data of an arrow $R : A \dashrightarrow B$ is exactly the data of a relation $R \subseteq A \times B$. Representable arrows are exactly the (total) functional relations. The unit is monomorphic, and therefore the functor $F_{\mathcal{P}} : \mathbf{Set} \longrightarrow \mathbf{Rel}$ is faithful (but trivially not full).

$\mathcal{C}_T$ is, up to embedding, the full subcategory of free T -algebras of the category $\mathcal{C}^T$ of algebras over T – the so-called *Eilenberg-Moore category*. While $\mathcal{C}^T$ may often enjoy good properties such as the existence of limits and colimits, this is rarely the case for $\mathcal{C}_T$ (see, e.g., [6, §4.3] for a discussion on the existence of limits and colimits in $\mathcal{C}^T$). In [28], the author gives sufficient conditions for $\mathcal{C}_T$ to be complete or cocomplete, depending on whether the obvious fully faithful inclusion $I : \mathcal{C}_T \hookrightarrow \mathcal{C}^T$ is a left or right adjoint. There is also a canonical functor $U^T : \mathcal{C}^T \longrightarrow \mathcal{C}$, and U_T factors through it via I , as depicted below

$$\begin{array}{ccc} \mathcal{C}_T & \xrightarrow{I} & \mathcal{C}^T \\ & \searrow U_T & \swarrow U^T \\ & \mathcal{C} & \end{array}$$

► **Proposition 2.4.** *The functor U_T both preserves and reflects limits.*

Proof. U_T preserves limits because it is a right adjoint. It is well known that the functor $U^T : \mathcal{C}^T \rightarrow \mathcal{C}$ creates limits and so, in particular, reflects them (see, *e.g.*, [27, Theorem 5.6.5] or [21, Exercise VI.2.2]). Now, if a cone in $\mathcal{C}_T$ is sent to a limit cone in $\mathcal{C}$ via $U_T = U^T \circ I$, then it is a limit cone in $\mathcal{C}^T$. But $\mathcal{C}_T$ is a full subcategory of $\mathcal{C}^T$, so it must be a limit cone in $\mathcal{C}_T$ as well. This proves that U_T reflects limits. ◀

Fix a subclass $\mathcal{A} \subseteq \text{Arr}(\mathcal{C})$ of arrows of $\mathcal{C}$. We want to study the stability of $F_T(\mathcal{A})$ – which is a subclass of the representable arrows of $\mathcal{C}_T$ – under pullbacks. Proposition 2.4 above provides the most general (and therefore least convenient) criterion. Suppose we have the following cospan in $\mathcal{C}_T$

$$\begin{array}{ccc} & A & \\ & \downarrow F_T(f) & \\ Y & \xrightarrow{\sigma} & B \end{array}$$

where the vertical arrow is representable and belongs to $F_T(\mathcal{A})$. Assuming $\mathcal{C}$ has enough pullbacks, we may look at the pullback of its image under U_T

$$\begin{array}{ccccc} TA \times_{TB} TY & \xrightarrow{\psi} & TA & & \\ \downarrow \phi & \lrcorner & \downarrow Tf & & \\ TY & \xrightarrow{U_T(\sigma)} & TB & & \end{array} \quad (1)$$

If we can prove that $TA \times_{TB} TY \cong TX$ for some $X \in \mathcal{C}$, $\phi \cong Tp$ for some $p : X \rightarrow Y$ belonging to $\mathcal{A}$, and $\psi \cong U_T(\tau)$ for some $\tau : X \rightarrow A$ in $\mathcal{C}_T$, then, since U_T reflects pullbacks by Proposition 2.4, the square in $\mathcal{C}_T$ below must be a pullback as well

$$\begin{array}{ccc} X & \xrightarrow{\tau} & A \\ \downarrow F_T(p) & \lrcorner & \downarrow F_T(f) \\ Y & \xrightarrow{\sigma} & B \end{array}$$

and both vertical arrows belong to $F_T(\mathcal{A})$. Instead of (1) we may consider the following simpler pullback

$$\begin{array}{ccc} Y \times_{TB} TA & \xrightarrow{\quad} & TA \\ \downarrow \sigma^* Tf & \lrcorner & \downarrow Tf \\ Y & \xrightarrow{\sigma} & TB \end{array}$$

and hope that $F_T(\sigma^* Tf)$ is somehow related to the pullback of $F_T(f)$ along σ in $\mathcal{C}_T$ when it exists. The rest of this section is dedicated to making this idea precise. We first present a simple criterion, due to Dominique Bourn, for a class $\mathcal{A} \subseteq \text{Arr}(\mathcal{C})$ of arrows to be such that $F_T(\mathcal{A})$ is stable under pullbacks (Theorem 2.7).

► **Definition 2.5.** A natural transformation is said to be *cartesian* when the naturality squares are all pullbacks. Similarly, we say that a functor is *cartesian* when it sends pullback squares to pullback squares. A monad is *cartesian* when all of its components are cartesian³.

► **Example 2.6.** The *partiality monad*

$$\begin{aligned} P : \mathbf{Set} &\longrightarrow \mathbf{Set} \\ X &\longmapsto X + * \end{aligned}$$

is cartesian. Its Kleisli category consists of sets and partial functions between them, and the representable arrows are exactly the total functions. Note that **Set** has all pullbacks, and so P preserves them.

► **Theorem 2.7** (Dominique Bourn [7, Proposition 3.12]). Let (T, η, μ) be a monad on a category $\mathcal{C}$. Suppose that we have a class $\mathcal{A} \subseteq \text{Arr}(\mathcal{C})$ of arrows of $\mathcal{C}$ such that the following conditions hold:

1. For every $f : A \longrightarrow B$ belonging to $\mathcal{A}$ and $\sigma : Y \longrightarrow TB$, the pullback

$$\begin{array}{ccc} Y \times_{TB} TA & \xrightarrow{\quad\quad\quad} & TA \\ \sigma^* T f \downarrow & \lrcorner & \downarrow T f \\ Y & \xrightarrow{\quad\quad\quad \sigma \quad\quad\quad} & TB \end{array} \quad (2)$$

does exist in $\mathcal{C}$, is preserved by T , and is such that $\sigma^* T f$ belongs to $\mathcal{A}$;

2. μ is cartesian.

Then the class $F_T(\mathcal{A})$ is stable under pullbacks.

Proof. We reproduce the proof from Dominique Bourn. Using both hypotheses, we can paste together the following pullbacks:

$$\begin{array}{ccccc} T(Y \times_{TB} TA) & \xrightarrow{\quad\quad\quad} & T^2 A & \xrightarrow{\mu_A} & TA \\ T(\sigma^* T f) \downarrow & \lrcorner & \downarrow T^2 f & \lrcorner & \downarrow T f \\ TY & \xrightarrow{\quad\quad\quad T(\sigma) \quad\quad\quad} & T^2 B & \xrightarrow{\mu_B} & TB \end{array}$$

The outer rectangle is a pullback by the pasting law. But it is also the image of the commutative square in $\mathcal{C}_T$

$$\begin{array}{ccc} Y \times_{TB} TA & \xrightarrow{\quad\quad\quad} & A \\ F_T(\sigma^* T f) \downarrow & & \downarrow F_T(f) \\ Y & \xrightarrow{\quad\quad\quad \sigma \quad\quad\quad} & B \end{array}$$

via the functor $U_T : \mathcal{C}_T \longrightarrow \mathcal{C}$. Now, U_T reflects pullbacks by Proposition 2.4, hence the desired result. ◀

³ Some authors additionally require that the underlying category has all pullbacks.

Dominique Bourn's criterion is not only sufficient but also necessary when the monad is cartesian. If a representable pullback exists, then it *must* be of the form given by the pullback in diagram (2). This is a new result highlighting the robustness of this construction. We first prove the proposition below, showing that the representable arrow $F_T(\sigma^*Tf)$ is always a retract of the actual pullback in $\mathcal{C}_T$, provided it exists and is itself representable.

► **Proposition 2.8.** *Suppose we have a pullback square in $\mathcal{C}_T$ as below*

$$\begin{array}{ccc} X & \xrightarrow{\quad} & A \\ F_T(p) \downarrow & \lrcorner & \downarrow F_T(f) \\ Y & \xrightarrow[\sigma]{} & B \end{array}$$

where the left and right vertical arrows are representable. If the following pullback in $\mathcal{C}$ exists

$$\begin{array}{ccc} Y \times_{TB} TA & \dashrightarrow & TA \\ \sigma^*Tf \downarrow & \lrcorner & \downarrow Tf \\ Y & \xrightarrow[\sigma]{} & TB \end{array}$$

then the arrow $F_T(\sigma^*Tf)$ is a retract of $F_T(p)$ in $\mathcal{C}_T/Y$.

Proof. Since U_T preserves limits, it sends the first pullback square to a pullback square in $\mathcal{C}$

$$\begin{array}{ccc} TX & \longrightarrow & TA \\ Tp \downarrow & \lrcorner & \downarrow Tf \\ TY & \xrightarrow[U_T(\sigma)]{} & TB \end{array}$$

We now put together the two diagrams above and insert the naturality square of the unit η

$$\begin{array}{ccccc} & & Y \times_{TB} TA & & \\ & \searrow & \lrcorner & \searrow & \\ & & X & \xrightarrow{\eta_X} & TX & \longrightarrow & TA \\ & & \downarrow p & & \downarrow Tp & \lrcorner & \downarrow Tf \\ & & Y & \xrightarrow{\eta_Y} & TY & \xrightarrow[U_T(\sigma)]{} & TB \\ & \nearrow & \sigma & \nearrow & & & \end{array} \quad (3)$$

The bottom triangle commutes by drawing the appropriate string diagram. By the universal property of the pullback, the commutation of the outer rectangle implies the existence of a unique arrow $r : Y \times_{TB} TA \longrightarrow TX$. Similarly, the commutation of the inner rectangle implies the existence of a unique arrow $s : X \longrightarrow Y \times_{TB} TA$. Both arrows fit into the

following commutative diagram in $\mathcal{C}$

$$\begin{array}{ccccc}
 & & \eta_X & & \\
 & \nearrow & & \searrow & \\
 X & \xrightarrow{s} & Y \times_{TB} TA & \xrightarrow{r} & TX \\
 \downarrow p & & \downarrow \sigma^* T f & & \downarrow T p \\
 Y & \xlongequal{\quad} & Y & \xrightarrow{\eta_Y} & TY
 \end{array}$$

The top triangle commutes by the universal property of the pullback and uniqueness of arrows. We get that the diagram in $\mathcal{C}_T$ below commutes, using the fact that $Tp \circ r$ in $\mathcal{C}$ is equal to $F_T(p) \circ r$ in $\mathcal{C}_T$, as can be checked by drawing the appropriate string diagram

$$\begin{array}{ccccc}
 X & \xrightarrow{F_T(s)} & Y \times_{TB} TA & \xrightarrow{r} & X \\
 \downarrow F_T(p) & & \downarrow F_T(\sigma^* T f) & & \downarrow F_T(p) \\
 Y & \xlongequal{\quad} & Y & \xlongequal{\quad} & Y
 \end{array}$$

We are left to show that $r \circ F_T(s) = 1_X$ in $\mathcal{C}_T$. This unfolds in $\mathcal{C}$ as

$$\mu_X \circ Tr \circ \eta_{Y \times_{TB} TA} \circ s = \eta_X$$

This is true by rewriting and diagram chasing. $\blacktriangleleft$

With the additional hypothesis that the unit η of the monad is *cartesian*, we can finally show that this retract collapses and that $F_T(\sigma^* T f)$ is indeed isomorphic to the pullback. This is the statement of Proposition 2.9.

► **Proposition 2.9.** *The arrows $\sigma^* T f : Y \times_{TB} TA \longrightarrow Y$ and $p : X \longrightarrow Y$ in the diagram (3) are isomorphic when η is cartesian.*

Proof. The left inner square in the diagram (3) is a pullback because η is cartesian, and the right inner square is also a pullback by hypothesis. The whole inner rectangle must be a pullback by the pasting law, and by uniqueness of pullbacks, it must be isomorphic to the outer rectangle, *i.e.*, $\sigma^* T f : Y \times_{TB} TA \longrightarrow Y$ is isomorphic to $p : X \longrightarrow Y$. $\blacktriangleleft$

To conclude this section, we discuss two general examples.

► **Example 2.10.** Let $\mathcal{C}$ be a category with all pullbacks and fix $\mathcal{A} := \text{Arr}(\mathcal{C})$. Cartesian monads over $\mathcal{C}$ are the best of all possible worlds, since they trivially satisfy the hypotheses of Theorem 2.7. As a result, $F_T(\mathcal{A}) =: \text{Repr}(T)$ is automatically stable under pullbacks. When applied to Example 2.6, we recover the fact that the pullback of a total function along a partial function is again a total function⁴. Many more examples will be covered in Section 4.

► **Example 2.11.** If T is a cartesian monad and $\mathcal{A} \subseteq \text{Arr}(\mathcal{C})$ is a class of arrows of $\mathcal{C}$ stable under pullbacks such that $T(\mathcal{A}) \subseteq \mathcal{A}$, then the hypotheses of Theorem 2.7 are also satisfied. For example, again with the partiality monad, the class **Mono** of injective maps in **Set** is stable under pullbacks and we indeed have $P(\text{Mono}) \subseteq \text{Mono}$. We conclude that the class $F_P(\text{Mono})$ of arrows of **Set**_P is stable under pullbacks.

⁴ Note that the Kleisli category **Set**_P of the partiality monad has, in fact, all pullbacks.

3 Structured Display Map Categories

3.1 Definition and Properties

► **Definition 3.1** (Benedikt Ahrens et al. [1, Definition 18]). Let $\mathcal{C}$ be an arbitrary category and $\mathcal{A} \subseteq \text{Arr}(\mathcal{C})$ a class of arrows of $\mathcal{C}$. The pair $(\mathcal{C}, \mathcal{A})$ is a structured display map category if $\mathcal{A}$ is stable under pullbacks, meaning that for any arrows $f : A \longrightarrow B \in \mathcal{A}$ and $g : Y \longrightarrow B \in \mathcal{C}$ there exists a pullback square

$$\begin{array}{ccc} X & \xrightarrow{\quad} & A \\ p \downarrow & \lrcorner & \downarrow f \\ Y & \xrightarrow{\quad g \quad} & B \end{array}$$

such that $p : X \longrightarrow Y \in \mathcal{A}$. The arrows in $\mathcal{A}$ are called display maps.

► **Remark 3.2.** To say that $\mathcal{A}$ is stable under pullbacks is different from saying that *every* pullback of an arrow of $\mathcal{A}$ belongs to $\mathcal{A}$. We merely ask that among the possible choices of pullbacks up to isomorphism, at least one of them is in $\mathcal{A}$.

If we ask for $\mathcal{A}$ to contain every pullback, we get the stronger notion of *display map category* – without the “structured” – which is much more familiar to type theorists. A display map category is a structured display map category such that the class of arrows is invariant under isomorphisms (the class is said to be *replete*). Display map categories are more restrictive and therefore better-behaved.

We chose the more general setting of structured display map categories because it does not require us later to think too much about the isomorphisms inside a Kleisli category, as they are too often mysterious. We would like to point out that a structured display map category with no nontrivial isomorphisms is automatically a display map category by definition. This is, for example, the case of Δ^{op} in Example 4.8.

► **Remark 3.3.** It is, of course, always possible to saturate the class $\mathcal{A}$ of a structured display map category $(\mathcal{C}, \mathcal{A})$ so that it is invariant under isomorphisms. This defines a 2-functor $\mathbf{sDMC} \longrightarrow \mathbf{DMC}$ into the full sub-2-category of “mere” display map categories, and this 2-functor is in fact left 2-adjoint to the inclusion of $\mathbf{DMC}$ into $\mathbf{sDMC}$. We also mention that structured display map categories are exactly the *full* comprehension categories where the comprehension is a “true” category inclusion, in that there is an *isomorphism* of 2-categories between $\mathbf{sDMC}$ and some 2-category of full comprehension categories. These two observations may be found in more detail in [1, §3.1.1].

► **Example 3.4.** Given a dependent type theory $\mathbb{T}$, we may construct a structured display map category $(\mathcal{C}_{\mathbb{T}}, \mathcal{A}_{\mathbb{T}})$ in the following way: the objects of $\mathcal{C}_{\mathbb{T}}$ are the valid contexts $\Gamma, \Delta, \Xi, \dots$ of $\mathbb{T}$, and an arrow $\sigma : \Delta \longrightarrow \Gamma$ between contexts is a *substitution* between contexts (see, e.g., the definition of *context morphism* in [18, §2.2.2]). We will keep the terminology “substitution” specific to type theory, *i.e.*, it is an n -tuple of terms satisfying some properties. Otherwise, we will use the more generic term “context morphism” for a general structured display map category. If we have a dependent type $\Gamma \vdash A$ of $\mathbb{T}$, there is a canonical substitution

$$p_A : \Gamma, x : A \longrightarrow \Gamma$$

from the context Γ extended with a fresh variable x of type A to Γ . We write $\Gamma.A$ rather than $\Gamma, x : A$, and $p_A : \Gamma.A \longrightarrow \Gamma$ is essentially a projection (which “forgets” the variable of

type A). We now define $\mathcal{A}_{\mathbb{T}}$ to be the collection of all such canonical substitutions. This class is indeed stable under pullbacks, as dependent types are stable under substitution

$$\begin{array}{ccc}
 \Delta.A[\sigma] & \dashrightarrow & \Gamma.A \\
 \downarrow \sigma^* p_A = p_{A[\sigma]} & \lrcorner & \downarrow p_A \\
 \Delta & \xrightarrow{\sigma} & \Gamma
 \end{array}$$

The above example shows how a structured display map category $(\mathcal{C}, \mathcal{A})$ may be seen as a minimal categorical model of “core” dependent type theory, where by “core” we mean the pure structural aspects of type dependency, prior to any type constructions. The contexts of the type theory are interpreted as objects of $\mathcal{C}$, the substitutions between contexts are interpreted as arrows of $\mathcal{C}$, and the dependent types are interpreted as display maps – a display map $p_A : \Gamma, x : A \longrightarrow \Gamma$ interprets the judgment $\Gamma \vdash A$ in the theory. To have, e.g., Π -types or Σ -types, one would need additional structure on a structured display map category, just as is the case in other minimal settings like comprehension categories. We will discuss contexts, types, and terms of structured display map categories in Section 3.2 below.

► **Corollary 3.5.** *Suppose we have a monad (T, η, μ) on a category $\mathcal{C}$ and a class $\mathcal{A} \subseteq \text{Arr}(\mathcal{C})$. Under the hypotheses of Theorem 2.7, the pair $(\mathcal{C}_T, F_T(\mathcal{A}))$ is a structured display map category.*

Proof. By definition. ◀

Structured display map categories come with their own flavor of morphism. We will see that when $(\mathcal{C}, \mathcal{A})$ and $(\mathcal{C}_T, F_T(\mathcal{A}))$ are structured display map categories, then under mild conditions the canonical functors $F_T : \mathcal{C} \longrightarrow \mathcal{C}_T$ and $U_T : \mathcal{C}_T \longrightarrow \mathcal{C}$ are instances of such morphisms.

► **Definition 3.6** (Benedikt Ahrens et al. [1, Definition 21]). *A morphism $F : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{D}, \mathcal{B})$ of structured display map categories is the data of a functor $F : \mathcal{C} \longrightarrow \mathcal{D}$ such that*

- $F(\mathcal{A}) \subseteq \mathcal{B}$;
- *pullbacks along arrows of $\mathcal{A}$ are sent to pullbacks along arrows of $\mathcal{B}$.*

*Structured display map categories assemble into a 2-category **sDMC** where the 2-cells are all the natural transformations.*

Let us now fix a structured display map category $(\mathcal{C}, \mathcal{A})$.

► **Proposition 3.7.** *If (T, η, μ) is a monad on $\mathcal{C}$ such that μ is cartesian and*

$$T : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{C}, \mathcal{A})$$

is a morphism of structured display map categories, then the pair $(\mathcal{C}_T, F_T(\mathcal{A}))$ is a structured display map category.

Proof. Unfolding the definitions, it is easy to check that the hypotheses of Theorem 2.7 are satisfied (see Example 2.11). ◀

► **Example 3.8.** If T is a cartesian monad on $\mathcal{C}$ such that $T(\mathcal{A}) \subseteq \mathcal{A}$, then

$$T : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{C}, \mathcal{A})$$

is a morphism of structured display map categories, and by applying the above proposition, we get that $(\mathcal{C}_T, F_T(\mathcal{A}))$ is a structured display map category.

9:10 Kleisli Categories with Display Maps

► **Proposition 3.9.** *Suppose that (T, η, μ) is a monad on $\mathcal{C}$ such that $(\mathcal{C}_T, F_T(\mathcal{A}))$ is a structured display map category. If $T(\mathcal{A}) \subseteq \mathcal{A}$, then*

$$U_T : (\mathcal{C}_T, F_T(\mathcal{A})) \longrightarrow (\mathcal{C}, \mathcal{A})$$

is a morphism of structured display map categories.

Proof. We have that $U_T \circ F_T = T$; therefore, $U_T(F_T(\mathcal{A})) = T(\mathcal{A}) \subseteq \mathcal{A}$ by hypothesis. Now, U_T preserves limits by Proposition 2.4 and therefore sends pullbacks of arrows in $F_T(\mathcal{A})$ to pullbacks of arrows in $\mathcal{A}$. ◀

► **Theorem 3.10.** *Suppose that (T, η, μ) is a monad on $\mathcal{C}$ such that $(\mathcal{C}_T, F_T(\mathcal{A}))$ is a structured display map category, and $T : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{C}, \mathcal{A})$ is a morphism of structured display map categories. Then*

$$F_T : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{C}_T, F_T(\mathcal{A}))$$

is also a morphism of structured display map categories. Moreover, there is an adjunction

$$(\mathcal{C}, \mathcal{A}) \begin{array}{c} \xrightarrow{F_T} \\ \perp \\ \xleftarrow{U_T} \end{array} (\mathcal{C}_T, F_T(\mathcal{A}))$$

*in the 2-category **sDMC**.*

Proof. The first condition for F_T to be a morphism of structured display map categories is trivially satisfied. A pullback square in $\mathcal{C}$ with the vertical arrows in $\mathcal{A}$ is sent by $U_T \circ F_T = T$ to a pullback square in $\mathcal{C}$ with the vertical arrows in $\mathcal{A}$ by hypothesis. But U_T reflects limits by Proposition 2.4; hence, the image of the pullback square under F_T must be a pullback square as well. Now, U_T is also a morphism of structured display map categories by Proposition 3.9, and since $F_T \dashv U_T$ in **Cat**, they must also be adjoint in **sDMC** because the 2-category contains all natural transformations – i.e., the unit and counit exist in **sDMC** and satisfy the triangle identities. ◀

► **Corollary 3.11.** *Suppose that $(\mathcal{C}, \mathcal{A})$ is a structured display map category, and (T, η, μ) is a cartesian monad on $\mathcal{C}$ such that $T(\mathcal{A}) \subseteq \mathcal{A}$. Then*

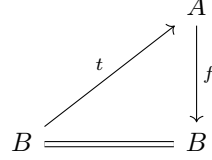
1. $T : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{C}, \mathcal{A})$ is a morphism of structured display map categories;
2. $(\mathcal{C}_T, F_T(\mathcal{A}))$ is a structured display map category;
3. there is an adjunction $F_T \dashv U_T$ in the 2-category **sDMC** between $(\mathcal{C}, \mathcal{A})$ and $(\mathcal{C}_T, F_T(\mathcal{A}))$.

Proof. $T : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{C}, \mathcal{A})$ is a morphism of structured display map categories by Example 3.8. We can then apply Proposition 3.7 to prove item 2, and we get item 3 by combining items 1 and 2 and applying Theorem 3.10 above. ◀

This last corollary is arguably one of the most useful results for “generating” new structured display map categories, as cartesian monads satisfying the hypotheses seem to be somewhat frequent in the wild. For example, it instantly applies to the partiality monad on **Set** with the class of monomorphisms, as seen in Example 2.11. We give a table with more examples in Section 4.1.

3.2 Contexts, Types, and Terms

► **Definition 3.12.** Consider a structured display map category $(\mathcal{C}, \mathcal{A})$. We call contexts the objects of $\mathcal{C}$ and types the arrows in $\mathcal{A}$. Given a type $f : A \longrightarrow B \in \mathcal{A}$, a term of f is a section $t : B \longrightarrow A$ of f in $\mathcal{C}$, as illustrated below



Continuing with Example 3.4, one can check that a substitution $t : \Gamma \longrightarrow \Gamma.A$ which is also a section of a display map $p_A : \Gamma.A \longrightarrow \Gamma$ indeed amounts to a judgment $\Gamma \vdash t : A$ of $\mathbb{T}$.

Let us fix a monad (T, η, μ) on a category $\mathcal{C}$ such that both $(\mathcal{C}, \mathcal{A})$ and $(\mathcal{C}_T, F_T(\mathcal{A}))$ are structured display map categories. When $F_T : \mathcal{C} \longrightarrow \mathcal{C}_T$ is a morphism of structured display map categories, then, since it is also *bijective-on-objects*, the functor establishes a bijective correspondence between the contexts of $(\mathcal{C}, \mathcal{A})$ and $(\mathcal{C}_T, F_T(\mathcal{A}))$. When η is monomorphic, the functor F_T is faithful. This entails, in particular, that both $\mathcal{A}$ and $F_T(\mathcal{A})$ are in bijective correspondence, witnessed by the functor F_T . Since types are modeled by display maps, F_T bijectively sends types of $(\mathcal{C}, \mathcal{A})$ to types of $(\mathcal{C}_T, F_T(\mathcal{A}))$. It thus only remains to compare the terms of $(\mathcal{C}, \mathcal{A})$ and $(\mathcal{C}_T, F_T(\mathcal{A}))$. The goal of the rest of this section is to see that under some conditions, the functor F_T also bijectively sends terms of types of $(\mathcal{C}, \mathcal{A})$ to terms of types of $(\mathcal{C}_T, F_T(\mathcal{A}))$.

► **Lemma 3.13.** Suppose we have a monad (T, η, μ) on a category $\mathcal{C}$ such that η is both monomorphic and cartesian. Then, for any pair σ, τ of composable arrows of $\mathcal{C}_T$, if $\tau \circ \sigma$ and τ are representable, then so is σ .

Proof. This is exactly one implication of [7, Proposition 3.6]. ◀

► **Theorem 3.14.** Suppose we have a monad (T, η, μ) on a category $\mathcal{C}$ such that η is both monomorphic and cartesian. Then, given an arrow $f : A \longrightarrow B$ of $\mathcal{C}$, the sections of f in $\mathcal{C}$ and the sections of $F_T(f) : A \longrightarrow B$ in $\mathcal{C}_T$ are in one-to-one correspondence.

Proof. F_T trivially sends sections of f to sections of $F_T(f)$ by functoriality. In the other direction, if $t : B \dashrightarrow A$ is a section of $F_T(f)$ in $\mathcal{C}_T$, then both $F_T(f) \circ t = 1_B$ and $F_T(f)$ are representable. By Lemma 3.13, we conclude that t must be representable as well, and since η is monomorphic, the representing arrow, which we write $t' : B \longrightarrow A$, is unique. Now we have that $F_T(f \circ t') = F_T(f) \circ F_T(t') = F_T(f) \circ t = 1_B$ in $\mathcal{C}_T$. By uniqueness of the representing arrow of 1_B in $\mathcal{C}_T$, we conclude that $f \circ t' = 1_B$ in $\mathcal{C}$, i.e., t' is a section of f . It is easy to see that these operations are inverses of each other. ◀

We immediately deduce the following “collapsing result”.

► **Corollary 3.15.** Suppose we have a monad (T, η, μ) on a category $\mathcal{C}$ such that η is both monomorphic and cartesian. Suppose moreover that we are given a class $\mathcal{A} \subseteq \text{Arr}(\mathcal{C})$ of arrows such that $(\mathcal{C}, \mathcal{A})$ and $(\mathcal{C}_T, F_T(\mathcal{A}))$ are structured display map categories. Then the contexts, types, and terms of both $(\mathcal{C}, \mathcal{A})$ and $(\mathcal{C}_T, F_T(\mathcal{A}))$ are in bijective correspondence.

This bijective correspondence is witnessed by the functor $F_T : \mathcal{C} \longrightarrow \mathcal{C}_T$ when it is a morphism of structured display map categories (cf. Theorem 3.10).

Proof. This is trivially a consequence of Theorem 3.14 above and the various remarks we made about F_T . $\blacktriangleleft$

► **Example 3.16.** Consider the partiality monad $P : \mathbf{Set} \longrightarrow \mathbf{Set}$ again. This monad is cartesian, and therefore $(\mathbf{Set}_P, \text{Repr}(P))$ is a structured display map category. An arrow $\sigma : \Delta \dashrightarrow \Gamma$ of $\mathbf{Set}_P$, *i.e.*, an arrow $\Delta \longrightarrow P(\Gamma)$ in $\mathbf{Set}$, may be thought of as a partially defined substitution from a context Δ to a context Γ . The unit of the monad is both monomorphic and cartesian, and by Corollary 3.15 we conclude that the contexts, types, and terms of both $(\mathbf{Set}, \text{Arr}(\mathbf{Set}))$ and $(\mathbf{Set}_P, \text{Repr}(P))$ are in bijection. But the structured display map category $(\mathbf{Set}_P, \text{Repr}(P))$ has, by contrast, strictly more *context morphisms*, *i.e.*, arrows between contexts.

In general, a context $F_T(\Gamma) \in \mathcal{C}_T$ may be semantically very different from $\Gamma \in \mathcal{C}$.

► **Example 3.17.** Consider again the partiality monad $P : \mathbf{Set} \longrightarrow \mathbf{Set}$ from Example 3.16. The terminal object $* \in \mathbf{Set}$ interprets the *empty context*, and the inclusion $\emptyset \longrightarrow *$ of the empty set into the singleton interprets the empty type over the empty context, *i.e.*, the object $\emptyset \in \mathbf{Set}$ interprets the *empty-type-in-empty-context*.

But $*$ is *not* the terminal object in $\mathbf{Set}_P$. The terminal object, which also happens to be initial, is in that case $\emptyset \in \mathbf{Set}_P$. The morphism F_T actually sends the empty-type-in-empty-context $\emptyset \in \mathbf{Set}$ to the terminal object $\emptyset \in \mathbf{Set}_P$, and the empty context $* \in \mathbf{Set}$ is sent to something else.

To summarize, under the hypotheses of both Theorem 3.10 and Corollary 3.15, the functor $F_T : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{C}_T, F_T(\mathcal{A}))$ bijectively transports contexts, types, and terms of $(\mathcal{C}, \mathcal{A})$ to $(\mathcal{C}_T, F_T(\mathcal{A}))$, although the images under F_T may be very different from the point of view of type theory (see Example 3.17). This can be read both positively and negatively. On the one hand, since $\mathcal{C}$ is generally conceptually simpler to understand than the Kleisli category $\mathcal{C}_T$, the fact that $(\mathcal{C}_T, F_T(\mathcal{A}))$ is “essentially the same as” $(\mathcal{C}, \mathcal{A})$ is useful in itself. On the other hand, this suggests that a structured display map category $(\mathcal{C}_T, F_T(\mathcal{A}))$ is really interesting only when η is not monomorphic. We would like to stress again that even if $F_T : (\mathcal{C}, \mathcal{A}) \longrightarrow (\mathcal{C}_T, F_T(\mathcal{A}))$ “collapses” the contexts, types, and terms, the operation of taking the structured display map category $(\mathcal{C}_T, F_T(\mathcal{A}))$ almost always strictly adds new context morphisms to $(\mathcal{C}, \mathcal{A})$.

Finally, a bijection between terms in type theory implies a bijection between substitutions, as substitutions are n -tuples of terms. Since substitutions are interpreted as context morphisms in structured display map categories, if two structured display map categories have the same terms, then surely they should have the same context morphisms. The discussion and examples above show that this is not the case. This illustrates the “frugal framework” of structured display map categories: to have a fine-grained categorical model, matching the type-theoretic intuitions, one would need additional structure and properties on the base category.

4 Examples

We will now see several other examples and nonexamples.

► **Example 4.1.** The identity monad on a category $\mathcal{C}$ trivially satisfies the hypotheses of both Theorem 2.7 and Theorem 3.10.

► **Example 4.2.** When $\mathcal{C}$ has all pullbacks and $\mathcal{A} = \text{Arr}(\mathcal{C})$, then cartesian monads over $\mathcal{C}$ satisfy the hypotheses of both theorems as well. This includes, in particular, *parametric right adjoint* monads and therefore *polynomial* monads. Indeed, we have the following inclusions

$$\text{Polynomial monads} \subseteq \text{Parametric right adjoint monads} \subseteq \text{Cartesian monads}$$

► **Example 4.3.** Suppose that $\mathcal{E}$ is an elementary topos. Then $(\text{Epi}, \text{Mono})$ is an *orthogonal factorization system* on $\mathcal{E}$, where the left class is the class of all epimorphisms and the right class is the class of all monomorphisms. By definition, $\mathcal{E}$ has finite limits and so, in particular, all pullbacks, and it is well known that the right class of an orthogonal factorization system is stable under pullbacks. We conclude that $(\mathcal{E}, \text{Mono})$ is a structured display map category – in this case, the display maps are the monomorphisms,⁵ and so, in essence, the dependent types are merely the propositions, *i.e.*, they express the *subobjects*.

Now, let $T : \mathcal{E} \rightarrow \mathcal{E}$ be a cartesian monad on $\mathcal{E}$. It turns out that in many cases – most notably in the examples below – we have that $T(\text{Mono}) \subseteq \text{Mono}$. As a result, we are in the special case of Corollary 3.11 and so we conclude that $(\mathcal{E}_T, F_T(\text{Mono}))$ is a structured display map category, and moreover we have an adjunction

$$(\mathcal{E}, \text{Mono}) \begin{array}{c} \xrightarrow{F_T} \\ \perp \\ \xleftarrow{U_T} \end{array} (\mathcal{E}_T, F_T(\text{Mono}))$$

in the 2-category **sDMC**. Furthermore, if the unit of the monad is monomorphic, then by Corollary 3.15 the contexts, types, and terms of the structured display map categories $(\mathcal{E}, \text{Mono})$ and $(\mathcal{E}_T, F_T(\text{Mono}))$ are in bijective correspondence.

► **Example 4.4.** Every locally cartesian closed category $\mathcal{C}$ has all pullbacks, and therefore $(\mathcal{C}, \text{Arr}(\mathcal{C}))$ is always a structured display map category. Following Example 4.3, all elementary topoi are locally cartesian closed categories. In this case, we know precisely that the syntax of $(\mathcal{C}, \text{Arr}(\mathcal{C}))$ is *extensional Martin-Löf type theory* [10]. This translates concretely into the fact that the internal logic of $\mathcal{C}$ is dependent type theory with Σ -types, Π -types, and extensional identity types.

Now, suppose we have a monad (T, η, μ) on $\mathcal{C}$ such that $(\mathcal{C}_T, \text{Repr}(T))$ is a structured display map category. We know that a simple way to achieve this is for the monad T to be cartesian or, even better, polynomial⁶.

We pointed out earlier that not all arrows of $\mathcal{C}_T$ are representable in general. As a consequence, $(\mathcal{C}_T, \text{Repr}(T))$ *cannot* be of the form $(\mathcal{D}, \text{Arr}(\mathcal{D}))$ since $\text{Repr}(T) \neq \text{Arr}(\mathcal{C}_T)$, and therefore $(\mathcal{C}_T, \text{Repr}(T))$ is not the structured display map category coming from a locally cartesian closed category. Because of the biequivalence in [10], we conclude that the syntax of $(\mathcal{C}_T, \text{Repr}(T))$ cannot be extensional Martin-Löf type theory, even though the contexts, types, and terms of both $(\mathcal{C}, \text{Arr}(\mathcal{C}))$ and $(\mathcal{C}_T, \text{Repr}(T))$ may be in bijection by Corollary 3.15.

4.1 Notions of Computation

Eugenio Moggi had the spectacular insight to provide categorical semantics for programs with computational effects using monads [22]. Given a monad T that represents a *notion of computation*, a program “with computational effects T ” may be viewed as an arrow $A \rightarrow TB$,

⁵ We have in this case that the display maps contain the isomorphisms and are stable under composition. This brings us one step closer to the notion of *clan* introduced by André Joyal [20].

⁶ See, *e.g.*, [17] for an illustration “in the wild” of polynomial monads over locally cartesian closed categories.

i.e., an arrow $A \multimap B$ in the associated Kleisli category. Composition of programs works as expected, *i.e.*, by composing the arrows in the Kleisli category. We provide a table with some of the notions of computation originally found in Eugenio Moggi’s paper, together with an indication of whether the monad is cartesian, whether its unit is monomorphic, and whether the monad sends monomorphisms to monomorphisms. The en dashes (–) indicate that the property is irrelevant in that case.

$T : \mathbf{Set} \longrightarrow \mathbf{Set}$			cartesian?	η monomorphic?	$T(\mathbf{Mono}) \subseteq \mathbf{Mono}$?
X	$\longmapsto$	$X + *$	yes	yes	yes
X	$\longmapsto$	$\mathcal{P}_{\mathbf{Fin}}(X)$	no	–	–
X	$\longmapsto$	$(S \times X)^S$	no	–	–
X	$\longmapsto$	$X + E$	yes	yes	yes
X	$\longmapsto$	R^{R^X}	no	–	–
X	$\longmapsto$	$\mu A.X + A^U$	yes	yes	yes
X	$\longmapsto$	$A^* \times X$	yes	yes	yes
X	$\longmapsto$	X^A	no	–	–

Consider, for example, the *exception monad* $T_E : X \longmapsto X + E$, for which Example 3.16 is a special case. We see in the table above that the monad is cartesian, its unit is monomorphic, and it is such that $T_E(\mathbf{Mono}) \subseteq \mathbf{Mono}$. Moreover, **Set** is the primary example of an elementary topos, so we conclude from Example 4.3 that there is an adjunction

$$(\mathbf{Set}, \mathbf{Mono}) \begin{array}{c} \xrightarrow{F_{T_E}} \\ \perp \\ \xleftarrow{U_{T_E}} \end{array} (\mathbf{Set}_{T_E}, F_{T_E}(\mathbf{Mono}))$$

in the 2-category **sDMC**, and F_{T_E} bijectively sends contexts, types, and terms of $(\mathbf{Set}, \mathbf{Mono})$ to $(\mathbf{Set}_{T_E}, F_{T_E}(\mathbf{Mono}))$.

► **Remark 4.5.** Pierre-Marie Pédro and Nicolas Tabareau developed their own monadic approach to implementing effects in dependent type theory called the *weaning translation* [25]. They consequently defined an *exceptional translation* of the Calculus of Inductive Constructions into itself [26]. The “collapsing” result of Corollary 3.15 suggests that our approach is *not* the correct one for implementing effects in dependent type theory. Nonetheless, we would like to know how our work relates to theirs and, more specifically, how their exceptional translation compares with our framework when applied to the exception monad.

4.2 Perfect Operator Categories

Barwick introduced *operator categories* to capture the properties of various higher operads and their algebras [4]. In this setting, for example, the categories Γ^{op} and Δ^{op} , opposites of Segal’s category and the simplex category, respectively, are Kleisli categories⁷ for monads over so-called *perfect operator categories*. We refer to Barwick for the exact definitions of operator categories and their “perfect” version.

Every perfect operator category Φ comes with a canonical parametric right adjoint monad $T_\Phi : \Phi \longrightarrow \Phi$ [4, §5.12]. While a perfect operator category is some kind of category of finite sets with structure, its canonical monad, to quote Barwick, “adds as few points as possible in as many directions as possible”. When Φ has all pullbacks, we get, for free, that $(\Lambda(\Phi), \text{Repr}(T_\Phi))$ is a structured display map category (where $\Lambda(\Phi) := \Phi_{T_\Phi}$ is the Kleisli category for the monad T_Φ and is called the *Leinster category* by Barwick).

⁷ This is reportedly an old observation of André Joyal and Ross Street.

As a result of T_Φ being a parametric right adjoint, the category $\Lambda(\Phi)$ also has a strict factorization system (see [4, Lemma 7.3], or [29, Proposition 2.6]) where the right class is exactly the class of all representable arrows (they are called *active* in this context). Note that a strict factorization system is a slightly weaker notion than an orthogonal factorization system, in that the classes are required to contain all isomorphisms in the latter case.

► **Example 4.6.** The most trivial example of a perfect operator category is the one-object category $*$. Its canonical monad $T_* : * \longrightarrow *$ must be the only endofunctor on $*$, *i.e.*, the identity functor. The category $*$ obviously has all pullbacks, and therefore we have that $(\Lambda(*), \text{Repr}(T_*)) = (*, \text{Arr}(*))$ is the “trivial” structured display map category.

► **Example 4.7.** Let us denote by $\mathbf{F}$ the category of finite sets. This is a perfect operator category, and the canonical monad $T_{\mathbf{F}} : \mathbf{F} \longrightarrow \mathbf{F}$ merely adds a point, meaning that $T_{\mathbf{F}}$ is nothing but the partiality monad on finite sets.

Its associated Kleisli category is Γ^{op} , the category opposite to Segal’s category, and since $\mathbf{F}$ obviously has all pullbacks, we conclude that $(\Gamma^{\text{op}}, \text{Repr}(T_{\mathbf{F}}))$ is a structured display map category.

► **Example 4.8.** We will now see our most important example. The category $\mathbf{O}$ of totally ordered finite sets and order-preserving maps is also a perfect operator category, and the canonical monad $T_{\mathbf{O}} : \mathbf{O} \longrightarrow \mathbf{O}$ adds both a maximal and a minimal element to a totally ordered finite set. Its associated Kleisli category turns out to be Δ^{op} , and so we could secretly hope that Δ^{op} may be equipped with display maps in some way. At first glance, $\mathbf{O}$ does not have all pullbacks, so the pair $(\Delta^{\text{op}}, \text{Repr}(T_{\mathbf{O}}))$ fails to be a structured display map category right away. One could think that this could be rectified by noticing that Δ^{op} has no nontrivial isomorphisms, and therefore the factorization system induced by $T_{\mathbf{O}}$ on Δ^{op} is orthogonal on the nose. As such, we would have that the right class – namely, the representable arrows – is stable under pullbacks as is the case for every orthogonal factorization system, and therefore the pair $(\Delta^{\text{op}}, \text{Repr}(T_{\mathbf{O}}))$ would be a structured display map category. This way, we would completely bypass the need for $\mathbf{O}$ to have all pullbacks. Again, we are defeated by the basic fact that Δ^{op} does not have all pullbacks either – Δ does not have all *pushouts* – and so the representable arrows could not possibly be stable under them. The pullbacks that do exist in Δ^{op} are entirely characterized in [13, §3], and the representable arrows are notoriously stable under pullback along arrows from the left class of the factorization system [16, Lemma 2.7].

There is a solution if we consider a restricted class of arrows of $\text{Repr}(T_{\mathbf{O}})$. The class Mono of monomorphisms of $\mathbf{O}$, which consists of all the injective maps between totally ordered finite sets, is actually stable under pullbacks. As a consequence, we have that $(\mathbf{O}, \text{Mono})$ is a structured display map category. We also observe that $T_{\mathbf{O}}(\text{Mono}) \subseteq \text{Mono}$.

We can explicitly spell out the arrows in $F_{T_{\mathbf{O}}}(\text{Mono})$. Let us denote by $\mathbf{n}$ the totally ordered finite set with n elements in $\mathbf{O}$. We have that $F_{T_{\mathbf{O}}}$ sends every $\mathbf{n}$ to the object $[n]$ of Δ^{op} , and an arrow $f : \mathbf{m} \longrightarrow T_{\mathbf{O}}(\mathbf{n})$ in $\mathbf{O}$ corresponds to an arrow $f : [m] \longleftarrow [n]$ in Δ^{op} (that is, an arrow $f : [n] \longrightarrow [m]$ in Δ). The exact translation is outside the scope of this paper. One can check, through this translation, that the class $F_{T_{\mathbf{O}}}(\text{Mono})$ consists exactly of all the arrows $f : [m] \longleftarrow [n]$ of Δ^{op} that are *surjective maps* in Δ .

Let us denote this class of surjective maps by Surj_Δ . Since $T_{\mathbf{O}}(\text{Mono}) \subseteq \text{Mono}$, we are in the situation of Corollary 3.11. We conclude that $(\Delta^{\text{op}}, \text{Surj}_\Delta)$ is a structured display map category,⁸ and we have an adjunction

$$(\mathbf{O}, \text{Mono}) \begin{array}{c} \xrightarrow{F_{T_{\mathbf{O}}}} \\ \perp \\ \xleftarrow{U_{T_{\mathbf{O}}}} \end{array} (\Delta^{\text{op}}, \text{Surj}_\Delta)$$

in the 2-category **sDMC**. Finally, the unit of the monad $T_{\mathbf{O}}$ is monomorphic; therefore, $F_{T_{\mathbf{O}}}$ bijectively sends contexts, types, and terms of $(\mathbf{O}, \text{Mono})$ to $(\Delta^{\text{op}}, \text{Surj}_\Delta)$.

There are two 2-categories of perfect operator categories $\mathbf{Op}^{\text{perf}} \subseteq \mathbf{Adm}^{\text{perf}}$ defined by Barwick. The category $*$ is initial in both 2-categories and terminal in $\mathbf{Adm}^{\text{perf}}$, while $\mathbf{F}$ is terminal in $\mathbf{Op}^{\text{perf}}$. The category $\mathbf{O}$ sits somewhere in between. It is an open problem to characterize all perfect operator categories that have all pullbacks, or whose class of monomorphisms is stable under pullbacks. Additionally, it is not known to us whether the opposite $\square^{\text{op}}$ of any of the various flavors of *categories of cubes*, which are of great interest to both homotopy theorists and homotopy type theorists, is the Kleisli category associated to some perfect operator category.

4.3 Other Examples

► **Example 4.9.** Given an *alphabet* A , we can define the set of *words* A^* over this alphabet, i.e., A^* is the *free monoid* generated by A . The assignment

$$\begin{aligned} L : \mathbf{Set} &\longrightarrow \mathbf{Set} \\ A &\longmapsto A^* \end{aligned}$$

induces a polynomial monad on **Set**, and therefore $(\mathbf{Set}_L, \text{Repr}(L))$ is a structured display map category. Observe that the unit is also monomorphic. L is called the *list monad* by computer scientists.

► **Example 4.10.** Thomas Colcombet and Daniela Petrişan proved that *subsequential transducers* from an input alphabet A to an output alphabet B are in one-to-one correspondence with functors $\mathcal{I}_A \longrightarrow \mathbf{Set}_{T_B}$ satisfying some conditions [12, Lemma 11]. T_B is defined as the following monad

$$\begin{aligned} T_B : \mathbf{Set} &\longrightarrow \mathbf{Set} \\ X &\longmapsto B^* \times X + * \end{aligned}$$

This is the writer monad composed with the partiality monad. This monad is cartesian, and its unit is monomorphic; therefore, we can directly apply Example 4.3 and get that $(\mathbf{Set}_{T_B}, \text{Repr}(T_B))$ is a structured display map category with the same contexts, types, and terms as $(\mathbf{Set}, \text{Arr}(\mathbf{Set}))$.

In their article, Colcombet and Petrişan also showed that $\mathbf{Set}_{T_B}$ has an (orthogonal) factorization system (E, M) [12, Lemma 15]. One can check that the right class M is in fact exactly $F_{T_B}(\text{Mono})$, and since $T_B(\text{Mono}) \subseteq \text{Mono}$, we have an adjunction

$$(\mathbf{Set}, \text{Mono}) \begin{array}{c} \xrightarrow{F_{T_B}} \\ \perp \\ \xleftarrow{U_{T_B}} \end{array} (\mathbf{Set}_{T_B}, M)$$

in the 2-category **sDMC**. Moreover, the structured display map categories $(\mathbf{Set}, \text{Mono})$ and $(\mathbf{Set}_{T_B}, M)$ have the same contexts, types, and terms.

⁸ It is even a display map category because Δ^{op} has no nontrivial isomorphisms.

4.4 Nonexamples

► **Example 4.11.** The free *commutative* monoid monad, in contrast with Example 4.9, is *not* cartesian for deep reasons that are detailed in [11, Proposition 5.1].

► **Example 4.12.** There is no hope for the powerset monad $\mathcal{P} : X \mapsto \mathcal{P}(X)$ and its variants to be such that $(\mathbf{Set}_{\mathcal{P}}, \mathbf{Repr}(\mathcal{P}))$ is a structured display map category. One can check, using the pullback of $\mathcal{P}(\{0, 1\}) \longrightarrow \mathcal{P}(\{0\})$ along itself, that this is indeed impossible.

5 Conclusion and Future Directions

The final example, Example 4.12, seems to indicate that the approach presented in this paper is *not* the correct one for **Prof**, as the relative pseudomonad $\mathcal{C} \longmapsto \widehat{\mathcal{C}}$ may be seen as a “categorification” of the powerset monad. Most critically, an obstruction at the level of sets surely carries over to the categorification through the canonical embedding $\mathbf{Set} \hookrightarrow \mathbf{Cat}$.

There is still more work to be done; for example, we could focus on a simple monad like the exception monad $T_E : X \mapsto X + E$ and see whether we can extract a syntax from $(\mathbf{Set}_{T_E}, F_{T_E}(\mathcal{A}))$ for different choices of $\mathcal{A} \subseteq \mathbf{Arr}(\mathbf{Set})$. Then we can compare the result with the exceptional translation of Pierre-Marie Pédrot and Nicolas Tabareau (*cf.* Remark 4.5). We also mentioned in Example 4.4 that $(\mathcal{C}_T, \mathbf{Repr}(T))$ cannot be a model of extensional Martin-Löf type theory in general. This entails that the basic type constructors in $\mathcal{C}_T$, whatever they may be, must be noticeably different from those in extensional theory in a way that may be worth investigating.

References

- 1 Benedikt Ahrens, Peter LeFanu Lumsdaine, and Paige Randall North. Comparing Semantic Frameworks for Dependently-Sorted Algebraic Theories. In Oleg Kiselyov, editor, *Programming Languages and Systems*, pages 3–22. Springer Nature, 2025. doi:10.1007/978-981-97-8943-6_1.
- 2 Benedikt Ahrens, Paige Randall North, and Niels van der Weide. Bicategorical type theory: Semantics and syntax. *Mathematical Structures in Computer Science*, 33(10):868–912, 2023. doi:10.1017/S0960129523000312.
- 3 Thosten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, Volume 11, Issue 1, 2015. doi:10.2168/LMCS-11(1:3)2015.
- 4 Clark Barwick. From operator categories to higher operads. *Geometry & Topology*, 22(4):1893–1959, April 2018. doi:10.2140/gt.2018.22.1893.
- 5 Jean Bénabou. Distributors at Work, 2000.
- 6 Francis Borceux. *Handbook of Categorical Algebra: Volume 2: Categories and Structures*, volume 2 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, 1994. doi:10.1017/CB09780511525865.
- 7 Dominique Bourn. Kleisli categories, T-categories and internal categories. *Theory and Applications of Categories*, 41(34):1108–1159, 2024.
- 8 Albert Burroni. T-catégories (catégories dans un triple). *Cahiers de topologie et géométrie différentielle*, 12(3):215–321, 1971.
- 9 John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986. doi:10.1016/0168-0072(86)90053-9.
- 10 Pierre Clairambault and Peter Dybjer. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. *Mathematical Structures in Computer Science*, 24(6):e240606, 2014. doi:10.1017/S0960129513000881.

- 11 Maria Manuel Clementino, Dirk Hofmann, and George Janelidze. The monads of classical algebra are seldom weakly cartesian. *Journal of Homotopy and Related Structures*, 9(1):175–197, 2014. doi:10.1007/s40062-013-0063-2.
- 12 Thomas Colcombet and Daniela Petrişan. Automata Minimization: A Functorial Approach. *Logical Methods in Computer Science*, Volume 16, Issue 1, 2020. doi:10.23638/LMCS-16(1:32)2020.
- 13 Carmen Constantin, Tobias Fritz, Paolo Perrone, and Brandon T. Shapiro. Weak cartesian properties of simplicial sets. *Journal of Homotopy and Related Structures*, 18(4):477–520, 2023. doi:10.1007/s40062-023-00334-1.
- 14 Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 120–134. Springer, 1996. doi:10.1007/3-540-61780-9_66.
- 15 M. Fiore, N. Gambino, M. Hyland, and G. Winskel. Relative pseudomonads, Kleisli bicategories, and substitution monoidal structures. *Selecta Mathematica*, 24(3):2791–2830, 2018. doi:10.1007/s00029-017-0361-3.
- 16 Imma Gálvez-Carrillo, Joachim Kock, and Andrew Tonks. Decomposition spaces, incidence algebras and Möbius inversion I: Basic theory. *Advances in Mathematics*, 331:952–1015, 2018. doi:10.1016/j.aim.2018.03.016.
- 17 Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154(1):153–192, 2013. doi:10.1017/S0305004112000394.
- 18 Martin Hofmann. Syntax and semantics of dependent types. In Martin Hofmann, editor, *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, 1997. doi:10.1007/978-1-4471-0963-1_2.
- 19 Bart Jacobs. Comprehension categories and the semantics of type dependency. *Theoretical Computer Science*, 107(2):169–207, 1993. doi:10.1016/0304-3975(93)90169-T.
- 20 Andre Joyal. Notes on Clans and Tribes, 2017. doi:10.48550/arXiv.1710.10238.
- 21 Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 2nd. ed., softcover version of original hardcover edition 1998 edition, 1978. doi:10.1007/978-1-4757-4721-8.
- 22 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.
- 23 Hayato Nasu. An Internal Logic of Virtual Double Categories, 2025. doi:10.48550/arXiv.2410.06792.
- 24 Max S. New and Daniel R. Licata. A Formal Logic for Formal Category Theory. In Orna Kupferman and Pawel Sobocinski, editors, *Foundations of Software Science and Computation Structures*, pages 113–134, 2023. doi:10.1007/978-3-031-30829-1_6.
- 25 Pierre-Marie Pédro and Nicolas Tabareau. An effectful way to eliminate addiction to dependence. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005113.
- 26 Pierre-Marie Pédro and Nicolas Tabareau. Failure is Not an Option. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 245–271. Springer International Publishing, 2018. doi:10.1007/978-3-319-89884-1_9.
- 27 Emily Riehl. *Category Theory in Context*. Aurora Dover Modern Math Originals. Dover Publications, Inc, 2016.
- 28 Jenő Szegedy. On limits and colimits in the Kleisli category. *Cahiers de topologie et géométrie différentielle*, 24(4):381–391, 1983.
- 29 Mark Weber. Familial 2-functors and parametric right adjoints. *Theory and Applications of Categories*, 18(22):665–732, 2007.
- 30 R. J. Wood. Abstract pro arrows I. *Cahiers de topologie et géométrie différentielle*, 23(3):279–290, 1982.

Type-Theoretic Replacement and Univalent Completion: Applications and Interpretations

Evan Cavallo   

Department of Computer Science and Engineering, University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden

Thierry Coquand   

Department of Computer Science and Engineering, University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden

Abstract

We study Rijke’s type theoretic axiom of replacement, which closes a type-theoretic universe under certain images, and its cousin the axiom of univalent completions, which postulates an extension of each type family to a univalent family. In a long expository section, we survey applications of these two axioms in the literature, from the construction of truncations to the construction of Eilenberg–MacLane spaces to the interpretation of material set theory, making a case for their value as foundational principles. We then give direct, constructive interpretations of the axioms in cubical sets models of type theory and suggest how they can be understood as higher inductive constructions.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Homotopy type theory, axiom of replacement, univalent completion, higher inductive type, cubical sets

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.10

Funding *Evan Cavallo*: Knut and Alice Wallenberg Foundation (KAW), Grant No. 2019.0116. *Thierry Coquand*: ForCUTT project, ERC advanced grant 101053291.

Acknowledgements We thank David Wärn for introducing us to univalent completion and for supplying the proof of Proposition 17, which relates completion to replacement. We thank Amélia Liao and David Wärn for independently pointing out a simplification of our original inductive recursive formulation of replacement. We thank our reviewers for thorough comments.

1 Introduction

Rijke’s *type theoretic axiom of replacement* for a universe $\mathcal{U}$ states that every map $f : A \rightarrow B$ from a $\mathcal{U}$ -small type A (i.e., one equivalent to an element of $\mathcal{U}$) to a locally $\mathcal{U}$ -small type B (i.e., one whose identity types are $\mathcal{U}$ -small) admits a $\mathcal{U}$ -small image.

Rijke first states replacement as a *theorem* of intensional type theory with function extensionality and a univalent universe $\mathcal{U}$ closed under pushouts and a natural numbers type [51, Theorem 4.6] and explicitly connects it with set theory’s Replacement axiom schema in his dissertation [52, §4.2.3]. His construction is non-trivial, building the image as a sequential colimit of fiberwise joins of f with itself. The special case of replacement where the codomain is an h -set appears earlier, explicitly connected with set theoretic replacement, in Rijke and Spitters’s description of the IHW-pretopos of sets [54]; De Jong and Escardó also discuss this case [27, §3.4] [26, §2.11.4]. The general principle is first taken as an axiom in Rijke’s *Introduction to Homotopy Type Theory* [53, Axiom 18.1.8]. Bezem, Buchholtz, Cagne, Dundas, and Grayson’s book-in-progress *Symmetry* [10] also isolates the axiom and points out several applications that go beyond the world of h -sets. Although replacement is a relatively weak principle, being derivable from basic higher inductive types, we shall see that it is also surprisingly flexible.



© Evan Cavallo and Thierry Coquand;
licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 10; pp. 10:1–10:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 Type-Theoretic Replacement and Univalent Completion

We work in type theory with intensional Id -types, Σ -types, Π -types with extensionality, and empty and unit types. By a *universe*, we mean one with these type formers. We are mainly interested in *univalent* universes but assume univalence explicitly when we need it.

► **Definition 1.** A map $f : A \rightarrow B$ is an embedding if for all $a, a' : A$, the action on identities $\text{ap}_f^{a,a'} : (a = a') \rightarrow (f\ a = f\ a')$ is an equivalence. Set $\text{isEmbedding } f \equiv \prod_{a,a'} \text{isEquiv } \text{ap}_f^{a,a'}$.

► **Definition 2.** A map $f : A \rightarrow B$ is a surjection when for every family of propositions P over B , the precomposition map $(- \circ f) : \prod_{b:B} P\ b \rightarrow \prod_{a:A} P\ (f\ a)$ is an equivalence.

► **Remark 3.** Definition 2 quantifies over *all* propositions over B , so it cannot a priori be captured by a type. When we quantify over types, we implicitly quantify over substitutions: $\Gamma \vdash f : A \rightarrow B$ is surjective if for each $\sigma : \Gamma' \rightarrow \Gamma$ and family of propositions P over $B\sigma$, $(- \circ f\sigma)$ is an equivalence, and our choice of witnesses commutes with substitution.

Given a (-1) -truncation $\|- \|$, surjectivity of $f : A \rightarrow B$ is equivalent to $\prod_{b:B} \|\text{Fiber}_f\ b\|$ where $\text{Fiber}_f\ b \equiv \sum_{a:A} f\ a =_B b$ [53, Proposition 15.2.3]. This reduction requires that the truncation's eliminator can target type families of arbitrary size, i.e., that the maps $T \rightarrow \|T\|$ are surjections in the sense of Definition 2.

► **Definition 4.** An image of $f : A \rightarrow B$ is a factorization $A \xrightarrow{p} X \xrightarrow{i} B$ of f as a surjection followed by an embedding.

► **Definition 5.** A type is $\mathcal{U}$ -small when it is equivalent to a type in $\mathcal{U}$ and locally $\mathcal{U}$ -small when its identity types are $\mathcal{U}$ -small. Thus we set $\text{Small}_{\mathcal{U}}\ A \equiv \sum_{X:\mathcal{U}} (X \simeq A)$ and $\text{LocSmall}_{\mathcal{U}}\ A \equiv \prod_{a,a':A} \text{Small}_{\mathcal{U}}\ (a =_A a')$.

► **Definition 6 (Replacement).** A universe $\mathcal{U}$ has replacement when for every $\mathcal{U}$ -small type A , locally $\mathcal{U}$ -small type B , and $f : A \rightarrow B$, we have a $\mathcal{U}$ -small image of f .

► **Remark 7.** Rijke [53, Axiom 18.1.8] states replacement assuming (-1) -truncation. In this case, we have the image of $f : A \rightarrow B$ as $\sum_{b:B} \|\text{Fiber}_f\ b\|$. Thus, Rijke can formulate replacement as *the image of a map from a $\mathcal{U}$ -small type into a locally $\mathcal{U}$ -small type is $\mathcal{U}$ -small*. We instead bundle the existence and $\mathcal{U}$ -smallness of the relevant images in one axiom schema.

Replacement immediately implies the existence of images of maps in $\mathcal{U}$. In fact, assuming $\mathcal{U}$ is univalent, it can be used to define n -images (§ 2.1) and quotients by equivalence relations (§ 2.2) and even implies that $\mathcal{U}$ contains non-sets (§ 2.3).

Replacement is closely related to another principle, the axiom schema of *univalent completions*, which intuitively asks that the “universe of *all* types” behaves like a universe with replacement. Recall that we can formulate *univalence* as a condition on any type family, in such a way that universe $\mathcal{U}$ is univalent precisely when $A : \mathcal{U} \vdash A$ type is univalent.

► **Definition 8 (Voevodsky [69, Definition 3.4]).** A family $a : A \vdash P(a)$ type is univalent if for all $a_0, a_1 : A$ the induced map $a_0 =_A a_1 \rightarrow P(a_0) \simeq P(a_1)$ is an equivalence.

► **Definition 9.** A cartesian morphism $(f, e) : P \rightarrow Q$ between families $a : A \vdash P(a)$ type and $b : B \vdash Q(b)$ type consists of a map $f : A \rightarrow B$ and equivalences $e : \prod_{a:A} P(a) \simeq Q(f\ a)$.

► **Definition 10 (Univalent extensions and completions).** A univalent extension of a family $a : A \vdash P(a)$ type is a univalent family $\bar{a} : \bar{A} \vdash \bar{P}(\bar{a})$ type together with a cartesian morphism $(q, e) : P \rightarrow \bar{P}$. It is a univalent completion if $q : A \rightarrow \bar{A}$ is a surjection. A type theory has enough univalent families if every family admits a univalent extension, and has univalent completions if every family admits a univalent completion.

Univalent completion has been more thoroughly studied from semantic and categorical perspectives, where universes are less casually assumed than in type theory. Van den Berg and Moerdijk [65] show that the simplicial model of homotopy type theory [40] has enough univalent families. They are the first to use the term “univalent completion”, to our knowledge; they do not define it, but their theorem takes a fibration $a : A \vdash P(a)$ type to a univalent extension $(f, e) : P \rightarrow \overline{P}$ with f a cofibration. Stenzel [61, §6] develops the theory of univalent completions in model categories. Gepner and Kock [33] and Rasekh [48] discuss univalent families and completion respectively in $(\infty, 1)$ -categories; Anel [5, §A.2–3] continues this study. Uemura [63, Proposition 4.7] shows that ∞ -logoi have univalent completions following Rijke’s join construction. On the syntactic side, Christensen [21] discusses univalent families in type theory, while Awodey has proposed taking existence of univalent completions as a type theoretic principle at a MURI grant meeting in Pittsburgh in August 2025.

Working in homotopy type theory, we often restrict attention to types in some univalent universe. Such a universe is required for calculations with identity types such as the characterization $\|a =_A a'\|_n \simeq (\|a\|_{n+1} =_{\|A\|_{n+1}} \|a'\|_{n+1})$ of truncated identity types. One may assume that every type (or finite collection of types) belongs to some univalent universe (as, e.g., in Rijke [53, Postulate 6.2.1]), mimicking Grothendieck’s set theoretic axiom of universes [8, Exposé I, Appendice, §4]. For many applications, however, we do not need these universes to be closed under any type formers. The assumption of enough univalent families can thus decouple uses of univalence from size issues.

► **Example 11** (Identities in truncations). Given n - and $(n + 1)$ -truncations and enough univalent families, we have $\|a =_A a'\|_n \simeq (\|a\|_{n+1} =_{\|A\|_{n+1}} \|a'\|_{n+1})$. By assumption, the family $(a, a') : A \times A \vdash \|a =_A a'\|_n$ type admits a cartesian morphism to a univalent family $u : U \vdash E(u)$ type, so $q : A \times A \rightarrow U$ with $\|a =_A a'\|_n \simeq E(q(a, a'))$. We then use U as a universe to follow the standard proof [64, Theorem 7.3.12]: we only need that U is univalent and has codes $q(a, a')$ for the types $\|a =_A a'\|_n$, not that it is closed under any type formers.

With univalent completions, we can derive many of the same operations as we can get from replacement. To begin with, we have images:

► **Example 12.** A type theory with univalent completions has (-1) -truncations.

Proof. Let A type and consider the constant family $a : A \vdash 1$ type. By assumption, we have a univalent $\overline{a} : \overline{A} \vdash E(\overline{a})$ type with $q : A \rightarrow \overline{A}$ and $1 \simeq E(q a)$ for all $a : A$. To show $q : A \rightarrow \overline{A}$ is the propositional truncation of A , it suffices to check that $\overline{A}$ is a proposition, i.e., $\overline{a} =_{\overline{A}} \overline{a}'$ is contractible for all $\overline{a}, \overline{a}' : \overline{A}$. Since being contractible is a proposition and q is surjective, we can assume $\overline{a} \equiv q a$ and $\overline{a}' \equiv q a'$ for some $a, a' : A$. In this case $(q a =_{\overline{A}} q a') \simeq (E(q a) \simeq E(q a')) \simeq (1 \simeq 1) \simeq 1$ by univalence of E . ◀

► **Corollary 13** (cf. Anel [5, Remark A.3.3]). *We have univalent completions if and only if we have image factorizations and enough univalent families.*

Proof. If we have univalent completions, then by Example 12 any $f : A \rightarrow B$ has as image $\sum_{b:B} \|\text{Fiber}_f b\|$. Conversely, if we have enough univalent families, then for any $a : A \vdash P$ type we have a univalent $\overline{a} : \overline{A} \vdash \overline{P}(\overline{a})$ type and $(q, e) : P \rightarrow \overline{P}$. Taking an image factorization $q = iq'$, the family $\overline{a}' : \overline{A}' \vdash \overline{P}(i \overline{a}')$ type is a univalent completion of P . ◀

Say a *type family in $\mathcal{U}$* is a pair $(A : \mathcal{U}, P : A \rightarrow \mathcal{U})$. When $\mathcal{U}$ is univalent, $P : A \rightarrow \mathcal{U}$ is univalent if and only if P is an embedding $A \hookrightarrow \mathcal{U}$. Thus:

► **Proposition 14.** *For a type family $P : A \rightarrow \mathcal{U}$ in a univalent universe $\mathcal{U}$, a univalent completion of P in $\mathcal{U}$ consists of an image factorization $A \rightarrow \overline{A} \hookrightarrow \mathcal{U}$ of P with $\overline{A} : \mathcal{U}$.* ◀

10:4 Type-Theoretic Replacement and Univalent Completion

Rijke accordingly uses “univalent completion” to mean just such an image [52, Corollary 4.2.18]. Starting from Proposition 14, we now make formal the correspondence between replacement and univalent completion within a universe. This argument was suggested and explained to us by David Wörn.

► **Definition 15.** *A universe $\mathcal{U}$ has enough univalent families if for every family P in $\mathcal{U}$ there is a cartesian morphism $P \rightarrow \bar{P}$ to a univalent family in $\mathcal{U}$, and has univalent completions when every family in P admits a univalent completion by a family in $\mathcal{U}$.*

► **Remark 16.** Given a universe $\mathcal{U}$, the $\mathcal{U}$ -small types form a model of type theory. If this model has univalent completions, it may not imply that $\mathcal{U}$ has univalent completions as a universe: a surjection among $\mathcal{U}$ -small types need not be a surjection in the larger theory.

► **Proposition 17 (Wörn).** *A univalent universe $\mathcal{U}$ has replacement if and only if it has univalent completions.*

Proof. If $\mathcal{U}$ has replacement, then it has univalent completions by Proposition 14. Suppose $\mathcal{U}$ has univalent completions and let $f : A \rightarrow B$ where $A : \mathcal{U}$ and B is locally $\mathcal{U}$ -small. By Example 12 and local $\mathcal{U}$ -smallness of B , f admits an image $X \equiv \sum_{b:B} \|\text{Fiber}_f b\|$. Write $p : A \rightarrow X$ for the accompanying surjection. We want to show X is $\mathcal{U}$ -small.

Since X is locally $\mathcal{U}$ -small, we have some $F : X \rightarrow \mathcal{U}$ with $F x \simeq \text{Fiber}_p x$. Taking the univalent completion of $P \equiv F \circ p : A \rightarrow \mathcal{U}$, we have $\bar{A} : \mathcal{U}$ and $\bar{P} : \bar{A} \hookrightarrow \mathcal{U}$ with a surjection $q : A \rightarrow \bar{A}$ and $F(p a) \simeq \bar{P}(q a)$ for $a : A$. By the orthogonality of surjections and embeddings [64, Theorem 7.6.7], there is some $r : X \rightarrow \bar{A}$ such that $r \circ p = q$ and $\bar{P} \circ r = F$. We have $X \simeq \sum_{\bar{a}:\bar{A}} \text{Fiber}_r \bar{a}$, so it suffices to show that $\text{Fiber}_r \bar{a}$ is $\mathcal{U}$ -small for $\bar{a} : \bar{A}$. Since being $\mathcal{U}$ -small is a proposition and q is surjective, it suffices to check that $\text{Fiber}_r(q a)$ is $\mathcal{U}$ -small for $a : A$. For any $(x, \alpha) : \text{Fiber}_r(q a)$, we have $(a, \text{refl}) : \text{Fiber}_p(p a)$, which since $\text{Fiber}_p(p a) \simeq F(p a) \simeq \bar{P}(q a) \simeq \bar{P}(r x) \simeq F x$ induces some $\phi_0 a x \alpha : F x$. Now

$$\begin{aligned} \text{Fiber}_r(q a) &\equiv \sum_{x:X} r x =_{\bar{A}} q a \\ &\simeq \sum_{x:X} \sum_{\alpha:r x =_{\bar{A}} q a} \sum_{\phi:F x} (\phi =_{F x} \phi_0 a x \alpha) && \text{(singleton contractibility)} \\ &\simeq \sum_{(x,\phi):\sum_{x:X} F x} \sum_{\alpha:r x =_{\bar{A}} q a} (\phi =_{F x} \phi_0 a x \alpha) && \text{(rearranging)} \end{aligned}$$

Since $\sum_{x:X} F x$ is equivalent to A and thus $\mathcal{U}$ -small, we see that this type is $\mathcal{U}$ -small. ◀

We aim to highlight replacement and univalent completion as foundational type theoretic principles. We begin in § 2 with an expository survey of applications of the two in the literature. In this section we do not claim any novelty. In § 3, we give direct semantics for replacement and univalent completion in cubical models of univalent type theory, bypassing Rijke’s join construction. Finally, in § 4 we observe how the two can be rendered as instances of higher induction recursion.

2 A survey of applications

2.1 Truncations and images

We have seen in Example 12 that a type theory with univalent completions has propositional truncations. In fact, we can construct n -truncations for every $n \geq 1$. Rijke [51, Theorem 7.1] shows this using replacement; we adapt the argument to use univalent completions.

► **Definition 18** (e.g., HoTT Book [64, §7.6]). A map $f : A \rightarrow B$ is n -connected when for every family of n -types P over B , precomposition $(- \circ f) : \prod_{b:B} P b \rightarrow \prod_{a:A} P (f a)$ is an equivalence, and is n -truncated if its fibers are n -truncated. An n -image is a factorization as an n -connected map followed by an n -truncated map. An n -truncation of a type A is an n -image of $A \rightarrow 1$.

► **Definition 19.** $f : A \rightarrow B$ is locally n -connected if $\mathbf{ap}_f^{a,a'}$ is n -connected for all $a, a' : A$.

► **Lemma 20.** If f is surjective and locally n -connected, then f is $(n + 1)$ -connected.

Proof. Let $f : A \rightarrow B$ and let P be a family of $(n + 1)$ -types over B . We show that precomposition $(- \circ f) : \prod_{b:B} P b \rightarrow \prod_{a:A} P (f a)$ is an equivalence by showing that its fibers are contractible. Given $g : \prod_{a:A} P (f a)$, the fiber of $(- \circ f)$ at g is equivalent to $\prod_{b:B} \sum_{p:P b} \prod_{a:A} \prod_{\beta:f a=Bb} p =_{\beta}^g g a$. We show that $\sum_{p:P b} \prod_{a:A} \prod_{\beta:f a=Bb} p =_{\beta}^g g a$ is contractible for all $b : B$. Since f is surjective and being contractible is a proposition, it is enough to check this when b is $f a'$ for some $a' : A$. In this case the type is equivalent to $\sum_{p:P (f a')} \prod_{a:A} \prod_{\alpha:a=Ba'} p =_{\alpha}^{\mathbf{ap}_f^{a,a'}} g a$ by local n -connectedness of f , and the latter type is contractible by two applications of singleton contractibility. ◀

► **Example 21** (Truncations, cf. Rijke [51, Theorem 7.1]). If we have univalent completions, then we have n -truncations for all $n \geq -2$. Relativizing, if a univalent universe $\mathcal{U}$ has replacement, then $\mathcal{U}$ has n -truncations for all $n \geq -2$.

Proof. For $n = -2$, the n -truncation of a type is simply the unit type. For $n = -1$, we gave an argument in Example 12. Suppose we have n -truncations for some $n \geq -1$; we construct the $(n + 1)$ -truncation of A . By assumption, $(a, a') : A \times A \vdash \|a =_A a'\|_n$ type has a univalent completion $u : U \vdash E(u)$ type with $q : A \times A \rightarrow U$. Since $E(q(a, a'))$ is n -truncated and q is surjective, $E(u)$ is n -truncated for all $u : U$. Thus U is $(n + 1)$ -truncated by univalence.

The map $y \equiv \lambda a. \lambda x. q(x, a) : A \rightarrow (A \rightarrow U)$ has an image $A \xrightarrow{p} T \xrightarrow{i} (A \rightarrow U)$ by Corollary 13. Since $A \rightarrow U$ is $(n + 1)$ -truncated and i is (-1) -truncated, T is also $(n + 1)$ -truncated. If $p : A \rightarrow T$ is $(n + 1)$ -connected, then it is our desired $(n + 1)$ -truncation. Since p is surjective, it suffices by Lemma 20 to show that p is locally n -connected. Because i is an embedding, it is equivalent to show that y is locally n -connected.

The map $\mathbf{ap}_y^{a,a'}$ corresponds, by univalence of E and $E(q(a, a')) \simeq \|a =_A a'\|_n$, to the map $a =_A a' \rightarrow \prod_{x:A} (\|x =_A a\|_n \simeq \|x =_A a'\|_n)$ sending $\mathbf{refl}$ to the family of identity equivalences. The codomain is an n -type, so $\mathbf{ap}_y^{a,a'}$ factors through $a =_A a' \rightarrow \|a =_A a'\|_n$; that map is n -connected, so it suffices to show $\|a =_A a'\|_n \rightarrow \prod_{x:A} (\|x =_A a\|_n \simeq \|x =_A a'\|_n)$ is an equivalence. This is an n -truncated version of the Yoneda lemma: by path induction, a family $h : \prod_{x:A} (\|x =_A a\|_n \simeq \|x =_A a'\|_n)$ is uniquely determined by $h a | \mathbf{refl} |_n : \|a =_A a'\|_n$. ◀

Assuming a univalent universe $\mathcal{U}$, Rijke describes the n -truncation of $A : \mathcal{U}$ more directly as the replacement of $\lambda a. \lambda a'. \|a' =_A a\|_n : A \rightarrow (A \rightarrow \mathcal{U})$. As with propositional truncation, n -truncations give us n -images as a corollary [64, Lemma 7.6.4]:

► **Example 22** (Higher images). If we have univalent completions, then we have n -images for all $n \geq -2$. If a univalent universe $\mathcal{U}$ has replacement, then $\mathcal{U}$ has n -images for all $n \geq -2$.

2.2 Set quotients

Rijke's stated motive for assuming the type theoretic replacement axiom in his *Introduction to Homotopy Type Theory* [53] is that it enables the construction of quotients of sets (that is, 0-types) by equivalence relations.

► **Definition 23** (e.g., Rijke [53, Definition 18.2.1]). *For a type A and type-valued relation $(a, a') : A \times A \vdash R(a, a')$ type, a set quotient of A by R is a set A/R and map $[-] : A \rightarrow A/R$ such that (1) $R(a, a')$ implies $[a] =_{A/R} [a']$, and (2) for any set B and $f : A \rightarrow B$ such that $R(a, a')$ implies $f a =_B f a'$ for all $a, a' : A$, there is a unique $f' : A/R \rightarrow B$ with $f =_{A \rightarrow B} f' \circ [-]$.*

► **Example 24** (Set quotients, cf. Rijke [53, §18.1]). If a type theory has univalent completions, then it has set quotients by propositional equivalence relations. Relativizing, if a univalent universe $\mathcal{U}$ has replacement, then $\mathcal{U}$ has set quotients by propositional equivalence relations.

Proof. Let A and a propositional equivalence relation $(a, a') : A \times A \vdash R(a, a')$ type be given. By assumption, R has a univalent completion $u : U \vdash E(u)$ type with $q : A \times A \rightarrow U$.

Take the image of $\lambda a. \lambda x. q(x, a) : A \rightarrow (A \rightarrow U)$ to get $A \xrightarrow{[-]} A/R \xrightarrow{i} (A \rightarrow U)$. We have $([a] =_{A/R} [a']) \simeq (q(-, a) =_{A \rightarrow U} q(-, a')) \simeq (\prod_{x:A} R(x, a) \simeq R(x, a')) \simeq R(a, a')$, where the last step uses that R is an equivalence relation. As $[-]$ is surjective, this makes A/R a set. The equivalence then implies that A/R is the quotient of A by R [53, Theorem 18.2.3]. ◀

► **Example 25** (Set pushouts). If we have univalent completions and inductive families, we can construct the reflexive transitive closure of a propositional relation using truncation. Example 24 then gives us set quotients by arbitrary (that is, non-equivalence) propositional relations, equivalently set-truncated pushouts. The work of Van der Weide and Geuvers [66] shows that all finitary set-truncated higher inductive types can be constructed in this fashion.¹

De Jong and Escardó [27, Theorem 3.29] show that if we restrict our attention to sets, replacement for $\mathcal{U}$ is in fact equivalent to the existence of set quotients by $\mathcal{U}_{\text{prop}}$ -valued equivalence relations in $\mathcal{U}$. Still, “set replacement” is a useful abstraction; compare De Jong and Escardó’s two proofs that the ordinal of ordinals in a universe $\mathcal{U}$ is closed under A -indexed suprema for $A : \mathcal{U}$ [27, Theorems 5.8 and 5.9]. We can similarly characterize univalent completion for sets using a form of descent (cf. Rijke [52, §2]) for set quotients.

► **Definition 26.** *A set quotient A/R has descent when for every propositional family $a : A \vdash P(a)$ type such that $R(a, a')$ implies $P(a) \simeq P(a')$ for all $a, a' : A$, there is a propositional family $c : A/R \vdash \bar{P}(c)$ type with $\bar{P}([a]) \simeq P(a)$ for $a : A$.*

This principle could also be called *large elimination*, although in that tradition (e.g., Smith [58, §5]) one usually deals in judgmental equations rather than equivalences.

► **Proposition 27.** *If every family of propositions admits a cartesian morphism to a univalent family, then every set quotient by a propositional equivalence relation has descent.*

Proof. Let a propositional equivalence relation R be given with set quotient $[-] : A \rightarrow A/R$ and let $a : A \vdash P(a)$ type be a propositional family such that $P(a) \simeq P(a')$ whenever $R(a, a')$. We have a univalent family $u : U \vdash E(u)$ type with $q : A \rightarrow U$ and $E(q a) \simeq P(a)$ for $a : A$. For any $a, a' : A$ such that $R(a, a')$, we have $E(q a) \simeq P(a) \simeq P(a') \simeq E(q a')$ and thus $q a =_U q a'$ by univalence. Hence there is a unique $q' : A/R \rightarrow U$ with $q =_{A \rightarrow U} q' \circ [-]$, from which we define the desired family $x : A/R \vdash \bar{P}(x)$ type by setting $\bar{P}(x) \equiv E(q' x)$. ◀

► **Proposition 28.** *Every propositional family $a : A \vdash P$ type admits a univalent completion if and only if every propositional equivalence relation admits a set quotient with descent.*

¹ Van der Weide and Geuvers assume propositional truncation in addition to set quotients, but the propositional truncation of A can be obtained as its set quotient by the total relation $\lambda\_.\lambda\_ .1$.

Proof. The forward direction is Example 24 and Proposition 27. For the converse, let $a : A \vdash P(a)$ type be a family of propositions. The relation $R(a, a') := P(a) \simeq P(a')$ is valued in propositions, so we have a set quotient $[-] : A \twoheadrightarrow A/R$. By descent, there is a family $c : A/R \vdash \bar{P}(c)$ type such that $\bar{P}([a]) \simeq P(a)$ for $a : A$. To show that $\bar{P}$ is a univalent completion of P , it remains to check that $\bar{P}$ is univalent. By surjectivity of $[-]$, it suffices to check that $[a] =_{A/R} [a'] \rightarrow \bar{P}([a]) \simeq \bar{P}([a'])$ is an equivalence for $a, a' : A$, and since this is a map between propositions it is enough to give a map in the opposite direction. Given $\bar{P}([a]) \simeq \bar{P}([a'])$ we have $P(a) \simeq P(a')$, thus $[a] =_{A/R} [a']$ by definition of R . ◀

2.3 Higher types

So far, we have described constructions that preserve h -level or land in sets. However, univalent completion and replacement can also be used to define operations that raise h -level. In fact, starting from univalent completion and some non-propositional set such as a boolean type, we can construct a non- n -type for every $n \geq -2$.

► **Example 29** (Delooping automorphism types). For a type X , a univalent completion of $\_ : 1 \vdash X$ type is a type $\mathbf{BAut}(X)$ with surjection $q : 1 \twoheadrightarrow \mathbf{BAut}(X)$ and univalent family $z : \mathbf{BAut}(X) \vdash E(z)$ type such that $E(q \star) \simeq X$. This makes $\mathbf{BAut}(X)$ a *delooping* of $X \simeq X$: a 0-connected type whose point $\mathbf{pt} := q \star$ has loop space $X \simeq X$. In the formalism of Buchholtz, Van Doorn, and Rijke [15, §3], $\mathbf{BAut}(X)$ exhibits $X \simeq X$ as the carrier of the automorphism ∞ -group of X . If X lies in a univalent universe $\mathcal{U}$ with replacement, we get $\mathbf{BAut}(X) : \mathcal{U}$ as the image of $\lambda\_ . X : 1 \rightarrow \mathcal{U}$ [15, §4.1].

Given a boolean type $\mathbf{Bool}$, the loop space $\Omega(\mathbf{BAut}(\mathbf{Bool}), \mathbf{pt})$ contains two identities from its basepoint to itself. If the points $\mathbf{true}, \mathbf{false} : \mathbf{Bool}$ are distinct – as follows from a universe or large elimination for $\mathbf{Bool}$ – then $\mathbf{BAut}(\mathbf{Bool})$ is an example of a non-set.

► **Example 30** (Delooping groups). Assume we have univalent completions and let G be a group, in the sense of a set equipped with the usual operations and equations. Writing $|G|$ for its carrier, we have a univalent $z : \mathbf{BAut}(|G|) \vdash E(z)$ type with $E(\mathbf{pt}) \simeq |G|$ per Example 29.

Write $\mathbf{Act}_G(X)$ for the type of left G -actions on a set X . The *principal G -torsor* is the action of G on $|G|$ by multiplication; write $\mathbf{Pr}_G : 1 \rightarrow \sum_{z : \mathbf{BAut}(|G|)} \mathbf{Act}_G(E(z))$ for the map that picks out the corresponding action on $E(\mathbf{pt}) (\simeq |G|)$ and define the *type of G -torsors* $\mathbf{Torsor}(G)$ to be the image of $\mathbf{Pr}_G$. The type of G -torsors is a *delooping of the group G* : a 0-connected 1-groupoid with $\pi_1(\mathbf{Torsor}(G)) \cong G$. Such a type is unique up to equivalence and may also be called the *first Eilenberg–MacLane space*, $K(G, 1)$, associated to G [30]. If G lies in a univalent universe $\mathcal{U}$ with replacement, we can more directly define $\mathbf{Torsor}(G) : \mathcal{U}$ as the image of the principal G -torsor in $\sum_{X : \mathcal{U}} \mathbf{Act}_G(X)$.

As an early example, Bezem, Buchholtz, Grayson, and Shulman [11] defined a circle type in the Rocq UniMath library [71] as the type of $\mathbb{Z}$ -torsors. They do not use replacement or univalent completion; they assume a univalent universe $\mathcal{U}$ to define $\mathbf{Torsor}(\mathbb{Z})$ and do not claim that this type is $\mathcal{U}$ -small. *Symmetry* [10, §5.5] describes the case of a general G and observes that replacement implies that the delooping is $\mathcal{U}$ -small. Licata and Finster [42, §3], who gave the first construction of general Eilenberg–MacLane spaces in a form of HoTT, work with a direct specification of $K(G, 1)$ as a higher inductive type, which they assume lives within a univalent universe.

► **Example 31** (Rezk completion). Example 30 is a special case of *Rezk completion*. Let $\mathcal{C}$ be a category: a type of objects $\mathcal{C}_0$, a hom-set $\mathcal{C}_1(x, y)$ for each $x, y : \mathcal{C}$, and identities and compositions satisfying the usual laws. A Rezk completion of $\mathcal{C}$ is a fully faithful and

essentially surjective functor $i : \mathcal{C} \rightarrow \bar{\mathcal{C}}$ into a *univalent* category $\bar{\mathcal{C}}$: one for which the maps $x = y \rightarrow x \cong y$ from equalities to categorical isomorphisms are equivalences for all $x, y : \bar{\mathcal{C}}$ [64, §9.9] [3]. A delooping of a group G is the Rezk completion of G seen as a 1-object category.

The HoTT Book [64, Theorem 9.9.5] shows that for a category $\mathcal{C}$ with $\mathcal{U}$ -small hom-sets, a Rezk completion can be defined as the image of the Yoneda map $y : \mathcal{C} \rightarrow [\mathcal{C}^{op}, \mathcal{U}_{\text{set}}]$ into the univalent category of $\mathcal{U}_{\text{set}}$ -valued presheaves on $\mathcal{C}$. When $\mathcal{U}$ has replacement, the Rezk completion of a $\mathcal{U}$ -small category is thus $\mathcal{U}$ -small [51, Corollary 5.2]. We can translate this into a construction using univalent completions as in Example 30. Given a category $\mathcal{C}$, take a univalent completion $u : U \vdash E(u)$ type of $(x, y) : \mathcal{C}_0 \times \mathcal{C}_0 \vdash \mathcal{C}_1(x, y)$ type with surjection $q : \mathcal{C}_0 \times \mathcal{C}_0 \rightarrow U$. We have a univalent category $\mathcal{U}_{\mathcal{C}}$ with $(\mathcal{U}_{\mathcal{C}})_0 \equiv U$ and $(\mathcal{U}_{\mathcal{C}})_1(u, u') \equiv (E(u) \rightarrow E(u'))$. The category $[\mathcal{C}^{op}, \mathcal{U}_{\mathcal{C}}]$ of functors from $\mathcal{C}^{op}$ to $\mathcal{U}_{\mathcal{C}}$ is also univalent, and the image of the map $\lambda x. q(-, x) : \mathcal{C} \rightarrow [\mathcal{C}^{op}, \mathcal{U}_{\mathcal{C}}]$ is a Rezk completion of $\mathcal{C}$.

► **Example 32** (Groupoid quotients). Another special case of Rezk completion (and generalization of Example 30) is the *groupoid quotient* of a type A by a groupoid $R : A \rightarrow A \rightarrow \mathcal{U}_{\text{set}}$ (in the sense of a family defining a category (A, R) with all morphisms invertible). Sojakova [60] specifies the groupoid quotient as a higher inductive type; Veltri and Van der Weide [68] point out that its universal property is satisfied by the Rezk completion of the category (A, R) . Veltri and Van der Weide moreover show that a general class of finitary groupoid-truncated higher inductive types can be derived from groupoid quotients assuming inductive families.

We expect that the trend indicated by Examples 24 and 32 continues, i.e., that univalent completion can be used to construct n -groupoid quotients for all n . We will not pursue this here. We will, however, observe individual examples of proper n -types for all n :

► **Example 33** (Eilenberg–MacLane spaces). For a group G , the n th *Eilenberg–MacLane space* [30] $K(G, n)$ is the unique $(n - 1)$ -connected type satisfying $\Omega^n(K(G, n), \text{pt}) \simeq G$. These were first constructed by in homotopy type theory Licata and Finster [42], who used a bespoke higher inductive type for $K(G, 1)$ and truncated suspensions for higher $K(G, n)$. Assuming only a univalent universe $\mathcal{U}$ and that G lives in $\mathcal{U}$, Buchholtz, Christensen, Flatén, and Rijke [14] and Wärrn [73] give alternative constructions of the Eilenberg–MacLane spaces that are $\mathcal{U}$ -small when $\mathcal{U}$ has replacement.

The first space $K(G, 1)$ is the delooping of G from Example 30. For $n \geq 1$, Buchholtz et al. construct $K(G, n + 1) \equiv \sum_{X : \mathcal{U}} \|K(G, n) \simeq X\|_0$ [14, Proposition 5.9]. Replacement implies these are $\mathcal{U}$ -small: any connected and locally $\mathcal{U}$ -small type A is $\mathcal{U}$ -small, as it merely admits a point $1 \rightarrow A$ and is then equivalent to the image of that point. If we assume univalent completion instead of a universe, we can instead define $K(G, n + 1) \equiv \sum_{z : \text{BAut}(K(G, n))} \|K(G, n) \simeq E(z)\|_0$ using the family E from Example 29. Similar arguments apply to Wärrn’s construction [73, Footnote to Definition 1].

In particular, univalent completion and a natural numbers type together imply the existence of types that are not n -truncated for any n , such as $\prod_{n : \mathbb{N}} K(\mathbb{Z}, n)$. We refer to Buchholtz et al. [14] and Wärrn [73] for further applications of replacement to constructing deloopings.

2.4 Higher replacement

The concept of image being an instance of the more general concept of n -image, replacement admits a natural generalization to axiom schemata of “ n -replacement” for $n \geq -2$. Following Christensen [22], say that a type A is *0-locally $\mathcal{U}$ -small* if it is $\mathcal{U}$ -small and $(n + 1)$ -*locally small* when its identity types are n -small.

► **Definition 34.** For $n \geq 2$, we say that $\mathcal{U}$ has n -replacement when every $f : A \rightarrow B$ where A is $\mathcal{U}$ -small and B is $(n+2)$ -locally $\mathcal{U}$ -small admits a $\mathcal{U}$ -small n -image.

Replacement is then (-1) -replacement. Christensen [22, Proposition 2.2] observes that in type theory with pushouts, Rijke’s join construction proves not only (-1) -replacement but n -replacement for all n . Gylterud and Stenholm [35, Proposition 17] abstract from the join construction to derive n -replacement from (-1) -replacement. Their argument assumes an enveloping univalent universe (“Type”), but we can also make do with univalent completions:

► **Example 35** (Higher replacement, cf. Gylterud and Stenholm [35, Proposition 17]). Given univalent completions and a univalent universe $\mathcal{U}$ with replacement (i.e., closed under univalent completions), $\mathcal{U}$ has n -replacement for all $n \geq -2$.

Proof. Let $f : A \rightarrow B$ where A is $\mathcal{U}$ -small and B and $(n+2)$ -locally $\mathcal{U}$ -small. By Example 22, f has an n -image $A \xrightarrow{p} X \xrightarrow{i} B$. Since i is n -truncated, X is also $(n+2)$ -locally $\mathcal{U}$ -small. Thus it is equivalent to show that any $(n+2)$ -locally $\mathcal{U}$ -small type X admitting an n -connected map $p : A \rightarrow X$ from a $\mathcal{U}$ -small type A is $\mathcal{U}$ -small. We show this by induction on n .

The case $n = -2$ is trivial and the case $n = -1$ holds by replacement. For $n \geq 0$, let $p : A \rightarrow X$ be as above. By replacement, it is enough to show that X is locally $\mathcal{U}$ -small. Since p is in particular surjective, we can show that all identity types in X are $\mathcal{U}$ -small by showing that $p a =_X p a'$ is $\mathcal{U}$ -small for all $a, a' : A$. This follows by induction hypothesis, since for $a, a' : A$ the map $\mathbf{ap}_p^{a,a'} : (a =_A a') \rightarrow (p a =_X p a')$ is an $(n-1)$ -connected map from a $\mathcal{U}$ -small type to a $(n-1)$ -locally $\mathcal{U}$ -small type. We use univalent completions for the fact that an n -connected map induces $(n-1)$ -connected maps on identity types: since truncations exist, a map g is k -connected if and only if the k -truncations of its fibers are contractible [64, Lemma 7.5.7], and with this characterization the fact follows from Example 11. ◀

As a consequence (cf. Christensen [22, Theorem 2.6, Proof 2]), given univalent completions and a univalent universe $\mathcal{U}$ with replacement, a type A is $\mathcal{U}$ -small as soon as there is some $n \geq -1$ for which $\|A\|_n$ is $\mathcal{U}$ -small and A is $(n+1)$ -locally $\mathcal{U}$ -small.

2.5 Set theory

Type theoretic replacement is named by analogy to the Replacement Axiom Schema of set theory. In Zermelo–Fraenkel set theory, Replacement states that for any functional first-order formula $\phi(a, b)$ and set A , there is a set B such that $b \in B$ exactly when $\phi(a, b)$ for some $a \in A$. Replacement was proposed, by at least Fraenkel [32] and Skolem [57], after it was observed that Zermelo’s set theory [74] cannot define sets of cardinality $\geq \aleph_\omega$. With Replacement, one can define sets such as $\bigcup\{\beth_0, \beth_1, \dots\}$ that have at least this cardinality: we obtain $\{\beth_0, \beth_1, \dots\}$ by Replacement as the image of $\{0, 1, \dots\}$ under the mapping $\alpha \mapsto \beth_\alpha$. Hallett [36, §8.2] credits Von Neumann [72] with recognizing the importance of Replacement to the theory of transfinite arithmetic. We refer to Van Heijenoort [67], Hallett [36, §8.2], Ferreirós [31, §§XI.1–2], and Kanamori [38] for more of the set theoretic history.

This motivating example in set theory, going from $\{0, 1, \dots\}$ to $\{\beth_0, \beth_1, \dots\}$, does not have an immediate analogue in a structural foundation such as type theory. However, the applications of type theoretic replacement do parallel applications of set theoretic replacement in predicative constructive set theories such as Aczel’s CZF [1]. For example, Aczel and Rathjen [2, Lemma 4.3.5] use Replacement to define quotients, as we have done in Definition 23 and Example 25. Kanamori [38, §1] also highlights the construction of quotient sets as an instance of the “Replacement motif” in classical set theory. Gylterud and Stenholm [34, 35] connect type theoretic and set theoretic replacement in their work on (constructive, predicative) material set theory in type theory:

► **Example 36** (Replacement for iterated sets). Aczel’s universe $V_\infty := W_{A:\mathcal{U}} A$ [1, §3], defined using a W -type [44], is a model of his constructive set theory CZF. Gylterud [34] defines a universe of *iterative sets* in univalent type theory as the subtype V of V_∞ carved out by a predicate $\text{isIterativeSet} : V_\infty \rightarrow \mathcal{U}$, which is defined by induction so that $\text{isIterativeSet}(\sup a f) \equiv \text{isEmbedding } f \times \prod_{a:A} \text{isIterativeSet}(f a)$. The universe V interprets Myhill’s Constructive Set Theory (CST) [46]. Unlike Aczel’s model of CZF in V_∞ , Gylterud’s model interprets equality by V ’s identity type. To show that V models Pairing, Union, and Replacement, Gylterud assumes $\mathcal{U}$ -small images of maps from $\mathcal{U}$ -small types into the locally $\mathcal{U}$ -small V . As V is a set, set quotients in $\mathcal{U}$ would suffice [27, Theorem 3.29].

Gylterud and Stenholm [35] define universes of *iterative n -types* by replacing $\text{isEmbedding } f$ in isIterativeSet with a requirement that f is $(n-1)$ -connected. To prove a form of Replacement for the universe of iterative n -types, they assume type theoretic replacement [35, §4.1] and use in particular that it implies $(n-1)$ -replacement (Example 35).

► **Remark 37.** In some constructive set theories, Replacement is derived from a stronger Collection or Strong Collection axiom schema; see Rathjen [49] for some discussion of the added power of Collection. Rijke and Spitters [54, Definition 4.7] state a Collection axiom for sets in homotopy type theory. Gylterud [34, §7.1] considers two type theoretic renderings of Strong Collection for his V . One follows from type theoretic replacement for $\mathcal{U}$; for the other, he formulates a collection principle for $\mathcal{U}$ that implies it [34, Definition 7.2].

Still, in impredicative foundations one customarily constructs quotients using Power Set. Analogously, Rijke and Spitters [54] observe that Voevodsky’s *propositional resizing* axioms [70] [64, §3.5] imply type theoretic replacement for sets:

► **Proposition 38** (Rijke and Spitters [54, Lemma 3.28]). *If $\mathcal{U}$ contains a univalent universe of all propositions, i.e., a univalent family $p : \Omega \vdash \text{El}_\Omega(p) : \mathcal{U}_{\text{prop}}$ such that every proposition is equivalent to one in Ω , then every $f : A \rightarrow B$ where A is $\mathcal{U}$ -small and B is a set has a $\mathcal{U}$ -small image.*

Proof. Combine the equivalence of set replacement and set quotients [27, Theorem 3.29] with Voevodsky’s construction of set quotients from propositional resizing [4, §2.5] [54, §3.4]. ◀

Contrary to what its set theoretic name suggests, then, type theoretic replacement has no power as an axiom among the sets of type theory with resizing – it is simply a theorem.

► **Remark 39.** Martin-Löf [43] writes of his type theory that “there are axioms for universes (in the sense of category theory) which link the generation of objects and types and play somewhat the same role for the present theory as does the replacement axiom for Zermelo–Fraenkel set theory.” Streicher [62] also emphasizes the connection between Replacement and universes. Montague [45] and Lévy [41, Theorem 6] connect Replacement with *reflection principles*, which can be likened to the existence of enough universes. Returning to the example of $\{\sqsupset_0, \sqsupset_1, \dots\}$, a universe $\mathcal{U}$ enables the definition of a family $\sqsupset : \mathbb{N} \rightarrow \mathcal{U}$ by recursion with $\sqsupset_0 \equiv \mathbb{N}$ and $\sqsupset_{n+1} \equiv (\sqsupset_n \rightarrow 2)$. In the absence of universes, large eliminators à la Smith [58] can play a similar role. Although Rijke’s axiom of replacement is irrelevant to the definability of $\sqsupset$, we have seen that univalent completion (and by analogy replacement) can also be seen as providing “enough universes” with some property, in this case univalence.

3 Replacement and univalent completion in cubical models

Cubical interpretations provide constructive semantics of homotopy type theory [23, 7, 47, 6]. Their univalent universes are closed under pushouts [25, 17], so have replacement by Rijke’s join construction [51, Theorem 4.6]. Here we give more direct constructive justifications for

replacement and univalent completion. Besides providing intuition for the meaning of the principles in these models, these will point to connections to higher inductive and universe constructions in prior work. We first recall background on cubical models in § 3.1, followed by a free algebra construction in § 3.2. In § 3.3 we give an interpretation of (-1) -truncation previously sketched by Cherubini, Coquand, and Hutzler [20, §8.3]. This case study motivates our justifications of replacement in § 3.4 and univalent completion in § 3.5.

3.1 Preliminaries

First, we fix some categorical notation. Given a small category $\mathcal{C}$, we write $\widehat{\mathcal{C}}$ for presheaves on $\mathcal{C}$. For $X \in \widehat{\mathcal{C}}$ and $I \in \mathcal{C}$, we write $X(I) \in \mathbf{Set}$ for the value of X at I . For $f: X \rightarrow Y$ in $\widehat{\mathcal{C}}$, we write $f_I: X(I) \rightarrow Y(I)$ for the value of f at I . We write $\mathbf{y}: \mathcal{C} \rightarrow \widehat{\mathcal{C}}$ for the Yoneda embedding $\mathbf{y}I := \mathcal{C}(-, I)$. A monomorphism $m: X \rightarrow Y$ in $\widehat{\mathcal{C}}$ is *levelwise decidable* when m_I is a decidable monomorphism for $I \in \mathcal{C}$.

For the remainder of §3, we assume the following basic data, with which we can build a homotopical model of type theory following Angiuli, Brunerie, Coquand, Harper, Favonia, and Licata [6].

► **Setting 40.** We fix a finite product category $\mathcal{C}$ whose hom-sets have decidable equality. We require:

- A representable *interval*: an object $\mathbb{I} \in \mathcal{C}$ with disjoint points $0, 1: 1 \rightarrow \mathbb{I}$.
- A *cofibration classifier*: a levelwise decidable mono $\tilde{\Phi} \rightarrow \Phi$ in $\widehat{\mathcal{C}}$, together with maps $(\equiv): \mathbf{y}\mathbb{I} \times \mathbf{y}\mathbb{I} \rightarrow \Phi$, $\perp \in 1 \rightarrow \Phi$, and $(\vee): \Phi \times \Phi \rightarrow \Phi$ fitting in pullback squares

$$\begin{array}{ccccc}
 \mathbf{y}\mathbb{I} & \longrightarrow & \tilde{\Phi} & & 0 & \longrightarrow & \tilde{\Phi} & & (\tilde{\Phi} \times \Phi) \cup (\Phi \times \tilde{\Phi}) & \longrightarrow & \tilde{\Phi} \\
 (\text{id}_{\mathbf{y}\mathbb{I}}, \text{id}_{\mathbf{y}\mathbb{I}}) \downarrow \lrcorner & & \downarrow & & \downarrow \lrcorner & & \downarrow & & \downarrow \lrcorner & & \downarrow \\
 \mathbf{y}\mathbb{I} \times \mathbf{y}\mathbb{I} & \xrightarrow{(\equiv)} & \Phi & & 1 & \xrightarrow{\perp} & \Phi & & \Phi \times \Phi & \xrightarrow{(\vee)} & \Phi
 \end{array}$$

We write $r \equiv r'$ for $(\equiv)_I(r, r')$, likewise $\perp$ for $\perp_I(\star)$ and $\phi \vee \phi'$ for $(\vee)_I(\phi, \phi')$.

- For each $I \in \mathcal{C}$ and $\phi \in \Phi(I)$, a *choice of finite presentation* C_ϕ of ϕ : a finite set $C_\phi \subseteq \mathcal{C}/I$ of morphisms into I such that for every $\alpha: J \rightarrow I$, $\phi\alpha$ belongs to $\tilde{\Phi}(J)$ if and only if $\alpha = \alpha'\beta$ for some $\alpha' \in C_\phi$ and morphism β of $\mathcal{C}$.

A cofibration $\phi \in \Phi(I)$ corresponds by the Yoneda lemma to a morphism $\phi: \mathbf{y}I \rightarrow \Phi$, and pulling back $\tilde{\Phi} \rightarrow \Phi$ along this morphism yields a levelwise decidable mono $\phi^*\tilde{\Phi} \rightarrow \mathbf{y}I$. That is, each $\Phi(I)$ specifies a class of subobjects of $\mathbf{y}I$, which we call *cofibrations*. The operations $(\equiv)$, $\perp$, and $(\vee)$ encode closure properties of this class. The finite presentations tell us that each cofibration $\phi \in \Phi(I)$ has a representation as a finite union of “atomic” cofibrations. We use these to avoid depending on quotients in our metatheory and to ensure that our inductive constructions converge (see Construction 55).

Setting 40 suffices for our purposes, but we would need more to interpret all of univalent type theory; we refer to Angiuli et al. [6] for details. We use Angiuli et al.’s style of model, which is characterized by the use of “diagonal cofibrations” $r \equiv r'$ and a decomposition of filling into *homogeneous filling* and *coercion*. Our constructions adapt straightforwardly to Cohen, Coquand, Huber, and Mörtberg’s style of model [23, 25] and could be described uniformly for both styles using “weak” filling [18], but for simplicity we focus on one style.

One instance of Setting 40 takes $\mathcal{C}$ to be the cartesian cube category, $\mathbb{I}$ to be the 1-cube, and Φ to be the levelwise decidable subobject classifier Ω_{ldc} [6, §3.2]. Here $\Omega_{\text{ldc}}(I)$ is the set of levelwise decidable subobjects of $\mathbf{y}I$; see Orton and Pitts [47, Definition 8.3]. The cube

category admits an Eilenberg–Zilber structure (see, e.g., Campion [16, Theorem 8.12]) which implies that each $\phi \in \Omega_{\text{dec}}(I)$ has a finite presentation by the set of monomorphisms $J \hookrightarrow I$ that factor through ϕ . Another instance, sketched by Coquand and Höfer [24], takes $\mathcal{C}$ to be the category of finite posets with an inductively generated cofibration classifier.

We now describe the interpretation of type theory in $\widehat{\mathcal{C}}$ induced by the data of Setting 40.

► **Definition 41.** A context is an object of $\widehat{\mathcal{C}}$. A (non-fibrant) type over $\Gamma \in \widehat{\mathcal{C}}$ is a presheaf over the category of elements $\int_{\mathcal{C}} \Gamma$; write $\text{Ty}(\Gamma) := \widehat{\int_{\mathcal{C}} \Gamma}$. A term over $A \in \text{Ty}(\Gamma)$ is a global section $a: 1 \rightarrow A$, i.e., a family of elements $a(I, \gamma) \in A(I, \gamma)$ for all $I \in \mathcal{C}$ and $\gamma \in \Gamma(I)$ satisfying $a(I, \gamma)\alpha = a(J, \gamma\alpha)$ for $\alpha: J \rightarrow I$. Write $\text{Ter}(\Gamma; A)$ for the collection of terms of A .

Despite the notation, $\text{Ty}(\Gamma)$ is not the interpretation of the typing judgment. Rather, syntactic types are interpreted as *fibrant types*, which are elements of $\text{Ty}(\Gamma)$ equipped with *filling operations*. Before defining these, we first fix some notation.

► **Notation 42.** For $A \in \text{Ty}(\Gamma)$, we write $\mathbf{p}_A: \Gamma.A \rightarrow \Gamma$ for the extended context and projection and $\mathbf{q}_A \in \text{Ter}(\Gamma.A; \mathbf{A}\mathbf{p}_A)$ for the variable. For $\sigma: \Delta \rightarrow \Gamma$, $A \in \text{Ty}(\Gamma)$, and $t \in \text{Ter}(\Delta; A\sigma)$, we write $(\sigma, t): \Delta \rightarrow \Gamma.A$ for the extended substitution. For $\sigma: \Delta \rightarrow \Gamma$ and $f: A \rightarrow B\sigma$ in $\text{Ty}(\Delta)$, we write $\sigma.f: \Delta.A \rightarrow \Gamma.B$ for the induced substitution.

We use similar notation for “interval variables”. For $I \in \mathcal{C}$, we write $\mathbf{p}_{\mathbb{I}}: I \times \mathbb{I} \rightarrow I$, $\mathbf{q}_{\mathbb{I}}: I \times \mathbb{I} \rightarrow \mathbb{I}$ for the projections; for $\Gamma \in \widehat{\mathcal{C}}$, we write $\mathbf{p}_{\mathbb{I}}: \Gamma \times \mathbf{y}\mathbb{I} \rightarrow \Gamma$, $\mathbf{q}_{\mathbb{I}}: \Gamma \times \mathbf{y}\mathbb{I} \rightarrow \mathbf{y}\mathbb{I}$. For $r: I \rightarrow \mathbb{I}$, we write $\underline{r} := (\text{id}_I, r): I \rightarrow I \times \mathbb{I}$; for $r: \Gamma \rightarrow \mathbf{y}\mathbb{I}$ we write $\underline{r} := (\text{id}_{\Gamma}, r): \Gamma \rightarrow \Gamma \times \mathbf{y}\mathbb{I}$.

The interpretation of types involves the concept of *partial elements*, which are elements of a presheaf or type defined on some subobject of a cube given by a cofibration.

► **Notation 43.** We use the following notations for partial elements:

Sets of partial elements: Given $I \in \mathcal{C}$ and $\phi \in \Phi(I)$, we write $\Gamma(I \upharpoonright \phi)$ for the set of families $(\gamma_{\alpha} \in \Gamma(J) \mid \alpha: J \rightarrow I \text{ s.t. } \phi\alpha \in \widetilde{\Phi}(J))$ satisfying $\gamma_{\alpha}\beta = \gamma_{\alpha\beta}$ for all $K \xrightarrow{\beta} J \xrightarrow{\alpha} I$. Given $\gamma \in \Gamma(I \upharpoonright \phi)$ and $\alpha: J \rightarrow I$, we define $\gamma\alpha \in \Gamma(J \upharpoonright \phi\alpha)$ by $(\gamma\alpha)_{\beta} := \gamma_{\alpha\beta}$. Given $\gamma \in \Gamma(I)$, we define $\gamma \upharpoonright \phi \in \Gamma(I \upharpoonright \phi)$ by $(\gamma \upharpoonright \phi)_{\alpha} := \gamma_{\alpha}$. When ϕ has a finite presentation by some single $\alpha_0: J_0 \rightarrow I$ and we have $\gamma \in \Gamma(J_0)$, we may write $\gamma \in \Gamma(I \upharpoonright \phi)$. Given $A \in \text{Ty}(\Gamma)$, $I \in \mathcal{C}$, $\phi \in \Phi(I)$, and $\gamma \in \Gamma(I \upharpoonright \phi)$, we similarly write $A(I \upharpoonright \phi, \gamma)$ for the set of families $(a_{\alpha} \in A(J, \gamma_{\alpha}) \mid \alpha: J \rightarrow I \text{ s.t. } \phi\alpha \in \widetilde{\Phi}(J))$ such that $a_{\alpha}\beta = a_{\alpha\beta}$ for all α, β .

Definition by cases: For ϕ_0, ϕ_1 and $a_k \in A(I \upharpoonright \phi_k, \gamma)$ for each k , we write $[\phi_0 \mapsto a_0 \mid \phi_1 \mapsto a_1]$ for the unique $a \in A(I \upharpoonright (\phi_0 \vee \phi_1), \gamma)$ with $a \upharpoonright \phi_k = a_k$ for each k , should such an element exist (i.e., if $(a_0)_{\alpha} = (a_1)_{\alpha}$ whenever both sides are defined).

Maps: Given $f: A \rightarrow B$ in $\text{Ty}(\Gamma)$, we write $f_{I \upharpoonright \phi, \gamma}: A(I \upharpoonright \phi, \gamma) \rightarrow B(I \upharpoonright \phi, \gamma)$ for the action on partial elements. In general, when we apply some operation (e.g., a pair projection) to a partial element, we mean to apply the operation levelwise to the family.

Context extension: Given $\Gamma \in \widehat{\mathcal{C}}$ and $\phi: \Gamma \rightarrow \Phi$, we write $\Gamma.\phi \in \widehat{\mathcal{C}}$ for the presheaf $(\Gamma.\phi)(I) := \{\gamma \in \Gamma(I) \mid \phi_I(\gamma) \in \widetilde{\Phi}(I)\}$ and $\mathbf{p}_{\phi}: \Gamma.\phi \rightarrow \Gamma$ for the inclusion.

► **Definition 44.** A filling operation $\mathbf{a}$ for $A \in \text{Ty}(\Gamma)$ is an operation implementing the rule

$$\frac{\gamma \in \Gamma(I \times \mathbb{I}) \quad r \in \mathbf{y}\mathbb{I}(I) \quad a_0 \in A(I, \gamma\underline{r}) \quad \phi \in \Phi(I) \quad a \in A(I \times \mathbb{I} \upharpoonright \phi\mathbf{p}_{\mathbb{I}}, \gamma) \quad a\underline{r} = a_0 \upharpoonright \phi}{\mathbf{a}_{I, \gamma} r a_0 \phi a \in A(I \times \mathbb{I}, \gamma)}$$

and equations $(\mathbf{a}_{I, \gamma} r a_0 \phi a) \upharpoonright \phi = a$ and $(\mathbf{a}_{I, \gamma} r a_0 \phi a)\underline{r} = a_0$ stably under substitution, i.e., with $(\mathbf{a}_{I, \gamma} r a_0 \phi a)(\alpha \times \mathbb{I}) = \mathbf{a}_{J, \gamma(\alpha \times \mathbb{I})} (r\alpha) (a_0\alpha) (\phi\alpha) (a(\alpha \times \mathbb{I}))$ for all $\alpha: J \rightarrow I$.

► **Definition 45.** A fibrant type over $\Gamma \in \widehat{\mathcal{C}}$ is a pair $(A, \mathbf{a})$ of an $A \in \text{Ty}(\Gamma)$ and a filling operation $\mathbf{a}$ for A . We write $\text{Ty}^f(\Gamma)$ for the collection of fibrant types over Γ . We may leave the projection to $\text{Ty}(\Gamma)$ implicit and write A for both a fibrant type and its underlying type.

The filling operation can be decomposed into two simpler operations. This is key to the interpretation of higher inductive types [25, 17] and will be similarly important for us.

► **Definition 46.** A homogeneous filling operation $\mathbf{a}$ for $A \in \text{Ty}(\Gamma)$ is an operation implementing the rule

$$\frac{\gamma \in \Gamma(I) \quad r \in \mathbf{y}\mathbb{I}(I) \quad a_0 \in A(I, \gamma) \quad \phi \in \Phi(I) \quad a \in A(I \times \mathbb{I} \upharpoonright \phi \mathbf{p}_{\mathbb{I}}, \gamma \mathbf{p}_{\mathbb{I}}) \quad a\underline{r} = a_0 \upharpoonright \phi}{\mathbf{a}_{I, \gamma} r a_0 \phi a \in A(I \times \mathbb{I}, \gamma \mathbf{p}_{\mathbb{I}})}$$

and equations $(\mathbf{a}_{I, \gamma} r a_0 \phi a) \upharpoonright \phi = a$ and $(\mathbf{a}_{I, \gamma} r a_0 \phi a)\underline{r} = a_0$ stably under substitution. A coercion operation $\mathbf{a}$ for $A \in \text{Ty}(\Gamma)$ is an operation implementing the rule

$$\frac{\gamma \in \Gamma(I \times \mathbb{I}) \quad r \in \mathbf{y}\mathbb{I}(I) \quad a_0 \in A(I, \gamma \underline{r})}{\mathbf{a}_{I, \gamma} r a_0 \in A(I \times \mathbb{I}, \gamma)}$$

and equation $(\mathbf{a}_{I, \gamma} r a_0)\underline{r} = a_0$ stably under substitution.

A filling operation $\mathbf{a}$ on A immediately restricts to a homogeneous filling operation $\mathbf{a}^h$ and coercion operation $\mathbf{a}^c$. Conversely, given homogeneous filling and coercion operations for $A \in \text{Ty}(\Gamma)$, we can derive a filling operation for A [6, §2.7].

► **Notation 47.** We reuse type-theoretic notation such as Σ and Π for the corresponding operations on semantic types; see [23, §8.2] or [6, §3] for their definitions. For $A \in \text{Ty}(\Gamma)$ and $a_0, a_1 \in \text{Ter}(\Gamma; A)$, $\text{Path}_A a_0 a_1 \in \text{Ty}(\Gamma)$ is the type of paths from a_0 to a_1 : $(\text{Path}_A a_0 a_1)(I, \gamma) = \{p \in A(I \times \mathbb{I}, \gamma \mathbf{p}_{\mathbb{I}}) \mid p\underline{0} = a_0 \text{ and } p\underline{1} = a_1\}$. We use $\text{Path}_A a_0 a_1$ as the semantic analogue of the identity type $a_0 =_A a_1$. The path type at a fibrant A does not quite interpret the computation rule of Martin-Löf's J eliminator, but under appropriate assumptions is equivalent to a type that does [23, §9.1] [17, §3.3]. These basic type constructors give us semantic equivalents of compound types, such as types of fibers and equivalences, which we write in **monospace** (e.g., Fiber_f , $A \simeq B$, $\text{isProp } A$, etc.).

► **Remark 48.** *Cubical type theories* [23, 7] extend Martin-Löf type theory with judgments inspired by the interval and cofibrations of cubical models: an *interval term* judgment $\Gamma \vdash r : \mathbb{I}$, and a *cofibration* judgment $\Gamma \vdash \phi \text{ cof}$ supporting operations $(\equiv)$, $\perp$, and $(\vee)$. Partial elements are specified by restricting contexts by cofibrations: $\Gamma, \phi \vdash a : A$ means A and a are defined on the part of Γ where ϕ holds. Given a total type A and partial $\Gamma, \phi \vdash u : A$, the notation $\Gamma \vdash a : A [\phi \mapsto u]$ means that $\Gamma \vdash a : A$ and $\Gamma, \phi \vdash a \equiv u : A$, i.e., a is a total element extending u . Space does not permit us to develop the cubical type theory corresponding to our semantic arguments, but we will present a few rules that would appear in such a theory in order to aid the reader.

3.2 Constructing free algebras

We interpret (-1) -truncation, replacement, and completion as least cubical sets closed under some constructors. Each can be expressed as a free algebra for a pointed endofunctor in a different category: (-1) -truncation in $\text{Ty}(\Gamma)$, replacement in a slice category $\text{Ty}(\Gamma)/B$, and completion in a category of type families. We describe here a free algebra construction that we can apply in each case, specializing a result in Cavallo and Sattler [19, Theorem 2.3.27, Remark 2.3.8]. This construction uses only colimits in **Set** that can be constructed from coproducts, avoiding a dependency on quotients, and is an alternative to Coquand, Huber, and Mörtberg's approach to constructing higher inductive types without quotients [25].

► **Definition 49.** A wide subcategory $\mathcal{M} \hookrightarrow \mathcal{E}$ has cobase changes if (a) every span of the form $B \leftarrow A \xrightarrow{\in \mathcal{M}} C$ in $\mathcal{E}$ admits a pushout $B \rightarrow D \leftarrow C$, and (b) for every such pushout, the map $B \rightarrow D$ is also in $\mathcal{M}$.

► **Proposition 50.** The wide subcategory of decidable monos in **Set** has cobase changes, and every ω -chain of decidable monos in **Set** has a colimit in **Set**.

Proof. Given a span $B \xleftarrow{f} A \xrightarrow{m} C$ where m is a decidable mono, we have a pushout $B \rightarrow B \sqcup (C \setminus A) \leftarrow C$ where $B \rightarrow B \sqcup (C \setminus A)$ is the coproduct inclusion and $C \rightarrow B \sqcup (C \setminus A)$ is the composite of $C \cong A \sqcup (C \setminus A)$ with $f \sqcup (C \setminus A): A \sqcup (C \setminus A) \rightarrow B \sqcup (C \setminus A)$. If $A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow \dots$ is a chain of decidable monos, then the coproduct $\coprod_{n \in \mathbb{N}} (A_n \setminus A_{n-1})$ (taking A_{-1} to be the empty set) is its colimit. ◀

► **Remark 51.** In set theory, constructing a coproduct $\coprod_{n \in \mathbb{N}} S_n$ uses Replacement, namely to form the set $\{(n, S_n) \mid n \in \mathbb{N}\}$ whose union is that coproduct. Countable Replacement, i.e., Replacement for class functions with domain $\mathbb{N}$, is not provable in Zermelo set theory but is weaker than full Replacement: within ZFC, the Von Neumann universe V_α is a model of Zermelo set theory for any limit ordinal $\alpha > \omega$ (see, e.g., Smullyan and Fitting [59, Chapter 7, Theorem 4.1]) and models Countable Replacement when α has cofinality $> \omega$. In particular, V_{ω_1} (where ω_1 is the first uncountable ordinal) models Countable Replacement.

In a constructive set theory such as CZF, the existence of $\coprod_{n \in \mathbb{N}} S_n$ is likewise guaranteed by Replacement. In a type theoretic metatheory, we must either assume that $\mathbb{N}$ has large elimination or else assume a universe and take **Set** to be the category of sets in that universe.

► **Corollary 52.** For any small category $\mathcal{C}$, the class of levelwise decidable monos in $\widehat{\mathcal{C}}$ has cobase changes and every ω -chain of levelwise decidable monos in $\widehat{\mathcal{C}}$ has a colimit in $\widehat{\mathcal{C}}$.

Proof. Colimits in presheaves are computed levelwise, so this follows from Proposition 50. ◀

► **Construction 53 (Free algebras).** Let $\mathcal{M} \hookrightarrow \mathcal{E}$ be a wide subcategory such that $\mathcal{M}$ is closed under cobase change and unions and every ω -chain in $\mathcal{M}$ has a colimit in $\mathcal{E}$. Let $T = (T, \tau)$ be a pointed endofunctor on $\mathcal{E}$. If T restricts to a pointed endofunctor on $\mathcal{M}$, τ is cartesian, and T preserves colimits of ω -chains in $\mathcal{M}$, then for $A \in \mathcal{E}$ we have a free T -algebra on A .

Proof. Given $A \in \mathcal{E}$, we first define sequences $(A_n \in \mathcal{E})_{n \in \mathbb{N}}$, $(f_n: A_n \xrightarrow{\in \mathcal{M}} A_{n+1})_{n \in \mathbb{N}}$, and $(a_n: TA_n \rightarrow A_{n+1})_{n \in \mathbb{N}}$ such that $a_n \tau_{A_n} = f_n$ and $f_{n+1} a_n = a_{n+1}(Tf_n)$. Set $A_0 := A$, $A_1 := TA$, $f_0 := \tau_A$, and $a_0 := \text{id}_{TA}$. Given A_n , A_{n+1} , f_n , and a_n , take a pushout to define

$$\begin{array}{ccc} A_{n+1} \sqcup_{A_n} TA_n & \xrightarrow{[\tau_{A_{n+1}}, Tf_n]} & TA_{n+1} \\ \downarrow [\text{id}_{A_{n+1}}, a_n] & & \downarrow a_{n+1} \\ A_{n+1} & \xrightarrow{f_{n+1}} & A_{n+2} \end{array}$$

where the upper left object is the pushout of $A_{n+1} \xleftarrow{f_n} A_n \xrightarrow{\tau_A} TA_n$. The bottom right pushout exists because $[\tau_{A_{n+1}}, Tf_n]$ is the union of $\tau_{A_{n+1}}$ and Tf_n and thus in $\mathcal{M}$. By closure under cobase change, f_{n+1} also belongs to $\mathcal{M}$ as required.

Set $A_\omega := \text{colim}_{n < \omega} A_n$. By the assumption that T preserves colimits of ω -chains in $\mathcal{M}$, there is a unique $a_\omega: TA_\omega \rightarrow A_\omega$ such that $a_\omega(T\iota_n) = \iota_{n+1} a_n$ for all $n \in \mathbb{N}$. It follows from the universal property that $a_\omega \tau_{A_\omega} = \text{id}_{A_\omega}$, making (A_ω, a_ω) a T -algebra.

We claim that $\iota_0: A \rightarrow A_\omega$ exhibits (A_ω, a_ω) as the free T -algebra on A . Suppose that $(B, b: TB \rightarrow B)$ is a T -algebra and we have some $g: A \rightarrow B$. We first show by induction that there is a unique sequence $(g_n: A_n \rightarrow B)_{n \in \mathbb{N}}$ such that $g_0 = g$ and $g_{n+1} a_n = b(Tg_n)$

(which also implies $g_{n+1}f_n = g_n$) for all $n \in \mathbb{N}$. Set $g_0 := g$ and $g_1 := b(Tg_0)$. Given unique g_i for $i \leq n+1$, the universal property of A_{n+2} induces a unique $g_{n+2}: A_{n+2} \rightarrow B$ such that $g_{n+2}f_{n+1} = g_{n+1}$ and $g_{n+2}a_{n+1} = b(Tg_{n+1})$. Now there is a unique $g_\omega: A_\omega \rightarrow B$ such that $g_\omega \iota_n = g_n$ for all $n \in \mathbb{N}$. This g_ω defines a morphism of T -algebras from $(A_\omega, a_\omega) \rightarrow (B, b)$, and its uniqueness follows from uniqueness of the g_n . $\blacktriangleleft$

By definition, the free algebras of Construction 53 are preserved by any functor that preserves the relevant colimits. We will use the following in particular to check that our interpretations of replacement and completion are stable under the substitution.

► **Proposition 54.** *For $k \in \{0, 1\}$, let $\mathcal{M}_k \hookrightarrow \mathcal{E}_k$ and $\mathsf{T}_k = (T_k, \tau_k)$ on $\mathcal{E}_k$ be given satisfying the hypotheses of Construction 53. If $F: \mathcal{E}_0 \rightarrow \mathcal{E}_1$ preserves the assumed colimits in $\mathcal{E}_0$ and we have $\theta: FT_0 \cong T_1F$ with $\theta \circ F\tau_0 = \tau_1F$, then F sends free T_0 -algebras to free T_1 -algebras. If F strictly preserves the assumed choices of colimits and θ is an identity, then F strictly sends the constructed free T_0 -algebra on $A \in \mathcal{E}_0$ to the constructed free T_1 -algebra on FA . $\blacktriangleleft$*

3.3 Propositional truncation

We motivate our semantics of univalent completion and replacement with a special case: (-1) -truncation. We recall an interpretation of (-1) -truncation that first appeared in Cherubini, Coquand, and Hutzler [20, §8.3] and has also been used by Bocquet [12, §2.5]. Said interpretation can be translated to a type former of cubical type theory, in which case its constructors are given as follows:

$$\frac{a : A}{|a| : \|A\|} \qquad \frac{t_0 : \|A\| \quad \phi \text{ cof} \quad \phi \vdash t : \|A\|}{\text{ext } t_0 [\phi \mapsto t] : \|A\| [\phi \mapsto t]}$$

That is, every element of A gives an element of $\|A\|$, and if $\|A\|$ is inhabited by some $t_0 : \|A\|$ then every partial element $\phi \vdash t : \|A\|$ extends to a total element $\text{ext } t_0 [\phi \mapsto t] : \|A\|$. Cohen, Coquand, Huber, and Mörtberg [23, §5.1] show that a fibrant type B is contractible if and only if every partial element in B extends to a total element. Thus, by the equivalence $\text{isProp } B \simeq (B \rightarrow \text{isContr } B)$, a type B is a proposition if and only if for every point of B , every partial element in B can be extended to a total element [20, §8.3].

We now translate this type-theoretic description into a semantic definition using Construction 53. After constructing $\|A\|$ as a cubical set, we first prove it is fibrant and then that it supports the universal property (i.e., eliminator) for the (-1) -truncation.

► **Construction 55 (Truncation).** For every $A \in \text{Ty}(\Gamma)$, we have $\|A\| \in \text{Ty}(\Gamma)$ generated by

$$\frac{a \in A(I, \gamma)}{\text{in } a \in \|A\|(I, \gamma)} \qquad \frac{t_0 \in \|A\|(I, \gamma) \quad \phi \in \Phi(I) \quad t \in \|A\|(I \upharpoonright \phi, \gamma)}{\text{ext } t_0 \phi t \in \|A\|(I, \gamma)}$$

$$\text{ext } t_0 \phi t \upharpoonright \phi = t \qquad (\text{in } a)\alpha = \text{in } (a\alpha) \qquad (\text{ext } t_0 \phi t)\alpha = \text{ext } (t_0\alpha) (\phi\alpha) (t\alpha)$$

in the sense that every $B \in \text{Ty}(\Gamma)$ interpreting this signature admits a unique interpretation-preserving $\|A\| \rightarrow B$. The construction is stable under substitution into Γ .

Proof. For every $A \in \text{Ty}(\Gamma)$ and $(I, \gamma) \in \int_e \Gamma$, define

$$TA(I, \gamma) := A(I, \gamma) \sqcup \{(a_0, \phi, a) \mid a_0 \in A(I, \gamma), \phi \in \Phi(I) \setminus \tilde{\Phi}(I), a \in A(I \upharpoonright \phi, \gamma)\}.$$

Write $\iota_0(a)$ for elements in the left summand and $\iota_1(a_0, \phi, a)$ for elements in the right summand. The $TA(I, \gamma)$ define a $TA \in \text{Ty}(\Gamma)$ with action by $\alpha: J \rightarrow I$ given by $\iota_0(a)\alpha := \iota_0(a\alpha)$, $\iota_1(a_0, \phi, a)\alpha := \iota_0(a_\alpha)$ if $\phi\alpha \in \tilde{\Phi}(J)$, and $\iota_1(a_0, \phi, a)\alpha := \iota_1(a_0\alpha, \phi\alpha, a\alpha)$ otherwise.

The mapping $A \mapsto TA$ extends to a pointed endofunctor $\mathbf{T} = (T, \tau)$ with unit $\tau_A: A \rightarrow TA$ given by the inclusions ι_0 . The unit is levelwise decidable and cartesian. Thanks to our choices of finite presentations, any levelwise decidable $A \rightarrow B$ induces decidable monos $TA(I, \gamma) \rightarrow TB(I, \gamma)$: we can decide if an element $\iota_1(b_0, \phi, b)$ in TB belongs to TA by testing whether b_0 belongs to A and whether the components of b on the finite presentation for ϕ belong to A . Finally, if $A: \omega \rightarrow \text{Ty}(\Gamma)$ is an ω -chain of levelwise decidable monos, then the maps $\text{colim}_{n < \omega} TA_n(I, \gamma) \rightarrow T(\text{colim}_{n < \omega} A_n)(I, \gamma)$ are isomorphisms: given $\iota_0(a)$ we have some $n < \omega$ for which a lies in a_n , and given $\iota_1(a_0, \phi, a) \in T(\text{colim}_{n < \omega} A_n)(I, \gamma)$ we can use our finite presentation for ϕ to identify some $n < \omega$ for which both a_0 and a lie in A_n .

Apply Construction 53 with $\mathbf{T}$ and the class of levelwise decidable monos (using Corollary 52) and take $\|A\| \in \text{Ty}(\Gamma)$ to be the free $\mathbf{T}$ -algebra on $A \in \text{Ty}(\Gamma)$. The map $A \rightarrow \|A\|$ provides the constructor **in**. We define **ext** $t_0 \phi t$ to be t_{id} when $\phi \in \tilde{\Phi}(I)$ and use the $\mathbf{T}$ -algebra structure when $\phi \in \Phi(I) \setminus \tilde{\Phi}(I)$. Stability under substitution is Proposition 54. ◀

► **Remark 56.** This is an alternative to Cohen, Coquand, Huber, and Mörtberg’s semantics for (-1) -truncation [23, 25], in which the role of **ext** is played by two constructors: **squash**, which introduces a path between each pair of elements, and **hcomp**, which implements homogeneous filling directly. The merging of these roles in **ext** parallels Cohen et al.’s use of **Glue types** to interpret the univalence axiom and fibrancy of the universe, as opposed to Angiuli, Favonia, and Harper’s [7] use of separate **V types** [7, §5.6] (for univalence) and **hcom types** [7, §6] (for fibrancy). This parallel will show itself more clearly in our semantics of univalent completion.

► **Lemma 57.** For $A \in \text{Ty}(\Gamma)$ we have a homogeneous filling operation $\mathfrak{h}$ on $\|A\|$, stably under substitution.

Proof. Define $\mathfrak{h}_{I, \gamma} r t_0 \phi t := \text{ext } (t_0 \mathbf{p}_{\mathbb{I}}) (\phi \mathbf{p}_{\mathbb{I}}) t$. ◀

► **Notation 58.** For $A \in \text{Ty}(\Gamma \times \mathbf{y}\mathbb{I})$, write $\prod_{\mathbb{I}} A \in \text{Ty}(\Gamma)$ for dependent product over the interval, $(\prod_{\mathbb{I}} A)(I, \gamma) := A(I \times \mathbb{I}, \gamma \mathbf{p}_{\mathbb{I}})$. Write $\text{ev}_r: (\prod_{\mathbb{I}} A) \rightarrow A_r$ for evaluation at $r: \Gamma \rightarrow \mathbf{y}\mathbb{I}$.

► **Lemma 59.** For each $A \in \text{Ty}(\Gamma \times \mathbf{y}\mathbb{I})$, coercion operation $\mathbf{a}$ for A , and $r: \Gamma \rightarrow \mathbf{y}\mathbb{I}$, we have a choice of section $\text{coe}^{A, r}: \|A\|_r \rightarrow \prod_{\mathbb{I}} \|A\|$ of ev_r in $\text{Ty}(\Gamma)$, stably under substitution. As a corollary, we have a coercion operation $\mathbf{c}$ on $\|A\|$ defined by $\mathbf{c}_{I, \gamma} r t_0 := \text{coe}_{I, \text{id}_I}^{A, r}(t_0)$.

Proof. We define $\text{coe}^{A, r}$ by universal property of $\|A\|_r$, i.e., recursion: set $\text{coe}_{I, \gamma}^{A, r}(\mathbf{in } a) := \mathbf{in } (\mathbf{a}_{I, \gamma}(r_I(\gamma)) a)$ and $\text{coe}_{I, \gamma}^{A, r}(\text{ext } t_0 \phi t) := \text{ext } (\text{coe}_{I, \gamma}^{A, r}(t_0)) (\phi \mathbf{p}_{\mathbb{I}}) (\text{coe}_{I \upharpoonright \phi, \gamma}^{A, r}(t))$. ◀

► **Theorem 60** (Fibrancy of truncation). Construction 55 extends to a substitution-stable $\|-\|: \text{Ty}^f(\Gamma) \rightarrow \text{Ty}^f(\Gamma)$. ◀

For $\|A\|$ to be the (-1) -truncation of A , it suffices that $\|A\|$ is a proposition and that for any fibrant proposition P with $A \rightarrow P$, we have $\|A\| \rightarrow P$ [53, Remark 14.1.3].

► **Lemma 61** (Cherubini, Coquand, and Hutzler [20, §8.3]). For every $A \in \text{Ty}^f(\Gamma)$, each $p \in \text{Ter}(\Gamma; \mathbf{isProp } A)$ induces an operation ext_p sending $a_0 \in A(I, \gamma)$, $\phi \in \Phi(I)$, and $a \in A(I \upharpoonright \phi, \gamma)$ to an element $\text{ext}_p a_0 \phi a \in A(I, \gamma)$ satisfying $(\text{ext}_p a_0 \phi a) \upharpoonright \phi = a$. Conversely, any such operation induces an inhabitant of $\text{Ter}(\Gamma; \mathbf{isProp } A)$. ◀

The **ext** constructor thus makes $\|A\|$ a proposition. For mapping out, we have:

► **Theorem 62** (Universal property of $\|A\|$). *For every $A \in \text{Ty}(\Gamma)$ and proposition $P \in \text{Ty}^f(\Gamma)$ with a map $A \rightarrow P$, we have a map $\|A\| \rightarrow P$, stably under substitution.*

Proof. Let $P \in \text{Ty}^f(\Gamma)$, $p \in \text{isProp } A$, and $f : A \rightarrow P$. We define $f' : \|A\| \rightarrow P$ by recursion, setting $f'_{I,\gamma}(\text{in } a) := f_{I,\gamma}(a)$ and $f'_{I,\gamma}(\text{ext } t_0 \phi t) := \text{ext}_p(f'_{I,\gamma}(t)) \phi (f'_{I|\phi,\gamma}(t))$. ◀

3.4 Replacement

Following the pattern for (-1) -truncation, we will interpret replacement of a map $f : A \rightarrow B$ as the closure of A under an “extension” operation. In a cubical type theory, given B type, $\beta : \text{LocSmall}_{\mathcal{U}} B$, $A : \mathcal{U}$, and $f : A \rightarrow B$, the type $\text{Rep}_{B,\beta}(A, f) : \mathcal{U}$ would be specified beginning as follows:

$$\frac{t : \text{Rep}_{B,\beta}(A, f)}{\text{out } t : B} \quad \frac{a : A}{\text{in } a : \text{Rep}_{B,\beta}(A, f)} \quad \text{out } (\text{in } a) \equiv f a : B$$

$$\frac{t_0 : \text{Rep}_{B,\beta}(A, f) \quad \phi \text{ cof} \quad \phi \vdash w : \sum_{t : \text{Rep}_{B,\beta}(A, f)} (\beta (\text{out } t) (\text{out } t_0)).1}{\text{ext } t_0 [\phi \mapsto w] : \text{Rep}_{B,\beta}(A, f) [\phi \mapsto w.1]}$$

$$\text{out } (\text{ext } t_0 [\phi \mapsto w]) \equiv \text{hcom}_{B}^{0 \rightarrow 1} [\phi \mapsto \dots] (\dots) : B$$

Here the type $(\beta (\text{out } t) (\text{out } t_0)).1$ is a $\mathcal{U}$ -small equivalent to $\text{out } t =_B \text{out } t_0$. The reduction of $\text{out } (\text{ext } t_0 [\phi \mapsto w])$ and hcom uses homogeneous filling in B [6, §2.7], as we see below. Now ext can be roughly understood as forcing the fiber types $\sum_{t : \text{Rep}_{B,\beta}(A, f)} \text{out } t =_B \text{out } t_0$ to be propositions, which has the effect of making out an embedding.

To state replacement, we first need a universe; to interpret type theory with a universe, we assume a universe in our metatheory.

► **Assumption 63** (for § 3.4). We assume a fixed $\mathcal{V} \in \mathbf{Set}$ such that the class of $\mathcal{V}$ -small sets (that is, sets isomorphic to an element of $\mathcal{V}$) is closed under finite limits and coproducts indexed by $\mathcal{V}$ -small sets and contains all finite sets, $\mathbb{N}$, and $\Phi(I) \setminus \tilde{\Phi}(I)$ for all $I \in \mathbb{C}$.

► **Definition 64.** For $\Gamma \in \widehat{\mathbb{C}}$, write $\text{Ty}_{\mathcal{V}}(\Gamma)$ for the category of presheaves on $\int_{\mathbb{C}} \Gamma$ valued in elements of $\mathcal{V}$, and $\text{Ty}_{\mathcal{V}}^f(\Gamma)$ for the category of such presheaves equipped with a filling operation. Write $\mathcal{U}_{\mathcal{V}} \in \text{Ty}(\Gamma)$ for the universe of fibrant types $\mathcal{U}_{\mathcal{V}}(I, \_) := \text{Ty}_{\mathcal{V}}^f(\mathbf{y}I)$. Given $A \in \text{Ter}(\Gamma; \mathcal{U}_{\mathcal{V}})$, write $\text{El}_{\mathcal{V}} A \in \text{Ty}(\Gamma)$ for its decoding. Abbreviate $\text{LocSmall}_{\mathcal{V}} := \text{LocSmall}_{\mathcal{U}_{\mathcal{V}}}$.

Assumption 63 is enough to interpret replacement in $\mathcal{U}_{\mathcal{V}}$, but closing $\mathcal{U}_{\mathcal{V}}$ under other type formers may require more. Additional assumptions on Φ are also needed to make $\mathcal{U}_{\mathcal{V}}$ fibrant and univalent; see Cohen et al. [23, §8.2] or Angiuli et al. [6, §§2.12 & 3.5]. We do not need univalence of $\mathcal{U}_{\mathcal{V}}$ here, but, as we have seen, most *uses* of replacement depend on it.

► **Definition 65.** Given $(A, f) \in \text{Ty}_{\mathcal{V}}(\Gamma)/B$ and $\beta \in \text{Ter}(\Gamma; \text{LocSmall}_{\mathcal{V}} B)$, define the small family of fibers $(\text{Fib}_{\mathcal{V}} \beta f)(b) := \sum_{a:A} \text{El}_{\mathcal{V}} ((\beta (f a) b).1) \in \text{Ty}_{\mathcal{V}}(\Gamma.B)$. Given $a \in A(I, \gamma)$ and $p \in B(I \times \mathbb{I}, \gamma_{\mathbb{P}\mathbb{I}})$ with $p\mathbf{0} = f_{I,\gamma}(a)$, write $\langle a, p \rangle \in (\text{Fib}_{\mathcal{V}} \beta f)(I, (\gamma, p\mathbf{1}))$ for the corresponding element of the small fiber. Given $w \in (\text{Fib}_{\mathcal{V}} \beta f)(I, (\gamma, b))$, write $\text{ptOf } w \in A(I, \gamma)$ and $\text{pathOf } w \in B(I \times \mathbb{I}, \gamma_{\mathbb{P}\mathbb{I}})$ for the corresponding point and path.

► **Construction 66** (Replacement). Let $(B, \mathbf{b}) \in \mathrm{Ty}^f(\Gamma)$ and $\beta \in \mathrm{Ter}(\Gamma; \mathrm{LocSmall}_{\mathcal{V}} B)$. For every pair $(A, f) \in \mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$, we have $(\mathbf{Rep}_{B,\beta}(A, f), \mathbf{out}) \in \mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$ generated by

$$\frac{a \in A(I, \gamma)}{\mathbf{in} \ a \in (\mathbf{Rep}_{B,\beta}(A, f))(I, \gamma)} \quad \frac{\begin{array}{l} t_0 \in (\mathbf{Rep}_{B,\beta}(A, f))(I, \gamma) \\ \phi \in \Phi(I) \quad w \in (\mathbf{Fib}_{\mathcal{V}} \beta \ \mathbf{out})(I \upharpoonright \phi, (\gamma, \mathbf{out}_{I,\gamma}(t_0))) \end{array}}{\mathbf{ext} \ t_0 \ \phi \ w \in (\mathbf{Rep}_{B,\beta}(A, f))(I, \gamma)}$$

$$(\mathbf{in} \ a)\alpha = \mathbf{in} \ (a\alpha) \quad (\mathbf{ext} \ t_0 \ \phi \ w)\alpha = \mathbf{ext} \ (t_0\alpha) \ (\phi\alpha) \ (w\alpha) \quad (\mathbf{ext} \ t_0 \ \phi \ w) \upharpoonright \phi = w.1$$

$$\mathbf{out}_{I,\gamma}(\mathbf{in} \ a) = f_{I,\gamma}(a) \quad \mathbf{out}_{I,\gamma}(\mathbf{ext} \ t_0 \ \phi \ w) = (\mathbf{b}_{I,\gamma}^h \ 1 \ (\mathbf{out}_{I,\gamma}(t_0)) \ \phi \ (\mathbf{pathOf} \ w))\mathbf{0}$$

in the sense that each $F \in \mathrm{Ty}(\Gamma)/B$ interpreting this signature admits a unique interpretation-preserving map $(\mathbf{Rep}_{B,\beta}(A, f), \mathbf{out}) \rightarrow F$. The construction is stable under substitution.

Proof. For $(A, f) \in \mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$, take $\partial(A, f)(I, \gamma) \in \mathcal{V}$ for $(I, \gamma) \in \int_{\mathcal{C}} \Gamma$ to be the set of triples (a_0, ϕ, w) for $a_0 \in A(I, \gamma)$, $\phi \in \Phi(I) \setminus \tilde{\Phi}(I)$, and $w \in (\mathbf{Fib}_{\mathcal{V}} \beta \ f)(I \upharpoonright \phi, (\gamma, f_{I,\gamma}(a_0)))$. To get $\partial(A, f)(I, \gamma)$ in $\mathcal{V}$, we use our choices of finite presentations and the closure of $\mathcal{V}$ -small sets under $(\Phi(I) \setminus \tilde{\Phi}(I))$ -indexed coproducts and finite limits. We define $T_0(A, f) \in \mathrm{Ty}_{\mathcal{V}}(\Gamma)$ by setting $T_0(A, f)(I, \gamma) := A(I, \gamma) \sqcup \partial(A, f)(I, \gamma) \in \mathcal{V}$ and defining the action by $\alpha: J \rightarrow I$ by $\iota_0(a)\alpha := \iota_0(a\alpha)$, $\iota_1(a_0, \phi, w)\alpha := \iota_0(w\alpha.1)$ if $\phi\alpha \in \tilde{\Phi}(J)$, and $\iota_1(a_0, \phi, w)\alpha := \iota_1(a_0\alpha, \phi\alpha, w\alpha)$ if $\phi\alpha \notin \tilde{\Phi}(J)$. We then define $T_1(A, f): T_0(A, f) \rightarrow B$ by cases: $T_1(A, f)_{I,\gamma}(\iota_0(a)) := f_{I,\gamma}(a)$ and $T_1(A, f)_{I,\gamma}(\iota_1(t_0, \phi, w)) := (\mathbf{b}_{I,\gamma}^h \ 1 \ (\mathbf{out}_{I,\gamma}(t_0)) \ \phi \ (\mathbf{pathOf} \ w))\mathbf{0}$.

We get a pointed endofunctor $\mathbf{T} = (T, \tau)$ on $\mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$ with $T(A, f) := (T_0(A, f), T_1(A, f))$ and point τ given by the inclusions ι_0 . Write $\mathcal{M} \hookrightarrow \mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$ for the wide subcategory of $m: (A, f) \rightarrow (A', f')$ such that $m: A \rightarrow A'$ is a levelwise decidable mono in $\mathrm{Ty}_{\mathcal{V}}(\Gamma)$ with complements $A'(I, \gamma) \setminus A(I, \gamma)$ in $\mathcal{V}$. As colimits in $\mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$ are computed in $\mathrm{Ty}_{\mathcal{V}}(\Gamma)$, it follows from the proof of Corollary 52 and our assumptions on $\mathcal{V}$ that $\mathcal{M}$ is closed under cobase change and every ω -chain in $\mathcal{M}$ has a colimit in $\mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$. The unit τ is cartesian and valued in $\mathcal{M}$. As in Construction 55, T preserves $\mathcal{M}$ and colimits of ω -chains in $\mathcal{M}$.

We can thus apply Construction 53 with $\mathcal{M}$ and $\mathbf{T}$. Take $(\mathbf{Rep}_{B,\beta}(A, f), \mathbf{out}) \in \mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$ to be the free $\mathbf{T}$ -algebra on (A, f) , interpreting the signature as in Construction 55. As the inclusion of $\mathrm{Ty}_{\mathcal{V}}(\Gamma)/B$ into $\mathrm{Ty}(\Gamma)/B$ preserves the relevant colimits, it follows from Proposition 54 that $(\mathbf{Rep}_{B,\beta}(A, f), \mathbf{out})$ is also the free object supporting the required signature in $\mathrm{Ty}(\Gamma)/B$. Stability under substitution also follows from Proposition 54. ◀

Construction 66 brings to mind Dybjer and Setzer's description of inductive recursive definitions as initial algebras in slice categories [29]; we return to this connection in § 4.

► **Remark 67.** Like the **ext** constructor of the (-1) -truncation, the **ext** constructor for the replacement has two purposes: to provide a homogeneous filling operation and to make **out** into an embedding. Like the (-1) -truncation (see Remark 56), there is an alternative construction that divides **ext** into two constructors: one which reflects paths in B between elements in the image of **out** to paths in $\mathbf{Rep}_{B,\beta}(A, f)$, and one which closes $\mathbf{Rep}_{B,\beta}(A, f)$ under homogeneous filling. We leave the formulation of this alternative as an exercise.

To define homogeneous filling with **ext**, we use the following path-scaling operation.

► **Lemma 68.** Let $(A, \mathbf{a}) \in \mathrm{Ty}^f(\Gamma)$. Given $\gamma \in \Gamma(I)$, $r \in \mathbf{y}\mathbb{I}(I)$, and $p \in A(I \times \mathbb{I}, \gamma \mathbf{p}_{\mathbb{I}})$, we have $\mathbf{scale} \ \mathbf{a} \ r \ p \in A(I \times \mathbb{I} \times \mathbb{I}, \gamma \mathbf{p}_{\mathbb{I}}\mathbf{p}_{\mathbb{I}})$ satisfying $(\mathbf{scale} \ \mathbf{a} \ r \ p)\mathbf{0} = p$, $(\mathbf{scale} \ \mathbf{a} \ r \ p)\mathbf{1} = \mathbf{prp}_{\mathbb{I}}$, and $(\mathbf{scale} \ \mathbf{a} \ r \ p)(\underline{r} \times \mathbb{I}) = \mathbf{prp}_{\mathbb{I}}$, stably under substitution.

Proof. Define $\text{scale } a \ r \ p$ to be the reindexing by $I \times \langle (.2), (.1) \rangle : I \times \mathbb{I} \times \mathbb{I} \rightarrow I \times \mathbb{I} \times \mathbb{I}$ of $\mathbf{a}_{I \times \mathbb{I}, \gamma \mathbf{p}_\mathbb{I}}^h (r \mathbf{p}_\mathbb{I}) (p \mathbf{r} \mathbf{p}_\mathbb{I}) (q_\mathbb{I} \equiv 0 \vee q_\mathbb{I} \equiv 1) [(q_\mathbb{I} \equiv 0) \mathbf{p}_\mathbb{I} \mapsto p \mid (q_\mathbb{I} \equiv 1) \mathbf{p}_\mathbb{I} \mapsto p \mathbf{r} \mathbf{p}_\mathbb{I}]$. ◀

► **Lemma 69.** *Let $(B, \mathbf{b}) \in \text{Ty}^f(\Gamma)$ and $\beta \in \text{Ter}(\Gamma; \text{LocSmall}_V B)$. For $(A, f) \in \text{Ty}_V(\Gamma)/B$ we have a homogeneous filling operation $\mathbf{h}$ on $\text{Rep}_{B, \beta}(A, f)$, stably under substitution.*

Proof. In order to define $\mathbf{h}_{I, \gamma} \ r \ t_0 \ \phi \ t$, we first combine t_0 and t into a single element $t_+ := [q_\mathbb{I} \equiv r \mathbf{p}_\mathbb{I} \mapsto t_0 \mathbf{p}_\mathbb{I} \mid \phi \mathbf{p}_\mathbb{I} \mapsto t] \in \text{Rep}_{B, \beta}(A, f)(I \times \mathbb{I} \upharpoonright (q_\mathbb{I} \equiv r \mathbf{p}_\mathbb{I} \vee \phi \mathbf{p}_\mathbb{I}), \gamma \mathbf{p}_\mathbb{I})$. We then take $\mathbf{h}_{I, \gamma} \ r \ t_0 \ \phi \ t := \text{ext } (t_0 \mathbf{p}_\mathbb{I}) (q_\mathbb{I} \equiv r \mathbf{p}_\mathbb{I} \vee \phi \mathbf{p}_\mathbb{I}) (t_+, \text{scale } \mathbf{b} \ r \ (\text{out } t_+))$. ◀

► **Notation 70.** We use the following notation for coercion inside of $\mathcal{U}_V$: for $(A, \mathbf{a}) \in \mathcal{U}_V(I \times \mathbb{I}, \gamma)$, $r \in \mathbf{y}\mathbb{I}(I)$, and $a_0 \in A(I, \gamma \mathbf{r})$, write $\text{coe}_{\in V}^r(A, \mathbf{a}) \ a_0 := \mathbf{a}_{I, \text{id}_{I \times \mathbb{I}}}^c \ r \ a_0 \in A(I \times \mathbb{I}, \gamma)$.

► **Lemma 71.** *Let $(B, \mathbf{b}) \in \text{Ty}^f(\Gamma \times \mathbf{y}\mathbb{I})$ and $\beta \in \text{Ter}(\Gamma \times \mathbf{y}\mathbb{I}; \text{LocSmall}_V B)$. For every $(A, f) \in \text{Ty}_V(\Gamma \times \mathbf{y}\mathbb{I})/B$, coercion operation $\mathbf{a}$ for A , and $r : \Gamma \rightarrow \mathbf{y}\mathbb{I}$, we have a choice of section $\text{coe}^r : (\text{Rep}_{B, \beta}(A, f))_{\mathbf{r}} \rightarrow \prod_{\mathbb{I}} \text{Rep}_{B, \beta}(A, f)$ of ev_r in $\text{Ty}(\Gamma)$, stably under substitution. As a corollary, we have a coercion operation $\mathbf{c}$ on $\text{Rep}_{B, \beta}(A, f)$.*

Proof. We define coe^r as a map $((\text{Rep}_{B, \beta}(A, f))_{\mathbf{r}}, \text{out}) \rightarrow (\prod_{\mathbb{I}} \text{Rep}_{B, \beta}(A, f), \text{out} \circ \text{ev}_r)$ in $\text{Ty}(\Gamma)/B_{\mathbf{r}}$ by universal property, i.e., recursion: set $\text{coe}_{I, \gamma}^r(\text{in } a) := \text{in } (\mathbf{a}_{I, \gamma}^c (r_I(\gamma)) \ a)$ and

$$\text{coe}_{I, \gamma}^r(\text{ext } t_0 \ \phi \ (t, u)) \quad := \quad \text{ext } (\text{coe}_{I, \gamma}^r(t_0)) \ (\phi \mathbf{p}_\mathbb{I}) \ (\text{coe}_{I \upharpoonright \phi, \gamma}^r(t), \text{coe}_{\in V}^{r_I(\gamma)}(\beta'.1) \ u)$$

where $\beta' := \beta(I \times \mathbb{I} \upharpoonright \phi \mathbf{p}_\mathbb{I}, (\gamma \mathbf{p}_\mathbb{I}, q_\mathbb{I})) (\text{out } \text{coe}_{I \upharpoonright \phi, \gamma}^r(t)) (\text{out } \text{coe}_{I, \gamma}^r(t_0))$. By uniqueness, this definition satisfies $\text{ev}_r \circ \text{coe}^r = \text{id}$ and is stable under substitution. ◀

► **Theorem 72 (Fibrancy of replacement).** *For $B \in \text{Ty}^f(\Gamma)$ and $\beta \in \text{Ter}(\Gamma; \text{LocSmall}_V B)$, replacement extends to a substitution-stable $\text{Rep}_{B, \beta} : \text{Ty}_V^f(\Gamma)/B \rightarrow \text{Ty}_V^f(\Gamma)/B$.* ◀

Finally, we check that $\text{Rep}_{B, \beta}(A, f)$ defines an image factorization of f .

► **Theorem 73 (out is an embedding).** *For $(B, \mathbf{b}) \in \text{Ty}^f(\Gamma)$, $\beta \in \text{Ter}(\Gamma; \text{LocSmall}_V B)$, and $(A, f) \in \text{Ty}_V^f(\Gamma)/B$, $\text{out} : \text{Rep}_{B, \beta}(A, f) \rightarrow B$ is an embedding.*

Proof. A map $g : C \rightarrow D$ is an embedding if and only if it is (-1) -truncated, i.e., its fibers are propositions [64, Lemma 7.6.2]. It is equivalent to show that given $c : C$, the fiber of g at $g \ c$ is contractible. We show in the semantics that in every such fiber of out (in or rather its equivalent small fibers), every partial element extends to a total element [23, §5.1].

Given $t_0 \in (\text{Rep}_{B, \beta}(A, f))(I, \gamma)$ and $w \in (\text{Fib}_V \beta \ \text{out})(I \upharpoonright \phi, (\gamma, \text{out}_{I, \gamma}(t_0)))$, the element $(\text{ext } t_0 \ \phi \ w, \mathbf{b}_{I, \gamma}^h \ 1 \ (\text{out}_{I, \gamma}(t_0)) \ \phi \ (\text{pathOf } w))$ extends $(\text{ptOf } w, \text{pathOf } w)$ to a total element. As the type of small fibers is fibrant by Theorem 72, we can correct this extension along the path from $(\text{ptOf } w, \text{pathOf } w)$ to w to obtain an extension of w . ◀

► **Theorem 74 (in is surjective).** *For $(B, \mathbf{b}) \in \text{Ty}^f(\Gamma)$, $\beta \in \text{Ter}(\Gamma; \text{LocSmall}_V B)$, and $(A, f) \in \text{Ty}_V^f(\Gamma)/B$, $\text{in} : \text{Rep}_{B, \beta}(A, f) \rightarrow B$ is a surjection.*

Proof. By Theorem 73, it suffices to show that each fiber of f over an element in the image of out is merely inhabited. We construct $s : (\text{Rep}_{B, \beta}(A, f), \text{out}) \rightarrow (\Sigma B \parallel \text{Fiber}_f \parallel, (.1))$ in $\text{Ty}^f(\Gamma)/B$ by recursion. Define $s_{I, \gamma}(\text{in } a) := (f_{I, \gamma}(a), \text{in } (a, \text{refl}_{f_{I, \gamma}(a)}))$. On $\text{ext } t_0 \ \phi \ w$, we take the first component to be $s_{I, \gamma}(\text{ext } t_0 \ \phi \ w).1 := (\mathbf{b}_{I, \gamma}^h \ 1 \ (\text{out } t_0) \ \phi \ (\text{pathOf } w)) \underline{0}$, as required by the definition of out on ext . For the second, note that we have a path from $s_{I, \gamma}(\text{ext } t_0 \ \phi \ w).1$ to $\text{out } t_0$, namely $\mathbf{b}_{I, \gamma}^h \ 1 \ (\text{out } t_0) \ \phi \ (\text{pathOf } w)$. Coercing $s_{I, \gamma}(t_0).2$ backward along this path gives us a mere inhabitant of the fiber at $s_{I, \gamma}(\text{ext } t_0 \ \phi \ w).1$. By Lemma 61, we can correct it to get a new element with the required boundary on ϕ . ◀

Combining Construction 66, and Theorems 72–74, we have the fibrant image factorization specified by replacement. The desired $\mathcal{V}$ -smallness of the image type is achieved in Construction 66, using the (local) $\mathcal{V}$ -smallness of the inputs.

3.5 Univalent completion

Finally, the interpretation of univalent completion closely follows the pattern of replacement. In a cubical type theory, given A type and $a : A \vdash P(a)$ type, their univalent completion $\mathbf{U}_{A,P}$ type and $t : \mathbf{U}_{A,P} \vdash \mathbf{El}_{A,P}(t)$ type would be populated as follows:

$$\frac{a : A}{\text{in } a : \mathbf{U}_{A,P}} \quad \frac{t_0 : \mathbf{U}_{A,P} \quad \phi \text{ cof} \quad \phi \vdash w : \sum_{t : \mathbf{U}_{A,P}} \mathbf{El}_{A,P}(t) \simeq \mathbf{El}_{A,P}(t_0)}{\text{ext } t_0 [\phi \mapsto w] : \mathbf{U}_{A,P} [\phi \mapsto w.1]}$$

$$\mathbf{El}_{A,P}(\text{in } a) \equiv P(a) \text{ type} \quad \mathbf{El}_{A,P}(\text{ext } t_0 [\phi \mapsto w]) \equiv \mathbf{Glue} [\phi \mapsto \dots] (\dots) \text{ type}$$

The reduction for $\mathbf{El}_{A,P}(\text{ext } t_0 [\phi \mapsto w])$ uses so-called **Glue** types [23, §6] [6, §2.11]. To interpret univalent completion, we need the collection of all types to itself be “fibrant and univalent”. While we cannot speak of identities between large types in type theory, we can speak of *paths* between cubical types: a path between types is an element of $\mathbf{Ty}^f(\Gamma \times \mathbf{y}\mathbb{I})$. The fibrancy and univalence of $\mathbf{Ty}^f(\Gamma)$ with respect to this notion of path is captured by its closure under the **Glue** type former introduced by Cohen, Coquand, Huber, and Mörtberg [23]. There are various interpretations of **Glue** types in the literature using different conditions on $\mathbb{I}$ and Φ , so for generality’s sake we will simply assume they are given to us. We refer to Angiuli et al. [6, Theorem 2] for sufficient conditions to fulfill this hypothesis.

► **Assumption 75** (for § 3.5). Given $A_0 \in \mathbf{Ty}^f(\Gamma)$, $A \in \mathbf{Ty}^f(\Gamma.\phi)$, and $e \in \text{Ter}(\Gamma.\phi; A \simeq A_0 \mathbf{p}_\phi)$, we have $\mathbf{Glue} A_0 \phi (A, e) \in \mathbf{Ty}^f(\Gamma)$ and $\mathbf{unglue} \in \text{Ter}(\Gamma; \mathbf{Glue} A_0 \phi (A, e) \simeq A_0)$ such that $(\mathbf{Glue} A_0 \phi (A, e))\mathbf{p}_\phi = A$ and $(\mathbf{unglue})\mathbf{p}_\phi = e$, stably under substitution.

► **Definition 76.** For $\Gamma \in \widehat{\mathcal{C}}$, write $\mathbf{Fam}^f(\Gamma)$ for the category whose objects are pairs (A, P) with $A \in \mathbf{Ty}(\Gamma)$ and $P \in \mathbf{Ty}^f(\Gamma.A)$ and whose morphisms $f : (A, P) \rightarrow (A', P')$ are morphisms $f : A \rightarrow A'$ in $\mathbf{Ty}(\Gamma)$ such that $P = P'(\text{id}_\Gamma.f)$.

► **Definition 77.** For $(A, P) \in \mathbf{Fam}^f(\Gamma)$, $\mathbf{Equiv}_P \in \mathbf{Ty}(\Gamma.A.\mathbf{Ap}_A)$ is the family informally written $\mathbf{Equiv}_P(a, a') := (P(a') \simeq P(a))$. Set $\mathbf{Sing}_P := \Sigma (\mathbf{Ap}_A) \mathbf{Equiv}_P \in \mathbf{Ty}(\Gamma.A)$.

► **Notation 78.** Given $A \in \mathbf{Ty}(\Gamma)$, write $A^+ \in \mathbf{Ty}(\Gamma)$ for the type of partial elements $A^+(I, \gamma) := \{(\phi, a) \mid \phi \in \Phi(I), a \in A(I \upharpoonright \phi, \gamma)\}$.

► **Notation 79.** Given $(A, P) \in \mathbf{Fam}^f(\Gamma)$, we write $\mathbf{Glue}_P \in \mathbf{Ty}(\Gamma.A.\mathbf{Sing}_P^+)$ for the family given informally by $(\mathbf{Glue}_P)(a, (\phi, w)) := \mathbf{Glue} P(a) \phi (P(w.1), w.2)$. We then write $\mathbf{unglue}_P \in \text{Ter}(\Gamma.A.\mathbf{Sing}_P^+; \mathbf{Glue}_P \simeq P\mathbf{p}_{\mathbf{Sing}_P^+})$ for the associated instance of **unglue**.

► **Construction 80** (Univalent completion). Let $\Gamma \in \widehat{\mathcal{C}}$. For $(A, P) \in \mathbf{Fam}^f(\Gamma)$, we have $(\mathbf{U}_P, \mathbf{El}_P) \in \mathbf{Fam}^f(\Gamma)$ generated by the rules and equations

$$\frac{\gamma \in \Gamma(I) \quad a \in A(I, \gamma)}{\text{in } a \in \mathbf{U}_P(I, \gamma)} \quad \frac{\gamma \in \Gamma(I) \quad t_0 \in \mathbf{U}_P(I, \gamma) \quad z \in \mathbf{Sing}_{\mathbf{El}_P}^+(I, (\gamma, t_0))}{\text{ext } t_0 z \in \mathbf{U}_P(I, \gamma)}$$

$$(\text{ext } t_0 (\phi, w)) \upharpoonright \phi = w.1 \quad (\text{in } a)\alpha = \text{in } (a\alpha) \quad (\text{ext } t_0 z)\alpha = \text{ext } (t_0\alpha) (z\alpha)$$

$$\begin{aligned} \mathbf{El}_P(I, (\gamma, \text{in } a)) &= P(I, (\gamma, a)) \\ \mathbf{El}_P(I, (\gamma, \text{ext } t_0 z)) &= (\mathbf{Glue}_{\mathbf{El}_P})(I, (\gamma, t_0, z)) \end{aligned}$$

in the sense that any $F \in \mathbf{Fam}^f(\Gamma)$ interpreting the signature admits a unique interpretation-preserving $(\mathbf{U}_P, \mathbf{El}_P) \rightarrow F$. The construction is stable under substitution.

Proof. Given $(A, P) \in \text{Fam}^f(\Gamma)$, take $\partial(A, P)(I, \gamma)$ for $(I, \gamma) \in \int_{\mathbb{C}} \Gamma$ to be the set of pairs (a_0, z) where $a_0 \in A(I, \gamma)$ and $z = (\phi, a) \in \text{Sing}_P^+(I, (\gamma, a_0))$ with $\phi \notin \tilde{\Phi}(I)$. Set $T_0(A, P)(I, \gamma) := A(I, \gamma) \sqcup \partial(A, P)(I, \gamma)$. The sets $T_0(A, P)(I, \gamma)$ define a presheaf $T_0(A, P) \in \text{Ty}(\Gamma)$ with action by $\alpha: J \rightarrow I$ given by $\iota_0(a)\alpha := \iota_0(a\alpha)$, $\iota_1(a, (\phi, w))\alpha := \iota_0(w_\alpha.1)$ if $\phi\alpha \in \tilde{\Phi}(J)$, and $\iota_1(a, (\phi, w))\alpha := \iota_1(a_0\alpha, (\phi\alpha, w\alpha))$ if $\phi\alpha \notin \tilde{\Phi}(J)$.

Define $T_1(A, P) \in \text{Ty}(\Gamma.T_0(A, P))$ by cases, setting $T_1(A, P)(I, (\gamma, \iota_0(a))) := P(I, (\gamma, a))$ and $T_1(A, P)(I, (\gamma, \iota_1(a_0, z))) := \text{Glue}_P(I, (\gamma, a_0, z))$. The type $T_1(A, P)$ extends to a fibrant type: given filling inputs, we compute a filler either in P or in Glue_P depending on which case is applicable, and the boundary equation for Glue types (in Assumption 75) ensures that the result is coherent with respect to reindexing.

Set $T(A, P) := (T_0(A, P), T_1(A, P)) \in \text{Fam}^f(\Gamma)$; then T with the inclusions ι_0 defines a pointed endofunctor $\mathbb{T} = (T, \tau)$ on $\text{Fam}^f(\Gamma)$. Write $\mathcal{M} \hookrightarrow \text{Fam}^f(\Gamma)$ for the wide subcategory of $m: (A, P) \rightarrow (A', P')$ for which $m: A \rightarrow A'$ is a levelwise decidable mono in $\text{Ty}(\Gamma)$. To compute the colimit of an ω -chain $(A_n, P_n): \omega \rightarrow \text{Fam}^f(\Gamma)$ in $\mathcal{M}$, we take the first component A_ω to be the colimit of the A_n 's, computed using Corollary 52, then define the values and filling operation of $P_\omega \in \text{Ty}^f(\text{id}_\Gamma.A_\omega)$ by case analysis so that substituting along $\iota_n: A_n \rightarrow A_\omega$ yields $P_\omega(\text{id}_\Gamma.\iota_n) = P_n$. We compute the cobase change of a map in $\mathcal{M}$ similarly.

As in Constructions 55 and 66, $\mathbb{T}$ and $\mathcal{M}$ are suitable inputs to our free algebra construction, Construction 53. We take $(\mathbb{U}_P, \text{El}_P)$ to be the free $\mathbb{T}$ -algebra on (A, P) and get stability under substitution from Proposition 54. $\blacktriangleleft$

The proof that $\mathbb{U}_P$ has homogeneous filling is similar to those we have seen in previous sections, but is also essentially an application of the known fact that a universe closed under Glue types has homogeneous filling [23, §7.1] [6, §2.12]: by definition of El_P , the **ext** constructor closes $\mathbb{U}_P$ under Glue types. The proof that El_P is a univalent family is likewise an instance of the derivation of univalence from Glue types [23, §7.2] [6, §2.12].

► **Lemma 81.** *Let $(P, \mathfrak{p}) \in \text{Ty}^f(\Gamma.A)$. For $\gamma \in \Gamma(I)$, $r \in \mathbf{y}\mathbb{I}(I)$, and $a \in A(I \times \mathbb{I}, \gamma \mathfrak{p}_\mathbb{I})$, we have $\approx \text{scale } \mathfrak{p} \ r \ a \in \text{Equiv}_P(I \times \mathbb{I}, (\gamma \mathfrak{p}_\mathbb{I}, a, a \mathfrak{p}_\mathbb{I}))$, stably under substitution.*

Proof. Because P is fibrant, Equiv_P is also fibrant; write $\mathfrak{e}$ for its filling operator. Set $\approx \text{scale } \mathfrak{p} \ r \ a := \mathfrak{e}_{I, (\gamma \mathfrak{p}_\mathbb{I}, a, a \mathfrak{p}_\mathbb{I})}^c \ r \ (\text{id}_P(I, (\gamma, a \mathfrak{p}_\mathbb{I})))$ where id_P is the identity equivalence. $\blacktriangleleft$

► **Theorem 82 (Fibrancy of univalent completion).** *The mapping $(A, P) \mapsto (\mathbb{U}_P, \text{El}_P)$ extends to a substitution-stable mapping $\text{Ty}^f(\Gamma) \times_{\text{Ty}(\Gamma)} \text{Fam}^f(\Gamma) \rightarrow \text{Ty}^f(\Gamma) \times_{\text{Ty}(\Gamma)} \text{Fam}^f(\Gamma)$.*

Proof. Let $(A, \mathfrak{a}) \in \text{Ty}^f(\Gamma)$ and $(P, \mathfrak{p}) \in \text{Ty}^f(\Gamma.A)$; we define homogeneous filling and coercion for $\mathbb{U}_P$. Write $\bar{\mathfrak{p}}$ for the filling operator for El_P .

For the homogeneous filling $\mathfrak{h}_{I, \gamma} \ r \ t_0 \ \phi \ t$, we define t_+ from t_0 and t as in the proof of Lemma 69, then take $\text{ext } (t_0 \mathfrak{p}_\mathbb{I}) \ (\phi \mathfrak{p}_\mathbb{I} \vee \mathfrak{q}_\mathbb{I} \equiv r \mathfrak{p}_\mathbb{I}, \approx \text{scale } \bar{\mathfrak{p}} \ r \ t_+)$ for the filler.

For $r: \Gamma \rightarrow \mathbf{y}\mathbb{I}$, we have $\text{ev}_r: (\prod_{\mathbb{I}} \mathbb{U}_P, \text{El}_P(r.\text{ev}_r)) \rightarrow (\mathbb{U}_P r, \text{El}_P(r.\text{id}_{A_r}))$ in $\text{Fam}^f(\Gamma)$. We build a section coe^r using the universal property from Construction 80, which gives a coercion operator for $\mathbb{U}_P$ (cf. Lemma 71). Set $\text{coe}_{I, \gamma}^r(\text{in } a) := \text{in } (a_{I, \gamma}^r(r_I(\gamma)) \ a)$ and

$$\text{coe}_{I, \gamma}^r(\text{ext } t_0 \ (\phi, (t, e))) \quad := \quad \text{ext } (\text{coe}_{I, \gamma}^r(t_0)) \ (\phi \mathfrak{p}_\mathbb{I}, (\text{coe}_{I \upharpoonright \phi, \gamma}^r(t), \bar{\mathfrak{e}}_{I \upharpoonright \phi, \gamma}^r(r_I(\gamma)) \ e))$$

where $\bar{\mathfrak{e}}$ is the filling operator for $\text{Equiv}_{\text{El}_P}$ and $\gamma' := (\gamma, \text{coe}_{I \upharpoonright \phi, \gamma}^r(t), \text{coe}_{I, \gamma}^r(t_0))$. $\blacktriangleleft$

► **Theorem 83 (Univalence).** *For $A \in \text{Ty}^f(\Gamma)$ and $P \in \text{Ty}^f(\Gamma.A)$, the family El_P is univalent.*

Proof. It suffices to show that $\text{Sing}_{\text{El}_P} \in \text{Ty}(\Gamma.\mathbb{U}_P)$ is contractible (cf. [23, Corollary 11]). We show partial elements extend: given a partial element $(\phi, (t, e)) \in \text{Sing}_{\text{El}_P}^+(I, (\gamma, t_0))$, we have the total element $(\text{ext } t_0 \ (\phi, (t, e)), \text{unglue}_{\text{El}_P}(I, (\gamma, t_0, (\phi, (t, e)))) \in \text{Sing}_{\text{El}_P}(I, (\gamma, t_0))$. $\blacktriangleleft$

► **Theorem 84** (Surjectivity). *For $A \in \text{Ty}^f(\Gamma)$ and $P \in \text{Ty}^f(\Gamma.A)$, $\text{in} : A \rightarrow \mathbb{U}_P$ is a surjection.*

Proof. By Theorem 83, it suffices to construct a section in $\text{Fam}^f(\Gamma)$ of the projection $(\Sigma (\mathbb{U}_P) \parallel \Sigma (A \mathbf{p}_{\mathbb{U}_P}) (\text{Equiv}_{\text{El}_P}(\text{id}_{\Gamma.\mathbb{U}_P}.\text{in}))) \parallel \text{El}_P \circ (.1)) \rightarrow (\mathbb{U}_P, \text{El}_P)$. We can do so using the universal property of $(\mathbb{U}_P, \text{El}_P)$: as in Theorem 74, we send $\text{in } a$ to $(\text{in } a, \text{in } (a, \text{id}_e))$, then handle the case of $\text{ext } t_0 z$ using the propositionality of the truncation. ◀

4 Replacement as an instance of higher induction recursion

Propositional truncation can be presented as a *higher inductive type* [64, §6], i.e., as inductively generated by point and path constructors: $\|A\|$ is generated by $\text{in} : A \rightarrow \|A\|$ and $\text{squash} : \prod_{t,t':\|A\|} t =_{\|A\|} t'$ [64, §6.9]. We saw in § 2 that univalent completion and replacement can be used to construct many higher inductive types, and in § 3 we have justified the two by inductive constructions. Now we situate them among established forms of induction by observing that the two are instances of “higher induction recursion”.

An *inductive recursive definition*, as formulated by Dybjer [28], introduces an inductive type $\mathbb{I}$ simultaneously with a recursive function from $\mathbb{I}$ into some existing type. The power of induction recursion [29] lies in its allowing the inductive constructors to refer to the recursive function. We know of no framework for *higher* inductive recursion in the literature. Still, we can naively extrapolate what a signature and elimination principle for a higher inductive recursive type would be and see that the results indeed capture replacement and completion.

► **Definition 85** (Inductive recursive replacement). *For B type, $\beta : \text{LocSmall}_{\mathcal{U}} B$, $A : \mathcal{U}$, and $f : A \rightarrow B$, we specify replacement by the following higher inductive recursive signature:*

$$\begin{array}{ll} \text{Rep}_{B,\beta}(A, f) : \mathcal{U} & \text{out} : \text{Rep}_{B,\beta}(A, f) \rightarrow B \\ \text{in} : A \rightarrow \text{Rep}_{B,\beta}(A, f) & \text{out}(\text{in } a) := f a \\ \text{quo} : \prod_{t,t'} (\beta(\text{out } t) (\text{out } t')).1 \rightarrow t = t' & \text{ap}_{\text{out}}^{c,c'}(\text{quo } c c' q) := (\beta(\text{out } c) (\text{out } c')).2 q \end{array}$$

The semantic **Rep** and **out** of § 3.4 interpret this signature, using the fact that **out** is an embedding (Theorem 73) for the interpretation of **quo** and its **out** computation rule. The equation for **out** ($\text{in } a$) holds strictly in said interpretation, while the equation for $\text{ap}_{\text{out}}^{c,c'}(\text{quo } c c' q)$ holds up to a path. If we understand Definition 85 as a signature in *cubical* type theory, with **ap** defined as application under an interval variable, then the alternative interpretation mentioned in Remark 67 would interpret both equations as strict equalities.

Naively combining frameworks for higher inductive [9, 25, 17, 39] and inductive recursive definitions, the eliminator for this signature would have the following rule and equations:

$$\frac{t : \text{Rep}_{B,\beta}(A, f) \vdash C(t) \text{ type} \quad x : A \vdash n : C(\text{in } x) \quad z, z' : \text{Rep}_{B,\beta}(A, f), c : C(z), c' : C(z'), r : (\beta(\text{out } z) (\text{out } z')).1 \vdash q : c =_{\text{quo } z z' r}^C c'}{t : \text{Rep}_{B,\beta}(A, f) \vdash \text{elim}_C [\text{in } x \mapsto n \mid \text{quo } (z, c) (z', c') r \mapsto q] t : C(t)}$$

$$\text{elim}_C [\text{in } x \mapsto n \mid \text{quo } (z, c) (z', c') r \mapsto q] (\text{in } a) = n[a/x]$$

$$\text{apd}_{\text{elim}_C}^{t,t'} [\text{in } x \mapsto n \mid \text{quo } (z, c) (z', c') r \mapsto q] (\text{quo } t t' u) = q[t/z, t'/z', \text{elim } \dots t/c, \text{elim } \dots t'/c', u/r]$$

The **quo** case asks for a family of dependent paths [64, §6.2] and its reduction rule uses the action of dependent functions on paths. Because the relation β (i.e., B ’s identity type) is reflexive, this premise can be instantiated if and only if C is a family of propositions. The eliminator thus expresses precisely that $\text{in} : A \rightarrow \text{Rep}_{B,\beta}(A, f)$ is a surjection in the sense of Definition 2, and hence that $\text{out} \circ \text{in}$ is an image factorization of f .

We can similarly cast univalent completion as a higher inductive definition of a type $U_{A,P}$ together with a recursive definition of a family $u : U_{A,P} \vdash \text{El}_{A,P}(u)$ type. Dybjer’s formulation of inductive recursive types does allow for such recursive definitions of large type families. It is less obvious how to integrate path constructors, because we cannot speak of paths between types in HoTT (although we can in cubical type theories). We opt to replace paths with equivalences and ap with $\text{subst}_P^{a,a'} : (a =_A a') \rightarrow P(a) \simeq P(a')$.

► **Definition 86** (Inductive recursive univalent completion). *For A type and $a : A \vdash P(a)$ type, we specify univalent completion by the following higher inductive recursive signature:*

$$\begin{array}{ll} U_{A,P} \text{ type} & u : U_{A,P} \vdash \text{El}_{A,P}(u) \text{ type} \\ \text{in} : A \rightarrow U_{A,P} & \text{El}_{A,P}(\text{in } a) := P(a) \\ \text{ua} : \prod_{u,u'} \text{El}_{A,P}(u) \simeq \text{El}_{A,P}(u') \rightarrow u = u' & \text{subst}_{\text{El}_{A,P}(-)}^{u,u'} (\text{ua } u \ u' \ e) := e \end{array}$$

The existence of type formers $U_{A,P}$ and $\text{El}_{A,P}(-)$ with the constructors and equations-up-to-equivalence/path above is equivalent to existence of enough univalent families. Elimination for $U_{A,P}$, formulated as for replacement above, is then equivalent to surjectivity of in .

Inductive recursive definitions have always been associated with the construction of universes [28]. Definition 86 is only a slight variation on Shulman’s observation [56] that higher induction recursion can define univalent universes: we simply replace closure under type formers such as Σ or Π with a single “type former” $\text{in} : A \rightarrow U_{A,P}$. Sattler and Vezzosi [55, §2.1] mention a similar higher inductive recursive definition.

► **Remark 87.** Replacement holds in univalent type theory with pushouts [51], so the proof-theoretic strength of the types $\text{Rep}_{B,\beta}(A, f)$ and $U_{A,P}$ is much less than that of general induction recursion [29, 50, 13]. We may note that the constructors of $\text{Rep}_{B,\beta}(A, f)$ and $U_{A,P}$ are finitely branching – whereas a constructor such as $\Sigma : \prod_{A:U} (\text{El } A \rightarrow U) \rightarrow U$ is not – and that the convergence of our free algebra constructions at stage ω depends on this.

We remark, however, that Definitions 85 and 86 are *not* instances of what Hancock, McBride, Ghani, Malatesta, and Altenkirch call *small induction recursion* [37], where the size of the target of the recursion is no larger than that of the inductive type. At least in extensional type theory, small induction recursion reduces to indexed induction. Here, however, it is precisely the point that $\text{Rep}_{B,\beta}(A, f)$ can be smaller than B .

5 Conclusion

We have seen that replacement and univalent completion are versatile and natural principles. We have taken a tour through their uses within type theory, and we have made a case for their foundational character by introducing direct interpretations and a comparison with induction recursion. We conclude with a couple of remarks on future work.

We saw that univalent completion yields both set and groupoid quotients (Examples 24 and 32). One suspects that completion is in fact equivalent to the existence of general quotients by higher equivalence relations. First, we need a meaningful definition of “higher equivalence relation”. This is the subject of work in preparation by David Wörn.

On the other hand, we know thanks to Rijke [51] that the power of replacement is bounded above by the power of pushouts and $\mathbb{N}$. It is natural to wonder whether replacement is *strictly* less powerful than pushouts and, if so, whether there are *useful* models that only support replacement. If this is the case, it is a state of affairs exclusive to the higher setting: set quotients suffice to encode set pushouts. It may be that higher categories with “quotients but not colimits” appear naturally, but have been missed because they have no analogue in 1-category theory.

References

- 1 Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium '77*, volume 96 of *Studies in Logic and the Foundations of Mathematics*, pages 55–66. Elsevier, 1978. doi:10.1016/S0049-237X(08)71989-X.
- 2 Peter Aczel and Michael Rathjen. CST book draft, 2010. version from August 19, 2010. URL: <https://michrathjen.github.io/book.pdf>.
- 3 Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. In Maria del Mar González, Paul C. Yang, Nicola Gambino, and Joachim Kock, editors, *Extended Abstracts Fall 2013*, pages 75–76. Springer International Publishing, 2015. doi:10.1007/978-3-319-21284-5_14.
- 4 Álvaro Pelayo, Vladimir Voevodsky, and Michael A. Warren. A univalent formalization of the p-adic numbers. *Mathematical Structures in Computer Science*, 25(5):1147–1171, 2015. doi:10.1017/S0960129514000541.
- 5 Mathieu Anel. The category of π -finite spaces. *Journal of Pure and Applied Algebra*, 229(6):107925, 2025. doi:10.1016/j.jpaa.2025.107925.
- 6 Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. Syntax and models of cartesian cubical type theory. *Math. Structures in Comput. Sci.*, 31(4), 2021. doi:10.1017/S0960129521000347.
- 7 Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. Cartesian cubical computational type theory: Constructive reasoning with paths and equalities. In *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, pages 6:1–6:17, 2018. doi:10.4230/LIPIcs.CSL.2018.6.
- 8 Michael Artin, Alexander Grothendieck, and Jean-Louis Verdier, editors. *Théorie des topos et cohomologie étale des schémas: Séminaire de Géométrie Algébrique du Bois-Marie, 1963/64, SGA 4, Tome 1*, volume 269 of *Lecture Notes in Mathematics*. Springer-Verlag, 1972. doi:10.1007/BFb0081551.
- 9 Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *Journal of Universal Computer Science*, 23(1):63–88, 2017. doi:10.3217/jucs-023-01-0063.
- 10 Marc Bezem, Ulrik Buchholtz, Pierre Cagne, Bjørn Ian Dundas, and Daniel R. Grayson. Symmetry, 2026. Commit: ca37354. URL: <https://unimath.github.io/SymmetryBook/book.pdf>.
- 11 Marc Bezem, Ulrik Buchholtz, Daniel R. Grayson, and Michael Shulman. Construction of the circle in UniMath. *Journal of Pure and Applied Algebra*, 225(10), 2021. doi:10.1016/j.jpaa.2021.106687.
- 12 Rafaël Bocquet. Strict Rezk completions of models of HoTT and homotopy canonicity, 2023. doi:10.48550/arXiv.2311.05849.
- 13 Ulrik Buchholtz. Proof theory of homotopy type theory: what we know so far, 2016. Slides from a talk at FOMUS - Foundations of Mathematics: Univalent foundations and set theory. URL: <https://ulrikbuchholtz.dk/bielefeld-proof-theory.pdf>.
- 14 Ulrik Buchholtz, J. Daniel Christensen, Jarl G. Taxerås Flaten, and Egbert Rijke. Central H-spaces and banded types. *Journal of Pure and Applied Algebra*, 229(6):107963, 2025. doi:10.1016/j.jpaa.2025.107963.
- 15 Ulrik Buchholtz, Floris van Doorn, and Egbert Rijke. Higher groups in homotopy type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '18*, pages 205–214. ACM, 2018. doi:10.1145/3209108.3209150.
- 16 Timothy Champion. Cubical sites as Eilenberg-Zilber categories, 2023. arXiv:2303.06206.
- 17 Evan Cavallo and Robert Harper. Higher inductive types in cubical computational type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL):1:1–1:27, 2019. doi:10.1145/3290314.

- 18 Evan Cavallo, Anders Mörtberg, and Andrew W. Swan. Unifying cubical models of univalent type theory. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13–16, 2020, Barcelona, Spain*, volume 152, pages 14:1–14:17, 2020. doi:10.4230/LIPIcs.CSL.2020.14.
- 19 Evan Cavallo and Christian Sattler. The algebraic small object argument as a saturation, 2025. arXiv:2506.02759.
- 20 Felix Cherubini, Thierry Coquand, and Matthias Hutzler. A foundation for synthetic algebraic geometry. *Mathematical Structures in Computer Science*, 34(9):1008–1053, 2024. doi:10.1017/S0960129524000239.
- 21 J. Daniel Christensen. A characterization of univalent fibrations, June 2015. Slides from a talk at Category Theory 2015 (CT2015). URL: <https://jdc.math.uwo.ca/papers/univalence-ct2015.pdf>.
- 22 J. Daniel Christensen. Non-accessible localizations. *Journal of Topology*, 17(2):e12336, 2024. doi:10.1112/topo.12336.
- 23 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs, TYPES 2015*, pages 5:1–5:34, 2015. doi:10.4230/LIPIcs.TYPES.2015.5.
- 24 Thierry Coquand and Jonas Höfer. Poset type theory, 2024. Presented at the 30th International Conference on Types for Proofs and Programs, TYPES 2024. URL: <https://vipwww.itu.dk/research/types2024/abstracts.pdf>.
- 25 Thierry Coquand, Simon Huber, and Anders Mörtberg. On higher inductive types in cubical type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09–12, 2018*, pages 255–264. ACM, 2018. doi:10.1145/3209108.3209197.
- 26 Tom de Jong. *Domain Theory in Constructive and Predicative Univalent Foundations*. PhD thesis, University of Birmingham, 2023. doi:10.48550/arXiv.2301.12405.
- 27 Tom de Jong and Martín Hötzel Escardó. On small types in univalent foundations. *Logical Methods in Computer Science*, 19(2):8:1–8:33, 2023. doi:10.46298/LMCS-19(2:8)2023.
- 28 Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000. doi:10.2307/2586554.
- 29 Peter Dybjer and Anton Setzer. Induction–recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1):1–47, 2003. doi:10.1016/S0168-0072(02)00096-9.
- 30 Samuel Eilenberg and Saunders MacLane. Relations between homology and homotopy groups of spaces. ii. *Annals of Mathematics*, 51(3):514–533, 1950. doi:10.2307/1969365.
- 31 José Ferreirós. *Labyrinth of Thought: A History of Set Theory and Its Role in Modern Mathematics*. Birkhäuser Basel, 2007.
- 32 Abraham A. Fraenkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86:230–237, 1922. doi:10.1007/BF01457986.
- 33 David Gepner and Joachim Kock. Univalence in locally cartesian closed ∞ -categories. *Forum Mathematicum*, 29(3):617–652, 2017. doi:10.1515/forum-2015-0228.
- 34 Håkon Robbestad Gylterud. From multisets to sets in homotopy type theory. *Journal of Symbolic Logic*, 83(3):1132–1146, 2018. doi:10.1017/jsl.2017.84.
- 35 Håkon Robbestad Gylterud and Elisabeth Stenholm. Univalent material set theory, 2024. arXiv:2312.13024.
- 36 Michael Hallett. *Cantorian set theory and limitation of size*. Number 10 in Oxford Logic Guides. Clarendon Press, 1984.
- 37 Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small induction recursion. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications*, pages 156–172, 2013. doi:10.1007/978-3-642-38946-7_13.
- 38 Akihiro Kanamori. In praise of replacement. *Bulletin of Symbolic Logic*, 18(1):46–90, 2012. doi:10.2178/bsl/1327328439.

- 39 Ambrus Kaposi and András Kovács. Signatures and induction principles for higher inductive-inductive types. *Logical Methods in Computer Science*, 16(1), 2020. doi:10.23638/lmcs-16(1:10)2020.
- 40 Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after voevodsky). *Journal of the European Mathematical Society*, 23(6):2071–2126, 2021. doi:10.4171/JEMS/1050.
- 41 Azriel Lévy. Axiom schemata of strong infinity in axiomatic set theory. *Pacific Journal of Mathematics*, 10(1):223–238, 1960.
- 42 Daniel R. Licata and Eric Finster. Eilenberg-macLane spaces in homotopy type theory. In *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–9. ACM, 2014. doi:10.1145/2603088.2603153.
- 43 Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975. doi:10.1016/S0049-237X(08)71945-1.
- 44 Per Martin-Löf. *Constructive Mathematics and Computer Programming*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. Elsevier, 1982. doi:10.1016/S0049-237X(09)70189-2.
- 45 Richard Montague. Fraenkel’s addition to the axioms of Zermelo. In Yehoshua Bar-Hillel, E. I. J. Poznanski, M. O. Rabin, and Abraham Robinson, editors, *Essays on the foundations of mathematics*, pages 91–114. Magnes Press, 1961.
- 46 John Myhill. Constructive set theory. *Journal of Symbolic Logic*, 40(3):347–382, 1975. doi:10.2307/2272159.
- 47 Ian Orton and Andrew M. Pitts. Axioms for modelling cubical type theory in a topos. *Log. Methods Comput. Sci.*, 14(4), 2018. doi:10.23638/LMCS-14(4:23)2018.
- 48 Nima Rasekh. Univalence in higher category theory, 2021. arXiv:2103.12762.
- 49 Michael Rathjen. Replacement versus collection and related topics in constructive Zermelo–Fraenkel set theory. *Annals of Pure and Applied Logic*, 136:156–174, 2005. doi:10.1016/j.apal.2005.05.010.
- 50 Michael Rathjen. *Proof Theory of Constructive Systems: Inductive Types and Univalence*, volume 13 of *Outstanding Contributions to Logic*, pages 385–419. Springer International Publishing, 2017. doi:10.1007/978-3-319-63334-3_14.
- 51 Egbert Rijke. The join construction, 2017. arXiv:1701.07538.
- 52 Egbert Rijke. *Topics in synthetic homotopy theory*. PhD thesis, Carnegie Mellon University, 2018. arXiv:1906.09435.
- 53 Egbert Rijke. *Introduction to Homotopy Type Theory*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2025. doi:10.1017/9781108933568.
- 54 Egbert Rijke and Bas Spitters. Sets in homotopy type theory. *Mathematical Structures in Computer Science*, 25(5):1172–1202, 2015. doi:10.1017/S0960129514000553.
- 55 Christian Sattler and Andrea Vezzosi. Partial univalence in n -truncated type theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*, pages 807–819, 2020. doi:10.1145/3373718.3394759.
- 56 Michael Shulman. Higher inductive-recursive univalence and type-directed definitions, 2014. Blog post on the Homotopy Type Theory blog. URL: <https://homotopytypetheory.org/2014/06/08/hiru-tdd/>.
- 57 Thoralf Skolem. Einige Bemerkungen zur axiomatischen Begründung der Mengenlehre. In *Matematikerkongressen i Helsingfors den 4-7 Juli 1922, Den femte skandinaviska matematikerkongressen, Redogörelse*, pages 217–232. Akademiska Bokhandeln, 1923.
- 58 Jan M. Smith. Propositional functions and families of types. *Notre Dame Journal of Formal Logic*, 30(3), 1989. doi:10.1305/ndjfl/1093635159.
- 59 Raymond M. Smullyan and Melvin Fitting. *Set Theory and the Continuum Problem*. Dover Books on Mathematics. Dover Publications, 2010.

- 60 Kristina Sojakova. *Higher Inductive Types as Homotopy-Initial Algebras*. PhD thesis, Carnegie Mellon University, 2016. URL: <http://ra.adm.cs.cmu.edu/anon/2016/abstracts/16-125.html>.
- 61 Raffael Stenzel. *On univalence, Rezk completeness and presentable quasi-categories*. PhD thesis, University of Leeds, 2019. URL: <https://etheses.whiterose.ac.uk/id/eprint/24342>.
- 62 Thomas Streicher. Universes in toposes. In Laura Crosilla and Peter M. Schuster, editors, *From sets and types to topology and analysis – Towards practicable foundations for constructive mathematics*, volume 48 of *Oxford Logic Guides*. Oxford University Press, 2005. doi:10.1093/acprof:oso/9780198566519.003.0005.
- 63 Taichi Uemura. Colimits in the ∞ -category of ∞ -topoi and étale morphisms, 2025. arXiv: 2506.10431.
- 64 The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. URL: <https://homotopytypetheory.org/book/>, 2013.
- 65 Benno van den Berg and Ieke Moerdijk. Univalent completion. *Mathematische Annalen*, 371:1337–1350, 2018. doi:10.1007/s00208-017-1614-3.
- 66 Niels van der Weide and Herman Geuvers. The construction of set-truncated higher inductive types. *Electronic Notes in Theoretical Computer Science*, 347:261–280, 2019. doi:10.1016/j.entcs.2019.09.014.
- 67 Jean van Heijenoort. *From Frege to Gödel: A source book in mathematical logic, 1879–1931*. Harvard University Press, 1967.
- 68 Niccolò Veltri and Niels van der Weide. Constructing higher inductive types as groupoid quotients. *Logical Methods in Computer Science*, 17(2):8:1–8:42, 2021. doi:10.23638/LMCS-17(2:8)2021.
- 69 Vladimir Voevodsky. Univalent foundations project, 2010. Modified version of an NSF grant application. URL: http://www.math.ias.edu/vladimir/files/univalent_foundations_project.pdf.
- 70 Vladimir Voevodsky. Resizing Rules – their use and semantic justification, September 2011. Slides from a talk at TYPES 2011. URL: https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2011_Bergen.pdf.
- 71 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. Available at <http://unimath.org>. doi:10.5281/zenodo.10849216.
- 72 John von Neumann. Zur Einführung der transfiniten Zahlen. *Acta litterarum ac scientiarum Regiae Universitatis Hungaricae Francisco-Josephinae, Sectio scientiarum mathematicarum*, 1:199–208, 1923.
- 73 David Wärn. Eilenberg–MacLane spaces and stabilisation in homotopy type theory. *Journal of Homotopy and Related Structures*, 18:357–368, 2023. doi:10.1007/s40062-023-00330-5.
- 74 Ernst Zermelo. Untersuchungen über die Grundlagen der Mengenlehre. I. *Mathematische Annalen*, 65:261–281, 1908. doi:10.1007/BF01449999.

Multi-Clocked Guarded Recursion Beyond ω

Rasmus Ejlers Møgelberg  

IT University of Copenhagen, Denmark

Abstract

Type theories with multi-clocked guarded recursion provide a flexible framework for programming with coinductive types encoding productivity in types. Combining this with solutions to general guarded domain equations one can also construct relatively simple denotational models of programming languages with advanced features. These constructions have previously been explored in the setting of extensional type theory through a presheaf model, which proves correctness of encodings of W-types. That model has been adapted to presheaves of cubical sets (functors into the category of cubical sets), where the model verifies correctness of encodings also of coinductive types whose definitions involve quotient inductive types such as finite powersets or finite distributions. Likewise the cubical model also verifies correctness of coinductive predicates defined using existential quantification and allows the results to be related to the global world of cubical sets.

This paper looks at how to extend the extensional presheaf model of multi-clocked guarded recursion to higher ordinals, so that correctness of encodings of coinductive types can be extended from W-types to those involving finite powersets and finite distributions, as well as coinductive predicates involving existential quantification. This extension will allow results previously proved in Clocked Cubical Type Theory to be interpreted in a model based on set-theory, proving the correctness of these results as understood in their usual set theoretic interpretation.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Type theory; Theory of computation $\rightarrow$ Denotational semantics; Theory of computation $\rightarrow$ Categorical semantics

Keywords and phrases Guarded Recursion, Coinductive Types, Dependent Type Theory, Accessible Functors, Algebraic Theories

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.11

Funding *Rasmus Ejlers Møgelberg*: This work was supported by the Independent Research Fund Denmark grant number 2032-00134B.

Acknowledgements I thank Jiří Adámek for discussions of the material of Section 6.1.

1 Introduction

The complex numbers form a mathematical paradise in which all polynomials have roots. However, they are not just a dream world without connection to the more earthly world of the real numbers. For example, computing with complex numbers sometimes leads to the construction of real solutions to polynomial equations. In the world of programming language semantics the equations of interest are recursive domain equations $X \cong F(X)$. Many of these do not have solutions in the category of sets, but the topos of trees [7] serves as a mathematical paradise in which guarded versions of these equations $X \cong F(\triangleright X)$ all have solutions. One can therefore construct models of advanced programming languages using the topos of trees, but can one then relate these results to the real world?

The answer depends on the type of results, and what is meant by the “real world”. For example, the original paper on the topos of trees [7] constructs a model of a programming language with higher order store in the internal language of the topos of trees ($\mathbf{Set}^{\omega_{\text{op}}}$), and then argues how these results relate to the real world of the category of sets. In some cases, this is not possible, however. For example, a proof of an existential quantification of a predicate in the topos of trees does not always externalise to a proof of an existential in sets. For this reason, Transfinite Iris [22] uses higher indexing than ω to reason about liveness



© Rasmus Ejlers Møgelberg;

licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 11; pp. 11:1–11:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

properties. Likewise, as we shall explain below, passing to the real world does not preserve functors like P_f and D_f used to model non-determinism and probabilistic choice, and this causes problems when relating models of languages with these constructs to the real world.

The problem can be stated type theoretically using *multi-clocked guarded recursion* as in Clocked Type Theory (CloTT) [6] and its presheaf model [18]. In CloTT the delay modality $\triangleright^\kappa$ is indexed by a clock κ , and one can quantify these clocks universally. The model contains a copy of the standard set theoretic model of type theory as well as a copy of the topos of trees, and universal quantification acts as a form of externalisation, relating results proved in the topos of tree to those in the real world of set theory. In this setting, the problem of concluding from existential quantification in the topos of trees to actual existence in sets can be expressed as commutativity of quantification over clocks with existential quantification.

Multi-clocked guarded recursion also offers a way of programming with coinductive types and predicates in type theory, using guarded recursion to express the requirement of productivity in types [5, 8]. Specifically, for functors F commuting with clock quantification, the solution to the guarded recursive domain equation $X(\kappa) \cong F(\triangleright^\kappa X(\kappa))$ externalises to a terminal coalgebra with carrier $\forall \kappa. X(\kappa)$ in the real world of set theory. Unfortunately, standard functors such as finite powerset functor P_f or the finite distribution functor D_f do not commute with with clock quantification in the standard model, and so one cannot use these tools to program with non-deterministic or probabilistic processes.

By varying the notion of “real world”, however, the situation can be improved. Kristensen et al. [17] construct a model of the combination of CloTT and Cubical Type Theory [13] by considering presheaves valued in cubical sets. In this model, existential quantification and functors like P_f and D_f , constructed as quotient inductive types on the category of hsets *do* commute with clock quantification. This means that guarded recursion can be used as a way of programming with coinductive types and predicates in Cubical Type Theory, including those involving P_f and D_f and existential quantification. Clocked Cubical Type Theory has been used to model programming languages with non-determinism [20] and probabilistic choice [23]. These models are constructed by solving guarded recursive domain equations (also with negative occurrences), and using the commutativity results above, the models can be related to the operational semantics of these languages, defined as coinductive non-deterministic and probabilistic processes. Likewise, Clocked Cubical Type Theory has been used to prove existence and non-existence of distributive laws between the coinductive delay monad and other monads [21], and to model programming languages with gradual typing [14]. Some of these results have been formalised in Guarded Cubical Agda [25], an extension of the Cubical Agda proof assistant [26].

The question is now how tied these results are to cubical type theory. For example, the results proved about existence and non-existence of distributive laws between monads as interpreted in the model of Kristensen et al. are results about these monads as seen as monads of hsets in the cubical model. Do the same results hold for these monads on **Set**? Likewise, results about contextual equivalence of programs proved using the cubical model a priori only say something about their semantics as read in cubical sets. How do these relate to the ordinary set theoretic interpretations?

In this paper we return to the set theoretic model of CloTT and look at how varying the indexing ordinal used in the model construction implies commutativity of clock quantification with certain existential quantifications and with functors constructed as quotient inductive types such as P_f and D_f . The contributions are as follows

- We analyse existing applications of Clocked Cubical Type Theory [17, 23, 21] and isolate requirements on the model that will allow for these to be interpreted in the set theoretic

$$\begin{array}{c}
\frac{}{\Gamma, \kappa : \text{clock} \vdash} \quad \frac{\kappa : \text{clock} \in \Gamma}{\Gamma, \alpha : \kappa \vdash} \quad \frac{\Gamma \vdash t : \triangleright (\alpha : \kappa).A \quad \Gamma, \beta : \kappa, \Gamma' \vdash}{\Gamma, \beta : \kappa, \Gamma' \vdash t[\beta] : A[\beta/\alpha]} \\
\\
\frac{\Gamma, \alpha : \kappa \vdash t : A}{\Gamma \vdash \lambda(\alpha : \kappa).t : \triangleright (\alpha : \kappa).A} \quad \frac{\Gamma, \kappa : \text{clock} \vdash t : A}{\Gamma \vdash \Lambda \kappa. t : \forall \kappa. A} \quad \frac{\Gamma \vdash t : \forall \kappa. A \quad \kappa' : \text{clock} \in \Gamma}{\Gamma \vdash t[\kappa'] : A[\kappa'/\kappa]} \\
\\
\frac{\kappa : \text{clock} \in \Gamma \quad \Gamma \vdash A \text{ type}}{\Gamma \vdash \text{fix}^\kappa : (\triangleright^\kappa A \rightarrow A) \rightarrow A} \quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}
\end{array}$$

■ **Figure 1** Selected typing rules for Clocked Type Theory [6, 18].

model.

- We state the conservativity of the model of Clocked Type Theory over the standard set theoretic model of type theory. This is an easy consequence of the model construction of [18], but has not previously been stated explicitly. We show how this allows us to relate results proved in CloTT to the real world of set theory.
- We give semantic conditions for monads on sets to commute with clock quantification in the model of Clocked Type Theory indexed by regular ordinals, and we give a simple condition on algebraic theories to generate such monads. This accounts for the functors P_f and D_f mentioned above, but also monads of higher arity, such as the countable powerset monad.
- We give conditions allowing existential quantification to commute with quantification over clocks in the model

As a result we see that the applications mentioned above can all be expressed in the internal language of a variant of the set theoretic model indexed by a sufficiently large ordinal. We hope that these results can form part of the semantic basis for formalisation of applications as those mentioned above also in proof assistants not based on cubical type theory.

We remark that the metatheory in which the models are constructed is classical set theory. In particular, the proof of Theorem 8 is very classical.

Overview

We start by recalling Clocked Type Theory and the encoding of coinductive types from guarded recursive types in Sections 2 and 3. Section 4 recalls the applications of Clocked Cubical Type Theory and derives the requirements needed to express these in the version of Clocked Type Theory based on the set theoretic model. Section 5 recalls the presheaf model of Clocked Type Theory and describes the conservativity result over the set theoretic model. Section 6 gives conditions on finitary monads to commute with clock quantification as well as the syntactic condition on algebraic theories ensuring this. Section 7 considers commutativity of existential quantification and clock quantification, and Section 8 concludes.

2 Clocked Type Theory

This section gives a brief introduction to Clocked Type Theory. We refer readers to [18] for full details.

Clocked Type Theory is a dependently typed type theory with Π - and Σ -types, extensional identity types, binary and nullary sums and products, as well as a modality $\triangleright$ for delays

and a guarded fixed point operator. Delays are associated with clocks, so that $\triangleright^\kappa A$ can be understood as a type of data of type A delayed by one time step on clock κ . To program with clocks, the type theory has a special form of assumptions $\kappa : \text{clock}$ that can be placed in a context. Note that *clock* is not a type, but rather a pretype, similar to the status of the interval in Cubical Type Theory [13]. Introduction and elimination for $\triangleright$ is by abstracting and applying ticks on clocks, following the Fitch-style rules for programming with modal types [11, 12]. Selected typing rules can be found in Figure 1. The notation $\triangleright^\kappa A$ is used for $\triangleright(\alpha:\kappa).A$ when the tick α does not appear free in A . Note that variables can be introduced from anywhere in a context, also if they are behind ticks. This means that one can define a term $\text{next}^\kappa \stackrel{\text{def}}{=} \lambda x. \lambda(\alpha:\kappa). x$ of type $A \rightarrow \triangleright^\kappa A$. Given $f : \triangleright^\kappa A \rightarrow A$, the fixed point operator computes the unique fixed point of $f \circ \text{next}^\kappa$. The language of ticks also allows one to define a dependent version of an applicative functor action for $\triangleright$:

$$\lambda f. \lambda x. \lambda(\alpha:\kappa). f[\alpha](x[\alpha]) : \triangleright^\kappa(\Pi(x:A). B(x)) \rightarrow \Pi(x:\triangleright^\kappa A). \triangleright(\alpha:\kappa). B(x[\alpha])$$

This is, for example, useful for reasoning about elements of modal type. It can also be used to express an extensionality principle for $\triangleright$:

$$(t =_{\triangleright(\alpha:\kappa).A} u) \cong \triangleright(\alpha:\kappa). (t[\alpha] =_A u[\alpha])$$

For these applications, it is important that the same ticks can be used on both the logical and term side, and this is one benefit of using type theory rather than a language with a clear separation between term language and logic, such as topos logic.

2.1 Axioms

One of the main purposes of quantification over clocks is to allow for a controlled elimination of $\triangleright$. In Clocked Type Theory this is done by an operation of application to a tick constant $\diamond$, with very special typing rules, which can be used to construct a term of type

$$\text{force} : (\forall \kappa. \triangleright^\kappa A) \rightarrow \forall \kappa. A.$$

This, together with the *tick irrelevance axiom*

$$\frac{\Gamma \vdash t : \triangleright^\kappa A}{\text{tirr}^\kappa t : \triangleright(\alpha:\kappa). \triangleright(\alpha':\kappa). t[\alpha] =_A t[\alpha']} \quad (1)$$

allows one to prove the following

$$\text{IsEquiv}(\lambda(x:\forall \kappa. A). \lambda \kappa. \lambda(\alpha:\kappa). x[\alpha]) \quad (2)$$

where $\text{IsEquiv}(-)$ is defined precisely as in homotopy type theory [24]. In the present case of extensional type theory, this axiom just states that the canonical map $(\forall \kappa. A) \rightarrow \forall \kappa. \triangleright^\kappa A$ is an isomorphism.

The axiom of *clock irrelevance* states

$$\text{IsEquiv}(\lambda x : A. \lambda \kappa. x) \quad (3)$$

Again, in our setting, this just states that the canonical map $A \rightarrow \forall \kappa. A$ is an isomorphism when κ is not free in A . In Clocked Cubical Type Theory [17] clock irrelevance is not an axiom, but can be proved for many types. The other axioms mentioned can all be given computational content in Clocked Cubical Type Theory, but it is unclear how to generalise this to type theories not based on Cubical Type Theory. In this paper we are mostly concerned with properties of the model, and so we will assume clock irrelevance and tick irrelevance as axioms, deriving (2) as a consequence.

$$\begin{array}{c}
\frac{\Delta = \{\kappa_1, \dots, \kappa_n\} \quad \Gamma \vdash \kappa_i : \text{clock for } i = 1, \dots, n}{\Gamma \vdash U_\Delta \text{ type}} \quad \frac{\Gamma \vdash t : U_\Delta}{\Gamma \vdash \text{El}_\Delta(t) \text{ type}} \\
\\
\frac{\Gamma \vdash t : U_\Delta \quad \Delta \subseteq \Delta'}{\Gamma \vdash \text{in}_{\Delta, \Delta'}(t) : U_{\Delta'}} \quad \frac{\Gamma, \kappa : \text{clock} \vdash A : U_{\Delta, \kappa}}{\Gamma \vdash \bar{\forall} \kappa. A : U_\Delta} \quad \frac{\Gamma, \alpha : \kappa \vdash A : U_\Delta \quad \kappa \in \Delta}{\Gamma \vdash \bar{\exists}(\alpha : \kappa). A : U_\Delta}
\end{array}$$

■ **Figure 2** Selected typing rules for universes.

2.2 Universes

It is inconsistent with the clock irrelevance axiom to assume a single universe closed under $\triangleright$ [18]. To solve this, CloTT has a family of universes indexed by finite sets of clocks. Figure 2 shows some basic rules for these. Judgemental equality rules state that codes for type operations indeed encode those type operation, as well as commutativity rules for universe inclusions $\text{in}_{\Delta, \Delta'}$ and codes for type operations. For example

$$\text{El}_{\Delta'}(\text{in}_{\Delta, \Delta'}(t)) = \text{El}_\Delta(t) \quad \text{El}_\Delta(\bar{\exists}(\alpha : \kappa). A) = \triangleright(\alpha : \kappa).(\text{El}_\Delta(A))$$

Note that a universe U_Δ is only closed under $\triangleright^\kappa$ for the κ in Δ . The notation Δ, κ is shorthand for the union of Δ and $\{\kappa\}$.

Unlike previous presentations of CloTT, we include a family of universes of propositions Prop_Δ , also indexed by clock contexts Δ . All rules of Figure 2 have corresponding rules for universes of propositions. Moreover, these universes are closed under basic operations of predicate logic such as

$$\frac{\Gamma \vdash t : U_\Delta \quad \Gamma \vdash \phi : \text{El}_\Delta(t) \rightarrow \text{Prop}_\Delta}{\Gamma \vdash \exists(x : \text{El}_\Delta(t)). \phi(x) : \text{Prop}_\Delta} \quad \frac{\Gamma \vdash t : U_\Delta \quad \Gamma \vdash u, s : \text{El}_\Delta(t)}{\Gamma \vdash u = s : \text{Prop}_\Delta}$$

and a similar rule for universal quantification.

3 Encoding Coinductive Types

One benefit of universes is that they allow for encoding guarded recursive types simply as fixed points for endomaps on universes. Suppose, for example, that $A : U_{\Delta, \kappa}$. Defining $\bar{D}^\kappa(A)$ as the fixed point of the map

$$\lambda(x : \triangleright^\kappa U_{\Delta, \kappa}). A + \bar{\exists}^\kappa x$$

then $D^\kappa(A) \stackrel{\text{def}}{=} \text{El}_{\Delta, \kappa}(\bar{D}^\kappa(A))$ satisfies $D^\kappa(A) = A + \triangleright^\kappa D^\kappa(A)$. Note that this is an actual judgemental equality of types. In Clocked Cubical Type Theory, fixed points only unfold up to propositional equality, and the above type equation is only an equivalence of types. The type constructor D^κ is often referred to as the *guarded delay monad*.

Guarded recursive types can be used as a stepping stone towards defining coinductive types. For example, the more standard coinductive delay monad D can be encoded as the map that maps $A : U_\Delta$ to $\bar{\forall} \kappa. \bar{D}^\kappa(A)$ where κ is a fresh clock. This gives the coinductive solution to $D(A) \cong A + D(A)$. Using guarded recursion has the benefit that one can program with D using the guarded fixed point operator, which allows productivity of coinductive definitions to be captured in types.

To describe the general setting of the encoding of inductive types, note first that for any type I , the type $I \rightarrow U_\Delta$ describes the objects of a category in the naive sense: Morphisms from X to Y are simply terms of the type $\Pi(i : I).X(i) \rightarrow Y(i)$. Here we have left the “elements of” operation El_Δ implicit as we shall do below also for operations of universe inclusions to avoid clutter. If $\Delta \subset \Delta'$, the universe inclusion $\text{in}_{\Delta, \Delta'}$ induces a functor from $I \rightarrow U_\Delta$ to $I \rightarrow U_{\Delta'}$. In the opposite direction, universal quantification over clocks defines a functor $(\forall \kappa.(I \rightarrow U_{\Delta, \kappa})) \rightarrow (I \rightarrow U_\Delta)$ (assuming κ fresh for I) by

$$(\forall \kappa.X)(i) \stackrel{\text{def}}{=} \bar{\forall} \kappa.X[\kappa](i)$$

where $\forall \kappa.(I \rightarrow U_{\Delta, \kappa})$ is considered a category in the naive sense. In the next definition we write $\text{Func}(\mathbb{C}, \mathbb{D})$ for the type of functors between categories $\mathbb{C}$ and $\mathbb{D}$.

► **Definition 1.** A functor $F : (I \rightarrow U_\Delta) \rightarrow (J \rightarrow U_\Delta)$ commutes with clock quantification if there exists a functor $F' : \forall \kappa.\text{Func}(I \rightarrow U_{\Delta, \kappa}, J \rightarrow U_{\Delta, \kappa})$ extending F in the sense that

$$\begin{array}{ccc} (I \rightarrow U_\Delta) & \xrightarrow{F} & (J \rightarrow U_\Delta) \\ \downarrow & & \downarrow \\ (I \rightarrow U_{\Delta, \kappa}) & \xrightarrow{F'} & (J \rightarrow U_{\Delta, \kappa}) \end{array}$$

such that the canonical morphism

$$F(\forall \kappa.X) \rightarrow \forall \kappa.F'[\kappa](X)$$

is an isomorphism in $J \rightarrow U_\Delta$ for all X in $\forall \kappa.(I \rightarrow U_{\Delta, \kappa})$.

Suppose, for example, that $A : U_\Delta$. Then $F(X) \stackrel{\text{def}}{=} A + X$ defines a functor $U_\Delta \rightarrow U_\Delta$ (omitting $I = 1$), which extends to a functor $F' : \forall \kappa.(U_{\Delta, \kappa} \rightarrow U_{\Delta, \kappa})$ because the universe inclusions commute with binary sums. To prove that F commutes with clock quantification, one needs universal quantification to distribute over sums ($\forall \kappa.(B + C) \cong \forall \kappa.B + \forall \kappa.C$) and that A is clock irrelevant which is an axiom (3). In the following we shall use the same symbol for the functor F and its extension F' .

► **Theorem 2** ([5, 19]). Let $F : (I \rightarrow U_\Delta) \rightarrow (I \rightarrow U_\Delta)$ be a functor commuting with universal quantification over clocks. There is a unique object $\mu^\kappa X.F(\triangleright^\kappa X)$ in $(I \rightarrow U_{\Delta, \kappa})$ satisfying $F(\triangleright^\kappa(\mu^\kappa X.F(\triangleright^\kappa X))) \cong \mu^\kappa X.F(\triangleright^\kappa X)$. The object $\forall \kappa.\mu^\kappa X.F(\triangleright^\kappa X)$ is a final coalgebra for F .

The encoding was first proved with respect to a denotational model by Atkey and McBride [5], and later internalised in type theory by the author [19]. Although the setting above is slightly different from the original one, due to the indexing of universes, the proof carries over verbatim from [19]. To get a feeling for the proof, note how the structure isomorphism of the final coalgebra can be easily constructed:

$$\begin{aligned} \forall \kappa.\mu^\kappa X.F(\triangleright^\kappa X) &\cong \forall \kappa.F(\triangleright^\kappa(\mu^\kappa X.F(\triangleright^\kappa X))) \\ &\cong F(\forall \kappa.\triangleright^\kappa(\mu^\kappa X.F(\triangleright^\kappa X))) \\ &\cong F(\forall \kappa.\mu^\kappa X.F(\triangleright^\kappa X)) \end{aligned}$$

A version of Theorem 2 can be proved also for the universes of propositions Prop_Δ . This allows for encoding of inductive predicates.

For Theorem 2 to be useful, one needs a large collection of functors commuting with clock quantification. Many functors have this property just by virtue of being defined by type constructors that commute with clock quantification. For example, $\forall\kappa$ distributes over Σ -types just like Π does, and commutes with Π , so polynomial functors commute with clock quantification:

$$\begin{aligned}\forall\kappa.(\Sigma(a : A).\Pi(b : B(a).X)) &\cong \Sigma(a : \forall\kappa.A).\forall\kappa.\Pi(b : B(a[\kappa]).X) \\ &\cong \Sigma(a : A).\forall\kappa.\Pi(b : B(a).X) \\ &\cong \Sigma(a : A).\Pi(b : B(a).\forall\kappa.X)\end{aligned}$$

using clock irrelevance of A in the second line. In the model, universal quantification over clocks distributes over binary sums of types, but this is not directly provable in the type theory unless one encodes sum as a Σ -type indexed by booleans and uses clock-irrelevance for booleans. In this paper we will just consider this an axiom, and so it proves correctness of the encoding of the coinductive delay monad. The model also verifies that $\forall\kappa$ commutes with $\triangleright^{\kappa'}$, and this can be proved internally if the rule for tick application is extended as in [17]. In this paper we will look at conditions for $\forall\kappa$ commuting with quotient inductive types, in particular monads induced by algebraic theories.

On the propositional side, one can prove that $\forall\kappa$ commutes with conjunction, as well as universal quantification over clocks as well as ordinary variables. Similarly to the case of types, the model also verifies that $\forall\kappa$ commutes with disjunction and $\triangleright^{\kappa'}$. Existential quantification is more difficult. In Clocked Cubical Type Theory, existential quantification can be encoded using Σ and propositional truncation, and both of these can be proved to commute with clock quantification. In the extensional model clock quantification does not commute with propositional truncation (see Example 7 below). We will see, however, that by varying the indexing ordinal of the model, it is safe to assume commutativity with quantification over sets of small cardinality. E.g., by indexing over ω_1 (the first uncountable ordinal), clock quantification commutes with existential quantification over countable set. We first look at what is needed for applications.

4 Applications

This section recalls the applications of Clocked Cubical Type Theory to coalgebra [17], semantics of programming languages with probabilistic choice [23] and to reasoning about weak bisimilarity [21]. The aim is to determine the precise requirements needed to interpret these in a model of Clocked Type Theory.

4.1 Coalgebra

Coalgebras in general describe systems, and final coalgebras behaviours of such systems. Often the systems of interest are non-deterministic or probabilistic, and can be described as coalgebras for functors involving powersets or distributions. For example, coalgebras for the functor $F(X) = P_f(A \times X)$, where P_f is the finite powerset functor, are finitely branching labelled transition systems with labels in A . Similarly, coalgebras for $F(X) = A \times D_f(X)$, where D_f is the finite distribution functor, are Markov-processes with labels in A . Since clock quantification commutes with products, Theorem 2 will give encodings of final coalgebras for these functors, if P_f and D_f commute with quantification over clocks. Note that P_f and D_f are both generated by algebraic theories. For example, $P_f(X)$ is the free join semilattice on X , so generated by $\emptyset$, finite union, and equations for associativity, commutativity, idempotency

and a unit axiom for $\emptyset$. (See Section 6.2 for details.) The natural way to program with these in type theory is therefore as quotient inductive types, using the associated recursion and induction principles. As an example, we recall the induction principle for $P_f(X)$ just in the case of predicates: Given $\phi : P_f(X) \rightarrow \text{Prop}_\Delta$, in order to prove $\forall A : P_f(X). \phi(A)$, it suffices to prove $\phi(\emptyset)$, $\phi(\{x\})$ for all $x : X$, and that $\phi(A)$ and $\phi(B)$ implies $\phi(A \cup B)$.

Given a coalgebra $\xi : X \rightarrow F(X)$ one would also like to define notions of similarity and bisimilarity as coinductive predicates. For example, in the case of $F(X) = P_f(A \times X)$, R is a simulation relation if whenever $R(x, x')$ holds, also the following holds

$$\forall a, y. (a, y) \in \xi(x) \rightarrow \exists y'. (a, y') \in \xi(x') \wedge R(y, y')$$

and simulation is the largest simulation relation. Here, the predicate $(a, y) \in \xi(x)$ is defined by P_f -recursion. Supposing, for example, that $A, X : \mathbf{U}_\emptyset$, simulation should be the final coalgebra for the functor mapping $R : X \times X \rightarrow \text{Prop}_\emptyset$ to

$$G(R)(x, x') \stackrel{\text{def}}{=} \forall a, y. (a, y) \in \xi(x) \rightarrow \exists y'. (a, y') \in \xi(x') \wedge R(y, y')$$

In order to define simulation, one first defines $\lesssim^\kappa$ by guarded recursion to satisfy

$$x \lesssim^\kappa x' \stackrel{\text{def}}{=} \forall a, y. (a, y) \in \xi(x) \rightarrow \exists y'. (a, y') \in \xi(x') \wedge \triangleright^\kappa (y \lesssim^\kappa y')$$

then define $x \lesssim x'$ as $\forall \kappa. x \lesssim^\kappa x'$. This definition will work if G commutes with clock quantification. The crucial step in showing that, is to show that existential quantification and universal quantification over clocks commute. As we shall see, this is not a property that holds generally in the model. However, in this particular case, the natural thing to do is to redefine the predicate $\exists y'. (a, y') \in \xi(x') \wedge R(y, y')$ by P_f -recursion over $\xi(x')$:

- If $A = \emptyset$ then $\exists y'. (a, y') \in A \wedge R(y, y')$ is false
- If $A = \{(a', y')\}$ then $\exists y'. (a, y') \in A \wedge R(y, y')$ is defined to be $a = a' \wedge R(y, y')$
- If $A = B \cup C$ then $\exists y'. (a, y') \in A \wedge R(y, y')$ is defined as

$$(\exists y'. (a, y') \in B \wedge R(y, y')) \vee (\exists y'. (a, y') \in C \wedge R(y, y'))$$

Commutativity of the two quantifications can then be proved by P_f -induction if just $\forall \kappa. (\phi \vee \psi)$ is equivalent to $(\forall \kappa. \phi) \vee (\forall \kappa. \psi)$ and $\forall \kappa. \text{false}$ is false. Both of these hold in the presheaf model of Clocked Type Theory independently of the indexing ordinal.

4.2 Modelling Programming Languages with Probabilistic Choice

Stassen et al. [23] construct a model of Probabilistic FPC, a programming language with recursive types and finite probabilistic choice, in Clocked Cubical Type Theory. One of the basic ingredients in this interpretation is a monad modelling the combination of recursion and probabilistic choice. In its guarded form, this monad maps a type $A : \mathbf{U}_\emptyset$ to the guarded recursive type $D^\kappa A$ satisfying $D^\kappa A \cong D_f(A + \triangleright^\kappa(D^\kappa A))$ where D_f is the finite distribution monad. The coinductive version of this monad satisfies $D^\forall A \cong D_f(A + D^\forall A)$. Elements of this monad are distributions of values of type A and computations that can run for one more step. As before, the correctness of the definition $D^\forall A \stackrel{\text{def}}{=} \forall \kappa. D^\kappa A$ requires D_f commutes with clock quantification.

The monads D^κ and D are used for both denotational and operational semantics. In order to prove an adequacy theorem, the paper constructs a relation between syntax and semantics using guarded recursion. One crucial ingredient in this definition is a notion of lifting of a relation on values to one on computations. In terms of the universes of this paper,

one can express this lifting as mapping a relation $R : A \rightarrow B \rightarrow \text{Prop}_{\Delta, \kappa}$ where $A : \mathcal{U}_{\Delta, \kappa}$ and $B : \mathcal{U}_{\Delta}$ to a relation $\bar{R}^{\kappa} : D^{\kappa}A \rightarrow D^{\vee}B \rightarrow \text{Prop}_{\Delta, \kappa}$. Note the asymmetry: A and B live in different universes, and different monads are applied to them. The definition of $\bar{R}^{\kappa}$ is by case analysis of the first argument: An element in $D^{\kappa}A$ is either a distribution just of values of type A , one just of delayed computations, or a convex combination of the two: $\mu_1 \oplus_p \mu_2$ where $\mu_1 : D_f(A)$ and $\mu_2 : D_f(\triangleright^{\kappa}(D^{\kappa}A))$. (Here and below we leave the inclusions of $D_f(A)$ and $D_f(\triangleright^{\kappa}(D^{\kappa}A))$ into $D^{\kappa}A$ implicit for readability.) Which of the three cases an input is in is decidable and, in the latter case μ_1 and μ_2 and p can be computed from the input. In the latter case the definition reads

$$(\mu_1 \oplus_p \mu_2) \bar{R}^{\kappa} \nu \stackrel{\text{def}}{=} \exists \nu_1 : D_f(B), \nu_2 : D_f(D^{\vee}B). (\nu \rightsquigarrow^{\approx} \nu_1 \oplus_p \nu_2) \wedge \mu_1 \bar{R}^{\kappa} \nu_1 \wedge \mu_2 \bar{R}^{\kappa} \nu_2.$$

where $\rightsquigarrow^{\approx}$ is a form of reduction relation on $D^{\vee}$.

One important result of the paper (Lemma 7.3) states that if $A : \mathcal{U}_{\Delta}$, the relation $\bar{R} : D^{\vee}A \rightarrow D^{\vee}B \rightarrow \text{Prop}_{\Delta}$ defined as $\bar{R}(\mu, \nu) \stackrel{\text{def}}{=} \forall \kappa. \bar{R}^{\kappa}(\mu[\kappa], \nu)$ is equivalent to a coinductive definition of $\bar{R}$ which is similar to the definition of $\bar{R}^{\kappa}$. The proof uses that this coinductive definition can be defined via a guarded recursive definition, and this in turn relies on the existential quantifications of ν_1 and ν_2 to commute with quantification over clocks. Unlike the existential quantifications used in the previous paragraph, these quantifications cannot be rewritten to inductions over a quotient inductive type. In the models presented in this paper, existential quantification will only commute with clock quantification if the existential quantification is of bounded size. The result as stated in this generality will therefore not be true in these models. However, Stassen et al. only apply this theorem to the very specific case of $\nu : D^{\vee}1$, and the quantifications are therefore over $\nu_1 : D_f(1)$ and $\nu_2 : D_f(D^{\vee}1)$, so of bounded size.

4.3 Weak Bisimilarity

The coinductive delay monad satisfying $D(A) \cong A + D(A)$, is an intensional model of recursion because it counts computation steps. Often one wants to reason up to weak bisimilarity. However, working with weak bisimilarity is quite subtle. For example, proving that the quotient of the coinductive delay monad by weak bisimilarity is a monad seems to require countable choice [10]. Rather than working with the quotient, one therefore often works in a category of setoids. For example, Møgelberg and Zwart [21] prove that if T is a free model monad for an algebraic theory with no drop equations, then there is a distributive law of T over the coinductive delay monad quotiented as a setoid.

Recall that coinductive delay monad can be defined as $D(A) \stackrel{\text{def}}{=} \forall \kappa. D^{\kappa}(A)$ where $D^{\kappa}(A) \cong A + \triangleright^{\kappa} D^{\kappa}(A)$. Given a setoid (X, R) , we would like to define the setoid $(D(X), \sim_R)$ capturing weak bisimilarity. Writing

$$\begin{aligned} \text{now} & : A \rightarrow D(A) & \text{step} & : D(A) \rightarrow D(A) \\ \text{now}^{\kappa} & : A \rightarrow D^{\kappa}(A) & \text{step}^{\kappa} & : \triangleright^{\kappa} D^{\kappa}(A) \rightarrow D^{\kappa}(A) \end{aligned}$$

for the left and right inclusions into the sum, the relation $\sim_R$ should be defined by coinduction and cases:

$$\begin{aligned} \text{now}(a) \sim_R y & \stackrel{\text{def}}{=} \exists (n : \mathbb{N}, y' : X). y = \text{step}^n(\text{now}(y')) \wedge R(a, y') \\ x \sim_R \text{now}(a) & \stackrel{\text{def}}{=} \exists (n : \mathbb{N}, x' : X). x = \text{step}^n(\text{now}(x')) \wedge R(x', a) \\ \text{step}(x) \sim_R \text{step}(y) & \stackrel{\text{def}}{=} x \sim_R y \end{aligned}$$

This coinductive predicate can be encoded using guarded recursion, because the function mapping a relation $\sim_R$ to $F(\sim_R)$ defined as above commutes with clock quantification. Indeed, $x \sim_R y$ does not appear in the two first clauses, and so in these cases the required condition follows from the axiom of clock irrelevance. In the last clause the case is immediate.

However, in order to use guarded recursion for programming with weak bisimilarity, Møgelberg and Zwart [21] use a different definition: They define

$$x \sim_R y \stackrel{\text{def}}{=} \forall \kappa. x[\kappa] \sim_R^\kappa y[\kappa]$$

where $\sim_R^\kappa: D^\kappa X \rightarrow D^\kappa X \rightarrow \text{Prop}_{\Delta, \kappa}$ is defined also for $X: \mathbf{U}_{\Delta, \kappa}$ and $R: X \rightarrow X \rightarrow \text{Prop}_{\Delta, \kappa}$ as the unique relation satisfying

$$\text{now}^\kappa(a) \sim_R^\kappa y \stackrel{\text{def}}{=} \exists(n: \mathbb{N}, y': X). y = (\text{step}^\kappa \circ \text{next})^n(\text{now}^\kappa(y')) \wedge R(a, y')$$

$$x \sim_R^\kappa \text{now}^\kappa(a) \stackrel{\text{def}}{=} \exists(n: \mathbb{N}, x': X). x = (\text{step}^\kappa \circ \text{next})^n(\text{now}^\kappa(x')) \wedge R(x', a)$$

$$\text{step}^\kappa(x) \sim_R^\kappa \text{step}^\kappa(y) \stackrel{\text{def}}{=} \triangleright(\alpha: \kappa). (x[\alpha] \sim_R^\kappa y[\alpha])$$

Is this the same as the usual coinductive definition? When comparing the two we can assume $X: \mathbf{U}_\Delta$ and $R: X \rightarrow X \rightarrow \text{Prop}_\Delta$. The critical cases are the first two, so suppose x is $\text{now}(a)$. Then $\forall \kappa. x[\kappa] \sim_R^\kappa y[\kappa]$ reduces to

$$\forall \kappa. \exists(n: \mathbb{N}, y': X). y[\kappa] = (\text{step}^\kappa \circ \text{next})^n(\text{now}^\kappa(y')) \wedge R(a, y')$$

We would like to commute $\forall \kappa$ past the two existential quantification, after which the equivalence follows by the relationship between step and step^κ and the clock irrelevance axiom applied to $R(a, y')$. The first quantification is over a countable set, so will commute with $\forall \kappa$ if the indexing in the model is over a sufficiently high ordinal. The other one is more difficult because we want to define a generally applicable monad, and so should not put restrictions on the cardinality of X . On the other hand, the y' is essentially unique because

$$(\text{step}^\kappa \circ \text{next})^n(\text{now}^\kappa(y')) = (\text{step}^\kappa \circ \text{next})^n(\text{now}^\kappa(y''))$$

implies $\triangleright^n(y' = y'')$. It therefore suffices to show that clock quantification commutes with existential quantifications that are unique in this sense.

4.4 Summary Requirements

Having analysed the applications in detail, we now summarise the requirements needed for universal quantification over clocks in order to express the applications above. They are

- Universal quantification needs to commute with certain quotient-inductive type constructors such as finite powersets and finite distributions
- Universal quantification needs to commute with bounded existential quantification, so over sets that are below a certain cardinality
- Universal quantification needs to commute with existential quantification that is essentially unique in the sense that if $X: \mathbf{U}_\Delta$ and $\phi: X \rightarrow \text{Prop}_{\Delta, \kappa}$ is such that $\phi(x) \wedge \phi(y)$ implies $(\triangleright^\kappa)^n(x = y)$, then $\forall \kappa. \exists x: X. \phi(x)$ implies $\exists(x: X). \forall \kappa. \phi(x)$.

5 The Presheaf Model

The section recalls the model of multi-clocked guarded recursion described by Manna et al. [18]. The model is indexed by a limit ordinal ρ , but Manna et al. only describe the specific case of $\rho = \omega$. All constructions generalise directly, however¹. The model uses a

¹ The abstract description of the left adjoint to $\blacktriangleright$ does not immediately generalise, but the concrete version, which is used throughout that paper, does.

presheaf category over a category $\mathbb{T}_\rho$ of time objects defined as pairs $(\mathcal{E}; \delta)$ where $\mathcal{E}$ is a finite subset of some fixed countably infinite set of clock names and $\delta : \mathcal{E} \rightarrow \rho$. A morphism from $(\mathcal{E}; \delta)$ to $(\mathcal{E}'; \delta')$ is a map $\sigma : \mathcal{E} \rightarrow \mathcal{E}'$ such that $\delta' \circ \sigma \leq \delta$ in the pointwise order.

Let $\mathbf{GR}$ be the category $\mathbf{PSh}(\mathbb{T}_\rho) \stackrel{\text{def}}{=} \mathbf{Set}^{\mathbb{T}_\rho}$ of *covariant* presheaves on $\mathbb{T}_\rho$. We write $\sigma \cdot x$ for $X(\sigma)(x)$ when X is a presheaf, $x \in X(\mathcal{E}; \delta)$ and $\sigma : (\mathcal{E}; \delta) \rightarrow (\mathcal{E}'; \delta')$. There is an object of clocks $\mathbf{Clk}$ in $\mathbf{GR}$ defined as $\mathbf{Clk}(\mathcal{E}; \delta) = \mathcal{E}$. This is used to model assumptions of the form $\kappa : \text{clock}$ in contexts.

The operation $\triangleright$ can be described semantically using an endofunctor $\blacktriangleright$ on the slice category $\mathbf{GR}/\mathbf{Clk}$. To describe this, first recall that any slice category of a presheaf category is equivalent to the category of presheaves over the elements of the slicing object. In this particular case, the category of elements $\int \mathbf{Clk}$ has as objects triples $(\mathcal{E}; \delta; \lambda)$, where $\lambda \in \mathcal{E}$ and morphisms from $(\mathcal{E}; \delta; \lambda)$ to $(\mathcal{E}'; \delta'; \lambda')$ are $\mathbb{T}_\rho$ -morphisms $\sigma : (\mathcal{E}; \delta) \rightarrow (\mathcal{E}'; \delta')$ such that $\sigma(\lambda) = \lambda'$. The modality $\blacktriangleright$ can then be defined as

$$(\blacktriangleright X)(\mathcal{E}; \delta; \lambda) = \lim_{\alpha < \delta(\lambda)} X(\mathcal{E}; \delta[\lambda \mapsto \alpha]; \lambda)$$

Note how this generalises the definition of $\blacktriangleright$ in the case of the topos of trees ($\mathbf{PSh}(\omega^{\text{op}})$): $(\blacktriangleright X)(0) = 1$, $(\blacktriangleright X)(n+1) = X(n)$ to higher ordinals and the multi-clocked setting.

The model constructed by Manna et al. follows the standard construction of a semantic model of type theory from presheaf models: Semantic contexts are objects Γ of $\mathbf{GR}$, and semantic types in semantic context Γ are presheaves over $\int \Gamma$. So a semantic type assigns, to each $(\mathcal{E}; \delta)$ and $\gamma \in \Gamma(\mathcal{E}; \delta)$ a set $X(\mathcal{E}; \delta; \gamma)$, and to each $\sigma : (\mathcal{E}; \delta) \rightarrow (\mathcal{E}'; \delta')$ a morphism $\sigma \cdot (-) : X(\mathcal{E}; \delta; \gamma) \rightarrow X(\mathcal{E}'; \delta'; \sigma \cdot \gamma)$. In order to interpret the clock irrelevance axiom, the model restricts attention to semantic types that are clock irrelevant in the following sense.

► Definition 3 ([9]). *Let Γ be an object of $\mathbf{GR}$. A presheaf X over $\int \Gamma$ is said to be invariant under clock introduction if, for any $(\mathcal{E}; \delta)$, $\lambda \notin \mathcal{E}$ and α , the map*

$$\iota \cdot (-) : X(\mathcal{E}; \delta; \gamma) \rightarrow X((\mathcal{E}, \lambda); \delta[\lambda \mapsto \alpha]; \iota \cdot \gamma)$$

is an isomorphism, where $\mathcal{E}, \lambda$ is shorthand for $\mathcal{E} \cup \{\lambda\}$ and $\iota : \mathcal{E} \rightarrow (\mathcal{E}, \lambda)$ is the inclusion.

For example, if Γ is the terminal object 1 , a presheaf X in $\mathbf{PSh}(\int 1) \cong \mathbf{GR}$ is invariant under clock introduction if and only if each $\sigma \cdot (-) : X(\emptyset; !) \rightarrow X(\mathcal{E}; \delta)$ is an isomorphism, and so X is a constant presheaf. Similarly, the subcategory of presheaves over $\int \mathbf{Clk}$ invariant under clock introduction is equivalent to the topos of trees $\mathbf{Set}^{\omega^{\text{op}}}$. On the other hand, $\mathbf{Clk}$ is an example of a presheaf that is not invariant under clock introduction.

The collection of presheaves invariant under clock introduction is closed under standard type formers such as Σ , Π and universal quantification over clocks [9]. Universal quantification over clocks in fact is just modelled as a Π -type indexed by $\mathbf{Clk}$. We recall [9] an alternative, direct definition of this which we will need in the following sections. Suppose Γ is an object of $\mathbf{GR}$ and A is a (semantic) dependent type over $\Gamma \times \mathbf{Clk}$. This means that for any $(\mathcal{E}; \delta)$, $\gamma \in \Gamma(\mathcal{E}; \delta)$ and $\lambda \in \mathcal{E}$, $A(\mathcal{E}; \delta; \gamma; \lambda)$ is a set. Suppose we have a function that assigns, to each $\mathcal{E}$ an element $\lambda_{\mathcal{E}} \notin \mathcal{E}$, and define $\forall(A)$ as a dependent type over Γ by

$$\forall(A)(\mathcal{E}; \delta; \gamma) \stackrel{\text{def}}{=} \lim_{\alpha < \rho} A((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \alpha]; \iota \cdot \gamma; \lambda_{\mathcal{E}})$$

where $\iota : \mathcal{E} \rightarrow (\mathcal{E}, \lambda_{\mathcal{E}})$ is the inclusion. The axiom (2) can then be proved as follows

$$\begin{aligned} \forall(\blacktriangleright A)(\mathcal{E}; \lambda; \gamma) &\stackrel{\text{def}}{=} \lim_{\alpha < \rho} \lim_{\beta < \alpha} A((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \beta]; \iota \cdot \gamma; \lambda_{\mathcal{E}}) \\ &\cong \lim_{\alpha < \rho} A((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \alpha]; \iota \cdot \gamma; \lambda_{\mathcal{E}}) \\ &= \forall(A)(\mathcal{E}; \lambda; \gamma) \end{aligned}$$

which works only because ρ is assumed to be a limit ordinal. Note also that this definition makes it clear that clock quantification distributes over sums.

► **Terminology.** In the following, given an object Γ of GR, we shall refer to a presheaf X over $\int \Gamma$ which is invariant under clock introduction as a family over Γ . The sets $X(\mathcal{E}; \delta; \gamma)$ are referred to as the fibres of X , and if all fibres of X are subsingletons, we say that X is a predicate on Γ .

5.1 Universes

If Δ is a clock context (so a finite set of clock variables), form the presheaf Clk^{Δ} in GR as the Δ -fold product of Clk by itself. Write $\text{GR}[\Delta]$ for $\text{PSh}(\int \text{Clk}^{\Delta})$. Assuming a set theoretic universe of small sets, there is an object $\mathcal{U}^{\Delta}$ in $\text{GR}[\Delta]$ and a family $\mathcal{E}l^{\Delta}$ over it such that

- $\mathcal{U}^{\Delta}$ is invariant under clock introduction
- For any Γ in $\text{GR}[\Delta]$ and A a family over Γ with small fibres, there is a unique map $\ulcorner A \urcorner : \Gamma \rightarrow \mathcal{U}^{\Delta}$ such that $A = \mathcal{E}l^{\Delta}[\ulcorner A \urcorner]$.
- This universe is closed under Π - and Σ -types, universal quantification over clocks, $\triangleright^{\kappa}$ for $\kappa \in \Delta$.

The construction uses a variant of the Hofmann-Streicher construction of universes in presheaf categories [16]: An element of $\mathcal{U}^{\Delta}(\mathcal{E}; \delta; f : \Delta \rightarrow \mathcal{E})$, is a family over $y(f[\Delta], \delta|_{f[\Delta]})$ in GR with small fibres, where y is the yoneda embedding, and $f[\Delta] \subseteq \mathcal{E}$ is the image of f . Concretely, this means that an element is a family of small sets A_{σ} indexed by the diagram whose components are maps $\sigma : (f[\Delta], \delta|_{f[\Delta]}) \rightarrow (\mathcal{E}', \delta')$, which is invariant under clocks in the sense that $\iota \cdot (-) : A_{\sigma} \rightarrow A_{\iota \circ \sigma}$ is an isomorphism for all ι as in Definition 3. For example, $\mathbf{U}_{\emptyset}$ is an object in $\text{GR}[\emptyset] \cong \text{GR}$, and this is the constant presheaf of small GR objects that are invariant under clock introduction. The universes of Clocked Type Theory can be modelled using these universes by appropriate reindexing. We refer to Manna et al. [18] for further details. For this paper we also need a family of universes of propositions.

► **Proposition 4.** *There exists an object $\mathcal{U}_{\text{Prop}}^{\Delta}$ of $\text{GR}[\Delta]$ and a predicate $\mathcal{E}l_{\text{Prop}}^{\Delta}$ on it such that*

1. $\mathcal{U}_{\text{Prop}}^{\Delta}$ is invariant under clock introduction
2. For any Γ in $\text{GR}[\Delta]$ and A predicate on Γ , there is a unique map $\ulcorner A \urcorner : \Gamma \rightarrow \mathcal{U}_{\text{Prop}}^{\Delta}$ such that $A \cong \mathcal{E}l_{\text{Prop}}^{\Delta}[\ulcorner A \urcorner]$.
3. This universe is closed under $\forall$ - and $\exists$ -quantification over objects invariant under clock introduction, finite conjunction and disjunction, universal quantification over clocks, and $\triangleright^{\kappa}$ for $\kappa \in \Delta$.

Proof. Let $\mathcal{U}_{\text{Prop}}^{\Delta}(\mathcal{E}; \delta; f : \Delta \rightarrow \mathcal{E})$ be the subset of $\mathcal{U}^{\Delta}(\mathcal{E}; \delta; f : \Delta \rightarrow \mathcal{E})$ consisting of families where each A_{σ} is a subset of $1 = \{\star\}$. The proof of almost all the properties above then carry over verbatim from the case of $\mathcal{U}^{\Delta}$, except for closure under existential quantification. For that, suppose Γ is an object of $\text{GR}[\Delta]$, A a family over Γ , $\ulcorner B \urcorner : \Gamma.A \rightarrow \mathcal{U}_{\text{Prop}}^{\Delta}$, and $B = \mathcal{E}l_{\text{Prop}}^{\Delta}[\ulcorner B \urcorner]$. Here

$$\Gamma.A(\mathcal{E}; \delta) = \{(\gamma, a) \mid \gamma \in \Gamma(\mathcal{E}; \delta) \wedge a \in A(\mathcal{E}; \delta; \gamma)\}$$

We must show that the presheaf $\exists(A, B)$ over $\int \Gamma$ defined by

$$\exists(A, B)(\mathcal{E}; \delta; \gamma) \stackrel{\text{def}}{=} \bigcup_{a \in A(\mathcal{E}; \delta; \gamma)} B(\mathcal{E}; \delta; \gamma; a)$$

has a code $\ulcorner \exists(A, B) \urcorner : \Gamma \rightarrow \mathcal{U}_{\text{Prop}}^\Delta$. By item 2 of the proposition, it suffices to show that $\exists(A, B)$ is invariant under clock introduction, and so a predicate. So let $\iota : (\mathcal{E}; \delta) \rightarrow ((\mathcal{E}, \lambda); \delta[\lambda \mapsto \alpha])$ be given by the inclusion of $\mathcal{E}$ into $(\mathcal{E}, \lambda)$. Then

$$\begin{aligned} \exists(A, B)((\mathcal{E}, \lambda); \delta[\lambda \mapsto \alpha]; \iota \cdot \gamma) &\stackrel{\text{def}}{=} \bigcup_{a \in A((\mathcal{E}, \lambda); \delta[\lambda \mapsto \alpha]; \iota \cdot \gamma)} B((\mathcal{E}, \lambda); \delta[\lambda \mapsto \alpha]; \iota \cdot \gamma; a) \\ &= \bigcup_{a \in A(\mathcal{E}; \delta; \gamma)} B((\mathcal{E}, \lambda); \delta[\lambda \mapsto \alpha]; \iota \cdot \gamma; \iota \cdot a) \\ &= \bigcup_{a \in A(\mathcal{E}; \delta; \gamma)} B(\mathcal{E}; \delta; \gamma; a) \\ &= \exists(A, B)(\mathcal{E}; \delta; \gamma) \end{aligned}$$

using invariance under clock introduction for A and B . ◀

5.2 Conservativity Over Set

One important consequence of the requirement that families be invariant under clock introduction is that the fragment of the type theory that does not mention clocks, ticks or guarded recursion is interpreted using constant presheaves only, and so the interpretation of this fragment agrees with that of the standard model in **Set**. In particular, any proof of such a statement in the model based on **GR** gives rise to a proof in the model based on **Set**.

► **Theorem 5.** *Let $\Gamma \vdash A$ type be a wellformed statement in Clocked Type Theory that can also be expressed in the internal type theory of **Set**. If there is a term t such that $\Gamma \vdash t : A$, then also the dependent type constructed by interpreting $\Gamma \vdash A$ type in **Set** can be inhabited.*

Note that Γ and A can be constructed using guarded recursion as long as they can be expressed in the internal language of **Set**. For example, any endofunctor F on **Set** mapping small sets to small sets induces a functor $F : \mathcal{U}_\emptyset \rightarrow \mathcal{U}_\emptyset$ in the internal language of our model. The type $\forall \kappa. \mu X. F(\mathfrak{p}^\kappa X)$ is a closed type, so is interpreted as a set. We can therefore view the statement that $\forall \kappa. \mu X. F(\mathfrak{p}^\kappa X)$ is a final coalgebra for F (Theorem 2) as a statement in the internal language of **Set**, and since this statement has a proof in Clocked Type Theory if F commutes with clock quantification, then, by Theorem 5, it also holds in the set-theoretic interpretation. Of course, this statement will talk about the action of F as an endofunctor on $\mathcal{U}_\emptyset$, which, as mentioned above is the set of all small objects of **GR** that are invariant under clock introduction. However, the notion of functor expressed using this universe in **Set** is equivalent to the one expressed using the usual universe of small sets. Many of the end results proved in the applications mentioned in Section 4, are of this form, for example, existence or non-existence of distributive laws of monads on **Set** or setoids, and contextual equivalence of programs of Probabilistic FPC. Their proofs use guarded recursion, but by Theorem 5 these proofs also prove their usual set theoretic interpretation.

6 Algebraic data types

The construction of lifting a set functor F preserving smallness to an endomap on a universe $\mathcal{U}^\emptyset$ extends to any of the universes $\mathcal{U}^\Delta$. For such a functor to commute with universal quantification over clocks means that the canonical map

$$F(\lim_{\alpha < \rho} A((\mathcal{E}, \lambda_\mathcal{E}); \delta[\lambda_\mathcal{E} \mapsto \alpha]; \iota \cdot \gamma; \lambda_\mathcal{E})) \rightarrow \lim_{\alpha < \rho} F(A((\mathcal{E}, \lambda_\mathcal{E}); \delta[\lambda_\mathcal{E} \mapsto \alpha]; \iota \cdot \gamma; \lambda_\mathcal{E}))$$

must be an isomorphism. Since A can be anything, this means that F needs to commute with ρ^{op} -indexed limits. Theorem 2 is therefore simply a synthetic restatement of Adámek’s fixed point theorem [1]: The type $\forall \kappa. \mu^\kappa X. F(\triangleright^\kappa X)$ is the limit of the terminal sequence

$$1 \leftarrow F(1) \leftarrow F^2(1) \leftarrow \dots \leftarrow F^\omega(1) \leftarrow F^{\omega+1} \leftarrow \dots$$

(where $F^\beta(1) = \lim_{\alpha < \beta} F^\alpha$ for limit ordinals β), and Adámek’s theorem states that this is a terminal coalgebra if F preserves ρ^{op} -indexed limits.

In this section we look at conditions for functors to commute with ordinal-indexed limits. We are particularly interested in the cases of monads generated by algebraic theories, as in the examples of finite powersets and finite distributions. First a few examples.

► **Example 6.** The finite powerset monad P_f does not commute with ω^{op} -limits. In fact the limit of the ω^{op} -long terminal sequence

$$1 \leftarrow P_f(1) \leftarrow P_f^2(1) \leftarrow \dots$$

contains infinite sets [2, 27]. Worrel [27] proved that the terminal sequence converges after $\omega + \omega$ steps, but in general P_f does not commute with $(\omega + \omega)^{\text{op}}$ -indexed limits. For a counter example, just append the terminal sequence to an ω -long sequence of 1. In general, however, P_f does commute with ω_1^{op} -limits, where ω_1 is the first uncountable ordinal [2].

The next example shows that some monads generated by algebraic theories do not commute with any ρ^{op} -limits.

► **Example 7.** Propositional truncation is the monad $T(X) = \{\star \mid \exists x : X.\text{true}\}$. This monad does not commute with ρ^{op} -limits for any ρ . Consider, for example, the sequence

$$X_\alpha = \{\beta \in \rho \mid \beta > \alpha\}$$

with maps $X_\alpha \rightarrow X_\gamma$ for $\gamma \leq \alpha$ given by inclusions. The limit of this sequence is empty, so also $T(\lim X) = \emptyset$, but $T(X_\alpha) = 1$, so $\lim T(X) = 1$.

6.1 A Semantic Condition

We now state conditions that imply that a functor preserves ρ^{op} -indexed limits. Let κ be a regular cardinal. A set Y is κ -small if it has cardinality strictly less than κ , and we write $Y \subseteq_\kappa X$ to mean that Y is a κ -small subset of X . Recall [4] that a functor $F : \mathbf{Set} \rightarrow \mathbf{Set}$ is said to be κ -accessible, if it preserves κ -filtered colimits. In particular, for any X , the map

$$\text{colim}_{X' \subseteq_\kappa X} F(X') \rightarrow F(X)$$

must be an isomorphism (in fact this statement is equivalent to being κ -accessible [3]). In words, this means that any $x \in F(X)$ lives already in $F(X')$ for some κ -small subset X' of X . Say that a functor F preserves pullbacks of monos, if, for any pullback diagram

$$\begin{array}{ccc} A & \longrightarrow & B \\ \downarrow & \lrcorner & \downarrow m \\ C & \longrightarrow & D \end{array}$$

where m is a mono, also F applied to the diagram is a pullback. In particular, this means that F preserves monos, also the non-split ones from the empty set.

Say that a functor F preserves κ -small intersections if, for all families $(X_i \subseteq X)_{i \in I}$ indexed by κ -small sets I , $F(\cap_{i \in I} X_i)$ is the limit of the diagram

$$\begin{array}{ccccc} F(X_i) & & \dots & & F(X_j) & & \dots \\ & \searrow & & & \swarrow & & \\ & F(X) & & & & & \end{array}$$

► **Theorem 8.** *Let $F : \mathbf{Set} \rightarrow \mathbf{Set}$ be a κ -accessible functor which preserves κ -small intersections and pullbacks of monomorphisms. If ρ is a regular ordinal, and there exists an $\alpha \in \rho$ of the same cardinality as κ , then F commutes with ρ^{op} -indexed limits.*

For the proof we need the following.

► **Lemma 9.** *Let ρ be a regular ordinal, and let $X : \rho^{\text{op}} \rightarrow \mathbf{Set}$ be a diagram whose maps $X_\beta \rightarrow X_\alpha$ are all surjective. If, for any $\alpha \in \rho$, there exists an ordinal $\beta > \alpha$ such that $X_\beta \rightarrow X_\alpha$ is not injective, then, for any $\alpha \in \rho$ there is a $\beta(\alpha)$ and an injection $\alpha \rightarrow X_{\beta(\alpha)}$.*

Proof. To construct $\beta(\alpha)$ we construct first a map $\gamma : \rho \rightarrow \rho$ by induction on the input: If α is a successor, so $\alpha = s(\alpha')$, define $\gamma(\alpha)$ to be an ordinal such that $X_{\gamma(\alpha)} \rightarrow X_{\gamma(\alpha')}$ is not injective. If α is a limit ordinal, define

$$\gamma(\alpha) = \bigvee_{\alpha' < \alpha} \gamma(\alpha')$$

In this case $\gamma(\alpha)$ is in ρ because ρ is regular. Using this, define $\beta(\alpha) = \gamma(s(\alpha))$. Next we construct injections $f_{\alpha'} : \alpha' \rightarrow X_{\beta(\alpha)}$ for $\alpha' < s(\alpha)$ by induction on α' such that

- $f_{\alpha'}$ extends $f_{\alpha''}$ whenever $\alpha'' < \alpha'$ and
- the composition $X_{\beta(\alpha), \gamma(\alpha')} \circ f_{\alpha'} : \alpha' \rightarrow X_{\beta(\alpha)} \rightarrow X_{\gamma(\alpha')}$ is injective for all α' .

Again, we proceed by cases of α' : If $\alpha' = s(\alpha'')$, since $X_{\gamma(\alpha')} \rightarrow X_{\gamma(\alpha'')}$ is not injective, and $X_{\beta(\alpha), \gamma(\alpha'')} \circ f_{\alpha''}$ is injective, there must be an element $x \in X_{\gamma(\alpha')}$ outside the image of $X_{\beta(\alpha), \gamma(\alpha'')} \circ f_{\alpha''}$. We can therefore extend $f_{\alpha''}$ to $f_{\alpha'}$ by picking $f_{\alpha'}(\alpha'')$ to be an element in $X_{\beta(\alpha)}$ that maps to x by $X_{\beta(\alpha), \gamma(\alpha')}$. When α' is a limit ordinal, we can simply define $f_{\alpha'}(\alpha'') = f_{s(\alpha'')}(s(\alpha''))$. ◀

Proof of Theorem 8. First note that if $x \in F(X)$, then there exists a smallest subset $X' \subseteq_\kappa X$ such that $x \in F(X')$ (leaving the action of F on the inclusion implicit): There exists an X' because F is κ -accessible, and the smallest is then the intersection of all $X' \setminus \{x'\}$ such that $x \in F(X' \setminus \{x'\})$.

Let X be a ρ^{op} -indexed diagram. We must show that the canonical map

$$F(\lim X) \rightarrow \lim F(X)$$

is an isomorphism. To show surjectivity, suppose $x \in \lim F(X)$. Let X'_α be the smallest subset of X_α such that $x_\alpha \in F(X'_\alpha)$. We show that these form a subdiagram of X . Suppose $\beta \leq \alpha$ and let $r : X_\alpha \rightarrow X_\beta$ be the restriction map. Then clearly $X'_\beta \subseteq r[X'_\alpha]$ by minimality of X'_β . Since F preserves pullbacks of monomorphisms, the diagram

$$\begin{array}{ccc} F(r^{-1}(X'_\beta) \cap X'_\alpha) & \longrightarrow & F(X'_\beta) \\ \downarrow & & \downarrow \\ F(X'_\alpha) & \longrightarrow & F(r[X'_\alpha]) \end{array}$$

is a pullback, so $x_\alpha \in F(r^{-1}(X'_\beta))$, and by minimality then $r^{-1}(X'_\beta) \cap X'_\alpha = X'_\alpha$. So

$$r[X'_\alpha] = r[r^{-1}(X'_\beta) \cap X'_\alpha] \subseteq X'_\beta$$

and so $X'_\beta = r[X'_\alpha]$. So X' is a subdiagram, and each map in X' is surjective.

We next show that the maps $X'_\beta \rightarrow X'_\alpha$ must be bijections from some point on. If not, by Lemma 9, for any $\alpha \in \rho$ there must be an injection of α into some $X'_{\beta(\alpha)}$, but since there is an α of cardinality κ , this contradicts $X'_{\beta(\alpha)}$ being κ -small.

So there must be a stage α such that all maps $X'_\beta \rightarrow X'_\alpha$ are isomorphisms. Then $F(\lim X') \cong F(X'_\alpha) \cong \lim F(X')$, and so x is in the image of the comparison map via the inclusion of $F(\lim X')$ into $F(\lim X)$:

$$\begin{array}{ccc} F(\lim X') & \xrightarrow{\cong} & \lim F(X') \ni x \\ \downarrow & & \downarrow \\ F(\lim X) & \longrightarrow & \lim F(X) \end{array}$$

For injectivity, suppose $a, b \in F(\lim X)$ map to the same element in $\lim F(X)$, so that $F(\pi_\alpha)a = F(\pi_\alpha)b$ for any α . Let $X'_a \subseteq \lim X$ be the smallest subset such that $a \in F(X'_a)$ and similarly for X'_b . For each $x, y \in X'_a$ with $x \neq y$ there is an α such that $x_\alpha \neq y_\alpha$. Let α be the upper bound of these as x, y range over all pairs of different elements in X'_a . Note that this is an element in ρ because ρ is regular, and because X_a has cardinality strictly smaller than some element in ρ . Then $\pi_\beta : X'_a \rightarrow \pi_\beta[X'_a]$ is a bijection whenever $\beta \geq \alpha$. Similarly, there is a γ such that for all $\beta \geq \gamma$, $\pi_\beta : X'_b \rightarrow \pi_\beta[X'_b]$ is a bijection. Let β be the least upper bound of α and γ . Then $\pi_\beta[X'_a]$ is the smallest subset of X_β such that $F(\pi_\beta)(a)$ is in $F(\pi_\beta[X'_a])$, because a smaller subset $X'' \subset \pi_\beta[X'_a]$ would give a subset $\pi_\beta^{-1}(X'') \subset X'_a$ with $a \in F(\pi_\beta^{-1}(X''))$. Similarly, $\pi_\beta[X'_b]$ is the smallest subset such that $F(\pi_\beta)(b)$ is in $F(\pi_\beta[X'_b])$. Since $F(\pi_\beta)(a) = F(\pi_\beta)(b)$, we must have $\pi_\beta[X'_b] = \pi_\beta[X'_a]$. So then, $X'_a \cong \pi_\beta[X'_a] \cong X'_b$ via an isomorphism f such that $f(x)_\beta = x_\beta$ from some β on. We conclude that $X'_a = X'_b$. So $a, b \in F(X'_a)$ map to the same element in $F(\pi_\beta(X'_a))$ by the isomorphism $F(\pi_\beta)$, and therefore must be equal. $\blacktriangleleft$

6.2 A Syntactic Condition

We recall the notion of κ -ary algebraic theory for κ a regular cardinal. A signature consists of a set of operations Σ all associated with an arity, which is a κ -small set. Given a signature, one can form the set $\mathbf{Tm}(X)$ of terms with free variables in X inductively in the standard way:

- If $x \in X$ then $x \in \mathbf{Tm}(X)$
- If $\text{op} \in \Sigma$ has arity A and $t : A \rightarrow \mathbf{Tm}(X)$, then $\text{op}(t) \in \mathbf{Tm}(X)$

An equation is a pair of terms (t, s) in some $\mathbf{Tm}(X)$. An algebraic theory is a signature together with a set of equations over that signature.

An algebraic theory induces a monad on $\mathbf{Set}$ by the definition

$$T(X) = \mathbf{Tm}(X) / \sim$$

where $\sim$ is the smallest congruence relation generated by the equations of the theory. It is well-known (see e.g. [4]) that the underlying functor of T is then κ -accessible.

► **Example 10.** The finite powerset monad is generated by the finitary ($\aleph_0$ -ary) theory consisting of two operations $\perp$ of arity $\emptyset$ and $\vee$ of arity $\{0, 1\}$. We write the latter as an infix binary operator. The equations are

$$x \vee y = y \vee x \quad (x \vee y) \vee z = x \vee (y \vee z) \quad x \vee x = x \quad x \vee \perp = x$$

► **Example 11.** The countable powerset monad is generated by the theory consisting of $\perp$ of arity $\emptyset$ and $\bigvee$ of arity $\mathbb{N}$. Following Chapman et al. [10], the equations can be expressed by writing $x \vee y$ for $\bigvee(x, y, y, y, \dots)$ as

$$\begin{aligned} x \vee y &= y \vee x & (x \vee y) \vee z &= x \vee (y \vee z) & x \vee x &= x & x \vee \perp &= x \\ x_i \vee \left(\bigvee x\right) &= \bigvee x & \left(\bigvee x\right) \vee y &= \bigvee (x_i \vee y)_{i \in \mathbb{N}} \end{aligned}$$

This is an $\aleph_1$ -ary algebraic theory.

► **Example 12.** The finite distributions monad is generated by a family of binary operations $\oplus_p$ indexed by p in the open interval $(0, 1)$ together with the equations

$$x \oplus_p x = x \quad x \oplus_p y = y \oplus_{1-p} x \quad (x \oplus_p y) \oplus_q z = x \oplus_{pq} (y \oplus_{\frac{(1-p)q}{1-pq}} z)$$

► **Example 13.** The propositional truncation monad of Example 7 is generated by the algebraic theory of no constructors and a single equation $(x = y)$.

We now give a syntactic condition for an algebraic theory ensuring that the generated monad satisfies the conditions of Theorem 8.

► **Definition 14.** An equation (t, s) is a drop equation if $\text{fv}(t) \neq \text{fv}(s)$.

Here $\text{fv}(t)$ for $t \in \text{Tm}(X)$ is the smallest subset $X' \subseteq X$ such that $t \in \text{Tm}(X')$. Of the four examples mentioned above, only propositional truncation has drop equations. Note that if a theory has no drop equations, then also the least congruence on terms generated by the equations has no drop equations.

► **Theorem 15.** Suppose T is a monad generated by a κ -ary algebraic theory with no drop equations. Then T preserves pullbacks of monomorphisms and κ -small intersections. As a consequence, if ρ is a regular ordinal with an $\alpha \in \rho$ of same cardinality as κ , then T preserves ρ^{op} -limits.

Proof. Let $f : X \rightarrow Y$ be a map and let $Z \subseteq Y$. We must show that the following is a pullback

$$\begin{array}{ccc} T(f^{-1}(Z)) & \xrightarrow{T(f)} & T(Z) \\ \downarrow & & \downarrow T(i) \\ T(X) & \xrightarrow{T(f)} & T(Y) \end{array}$$

where $i : Z \rightarrow Y$ is the inclusion. So suppose given $[s]_Z \in T(Z)$ and $[t]_X \in T(X)$ such that $T(i)[s]_Z = T(f)[t]_X$. Let t_f be the result of substituting the free variables of t using f , so that $T(f)[t]_X = [t_f]_Y$. The assumption says that t_f and s are related in the congruence relation generated by the equalities of the algebraic theory. Since none of these are drop equations, t_f and s have the same free variables. Since the free variables of s are in Z , the free variables of t must be in $f^{-1}(Z)$. So $[t] \in T(f^{-1}(Z))$ and maps to $[s]_Z \in T(Z)$ and $[t]_X \in T(X)$. This proves existence.

To prove uniqueness, it suffices to show that T preserves monomorphisms, implying that the map $T(f^{-1}(Z)) \rightarrow T(X)$ is mono. So suppose $A \subseteq B$, $j : A \rightarrow B$ is the inclusion, and that t and s are terms with free variables in A such that $T(j)[s] = T(j)[t]$. Since there are no more equations between terms in B than in A , also $[t] = [s]$.

Showing that T preserves κ -small intersections is similar: Given a family $([t_i] \in T(X_i))_{i \in I}$, with each $X_i \subseteq X$, the free variables in all t_i must be the same, so that they all represent the same equivalence class in $T(\cap_{i \in I} X_i)$. $\blacktriangleleft$

As a non-example, the pullback

$$\begin{array}{ccc} \emptyset & \longrightarrow & \{0\} \\ \downarrow & \lrcorner & \downarrow \\ \{1\} & \longrightarrow & \{0, 1\} \end{array}$$

witnesses that propositional truncation does not preserve pullbacks of monomorphisms.

7 Existential quantification

The previous section handled the first requirement listed in Section 4.4. We now look at the last two, which both concern commutativity of universal quantification over clocks and existential quantification. The property we need can be stated in Clocked Type Theory as

$$\Delta, X : \mathbf{U}_\Delta, \phi : X \rightarrow \forall \kappa. \text{Prop}_{\Delta, \kappa} \vdash \exists (x : X). \forall \kappa. \phi(x) [\kappa] = \forall \kappa. \exists (x : X). \phi(x) [\kappa] : \text{Prop}_\Delta \quad (4)$$

where $\Delta = \kappa_1 : \text{clock}, \dots, \kappa_n : \text{clock}$ is a clock context of length n . We first note that this principle does not hold in general.

► **Example 16.** Let Δ be the empty context and let $X = \rho$ be the indexing ordinal. Let ϕ be the predicate on $\rho \times \text{Clk}$ defined as

$$\phi(\mathcal{E}; \delta; \alpha; \lambda) = \{\star \mid \delta(\lambda) \leq \alpha\}$$

Since this predicate is invariant under clock introduction, it corresponds to a map $X \rightarrow \forall \kappa. \text{Prop}_{\Delta, \kappa}$, but $\forall \kappa. \exists (\alpha : \rho). \phi(\alpha) [\kappa] = \{\star\}$ and

$$\begin{aligned} (\exists (\alpha : \rho). \forall \kappa. \phi(\alpha) [\kappa])(\mathcal{E}; \delta) &= \{\star \mid \exists \alpha : \rho. \forall \beta. \phi((\mathcal{E}, \lambda_\mathcal{E}); \delta[\lambda_\mathcal{E} \mapsto \beta]; \alpha; \lambda_\mathcal{E})\} \\ &= \emptyset \end{aligned}$$

because the condition fails when $\beta > \alpha$. Such β always exist, because ρ is assumed to be a limit ordinal, as it needs to be to interpret axiom (2).

The above example is a variant of the well-known phenomenon that existential quantification in the internal language of a presheaf topos is local and does not imply global existence. This has been observed previously also in the setting of guarded recursion [7, 22].

To understand the principle more generally, we look at substitutions from some object Γ in $\mathbf{GR}$ into the context of (4). A mapping $\Gamma \rightarrow \text{Clk}^\Delta$ corresponds to an object of $\mathbf{GR}/\text{Clk}^\Delta$ which is equivalent to $\mathbf{GR}[\Delta]$, so a mapping into the context of (4) corresponds to an object Γ in $\mathbf{GR}[\Delta]$, a family X over Γ , and finally, a predicate ϕ on $X \times \text{Clk}$. The equality (4) then translates to equality of $(\exists (x : X). \forall \kappa. \phi(x))((\mathcal{E}, \delta); \chi; \gamma)$ which is

$$\{\star \mid \exists x : X((\mathcal{E}, \delta); \chi; \gamma). \forall \alpha < \rho. \phi((\mathcal{E}, \lambda_\mathcal{E}); \delta[\lambda_\mathcal{E} \mapsto \alpha]; \iota\chi; \iota \cdot \gamma; \iota \cdot x; \lambda_\mathcal{E})\} \quad (5)$$

and $(\forall \kappa. \exists (x : X). \phi(x))(\mathcal{E}; \delta; \chi; \gamma)$ which is

$$\{\star \mid \forall \alpha < \rho. \exists x : X((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \alpha]; \iota\chi; \iota \cdot \gamma). \phi((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \alpha]; \iota\chi; \iota \cdot \gamma; x; \lambda_{\mathcal{E}})\} \quad (6)$$

for all $\gamma \in \Gamma(\mathcal{E}; \delta; \chi)$ and $\chi : \Delta \rightarrow \mathcal{E}$.

► **Theorem 17.** *If ρ is a regular ordinal and all fibres $X(\mathcal{E}; \delta; \chi; \gamma)$ are of cardinality strictly smaller than ρ , then (5) and (6) are equal.*

Proof. Since X is invariant under clock introduction, (6) can be rewritten as

$$\{\star \mid \forall \alpha < \rho. \exists x : X(\mathcal{E}; \delta; \chi; \gamma). \phi((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \alpha]; \iota\chi; \iota \cdot \gamma; \iota \cdot x; \lambda_{\mathcal{E}})\} \quad (7)$$

Clearly the condition of (5) implies that of (7). For the other direction, first note that the condition is downward closed in α because, for $\beta \leq \alpha$, the identity on $(\mathcal{E}, \lambda_{\mathcal{E}})$ tracks a map σ which gives

$$\sigma \cdot (-) : \phi((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \alpha]; \iota\chi; \iota \cdot \gamma; \iota \cdot x; \lambda_{\mathcal{E}}) \rightarrow \phi((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \beta]; \iota\chi; \iota \cdot \gamma; \iota \cdot x; \lambda_{\mathcal{E}})$$

Now, arguing classically, suppose (5) is false. For each x there must exist an α_x , such that

$$\phi((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \beta]; \iota\chi; \iota \cdot \gamma; \iota \cdot x; \lambda_{\mathcal{E}})$$

is false for all $\beta \geq \alpha_x$. By the condition of the cardinality of the fibres of X and since ρ is regular, the least upper bound α of all the α_x is an ordinal which is smaller than ρ . This ordinal then proves (6) false. ◀

The above proof is essentially the same as the proof of the “existential property” forming part of the basis of Transfinite Iris [22, Theorem 5.2], but adapted to the multi-clocked setting. The existential property states that any existential proved in the logic of Transfinite Iris externalises to an existential in the meta-logic. Transfinite Iris uses a large indexing ordinal to ensure that the indexing ordinal is larger than the existentially quantified type. We will discuss this point further in the conclusion.

Universal quantification over clocks also commutes with essentially unique existential quantifications.

► **Theorem 18.** *Let Γ be an object of $\text{GR}[\Delta]$, X a family over Γ , and ϕ a predicate over $X \times \text{Clk}$, and let $n : \Gamma \rightarrow \mathbb{N}$ be a map of presheaves. If $\phi(x, \kappa) \wedge \phi(y, \kappa)$ implies $(\triangleright^{\kappa})^n(x = y)$ then (5) and (6) are equal.*

Proof. The predicate

$$\Delta, \Gamma, x : X, y : X, \kappa : \text{clock} \vdash (\triangleright^{\kappa})^n(x = y)$$

is true at $(\mathcal{E}; \delta; \chi; \gamma; x; y; \lambda)$ if and only if either $x = y$ or $\delta(\lambda) < n(\gamma)$. So, for α larger than $n(\gamma)$, there exists at most a single x such that

$$\phi((\mathcal{E}, \lambda_{\mathcal{E}}); \delta[\lambda_{\mathcal{E}} \mapsto \alpha]; \iota\chi; \iota \cdot \gamma; \iota \cdot x; \lambda_{\mathcal{E}})$$

Suppose now that (7) holds, and pick α larger than $n(\gamma)$. By (7) there is an x such that the above holds for that α and x . By downward closedness, it then also holds at all $\beta \leq \alpha$. For $\beta \geq \alpha$, there is a y such that the above holds at β for that y . By downwards closure and uniqueness, we then conclude $x = y$, so that x witnesses (5). ◀

8 Conclusion

We have seen that by choosing the indexing ordinal of the model to be sufficiently big, all the requirements of the applications studied in Section 4 can be met. For example, to program with finitely branching non-deterministic processes, and reasoning about bisimilarity of these using guarded recursion, as done by Kristensen et al [17] the indexing ordinal can be ω_1 (the first uncountable ordinal) or higher. For modelling Probabilistic FPC and reasoning about contextual equivalence of programs, the indexing ordinal needs to be larger than the cardinality of $D_f(D^\forall 1)$, which is uncountable. Finally, for reasoning about the coinductive delay monad up to weak bisimilarity as a monad on the category of setoids, one needs the indexing monad to be ω_1 or higher, such that quantification over clocks commutes with existential quantification over $\mathbb{N}$. In all these examples, we can now interpret the results proved in previous papers into the model and conclude the expected set theoretic results. For example, Theorem 7.7 of [21] now proves the existence of a distributive law of the monad generated by an algebraic theory with no drop equations over the coinductive delay monad up to weak bisimilarity as monads on the usual category of setoids. Likewise the results about contextual equivalence of Probabilistic FPC programs proved in [23] now hold for the operational semantics given in that paper as interpreted in the usual set-theoretic way, where $D^\forall$ is a coinductively defined monad.

The syntactic requirements on algebraic theories to generate monads commuting with clock quantification are quite natural from a type theoretic point of view: The arities should be of a particular form and the equations non-drop. The latter corresponds to the equations being typeable in a relevant type system (in the technical sense of substructural types). The requirements on cardinality for existential quantifications, on the other hand, are less natural in type theory. One way to avoid this is to use the ordinal of all ordinals as indexing, as done in Transfinite Iris [22]. Since this ordinal is large (lives in the next universe), it suffices to commute clock quantification with all small existential quantifications. This simplification comes at a price, however: Universal quantification over clocks now raises the universe level for small types. Monads such as $D^\forall$ defined using universe quantification will therefore only be monads on the category of sets in the next universe. In particular, the type $D^\forall 1$ will not be small, and so existential quantification over it will not automatically commute with universal quantification over clocks, unless one can prove $D^\forall 1$ isomorphic to a small set.

An alternative to proving type constructors commute with clock quantification is to use Worrells proof [27] that for any κ -accessible endofunctor F on **Set**, the terminal sequence converges in $\kappa + \kappa$ steps, producing a terminal coalgebra for F . This proves that the encoding of coinductive types for those functors is correct, externally. While this result is more general (in the sense that it works for all cardinals κ), it is also less natural and flexible from the type theoretic point of view. From the computational point of view, Kristensen et al. [17] develop a notion of induction under clocks that provides computational content to axioms of higher-inductive types or quotient inductive types commuting with clock quantification. It is not clear how to provide computational content to axioms of being a terminal coalgebra.

Multimodal Type Theory [15] gives a different approach to reasoning about the relationship between **Set** and the topos of trees in one type theory. Each of these categories correspond to a mode of the type theory, and the functors of constant presheaves and global elements are internalised as modalities between these two modes. While the use of modes makes the relationship between these categories explicit in the language, as we saw in Section 5 the same two models form submodels of the model of multiclocked guarded recursion. The results of Sections 6 and 7 can also be applied in the setting of Multimodal Type Theory.

References

- 1 Jiří Adámek. Free algebras and automata realizations in the language of categories. *Commentationes Mathematicae Universitatis Carolinae*, 15(4):589–602, 1974.
- 2 Jiří Adámek and Václav Koubek. On the greatest fixed point of a set functor. *Theoretical Computer Science*, 150(1):57–75, 1995. doi:10.1016/0304-3975(95)00011-K.
- 3 Jiří Adámek, Stefan Milius, Lurdes Sousa, and Thorsten Wissmann. On finitary functors. *Theory and Applications of Categories*, 34(35):1134–1164, 2019.
- 4 Jiří Adámek and Jiří Rosický. *Locally Presentable and Accessible Categories*, volume 189 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1994.
- 5 Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. *ACM SIGPLAN Notices*, 48(9):197–208, 2013. doi:10.1145/2500365.2500597.
- 6 Patrick Bahr., Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The clocks are ticking: No more delays! In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2017. doi:10.1109/LICS.2017.8005097.
- 7 Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:1)2012.
- 8 Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 20–35. Springer, 2016.
- 9 Aleš Bizjak and Rasmus Ejlers Møgelberg. Denotational semantics for guarded dependent type theory. *Mathematical Structures in Computer Science*, 30(4):342–378, 2020. doi:10.1017/S0960129520000080.
- 10 James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the delay monad by weak bisimilarity. *Mathematical Structures in Computer Science*, 29(1):67–92, 2019. doi:10.1017/S0960129517000184.
- 11 Ranald Clouston. Fitch-style modal lambda calculi. In *International Conference on Foundations of Software Science and Computation Structures*, pages 258–275. Springer, 2018. doi:10.1007/978-3-319-89366-2_14.
- 12 Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science*, 30(2):118–138, 2020. doi:10.1017/S0960129519000197.
- 13 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.TYPES.2015.5.
- 14 Eric Giovannini, Tingting Ding, and Max S. New. Denotational semantics of gradual typing using synthetic guarded domain theory. *Proc. ACM Program. Lang.*, 9(POPL):772–801, 2025. doi:10.1145/3704863.
- 15 Daniel Gratzer, GA Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 492–506, 2020.
- 16 Martin Hofmann and Thomas Streicher. Lifting Grothendieck universes. Unpublished, 1999. URL: www.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf.
- 17 Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. Greatest hits: Higher inductive types in coinductive definitions via induction under clocks. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 42:1–42:13. ACM, 2022. doi:10.1145/3531130.3533359.
- 18 Bassel Mannaa, Rasmus Ejlers Møgelberg, and Niccolò Veltri. Ticking clocks as dependent right adjoints: Denotational semantics for clocked type theory. *Logical Methods in Computer Science*, 16, 2020. URL: <https://lmcs.episciences.org/6980>.

- 19 Rasmus Ejlers Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10, 2014. doi:10.1145/2603088.2603132.
- 20 Rasmus Ejlers Møgelberg and Andrea Vezzosi. Two guarded recursive powerdomains for applicative simulation. In Ana Sokolova, editor, *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics, MFPS 2021, Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021*, volume 351 of *EPTCS*, pages 200–217, 2021. doi:10.4204/EPTCS.351.13.
- 21 Rasmus Ejlers Møgelberg and Maaïke Zwart. What monads can and cannot do with a few extra pages. *Logical Methods in Computer Science*, 21, 2025. doi:10.46298/LMCS-21(4:5)2025.
- 22 Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite iris: resolving an existential dilemma of step-indexed separation logic. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 80–95, 2021. doi:10.1145/3453483.3454031.
- 23 Philipp Stassen, Rasmus Ejlers Møgelberg, Maaïke Zwart, Alejandro Aguirre, and Lars Birkedal. Modelling recursion and probabilistic choice in guarded type theory. *Proc. ACM Program. Lang.*, 9(POPL):1417–1445, 2025. doi:10.1145/3704884.
- 24 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- 25 Niccolò Veltri and Andrea Vezzosi. Formalizing π -calculus in guarded cubical agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 270–283, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373814.
- 26 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019. doi:10.1145/3341691.
- 27 James Worrell. On the final sequence of a finitary set functor. *Theor. Comput. Sci.*, 338(1-3):184–199, 2005. doi:10.1016/j.tcs.2004.12.009.

Nominal Type Theory by Nullary Internal Parametricity

Antoine Van Muylder  

DistriNet, KU Leuven, Belgium

Andreas Nuyts  

DistriNet, KU Leuven, Belgium

Dominique Devriese  

DistriNet, KU Leuven, Belgium

Abstract

There are many ways to represent the syntax of a language with binders. In particular, nominal frameworks are metalanguages that feature (among others) name abstraction types, which can be used to specify the type of binders. The resulting syntax representation, nominal data types, makes alpha-equivalent terms equal, has a closed induction principle and is well-behaved w.r.t. weakening. It is known that name abstraction types can be presented either as an existential or as a universal quantification on names. Existential name abstractions support matching on name-binding patterns but have cumbersome typing rules; universal ones have clean rules but apparently no such nominal matching. In this work we show that this matching ability and other nominal features are recovered in a type theory consisting of (1) Nullary (0-ary) Internally Parametric Type Theory (nullary PTT), (2) a type of names and a novel name induction principle, (3) nominal data types. This type theory is a legitimate nominal framework: it has universal and (non-primitive) existential name abstractions, a freshness type former, restricted name swapping and local-scope operations. Nominal pattern matching is recovered via term-relevant nullary parametricity. We provide an example involving synthetic Kripke parametricity.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Nominal techniques, Parametricity

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.12

Funding Andreas Nuyts holds a Postdoctoral Fellowship from the Research Foundation – Flanders (FWO; 12AB225N). This research is partially funded by the Research Fund KU Leuven and the Internal Funds KU Leuven.

Acknowledgements We thank the reviewers for their time and helpful comments.

1 Introduction

There are many ways to formally define the syntax of a language with binders. In particular, nominal frameworks [25, 33, 28, 32, 11, 35] are metalanguages featuring (among others) a primitive type of names Nm as well as a “name abstraction” type former which we write $@N \multimap -$. Name abstraction types can be used to specify the type of binders of a given object language. For example, we can define the syntax of the untyped lambda calculus (ULC) with the following data type.

```
data Ltm :  $\mathcal{U}$  where
  var :  $Nm \rightarrow Ltm$ 
  app :  $Ltm \rightarrow Ltm \rightarrow Ltm$ 
  lam :  $(@N \multimap Ltm) \rightarrow Ltm$ 
```



© Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese;
licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 12; pp. 12:1–12:23

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The resulting representation is called a “nominal data type” or “nominal syntax” and offers several advantages. First, α -equivalent terms are definitionally equal, a consequence of how $@N \multimap -$ is axiomatized. Second, nominal data types have a (closed) induction principle. Third, by contrast with the De Bruijn representation, weakening by a name simply means extending the context with that name, and all programs in the metalanguage are automatically stable under this operation. Fourth, formal reasoning with nominal syntax can sometimes match on-paper reasoning [35]. A drawback of this representation is that it is not definable in plain type theory.

Inference rules for the name abstraction type former can be defined in two ways: either in an existential/positive, or in a universal/negative fashion. Interestingly, these presentations are semantically equivalent [32, 31] but syntactically have different pros and cons.

On the one hand, the existential presentation makes the name abstraction type former behave as an existential quantification on names. So an element of that type, i.e. a name abstraction, can be understood as a *name-term pair up to α -renaming* $\langle x, a \rangle$ where a may bind the freshly chosen name x . This presentation is convenient since the user is allowed to *pattern match* on such “binding” name-term pairs, an ability that we call nominal pattern matching. The following example is Example 2.1 from [33] and illustrates this ability. It is a (pseudo-) FreshML program computing the equality modulo α -renaming of two existential name abstractions. In the example A has decidable equality $\text{eq}_A : A \rightarrow A \rightarrow \text{Bool}$ and the existential name abstraction type is written $@N \cdot -$.

```
eqabs : (@N · A) → (@N · A) → Bool
eqabs ⟨x0, a0⟩ ⟨x1, a1⟩ = eqA (swap x0, x1 in a0) a1
```

Note (1) the occurrences of x_0, x_1 outside of $\langle x_0, a_0 \rangle, \langle x_1, a_1 \rangle$ and (2) the appearance of the **swap** operation exchanging free names.

Nominal pattern matching is convenient and for that reason nominal frameworks use the existential presentation for practical reasoning about nominal syntax. However, inference rules for existential abstraction types are cumbersome to specify. The issue is that a name x is considered fresh in a binding pair $\langle x, a \rangle$, and such information must be encoded at the type level and propagated when pattern matching. The consequence is that the rules for existential abstraction types are polluted with typal freshness information. This makes the implementation of such rules harder in a proof assistant environment.

On the other hand, the universal presentation treats name abstractions not as pairs, but rather as *functions* consuming fresh names (a.k.a. affine or fresh functions), as in [28, 11]. This has several consequences: names can only be used when they are in scope, no explicit swapping primitive is needed and the inference rules are straightforward to specify. However one seems to lose the important ability to pattern match.

In this paper we will propose a new foundation for nominal frameworks alleviating the above issues and relying on the notion of *parametricity*. Concisely put, n -ary parametricity asserts that types behave as n -ary reflexive graphs. More precisely, the n -ary parametricity of a type T computes by induction on T a certain n -ary reflexive graph structure for T . Even more precisely, (1) the n -ary parametricity of a (open) type T computes to a certain (open) n -ary relation written $[T]_n : T \rightarrow \dots \rightarrow T \rightarrow \mathcal{U}$, (2) the n -ary parametricity of a (open) term $t : T$ computes to a certain reflexivity edge at t , i.e. $[t]_n : [T]_n t \dots t$. Parametricity is most often used in its binary form, at types of polymorphic functions, where it specializes to a uniformity property [30, 38]. For example if $t : T = \forall(X : \mathcal{U}). X \rightarrow X$ then $[t]_2 : [T]_2 t t$ is a proof that t maps related types to related functions. Indeed $[-]_2$ is defined by induction (not given here) and $[T]_2 t t$ reduces to $\forall(X_0 X_1 : \mathcal{U})(R : X_0 \rightarrow X_1 \rightarrow \mathcal{U}). \forall x_0 x_1. R x_0 x_1 \rightarrow R (t X_0 x_0) (t X_1 x_1)$. Intuitively this property disallows t from inspecting its type argument $X : \mathcal{U}$. It can be shown that the property entails that t equals the identity $\lambda X x. x$.

$K : \mathcal{U}$	Bridge $K \ k_0 \ k_1 \simeq \dots$	$k_0 \equiv_K k_1 \simeq \dots$
$A \rightarrow B$	$\forall a_0 \ a_1. \text{Bridge } A \ a_0 \ a_1 \rightarrow \text{Bridge } B \ (k_0 \ a_0) \ (k_1 \ a_1)$	$\forall a_0 \ a_1. a_0 \equiv_A a_1 \rightarrow k_0 \ a_0 \equiv_B k_1 \ a_1$
$A \times B$	Bridge $A \ (k_0.\text{fst}) \ (k_1.\text{fst}) \times$ Bridge $B \ (k_0.\text{snd}) \ (k_1.\text{snd})$	$(k_0.\text{fst}) \equiv_A (k_1.\text{fst}) \times$ $(k_0.\text{snd}) \equiv_B (k_1.\text{snd})$
$\mathcal{U}$	$k_0 \rightarrow k_1 \rightarrow \mathcal{U}$	$k_0 \simeq k_1$

■ **Figure 1** The Bridge and Path type formers commute with some example type formers.

In the above account, parametricity is regarded as a property that is defined and proven to hold *externally* about a dependent type theory (DTT), as in [9]. In other words, the parametricity property, or translation, is a map $[-]_n$ defined by induction on the (open) types and terms of DTT. By contrast, n -ary *internally* parametric type theory (n -ary PTT for short) [10, 17, 20, 22, 2, 1] extends DTT with new type and term formers. These primitives make it possible to prove parametricity results *within* n -ary PTT.

To that end, n -ary PTT typically provides two kinds of primitives internalizing aspects of reflexive graphs. Firstly, Bridge types are provided, which intuitively are to a type what an edge is to a reflexive graph. Syntactically, a bridge $q : \text{Bridge } A \ a_0 \dots a_{n-1}$ at type A between $a_0, \dots, a_{n-1}$ is treated as a function $\mathbb{N} \rightarrow A$ out of a posited bridge interval $\mathbb{N}$. The $\mathbb{N}$ interval contains n endpoints $(e_i)_{i < n}$ and q must respect these definitionally $q \ e_i = a_i$. When $n = 2$ this is similar to the “paths” of cubical type theory [37] which play the role of equality proofs in the latter. However, bridges do not satisfy various properties of paths, e.g. they cannot be composed and one cannot transport values of a type $P \ x$ over a bridge $\text{Bridge } A \ x \ y$ to type $P \ y$. Moreover, contrary to paths, bridges may only be applied to variables $x : \mathbb{N}$ that do not appear freely in them, i.e. that are fresh for the bridge. Functions with such a freshness side condition are also known as affine or fresh functions and we will use the symbol $\multimap$ instead of $\rightarrow$ for their type. Secondly, the other primitives of n -ary PTT make it possible to prove that the Bridge type former has a commutation law with respect to every other type former. Figure 1 lists some of these laws (equivalences) for arity $n = 2$ and compares the situation for bridges and paths. Path types are written $\equiv$.

The last column in Figure 1 excerpts a collection of equivalences known as the Structure Identity Principle (SIP) in HoTT/UF [34]. Concisely, it states that at every type K , equality is equivalent to observational equality [3]. The Bridge column of Figure 1 shows an analogous Structure Relatedness Principle (SRP) [36] which expresses equivalence of K ’s Bridge type to the parametricity translation of K , i.e. the Bridge type former internalizes the parametricity translation for types *up to SRP equivalences*. Accordingly, in this setting the parametricity of a term k is k ’s reflexivity bridge $\lambda(\_ : \mathbb{N}). k : \text{Bridge } K \ k \dots k$ mapped through the SRP equivalence at K .

Parametric Nominal Type Theory. In this work, we put forward and elaborate an idea that existed in the community [18, 21], which is to use nullary PTT (i.e. 0-ary PTT) as the foundation for nominal dependent type theory. Specifically, we contribute Parametric Nominal Type Theory (PNTT). Concisely put, PNTT = nullary PTT + a type of names with an induction principle + rules for nominal data types.

In more detail, PNTT is obtained from nullary PTT in several steps. First, we make the simple observation that universal name abstraction types have the same rules as the nullary Bridge types from nullary PTT: nullary bridges and universal abstractions are simply affine functions out of the interval. Second, we show that, on its own, nullary PTT already supports important features of nominal frameworks: universal name abstractions (bridge

types), typal freshness, (non-primitive) existential name abstractions, a name swapping operation and the local-scoping primitive ν of [28] (roughly, the latter primitive is used to witness freshness). Third, we can recover nominal pattern matching in this parametric setting by extending nullary PTT with a full-fledged type of names $\vdash \mathbf{Nm}$. It comes equipped with a novel eliminator called *name induction*, which expresses for a term $\Gamma \vdash n : \mathbf{Nm}$ and a bridge variable x in Γ , that either n is just x , or x is *fresh* in n . Fourth, PNTT also has rules for the nominal data types we wish to study, like the $\mathbf{Ltm}$ type.

Nominal pattern matching can be indirectly recovered via an extended version of the nullary SRP: the Structure Abstraction Principle (SAP), as we call it. The SAP expresses that the nullary bridge type former $@N \multimap -$ commutes with every type former in a specific way, including (1) standard type formers, (2) the $\mathbf{Nm}$ type and (3) nominal data types. For standard type formers, the SAP asserts that name abstraction commutes as one might expect, e.g. $(@N \multimap A \rightarrow B) \simeq ((@N \multimap A) \rightarrow (@N \multimap B))$. For $\mathbf{Nm}$, the SAP asserts that $(@N \multimap \mathbf{Nm}) \simeq 1 + \mathbf{Nm}$, which can be proved by name induction. The SAP also holds for nominal data types. For example, the SAP for $\mathbf{Ltm}$ asserts that there is an equivalence $e : (@N \multimap \mathbf{Ltm}) \simeq \mathbf{Ltm}_1$ between $@N \multimap \mathbf{Ltm}$ and the following data type (this was proved semantically by Hofmann [14]). Intuitively, it corresponds to $\mathbf{Ltm}$ terms with $\mathbf{Nm}$ -shaped holes and we say it ought to be the nullary parametricity translation of $\mathbf{Ltm}$.

```
data Ltm1 :  $\mathcal{U}$  where
  hole : Ltm1
  var  : Nm  $\rightarrow$  Ltm1
  app  : Ltm1  $\rightarrow$  Ltm1  $\rightarrow$  Ltm1
  lam  : (@N  $\multimap$  Ltm1)  $\rightarrow$  Ltm1
```

When matching on $\text{lam } g : \mathbf{Ltm}$ we have $g : @N \multimap \mathbf{Ltm}$ and equivalently $e g : \mathbf{Ltm}_1$ for which the induction principle of $\mathbf{Ltm}_1$ applies.

Contributions and Outline. In this paper, we propose nullary PTT as the foundation for nominal dependent type theory, and provide evidence for the suitability of this foundation.

- In Section 2, we present Parametric Nominal Type Theory (PNTT). PNTT is a variant of the univalent parametric type theory of Cavallo and Harper [10] where (1) we replace the arity $n = 2$ by $n = 0$ (see Section 2.1), (2) we construct a type $\mathbf{Nm}$ from the bridge interval $@N$ and provide a name induction principle (see Section 2.2), (3) we add rules for the specific nominal data types we wish to study. We explain how the above primitives validate our Structure Abstraction Principle (SAP). We discuss semantics and soundness in Section 2.4.
- In Section 3, we systematically discuss the inference rules of existing nominal frameworks for typal freshness, name swapping, the local-scoping primitive ν , existential and universal name abstractions. We explain in detail how they can all be (adapted and) implemented in terms of nullary PTT primitives.
- Section 4 shows nominal techniques in PNTT in action. Section 4.1 shows that the nullary translation of data types lets us indirectly emulate nominal pattern matching: defining functions by matching on patterns which bind variables. Section 4.2 connects the $\mathbf{Ltm}$ nominal data type to a nominal HOAS representation, and serves two purposes. First, it illustrates that more often than not, proving the correctness of a function $f : D \rightarrow E$ defined by recursion out of a nominal data type D requires computing the parametricity translation of f , i.e. the parametricity of f is term-relevant. Second, the example sheds some light on the notion of *Kripke* binary parametricity [5]. A proof in Atkey’s Kripke model is ported to PNTT, and the Kripke aspect of it is emulated by nullary parametricity.

Binary parametric cubical type theory has already been implemented [36]. Thus our work offers a clear path to a practical implementation of nominal type theory. To our knowledge, we are also the first to make explicit and active use of nullary parametricity. We discuss related work in more detail in Section 5.

2 Parametric Nominal Type Theory (PNTT)

PNTT is largely based on the binary PTT of Cavallo and Harper (CH type theory, [10]). Concretely, our rules are obtained by (1) considering the rules of the parametricity primitives of the CH binary PTT and replacing the arity 2 by 0 instead, (2) adding novel rules to turn the bridge interval $@N$ into a full-fledged type Nm , including a name induction principle and (3) adding rules for the desired nominal data types. For the impatient reader, the relevant rules of PNTT appear in Figure 2 and $\text{Gel } Ax$ can be understood as the elements of A for which x is fresh.

Cubical type theory. Apart from its parametricity primitives, the CH theory is a cubical type theory in the way it handles equality proofs (*paths*), and so is our system. Cubical type theory (cubical TT, [12, 4]) is a form of homotopy type theory (HoTT, [34]) that adds new types, terms and equations on top of plain dependent type theory. These extra primitives make it possible, among other things, to prove univalence and more generally the SIP (see Figure 1) at any concrete, fully known type.

The SIP matters to us because, e.g., univalence is used to characterize the bridge type at the universe $(@N \multimap \mathcal{U}) \simeq \mathcal{U}$. Otherwise, the fact that our system is a cubical type theory is not of primary significance, although we wish to remain as close as possible to the CH theory to be able to reuse their proofs and semantics.

Accordingly, we only need to know that the primitives introduced by cubical type theory validate the SIP in order to showcase our theorems and examples. Yet we list these primitives for completeness: (1) the path interval I , dependent path types and their rules; paths play the role of equality proofs thus non-dependent path types are written $\equiv$, (2) the transport and cube-composition operations, a.k.a the Kan operations; these are used e.g. to prove transitivity of the path relation, (3) a type former to turn equivalences into paths in the universe, validating one direction of univalence, (4) higher inductive types (HITs) if desired.

Additionally, in HoTT an equivalence $A \simeq B$ is by definition a function $A \rightarrow B$ with contractible fibers. We rather build equivalences using Theorem 4.2.3 of [34]: let $f : A \rightarrow B$ have a quasi-inverse, i.e. a map $g : B \rightarrow A$ satisfying the two roundtrip equalities $\forall a. g(f a) \equiv a$ and $\forall b. f(g b) \equiv b$. Then f can be turned into an equivalence $A \simeq B$.

2.1 Nullary CH

Let us explain the rules of Figure 2. We begin with the nullary primitives of the CH type theory since our rules for the bridge interval Nm depend on them.

Contexts. There are two ways to extend a context. The first way is the usual “cartesian” comprehension operation where Γ gets extended with a type $\Gamma \vdash A \text{ type}$ resulting in a context $\Gamma, (a : A)$. The second, distinct way to extend a context Γ is an “affine” comprehension operation where Γ gets extended with a bridge variable x resulting in a context written $\Gamma, (x : @N)$. Note that $@N$ is *not* a type and morally always appears on the left of $\vdash$ (the Bridge type former will be written $@N \multimap -$ but this is just a suggestive notation).

$$\begin{array}{c}
\frac{\Gamma, (x : @N) \vdash A \text{ type}}{\Gamma \vdash (x : @N) \multimap A \text{ type}} \multimap F \quad \frac{\Gamma, (x : @N) \vdash a : A}{\Gamma \vdash \lambda x. a : (x : @N) \multimap A} \multimap I \\
\frac{(x : @N) \in \Gamma \quad \Gamma \setminus x \vdash a' : (y : @N) \multimap A}{\Gamma \vdash a' x : A[x/y]} \multimap E \quad \frac{(x : @N) \in \Gamma \quad \Gamma \setminus x, (y : @N) \vdash a : A}{\Gamma \vdash (\lambda y. a) x = a[x/y] : A[x/y]} \multimap \beta \\
\frac{\Gamma, (x : @N) \vdash a'_0 x = a'_1 x : A}{\Gamma \vdash a'_0 = a'_1 : (x : @N) \multimap A} \multimap \eta \quad \frac{\text{prem. of EXT} - a_x \quad \Gamma \setminus x, (y : @N) \vdash a : A}{\Gamma \vdash \text{ext}(\lambda a' y. b) x (a[x/y]) = b[\lambda y. a/a', x/y] : B[x/y, a[x/y]/a]} \text{EXT}\beta \\
\frac{(x : @N) \in \Gamma \quad \Gamma \setminus x, (y : @N) \vdash A \text{ type} \quad \Gamma \setminus x, (y : @N), (a : A y) \vdash B \text{ type} \quad \Gamma \setminus x, (a' : (z : @N) \multimap A[z/y]), (y : @N) \vdash b : B[a'y/a] \quad \Gamma \vdash a_x : A[x/y]}{\Gamma \vdash \text{ext}(\lambda a' y. b) x a_x : B[x/y, a_x/a]} \text{EXT} \\
\frac{(x : @N) \in \Gamma \quad \Gamma \setminus x \vdash A \text{ type}}{\Gamma \vdash \text{Gel } A x \text{ type}} \text{GELF} \quad \frac{\Gamma, (x : @N) \vdash g : \text{Gel } A x}{\Gamma \vdash \text{ung}(\lambda x. g) : A} \text{GELE} \\
\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{ung}(\lambda x. \text{gel } a x) = a} \text{GEL}\beta \\
\frac{(x : @N) \in \Gamma \quad \Gamma \setminus x \vdash a : A}{\Gamma \vdash \text{gel } a x : \text{Gel } a x} \text{GELI} \quad \frac{\text{prem. of GELF} \quad \Gamma \setminus x, (y : @N) \vdash g_0, g_1 : \text{Gel } A y}{\Gamma \vdash \text{ung}(\lambda y. g_0) = \text{ung}(\lambda y. g_1) : A} \text{GEL}\eta \\
\frac{\Gamma \vdash g_0[x/y] = g_1[x/y] : \text{Gel } A x}{\Gamma \vdash g_0[x/y] = g_1[x/y] : \text{Gel } A x} \text{GEL}\eta \\
\frac{\Gamma \text{ ctx}}{\Gamma \vdash \text{Nm type}} \text{NmF} \quad \frac{(x : @N) \in \Gamma}{\Gamma \vdash c x : \text{Nm}} \text{NmI} \\
\frac{(x : @N) \in \Gamma \quad \Gamma \vdash n : \text{Nm} \quad \Gamma, z : \text{Nm} \vdash B \text{ type} \quad \Gamma \vdash b_0 : B[c x/z]}{\Gamma, (g : \text{Gel Nm } x) \vdash b_1 : B[\text{forg } x g/z] \quad \Gamma, (g : \text{Gel Nm } x) \vdash \text{forg } x g := \text{ext}(\lambda g' y. \text{ung } g') x g : \text{Nm}} \text{NmE} \\
\frac{\Gamma \vdash \text{ind}_{\text{Nm}} x n b_0 (\lambda g. b_1) : B[n/z]}{\Gamma \vdash \text{ind}_{\text{Nm}} x n b_0 (\lambda g. b_1) : B[n/z]} \text{NmE} \\
\frac{\text{prem. of NmE} - n}{\Gamma \vdash \text{ind}_{\text{Nm}} x (c x) b_0 (\lambda g. b_1) = b_0 : B[c x/y]} \text{Nm}\beta_0 \quad \frac{\text{prem. of NmE} - n \quad \Gamma \setminus x \vdash n : \text{Nm}}{\Gamma \vdash \text{ind}_{\text{Nm}} x n b_0 (\lambda g. b_1) = b_1[\text{gel } n x/g] : B[n/z]} \text{Nm}\beta_1
\end{array}$$

■ **Figure 2** Parametricity primitives of PNTT. Prem. of $x - y$ means premises of rule x except y . For binding rules such as EXT, we rely on the invertibility of both variable and name abstraction to bind using λ .

The presence of $@N$ is required because the theory treats affine variables in a special way that is ultimately used to prove the SAP (briefly mentioned in Section 1). Specifically, terms, types and substitutions depending on an affine variable $x : @N$ are not allowed to duplicate x in affine positions. So typechecking an expression that depends on $x : @N$ may involve checking that x does not appear freely in some subexpressions. Since variables declared after x in the context may eventually be substituted by terms mentioning x , a similar verification must be performed for them. More formally, such “freshness” statements about free variables are specified in the inference rules using a context restriction operation $\setminus$ (as in [10], section 2.1). If Γ is a context containing $(x : @N)$ then $\Gamma \setminus x$ is the context obtained from Γ by removing x itself, as well as all the cartesian (i.e. not $@N$) variables to the right of x .¹ If $(x : @N) \in \Gamma$ and $\Gamma \setminus x \vdash a : A$ we say that x is fresh in a .

Nullary bridges. The type former of dependent bridges with dependent codomain $\Gamma, (x : @N) \vdash A$ is written $(x : @N) \multimap A$. The type of non-dependent bridges with codomain $\Gamma \vdash A$ is written $@N \multimap A$ and defined as $(\_ : @N) \multimap A$. Introducing a bridge requires providing a term in a context extended with a bridge variable $(x : @N)$. A bridge $a' : (y : @N) \multimap A$ can be eliminated at a variable $(x : @N)$ only if x is fresh in a' . In summary, nullary bridges are treated and written like (dependent) functions out of the $@N$ pretype but are restricted to consume fresh variables only. From a nominal point of view, a bridge $a' : @N \multimap A$ is a name-abstracted value in A .

¹ A detail: variables $i : I$ and cubical constraints are not removed [10, 36] as they cannot depend on x .

Since we will use the theory on non-trivial examples in the next sections, we prefer to explain the other primitives of Figure 2 from a user perspective: we derive programs corresponding to these primitives in the empty context and express types as elements of the universe type $\mathcal{U}$, whose rules are not listed but standard. Furthermore we explain what the equations of Figure 2 entail for these closed programs. The closed programs derived from the CH nullary primitives have a binary counterpart in [36], an implementation of the CH binary PTT. The nullary and binary variants operate in a similar way. The pseudo-code we write uses syntax similar to Agda. We omit universe levels, and $\multimap$ is parsed like $\rightarrow$, e.g., it is right associative. We use $;$ to group several declarations in one line.

Lastly we indicate that the SAP holds at equality types $(a'_0 a'_1 : @N \multimap A) \rightarrow ((x : @N) \multimap a'_0 x \equiv_A a'_1 x) \simeq a'_0 \equiv a'_1$. Proofs of this fact in the binary case can be found in [10, 36]. The nullary proof is obtained by erasing all mentions of (bridge) endpoints.

The extent primitive. The rule for the extent primitive (EXT) together with the rules of $\multimap$ and $\mathcal{U}$ provide a term `ext` in the empty context with the following type.

```
ext : {A : @N  $\multimap$   $\mathcal{U}$ } {B : (x : @N)  $\multimap$  A  $\rightarrow$   $\mathcal{U}$ }
      (f' : (a' : (x : @N)  $\multimap$  A x)  $\rightarrow$  (x : @N)  $\multimap$  B x (a' x))  $\rightarrow$ 
      (x : @N)  $\multimap$  (a : A x)  $\rightarrow$  B x a
```

From a function f' mapping a bridge in A to a bridge in B , the extent primitive lets us build a bridge in the dependent function type formed from A and B , or $\Pi A B$ for short. Concisely put, the extent primitive validates one direction of the SAP at Π . It can be upgraded into the following equivalence, using the β -rule of extent, described below (as a side note, the proof also uses a lemma akin to a propositional η -rule for extent).

$$((a' : (x : @N) \multimap A x) \rightarrow (x : @N) \multimap B x (a' x)) \simeq (x : @N) \multimap (a : A x) \rightarrow B x a$$

By way of comparison, this equivalence appears in [28] (Example 2.2) though the left-to-right direction is implemented via a composite capturing operation instead of a primitive like extent. Proofs of the above equivalence in the binary case can be found in [10, 36]. The nullary proof is obtained by erasing all mentions of endpoints. This equivalence entails $((@N \multimap A) \rightarrow (@N \multimap B)) \simeq @N \multimap (A \rightarrow B)$ in the non-dependent case.

The β -rule of extent (EXT β) is peculiar because a substituted premise $a[x/y]$ appears in the redex on the left-hand side. To compute the right-hand side out of the left-hand side only, the premise a is rebuilt $(*)$ and a term where a variable is bound in a is returned. The step $(*)$ is possible thanks to the restriction in the context $\Gamma \setminus x, (y : @N)$ of a . This is quite technical and explained in [10, 36]. We instead explain what that entails for our `ext` program above. Non-trivial examples of β -reductions for extent will appear in Section 2.2.

In a context Γ containing $(x : @N)$, the term $\Gamma \vdash \text{ext } f' x a$ reduces if $\Gamma \vdash a$ is a term that does not mention cartesian variables strictly to the right of x in Γ , i.e. declared after x in Γ . In particular a can mention x . If such a freshness condition holds, x can in fact be soundly captured in a and the reduction can trigger. By default the reduction does not trigger because $f', x, a \vdash \text{ext } f' x a$ does not satisfy the freshness condition as in this case a is a variable appearing to the right of x in the context.

Gel types. Internally parametric type theories that feature interval-based Bridge types [17, 10] need a primitive to convert an n -ary relation of types into an n -ary bridge. In the CH theory the primitive is called `Gel`. The rules for `Gel` types together with the rules of $\multimap$ and $\mathcal{U}$ imply the existence of the following programs in the empty context. Note that `ung` is called `ungel` in the CH theory.

```

Gel :  $\mathcal{U} \rightarrow @N \multimap \mathcal{U}$ 
gel :  $\{A : \mathcal{U}\} \rightarrow A \rightarrow (x : @N) \multimap \text{Gel } A \ x$ 
ung :  $\{A : \mathcal{U}\} \rightarrow ((x : @N) \multimap \text{Gel } A \ x) \rightarrow A$ 

```

Similar to extent, Gel validates one direction of the SAP, this time at the universe $\mathcal{U}$. The Gel function can be upgraded into an equivalence $\mathcal{U} \simeq (@N \multimap \mathcal{U})$ where $\text{Gel}^{-1} A' := (x : @N) \multimap A' x$, and by using the rules of Gel and ext, and notably univalence. The proof also requires showing that gel and ung are inverses, i.e. the SAP at Gel types $A \simeq (x : @N) \multimap \text{Gel } A \ x$. Again these theorems are proved in [10, 36] in the binary case, and the nullary proofs are obtained by erasing endpoints.

From a nominal perspective, we observe that Gel is a type former that can be used to express freshness information. This can be seen by looking at the GELI rule: canonical inhabitants of $\text{Gel } A \ x$ are equivalently terms $a : A$ such that x is fresh in a . Conversely the ung primitive is a binder that makes available a fresh variable x when a value of type A is being defined, as long as the result value is typically fresh w.r.t. x . The ung primitive can also be used to define a map forgetting freshness information.

```

forg :  $\{A : \mathcal{U}\} \rightarrow (x : @N) \multimap \text{Gel } A \ x \rightarrow A$ 
forg  $\{A\} = \text{ext } [\lambda(g' : (x : @N) \multimap \text{Gel } A \ x)). \lambda(_ : @N). \text{ung } g']$ 

```

To our knowledge the equivalence $\mathcal{U} \simeq (@N \multimap \mathcal{U})$ had not been considered in the literature on nominal techniques. Lastly, $\text{GEL}\eta$ can be used if a certain freshness side condition holds, similar to $\text{EXT}\beta$. We will not need to make explicit use of $\text{GEL}\eta$.

2.2 The Nm type, Name Induction, Nominal Data Types

So far the rules we presented involved occurrences of the bridge interval morally to the left of $\vdash$, as affine comprehensions. Since we wish to use nullary PTT as a nominal framework, we need a first-class type of names $\vdash \text{Nm type}$. Indeed this is needed to express nominal data types, whose constructors may take names as arguments. For example the var constructor of the Ltm type declared in Section 1 has type $\text{var} : \text{Nm} \rightarrow \text{Ltm}$, a regular “cartesian” function type. Constructors must be cartesian functions because that is what the initial algebra semantics of (even nominal) data types dictates. Besides, it is unclear whether there exists a sound notion of data types D with “constructors” of the form $@N \multimap D$.

Perhaps the most important result in our system is the SAP at the type of names Nm which reads $1 + \text{Nm} \simeq (@N \multimap \text{Nm})$. The principle expresses that a bridge at the type of names Nm is either the “identity” bridge $c : @N \multimap \text{Nm}$ or a constant bridge. This principle is proved based on our rules for Nm, explained now. The rules of Nm together with the other rules of Figure 2 entail the existence of the following programs in the empty context.

```

Nm :  $\mathcal{U}$ 
c :  $@N \multimap \text{Nm}$ 
indNm :  $\{B : \text{Nm} \rightarrow \mathcal{U}\} \rightarrow$ 
   $(x : @N) \multimap (n : \text{Nm}) \rightarrow$ 
   $(b\emptyset : B (c \ x)) \rightarrow (b1 : (g : \text{Gel } \text{Nm } x) \rightarrow B (forg \ x \ g)) \rightarrow B \ n$ 

```

The NmI rule expresses that the canonical inhabitants of Nm in a context Γ are simply the affine variables $(x : @N)$ appearing in Γ . The “identity” bridge program c above is derived from that rule. In simple terms, c coerces affine bridge variables x into names $c \ x : \text{Nm}$.

Name induction. The ind_{Nm} program/rule is an induction principle, or dependent eliminator for the type Nm . It expresses that in a context Γ containing an affine variable $(x : @N)$ we can do a case analysis on a term $\Gamma \vdash n : \text{Nm}$. Either x is bound in n and n is in fact exactly $\text{c } x$, or x is *fresh* in n . The call $\text{ind}_{\text{Nm}} x n b_0 b_1$ returns b_0 if $n = \text{c } x$ (see rule $\text{Nm}\beta_0$) and returns b_1 ($\text{gel } n x$) if x is fresh in n (see rule $\text{Nm}\beta_1$). The freshness assumption in b_1 is expressed typally using a Gel type.

We show that the rules $\text{Nm}\beta_0$ and $\text{Nm}\beta_1$ are well-typed, i.e. that the ind_{Nm} program above reduces to something of type Bn in both scenarios. If $n = \text{c } x$ then the reduction result is $\text{ind}_{\text{Nm}} x (\text{c } x) b_0 b_1 = b_0 : B(\text{c } x)$ and $B(\text{c } x) = Bn$. Else if x is fresh in n then $\text{gel } n x$ typechecks and the reduction result is $\text{ind}_{\text{Nm}} x n b_0 b_1 = b_1 (\text{gel } n x) : B(\text{forg } x (\text{gel } n x))$ where forg is the function defined in Section 2.1. And we have:

$$\begin{aligned} \text{forg } x (\text{gel } n x) &= \text{ext } [\lambda(g' : (x : @N) \multimap \text{Gel } \text{Nm } x). \lambda(\_ : @N). \text{ung } g'] x (\text{gel } n x) && (\text{def.}) \\ &= (\lambda(\_ : @N). \text{ung } (\lambda y. \text{gel } n y)) x && (\text{EXT}\beta) \\ &= \text{ung } (\lambda y. \text{gel } n y) = n && (\multimap\beta, \text{GEL}\beta) \end{aligned}$$

The $\text{EXT}\beta$ rule triggers because (1) by assumption, x is fresh in n , i.e. n does not mention x , nor cartesian variables strictly to the right of x (2) thus the term $\text{gel } n x$ mentions x but no cartesian variables declared later. The latter condition is exactly the condition under which this ext call can reduce. To that end, x is captured in the term $\text{gel } n x$ leading to a term $g' := \lambda y. \text{gel } n y$ appearing on the second line.

SAP at Nm. We now prove that $1 + \text{Nm} \simeq @N \multimap \text{Nm}$. We take inspiration from [10] who developed a relational encode-decode technique to prove the SRP at data types. The type Nm is defined via an induction principle, and a similar technique can be applied.

```
decode : 1 + Nm → @N → Nm
decode (inl tt) = λx. c x ;   decode (inr n) = λ_. n

t1 : (@N → Nm) → (x:@N) → 1 + Gel Nm x
t1 n' x = indNm x (n' x) (inl tt) (λ (g:Gel Nm x). inr g)

t2pre : (x:@N) → (1 + Gel Nm x) → Gel (1 + Nm) x
t2pre x (inl tt) = gel (inl tt) x
t2pre x (inr g) = ext [λg'. λy. gel (inr (ung g')) y] x g

t2 : ((x:@N) → 1 + Gel Nm x) → (x:@N) → Gel (1 + Nm) x
t2 = λ s' x. t2pre x (s' x)

encode : (@N → Nm) → 1 + Nm
encode n' = ung (t2 (t1 n'))
```

It remains to prove the roundtrip equalities. The roundtrip for $s : 1 + \text{Nm}$ is obtained by induction on s . If $s = \text{inl tt}$ then $(\text{ung} \circ t_2 \circ t_1 \circ \text{decode}) s = (\text{ung} \circ t_2 \circ t_1)(\lambda x. \text{c } x) \stackrel{\text{Nm}\beta_0}{=} (\text{ung} \circ t_2)(\lambda x. \text{inl tt}) = \text{ung } (\lambda x. \text{gel } (\text{inl tt}) x) = \text{inl tt} = s$. If $s = \text{inr } n$ then $(\text{ung} \circ t_2 \circ t_1 \circ \text{decode}) s = (\text{ung} \circ t_2 \circ t_1)(\lambda x. n) \stackrel{\text{Nm}\beta_1}{=} (\text{ung} \circ t_2)(\lambda x. \text{inr } (\text{gel } n x)) = \text{ung } (\lambda x. \text{ext } [\lambda g' y. \text{gel } (\text{inr } (\text{ung } g')) y] x (\text{gel } n x)) \stackrel{\text{EXT}\beta}{=} \text{ung } (\lambda x. \text{gel } (\text{inr } n) x) = \text{inr } n = s$.

The other roundtrip is the last type in the following chain of equivalences. The first type is a sufficient condition that can be understood as a propositional η -rule for Nm .

$$\begin{aligned} (x : @N) \multimap (n : \text{Nm}) &\rightarrow (n \equiv_{\text{Nm}} \text{ext } [\text{decode} \circ \text{encode}] x n) && \simeq_{(\text{SAP} \rightarrow)} \\ (n' : @N \multimap \text{Nm}) &\rightarrow (x : @N) \multimap (n' x \equiv_{\text{Nm}} (\text{decode} \circ \text{encode}) n' x) && \simeq_{(\text{SAP} \equiv)} \\ &\rightarrow (n' : @N \multimap \text{Nm}) \rightarrow n' \equiv (\text{decode} \circ \text{encode}) n' \end{aligned}$$

12:10 Nominal Type Theory by Nullary Internal Parametricity

■ **Table 1** The Structure Abstraction Principle.

Parameters	K'	$\simeq$	$(x : @N) \multimap K$
$A : @N \multimap \mathcal{U},$ $B : (x : @N) \multimap$ $A x \rightarrow \mathcal{U}.$	$(a' : (x : @N) \multimap A x) \rightarrow (x : @N) \multimap B x (a' x)$	$\simeq$	$(x : @N) \multimap (a : A x) \rightarrow B x a$
	$(a' : (x : @N) \multimap A x) \times ((x : @N) \multimap B x (a' x))$	$\simeq$	$(x : @N) \multimap (a : A x) \times (B x a)$
	$\mathcal{U}$	$\simeq$	$(x : @N) \multimap \mathcal{U}$
$A : \mathcal{U}, a_0, a_1 : @N \multimap A.$	$a_0 \equiv_{@N \multimap A} a_1$	$\simeq$	$(x : @N) \multimap a_0 x \equiv a_1 x$
	$1 + Nm$	$\simeq$	$(x : @N) \multimap Nm$
$A : \mathcal{U}$	A	$\simeq$	$(x : @N) \multimap \text{Gel } A x$
$A : \mathcal{U}, y \neq x$	$\text{Gel } ((x : @N) \multimap A) y$	$\simeq$	$(x : @N) \multimap \text{Gel } A y$
$A : (x : @N) \multimap$ $(y : @N) \multimap \mathcal{U}$	$(y : @N) \multimap (x : @N) \multimap A x y$	$\simeq$	$(x : @N) \multimap (y : @N) \multimap A x y$

The `ext` in the first line vanishes in the second because `EXT $\beta$` triggers. We prove the sufficient condition. Let $(x : @N), (n : Nm)$ in context. We reason by name induction. If $n = cx$ then the right-hand side is `ext [decode $\circ$ encode] $x (cx) \stackrel{\text{EXT}\beta}{=} ((\text{decode} \circ \text{encode}) c) x$. From the proof above we know that encode c = inl tt, and by definition decode (inl tt) = c. So both sides of the equality are equal to cx . Else, formally we must provide a b_1 argument to ind $_{Nm}$. We define $b_1 := b'_1 x$ where the type of b'_1 is the first type in this SAP derivation (performed in the empty context and where $[\dots] := \text{decode} \circ \text{encode}$):`

$$\begin{aligned}
& (x : @N) \multimap (g : \text{Gel } Nm x) \rightarrow \text{forg } x g \equiv \text{ext } [\dots] x (\text{forg } x g) \\
& \simeq (g' : (x : @N) \multimap \text{Gel } Nm x) \rightarrow (x : @N) \multimap \text{forg } x (g' x) \equiv \text{ext } [\dots] x (\text{forg } x (g' x)) \\
& \simeq (n : Nm) \rightarrow (x : @N) \multimap \text{forg } x (\text{gel } n x) \equiv \text{ext } [\dots] x (\text{forg } x (\text{gel } n x))
\end{aligned}$$

By a computation above, `forg  $x (\text{gel } n x) = n$` since x is fresh in n . The right-hand side computes to n as well by `EXT $\beta$` and definition of $[\dots] = \text{decode} \circ \text{encode}$.

Nominal data types. When looking at a specific example of a nominal data type D we temporarily extend the type system with the rules of D . This includes the dependent eliminator of D . For example in the case of the `Ltm` type of Section 1 we add a rule that entails the existence of this closed program:

```

indLtm : (P : Ltm  $\rightarrow$   $\mathcal{U}$ )  $\rightarrow$  ((n : Nm)  $\rightarrow$  P(var n))  $\rightarrow$ 
( $\forall$  a b. P a  $\rightarrow$  P b  $\rightarrow$  P(app a b))  $\rightarrow$ 
( $\forall$  g. ((x : @N)  $\multimap$  P(g x))  $\rightarrow$  P(lam g))  $\rightarrow$   $\forall$  t. P t

```

2.3 The Structure Abstraction Principle (SAP)

For types and terms. As hinted above, PNTT validates the Structure Abstraction Principle (SAP). Similar to the SIP of HoTT/UF, or the SRP of binary PTT [36], the SAP defines how each type former commutes with the (nullary) Bridge type former. Table 1 lists several SAP instances, including the ones we have encountered so far. Each instance is² of the form

² Orienting SAP_K this way enables $\text{SAP}_{\mathcal{U}} = \text{Gel}$ and $\text{SAP}_{\Pi} = \text{ext}$.

$\forall \dots. K' \simeq ((x : @N) \multimap K)$, where K, K' may depend on some parameters. Note that the instances in Table 1 involve primitive type formers K . In order to obtain the SAP instance of a composite type K , we can combine the SAP instances of the primitives used to define K (this was done e.g. in Section 2.2 when proving the propositional η -rule of Nm , and is further discussed below). Types of DTT can contain terms and accordingly there exists a SAP for terms. Let $SAP_K : K' \simeq (@N \multimap K)$ be a SAP instance. The SAP of the term $k : K$ is a pair (k', p) where $k' : K'$ and $p : SAP_K k' \equiv \lambda(\_ : @N).k$, i.e. k' is the reflexivity bridge $rfl\ k := \lambda(\_ : @N).k$ of $k : K$ through the SAP_K equivalence, up to propositional equality.

Relationship to the translation. For types, it is basically the case that the type K' provided by the SAP at K is $[K]_0 : \mathcal{U}$, the recursively and externally defined nullary parametricity translation of K . The n -ary translation is briefly discussed in Section 1. Note that in practice it is sometimes useful to consider SAP instances where K' is a type different but equivalent to $[K]_0$, but this detail can be ignored for clarity. For terms, it is also the case that the SAP at $k : K$ provides a k' which is $[k]_0 : [K]_0$, the recursively defined nullary translation of $k : K$ (analogously it is sometimes useful to let k' be a term different but propositionally equal to $[k]_0$). From that perspective, $SAP_K : [K]_0 \simeq @N \multimap K$ says that the Bridge type of K is its translation up to an equivalence, and the SAP of a term $k : K$ says that $[k]_0 \equiv SAP_K^{-1}(\lambda(\_ : @N).k)$, i.e. the reflexivity bridge at $k : K$ is its translation up to a propositional equality. It is useful to give a name to the latter right-hand side. If a SAP instance is fixed for K we define the *observational parametricity* of a term $k : K$ to be $[k]_0^\mathcal{O} := SAP_K^{-1}(\lambda(\_ : @N).k)$. The various parametricity mappings for types and terms can be informally summarized as follows, where $\overset{m}{\mapsto}$ denotes a metatheoretical function.

$$\begin{aligned} [-]_0 : \mathcal{U} &\overset{m}{\mapsto} \mathcal{U} & ; & \quad [-]_0, [-]_0^\mathcal{O} : \{K : \mathcal{U}\} \overset{m}{\mapsto} K \overset{m}{\mapsto} [K]_0 \\ @N \multimap - : \mathcal{U} &\rightarrow \mathcal{U} & ; & \quad rfl : \{K : \mathcal{U}\} \rightarrow (k : K) \rightarrow @N \multimap K \\ SAP : (K : \mathcal{U}) &\overset{m}{\mapsto} [K]_0 \simeq @N \multimap K & ; & \quad SAP : \{K : \mathcal{U}\} \overset{m}{\mapsto} (k : K) \overset{m}{\mapsto} [k]_0 \equiv [k]_0^\mathcal{O} \end{aligned}$$

As an example, we compute the observational parametricity of a function $f : A \rightarrow B$. Suppose A and B are types with SAP instances $A' \simeq (@N \multimap A)$ and $B' \simeq (@N \multimap B)$. The SAP at $A \rightarrow B$ is the composition $SAP_{A \rightarrow B}^{-1} : (@N \multimap (A \rightarrow B)) \simeq ((@N \multimap A) \rightarrow @N \multimap B) \simeq A' \rightarrow B'$. The map $SAP_{A \rightarrow B}^{-1}$ is given by $\lambda f'. SAP_B^{-1} \circ (\text{ext}^{-1} f') \circ SAP_A$ where $\text{ext}^{-1} f' = \lambda(a' : @N \multimap A). \lambda(x : @N). f' x (a' x)$. With this choice of SAP instances, we have $[f]_0^\mathcal{O} : A' \rightarrow B'$ and $[f]_0^\mathcal{O} = SAP_B^{-1} \circ (@N \multimap f) \circ SAP_A$ where $(@N \multimap f) = \lambda a' x. f(a' x)$ is the action of f on bridges. So the SAP for a function $f : A \rightarrow B$ asserts that its action on bridges agrees with its translation, up to the SAP at A, B .

The SAP of a composite type K is obtained by *manually* applying the SAP rules from Table 1. This process will produce a translation K' that is equivalent to $@N \multimap K$. In the binary case, we have shown in earlier work [36] that this process can be systematized, by organizing types and terms together with their SRP instances into a (shallowly embedded) type theory called ROTT. For types K and terms $k : K$ that fall in the ROTT syntax, the parametricity translations and SAP instances can be derived straightforwardly. Although we believe the same process applies here, we have not constructed the corresponding DSL, instead applying SAP instances manually in examples.

2.4 About Semantics

Similar to our theory PNTT, our model is a simple extension of Cavallo and Harper's presheaf model [10] with the arity of parametricity changed from 2 to 0. This model is $\text{Psh}(\boxtimes_2 \times \boxtimes_0)$, where $\boxtimes_2$ is the binary cartesian cube category [4] for path dimensions, and $\boxtimes_0$ is the 0-ary

affine cube category for bridge dimensions. Concretely $\square_0$ is the opposite of the category of finite ordinals and injections. We keep this section overall succinct, not for lack of formal understanding, because these semantics are a well-understood combination of established models of type theory [13, 15, 4, 8, 10].

In this model the type $\mathbf{Nm}$ is interpreted as the Yoneda embedding of the base object with 1 bridge dimension and no path dimensions. The elimination and computation rules for this type are based on a semantic isomorphism $x : @N \vdash \mathbf{Nm} \cong 1 + \mathbf{Gel Nm} x$ which is straightforwardly checked. In fact, the above (taken as an equivalence) is equivalent (by several invocations of the SAP) to $(@N \multimap \mathbf{Nm}) \simeq 1 + \mathbf{Nm}$, an equivalence semantically proven by Hofmann [14]. Kan fibrancy of $\mathbf{Nm}$ is semantically trivial, since we can tell from the base category that any path in $\mathbf{Nm}$ will be constant. As for computation of Kan operations at $\mathbf{Nm}$, we propose to wait until all arguments reduce to $\mathbf{c}x$ for the same affine name x , in which case we return $\mathbf{c}x$. This is in line with how Kan operations for positive types with multiple constructors are usually reduced [12, 4]. Regarding nominal data types, which we only consider through examples in this paper, we take the viewpoint that these arise from an interplay between the usual type formers, bridge types, $\mathbf{Nm}$, and an initial algebra operation for “nominal strictly positive functors”. We do not attempt to give a categorical description of such functors or prove that they have initial algebras. The Kan operation will reduce recursively and according to the Kan operations of all other type formers involved.

We note that $\square_0$ is the base category (carrier of the site) of the Schanuel sheaf topos [27, §6.3] which is equivalent to the category of nominal sets [27], used to model FreshMLTT [28]. The sheaf condition requires that presheaves $\Gamma : \square_0^{\text{op}} \rightarrow \mathbf{Set}$ preserve pullbacks, i.e. compatible triples in $\Gamma_{U \uplus V} \rightarrow \Gamma_{U \uplus V \uplus W} \leftarrow \Gamma_{U \uplus W}$ have a unique preimage in Γ_U . Conceptually, if a cell $\gamma \in \Gamma_{U \uplus V \uplus W}$ is both fresh for V and for W , then it is fresh for $V \uplus W$, with unique evidence. This property is not available internally in nullary PTT, nor do we have the impression that it is important to add it, so we content ourselves by modeling types as *presheaves* over $\square_0$.

3 Nominal Primitives for Free

In this section, we argue that the central features in a number of earlier nominal dependent type systems can essentially be recovered in nullary PTT. One notable exception is nominal pattern matching (discussed in the next section), which is recovered in our extension PNTT. Concretely, we consider here Shinwell, Pitts and Gabbay’s FreshML [33], Schöpp and Stark’s bunched nominal type theory BNTT [32, 31], Cheney’s λ^{IM} [11] and Pitts, Matthiesen and Derikx’s FreshMLTT [28]. The central features we identify there, are: existential and universal name quantification, a type former expressing freshness for a given name, name swapping, and locally scoped names. Existential and universal name quantification are known to be equivalent in the usual (pre)sheaf or nominal set semantics of nominal type theory, but generally have quite different typing rules: the former is an existential type former with pair-like constructor and matching eliminator (opening the door to matching more deeply), whereas the latter is a universal type operated through name abstraction and application (getting in the way of matching more deeply). We note that some systems (FreshMLTT, λ^{IM}) support multiple name types, something we could also easily accommodate, but leave out so as not to distract from the main contributions. BNTT even allows substructural quantification and typal freshness for arbitrary closed types (rather than just names), which is not something we intend to support and which inherently seems to require a bunched context structure. In what follows, we will speak of “our rules”, not to claim ownership (as they are inherited from Bernardy, Coquand and Moulin [8] and Cavallo and Harper [10]), but to distinguish them from the other systems.

Universal name quantification. Universal name quantification is available in BNTT, λ^{III} and FreshMLTT. In our system, it is done using the nullary bridge type $(x : @N) \multimap A$, whose rules are given in Figure 2. The rules $\multimap F$ and $\multimap I$ correspond almost perfectly with the other three systems; for BNTT we need to keep in mind that our context extension with a fresh name, is semantically a monoidal product. The application rules in λ^{III} and BNTT also correspond almost precisely to $\multimap E$, but the one in BNTT is less algorithmic than ours. Specifically, the BNTT bunched application rule takes a function $\Theta \vdash f : (x : T) \multimap Ax$ and an argument $\Delta \vdash t : T$ and produces $ft : At$ in a non-general context $\Theta * \Delta$. Our rule $\multimap E$ (inherited from [8, 10]) improves upon this by taking in an arbitrary context Γ and *computing* a context $\Gamma \setminus x$ such that there is a morphism $\Gamma \rightarrow (\Gamma \setminus x, (x : @N))$. The application rule in FreshMLTT is similar, but uses definitional freshness (based on a variable swapping test) to ensure that the argument is fresh for the function.

Typal freshness. A type former expressing freshness is available in BNTT (called the free-from type). FreshMLTT uses definitional freshness instead. In nullary PTT, elements of A for which x is fresh are classified by the type $\text{Gel } Ax$. BNTT's free-from types only apply to closed types A , so GELF is more general. BNTT's introduction rule corresponds to GELI, which is however again more algorithmic. BNTT's elimination rule is explained in terms of single-hole bunched contexts [31, §4.1.1] which specialize in our setting (where the only monoidal product is context extension with a name) to contexts with a hole up front. Essentially then, the BNTT rule can be phrased in nullary PTT as

$$\frac{\Gamma, x : @N, z : \text{Gel } Bx, \Theta \vdash T \text{ type} \quad \Gamma, y : B, x : @N, \Theta[\text{gel } yx/z] \vdash t : T[\text{gel } yx/z]}{\Gamma, x : @N, z : \text{Gel } Bx, \Theta \vdash t' : T}$$

where Γ must be empty, together with β - and η -rules establishing that the above operation is inverse to applying the substitution $[\text{gel } yx/z]$.³ We can in fact accommodate the rule for non-empty Γ . Without loss of generality, we can assume Θ is empty: by abstraction/application, we can subsume Θ in T .⁴ We then have an equivalence

$$\begin{aligned} (y : B) \rightarrow ((x : @N) \multimap T[\text{gel } yx/z]) &\simeq (y : (x : @N) \multimap \text{Gel } Bx) \rightarrow ((x : @N) \multimap T[yx/z]) \\ &\simeq (x : @N) \multimap (z : \text{Gel } Bx) \rightarrow T \end{aligned}$$

where in the first step, we precompose with the *gel/ung* isomorphism (the SAP for *Gel*), and in the second step, we apply the *ext* equivalence (the SAP for functions).

Existential name quantification. Existential name quantification is available in FreshML and BNTT. We first discuss how we can accommodate the BNTT rules, and then get back to FreshML. The BNTT existential quantifier is just translated to the nullary bridge type $(x : @N) \multimap A$ again, i.e. both quantifiers become definitionally the same type in nullary PTT. BNTT's introduction rule follows by applying a function $\text{bind} : (x : @N) \multimap Bx \rightarrow \text{Gel}((w : @N) \multimap Bw)x$ which is obtained from the identity function on $(w : @N) \multimap Bw$ by the SAP for functions and *Gel*. BNTT's elimination rule essentially provides a function $\text{matchbind} : ((x : @N) \multimap Bx \rightarrow \text{Gel } Cx) \rightarrow (((x : @N) \multimap Bx) \rightarrow C)$ such that if we apply $\text{matchbind } f$ under *Gel* to $\text{bind } x b$, then we obtain fxb . Again, the SAP for *Gel* and functions reveals that the source and target of *matchbind* are equivalent.

³ The original rule immediately subsumes a substitution $\Delta \rightarrow (x : @N, z : \text{Gel } Ax)$.

⁴ This may raise questions about preservation of substitution w.r.t. Θ . However, it is not at all standard for dependently typed elimination rules to guarantee preservation of substitution w.r.t. a telescope depending on the eliminee.

The non-dependently typed system FreshML has a similar type former, but lacks any typal or definitional notion of freshness. Their introduction rule then follows by postcomposing the above `bind` with the function `forg` that forgets freshness (Section 2.1). If we translate FreshML's declarations $\Gamma \vdash d : \Delta$ to operations that convert terms $\Gamma, \Delta \vdash t : T$ to terms $\Gamma \vdash t\{d\}$ (not necessarily by substitution), then we can translate their declaration $\Gamma \vdash \text{val } \langle x \rangle y = e : (x : @N, y : B)$, where e is an existential pair, as `matchdecl` where `matchdecl` : $(e : @N \multimap B) \rightarrow (t : @N \multimap B \rightarrow T) \rightarrow (@N \multimap T)$. This function is obtained by observing that the second argument type, by the SAP for functions, is equivalent to $(@N \multimap B) \rightarrow (@N \multimap T)$. It may be surprising that the result has type $@N \multimap T$ rather than just T ; this reflects the fact that FreshML cannot enforce freshness, and is justified by the fact that it allows arbitrary declaration of fresh names via the declaration $\Gamma \vdash \text{fresh } x : (x : @N)$, which we do not support.

Swapping names. The name swapping operation is available in FreshML and FreshMLTT. We can accommodate the full rule of FreshML (which is not dependently typed) and a restricted version of the rule in FreshMLTT, where we allow the type of the affected term to depend on the names being swapped, but other than that, only on variables fresh for those names. We simply use the function `swap` : $(xy : @N) \multimap Txy \rightarrow Tyx$ whose type is equivalent to $((xy : @N) \multimap Txy) \rightarrow ((xy : @N) \multimap Tyx)$ by the SAP, and the latter is clearly inhabited by $\lambda txy. tyx$. Note that name swapping needs to happen w.r.t. affine names: swapping the cartesian names $x, y : Nm$ in the term `app (app (var x) (var y)) (var z) : Ltm` does not commute with contraction $-[x/x, x/z]$. Finally, we note that the aforementioned restriction on the type can be mitigated in an ad hoc manner, because types $T : (xy : @N) \multimap \Delta \rightarrow \mathcal{U}$ (where Δ denotes any telescope consisting of both affine names and non-affine variables) are by the SAP in correspondence with types $\Delta'' \rightarrow (xy : @N) \multimap \mathcal{U}$ for a different telescope Δ'' . This however does not imply that we can accommodate the full FreshMLTT name swapping rule in a manner that commutes with substitution. We expect that this situation can be improved by integrating ideas related to the transpension type [21, 19] into the current system. Lastly, using `swap`, we can follow Pitts et al. [28] in defining non-binding abstraction $\langle x \rangle - = \lambda a y. \text{swap } x y a : Ax \rightarrow (y : @N) \multimap Ay$, where A can depend only on variables fresh for x .

Locally scoped names. Locally scoped names [23, 26] are available in FreshMLTT, by the following rule on the left, and allow us to spawn a name from nowhere, provided that we use it to form a term that is fresh for it:

$$\frac{\begin{array}{c} \Gamma, x : @N \vdash T \text{ type} \\ \Gamma, x : @N \vdash t : T \\ \hline x \text{ is fresh for } t : T \\ \Gamma \vdash \nu x. t : \nu x. T \end{array}}{\Gamma \vdash \nu x. t : \nu x. T} \quad \frac{\begin{array}{c} \Gamma, x : @N \vdash \nu x. t = t : T \\ \text{Added: } \nu x. t = t \text{ if } x \text{ is not free in } t. \end{array}}{\Gamma \vdash \nu x. t : \nu x. T} \quad \frac{\begin{array}{c} \Gamma \vdash S \text{ type} \\ \Gamma, x : @N \vdash t : S \\ \hline x \text{ is fresh for } t : S \\ \Gamma \vdash \nu x. t : S \end{array}}{\Gamma \vdash \nu x. t : S}$$

It is used [24] in FreshMLTT to define e.g.

$$\lambda c'. \nu x. \text{case } (c' x) \left\{ \begin{array}{l} \text{inl } a \mapsto \text{inl } \langle x \rangle a \\ \text{inr } b \mapsto \text{inr } \langle x \rangle b \end{array} \right. : (@N \multimap A + B) \rightarrow (@N \multimap A) + (@N \multimap B)$$

It has a computation rule which says that as soon as x comes into scope again, we can drop the ν -binder. Combined with α -renaming, this means $\nu x.$ says “let x be any name we have in scope, or a fresh one, it doesn't matter”. In particular, $\nu x.-$ is idempotent. Before translating to nullary PTT, we add another equation rule, which says that we can drop $\nu x.-$ if x is not used freely at all (in the example above, x is used freely but freshly). This way, the

FreshMLTT ν -rule above becomes equivalent to the one to its right. Indeed, the functions $T \mapsto \nu x. T$ and $S \mapsto S$ now constitute an isomorphism between types that are and are not dependent on $x : @N$. The advantage of the rule on the right is that it is not self-dependent. In nullary PTT, we express freshness using Gel, suggesting the following adapted rules:

$$\frac{\Gamma, x : @N \vdash t : \text{Gel } S x}{\Gamma \vdash \nu x. t : S} \quad \Gamma, x : @N \vdash \text{gel}(\nu x. t) x = t : \text{Gel } S x$$

$\nu x. \text{gel } t x = t$ (where x cannot be free in t by GELI).

In this formulation, it is now clear that $\nu x. t$ can be implemented as $\text{ung}(\lambda x. t)$, while the two computation rules follow from $\text{GEL}\eta$ and $\text{GEL}\beta$.

4 Nominal Pattern Matching

In this section we provide concrete examples of functions $D \rightarrow E$ defined by recursion on a nominal data type D , within PNTT, explained in Section 2.

4.1 Patterns That Bind

Some nominal frameworks with existential name-abstraction types [33] provide a convenient user interface to define functions $f : D \rightarrow E$ out of a nominal data type. The user can define f by matching on patterns that bind names (e.g. `eqabs` in Section 1). We explain how this feature is indirectly recovered in PNTT.

The following nominal data type [7] is the nominal syntax of the π -calculus [16], a formal language whose expressions represent concurrently communicating processes. The constructors stand for: terminate, silent computation step, parallelism, non-determinism, channel allocation, receiving⁵, and sending. The arguments of (output) type `Proc` can be thought of as continuations, e.g. `nu` prepends a channel allocation instruction.

```
data Proc :  $\mathcal{U}$  where
  nil : Proc
   $\tau\text{pre}$  : Proc  $\rightarrow$  Proc
  par, sum : Proc  $\rightarrow$  Proc  $\rightarrow$  Proc
```

```
nu : ( $@N \multimap$  Proc)  $\rightarrow$  Proc
inp : Nm  $\rightarrow$  ( $@N \multimap$  Proc)  $\rightarrow$  Proc
out : Nm  $\rightarrow$  Nm  $\rightarrow$  Proc  $\rightarrow$  Proc
```

An example of a process is `par (out a b q) (inp a ( $\lambda(x : @N). p' x$ ))`. The first argument of `par` emits a name b on channel a and continues with q . Simultaneously, the second argument waits for a name x on channel a and continues with $p' x$. The expectation is that such an expression should reduce to `par q ( $p' \{b/x\}$ )` where $p' \{b/x\}$ replaces occurrences of $(x : @N)$ in the body of p' by the name $b : Nm$. Note that the application $p' b$ does not typecheck and that this substitution operation called `nsub` below must be defined recursively, as done in [7]. The following is an informal definition of `nsub` where some patterns bind (bridge) variables.

```
nsub : Nm  $\rightarrow$  ( $@N \multimap$  Proc)  $\rightarrow$  Proc
nsub b ( $\lambda x. \text{par } (u' x) (v' x)$ ) = par (nsub b u') (nsub b v')
... --nil,  $\tau\text{pre}$ , sum similar
nsub b ( $\lambda x. \text{nu } (q' x)$ ) = nu ( $\lambda y. \text{nsub } b (\lambda x. q' x y)$ )
nsub b ( $\lambda x. \text{inp } a \ (q' x)$ ) = inp a ( $\lambda y. \text{nsub } b (\lambda x. q' x y)$ ) --( $\emptyset$ )
nsub b ( $\lambda x. \text{inp } (c x) \ (q' x)$ ) = inp b ( $\lambda y. \text{nsub } b (\lambda x. q' x y)$ ) --(1)
...
```

⁵ Binders have arguments $@N \multimap -$ and not $Nm \rightarrow -$ since the latter could lead to exotic process terms checking the bound name for equality to other names.

Note how patterns (0) and (1) match on a binding pattern of the form $\lambda x. \text{inp } m \ (q' \ x)$, and cover the case where m is different, resp. equal to the variable being substituted. The informal `nsub` function reduces accordingly.

More formally in our system we define `nsub` by using the SAP at `Proc`, so `nsub b` is the following composition $(@N \multimap \text{Proc}) \xrightarrow{\sim} \text{AProc} \longrightarrow \text{Proc}$. The SAP at `Proc` asserts that $(@N \multimap \text{Proc})$ is equivalent to `AProc`, the nullary translation of `Proc` (its constructors use a suggestive $[-]_0$ notation):

<pre>data AProc : $\mathcal{U}$ where [nil]₀ : AProc [τpre]₀ : AProc → AProc [par]₀ , [sum]₀ : AProc → AProc → AProc</pre>	<pre>[nu]₀ : (@N $\multimap$ AProc) → AProc [inp]₀ : 1 + Nm → (@N $\multimap$ AProc) → AProc [out]₀ : 1 + Nm → 1 + Nm → AProc → AProc</pre>

We can then define `nsub'` : $Nm \rightarrow \text{AProc} \rightarrow \text{Proc}$ by induction on the second argument. Indeed the informal clauses (0), (1) can be translated into formal ones using the $[\text{inp}]_0$ constructor. More precisely, clause (1) uses the function $[\text{inp}]_0 (\text{inl } tt)$ of type $(@N \multimap \text{AProc}) \rightarrow \text{AProc}$ and clause (0) uses the function $\lambda(n : Nm). [\text{inp}]_0 (\text{inr } n)$ of type $Nm \rightarrow (@N \multimap \text{AProc}) \rightarrow \text{AProc}$.

4.2 A HOAS Example

In the example sketched below, we connect the nominal syntax of the untyped lambda calculus (ULC) to a nominal, higher-order abstract syntax (HOAS) representation. The example is performed within PNTT plus a single (standard) binary parametricity axiom. Our proof is a port to PNTT of an existing argument by Atkey [5], which uses external *Kripke* binary parametricity to connect the De Bruijn (non-nominal) syntax of ULC to a (non-nominal) HOAS representation.

This example serves two purposes. The first is to illustrate, by porting a complex argument from one setting to the other, how the combination of nullary (for nominal reasoning) and binary parametricity allows for a reasoning style similar to that of Kripke binary parametricity, and moreover with closely related denotational semantics. This topic is elaborated further at the end of this section.

Secondly, the example illustrates that more often than not, proving the correctness of a function $f : D \rightarrow E$ defined by recursion out of a nominal data type D requires computing the parametricity translation of f . Indeed, for $\text{bind} : (@N \multimap D) \rightarrow D$ a binder (constructor) of D , the β -rule of D says that the redex $f(\text{bind } d')$ computes to a term involving $@N \multimap f$, the action of f on bridges (see Section 2.3). So proving f correct often requires characterizing $@N \multimap f$ up to propositional equality. This characterization is the SAP for f (also in Section 2.3) $[f]_0 \equiv_{[D]_0 \rightarrow [E]_0} [f]_0^O$, where $[f]_0$ is the recursive translation of f and $[f]_0^O := \text{SAP}_E^{-1} \circ (@N \multimap f) \circ \text{SAP}_D$ the observational parametricity of f .

SAP proofs for composite terms like f are unwieldy because one has to show why the SAP instances for the primitives in f compose to the global one and this involves manually reproducing the structure of the term f at the level of SAP proofs. For this reason some instances of the SAP for terms are assumed to hold below, without proof (see paragraph “Roundtrip at Ltm_j ”). The unwieldiness of interval-based PTTs is a good selling point for *observational* parametric type theories (see Section 5), where the SAP is not needed since a first-class recursive translation attempts to replace $@N \multimap -$ altogether.

As a side note regarding the above function $f : D \rightarrow E$, its domain D could feature a binder of type $(@N \multimap \dots \multimap @N \multimap D) \rightarrow D$ binding m fresh names. In that case proving the correctness of f may involve computing its m -fold iterated nullary parametricity, for the same reasons as above. No such examples are investigated here.

Let us now get to the actual argument. The nominal syntax of ULC is expressed as the following data type family Ltm , parametrized by a natural number $j : \text{nat}$.

```
data Ltm (j : nat) :  $\mathcal{U}$ 
  env : Fin j  $\rightarrow$  Ltm j
  var : Nm  $\rightarrow$  Ltm j
  app : Ltm j  $\rightarrow$  Ltm j  $\rightarrow$  Ltm j
  lam : (@N  $\multimap$  Ltm j)  $\rightarrow$  Ltm j
```

The type $\text{Fin } j$ is the finite type with j elements $\{0, \dots, j-1\}$. Ltm_j is a shorthand for $\text{Ltm } j$. The types Ltm and Ltm_1 of Section 1 are Ltm_0 and Ltm_1 respectively.

The corresponding HOAS representation, or encoding, is $\text{HEnc}_j : \mathcal{U}$ defined below. Since it uses a Π -type we say it is a Π -encoding.

```
HMod (j : nat) = (H :  $\mathcal{U}$ )  $\times$  (Fin j  $\rightarrow$  H)  $\times$  (Nm  $\rightarrow$  H)  $\times$ 
  (H  $\rightarrow$  H  $\rightarrow$  H)  $\times$  ((H  $\rightarrow$  H)  $\rightarrow$  H)

--projections
|_| :  $\forall \{j\}$ . HMod j  $\rightarrow$   $\mathcal{U}$ 
|_| M = M .fst
envOf, varOf, appOf, hlamOf = ... --other projections of HMod

HEnc (j : nat) = (M : HMod j)  $\rightarrow$  |M|

NMod (j : nat) = (H :  $\mathcal{U}$ )  $\times$  (Fin j  $\rightarrow$  H)  $\times$  (Nm  $\rightarrow$  H)  $\times$ 
  (H  $\rightarrow$  H  $\rightarrow$  H)  $\times$  ((@N  $\multimap$  H)  $\rightarrow$  H)
```

The type of “nominal models” NMod_j is also defined as it will be useful later on. The carrier function $|_|$ and other projections are defined similarly for nominal models. Additionally we define explicit constructors $\text{mkHM}_j, \text{mkNM}_j$ for $\text{HMod}_j, \text{NMod}_j$. For instance $\text{mkNM}_j : (H : \mathcal{U}) \rightarrow (\text{Fin } j \rightarrow H) \rightarrow (\text{Nm} \rightarrow H) \rightarrow (H \rightarrow H \rightarrow H) \rightarrow ((@N \multimap H) \rightarrow H) \rightarrow \text{NMod}_j$.

We will show that we can define maps between Ltm_j and the encoding HEnc_j in both ways, and sketch a conditional proof that they compose to the identity at Ltm_j .

```
ubd :  $\forall \{j\}$ . HEncj  $\rightarrow$  Ltmj
toh :  $\forall \{j\}$ . Ltmj  $\rightarrow$  HEncj
rdt-Ltm :  $\forall \{j\}$ . (t : Ltmj)  $\rightarrow$  ubd(tohj t)  $\equiv$  t
```

Unembedding. We begin by defining the “unembedding” map denoted by ubd , which has a straightforward definition that does not involve nominal recursion. The name and the idea behind the definition come from [5, 6], where Atkey et al. were interested in comparing (non-nominal) syntax and HOAS Π -encodings using a strengthened form of binary parametricity called Kripke parametricity (see end of section for a comparison with our system/model).

```
ubd {j} =  $\lambda(h : \text{HEnc}_j)$ . h (LtmHMod j) where
  LtmHMod :  $\forall j$ . HModj
  LtmHMod j = mkHMj Ltmj env var app (hLamLtm j)
  hLamLtm :  $\forall j$ . (Ltmj  $\rightarrow$  Ltmj)  $\rightarrow$  Ltmj
  hLamLtm j f = lam(  $\lambda(x : @N)$ . f(var (c x)) )
```

So unembedding a HOAS term h consists of applying h at Ltm_j . This is possible thanks to the fact that Ltm_j can be equipped with a higher-order operation hLamLtm_j .

12:18 Nominal Type Theory by Nullary Internal Parametricity

Defining the map into HOAS. The other map toh_j is defined by nominal recursion. In other words it is defined using the eliminator of Ltm_j . We write the (uncurried) non-dependent eliminator as rec_j . Its type expresses that Ltm_j is the initial nominal model.

```
recj : (N : NModj) → Ltmj → |N|
```

Hence in order to define toh_j we need to turn its codomain HEnc_j into a nominal model. For fields in NMod_j that are not binders this is straightforward.

<pre>envH : Fin j → HEnc_j varH : Nm → HEnc_j appH : HEnc_j → HEnc_j → HEnc_j</pre>	<pre>envH k = λ (M : HMod_j). envOf M k varH n = λ (M : HMod_j). varOf M n appH u v = λ M. appOf M (u M) (v M)</pre>

Additionally we need to provide a function $\text{lamH} : (@N \multimap \text{HEnc}_j) \rightarrow \text{HEnc}_j$. This is done by using the SAP at HEnc_j , i.e. the following characterization of $(@N \multimap \text{HEnc}_j)$.

► **Theorem 1.** *We have $\text{ebump}_j : (@N \multimap \text{HEnc}_j) \simeq \text{HEnc}_{j+1}$, $\text{mbump}_j : (@N \multimap \text{HMod}_j) \simeq \text{HMod}_{j+1}$, $\text{nbump}_j : (@N \multimap \text{NMod}_j) \simeq \text{NMod}_{j+1}$ and $\text{lbump}_j : (@N \multimap \text{Ltm}_j) \simeq \text{Ltm}_{j+1}$.*

Intuitively these results hold because $@N \multimap -$ preserves all primitive type formers except Nm , which gets translated to a sum $1 + \text{Nm}$. So translating each of the aforementioned types boils down to replacing Nm with $1 + \text{Nm}$, which refactors to incrementing j .

Proof. We only prove mbump_j . The proofs of ebump_j and nbump_j are similar, and lbump_j is proved using an encode-decode argument [10, 36] similar to the proof of SAP_{Nm} . We sometimes omit types for Σ and $\multimap$, e.g. we write $x \multimap T$ as shorthand for $(x : @N) \multimap T$.

$$\begin{aligned}
 x \multimap \text{HMod}_j &\simeq (H' : x \multimap \mathcal{U}) \times (x \multimap [(Fin\ j \rightarrow H'x) \times \dots]) && \text{SAP}_{\Sigma} \\
 &\simeq (H' : x \multimap \mathcal{U}) \times (x \multimap (Fin\ j \rightarrow H'x)) \times (x \multimap [\dots]) && \text{SAP}_{\Sigma} \\
 &\simeq (H' : x \multimap \mathcal{U}) \times (\text{env}' : Fin\ j \rightarrow (x \multimap H'x)) \times (x \multimap [\dots]) && \text{SAP}_{Fin\ j, \rightarrow}
 \end{aligned}$$

Since $Fin\ j$ is a non-nominal data type its SAP instance asserts $Fin\ j \simeq (x \multimap Fin\ j)$, i.e. the only bridges in $Fin\ j$ are reflexive bridges [10, 36]. Moving on,

$$\begin{aligned}
 &\simeq H' \times \text{env}' \times (x \multimap [(Nm \rightarrow H'x) \times \dots]) \\
 &\simeq H' \times \text{env}' \times (x \multimap (Nm \rightarrow H'x)) \times (x \multimap [\dots]) && \text{SAP}_{\Sigma} \\
 &\simeq H' \times \text{env}' \times ((x \multimap Nm) \rightarrow (x \multimap H'x)) \times (x \multimap [\dots]) && \text{SAP}_{\rightarrow} \\
 &\simeq H' \times \text{env}' \times (\text{foo} : (1 + Nm) \rightarrow (x \multimap H'x)) \times (x \multimap [\dots]) && \text{SAP}_{\text{Nm}} \\
 &\simeq H' \times \text{env}'_+ \times (\text{var}' : Nm \rightarrow x \multimap H'x) \times (x \multimap [\dots])
 \end{aligned}$$

where $\text{env}'_+ : Fin\ (j+1) \rightarrow (x \multimap H'x)$ is defined as $\text{env}'_+ 0 = \text{foo}(\text{inl}\ tt)$ and $\text{env}'_+(k+1) = \text{env}'k$. The next two types in $x \multimap [\dots]$ are computed using the SAP at $\rightarrow$. Thus so far we have shown that $@N \multimap \text{HMod}_j$ is equivalent to $(H' : x \multimap \mathcal{U}) \times \text{env}'_+ \times \text{var}' \times (\text{app}' : (x \multimap H'x) \rightarrow (x \multimap H'x) \rightarrow (x \multimap H'x)) \times (\text{hlam}' : ((x \multimap H'x) \rightarrow (x \multimap H'x)) \rightarrow (x \multimap H'x))$. Now since the SAP at $\mathcal{U}$ is $\text{Gel} : \mathcal{U} \xrightarrow{\sim} (@N \multimap \mathcal{U})$, we can do a change of variable in the above translation of HMod_j , i.e. use a variable $(K : \mathcal{U})$ instead of $(H' : x \multimap \mathcal{U})$ at the cost of replacing occurrences of H' by $\text{Gel}\ K$. Pleasantly, occurrences of $(x \multimap H'x)$ become $(x \multimap \text{Gel}\ K\ x)$, which is equivalent to K by the SAP for Gel types. Hence $@N \multimap \text{HMod}_j \simeq \text{HMod}_{j+1}$. ◀

Next, we can define the desired operation $\text{lamH} : (@N \multimap \text{HEnc}_j) \rightarrow \text{HEnc}_j$ as $\text{lamH } h' = \lambda(M : \text{HMod}_j). \text{lamOf } M (\lambda(m : |M|). (\text{ebump}_j h') (m :: M))$ where $(m :: M)$ is the HMod_{j+1} obtained out of $M : \text{HMod}_j$ by pushing m onto the list $\text{envOf } M$. In other words, $\text{envOf } (m :: M) 0 = m$ and $\text{envOf } (m :: M) (k+1) = \text{envOf } M k$. All in all we proved that HEnc_j is a nominal model. Since Ltm_j is the initial one we obtain the desired map $\text{toh}_j : \text{Ltm}_j \rightarrow \text{HEnc}_j$.

```
toh_j = rec_j (mkNM_j Henc_j envH varH appH
  (\lambda h' M. lamOf M (\lambda(m:|M|). ebump_j h' (m::M))))
```

Roundtrip at Ltm_j . We sketch a conditional proof that $\forall j (t : \text{Ltm}_j). t \equiv \text{ubd}_j (\text{toh}_j t)$, by (nominal) induction, i.e. using the dependent eliminator of Ltm_j . Our proof is conditional since it relies on two axioms. (1) We use a specific axiom called *binx* that is an instance of standard binary parametricity. It is natural to rely on such an axiom here since our proofs attempt to emulate Kripke *binary* parametricity (the Kripke aspect is in fact emulated by nullary parametricity). An expected model for this standard binary parametricity statement is $\text{Psh}(\Box_2 \times \Box_0 \times \Box_2)$ where $\Box_2$ is the original binary affine cube category used by Cavallo and Harper (CH) [10] to model binary parametricity, see Section 2.4. In fact, we expect *binx* to be provable in PNTT extended with CH internal binary parametricity operators. (2) Some steps require characterizing the observational parametricity $[-]_0^{\mathcal{O}}$ (defined in Section 2.3) of certain terms. These steps are written with a gray background below and are *assumed*. As mentioned above characterizing the observational parametricity $[f]_0^{\mathcal{O}}$ of a complex term f is unwieldy because such characterizations must prove that $[-]_0^{\mathcal{O}}$ commutes with the primitives used in f and reproduce by hand the structure of the term f . We currently have a partial paper proof of these steps. In the future we would like to implement our theory (or maybe an observational version of it) to fully validate this example.

The proofs for constructors other than lam_j are easy. For $t = \text{lam}_j (g : @N \multimap \text{Ltm}_j)$ we must prove that if the induction hypothesis holds ($g^\bullet : (z : @N) \multimap (gz \equiv \text{ubd}_j (\text{toh}_j (gz)))$) then $\text{lam}_j g \equiv \text{ubd}_j (\text{toh}_j (\text{lam}_j g))$. Let $g^\bullet$ be in context and let us compute the right-hand side *RHS* of the latter equation. We use the $\$$ application operator found e.g. in Haskell. This operator is defined as function application but associates to the right, improving clarity.

$$\begin{aligned}
RHS &\equiv \text{ubd}_j \$ \lambda M. \text{lamOf } M \$ \lambda m. (\text{ebump}_j \circ (@N \multimap \text{toh}_j)) g (m :: M) && (\text{def. toh}_j) \\
&\equiv \text{hLamLtm}_j \$ \lambda m. (\text{ebump}_j \circ (@N \multimap \text{toh}_j)) g (m :: \text{LtmHMod}_j) && (\text{def. ubd}_j) \\
&\equiv \text{hLamLtm}_j \$ \lambda m. (\text{toh}_{j+1} \circ \text{lbump}_j) g (m :: \text{LtmHMod}_j) && ([\text{toh}_j]_0^{\mathcal{O}} \equiv \text{toh}_{j+1}) \\
&\equiv \text{lam}_j \$ \lambda (x : @N). (\text{toh}_{j+1} \circ \text{lbump}_j) g (\text{var}(cx) :: \text{LtmHMod}_j) && (\text{def. hLamLtm}_j) \\
&\equiv \text{lam}_j \$ \lambda (x : @N). (\text{toh}_{j+1} \circ \text{lbump}_j) g \$ \\
&\quad \text{mkHM}_{j+1} \text{Ltm}_j (\text{var}(cx) :: \text{env}_j) \text{var}_j \text{app}_j \text{hLamLtm}_j && (\text{def. } ::)
\end{aligned}$$

The latter model is of type HMod_{j+1} and we observe that it looks similar to $\text{LtmHMod}_{j+1} = \text{mkHM}_{j+1} \text{Ltm}_{j+1} \text{env}_{j+1} \text{var}_{j+1} \text{app}_{j+1} \text{hLamLtm}_{j+1} : \text{HMod}_{j+1}$. More formally, for $(x : @N)$ in context, the graph of $\text{plug}_x : \text{Ltm}_{j+1} \rightarrow \text{Ltm}_j : u \mapsto \text{lbump}_j^{-1} u x$ turns out to be a structure-preserving relation between the two models $\text{LtmHMod}_{j+1}, (\text{var}(cx) :: \text{LtmHMod}_j) : \text{HMod}_{j+1}$. This suggests using the binary parametricity of the dependent function $\text{inEnc}_g := (\text{toh}_{j+1} \circ \text{lbump}_j) g : (M : \text{HMod}_{j+1}) \rightarrow M$ which appears in our computed *RHS*. The binary parametricity of inEnc_g is $[\text{inEnc}_g]_2 : \forall (M_0 M_1 : \text{HMod}_{j+1}) (R : [\text{HMod}_{j+1}]_2 M_0 M_1). |R|(\text{inEnc}_g M_0)(\text{inEnc}_g M_1)$, where $[\text{HMod}_{j+1}]_2 M_0 M_1$ is defined to be the type of structure-preserving relations between M_0, M_1 , and $|R| : |M_0| \rightarrow |M_1| \rightarrow \mathcal{U}$ extracts the carrier of R . The specific binary parametricity axiom we use is $\text{binx} := [\text{inEnc}_g]_2 \text{LtmHMod}_{j+1} (\text{var}(cx) :: \text{LtmHMod}_j) (\text{Graph}(\text{plug}_x))$, i.e. the dependent function inEnc_g commutes with the

function plug_x . We then have:

$$\begin{aligned}
RHS &\equiv \text{lam}_j \$ \lambda(x : @N). (\text{toh}_{j+1} \circ \text{lbump}_j) g (\text{var}(cx) :: \text{LtmHMod}_j) \\
&\equiv \text{lam}_j \$ \lambda(x : @N). \text{plug}_x \$ (\text{toh}_{j+1} \circ \text{lbump}_j) g \text{LtmHMod}_{j+1} && (\text{binx}) \\
&\equiv \text{lam}_j \$ \lambda(x : @N). \text{plug}_x \$ (\text{ubd}_{j+1} \circ \text{toh}_{j+1})(\text{lbump}_j g) && (\text{def. ubd}_{j+1}) \\
&\equiv \text{lam}_j \$ \lambda(x : @N). (\text{lbump}_j^{-1} \$ (\text{ubd}_{j+1} \circ \text{toh}_{j+1})(\text{lbump}_j g)) x && (\text{def. plug}_x)
\end{aligned}$$

Now it remains to show that the latter $\equiv \text{lam}_j \$ \lambda(x : @N). gx$, i.e. that the following equality holds $\text{eqNextWorld} : \text{lbump}_j g \equiv (\text{ubd}_{j+1} \circ \text{toh}_{j+1})(\text{lbump}_j g)$. Let us write $r_j := \text{ubd}_j \circ \text{toh}_j$. (1) The induction hypothesis tells us $g^\bullet : (z : @N) \multimap (gz \equiv r_j(gz))$ so by the SAP at $\equiv$ we get $g \equiv \lambda(z : @N). r_j(gz)$ and by applying lbump_j we get $\text{lbump}_j g \equiv \text{lbump}_j (\lambda(z : @N). r_j(gz))$. (2) The observational parametricity of r_j is such that $[r_j]_0^\mathcal{O} \equiv [\text{ubd}_j \circ \text{toh}_j]_0^\mathcal{O} \equiv [\text{ubd}_j]_0^\mathcal{O} \circ [\text{toh}_j]_0^\mathcal{O} \equiv \text{ubd}_{j+1} \circ \text{toh}_{j+1} = r_{j+1}$. The function r_j has type $\text{Ltm}_j \rightarrow \text{Ltm}_j$ so $[r_j]_0^\mathcal{O} \equiv r_{j+1}$ entails (by def. of $[r_j]_0^\mathcal{O}$, see Section 2.3) $r_{j+1} \circ \text{lbump}_j \equiv \text{lbump}_j \circ (@N \multimap r_j)$. Applying both sides to $g : @N \multimap \text{Ltm}_j$ we get $r_{j+1}(\text{lbump}_j g) \equiv \text{lbump}_j(\lambda(z : @N). r_j(gz))$. (3) Using step (1), (2) and transitivity of $\equiv$ we get $\text{lbump}_j g \equiv r_{j+1}(\text{lbump}_j g) = \text{ubd}_{j+1} (\text{toh}_{j+1} (\text{lbump}_j g))$. Hence eqNextWorld holds and this concludes our proof.

Synthetic Kripke parametricity. Kripke parametricity is a strengthened form of binary parametricity that was introduced in a denotational model built by Atkey [5] in order to prove an adequacy result for HOAS: the De Bruijn syntax of ULC is equivalent to the HOAS Π -encoding $\forall A.(A \rightarrow A \rightarrow A) \rightarrow ((A \rightarrow A) \rightarrow A) \rightarrow A$. One way to explain Kripke binary parametricity is to compare it to standard binary parametricity. The latter asserts that polymorphic functions $f : (A : \mathcal{U}) \rightarrow T(A)$ map relations R to proofs of relatedness over R (this is explained for $T(A) = A \rightarrow A$ in Section 1). By contrast, briefly speaking, the *Kripke* binary parametricity of a polymorphic function $f : (A : \mathcal{U}) \rightarrow T(A)$ is written $[f]_{\text{K2}}$ and asserts that for every $(*)$ preorder W , f maps any W -monotonic family of relations $(w \mapsto R_w) : W \rightarrow A_0 \rightarrow A_1 \rightarrow \mathcal{U}$ to a monotonic function/proof that for every $w : W$ the outputs $f A_0, f A_1$ are related over R_w ($\mathcal{U}$ is implicitly ordered by logical implication).

We think that the example discussed above externalizes to the fragment of Atkey’s Kripke model where the W preorder is systematically replaced by $(\mathbb{N}, \leq)$, natural numbers with the usual ordering.⁶ Indeed the example above is performed within PNTT with an additional binary parametricity axiom binx . The idea is then that the nullary parametricity aspect of our argument gets translated to the monotonic quantification aspect of Atkey’s Kripke model. That is, $(0, 2)$ -ary parametricity emulates $(\mathbb{N}, \leq)$ -restricted Kripke 2-ary parametricity.

Regarding unrestricted Kripke parametricity, we observe that the quantification $(*)$ on preorders is *hard-coded* into $[f]_{\text{K2}}$, i.e. $[f]_{\text{K2}}$ is not the mere meta-theoretical conjunction of its W -restricted Kripke parametricities. In particular $[f]_{\text{K2}}$ lives one universe level higher than f which is peculiar. Interestingly this hard-coding plays a crucial role in Atkey’s proof of the roundtrip at $\forall A.(A \rightarrow A \rightarrow A) \rightarrow ((A \rightarrow A) \rightarrow A) \rightarrow A$. Indeed, within Atkey’s model, the proof introduces a variable $B : \mathcal{U}$ and uses $[f]_{\text{K2}}$ at preorder $W := \text{List } B$ with prefix ordering, and at type $A_0 := B$ (and some other type for A_1). So B intervenes both to determine the kind of parametricity being used ($W := \text{List } B$), and as an input for this parametricity ($A_0 := B$). We see this as a form of diagonalization, which we were not able to internalize.

⁶ A mismatch: our model uses the category $\square_0$ (Section 2.4) while Atkey’s uses *preorders* like $(\mathbb{N}, \leq)$.

5 Related and Future Work

We have already discussed the most closely related work in internally parametric type theory [8, 10, 36] and in nominal type theory [33, 31, 32, 11, 28] in detail in Sections 2 and 3 respectively. Note that a subtle error in the construction of the model shown in [8] was recently spotted in [2] and investigated in [29].

Regarding the semantics of nominal frameworks, PNTT is modelled (see Section 2.4) in $\mathbf{Psh}(\Box_2 \times \Box_0)$, i.e. bicubical sets over the binary cartesian cube category $\Box_2$ and the nullary affine cube category $\Box_0$. The latter is also the base category of the Schanuel sheaf topos [27, §6.3], which is in turn equivalent to the category of nominal sets [27], used to model FreshMLTT [28]. FreshML [33] is modelled in the Fraenkel-Mostowski model of set theory, which inspired the development of nominal sets. Schöpp and Stark’s bunched system [32] is modelled in a class of categories, including the Schanuel topos, that are locally cartesian closed and also equipped with a semicartesian closed structure. Finally, λ^{PIII} [11] has a syntactic soundness proof.

$\Box_0$ is a category of cubes without diagonals, so the internal language of its associated presheaf topos is among the first candidates to get a transpension type [21] with workable typing rules [19]. Dual to the fact that universal and existential name quantification semantically coincide, so do freshness and transpension. As such, the transpension type is already present as *Gel* in the current system, but *Gel*’s typing rules are presently weaker: the rules *GELF* and *GELI* remove a part of the context, rather than quantifying it. The relationship between *Gel* and transpension is explained in more generality in [21].

Regarding the implementation of PNTT, given that the binary CH theory was implemented in [36] on top of Cubical Agda [37], implementing PNTT (essentially nullary CH + name induction) is feasible. We think that such an implementation would be usable but not particularly scalable by default to perform parametricity proofs. The issue is that such proofs require the SAP, which cannot be proved as a single internal theorem (see Section 4.2 and [36]). A practical solution could be to automate the process of building SAP proofs. Alternatively, one could instead build PNTT on top of *observational* [2] nullary PTT, where the SAP is not needed since a first-class recursive translation attempts to replace the Bridge type. The Narya proof assistant in progress [18] implements observational n -ary PTT (and more).

References

- 1 C. B. Aberlé. Parametricity via cohesion. *CoRR*, abs/2404.03825, 2024. doi:10.48550/arXiv.2404.03825.
- 2 Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. Internal parametricity, without an interval. *Proc. ACM Program. Lang.*, 8(POPL):2340–2369, 2024. doi:10.1145/3632920.
- 3 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007. doi:10.1145/1292597.1292608.
- 4 Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. Syntax and models of cartesian cubical type theory. *Math. Struct. Comput. Sci.*, 31(4):424–468, 2021. doi:10.1017/S0960129521000347.
- 5 Robert Atkey. Syntax for free: Representing syntax with binding using parametricity. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009. doi:10.1007/978-3-642-02273-9_5.

- 6 Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 37–48. ACM, 2009. doi:10.1145/1596638.1596644.
- 7 Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Log. Methods Comput. Sci.*, 5(2), 2009. arXiv:0809.3960.
- 8 Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 67–82. Elsevier, 2015. doi:10.1016/J.ENTCS.2015.12.006.
- 9 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free - parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012. doi:10.1017/S0956796812000056.
- 10 Evan Cavallo and Robert Harper. Internal parametricity for cubical type theory. *Log. Methods Comput. Sci.*, 17(4), 2021. doi:10.46298/LMCS-17(4:5)2021.
- 11 James Cheney. A dependent nominal type theory. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:8)2012.
- 12 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017. URL: <http://collegepublications.co.uk/ifcolog/?00019>.
- 13 Martin Hofmann. *Syntax and semantics of dependent types*, pages 13–54. Springer London, London, 1997. doi:10.1007/978-1-4471-0963-1_2.
- 14 Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 204–213. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782616.
- 15 Martin Hofmann and Thomas Streicher. Lifting grothendieck universes. Unpublished note, 1997.
- 16 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.
- 17 Guilhem Moulin. *Internalizing Parametricity*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016. URL: <http://publications.lib.chalmers.se/publication/235758-internalizing-parametricity>.
- 18 Narya development team. Nominal syntax — narya documentation, November 2025. Accessed: 2025-11-19. URL: <https://narya.readthedocs.io/en/latest/nominal.html>.
- 19 Andreas Nuyts. Transpension for cubes without diagonals. In *Workshop on Homotopy Type Theory / Univalent Foundations*, 2025. URL: <https://lirias.kuleuven.be/retrieve/803969>.
- 20 Andreas Nuyts and Dominique Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 779–788. ACM, 2018. doi:10.1145/3209108.3209119.
- 21 Andreas Nuyts and Dominique Devriese. Transpension: The right adjoint to the Pi-type. *Log. Methods Comput. Sci.*, 20(2), 2024. doi:10.46298/LMCS-20(2:16)2024.
- 22 Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *Proc. ACM Program. Lang.*, 1(ICFP):32:1–32:29, 2017. doi:10.1145/3110276.
- 23 Martin Odersky. A functional theory of local names. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL’94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 48–59. ACM Press, 1994. doi:10.1145/174675.175187.
- 24 Andrew Pitts. Nominal sets and dependent type theory. In *TYPES*, 2014. URL: <https://www.irif.fr/~letouzey/types2014/slides-inv3.pdf>.

- 25 Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003. doi:10.1016/S0890-5401(03)00138-X.
- 26 Andrew M. Pitts. Nominal system T. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 159–170. ACM, 2010. doi:10.1145/1706299.1706321.
- 27 Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.
- 28 Andrew M. Pitts, Justus Matthes, and Jasper Derikx. A dependent type theory with abstractable names. In Mauricio Ayala-Rincón and Ian Mackie, editors, *Ninth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2014, Brasília, Brazil, September 8-9, 2014*, volume 312 of *Electronic Notes in Theoretical Computer Science*, pages 19–50. Elsevier, 2014. doi:10.1016/J.ENTCS.2015.04.003.
- 29 Sarah Rebollet. *Universe discrepancies in nominal presheaf models of parametric type theory*. Theses, Université Paris Cité, September 2025. URL: <https://theses.hal.science/tel-05584055>.
- 30 John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.
- 31 Ulrich Schöpp. *Names and binding in type theory*. PhD thesis, University of Edinburgh, UK, 2006. URL: <https://hdl.handle.net/1842/1203>.
- 32 Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, pages 235–249, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-30124-0_20.
- 33 Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. FreshML: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 263–274. ACM, 2003. doi:10.1145/944705.944729.
- 34 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: <https://homotopytypetheory.org/book/>.
- 35 Christian Urban. Nominal techniques in isabelle/hol. *J. Autom. Reason.*, 40(4):327–356, 2008. doi:10.1007/S10817-008-9097-2.
- 36 Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese. Internal and observational parametricity for cubical agda. *Proc. ACM Program. Lang.*, 8(POPL):209–240, 2024. doi:10.1145/3632850.
- 37 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.*, 31:e8, 2021. doi:10.1017/S0956796821000034.
- 38 Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989. doi:10.1145/99370.99404.

A Data Type of Intrinsically Plane Graphs in Agda

Malin Altenmüller   

University of Edinburgh, UK

Conor Titania Mc Bride  

University of Strathclyde, Glasgow, UK

Quantinuum, Cambridge, UK

Abstract

This work develops a suitable data type for plane graph embeddings in Agda. Graphs are used as combinatorial representations for string diagrams, a graphical calculus for monoidal categories. Whenever a monoidal theory does not include any symmetry or braiding operations, it describes processes that are sensitive to their topology. To encode this information in the graphical language, we have to consider surface-embeddings of graphs. We study the simplest case, plane graphs, and present their implementation in Agda. We overcome issues like the cyclic nature of a graph by using one of its spanning trees as an underlying inductive structure. The graphs we implement are plane by construction and any operation on them is guaranteed to preserve this planarity. Additionally, we present a notion of focussing on a certain subgraph within a graph. This operation is crucial for the application of local rewrite rules which themselves are at the centre of diagrammatic reasoning in monoidal categories.

2012 ACM Subject Classification Theory of computation → Type theory; Mathematics of computing → Graphs and surfaces; Theory of computation → Rewrite systems

Keywords and phrases planar graph, spanning tree, dependent types, graph rewriting

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.13

Supplementary Material *Software*: <https://maltenmuller.github.io/thesis/src/Index.html>

Funding *Malin Altenmüller*: funded by EPSRC grant number EP/X025551/1.

Acknowledgements We would like to thank the anonymous reviewers for helpful suggestions.

1 Introduction

To model diagrammatic reasoning for string diagrams [26], a combinatorial representation is needed that can accommodate diagram rewriting. In the standard case of symmetric monoidal categories this is typically achieved by using (hyper-)graphs and their rewriting [22, 9, 7]. Here, only the *connectivity* information between edges and vertices matter. *Non-symmetric* monoidal categories have received little attention to date, even though they have important applications. Theories without symmetry are crucial for modelling of quantum circuits [10, 8] in which swapping two qubits is an expensive operation and thus cannot be represented by an implicit changing of wires. They also describe a larger class of theories as they contain the braided and symmetric cases. String diagrams for non-symmetric monoidal categories have to encode an additional *topological* component as the arrangement of edges matters. Consequently, for their combinatorial presentation we have to consider graphs embedded on surfaces. For this work, we are interested in *plane* graphs which are those that can be drawn onto the surface of a sphere without any edges crossing.

We present a library for plane graphs and their rewriting in Agda. We will use Agda’s rich type system to encode the planarity property as part of a graph’s type. A graph of this type is guaranteed to be plane and any operation on graphs must preserve the planarity property. We solve the challenge of graphs being a cyclic structure by using one of their



© Malin Altenmüller and Conor Titania Mc Bride;
licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 13; pp. 13:1–13:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

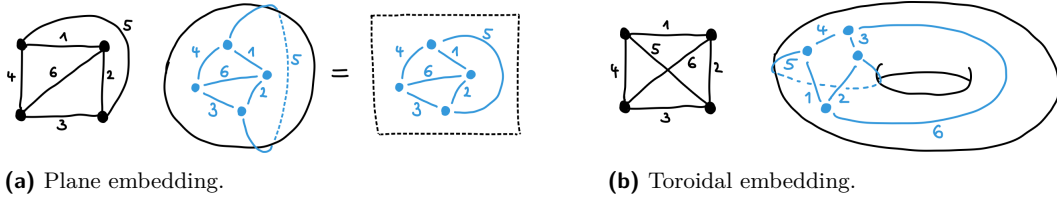
spanning trees as underlying inductive structure and store the remaining information on top. Instead of a standard definition of a surface-embeddings as additional information on top of a graph, we work with intrinsically plane graphs directly.

Surface-embedded graphs. We start by recalling some definitions and terminology on graphs and their surface-embeddings. More details can be found in the literature [16, 24].

► **Definition 1.** A directed graph G consists of a set V of vertices, a set E of edges, and two functions $s, t : E \rightarrow V$, source and target, respectively. A graph is called *total* (or *closed*), if s and t are total functions, and *partial* (or *open*) otherwise. An edge e is called *incident* to both its source and target vertices, and vice versa.

A *planar* graph can be embedded into the surface of a sphere (and, equivalently, into the plane), and the corresponding concrete embedding is called a *plane* graph.

► **Definition 2.** The embedding of a graph G into a surface S is a drawing of G onto S : Vertices are mapped to points of the surface, and edges to simple arcs, connecting two vertices each. As no two arcs share any point other than their endpoints, there are no crossing edges in a graph embedding. The faces of a graph embedding are regions of the surface enclosed by edge cycles of the graph.



■ **Figure 1** Two different surface embeddings of the same graph.

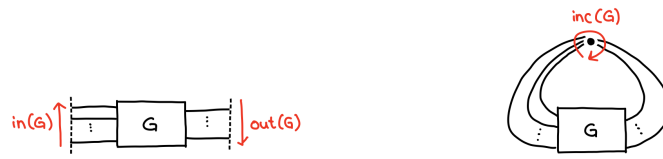
Surface-embeddings of graphs are determined by the arrangement of their faces on the surface. We will use a traversal of a graph's spanning tree to encode this information, but the following alternative presentation provides useful intuition:

► **Definition 3.** A rotation for a vertex v in a graph G is a cyclic lists $\text{inc}(v)$ of its incident edges of type $E \times \{\text{src}, \text{tgt}\}$. (e, src) occurs in v 's rotation if and only if $\text{src}(e) = v$ (and respectively for tgt). A rotation system for G consists of a rotation for each of its vertices.

Every rotation system for a connected graph G induces a unique (up to surface homeomorphism and orientation-preserving topological equivalence) embedding of G into an oriented surface, and vice versa [19, 11, 16].

► **Remark 4.** Throughout the formalisation we consider all graphs to be *closed*, meaning that both ends of every edge are connected to a vertex. We can represent open graphs by using *boundary vertices* which are auxiliary structures that represent the “outside” face of a graph [3, 2] and all input and output edges of an open graph are attached to its boundary vertex, see Figure 2. We can add topological information to these interface edges via the rotation at the boundary vertex.

► **Remark 5.** All graphs are considered *finite* and *connected* in this work. Additionally, we formalise *undirected* graphs only as directed edges do not affect topological considerations. Thus, instead of two different functions src and tgt as the data of a graph, we consider a function $\text{end} : E \rightarrow V \times V$ which encodes the two endpoints for each edge.



■ **Figure 2** Introduction of an auxiliary boundary vertex to capture input and output edges of a graph G . The interface edges together form the rotation $\text{inc}(G)$ at the boundary vertex.

Agda implementation. We use Agda’s *literate mode* to display type checked code throughout this document: all code blocks are extracts from an active buffer. In addition, we include links to the corresponding blocks in the source code with the  symbol, clickable in the PDF version of the document. When a bound variable is not explicitly quantified by a function definition, it assumed to be brought into scope as a module parameter or variable statement.

1.1 Related work

Graphs in functional languages. Representing cyclic and shared structures in a functional programming language has been the subject of various research. A number of projects use the technique of splitting a structure into a non-cyclic, inductive part and the remaining, potentially cyclic, information on top [6, 13, 27, 18], a similar approach to our work on plane graphs. Another approach focus on detecting cycles when defining operations on graphs, for example by marking nodes during an operation [12] or by using the features of a lazy functional language to unroll cycles a finite number of times only [21]. A different line of work suggests certain algebraic structures to represent graphs with a small set of generators and combinators [25], which are used to express properties of graphs and algorithms on them.

Implementations of plane graphs. The particular case of a representation of plane graphs is an important feature in the Rocq [5] proof of the Four Colour Theorem [15, 14]. This proof uses *hypermaps* in which the connection points between edges and vertices in the original graph are encoded as vertices and their relative position is encoded by relations on them. This presentation specifies an spatial arrangement of the faces of the graph embedding, similar to the information in a rotation system from which planarity can be checked. In our implementation we present the spatial arrangement of the graph more directly as an ordered traversal. We are therefore able to specify a data type of graphs that is intrinsically plane, thus the planarity property never has to be checked explicitly but holds by definition.

2 Plane Graphs in Agda

Typically, graph embeddings are defined as graphs together with additional structure, specifying a certain surface-embedding. This approach is unsuitable in the environment of Agda: Graphs are unordered data structures represented by sets and functions, but implementing unordered sets in Agda is a non-trivial task. It involves explicitly quotienting an inductive type, adding a layer of complexity. In contrast, surface-embeddings of graphs already come with an ordering on vertices and edges. In fact, it is exactly this order that turns a graph into a graph embedding. Therefore, instead of equipping graphs with their surface embedding, we will use graph embeddings as the first class objects straight away.

2.1 Graphs are cyclic structures

Another challenge for implementing graphs in Agda is their *cyclic* nature. Iterating a graph may unroll its cycles and continue indefinitely. In finite graphs we may encounter cycles, too, but we will eventually *revisit* nodes and edges in a traversal. These structures are on the one hand finite (as their closed representation contains finite sets only) but also infinite (as their traversal may continue forever). Infinite data types which have a finite representation have been studied as rational fixpoints of functors [23]. We would like to compare this approach to our work more formally in the future.

For our construction of a suitable representation of plane graphs, we will split a graph into two parts. The inductive part is a finite, cycle-free, connected, maximal subgraph which we can represent and manipulate in Agda easily. Alongside this structure, we store a graph's *looping* edges which form its cycles.

► **Observation 6.** *We observe the following properties about trees and graphs:*

1. *A tree is a connected, cycle-free graph, together with a choice of root.*
2. *Any maximal subtree of a graph G is a spanning tree of G .*

We take a graph G 's spanning tree as suitable maximal, non-cyclic subgraph. Any spanning tree includes all of G 's vertices, and some of its edges which we call *spanning* edges. The remaining information are all other edges of G , which we call *looping* edges. This leads to our encoding of graphs as *over-connected* trees. A tree is very straight-forward to represent as an inductive type in Agda. We then add the additional edges to the spanning tree at the relevant positions, see Figure 4a for an example.

► **Remark 7.** Throughout this work, we assume a choice of spanning tree and root for any given graph, which can be calculated for a given graph by standard graph algorithms. In traditional diagram rewriting, we have a canonical choice of root by taking the first input parameter of a system as its root. A potential extension to our implementation are equivalence classes of spanning trees for the same graph.

► **Remark 8.** The development of graphs as overconnected trees is similar to the notion of *blossoming trees* [1], which are trees equipped with additional half-edges at some vertices. Connecting these half-edges in a non-crossing manner constructs the faces of a plane graph embedding. A related structure are shuffles of parenthesis systems [4], a standard representation of plane graphs in combinatorics.

2.2 The order of edges matters

We now explain why careful inspection of the looping edges in a graph is enough to determine one of its surface-embeddings. Even though we do not implement rotation systems directly in this work, the ordering of edges is still a key characteristic of our construction. The translation from the spanning tree representation to rotation systems is straight-forward, which we demonstrate in Section 2.5. We observe the following properties about surface-embedded graphs and their plane subgraphs.

► **Definition 9.** *Given a graph $G = (V, E, s, t)$ and an edge $e \in E$ with $s(e) = u$ and $t(e) = v$, $v \neq u$, edge contraction is a function that maps G to $G' = (V', E', s', t')$ where:*

$$\begin{aligned} \text{— } V' &= V \setminus \{u\}, E' = E \setminus \{e\}, \\ \text{— } s'(e') &= \begin{cases} v & \text{if } s(e') = u \\ s(e') & \text{otherwise} \end{cases}, t'(e') = \begin{cases} v & \text{if } t(e') = u \\ t(e') & \text{otherwise} \end{cases}. \end{aligned}$$

We write G/e for the graph obtained by contracting e ; if H is a subgraph of G then G/H is the graph obtained by contracting all the edges of H .



■ **Figure 3** Edge contraction of edge e identifies its source and target vertices.

► **Example 10.** If the graph G contains another edge e' with the same source and target vertices as e , $s(e') = u$, $t(e') = v$, then e' forms a self-loop at v after the contraction of e .

► **Lemma 11.** Let G be a surface-embedded graph; then:

- The order of repeated edge contractions does not matter: for any edges $e_1, e_2 \in E_G$, we have $G/e_1/e_2 \cong G/e_2/e_1$.
- For any edge e , the edge contraction G/e embeds in the same surface.
- The edge contraction of any plane subgraph of G embeds in the same surface.

As a tree is trivially a plane graph, we can apply Lemma 11 and observe that contracting a graph's entire spanning tree does not change its genus. Correspondingly, a graph's genus is encoded by its looping edges alone.

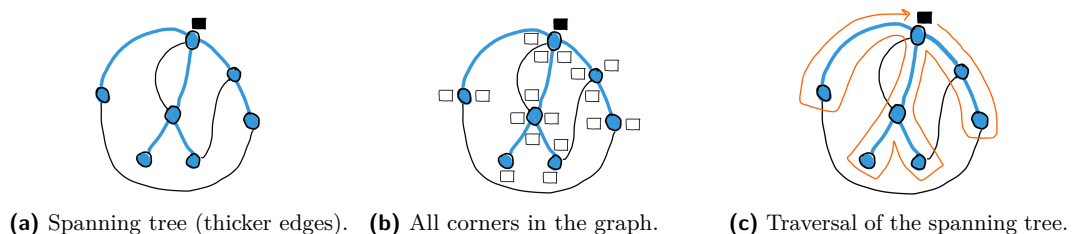
► **Corollary 12.** The structure of a graph's looping edges alone determine its genus.

As a graph's spanning tree contains all vertices of a graph, its contraction yields a graph containing a single vertex, with all the looping edges attached to it:

► **Definition 13.** A bouquet graph is a graph consisting of one vertex v and a set of edges E with $s(e) = t(e) = v$ for all edges $e \in E$. A rotation system for a bouquet graph is a single list in which each edge appears exactly twice.

We have observed that the structure of looping edges is key for specifying a graph's embedding. We therefore have record them in a certain order in our data type of plane graphs. This order corresponds to the rotation at the central vertex of the bouquet graph when contracting the graph's entire spanning tree.

Traversing a graph's spanning tree. We will record the occurrence of looping edges alongside a *clockwise* traversal of the graph's spanning tree. An example of this traversal operation is shown in Figure 4c. A spanning tree traversal moves through the entire graph and visits all possible *positions* inside it. Positions in a graph are assigned to its vertices and occur between any two adjacent edges. They are called *corners*, see Figure 4b.



■ **Figure 4** Spanning tree representation of an example graph.

► **Definition 14.** *The corners of a vertex $v \in V$ of graph $G = (V, E)$ are subdivisions of its neighbourhood, separated by the edges incident at v . The corners of all vertices $v \in V$ are the corners of the whole graph G . The root of a graph's spanning tree is a distinguished corner.*

► **Remark 15.** In case a vertex v has no incident edges, v 's only corner represents its entire neighbourhood. Otherwise, a corner specifies a triple (e_1, v, e_2) of a vertex $v \in V$ and two adjacent edges $e_1, e_2 \in E$ at v .

We now specify a graph traversal operation as a traversal of the spanning tree in Algorithm 1, see Figure 4c for an illustration. The algorithm visits all the corners in the graph in a specific order, defined by the structure of the spanning tree. Note that only one of the potential steps moves the traversal along a spanning edge. The other two options of steps in the algorithm record additional information, either a corner or a looping edge.

■ **Algorithm 1** Graph traversal.

This procedure defines the traversal of a graph by using its spanning tree. Starting from the root corner, the traversal steps through elements around vertices (edges and corners) in a clockwise order until it reaches the root again. Assume the current position of the traversal to be the corner $c1$ at vertex $v1$. The traversal can progress by one of three possible steps:

1. Following a spanning edge e : this operation progresses the traversal from vertex $v1$ to vertex $v2$ which is adjacent to $v1$ by e . The traversal then continues recursively at $v2$.
2. Passing a corner: this operation recognises a corner and progresses the traversal at the same vertex $v1$ to the next element in clockwise order.
3. Recording a looping edge: this operation passes a looping edge by storing it in a separate data structure (more details about this operation to follow). It progresses the traversal at the same vertex $v1$ to the next element in clockwise order.

We utilise the structure of this algorithm in the type of [Graphs](#) in Agda in Section 2.3. We introduce a type of [Steps](#) for the potential steps of a traversal, corresponding to the three steps in Algorithm 1. A full graph traversal is implemented as a sequence of [Steps](#).

We make the following important observation about tree traversals. This property will serve as a guide for the implementation of graphs as traversals of their spanning tree.

► **Proposition 16.** *In a graph traversal, edges and corners always alternate.*

Proof. In a graph with a single vertex and no edges, there is only one corner and the property is immediately true. Otherwise, a corner is defined by a vertex and two neighbouring edges at that vertex. We analyse the three possible steps in a graph traversal:

- The only way progress a traversal after a corner has been visited is to inspect one of its two defining edges.
- When recording a looping edge e at vertex v , the traversal moves past this edge at the same vertex. Every vertex in the graph is connected to at least one spanning edge (by the definition of spanning tree). Therefore, the vertex v has at least one more edge attached to it in addition to e . This means that the edge e specifies two different corners (one to its left and one to its right) and as every two neighbouring edges define a corner in between them, the next element in the traversal has to be a corner.
- When following a spanning edge t from v , we arrive at a different vertex v' . This vertex v' contains at least one corner, located next to (and characterised by) the spanning edge t . According to the clockwise graph traversal, this corner is visited next. ◀

2.3 The graph data type

To explain the data type of plane graphs we will first define one **Step** of a graph traversal. We then show how to combine multiple steps into a sequence to form a full traversal, and in the end consider some degenerate cases of graphs. We assume a type V of a graph's *vertices*, a type E of its *edges*, and a type C of its *corners*. We have observed in Proposition 16 that the traversal of a graph follows a strict pattern of alternating edges and corners. We use this information to guide the implementation of a **Step**, together with the state of looping edges at that position in the traversal.

Indexing type. A traversal type $\star$ contains two components:

```
TravTy : Set
TravTy = List E × Next
```

The first set in **TravTy** is a stack of edges E , represented as a **List** (the type of lists is defined below). This list records the looping edges throughout the traversal. Whenever we encounter one of these edges, the elements on the stack change: the first time we visit a looping edge we add it to the top of the stack, and the second time we pop it from the stack again. The stack-discipline ensures that edges which have been added most recently have to be removed first, which encodes precisely the planarity property of the graph embedding, see Section 2.4. Every step in the traversal is indexed by its effect on the stack of looping edges, with a full traversal starting and finishing on an empty stack.

data Next : Set where	after : Next → Next	What : Next → Set
edge : Next	after edge = corner	What edge = E
corner : Next	after corner = edge	What corner = C

■ **Figure 5** A traversal is guided by alternating edges and corners. $\star$.

The second set in a **TravTy** is a marker type **Next**. This is a two-element type with constructors **corner** and **edge**, see Figure 5. Throughout the traversal we use elements of **Next** as tags, specifying the kind of data we are currently visiting. The lifting of a tag to the data it labels is implemented by the function **What** (see Figure 5), mapping an **edge** label to the type of edges E and a **corner** label to the type of corners C . We use the tag system to specify how data in the traversal *changes* with every step. Recall Proposition 16: in a graph traversal, edges and corner always occur alternately. We enforce this alternation with the function **after** (see Figure 5) which changes the tag from an **edge** to a **corner**, and vice versa. This indexing structure ensures that graph data is well formed at fits together in the right way, by construction.

Sequencing relations with Stars. Before we specify the data type of steps in a traversal, we define a generic type of *sequences*, shown in Figure 6a $\star$.

Given a relation R , this type builds sequences of elements which are pairwise related via R . In addition to the relation parameter, this type abstract over the **Direction** of the sequence. The direction indicates to which end of a sequence new elements are added: the head element of a **cons** structure is located on the left, and that of a **snoc** structure on the right. We define graphically corresponding infix operators, **_.-_** for **cons** and **_>._** for **snoc**. Instantiating **FCat** with the two directions leads to the notions of **Star** (in analogy with the

13:8 A Data Type of Intrinsically Plane Graphs in Agda

```
data FCat (d : Dir) { X : Set } (R : X → X → Set) (x : X) : X → Set where
  nil' : { y : X } → x ≡ y → FCat d R x y
  cons : d ≡ cons → { y z : X } → R x y → FCat d R y z → FCat d R x z
  snoc : d ≡ snoc → { y z : X } → FCat d R x y → R y z → FCat d R x z
```

(a) Definition of relation composition.

```
Star = FCat cons
Rats = FCat snoc
```

```
List A = Star { One } (\ _ _ → A) ⟨⟩ ⟨⟩
Tsil A = Rats { One } (\ _ _ → A) ⟨⟩ ⟨⟩
```

(b) Instantiation with directions.

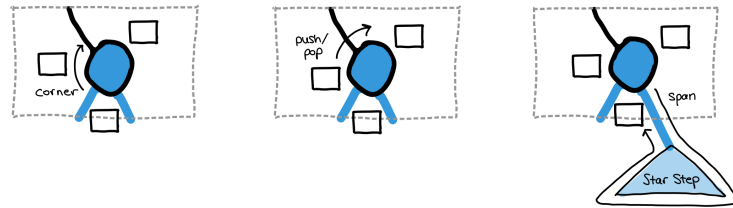
(c) Instantiation with the unit type.

Kleene star) and **Rats**, as shown in Figure 6b. We will use sequences in both directions later in this work. The simplest case of a forwards sequence is a **List**, see Figure 6c. Lists are implemented by instantiating **Star** with the trivial relation on the unit type, indicating that no particular property is enforced on neighbouring elements. Backwards “snoc”-lists, or **Tsils**, are the corresponding instance of **Rats**.

Steps of the graph traversal. The traversal type is used to index each **Step** in a traversal of a graph’s spanning tree. The different constructors distinguish different positions inside a tree, and the different data stored in a graph $\mathfrak{G}$. See Figure 7 for an illustration.

```
data Step : TravTy → TravTy → Set where
  corner : (c : C) → Step (es , corner) (es , edge)
  push : (e : E) → Step (es , edge) (e , es , corner)
  pop : (e : E) → Step (e , es , edge) (es , corner)
  span : (e : E) (v : V) → Star Step (es , corner) (es' , edge) → Step (es , edge) (es' , corner)
```

The first constructor defines visiting a **corner** c in a graph. This **Step** does not change the stack of edges es , but it enforces the next element in the traversal to be an **edge**. Another potential traversal **Step** visits a looping edge e . The first time we do so, we **push** this edges onto the stack es , and the second time it is **popped** from the stack. We will see shortly (in Theorem 20) that we can assume e to always be at the *top* of the stack whenever we encounter it for the second time in the graph traversal. Both **push** and **pop** change the traversal type such that the next step is required to be a **corner**. The final constructor describes the traversal of a spanning edge e . In this case, the algorithm follows e to reach a new subtree, rooted at vertex v . This subtree is represented by a *sequence* of steps around the child node v , defined as a **Star Step**. Throughout this **Star Step**, the indices of any two neighbouring **Steps** have to match, according to the indexing structure. In the constructor **span** we ask for an explicit



■ **Figure 7** The different cases of constructing a **Step** of the traversal.

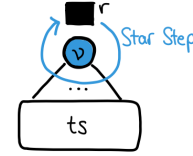
edge e connecting to the root vertex v of the subtree. The traversal of this subtree is passed as a **Star Step**, starting with the first **corner** at v and finishing at the **edge** e through which we have entered the subtree. The subtree must contain at least a corner, thus the **Star Step** is non-empty (encoded by the change in indexing type from **corner** to **edge**). Overall, **span** describes a **Step**, progressing from the **edge** e to expecting a **corner** as the next element in the traversal. As the traversal of a subtree may add or remove edge from the stack, the **span** constructor passes the stack es onto the subtree and returns the result es' to its parent.

The traversal of an entire graph is implemented as a traversal of all its subtrees, therefore it will itself be represented as an instance of a **Star Step**.

The data type of graphs. We have specified all the relevant operations to define graphs as traversals of their spanning trees. The data type **Graph** accommodates degenerate cases of graphs as well as the general case of a tree traversal.

► **Definition 17.** *The type **Graph** is defined as the clockwise traversal of a graph's spanning tree, together with the option to add additional edges at any position* ⚙️:

```
data Graph : Set where
  empty : C → Graph
  vertex : C → V → Graph
  tree : C → V → Star Step ([], edge) ([], corner) → Graph
```



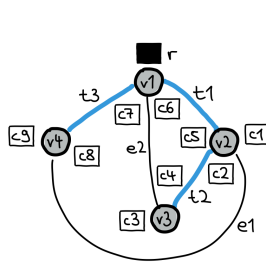
(a) Definition of the data type of **Graphs**.

(b) Illustration of **tree** $r\ v\ ts$.

A graph may be **empty** in which case it contains neither a vertex nor an edge, but a root corner. The other degenerate case is a graph containing a single **vertex** $r\ v$ with no edges attached. This graph contains one corner only: its root r . In the most general case, a graph is specified as a **tree** traversal. Together with a root corner r located at a vertex v , this constructor expects the traversal of all subgraphs, encoded as a **Star Step** (see Figure 8b for an illustration). For an entire **Graph** we ask for the stack of looping edges to be empty at the beginning of its traversal (in the first index of the **Star Step**), and empty again at the very end of it (in the second index). This reflects the fact that we consider closed graphs only in the implementation (cf. Remark 4). The **tree** constructor specifies a graph with a non-empty set of subgraphs, as the base cases are already covered by the other constructors. We achieve this by separating the root corner from the **Star Step** of subgraphs and indexing the **Star Step** by two different elements of **Next** (and thus requiring the sequence to contain at least one element). Therefore, starting from the root corner, the traversal starts with an **edge** and, at the other end, expects a **corner** to finish (which is the root corner again).

► **Example 18.** Assume variables $(v1, \dots, v4 : V)$ for vertices, $(t1, t2, t3 : E)$ for spanning edges, $(e1, e2 : E)$ for looping edges, and $(r, c1, \dots, c9 : C)$ for corners. Figure 9 shows an example graph, together with its implementation ⚙️. We observe in particular the alternation of edge operations (**span**, **push**, and **pop**) and **corners** (as discussed in Proposition 16). Figure 9 additionally shows the example graph's rotation system.

► **Example 19.** The bouquet graph with one vertex $v1$, two self-loops $e1$ and $e2$, a root corner r and three further corners $c1$, $c2$, and $c3$ is illustrated and implemented in Agda as shown in Figure 10.



```

exGraph : Graph
exGraph
= tree r v1
  (span t1 v2
    (corner c1 ,- push e1 ,- corner c2
      ,- span t2 v3 (corner c3 ,- push e2 ,- corner c4 ,- [])
        ,- corner c5 ,- [])
      ,- corner c6 ,- pop e2 ,- corner c7
        ,- span t3 v4 (corner c8 ,- pop e1 ,- corner c9 ,- []) ,- []))

```

```

exGraphR : rotations exGraph
≡ (t1 ,- e2 ,- t3 ,- []) ,- (e1 ,- t2 ,- t1 ,- []) ,- (e2 ,- t2 ,- []) ,- (e1 ,- t3 ,- []) ,- []

```

■ **Figure 9** Graph with vertices $v\cdot$, root r , corners $c\cdot$, spanning edges $t\cdot$, and looping edges $e\cdot$.

```

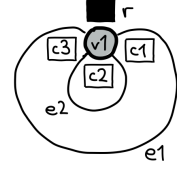
exBouquet : Graph
exBouquet = tree r v1 (push e1 ,- corner c1 ,- push e2 ,- corner c2
  ,- pop e2 ,- corner c3 ,- pop e1 ,- [])

```

```

exBouquetR : rotations exBouquet ≡ (e1 ,- e2 ,- e2 ,- e1 ,- []) ,- []

```



■ **Figure 10** Bouquet graph with one vertex $v1$, root r , corners $c\cdot$, and looping edges $e\cdot$.

2.4 Planarity

In Lemma 11 we have observed that the looping edges alone determine the surface a graph is embedded in. Therefore, the data structure we use to organise looping edges in the tree traversal is crucial for defining a certain graph embedding. Recall that in a plane graph no two edges are allowed to cross each other. Therefore, the rotation at the central vertex of a bouquet graph amounts to a well bracketed word. The stack discipline we use for organising the additional edges ensures that an edge can only be popped if it is positioned on the top of the stack. Therefore, the order of popping two edges from the stack has to be the reverse of pushing them onto the stack. This property expresses precisely the data in a plane graph.

► **Theorem 20.** *Terms of type `Graph` correspond to plane graphs with a spanning tree.*

Proof. “ $\Rightarrow$ ” We show that any term of the type `Graph` is a plane graph: The `empty` graph and the graph with one `vertex` are trivially plane graphs as they do not contain any edges. Consider the constructor `tree`, in particular the element `Star Step`: encountering a `corner` or `pushing` an edge e does not create any edge crossings (as the other end of the e is not yet attached to a vertex). When `spanning` a subtree, no edge crossing is introduced as the spanning tree of a graph is always plane. When `popping` an edge e from the stack, no crossing is introduced because of the stack discipline, meaning that for all other edges e_i in the graph, e_i is fully contained in one of the two regions that e defines (compare also Remark 22). Therefore, the constructor `tree` implements a plane graph.

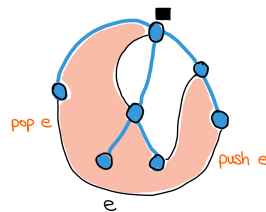
“ $\Leftarrow$ ” We show that each plane graph is determined by a spanning tree traversal as defined by the data type [Graph](#). Assume a plane graph G with a spanning tree. The proof consists of two steps. First we contract the graph’s spanning tree to a bouquet graph and then show that using a stack ensures planarity of the bouquet graph:

1. Assume a spanning edge e of the graph G and its two ends $s(e)$ and $t(e)$. We have that $s(e) \neq t(e)$, because by definition a spanning tree does not contain self-loops. Contracting e constructs a new vertex v whose rotation is the concatenation of the individual rotations of $s(e)$ and $t(e)$. After the contraction, a traversal of the tree in clockwise order yields the same order of looping edges as before the contraction. We repeat the process for all edges in the spanning tree. As a spanning tree contains all vertices in the graph, we get a bouquet graph with all edges being the looping edges. The rotation at the central vertex in the bouquet amounts to the order of looping edges in the clockwise (and therefore depth-first) traversal of the graph’s spanning tree.
2. Assume two edges e_1, e_2 in the rotation of edges around the central vertex in a bouquet graph. By Definition 13 of bouquet graphs, each edge appears in the rotation exactly twice. For e_1 and e_2 to be plane, they must not cross each other. This is the case if they appear in the rotation at the central vertex in the order e_1, e_1, e_2, e_2 (or a cyclic permutation of this order). An edge crossing would look like e_1, e_2, e_1, e_2 in the rotation. The plane order is enforced by pushing an edge on a stack the first time we visit it in the rotation, and popping it at the time of the second visit. With this stack discipline we are unable to express the edge crossing as above, because by the time we reach e_1 for the second time, e_2 is on top of the stack and we cannot pop e_1 . $\blacktriangleleft$

► **Corollary 21.** *A graph is plane if and only if it can be expressed as a term of type [Graph](#).*

Proof. By standard graph algorithms, we can calculate the spanning tree for any graph. Then apply Theorem 20. $\blacktriangleleft$

► **Remark 22.** The statement of Theorem 20 can be interpreted in an alternative way: Every looping edge e *completes* a face of a graph’s surface embedding, see Figure 11. Inserting a looping edge into a spanning tree amounts to splitting the surface into two regions, an “inner one” containing the newly completed face, and an “outer” one. This observation is matched in the structure of the stack, because every edge e can be interpreted to partition the stack into two segments. The “outer” segment contains edges that are stored below e on the stack and cannot be popped as long as e is still on the stack. In Figure 11 this is the case for all edges *outside* of the highlighted region. The “inner” segment of the stack describes edges that are stored on top of e and which have to be popped before e . This is the case for edges *inside* the region enclosed by e in Figure 11.



■ **Figure 11** A looping edge enclosing a region (shaded) of the embedding.

2.5 Translation to rotation systems

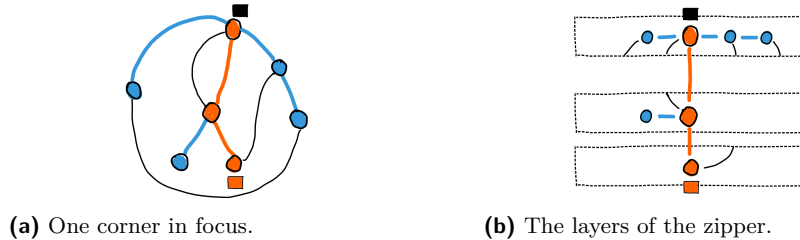
As we define graphs as clockwise traversals of trees, the translation into a rotation system is straight-forward. We can read off the order of edges around each vertex by inspecting all elements of its traversal [Star Step](#) in order ⚙️. As the spanning tree visits all vertices in the graph, we will eventually collect all rotations ⚙️.

The converse direction of translating between a [Graph](#) and its rotation system is not as simple. This is because the spanning tree representation includes a choice of spanning tree and root, compare Remark 7. A translation from a rotation system to a [Graph](#) would require choosing a spanning tree for a graph which is an operation we have deliberately left out of the development.

3 Zippers for Plane Graphs

Diagrammatic reasoning consists of applying rewrite rules to subdigrams. To implement this *local* operation, we now introduce data structures to isolate a subgraph of interest. We implement an instance of Huet’s Zipper [20] for the spanning tree representation of plane graphs. Zippers are defined for tree-like structures and consist of a subtree in focus together with a context, stored in a path from the tree’s root to the focus. For us, this separation will ensure that a rewrite rule is only affecting a particular subgraph and not the context. At every step in a zipper’s path, the sibling subtrees to its left and right are recorded. Paths are stored *bottom-up*, allowing for fast access to the subtrees closest to the focus. This provides zippers with a *cursor* operation: immediate neighbours in all directions can be accessed quickly.

► Remark 23. We define zippers to focus on a particular *corner* in a [Graph](#), rather than a whole subgraph. We are interested in extending the implementation in the future, but focussing on an individual corner is sufficient for specifying the construction of the zipper.



■ **Figure 12** Example of a graph’s zipper.

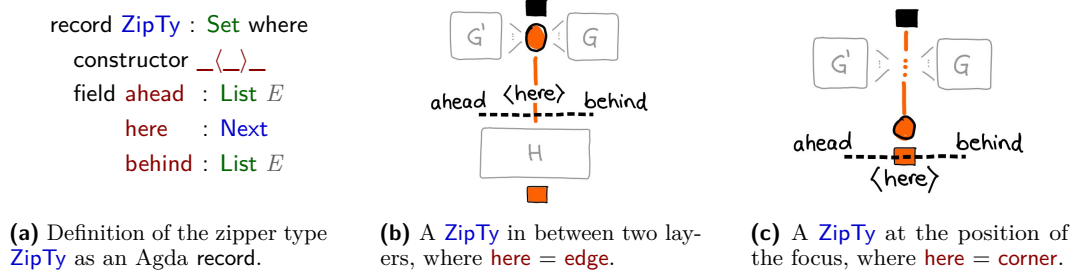
Zippers for plane graph consist of the corner in focus and the remaining graph as context. Alongside the path to the focus, we record neighbouring subgraphs and looping edges. The corner in the zipper’s focus is located in a particular face of the graph embedding, enclosed by looping edges. These edges form the stack at the corner’s position in a graph traversal and have to be included in the information stored in a zipper.

Path structure. Each step in a zipper’s path is associated with a central vertex and records a layer of substructures to the left and right of that vertex. Multiple layers are connected to each other via spanning edges between their central vertices. Overall, the path is represented as a sequence of layers from the focus to the root.

Looping edges. To record looping edges in the path, we introduce an indexing type for layers which contains a pair of edge stacks, one for each side of the path. In fact, every layer is indexed by two of these pairs, one at the “root-side” of the layer and one at the “focus-side” (i.e. four stacks in total). As we traverse subtrees in a layer, the looping edge stacks change. The overall path records the change of looping edges between the stack at the focus and the empty stack at the graph’s root. This indexing structure ensures that the planarity property is preserved when a number of layers is rewritten, as long as the change in edge stack remains the same.

3.1 Indexing type

To keep track of the stack of looping edges and preserve the surface of the embedding, each layer in the path will be relating elements of a *zipper type* $\mathbb{Z}$. To explain the elements of a zipper type, we imagine a horizontal cut through a path, as illustrated in Figure 13. The indexing type describes the state either in between two layers (Figure 13b) or at one end of the path (Figure 13c). As before, we carefully arrange the constructor arguments in the same diagrammatic order as the graphs they describe.



■ **Figure 13** Illustration of a zipper type as a horizontal cut through a path.

The field `here` stores where in the path the cut is made, either through an `edge` or through a `corner`. In a path, there are only two positions in which the field `here` is a `corner`: at the very top of the path (with `here` being the root corner) and the focus (where `here` is the corner in focus). At any other position on the path, the `ZipTy` describes the state of a spanning `edge`.

The other two fields in a `ZipTy` record the state of the looping edge stack when the tree traversal is approaching `here` (which occurs twice in a full graph traversal). In the (clockwise) traversal we first approach `here` from the top right hand side, with some edges already pushed onto the stack (by the subgraph G in Figure 13). The state of the stack is stored in the field `behind`, emphasising that it represents the information in the previous traversal steps. To continue the traversal from `here`, we move further down into the structure. In case of `here` = `edge` this means traversing the subtree H which the edge is leading to. In contrast, if `here` = `corner` there is no further inner subtree and we immediately return. As the traversal returns from a substructure, it visits the cut position `here` again, this time approaching from the bottom left hand side. The state of the stack at this point is stored in `ahead`. The edges on the stack `ahead` will be popped in the remaining traversal of G' between the cut and the root.

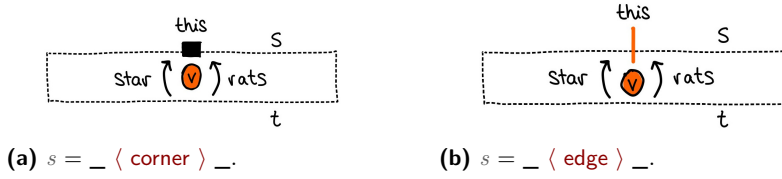
3.2 Path structure

Every step in the path is associated with a vertex v and contains v ’s subgraphs that are located to the left and right of the path. Additionally, it stores the edge through which the path continues to the next vertex towards the root. As we implement the path to start at

13:14 A Data Type of Intrinsically Plane Graphs in Agda

the focus and grow “upwards” to the root, we encode each step in the path as a backwards *layer*, called a **Reyal**. A **Reyal** at a vertex v is indexed by two elements s and t of the zipper type, with s encoding the **Reyal**’s root side and t its focus side. These elements store the state of the edge stacks before and after traversing v ’s side subgraphs ⚙ .

```
record Reyal (s t : ZipTy) : Set where
  constructor reyal
  field v      : V
        this   : What (here s)
        star   : Star Step (ahead t , after (here t)) (ahead s , here s)
        rats   : Rats Pets (behind s , after (here s)) (behind t , here t)
```



■ **Figure 14** Illustration of the information in a **Reyal** $s\ t$.

Each **Reyal** is linked to a vertex v on the path to the corner in focus. It relates two **ZipTys**, s towards the root, and t towards the focus. In addition to the central vertex, a **Reyal** contains a field **this** which stores **here** of s towards the root. This element is either the root corner (see Figure 14a) or an edge (Figure 14b). The sibling subtrees alongside the path in a **Reyal** are stored in the fields **star** and **rats**. On the left hand side of the path are subtrees that occur in the graph’s traversal *after* **here** t has been visited. They are stored as a list of **Steps**, similar to the traversal of subtrees in the definition of **Graphs** themselves. The subtrees traversed *before* reaching the focus are located on the right hand side in the **Reyal** and stored as the *backwards* list **rats** (whose structure we will explain in the following). The orientation of **star** and **rats** allow for easy access to the sibling subtrees closest to the focus of the zipper.

Lists and backward lists. To resemble the cursor-like nature of the original definition of Huet’s Zipper, we structure data both in a forwards and backwards way. The left of the vertex in a **Reyal** we store a (standard) forwards list of subtrees, represented as **Star Step** $s\ t$, just as we have seen in the definition of **Step** in Section 2.3. As a **Reyal** is oriented towards the root, the subtrees to the right of each vertex in the path are stored in an *anticlockwise* direction. Therefore, we introduce a backwards version of a **Step**, called a **Pets** ⚙ :

```
data Pets : TravTy → TravTy → Set where
  corner : (c : C) → Pets (es , corner) (es , edge)
  push   : (e : E) → Pets (es , edge) (e , - es , corner)
  pop     : (e : E) → Pets (e , - es , edge) (es , corner)
  span   : (e : E) (v : V) → Rats Pets (es , corner) (es' , edge) → Pets (es , edge) (es' , corner)
```

Similar to **Steps**, we can sequence **Pets**, this time using the reverse relation composition **Rats** (recall Section 3.2). Importantly, the information in a **Rats Pets** is the same as in a **Star Step**. The only difference is their orientation.

Sequences of *Reyals*. An entire path is defined as the concatenation of multiple *Reyals*, or, more precisely, a *Rats Reyals*. To construct this structure, we use relation composition with one additional requirement: concatenation of two *Reyals* is well defined only if they share an *edge* between them. Any *Reyal* with an index of shape $(\_ \langle \text{corner} \rangle \_)$ is located at one end of the path. This property of a *Rats Reyals* being well formed is called connectivity $\star$:

```

ConReyals :  $\forall \{ t \ t' \} \rightarrow \text{Rats Reyals } t \ t' \rightarrow \text{Set}$ 
ConReyals (snoc refl  $\{ \_ \langle n \rangle \_ \}$   $(rz \ -, \ r) \ r0$ ) = ConReyals  $(rz \ -, \ r) \times (n \equiv \text{edge})$ 
ConReyals  $rz = \text{One}$ 

```

Connectivity is only meaningful if a path consists of two or more *Reyals*. We encode this requirement in the first case of the definition of *ConReyals* by asking the inputs *Rats Reyals* to contain a *snoc* as well as its tail *rz* being a non-empty sequence of the format $(\_ \ -, \ \_)$. This pattern match exhibits two *reayls*, *r0* and *r*, together with a tail *rz*. The implementation of the predicate then asks for an *edge* in between the first two *Reyals* as well as for the remaining sequence *rz - , r* to be connected. The base case applies if there are less than two *Reyals* present because there is no connectivity boundary, and thus the connectivity is trivial.

► Remark 26. A *Reyal*'s distinguished element *this* is located at its root-side. A *Rats Reyals* does not include the focussed corner of the zipper, but rather leaves a *hole* in its position. Thus, a path stores precisely the *context* for a particular corner in the graph. The construction of a zipper adds the corner in focus to a path.

3.3 The type of zippers for *Graphs*

Overall, a zipper is implemented as a corner in focus together with a *Reyal* structure encoding the path between the root corner and the focus. The *Zipper* data type is indexed by a list of edges. This list represents the state of the stack of looping edges at the position of the focus when traversing the graph's spanning tree in clockwise direction $\star$.

```

data Zipper : List E  $\rightarrow$  Set where
  empty   : C  $\rightarrow$  Zipper []
  vertex  : C  $\rightarrow$  V  $\rightarrow$  Zipper []
  root    : C  $\rightarrow$  V  $\rightarrow$  Star Step ([], edge) ([], corner)  $\rightarrow$  Zipper []
  root-vtx : (r : Reyals ([], corner)) (es : List E)  $\rightarrow$  C  $\rightarrow$  Zipper es
  reyals   : (r : Reyals ([], corner)) (ds : List E)  $\rightarrow$ 
     $\rightarrow$  (rz : Rats Reyals (ds : List E) (es : List E))
     $\rightarrow$  ConReyals rz  $\rightarrow$  C  $\rightarrow$  Zipper es

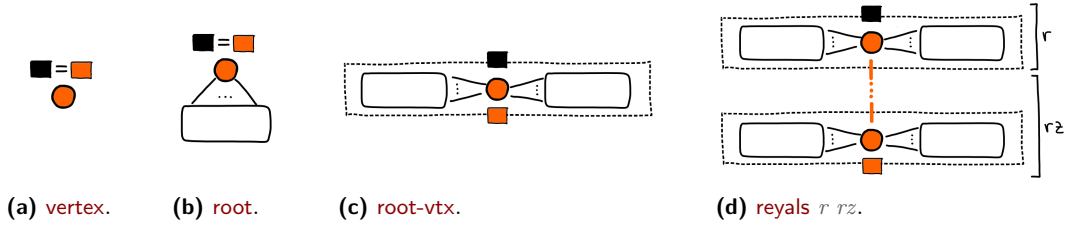
```

Let us unfold this definition slowly. As the focus consists of a corner only, the state of the stack is the same to either side of it, therefore we express it as a single list in the index of the data type.

All constructors for a *Zipper* ask for an element of type *C* which is the corner in focus. The remaining input information encodes the context of this focussed corner. Each constructor covers a different shape of graph and a different location of the corner in focus inside the graph. The different constructors are illustrated in Figures 15.

Degenerate graph. In case a graph is empty or consists of a single *vertex* with no edges attached to it, the only corner a zipper can focus on is the root, see Figure 15a. As there are no edges in the graph, the list of edges in the zipper's index is empty.

13:16 A Data Type of Intrinsically Plane Graphs in Agda



■ **Figure 15** The different constructors of a [Zipper](#).

Root corner. A [Zipper](#) may focus on the [root](#) corner of a more complex graph, see Figure 15b. In this case, the entire graph (except the root) constitutes the context. We do not need any layer structure to encode this case; a single [Star Step](#) is sufficient. Additionally, as the path from the root to the focus is empty, so is the stack of looping edges at the focus of the [Zipper](#).

Root vertex. If the zipper focuses on any corner that is incident to the root *vertex* (other than the root), the context of this corner may be expressed by a single [Reyal](#) r , as shown in Figure 15c. Importantly, this [Reyal](#) sits between *two corners*, the root corner and the corner in focus. Therefore, both indices are of the shape $(\_ \langle \text{corner} \rangle \_)$. Because of the definition of a [Reyal](#), the [star](#) and [rats](#) fields of r are guaranteed to be non-empty. This ensures no overlap with the [vertex](#) constructor. The behaviour of a single [Reyal](#) cannot be expressed by the more general construction of a zipper as [reysals](#) which is why we require the additional constructor [root-vtx](#).

General case. In the general case of a [Zipper](#), its path contains at least two vertices and thus at least two [reysals](#). The constructor asks for a non-empty [Rats Reyal](#) structure encoded as an individual [Reyal](#) and a [Rats Reyal](#), together with a proof that this non-empty sequence is connected. We enforce the structure to be non-empty by requiring the [Reyal](#) r , which is located closest to the root, as a separate argument from the remaining [Rats Reyal](#) rz , as is illustrated in Figure 15d. This argument structure in [reysals](#) allows for fast access to both the outermost layer r and the innermost layer (at the head of rz). We will make use of this property when reconstructing the original graph from its zipper.

As we have taken great care about storing the looping edges in a stack-like fashion when defining the type of [Zipper](#), we get exactly the right structure to partition a plane graph into a context and a focus:

► **Proposition 28.** A [Zipper](#) defines a plane graph context together with a corner in focus.

Proof. The stack of the looping edges is stored in the zipper type at each [Reyal](#). The way we compose [Reyals](#) ensures that the stack is preserved in between any two of them. Subtrees to the right and left of a [Reyal](#) can alter the stack but only by a valid push or pop operation (guarded by the valid changes of edge stack) which does not introduce any crossing edges. ◀

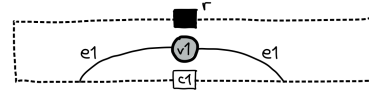
► **Example 29.** This example describes the zipper of a graph containing one vertex $v1$ and one self-loop $e1$. The focus of the zipper is the corner $c1$ which is the only other corner apart from the root r . The zipper is illustrated in Figure 16, and implemented using the [root-vtx](#) constructor as we only need one [Reyal](#) to describe the subtrees between root and focus.

The index of the zipper contains one edge $e1$, as this edge is on the stack at the corner in focus. In the graph traversal, $e1$ is pushed before reaching $c1$ and popped afterwards ⚙.

```

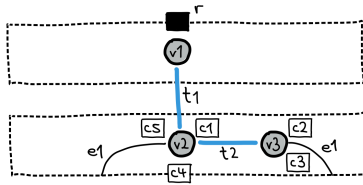
zi-root-vtx : Zipper (e1 , - [])
zi-root-vtx
  = root-vtx (reyal v1 r (pop e1 , - []) ([], push e1)) c1

```



■ **Figure 16** Example of a zipper focussed at the root vertex.

► **Example 30.** This larger example of a zipper assumes vertices $v1, v2, v3$, spanning edges $t1$ and $t2$, a looping edge $e1$, a root corner r , and further corners $c1, \dots, c5$. The zipper consists of two **Reyals** and focuses on the corner $c4$. The zipper's index indicates that the edge $e3$ lies on the stack at the corner in focus. The zipper is illustrated and implemented as shown in Figure 17 🌀.



```

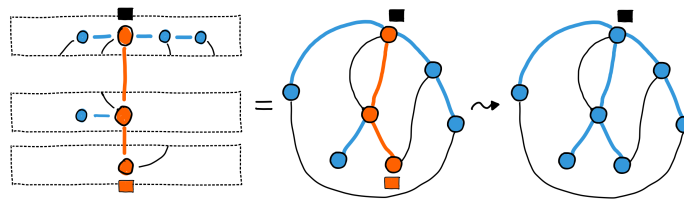
zi-reyals : Zipper (e1 , - [])
zi-reyals
  = reyals (reyal v1 r [] []) ([], reyal v2 t1
    (pop e1 , - corner c5 , - []))
    ([], corner c1 -, span t2 v3
    ([], corner c2 -, push e1 -, corner c3 )))
    < c4

```

■ **Figure 17** Example of a zipper with two reyals.

4 Computing the Original Tree

In this section we explain how to recover the original graph representation from one of its zippers. This operation is of relevance for the implementation of graph rewriting. The substitution of a subgraph typically operates locally, leaving the context graph unchanged. This property can be easily implemented with zippers. To compute the overall result of a rewrite, we then need to transform the layered structure of the zipper back into a standard spanning tree traversal. The translation operation is illustrated in Figure 18.



■ **Figure 18** Calculating the original tree from a zipper.

As the focus of a zipper is a corner only in our implementation, this operation has to merely insert the corner in focus at the correct position in the context graph. Whereas the specification seems simple, in the implementation the reorganisation of data from the zipper into a clockwise traversal requires great care as the stack property of the looping edges has to be maintained at all times. Let us first have a look at the implementation of the overall operation **ziToOldRoot** 🌀 and unfold all the details afterwards.

13:18 A Data Type of Intrinsically Plane Graphs in Agda

```

ziToOldRoot : { as : List E } → Zipper as → Graph
ziToOldRoot (empty r)    = empty r
ziToOldRoot (vertex r v) = vertex r v
ziToOldRoot (root r v st) = tree r v st
ziToOldRoot (root-vtx (reyal v r star rats) c) = tree r v (rats ⟨*⟩) (corner c ,- star))
ziToOldRoot (reyls (reyal v r star rats) rz cR c)
  = let (e , ls , cS) = toLayers rz cR c in starToGraph r (layer v rats e star) ls cS

```

In the first three cases the zipper is focussing on the root corner which makes for an easy calculation of the overall graph. All subtrees (if there are any) are stored in the right order already and together with the root corner r we can build the graph straight away.

Fish and chips. In all other cases, the zipper contains both forward and backward sequences of subtrees. As we move through a focussed structure, we will need to transform forwards and backwards components into each other, preserving their order like beads on an abacus. This is the purpose of two operations, called “fish” and “chips” [17]¹. The fish operator $\_⟨*⟩\_$ takes a backwards and a forwards sequence as inputs and computes a backwards sequence $\clubsuit$. Similarly, the chips operator $\_⟨*⟩\_$ transforms a backwards and a forwards sequence into a forwards sequence:

```

_⟨*⟩_ : Rats Pets k l → Star Step l m → Rats Pets k m
_⟨*⟩_ : Rats Pets k l → Star Step l m → Star Step k m

```

In case of a **root-vtx**, the zipper focuses on a corner at the root vertex other than the root corner. In this case we have to work with one **Reyal** of information containing all the subtrees attached to the root vertex, both to the left and right of the corner in focus. To restore the original graph traversal, we combine the two fields of the **Reyal** which contain the subgraphs, *star* and *rats*. To do this, we call “chips” on the two sequences subtrees *star* and *rats* while inserting the corner in focus in between them. This returns the traversal of the entire graph. Note that because of the indices of type **ZiTy**, this implementation is only accepted because we are inserting a corner in between *star* and *rats* to concatenate them. This makes sense as the two sequences are the direct neighbours of the corner in focus.

In the most general case, the zipper is a **reyls** structure with the path containing at least two vertices (and hence at least two reyls). In this case, calculating the original graph consists of two steps: First we transform the bottom-up **Reyal** structure into a top-down **Layer** structure and afterwards convert the layered representation into a clockwise traversal.

The first step involves stepping thorough the path and turning each **Reyal** into a **Layer** $\clubsuit$. This operation orientates the path such that it is centred around the original root (as opposed to the corner in focus), which will be helpful when calculating the clockwise traversal.

```

toLayers : { n : Next }
  → (rz : Rats Reyal (as ⟨ edge ⟩ bs) (ks ⟨ n ⟩ ms)) → ConReyls rz
  → (focus : What n)
  → E × Σ (Star Layer (as ⟨ edge ⟩ bs) (ks ⟨ n ⟩ ms)) λ ss → ConLayers ss

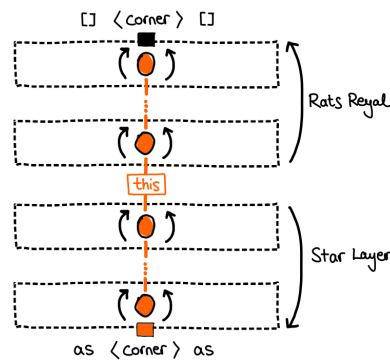
```

Recall that the most general constructor for a zipper, **reyls**, the **Reyal** closest to the root is stored separately from the rest of the path. This ensures the correct connectivity property between all **Reyls**, as the base case of **ConReyls** describes both the empty **Reyal**

¹ “Fish” is called fish, because it resembles a fish. And “chips” is the counterpart of fish.

and a single **Reyal**. We will turn the top-most **Reyal** into a **Layer** manually, but the turning of the remaining **Rats Reyal** is specified by the function **toLayers**. As the function only has to address these remaining **Rats Reyal**, the index at the root side contains an **edge**. This also ensures that we can recursively call **toLayers** on all subsequent **Rats Reyal**.

As we step through the entire path and turn backwards into forwards layers, the order of sibling subtrees in the fields **star** and **rats** stays the same. The main difference when changing the orientation of the path is the location of the element **this**: in a **Reyal** structure, **this** is located at the root-side and in a **Layer** structure it is stored at the focus-side. As the corner in focus is not part of the **Reyal** structure, **toLayers** requires an additional argument *focus*. After “re-shuffling” the information, the function returns not only a **Star Layer** but also the edge that is located closest towards the root. This edge was originally part of the **Reyal** structure but is not include in the **Layer** structure. Crucially, we are not allowed to forget about this edge: it will be needed in the second step of the calculation. Any function which is reordering the information in a graph or a zipper (such as **toLayers**) has to be linear in its input arguments. Even though the graph structure may be reorganised, all of its individual elements have to be preserved.



■ **Figure 19** Intermediate state of a **toLayers** operation.

The transformation from **Reyals** to **Layers** exposes an important intermediate state which is shown in Figure 19. The intermediate state occurs after a few (but not all!) recursive calls to **toLayers** on a zipper. In this intermediate state, the element **this** is located somewhere in the middle on the path. The path between this location and the focus is stored as a **Star Layer**. These **Layers** have already been turned around with the head of the sequence being located next to the element **here**. The other half of the path, between **here** and the root of the graph, is stored as a **Rats Reyal**. These **Reyals** have yet to be turned into **Layers**. Again, the head of the sequence is located next to the element **here**. The operation **toLayers** transforms **Reyals** one-by-one into **Layers** and moves the element **here** closer to the root with every step. Recall the intuition of zippers acting like cursors inside a graph structure, ready to access any neighbouring elements quickly. The intermediate state in the **toLayers** operation is precisely implementing this cursor structure. From the position **this** both the head **Reyal** and the head **Layer** are located just next to it. At each call to the function, the head **Reyal** is moved to the other side of **here** and forms the new head **Layer**.

During the entire operation we rely on the fact that every layer and reyal is connected to its neighbours via an edge. This is ensured by the connectivity properties **ConReyals** and **ConLayers** which we can easily translate between.

13:20 A Data Type of Intrinsically Plane Graphs in Agda

At the end of the `toLayers` operation, all `Reyals` have been turned into `Layers` and we are left with the root sector and one overall `Star Layer`. We now compute the clockwise traversal of the entire graph by concatenating the subtree sequences `star` and `rats` from each layer in the correct order. This is achieved by the function `layersToSteps` which returns a graph traversal in the form of a `Star Step` ⚙️:

```
layersToSteps : (ss : Star Layer (zs ⟨ edge ⟩ as) (ms ⟨ corner ⟩ ms)) → ConLayers ss
              → Star Step (as , corner) (zs , edge)
layersToSteps (layer v rats this star , - []) con = rats ⟨*⟩ (corner this , - star)
layersToSteps (layer v rats this star , - ss@(_ , - _)) (refl , con)
              = let st' = layersToSteps ss con in rats ⟨*⟩ (span this v st' , - star)
```

This function is defined for a non-empty `Star Layer` only which is encoded by the fact that its index types go from an `edge` to a `corner`. The base case covers a single layer, positioned at the very bottom of the path, precisely where the focussed corner is located. In this case we can “chips” the subtrees together, inserting the focus (`corner this`) in the middle.

At any layer further towards the root, `layersToSteps` concatenates the subtrees in a clockwise manner: the right hand side `rats` is added to the beginning of the `Star Step` and the left hand side `star` is added to its end. For this case, the property that the layers are connect with each other, expressed by `ConStar ss`, is crucial. The recursive call provides a traversal of the `Star Layer` structure except of its the topmost layer, as a `Star Step`. This traversal will be added as a `spanned` subtree in the overall result, requiring an edge via which it can be reached. The edge has to be located in between the current layer and the rest of the layer structure. This is precisely the property that is expressed by the connectivity of the `Star Layer`. We match on the proof `ConLayers ss` (which becomes `refl`) and hereby constraining `this` to be an `edge`. Therefore the expression `span this` on the right hand side of the definition is well formed.

Together, the operations `toLayers` and `layerToSteps` provide the functionality needed to turn a zipper into a graph, rooted at its original root. Carefully incorporating planarity as an intrinsic property of `Graphs` now pays back, as the proof of the following property is trivial.

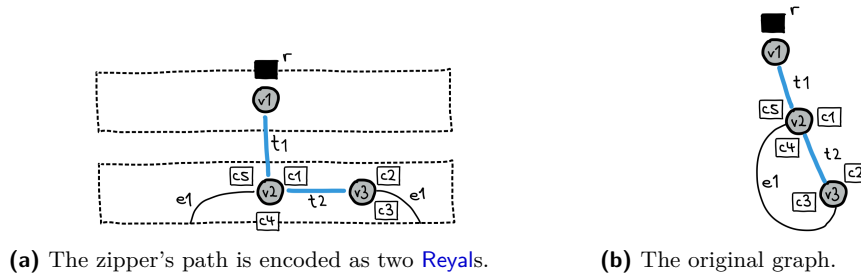
► **Proposition 32.** *Calculating the original graph using `starToGraph` and `toLayers` from a zipper returns a plane graph.*

► **Example 33.** The original tree of the zipper from Example 30 is calculated by the following function and depicted in Figure 20 ⚙️.

```
reyls-oldTree : ziToOldRoot zi-reyls
              ≡ tree r v1 (span t1 v2 (corner c1 , - span t2 v3
              (corner c2 , - push e1 , - corner c3 , - [])) , - corner c4 , - pop e1 , - corner c5 , - [])) , - [])
```

5 Rewriting

Zipper provide the structure to focus on a specific substructure inside a graph. For the goal of implementing diagram rewriting, this separation of a substructure from its context is an important step. A rewrite step then replaces the subgraph in focus with a new graph. In this section, we sketch a rewrite operation for our notion of zipper.

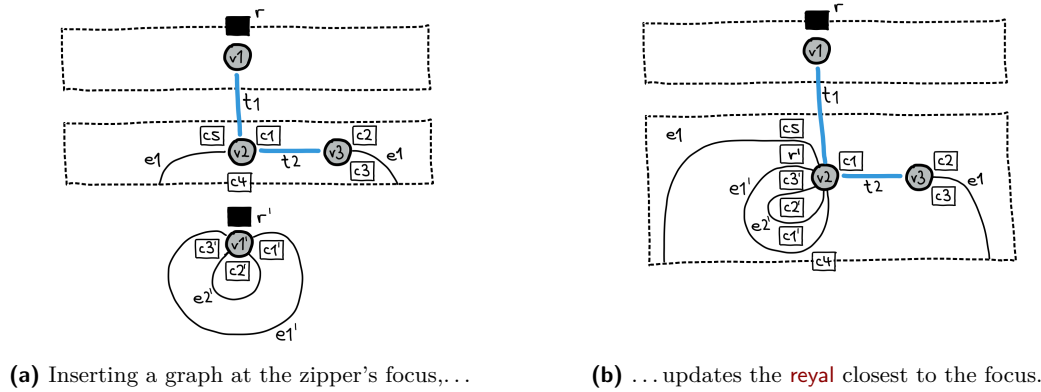


■ **Figure 20** Example of recalculating the original graph from a zipper.

As our specification of zipper focusses on a single corner only, a rewrite step inserts an entire subgraph at the position of the focussed corner. The operation amounts to placing the subgraph into an enclosed face of the context graph, potentially connecting some looping edges to other vertices in the graph without introducing any crossings. As the corner in focus itself has no influence of the edge stack, we can rewrite it by any graph which takes an empty stack to an empty stack. The rewrite operation inserts the new graph at the vertex where the corner in focus is located, positioned right next to it $\star$:

`rewriteFocus` : $\{ es : \text{List } E \} \rightarrow \text{Zipper } es \rightarrow \text{Graph} \rightarrow \text{Zipper } es$

We will omit the implementation details of `rewriteFocus` and refer to the additional material. Instead, let us look at an example.



■ **Figure 21** Example of inserting a graph at the focus of a zipper.

► **Example 34.** As an example, we use the zipper from Example 30 and insert the graph with two self-loops from Example 19 at the position of the focus $\star$. The graphs involved and the result of the operation are shown in Figure 21. Observe that the graph is inserted into the Reyal of the zipper closest to the focus c_4 . Just to the left of the corner in focus we now have the two self-loops from the input graph. This example shows that the insertion merges the two vertices v_2 and v_1' .

```
re-example : rewriteFocus (zi-reyls v1 v2 v3 t1 t2 e1 r c1 c2 c3 c4 c5)
              (exBouquet r' v1' e1' e2' c1' c2' c3')
≡ reyls (reyal v1 r [] [])
  ([ [], reyal v2 t1 (push e1' , - corner c1' , - push e2' , - corner c2' , - pop e2' , - corner c3'
                    , - pop e1' , - corner r' , - pop e1 , - corner c5 , - [])
    ([ [], corner c1 , span t2 v3 ([ [], corner c2 , - , push e1 , - , corner c3])) ) c4
```


6 Future Work

The work we present here discusses important design decisions in the implementation of plane graphs in Agda. The focussing operation is a crucial step towards formalising a full graph rewriting. The development of such an automated graph rewriter poses the following challenges which we would like to tackle as next steps in the project:

- Rewriting larger graphs: So far, our implementation covers the case of zippers focussing on a single corner. For a more general notion of graph rewriting, we are interested in focussing on and replacing larger subgraphs. We observe: As the `Reyal` structure grows bottom-up, we already have easy access to the region of the graph embedding immediately surrounding the focus. With every `Reyal` away from the focus, this region grows, but it always forms a disc-like part of the surface. Therefore it can first be removed without invalidating the planarity property and subsequently be rewritten by concatenating a number of `Reyals` to the front of the remaining path. The indexing type `ZipTy` ensures that both old and new graph change the edge stack in the same way.
- Finding a graph match: Our implementation assumes a given graph match by providing a certain zipper focussed on the root where a rewrite rule may be applied. For implementing a full rewriter we need an operation which matches the left hand side of a rewrite rule to a subgraph inside a `Graph`. In particular, this operation needs to compute a zipper structure such that some number of `Reyals` around the focus exactly match the left hand side of a rule. This is highly non-trivial as not even the choice of spanning tree is known in the beginning (c.f. Remark 7). One possible solution is to compute a spanning tree via a simple algorithm like a depth-first approach, and then changing it to match a given graph, one edge at a time. We can imagine an function which forces a looping edge to become a spanning edge, recomputing which edge will then has to become looping in order to keep the non-cyclic property. Crucially, any function which manipulates a `Graph`, whether by changing the spanning tree, or replacing a subgraph, has to preserve the planarity property.

As additional future work, we would like to extend the framework to incorporate equivalence classes of spanning trees (cf. Remark 7) instead of distinguishing different choices of spanning tree for the same graph. We have already done some work towards this goal by implementing a function `ziToNewRoot`  which takes a zipper and *re-roots* its spanning tree to the corner in focus.

Lastly, we are interested in graphs embedded into surfaces of a higher genus than the plane. In Lemma 11 we discussed that the structure of the looping edges alone determining the genus of the surface a graph can be embedded in. If this structure is a stack, then the graph embedding is plane (cf. Theorem 20). We would like to explore which structure can accommodate higher genus surface embeddings. For example, which data structure implements the graph shown in Figure 1b?

References

- 1 Marie Albenque and Dominique Poulalhon. A generic method for bijections between blossoming trees and planar maps. *Electron. J. Comb.*, 22:2, 2015. doi:10.37236/3386.
- 2 Malin Altenmüller. *Combinatorial Presentations of String Diagrams for Non-Symmetric Monoidal Categories*. PhD thesis, University of Strathclyde, 2025. URL: <https://maltenmuller.github.io/thesis/thesis.pdf>.
- 3 Malin Altenmüller and Ross Duncan. A category of surface-embedded graphs. In Jade Master and Martha Lewis, editors, *Proceedings Fifth International Conference on Applied Category Theory, ACT 2022, Glasgow, United Kingdom, 18-22 July 2022*, volume 380, pages 41–62, 2022. doi:10.4204/EPTCS.380.3.

- 4 Olivier Bernardi. Bijective counting of tree-rooted maps and shuffles of parenthesis systems. *Electronic Journal of Combinatorics*, 14:R9, 2007. doi:10.37236/928.
- 5 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004. doi:10.1007/978-3-662-07964-5.
- 6 Richard S Bird. On building cyclic and shared structures in haskell. *Formal Aspects Comput.*, 24:609–621, 2012. doi:10.1007/S00165-012-0243-6.
- 7 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. Rewriting modulo symmetric monoidal structure. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 710–719. ACM, 2016. doi:10.1145/2933575.2935316.
- 8 Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13:43016, 2011.
- 9 Lucas Dixon and Aleks Kissinger. Open-graphs and monoidal theories. *Math. Struct. Comput. Sci.*, 23:308–359, 2013. doi:10.1017/S0960129512000138.
- 10 Ross Duncan. *Types for quantum computing*. PhD thesis, University of Oxford, UK, 2006. URL: <http://ora.ox.ac.uk/objects/uuid:c2901ae8-9386-4dbf-879d-e37bbc2692bd>.
- 11 Jack R Edmonds. A combinatorial representation for oriented polyhedral surfaces. *Notices of the American Mathematical Society*, 7:646, 1960.
- 12 Martin Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.*, 11:467–492, 2001. doi:10.1017/S0956796801004075.
- 13 Neil Ghani, Makoto Hamana, Tarmo Uustalu, and Varmo Vene. Representing cyclic structures as nested datatypes. In *Proc. of 7th Symp. on Trends in Functional Programming, TFP*, volume 2006, 2006.
- 14 Georges Gonthier. Formal proof—the four-color theorem. *Notices of the AMS*, 55, 2008.
- 15 Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Dongming Wang and Zeng Zhibin, editors, *Computer Mathematics, 8th Asian Symposium, ASCM 2007*, volume 5081, pages 21–22. Springer, 2008. doi:10.1007/978-3-540-87827-8_3.
- 16 Jonathan L Gross and Thomas W Tucker. *Topological Graph Theory*. Dover Publications, 2001. Original work published 1987 by Wiley-Interscience.
- 17 Adam Gundry, Conor McBride, and James McKinna. Type inference in context. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 43–54. ACM, 2010. doi:10.1145/1863597.1863608.
- 18 Makoto Hamana. Initial algebra semantics for cyclic sharing structures. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, volume 5608, pages 127–141. Springer, 2009. doi:10.1007/978-3-642-02273-9_11.
- 19 Lothar Heffter. Über das problem der nachbargebiete. *Mathematische Annalen*, 38:477–508, 1891.
- 20 Gérard Huet. The zipper. *Journal of Functional Programming*, 7:549–554, 1997. doi:10.1017/S0956796897002864.
- 21 David J King and John Launchbury. Lazy depth-first search and linear graph algorithms in haskell. *Gla*, 1993.
- 22 Aleks Kissinger and Vladimir Zamdzhiev. Equational reasoning with context-free families of string diagrams. In Francesco Parisi-Presicce and Bernhard Westfechtel, editors, *Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 21-23, 2015. Proceedings*, volume 9151, pages 138–154. Springer, 2015. doi:10.1007/978-3-319-21145-9_9.

- 23 Stefan Milius, Lutz Schröder, and Thorsten Wißmann. Regular behaviours with names - on rational fixpoints of endofunctors on nominal sets. *Appl. Categorical Struct.*, 24:663–701, 2016. doi:10.1007/S10485-016-9457-8.
- 24 Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. Johns Hopkins University Press, 2001. URL: <http://jhupbooks.press.jhu.edu/ecom/MasterServlet/GetItemDetailsHandler?iN=9780801866890&qty=1&source=2&viewMode=3&loggedIN=false&JavaScript=y>.
- 25 Andrey Mokhov. Algebraic graphs with class (functional pearl). In Iavor S Diatchki, editor, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 2–13. ACM, 2017. doi:10.1145/3122955.3122956.
- 26 Peter Selinger. A survey of graphical languages for monoidal categories, 2011. doi:10.1007/978-3-642-12821-9_4.
- 27 Noam Zeilberger and Alain Giorgetti. A correspondence between rooted planar maps and normal planar lambda terms. *Log. Methods Comput. Sci.*, 11, 2015. doi:10.2168/LMCS-11(3:22)2015.

Functional Representability in Local Set Theories

Enrique Ruiz Hernández  

Centro de Investigación en Teoría de Categorías y sus Aplicaciones, A.C, Mexico

Pedro Solórzano  

Instituto de Matemáticas, Universidad Nacional Autónoma de México, Oaxaca de Juárez, Mexico

Abstract

In general local set theories (LST), there is an incomplete representability as $(x \mapsto \tau)$ for syntactic functions, i.e. those given by descriptions of the form $\forall x \exists ! y \gamma$, for some formula γ . The well-known equivalence theorem establishes among other things that it is possible to produce a well-termed local set theory that extends any given LST. However, the natural translation between the corresponding languages still does not guarantee strict functional representability, but only up to isomorphisms.

In this report, natural isomorphisms that are compatible with reasoning internally in the original LST are made explicit and explored in more generality. The internal translation along them is exhibited to be compatible with all logical operations; syntactic functions naturally represent themselves in a robust way as described in the main results.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Higher order logic; Theory of computation $\rightarrow$ Type theory

Keywords and phrases local set theories, function symbols, categorical logic

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.14

Related Version *Full Version*: <https://arxiv.org/abs/2104.07405> [9]

Funding The second named author is supported by the SECIHTI “Investigadoras e Investigadores por México” Program Project No. 61.

Pedro Solórzano: SECIHTI-UNAM Research fellow.

Acknowledgements The authors wish to thank the several anonymous referees for greatly helping improve the presentation of the main results.

1 Introduction

As part of the ongoing development of an axiomatic framework for Synthetic Differential Geometry (SDG), McLarty [7] introduced a list of axioms a topos should satisfy to be a good model for SDG. In [10], the authors of this report added an axiom that further describes such a model in terms of the program of Axiomatic Cohesion of Lawvere [6]. In the standard textbooks in the area (see the textbook by Kock [3], Lavendhomme [5], or McLarty [8]), internal reasoning is only indirectly present. One aspect that is hardly mentioned is that of the representability of functions described implicitly within the internal language of a topos, like the ones that commonly occur in the context of SDG. A function f is *represented* if there is a term τ_f in the internal language for which f coincides with

$$(x \mapsto \tau_f), \tag{1}$$

where x is a sequence of variables of the appropriate type.

To set things up, a local language and a local set theory will be considered. A local language is a typed language $\mathcal{L}$ with primitives $\in$, $\{ : \}$ and $=$; ground, product and power type symbols; distinguished type symbols $\mathbf{1}, \Omega$; and function symbols; together with a natural intuitionistic deduction system. A local set theory (LST) is a collection S of sequents $\Gamma : \alpha$,



© Enrique Ruiz Hernández and Pedro Solórzano;
licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 14; pp. 14:1–14:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

14:2 Functional Representability in Local Set Theories

closed under derivability. Thus, the resulting logic of an LST might still not be classical. A local set theory S produces a category $\mathcal{T}(S)$ with its syntactical sets and functions (called S -sets and S -functions) as in classical Set Theory. Said categories are also toposes.

Conversely, starting with a topos, a local language can be produced – e.g. Mitchell-Bénabou, Zangwill, Lambek-Scott (See [2], [1], or [4]) – whose linguistic topos is equivalent to the starting topos.

There is a lack of representability in general for syntactic functions in local set theories, since a function f might be given simply in terms of its graph $|f|$; without a term τ_f to describe it as in (1). For any such function $f : X \rightarrow Y$ – and thus for finitely many –, this can be resolved by changing both the language and the LST by adding a function symbol $\mathbf{f}$ and the corresponding axioms to guarantee that the symbolic function $(x \mapsto \mathbf{f}(x))$ coincides with the given function, when restricted to X , i.e. by adding

$$x \in X : \langle x, \mathbf{f}(x) \rangle \in |f|. \quad (2)$$

This procedure is a basic feature of everyday mathematical practice, yet it formally requires a change of topos every time a new function symbol is added.

The main purpose of this report is to describe a way to deal with this lack of representability for all syntactic functions, by way of suitable reparameterizations of the domain and codomain, in such a way that any expression in the internal language can be effectively translated.

To this effect, recall that the Equivalence Theorem can be proved as follows. For any topos $\mathcal{E}$, denote by $\text{Th}(\mathcal{E})$ the LST associated to it: its type and function symbols are the objects and arrows of $\mathcal{E}$, which are written in bold, i.e. $\mathbf{X}$ and $\mathbf{f}$ for the type symbol associated to the object X of $\mathcal{E}$ or the function symbol associated to the arrow f of $\mathcal{E}$. Alternatively, $\mathbf{Symb}(X)$ and $\mathbf{Symb}(f)$ may be used to avoid confusion.

There is a *canonical inclusion* $\iota_{\mathcal{E}} : \mathcal{E} \rightarrow \mathcal{T}(\text{Th}(\mathcal{E}))$, given by

$$\iota_{\mathcal{E}} X = U_{\mathbf{X}} := \{u_{\mathbf{X}} : \top\} \quad \text{and} \quad \iota_{\mathcal{E}} f = (x \mapsto \mathbf{f}(x)) : U_{\mathbf{X}} \rightarrow U_{\mathbf{Y}}, \quad (3)$$

where $U_{\mathbf{X}}$ is the universal set of type $\mathbf{X}$. The Equivalence Theorem (see Proposition II.13.3 in [4]) shows that for any topos $\mathcal{E}$, $\iota_{\mathcal{E}}$ is an equivalence of categories.

Now, a translation from a local set theory S to another one Σ is an assignment (with certain requirements) of a type and a function symbol in Σ for each in S with the evident assignment of the terms in S to terms in Σ of the corresponding types. For any S there is the *natural translation* $\eta_S : S \rightarrow \text{Th}(\mathcal{T}(S))$ which sends every type symbol $\mathbf{A}$ of S to $\mathbf{Symb}(U_{\mathbf{A}})$, the type symbol corresponding to the universal set $U_{\mathbf{A}}$, and every function symbol $\mathbf{f}$ to $\mathbf{Symb}((x \mapsto \mathbf{f}(x)))$, the function symbol corresponding to $(x \mapsto \mathbf{f}(x))$.

On the other hand, any translation ζ induces a logical functor $\mathcal{T}(\zeta)$ between the corresponding linguistic toposes. In particular, $\mathcal{T}(\eta_S)$ is given as follows:

$$\mathcal{T}(\eta_S)(X) = \{x_{\eta_S(\mathbf{A})} : \eta_S(\alpha)\} \quad \text{and} \quad \mathcal{T}(\eta_S)(f) = (\eta_S(|f|), \eta_S(X), \eta_S(Y)), \quad (4)$$

where $X = \{x_{\mathbf{A}} : \alpha\}$ and $f = (|f|, X, Y)$ with $|f| \subseteq X \times Y$ the graph of f .

By (3) and (4), one has two distinct functors:

$$\iota_{\mathcal{T}(S)}, \mathcal{T}(\eta_S) : \mathcal{T}(S) \rightarrow \mathcal{T}(\text{Th}(\mathcal{T}(S))). \quad (5)$$

Notice that the only way for $\mathcal{T}(\eta_S)(X)$ to coincide with $\iota_{\mathcal{T}(S)}X$ is for X to be $U_{\mathbf{A}}$. Likewise, the only way for $\mathcal{T}(\eta_S)(f)$ to coincide with $\iota_{\mathcal{T}(S)}f$ is for f to be of the form $(x \mapsto \mathbf{f}(x))$ for some function symbol of S . Otherwise, these two functors are very much distinct.

Within the proof of the Equivalence Theorem in [1, p.110], a quasi-inverse to ι ,

$$\rho_{\mathcal{E}} : \mathcal{T}(\text{Th}(\mathcal{E})) \rightarrow \mathcal{E}, \quad (6)$$

is constructed as follows: For objects $X = \{x_{\mathbf{A}} : \alpha\}$, $Y = \{y_{\mathbf{B}} : \beta\}$ in $\mathcal{T}(\text{Th}(\mathcal{E}))$, let i_X and i_Y be the inclusions $(x_{\mathbf{A}} \mapsto x_{\mathbf{A}}) : X \rightarrow U_{\mathbf{A}}$ and $(y_{\mathbf{B}} \mapsto y_{\mathbf{B}}) : Y \rightarrow U_{\mathbf{B}}$, respectively. Now, let $\rho_{\mathcal{E}} X$ be a corresponding subobject – frequently called *kernel* – of the natural interpretation $\llbracket \alpha \rrbracket_x$ of α in $\mathcal{E}$. For $X \xrightarrow{\varphi} Y$ in $\mathcal{T}(\text{Th}(\mathcal{E}))$, $\rho_{\mathcal{E}} \varphi : \rho_{\mathcal{E}} X \rightarrow \rho_{\mathcal{E}} Y$ is the unique ψ such that

$$\vdash_{\text{Th}(\mathcal{E})} \langle i_X(u), i_Y(\psi(u)) \rangle \in |\varphi|. \quad (7)$$

This derivation shows precisely the extent by which achieving strict representability (as the one obtained from (2)) fails: The representation of φ as an explicit function $(x \mapsto \tau)$ is only fulfilled up to isomorphism. In the case when $\mathcal{E}$ is $\mathcal{T}(S)$ for a local set theory S and φ is $\mathcal{T}(\eta_S)f$, more can be said.

► **Theorem A.** *Let $X = \{x_{\mathbf{A}} : \alpha\}$ and $Y = \{y_{\mathbf{B}} : \beta\}$ be objects in $\mathcal{T}(S)$ and let i_X and i_Y be the inclusions $(x_{\mathbf{A}} \mapsto x_{\mathbf{A}}) : X \rightarrow U_{\mathbf{A}}$ and $(y_{\mathbf{B}} \mapsto y_{\mathbf{B}}) : Y \rightarrow U_{\mathbf{B}}$, respectively. Then,*

$$\vdash_{\text{Th}(\mathcal{T}(S))} \langle i_X(u), i_Y(\text{Symb}(f)(u)) \rangle \in |\mathcal{T}(\eta_S)f|. \quad (8)$$

That is, f itself is the only syntactic function for which the following diagram commutes in $\mathcal{T}(\text{Th}(\mathcal{T}(S)))$:

$$\begin{array}{ccc} U_X & \xrightarrow{r_X} & \mathcal{T}(\eta_S)X \\ (x \mapsto \text{Symb}(f)(x)) \downarrow & & \downarrow \mathcal{T}(\eta_S)f \\ U_Y & \xrightarrow{r_Y} & \mathcal{T}(\eta_S)Y, \end{array} \quad (9)$$

where r_X is the corestriction of $(u \mapsto i_X(u))$ to $\mathcal{T}(\eta_S)X$.

Moreover, the inclusion functor $\iota_{\mathcal{T}(S)}$ and the logical functor induced by the canonical translation η_S are isomorphic:

$$\mathcal{T}(\eta_S) \cong \iota_{\mathcal{T}(S)}. \quad (10)$$

The way local set theories are defined makes their associated toposes have canonical subobjects. In this context, there is an adjunction $\mathcal{T} \dashv \text{Th}$ where η is its unit; the counit is the appropriate restriction, ε , of ρ . By Lemma II.15.3 in [4], one has that $\varepsilon \circ \iota = 1$. From this it follows – as pointed out to us by a kind referee – , the conclusion of (10). However, such proof does not yield (8), which we believe sheds light onto the lack of representability of syntactic functions even in well-termed local set theories.

In order to take advantage of the isomorphisms r_X , the next goal is to describe an internal translation of formulae with variables restricted to X into formulae with unrestricted variables, i.e. whose range is a universal set. The Equivalence Theorem establishes that any object in $\mathcal{T}(\text{Th}(\mathcal{T}(S)))$ is isomorphic to some universal set. The eliminability of descriptions in $\text{Th}\mathcal{T}(S)$ – which is called *well-termedness* (see the last paragraph of Subsection 2.4) – gives that any function f with universal domain $U_{\mathbf{A}}$ in $\mathcal{T}(\text{Th}(\mathcal{T}(S)))$ can indeed be represented by $(x \mapsto \tau_f)$ for some appropriate term τ_f (see Proposition 15). The guaranteed term τ_f is a term in the language of $\text{Th}\mathcal{T}(S)$ which need not be of the form $\eta_S(\sigma)$ for an appropriate term σ in the language of S .

The following result describes the desired translation of formulae along functions with arbitrary codomain but universal domain. This will be applied to r_X , r_Y , etc. correspondingly.

14:4 Functional Representability in Local Set Theories

In preparation for this, for any syntactic function $f : X \rightarrow Y$ in a local set theory S in a language $\mathcal{L}$ and for any formula θ in $\mathcal{L}$, let $\theta_f^{\natural}$ denote¹

$$\exists y(\langle x, y \rangle \in |f| \wedge \theta).$$

If Γ is a finite set of formulas, then $\Gamma_f^{\natural}$ is the finite set $\{\gamma_f^{\natural} : \gamma \in \Gamma\}$.

► **Theorem B.** *Let Σ be a well-termed local set theory in a local language $\mathcal{L}$. Let $\mathbf{A}$ and $\mathbf{B}$ be types in $\mathcal{L}$. Let $f : U_{\mathbf{B}} \rightarrow X$ be a surjective Σ -function with X determined by the formula α with unique variable of type $\mathbf{A}$, and τ_f a term that represents f . Let γ and Γ be a formula and a finite set of formulae in $\mathcal{L}$. Then,*

$$\Gamma, x \in X \vdash_{\Sigma} \gamma \quad \text{if and only if} \quad \Gamma(x/\tau_f) \vdash_{\Sigma} \gamma(x/\tau_f).$$

Additionally, if f is also injective, then, for every formula ϑ ,

$$\vdash_{\Sigma} \vartheta = \vartheta_{f^{-1}}^{\natural}(x/\tau_f),$$

In this case,

$$\Gamma \vdash_{\Sigma} \gamma \quad \text{if and only if} \quad \Gamma_{f^{-1}}^{\natural}, x \in X \vdash_{\Sigma} \gamma_{f^{-1}}^{\natural},$$

Furthermore, $(-)^{\natural}_{f^{-1}}$ is coherent with all logical operations.

The proof of Theorem B is the content of Theorems 16 and 18. The coherence result is explained in detail in the latter. When applied to $\Sigma = \text{Th}(\mathcal{T}(S))$, Theorem B provides an explicit natural parameterization of any $\mathcal{T}\eta_S(X)$ by the universal $\iota_{\mathcal{T}(S)}(X)$ preserving all logical operations.

The next result shows that the parameterizations given by the r_X , r_Y , etc. behave nicely with respect to syntactic functions: If one starts with an S -function, its graph can be expressed as $\{\langle x, y \rangle : \gamma\}$ for some appropriate formula γ . By (4), the formula $\mathcal{T}(\eta_S)\gamma$ determines the function $\mathcal{T}(\eta_S)f$ in $\text{Th}(\mathcal{T}(S))$, that is $\{\langle x, y \rangle : \mathcal{T}(\eta_S)\gamma\}$ is the graph of $\mathcal{T}(\eta_S)f$.

On the other hand, $\iota_{\mathcal{T}(S)}(f)$ is the function $(u \mapsto \mathbf{Symb}(f)(u)) : U_{\mathbf{X}} \rightarrow U_{\mathbf{Y}}$. By considering the term

$$\tilde{\gamma} \equiv \mathcal{T}(\eta_S)\gamma(x/\mathbf{i}_{\mathbf{X}}(u), y/\mathbf{i}_{\mathbf{Y}}(v)), \quad (11)$$

it remains to be verified that $\tilde{\gamma}$ also determines a function from $U_{\mathbf{X}}$ to $U_{\mathbf{Y}}$, i.e.

$$\vdash_{\text{Th}(\mathcal{T}(S))} \forall u \exists! v. \tilde{\gamma}.$$

One would also expect these two functions to agree. The next result verifies both claims. This further allows one to come back to (2) to settle the representability question up to suitable isomorphisms.

► **Theorem C.** *Let S be a local set theory in a local language $\mathcal{L}$. Let $f : X \rightarrow Y$ be an S -function, and let γ be a formula that determines $|f|$. Then, the graph of the function $(u \mapsto \mathbf{f}(u))$ is determined by $\tilde{\gamma}$ of (11). More precisely,*

$$\vdash_{\text{Th}(\mathcal{T}(S))} (\mathcal{T}(\eta_S)\gamma(x/\mathbf{i}_{\mathbf{X}}(u), y/\mathbf{i}_{\mathbf{Y}}(v)))_{r_X^{-1} \times r_Y^{-1}}^{\natural} = \mathcal{T}(\eta_S)\gamma, \quad (12)$$

and the formula $\mathcal{T}(\eta_S)\gamma(x/\mathbf{i}_{\mathbf{X}}(u), y/\mathbf{i}_{\mathbf{Y}}(v))$ indeed determines a function $f^* : U_{\mathbf{X}} \rightarrow U_{\mathbf{Y}}$ which furthermore coincides with $(u \mapsto \mathbf{f}(u))$. In other words,

$$\vdash_{\text{Th}(\mathcal{T}(S))} \langle u, \mathbf{Symb}(f)(u) \rangle \in |f^*|. \quad (13)$$

¹ In Definition 10, it is defined differently, yet in Lemma 12 it is proved to be equivalent to this.

The representability thus described allows the user of a local set theory to treat all functions as represented. This gives a far more intuitive way to use the local language in reasoning about objects in toposes, which in turn opens the door to studying axiomatic systems for Synthetic Differential Geometry (or other mathematical theories) while reasoning about them directly in a suitable set theoretical way – albeit with a more general intuitionistic deductive system.

The structure of this report is as follows. In Section 2, we review the necessary notions as well as set the notation. In Section 3, the notion of logical parameterization is investigated in order to prove Theorem B. Lastly, in Section 4, Theorems A and C are proved. The latter is obtained by applying the techniques developed in Section 3.

2 Notations and conventions

This lengthy section serves the purpose of introducing the notation to be used in the sequel. It is essentially a compendium of the results from the textbook of Bell [1]. The expert is compelled to skip it and proceed directly to Section 3 only to come back for clarification. The notation mostly follows that of the Bell's textbook. One important exception is given by $(-)^{\natural}$ in (19) below. Another exception is that graphs of functions f are denoted by $|f|$ as in the textbook of Lambek and Scott [4].

2.1 Toposes

An elementary topos – henceforth simply *topos* – is a category $\mathcal{C}$ with finite products, a subobject classifier and power objects. A subobject of an object A is an equivalence class $[m]$ of monic maps m with codomain A under an appropriate relation (see page 49 in [1]). A subobject classifier is an object Ω together with a map $\top : 1 \rightarrow \Omega$ such that for any monic arrow m there exists a unique characteristic $\chi(m)$ such that the following is a pullback diagram.

$$\begin{array}{ccc} \text{dom}(m) & \longrightarrow & 1 \\ m \downarrow & & \downarrow \top \\ \text{cod}(m) & \xrightarrow{\chi(m)} & \Omega \end{array}$$

And every $A \rightarrow \Omega$ is of this form.

Given an object A of $\mathcal{C}$, a power object is a pair (PA, e_A) of the object PA and an arrow $e_A : A \times PA \rightarrow \Omega$ such that for every $f : A \times B \rightarrow \Omega$ there is a unique arrow $\hat{f} : B \rightarrow PA$ such that the following diagram commutes.

$$\begin{array}{ccc} A \times B & & \\ 1_A \times \hat{f} \downarrow & \searrow f & \\ A \times PA & \xrightarrow{e_A} & \Omega. \end{array} \tag{14}$$

In a topos $\mathcal{C}$, given an object A , there is a bijection between the class $\text{Sub}(A)$ of subobjects of A and the class $\mathcal{C}(A, \Omega)$ of arrows $A \rightarrow \Omega$ in $\mathcal{C}$. Given a monic arrow m representing a subobject of A , its corresponding characteristic is $\chi(m) : A \rightarrow \Omega$; and given an arrow $u : A \rightarrow \Omega$ in $\mathcal{C}$, its corresponding subobject is represented by $\bar{u}$. It follows that $u = \chi(\bar{u})$; $[\chi(\bar{m})] = [m]$; and that $[m] = [n]$ if and only if $\chi(m) = \chi(n)$.

14:6 Functional Representability in Local Set Theories

From this, one can transfer the partial order $\subseteq$ unto $\mathcal{C}(A, \Omega)$, so that given $u, v \in \mathcal{C}(A, \Omega)$, $u \leq v$ if and only if $\bar{u} \subseteq \bar{v}$. In particular, this is equivalent to the existence of an arrow $f : \text{dom}(\bar{u}) \rightarrow \text{dom}(\bar{v})$ such that

$$\bar{u} = \bar{v} \circ f. \quad (15)$$

Recall that the maximal characteristic arrow of an object A , denoted by T_A , is $T_A := \chi(1_A) : A \rightarrow \Omega$. Wherever A is understood, one writes simply T . Notice that $\bar{T}_A = 1_A$, and that T_A is maximal since $u \leq T_A$ for all $u : A \rightarrow \Omega$. Furthermore, recall that $T = v \circ \bar{u}$ if and only if $u \leq v$ and that

$$T_C = v \circ f \text{ if and only if a } g : C \rightarrow \text{dom}(\bar{v}) \text{ exists with } f = \bar{v} \circ g. \quad (16)$$

2.2 Local languages

The central definition of this communication is the definition of a local language. A *local language* $\mathcal{L}$ is determined by the specification of the following classes of symbols: a truth-value symbol Ω and a unity type symbol $\mathbf{1}$; a possibly empty collection of ground type symbols $\mathbf{A}, \mathbf{B}, \mathbf{C}, \dots$; a possibly empty collection of function symbols $\mathbf{f}, \mathbf{g}, \mathbf{h}, \dots$. The type symbols of $\mathcal{L}$ are now defined recursively as follows: $\mathbf{1}, \Omega$ are type symbols; any ground type symbol is a type symbol; if $\mathbf{A}_1, \dots, \mathbf{A}_n$ are type symbols, so is the product $\mathbf{A}_1 \times \dots \times \mathbf{A}_n$ (with the proviso that, if $n = 1$, then $\mathbf{A}_1 \times \dots \times \mathbf{A}_n$ is $\mathbf{A}_1$, and if $n = 0$, then $\mathbf{A}_1 \times \dots \times \mathbf{A}_n$ is $\mathbf{1}$); and if $\mathbf{A}$ is a type symbol, so is the power $\mathbf{P}\mathbf{A}$.

For each type symbol $\mathbf{A}$, $\mathcal{L}$ contains a denumerable set of symbols $x_{\mathbf{A}}, y_{\mathbf{A}}, z_{\mathbf{A}}, \dots$ called *variables* of type $\mathbf{A}$. In addition, $\mathcal{L}$ contains the symbol $*$. Each function symbol of $\mathcal{L}$ has a signature assigned of the form $\mathbf{A} \rightarrow \mathbf{B}$, where $\mathbf{A}, \mathbf{B}$ are type symbols.

The terms with their corresponding types are recursively defined in the usual way (see Chapter 3 in [1]). In particular, there are primitive symbols $=, \in, \{:\}$ which yield terms $\sigma = \tau, \sigma \in \beta, \{x : \alpha\}$, where σ and τ are terms of the same type, β is of the power type of σ and x may or may not be a free variable of α .

Also, $*$ is a term of type $\mathbf{1}$ and there are further auxiliary symbols $()_i$ and $\langle, \rangle$ to deal with finite products. A term of type Ω is called a *formula*. Logical operations are also defined recursively (see p. 70 in [1]): Write

$\alpha = \beta$	for	$\alpha \Leftrightarrow \beta$	$\forall \omega. \omega$	for	$\perp$
$* = *$	for	$\top$	$\alpha \Rightarrow \perp$	for	$\neg \alpha$
$\langle \alpha, \beta \rangle = \langle \top, \top \rangle$	for	$\alpha \wedge \beta$	$\forall \omega ((\alpha \Rightarrow \omega \wedge \beta \Rightarrow \omega) \Rightarrow \omega)$	for	$\alpha \vee \beta$
$(\alpha \wedge \beta) \Leftrightarrow \alpha$	for	$\alpha \Rightarrow \beta$	$\forall \omega (\forall x (\alpha \Rightarrow \omega) \Rightarrow \omega)$	for	$\exists x \alpha$
$\{x : \alpha\} = \{x : \top\}$	for	$\forall x \alpha$			

The basic axioms, inference rules and thus all deductions in a local set theory are given in terms of *sequents* $\Gamma : \alpha$, where α is a formula and Γ is a (possibly empty) finite set of formulae. The *basic axioms* are the sequents (see page 71 in [1]): $\alpha : \alpha$ (Tautology); $: x_{\mathbf{1}} = *$ (Unity); $x = y, \alpha(z/x) : \alpha(z/y)$ with x, y free for z in α (Equality); $: (\langle x_1, \dots, x_n \rangle)_i = x_i, : x = \langle (x)_1, \dots, (x)_n \rangle$ (Products); and $: x \in \{x : \alpha\} \Leftrightarrow \alpha$ (Comprehension).

The inference rules are the following:

$$\begin{array}{ll}
\textit{Thinning} & \frac{\Gamma : \alpha}{\beta, \Gamma : \alpha} \\
\\
\textit{Cut} & \frac{\Gamma : \alpha \quad \alpha, \Gamma : \beta}{\Gamma : \beta} \text{ (free variables in } \alpha \text{ free in } \Gamma \text{ or } \beta) \\
\\
\textit{Substitution} & \frac{\Gamma : \alpha}{\Gamma(x/\tau) : \alpha(x/\tau)} \text{ } (\tau \text{ free for } x \text{ in } \Gamma \text{ and } \alpha) \\
\\
\textit{Extensionality} & \frac{\Gamma : x \in \sigma \Leftrightarrow x \in \tau}{\Gamma : \sigma = \tau} \text{ } (x \text{ not free in } \Gamma, \sigma, \tau) \\
\\
\textit{Equivalence} & \frac{\alpha, \Gamma : \beta \quad \beta, \Gamma : \alpha}{\Gamma : \alpha \Leftrightarrow \beta}
\end{array}$$

For any collection S of sequents, a *proof from S* is a finite tree with its vertex at the bottom whose nodes are correlated downwards by direct consequence by one of the rules of inference and topmost nodes are correlated with either a basic axiom or a member of S . The sequent at the vertex is the *conclusion* of the proof. A sequent $\Gamma : \alpha$ is *derivable from S* , and write $\Gamma \vdash_S \alpha$ provided there is a proof from S of which the sequent $\Gamma : \alpha$ is the conclusion.

2.3 Local set theories

A *local set theory* (LST) in $\mathcal{L}$ is a collection S of sequents which is closed under derivability: $\Gamma \vdash_S \alpha$ if and only if $(\Gamma : \alpha)$ is in S . Clearly, any collection of sequents generates an LST. A local set theory is said to be consistent if it is not the case that $\vdash_S \perp$.

Closed terms of power type are called $\mathcal{L}$ -sets. Capital letters $A, B, \dots, Z$ will be used to denote $\mathcal{L}$ -sets and the usual abbreviations will be used: $\forall x \in X. \alpha$ for $\forall x(x \in X \Rightarrow \alpha)$; $\exists x \in X. \alpha$ for $\exists x(x \in X \wedge \alpha)$; $\exists! x \in X. \alpha$ for $\exists! x(x \in X \wedge \alpha)$; and $\{x \in X : \alpha\}$ for $\{x : x \in X \wedge \alpha\}$.

The expected set operations $\subseteq, \cap, \cup, \dots$ can be defined in the usual way for sets of the same type. For each type $\mathbf{A}$ there is a universal set $\{x_{\mathbf{A}} : \top\}$ (resp. empty set $\{x_{\mathbf{A}} : \perp\}$) denoted by $U_{\mathbf{A}}$ or A (resp. $\emptyset_{\mathbf{A}}$ or $\emptyset$). For arbitrary sets, products and disjoint unions can be defined in the usual way. Also, function spaces $\{u : u \subseteq Y \times X \wedge \forall y \in Y \exists! x \in X. \langle y, x \rangle \in u\}$ are denoted by X^Y . Lastly, the familiar $\{\tau\}$ and $\{\tau : \alpha\}$ symbolize $\{x : x = \tau\}$ and $\{z : \exists x_1 \dots \exists x_n (z = \tau \wedge \alpha)\}$, respectively (see 3.10 in [1]).

An S -set is an equivalence class of $\mathcal{L}$ -sets under the relation $\sim_S$, given by $X \sim_S Y$ if and only if $\vdash_S X = Y$. Representative $\mathcal{L}$ -sets will sometimes be regarded as S -sets.

An S -function $X \xrightarrow{f} Y$ is a triplet of S -sets $(|f|, X, Y)$ such that $\vdash_S |f| \in Y^X$, where $|f|$ is called the graph of f as per usual.

► **Proposition 1.** *Let $X, Y, |f|$ be S -sets with $\vdash_S |f| \subseteq X \times Y$. Then, $x \in X \vdash_S \exists! y \in Y(\langle x, y \rangle \in |f|)$ if and only if $x \in X \vdash_S \exists y \in Y(\langle x, y \rangle \in |f|)$ and $\langle x, y \rangle \in |f|, \langle x, y' \rangle \in |f| \vdash_S y = y'$.*

Proof. Straightforward. ◀

For S -functions $X \xrightarrow{f} Y \xrightarrow{g} Z$ the composition is defined through its graph. Namely,

$$|g \circ f| = \{\langle x, z \rangle : \exists y(\langle x, y \rangle \in |f| \wedge \langle y, z \rangle \in |g|)\}. \quad (17)$$

A consequence of extensionality is that for S -functions $X \xrightarrow{f} Y, X \xrightarrow{g} Y$,

$$f \text{ is equal to } g \text{ if and only if } x \in X \vdash_S \langle x, y \rangle \in |f| \Leftrightarrow \langle x, y \rangle \in |g|. \quad (18)$$

► Remark 2. It is verified staring in page 85 in [1] that the collection of S -sets and S -functions forms a category $\mathcal{T}(S)$ that is also a topos, the *linguistic topos* of S .

2.4 Representability

Let X and Y be S -sets and τ be a term for which

$$\langle x_1, \dots, x_n \rangle \in X \vdash_S \tau \in Y,$$

with all the free variables of τ contained within the $x_1, \dots, x_n$. It is immediate to see that $\{\langle \langle x_1, \dots, x_n \rangle, \tau \rangle : \langle x_1, \dots, x_n \rangle \in X\}$ is the graph of a function which will be denoted by $\langle \langle x_1, \dots, x_n \rangle \mapsto \tau \rangle$.

Not every S -function is representable in this way. However, for an S -function $f : X \rightarrow \Omega$, let²

$$f^\natural(x) := \langle x, \top \rangle \in |f|. \quad (19)$$

Then, $x \in X \vdash \langle x, f^\natural(x) \rangle \in |f|$, and therefore

► **Proposition 3** ([1], 3.16.4).

- (i) $X \xrightarrow{x \mapsto f^\natural(x)} \Omega$ is f ,
- (ii) $\vdash_S (\langle x_1, \dots, x_n \rangle \mapsto \alpha)^\natural(\langle x_1, \dots, x_n \rangle) \Leftrightarrow \alpha$.

An LST is said to be *well-termed* if whenever $\vdash_S \exists! x \alpha$, there is a term τ such that its free variables are that of α except x and $\vdash_S \alpha(x/\tau)$; an LST for which every syntactic set is isomorphic to a universal is said to be *well-typed*. For a given LST S , let $\mathbf{C}^*(S)$ be the subcategory of S -sets of the form $U_{\mathbf{A}}$ and S -functions of the form $(x \mapsto \tau)$ with τ a term. It follows that S is well-termed if and only if $\mathbf{C}^*(S) \hookrightarrow \mathcal{T}(S)$ is full; and well-typed if and only if it is essentially surjective (see remark in page 113 in [1]).

2.5 Interpretations

An *interpretation* of a local language $\mathcal{L}$ is a pair $(\mathcal{E}, I)$, with $\mathcal{E}$ a topos and I an assignment to each type symbol Ξ of an object $\Xi_I = \Xi$ in $\mathcal{E}$ in a way compatible with products and powers that further assigns to Ω a subobject classifier $\Omega_I = \Omega$; to $\mathbf{1}$ a terminal object $\mathbf{1}_I = \mathbf{1}$; and to a function symbol $\mathbf{f}$ of signature $\mathbf{A} \rightarrow \mathbf{B}$ an arrow in $\mathcal{E}$ of the form $\mathbf{f}_I : \mathbf{A}_I \rightarrow \mathbf{B}_I$.

Given any interpretation, terms are further interpreted as arrows in the usual way. Let τ be a term of type $\mathbf{B}$ and let $\mathbf{x} = x_1, \dots, x_n$ be different variables of types $\mathbf{A}_1, \dots, \mathbf{A}_n$ including all free variables of τ . Define

$$\llbracket \tau \rrbracket_{I, x_1, \dots, x_n} = \llbracket \tau \rrbracket_{I, \mathbf{x}} = \llbracket \tau \rrbracket_{\mathbf{x}}$$

as an arrow $A_1 \times \dots \times A_n \rightarrow B$ recursively (see page 92 in [1]).

In particular, if τ is a term all of whose variables are among $z_1, \dots, z_m$ and $\sigma_1, \dots, \sigma_m$ are terms such that σ_i is free for z_i in τ for each $i = 1, \dots, m$; then, for each interpretation of $\mathcal{L}$ in a topos $\mathcal{E}$,

$$\llbracket \tau(z/\sigma) \rrbracket_{\mathbf{x}} = \llbracket \tau \rrbracket_{\mathbf{z}} \circ \llbracket \langle \sigma_1, \dots, \sigma_m \rangle \rrbracket_{\mathbf{x}} \quad (20)$$

where $\mathbf{x}$ includes all the free variables of $\sigma_1, \dots, \sigma_m$.

² Bell uses the symbol f^* but here it is reserved for the parameterizations of Section 3.

The interpretation of a finite collection Γ of formulae, $\llbracket \Gamma \rrbracket_{I, \mathbf{x}}$, is the interpretation of its conjunction.

Finally, if β is another formula and $\mathbf{x} = x_1, \dots, x_n$ is the list of the free variables of $\Gamma \cup \{\beta\}$, then write

$$\Gamma \models_I \beta \quad \text{or} \quad \Gamma \models_{\mathcal{E}} \beta \quad \text{instead of} \quad \llbracket \Gamma \rrbracket_{I, \mathbf{x}} \leq \llbracket \beta \rrbracket_{I, \mathbf{x}}. \quad (21)$$

A sequent $\Gamma : \beta$ is *valid* under I (or in $\mathcal{E}$) if $\Gamma \models_I \beta$. It is clear that $\models_I \beta$ if and only if $\llbracket \beta \rrbracket_{\mathbf{x}} = T$. From which if $\mathcal{E}$ is degenerate then $\models_I \alpha$ for every formula α in $\mathcal{L}$.

Given a local set theory S , recall that by Remark 2, the linguistic topos has objects the S -sets and arrows the syntactic functions. Now, the canonical interpretation $(\mathcal{T}(S), K_S)$ of its local language is the one that assigns $\mathbf{A}_{K_S} = U_{\mathbf{A}}$ for each type symbol $\mathbf{A}$ and $\mathbf{f}_{K_S} = U_{\mathbf{A}} \xrightarrow{x \mapsto \mathbf{f}(x)} U_{\mathbf{B}}$ for each function symbol $\mathbf{f}$ with signature $\mathbf{A} \rightarrow \mathbf{B}$. It follows that (see 3.28 in [1])

$$\Gamma \models_{K_S} \alpha \quad \text{if and only if} \quad \Gamma \vdash_S \alpha, \quad (22)$$

and that (see 3.27 in [1])

$$\llbracket \tau \rrbracket_{K_S, \mathbf{x}} = (\mathbf{x} \mapsto \tau). \quad (23)$$

In particular, the following expected result holds.

► **Lemma 4.** *Let S be a local set theory in a local language $\mathcal{L}$. Let α be a formula with a unique free variable x of type $\mathbf{A}$. The following is a pullback diagram in $\mathcal{T}(S)$.*

$$\begin{array}{ccc} \{x : \alpha\} & \longrightarrow & 1 \\ (x \mapsto x) \downarrow & & \downarrow \top \\ A & \xrightarrow[\substack{(x \mapsto \alpha)}]{\llbracket \alpha \rrbracket_x} & \Omega \end{array} \quad (24)$$

Proof. Straightforward. ◀

► **Remark 5.** The map $(x \mapsto x) : X \longrightarrow A$ will frequently be denoted by i_X .

2.6 Internal language and local set theory of a topos

Let $\mathcal{E}$ be a topos with specified terminal $1_{\mathcal{E}}$, subobject classifier $\Omega_{\mathcal{E}}$, products and power objects. The internal language of $\mathcal{E}$ is the local language $\mathcal{L}(\mathcal{E})$ whose ground type symbols are precisely the objects X of $\mathcal{E}$, denoted by $\mathbf{X}$, and whose function symbols the arrows f of $\mathcal{E}$, denoted by $\mathbf{f}$, with signature $\mathbf{dom}(\mathbf{f}) \rightarrow \mathbf{cod}(\mathbf{f})$. The obvious natural interpretation $(\mathcal{E}, H_{\mathcal{E}})$ of $\mathcal{L}(\mathcal{E})$ is given by $\mathbf{X}_{H_{\mathcal{E}}} = X$ and $\mathbf{f}_{H_{\mathcal{E}}} = f$. Equality, membership and comprehension in $\mathcal{L}(\mathcal{E})$ are written in bold; i.e. $=, \in, \{ : \}$.

The local set theory of a topos, $\text{Th}(\mathcal{E})$, is determined by the requirement that $\Gamma : \alpha$ is in $\text{Th}(\mathcal{E})$ if $\Gamma \models_{H_{\mathcal{E}}} \alpha$. It follows that (see 3.30 in [1])

$$\Gamma \vdash_{\text{Th}(\mathcal{E})} \alpha \quad \text{if and only if} \quad \Gamma \models_{H_{\mathcal{E}}} \alpha. \quad (25)$$

This interpretation is compatible with one's intuition, e.g. for any $\mathcal{E}$ -arrow f (see 3.33 in [1]),

$$f \text{ is monic if and only if } \mathbf{f}(x) = \mathbf{f}(y) \vdash_{\text{Th}(\mathcal{E})} x = y. \quad (26)$$

As a consequence of (20), [1, 3.23.2], and (16) explicit term representation can be described categorically as follows.

► **Theorem 6** ([1] 3.34). *The following are equivalent:*

- (i) $\vdash_{\text{Th}(\mathcal{E})} \alpha(y_1/\tau_1, \dots, y_n/\tau_n),$
- (ii) $\llbracket \alpha \rrbracket_{H_{\mathcal{E}}, \mathbf{y}} \circ f = \llbracket \langle \tau_1, \dots, \tau_n \rangle \rrbracket_{H_{\mathcal{E}}, \mathbf{x}}$ for some arrow f , where $y_1, \dots, y_n$ are the free variables of α and τ_i is free for y_i in α for each i .

► **Theorem 7** ([1] 3.35). *If m is monic in $\mathcal{E}$, then*

$$\vdash_{\text{Th}(\mathcal{E})} \chi(\mathbf{m})(x) \Leftrightarrow \exists y. x = \mathbf{m}(y).$$

$\text{Th}(\mathcal{E})$ is both well-termed and well-typed as the following two results show.

► **Theorem 8** ([1] 3.36). *If $\vdash_{\text{Th}(\mathcal{E})} \exists! y \alpha$, then there exists a unique $\mathcal{E}$ -arrow f such that*

$$\vdash_{\text{Th}(\mathcal{E})} \alpha(y/\mathbf{f}(\langle x_1, \dots, x_n \rangle)),$$

where $x_1, \dots, x_n, y$ are the free variables of α .

Finally, passing from $\mathcal{E}$ to the linguistic topos of $\text{Th}(\mathcal{E})$ gives no additional information. Indeed, the following equivalence theorem holds.

► **Theorem 9** ([1] 3.37). *Every topos $\mathcal{E}$ is equivalent to the linguistic topos $\mathcal{T}(\text{Th}(\mathcal{E}))$ of its local set theory.*

2.7 Translations

A *translation* $\kappa : \mathcal{L} \rightarrow \Lambda$ between local languages $\mathcal{L}$ and Λ is an assignment of a type (resp. function) symbol in Λ to each type (resp. function) symbol in $\mathcal{L}$ in a way compatible with products, powers, truth-value and unity types. Terms are translated in the expected way as to preserve the basic axioms. Between LSTs, S and Σ in $\mathcal{L}$ and Λ respectively, κ is said to be a translation if for every sequent $\Gamma : \alpha$ of $\mathcal{L}$, $\Gamma \vdash_S \alpha$ implies that $\kappa\Gamma \vdash_{\Sigma} \kappa\alpha$. If under the same conditions $\Gamma \vdash_S \alpha$ if and only if $\kappa\Gamma \vdash_{\Sigma} \kappa\alpha$, then κ is called *conservative*.

Any translation $\kappa : \mathcal{L} \rightarrow \Lambda$ induces a logical functor $\mathcal{T}(\kappa)$ (one that preserves the specified terminal object, products, subobject classifier, power objects and evaluation arrows) between the associated linguistic toposes.

For any local set theory S with underlying local language $\mathcal{L}$, let the *canonical translation* $\eta_S : S \rightarrow \text{Th}(\mathcal{T}(S))$ be the assignment to each type symbol Ξ of the type symbol associated to U_{Ξ} and to $\mathbf{f}$ of the function symbol associated to $(x \mapsto \mathbf{f}(x))$. It follows that it is a conservative translation and that

$$\llbracket \tau \rrbracket_{K_S, \mathbf{x}} = \llbracket \eta_S(\tau) \rrbracket_{H_{\mathcal{T}(S)}, \eta_S \mathbf{x}}. \quad (27)$$

Whenever there is no risk of confusion one can represent the type symbol associated to a type symbol $\mathbf{A}$ with the same letter; in essence writing $\mathbf{A}$ for $\eta_S(\mathbf{A})$. In particular, given an S -set $X := \{x_{\mathbf{A}} : \alpha\}$, $\eta_S(X)$ is described by $\{\eta_S(x_{\mathbf{A}}) : \eta_S(\alpha)\}$, and since η_S is conservative, there is no loss in representing $\eta_S(X) = X$. The same will be assumed for a syntactical function $f = (|f|, X, Y)$. However, $(x \mapsto \mathbf{f}(x))$ need not be an S -function, since the function symbol $\mathbf{f}$ need not be in $\mathcal{L}$, only in $\mathcal{L}(\mathcal{T}(S))$. That is, $(x \mapsto \mathbf{f}(x)) : \mathbf{X} \rightarrow \mathbf{Y}$ is an arrow in $\mathcal{T}(\text{Th}(\mathcal{T}(S)))$, not in $\mathcal{T}(S)$.

3 Parameterizations

The main purpose of this section is to prove Theorem B. To this effect, the central idea will be the translation of arbitrary formulas along arbitrary syntactic functions. This is introduced in the following definition. Next, some of its properties will be studied; in particular, an explicit description will be given in Lemma 12.

► **Definition 10.** Let S be a local set theory on a local language $\mathcal{L}$. Let $f : X \rightarrow Y$ be an S -function with X of type **PA** and Y of type **PB**. Let ϑ be an arbitrary formula, x a variable of type **A** not free in ϑ , and y a variable of type **B**. Let $\mathbf{y}$ be a finite sequence of variables containing the free variables of ϑ and y . Denote by $\mathbf{x}$ the sequence of variables obtained from $\mathbf{y}$ by replacing y with the variable x . For any formula ϑ in $\mathcal{L}$, its preimage translation by f , or simply f -translation, is the formula

$$\vartheta_{\mathbf{y},f,x}^{\natural} := (\llbracket \vartheta \rrbracket_{K,\mathbf{y}} \circ 1 \times i_Y \circ 1 \times f)^{\natural}(\mathbf{x}), \quad (28)$$

where for any function g , the term $g^{\natural}(x) := \langle x, \top \rangle \in |g|$ is given by 19.

► **Remark 11.** Under the same hypotheses of Definition 10, one has that $\vartheta_{\mathbf{y},f,x}^{\natural} \vdash_S x \in X$.

► **Lemma 12.** Let S be a local set theory in a local language $\mathcal{L}$. Let $f : X \rightarrow Y$ be an S -function with X of type **PA** and Y of type **PB**. Then $\vdash_S \vartheta_{\mathbf{y},f,x}^{\natural} \Leftrightarrow \exists y(\langle x, y \rangle \in |f| \wedge \vartheta)$.

Proof. First, to verify $\vartheta_{\mathbf{y},f,x}^{\natural} \vdash_S \exists y.\langle x, y \rangle \in |f| \wedge \vartheta$, by (28) and (19), is the same as to verify

$$\langle \langle x_1, \dots, x_n, x \rangle, \top \rangle \in |(\mathbf{y} \mapsto \vartheta) \circ 1 \times i_Y \circ 1 \times f| \vdash_S \exists y'(\langle x, y' \rangle \in |f| \wedge \vartheta(y/y')),$$

where $x_1, \dots, x_n, y$ contain the variables of ϑ . Notice that the replacement of y by y' requires y' to be free for y in ϑ . Equivalently,

$$\begin{aligned} \exists z \exists z'. \langle \langle x_1, \dots, x_n, x \rangle, z \rangle \in |1 \times f| \wedge \langle z, z' \rangle \in |1 \times i_Y| \wedge \langle z', \top \rangle \in |(\mathbf{y} \mapsto \vartheta)| \\ \vdash_S \exists y'(\langle x, y' \rangle \in |f| \wedge \vartheta(y/y')). \end{aligned}$$

By [1, 3.7.2], it suffices to verify

$$\begin{aligned} \langle \langle x_1, \dots, x_n, x \rangle, z \rangle \in |1 \times f| \wedge \langle z, z' \rangle \in |1 \times i_Y| \wedge \langle z', \top \rangle \in |(\mathbf{y} \mapsto \vartheta)| \\ \vdash_S \exists y'(\langle x, y' \rangle \in |f| \wedge \vartheta(y/y')). \end{aligned} \quad (29)$$

To this effect, let $x'_1, \dots, x'_n$ and $x''_1, \dots, x''_n$ and $x'''_1, \dots, x'''_n$ be variables of the same types as $x_1, \dots, x_n$, respectively. Let x' be of the same type as x , and y'', y''' be different from but of the same type as y . Let z, z' be variables of the same type as $\langle x_1, \dots, x_n, y \rangle$. Then,

$$\begin{aligned} \langle \langle x_1, \dots, x_n, x \rangle, z \rangle &= \langle \langle x'_1, \dots, x'_n, x' \rangle, \langle x'_1, \dots, x'_n, y' \rangle \rangle \wedge \\ &\quad x'_1 \in A_1 \wedge \dots \wedge x'_n \in A_n \wedge \langle x', y' \rangle \in |f| \wedge \\ \langle z, z' \rangle &= \langle \langle x''_1, \dots, x''_n, y'' \rangle, \langle x''_1, \dots, x''_n, y'' \rangle \rangle \wedge \\ &\quad y'' \in Y \wedge x''_1 \in A_1 \wedge \dots \wedge x''_n \in A_n \wedge \\ &\quad \langle z', \top \rangle = \langle \langle x'''_1, \dots, x'''_n, y''' \rangle, \vartheta \rangle \wedge \\ &\quad x'''_1 \in A_1 \wedge \dots \wedge x'''_n \in A_n \wedge y''' \in B \vdash_S \\ &\quad \langle x, y' \rangle \in |f| \wedge \vartheta = \top \end{aligned}$$

14:12 Functional Representability in Local Set Theories

By the substitution rule, replacing y with y' , and [1, 3.7.1]

$$\begin{aligned}
\langle \langle x_1, \dots, x_n, x \rangle, z \rangle &= \langle \langle x'_1, \dots, x'_n, x' \rangle, \langle x'_1, \dots, x'_n, y' \rangle \rangle \wedge \\
&\quad x'_1 \in A_1 \wedge \dots \wedge x'_n \in A_n \wedge \langle x', y' \rangle \in |f| \wedge \\
\langle z, z' \rangle &= \langle \langle x''_1, \dots, x''_n, y'' \rangle, \langle x''_1, \dots, x''_n, y'' \rangle \rangle \wedge \\
&\quad y'' \in Y \wedge x''_1 \in A_1 \wedge \dots \wedge x''_n \in A_n \wedge \\
\langle z', \top \rangle &= \langle \langle x'''_1, \dots, x'''_n, y''' \rangle, \vartheta \rangle \wedge \\
x'''_1 \in A_1 \wedge \dots \wedge x'''_n \in A_n \wedge y''' \in B &\vdash_S \\
\langle x, y' \rangle &\in |f| \wedge \vartheta(y/y') = \top \vdash_S \\
\langle x, y' \rangle &\in |f| \wedge \vartheta(y/y') \vdash_S \\
&\quad \exists y' (\langle x, y' \rangle \in |f| \wedge \vartheta(y/y'));
\end{aligned}$$

wherefrom, by [1, 3.7.2],

$$\begin{aligned}
\langle \langle x_1, \dots, x_n, x \rangle, z \rangle &= \langle \langle x'_1, \dots, x'_n, x' \rangle, \langle x'_1, \dots, x'_n, y' \rangle \rangle \wedge \\
&\quad x'_1 \in A_1 \wedge \dots \wedge x'_n \in A_n \wedge \langle x', y' \rangle \in |f| \wedge \\
\langle z, z' \rangle &= \langle \langle x''_1, \dots, x''_n, y'' \rangle, \langle x''_1, \dots, x''_n, y'' \rangle \rangle \wedge \\
&\quad y'' \in Y \wedge x''_1 \in A_1 \wedge \dots \wedge x''_n \in A_n \wedge \\
\exists x'''_1 \dots \exists x'''_n \exists y''' (\langle z', \top \rangle &= \langle \langle x'''_1, \dots, x'''_n, y''' \rangle, \vartheta \rangle \wedge \\
&\quad x'''_1 \in A_1 \wedge \dots \wedge x'''_n \in A_n \wedge y''' \in B) \vdash_S \\
&\quad \exists y' (\langle x, y' \rangle \in |f| \wedge \vartheta(y/y'));
\end{aligned}$$

thus,

$$\begin{aligned}
\langle \langle x_1, \dots, x_n, x \rangle, z \rangle &= \langle \langle x'_1, \dots, x'_n, x' \rangle, \langle x'_1, \dots, x'_n, y' \rangle \rangle \wedge \\
&\quad x'_1 \in A_1 \wedge \dots \wedge x'_n \in A_n \wedge \langle x', y' \rangle \in |f| \wedge \\
\langle z, z' \rangle &= \langle \langle x''_1, \dots, x''_n, y'' \rangle, \langle x''_1, \dots, x''_n, y'' \rangle \rangle \wedge \\
&\quad y'' \in Y \wedge x''_1 \in A_1 \wedge \dots \wedge x''_n \in A_n \wedge \\
\langle z', \top \rangle &\in |(\mathbf{y} \mapsto \vartheta)| \vdash_S \\
&\quad \exists y' (\langle x, y' \rangle \in |f| \wedge \vartheta(y/y')).
\end{aligned}$$

Following the same process, one obtains (29).

Conversely,

$$\langle x, y \rangle \in |f| \wedge \vartheta \vdash_S \vartheta \vdash_S \vartheta = \top \vdash_S \langle \langle x_1, \dots, x_n, y \rangle, \top \rangle = \langle \langle x_1, \dots, x_n, y \rangle, \vartheta \rangle;$$

wherefrom, by [1, 3.7.4],

$$\langle x, y \rangle \in |f| \wedge \vartheta \vdash_S \exists x'_1 \dots \exists x'_n \exists y' (\langle \langle x_1, \dots, x_n, y \rangle, \top \rangle = \langle \langle x'_1, \dots, x'_n, y' \rangle, \vartheta(\mathbf{y}/\mathbf{y}') \rangle),$$

so that $\langle x, y \rangle \in |f| \wedge \vartheta \vdash_S \langle \langle x_1, \dots, x_n, y \rangle, \top \rangle \in |(\mathbf{y} \mapsto \vartheta)|$. Similarly,

$$\langle x, y \rangle \in |f| \wedge \vartheta \vdash_S \langle \langle x_1, \dots, x_n, y \rangle, \langle x_1, \dots, x_n, y \rangle \rangle \in |1 \times i_Y| \quad \text{and}$$

$$\langle x, y \rangle \in |f| \wedge \vartheta \vdash_S \langle \langle x_1, \dots, x_n, x \rangle, \langle x_1, \dots, x_n, y \rangle \rangle \in |1 \times f|.$$

Therefore,

$$\begin{aligned}
\langle x, y \rangle &\in |f| \wedge \vartheta \vdash_S \langle \langle x_1, \dots, x_n, y \rangle, \top \rangle \in |(\mathbf{y} \mapsto \vartheta)| \wedge \\
&\quad \langle \langle x_1, \dots, x_n, y \rangle, \langle x_1, \dots, x_n, y \rangle \rangle \in |1 \times i_Y| \\
&\quad \wedge \langle \langle x_1, \dots, x_n, x \rangle, \langle x_1, \dots, x_n, y \rangle \rangle \in |1 \times f|.
\end{aligned}$$

Thus, by [1, 3.7.4], $\langle x, y \rangle \in |f| \wedge \vartheta \vdash_S \langle \langle x_1, \dots, x_n, x \rangle, \top \rangle \in |(\mathbf{y} \mapsto \vartheta) \circ 1 \times i_Y \circ 1 \times f|$. This, together with [1, 3.7.2], yields that $\exists y (\langle x, y \rangle \in |f| \wedge \vartheta) \vdash_S \vartheta_{\mathbf{y}, f, x}^{\mathbf{a}}$. $\blacktriangleleft$

This lemma has a couple of immediate consequences.

► **Corollary 13.** *Let S be an LST in a local language $\mathcal{L}$. Let $f : X \rightarrow Y$ be an S -function with X of type $\mathbf{PA}$ and Y of type $\mathbf{PB}$, and ϑ a formula in $\mathcal{L}$. Let $\mathbf{y}$ be a finite sequence of variables containing those of ϑ , as well as y . Let $\mathbf{y}'$ be a subsequence of $\mathbf{y}$ still containing the free variables of ϑ , as well as y . Then, $\vdash_S \vartheta_{\mathbf{y},f,x}^{\mathbf{h}} = \vartheta_{\mathbf{y}',f,x}^{\mathbf{h}}$.*

Under the same conditions, given that one can add superfluous variables, denote $\vartheta_{\mathbf{y},f,x}^{\mathbf{h}}$ as $\vartheta_{y,f,x}^{\mathbf{h}}$; if from context one can infer the fixed variables, denote it as $\vartheta_f^{\mathbf{h}}$; and if the S -function f is clear from context, denote it simply as $\vartheta^{\mathbf{h}}$.

► **Corollary 14.** *Let S be an LST in a local language $\mathcal{L}$. Let $f : X \rightarrow Y$ be an S -function with X of type $\mathbf{PA}$ determined by α and Y of type $\mathbf{PB}$, and ϑ a formula in $\mathcal{L}$ not having y as a free variable. Then, $\vdash_S \vartheta_f^{\mathbf{h}} = \vartheta \wedge x \in X$.*

Proof. From Lemma 12 and [1, 3.7.6], one obtains $\vartheta_f^{\mathbf{h}} \vdash_S \vartheta$ and, by Remark 11, $\vartheta_f^{\mathbf{h}} \vdash_S \vartheta \wedge x \in X$. Conversely, it follows that $\vartheta \wedge x \in X \vdash_S \vartheta$ and $x \in X \vdash_S \exists! y (\langle x, y \rangle \in |f|) \vdash_S \exists y (\langle x, y \rangle \in |f|)$. Wherefrom, by [1, 3.7.6],

$$\vartheta \wedge x \in X \vdash_S \exists y (\langle x, y \rangle \in |f|) \wedge \vartheta \vdash_S \exists y (\langle x, y \rangle \in |f| \wedge \vartheta) \vdash_S \vartheta_f^{\mathbf{h}}. \quad \blacktriangleleft$$

For a well-termed local set theory S in a local language $\mathcal{L}$, there is functional representability from universals even if the codomain is not universal.

► **Proposition 15.** *Let S be a well-termed LST in a local language $\mathcal{L}$. Let $\mathbf{A}$ and $\mathbf{B}$ be types. Let $f : U_{\mathbf{B}} \rightarrow X$ be an S -function with $X := \{x : \alpha\}$ with α a formula with a unique free variable x of type $\mathbf{A}$. Then f is the corestriction to X of $(u \mapsto \tau_f)$ for some term τ_f in $\mathcal{L}$ and $\vdash_S \tau_f \in X$. In this case, τ_f represents f .*

Proof. Consider the S -function $i_X \circ f$. Since S is well-termed, $i_X \circ f = (u \mapsto \tau_f)$ for some term τ_f . Thus, by (23), $\llbracket \alpha \rrbracket_{K,x} \circ f = \llbracket \tau_f \rrbracket_{K,u}$; wherefrom, $\llbracket \alpha \rrbracket_{K,x} \circ \llbracket \tau_f \rrbracket_{K,u} = T_B$. This, together with (20) and [1, 3.23.2], $\models_K \alpha(x/\tau_f)$. So that by (22), $\vdash_S \alpha(\tau_f)$; thus, $\vdash_S \tau_f \in X$.

Lastly, let $g := ((u \mapsto \tau_f), B, X)$. Since i_X is monic and $i_X \circ g = (u \mapsto \tau_f)$, the conclusion follows. $\blacktriangleleft$

One can now proceed to prove Theorem B. The statement is split into Theorem 16 and Theorem 18. The former provides the logical framework to the set-theoretical functional representability of Section 4.

► **Theorem 16.** *Let S be a well-termed local set theory in a local language $\mathcal{L}$. Let $\mathbf{A}$ and $\mathbf{B}$ be types in $\mathcal{L}$. Let $f : U_{\mathbf{B}} \rightarrow X$ be a surjective S -function with X determined by the formula α with unique variable of type $\mathbf{A}$, and τ_f a term that represents f . Let γ and Γ be a formula and a finite set of formulae in $\mathcal{L}$. Then,*

$$\Gamma, x \in X \vdash_S \gamma \quad \text{if and only if} \quad \Gamma(x/\tau_f) \vdash_S \gamma(x/\tau_f).$$

Furthermore, if f is injective, then, for every formula ϑ ,

$$\vdash_S \vartheta = \vartheta_{f^{-1}}^{\mathbf{h}}(x/\tau_f).$$

And in this case,

$$\Gamma \vdash_S \gamma \quad \text{if and only if} \quad \Gamma_{f^{-1}}^{\mathbf{h}}, x \in X \vdash_S \gamma_{f^{-1}}^{\mathbf{h}}.$$

► **Lemma 17.** *Let S be a local set theory in a local language $\mathcal{L}$. Let ξ, γ be formulae in $\mathcal{L}$. Let $x_1, \dots, x_n$ be variables of types $\mathbf{A}_1, \dots, \mathbf{A}_n$, containing the free variables of ξ and γ . Let*

$$Z := \{\langle x_1, \dots, x_n \rangle : \xi\}.$$

Then, $\xi \vdash_S \gamma$ if and only if the following diagram commutes in $\mathcal{T}(S)$:

$$\begin{array}{ccc} Z & \xrightarrow{\quad} & 1 \\ i_Z \downarrow & & \downarrow \top \\ A_1 \times \cdots \times A_n & \xrightarrow{(\mathbf{x} \mapsto \gamma)} & \Omega. \end{array}$$

Proof. Let $W := \{\langle x_1, \dots, x_n \rangle : \gamma\}$. Then,

$$\begin{aligned} \xi \vdash_S \gamma \text{ iff } & \xi \models_K \gamma && \text{(by (22))} \\ \text{iff } & \llbracket \xi \rrbracket_{K, \mathbf{x}} \leq \llbracket \gamma \rrbracket_{K, \mathbf{x}} && \text{(by (21))} \\ \text{iff } & \text{there is an arrow } f : Z \rightarrow W \text{ such that } i_Z = i_W \circ f \end{aligned}$$

which completes the proof. $\square$

Proof of Theorem 16. Without loss of generality, assume $\Gamma = \{\xi\}$ and that u , of type **B**, is neither in ξ nor in γ . Replace an empty Γ with $\{\top\}$.

Assume that $\xi, x \in X \vdash_S \gamma$. By substitution, $\xi(x/\tau_f), \tau_f \in X \vdash_S \gamma(\tau_f)$. By Proposition 15, it follows from the Cut Rule that $\xi(x/\tau_f) \vdash_S \gamma(\tau_f)$.

Conversely, assume $\xi(x/\tau_f) \vdash_S \gamma(\tau_f)$. Let $x_1, \dots, x_n$ be variables of types $\mathbf{A}_1, \dots, \mathbf{A}_n$, resp., containing the free variables of ξ and α , with $x_n = x$. Let

$$Z := \{\langle x_1, \dots, x_n \rangle : \xi \wedge \alpha\} \quad \text{and} \quad Z' := \{\langle x_1, \dots, x_{n-1}, u \rangle : \xi(x/\tau_f)\}.$$

Thus, the following are pullback diagrams:

$$\begin{array}{ccc} Z' & \xrightarrow{\quad} & 1 \\ i_{Z'} \downarrow & & \downarrow \top \\ A_1 \times \cdots \times A_{n-1} \times B & \xrightarrow{(\mathbf{x}' \mapsto \xi(\mathbf{x}/\tau_f))} & \Omega \end{array} \qquad \begin{array}{ccc} Z & \xrightarrow{\quad} & 1 \\ i_Z \downarrow & & \downarrow \top \\ A_1 \times \cdots \times A_n & \xrightarrow{(\mathbf{x} \mapsto \xi \wedge \alpha)} & \Omega, \end{array}$$

with $i_{Z'} = (\mathbf{x}' \mapsto \mathbf{x}')$ and $i_Z = (\mathbf{x} \mapsto \mathbf{x})$.

Now, since $\langle x_1, \dots, x_{n-1}, x \rangle \in Z \vdash_S \xi \wedge \alpha \vdash_S \alpha \vdash_S x \in X$, let $j_Z := (|i_Z|, Z, A_1 \times \dots \times X)$. Then $i_Z = 1 \times i_X \circ j_Z$, and the diagram

$$\begin{array}{ccccc} Z & \xrightarrow{\quad} & 1 \\ j_Z \downarrow & & \downarrow \tau \\ A_1 \times \cdots \times A_{n-1} \times X & \xrightarrow{1 \times i_X} & A_1 \times \cdots \times A_n & \xrightarrow{(\mathbf{x} \mapsto \xi \wedge \alpha)} & \Omega \end{array}$$

is a pullback diagram too. So is the following:

$$\begin{array}{ccccccc}
Z' & \xrightarrow{\quad} & & & & & 1 \\
\downarrow i_{Z'} & & & & & & \downarrow \top \\
A_1 \times \cdots \times B & \xrightarrow{1 \times f} & A_1 \times \cdots \times X & \xrightarrow{1 \times i_X} & A_1 \times \cdots \times A_n & \xrightarrow{(x \mapsto \xi \wedge \alpha)} & \Omega. \\
& \searrow 1 \times (u \mapsto \tau_f) & & & & & \\
& & & & & \searrow (\mathbf{x}' \mapsto \xi(x/\tau_f)) &
\end{array}$$

Consider now the following diagram:

$$\begin{array}{ccccc}
 Z' & \xrightarrow{\quad f' \quad} & Z & \xrightarrow{\quad} & 1 \\
 \downarrow i_{Z'} & & \downarrow j_Z & & \downarrow \top \\
 A_1 \times \cdots \times B & \xrightarrow{1 \times f} & A_1 \times \cdots \times X & \xrightarrow{1 \times i_X} & A_1 \times \cdots \times A_n \xrightarrow{(\mathbf{x} \mapsto \xi \wedge \alpha)} \Omega.
 \end{array}$$

Since the right square is a pullback and the exterior rectangle commutes, there is a unique $f' : Z' \rightarrow Z$ making the diagram commute. Since the exterior diagram is a pullback, the left square is too. Now, since f is epic, f is surjective; thus, so is $1 \times f$ and thus also epic. Therefore, f' is epic since in a topos the pullback of an epic is epic.

Now consider the following diagram:

$$\begin{array}{ccccc}
 Z' & \xrightarrow{\quad f' \quad} & Z & \xrightarrow{\quad} & 1 \\
 \downarrow i_{Z'} & & \downarrow j_Z & & \downarrow \top \\
 A_1 \times \cdots \times B & \xrightarrow{1 \times f} & A_1 \times \cdots \times X & \xrightarrow{1 \times i_X} & A_1 \times \cdots \times A_n \xrightarrow{(\mathbf{x} \mapsto \gamma)} \Omega. \\
 & & & & \text{---} \xrightarrow{(\mathbf{x}' \mapsto \gamma(x/\tau_f))} \text{---}
 \end{array}$$

Since $\xi(x/\tau_f) \vdash_S \gamma(x/\tau_f)$, the external diagram commutes by Lemma 17. Since the left square commutes, $(\mathbf{x} \mapsto \gamma) \circ i_Z \circ f' = \top \circ !_Z \circ f'$.

Since f' is epic, the right square commutes; therefore by Lemma 17, $\xi \wedge \alpha \vdash_S \gamma$ and thus $\xi, x \in X \vdash_S \gamma$.

Lastly, suppose that f is furthermore monic. Let ϑ be a formula the free variables of which are within $x_1, \dots, x_{n-1}, u$. Define

$$\mathfrak{X} := \{\langle x_1, \dots, x_{n-1}, u \rangle : \vartheta\} \quad N_{\mathfrak{X}} := \{\langle x_1, \dots, x_{n-1}, x \rangle : \vartheta^{\natural}\}.$$

Recall that $\llbracket \vartheta \rrbracket_{K, x_1, \dots, x_{n-1}, u} = (\langle x_1, \dots, x_{n-1}, u \rangle \mapsto \vartheta)$ by (23), and that by Proposition 3, $(\mathbf{x}' \mapsto \vartheta) \circ 1 \times f^{-1} = (\mathbf{x} \mapsto \vartheta^{\natural}_{f^{-1}})$. Consider the following diagram:

$$\begin{array}{ccccc}
 N_{\mathfrak{X}} & \xrightarrow{\quad g \quad} & \mathfrak{X} & \xrightarrow{\quad} & 1 \\
 \downarrow j_{N_{\mathfrak{X}}} & & \downarrow i_{\mathfrak{X}} & & \downarrow \top \\
 A_1 \times \cdots \times X & \xrightarrow{1 \times f^{-1}} & A_1 \times \cdots \times B & \xrightarrow{(\mathbf{x}' \mapsto \vartheta)} & \Omega.
 \end{array}$$

Again, since the right square is a pullback and the exterior diagram commutes, there is a unique g making the whole diagram commute. Since the rectangle is a pullback (the external diagram was a pullback already), the left square is too, so that $N_{\mathfrak{X}} \cong \mathfrak{X}$.

Now, considering the following commutative diagram,

$$\begin{array}{ccccccc}
 \mathfrak{X} & \xrightarrow{\quad} & N_{\mathfrak{X}} & \xrightarrow{\quad g \quad} & \mathfrak{X} & \xrightarrow{\quad} & 1 \\
 \downarrow i_{\mathfrak{X}} & & \downarrow j_{N_{\mathfrak{X}}} & & \downarrow i_{\mathfrak{X}} & & \downarrow \top \\
 A_1 \times \cdots \times B & \xrightarrow{1 \times f} & A_1 \times \cdots \times X & \xrightarrow{1 \times f^{-1}} & A_1 \times \cdots \times B & \xrightarrow{(\mathbf{x}' \mapsto \vartheta)} & \Omega, \\
 & & & & & & \text{---} \xrightarrow{(\mathbf{x} \mapsto \vartheta^{\natural})} \text{---}
 \end{array}$$

14:16 Functional Representability in Local Set Theories

one clearly obtains $\vdash_S \vartheta^\sharp(\tau_f) = \vartheta$, since f is the corestriction to X of $(u \mapsto \tau_f)$. From here, $\xi \vdash_S \gamma$ if and only if $\xi^\sharp, x \in X \vdash_S \gamma^\sharp$, since $\xi^\sharp(\tau_f) \vdash_S \gamma^\sharp(\tau_f)$ if and only if $\xi^\sharp, x \in X \vdash_S \gamma^\sharp$, by the first part of the theorem. $\blacktriangleleft$

► **Theorem 18.** *Let S be a well-termed local set theory in a local language $\mathcal{L}$. Let $f : U_{\mathbf{B}} \rightarrow X$ be an S -bijection, with X determined by a formula α with unique variable of type $\mathbf{A}$. Let τ_f be a term representing f and let ϑ and γ be formulae in $\mathcal{L}$. Then,*

$$\vdash_S (\vartheta \square \gamma)_{f^{-1}}^\sharp = \vartheta_{f^{-1}}^\sharp \square \gamma_{f^{-1}}^\sharp,$$

where $\square$ is any binary logical connective. Furthermore,

$$\vdash_S (\neg \vartheta)_{f^{-1}}^\sharp = \neg \vartheta_{f^{-1}}^\sharp,$$

and if u is a variable of type $\mathbf{B}$ free in ϑ and x is not free in ϑ , then

$$\vdash_S \forall u \vartheta = \forall x \in X. \vartheta_{f^{-1}}^\sharp \text{ and } \vdash_S \exists u \vartheta = \exists x \in X. \vartheta_{f^{-1}}^\sharp.$$

Proof. Throughout this proof, denote $(-)^{\sharp}_{\langle x_1, \dots, x_{n-1}, u \rangle, f^{-1}, x}$ with $(-)^{\sharp}_{f^{-1}}$, where $x_1, \dots, x_{n-1}$ are the appropriate variables. By the previous theorem,

$$\vdash_S (\vartheta \square \gamma)_{f^{-1}}^\sharp(\tau_f) = \vartheta \square \gamma, \quad \vdash_S \vartheta_{f^{-1}}^\sharp(\tau_f) = \vartheta, \quad \vdash_S \gamma_{f^{-1}}^\sharp(\tau_f) = \gamma.$$

Wherefrom, $\vdash_S (\vartheta \square \gamma)_{f^{-1}}^\sharp(\tau_f) = \vartheta_{f^{-1}}^\sharp(\tau_f) \square \gamma_{f^{-1}}^\sharp(\tau_f)$. Now, by Proposition 15, $f = (|(u \mapsto \tau_f)|, B, X)$. So that

$$(\mathbf{x} \mapsto (\vartheta \square \gamma)_{f^{-1}}^\sharp) \circ 1 \times f = (\mathbf{x} \mapsto \vartheta_{f^{-1}}^\sharp \square \gamma_{f^{-1}}^\sharp) \circ 1 \times f; \quad (30)$$

thus since f is iso, $1 \times f$ is too. Therefore,

$$(\mathbf{x} \mapsto (\vartheta \square \gamma)_{f^{-1}}^\sharp) \text{ is equal to } (\mathbf{x} \mapsto \vartheta_{f^{-1}}^\sharp \square \gamma_{f^{-1}}^\sharp).$$

Notice that both functions go from $A_1 \times \dots \times X$ to Ω ; however,

$$N_{(\vartheta \square \gamma)_{f^{-1}}^\sharp} := \{\langle x_1, \dots, x \rangle : (\vartheta \square \gamma)_{f^{-1}}^\sharp\}, \quad N_{\vartheta_{f^{-1}}^\sharp \square \gamma_{f^{-1}}^\sharp} := \{\langle x_1, \dots, x \rangle : \vartheta_{f^{-1}}^\sharp \square \gamma_{f^{-1}}^\sharp\}$$

are each the pullback object of the S -functions $(\mathbf{x} \mapsto (\vartheta \square \gamma)_{f^{-1}}^\sharp)$ and $(\mathbf{x} \mapsto \vartheta_{f^{-1}}^\sharp \square \gamma_{f^{-1}}^\sharp)$ from $A_1 \times \dots \times A$ to Ω , so that $\vdash_S (\vartheta \square \gamma)_{f^{-1}}^\sharp = \vartheta_{f^{-1}}^\sharp \square \gamma_{f^{-1}}^\sharp$. Likewise, $\vdash_S (\neg \vartheta)_{f^{-1}}^\sharp = \neg \vartheta_{f^{-1}}^\sharp$.

Now let ϑ be a formula with u among its free variables and let $x_1, \dots, x_{n-1}$ contain the rest of the free variables of ϑ . Then, by [1, 3.4.4], $\forall u \vartheta \vdash_S \vartheta$. Thus, by Theorem 16 and Remark 11, $(\forall u \vartheta)_{f^{-1}}^\sharp \vdash_S \vartheta_{f^{-1}}^\sharp$; so that by Corollary 14 $\vdash_S (\forall u \vartheta)_{f^{-1}}^\sharp = (\forall u \vartheta \wedge x \in X)$. Therefore, $\forall u \vartheta \wedge x \in X \vdash_S \vartheta_{f^{-1}}^\sharp$. From here, $\forall u \vartheta \vdash_S x \in X \Rightarrow \vartheta_{f^{-1}}^\sharp$, and thus, by [1, 3.4.2], $\forall u \vartheta \vdash_S \forall x \in X. \vartheta_{f^{-1}}^\sharp$.

Conversely, by [1, 3.4.4] $\forall x \in X. \vartheta_{f^{-1}}^\sharp \vdash_S x \in X \Rightarrow \vartheta_{f^{-1}}^\sharp$. From here, $x \in X, \forall x \in X. \vartheta_{f^{-1}}^\sharp \vdash_S \vartheta_{f^{-1}}^\sharp$. Thus by Theorem 16, $(\forall x \in X. \vartheta_{f^{-1}}^\sharp)(\tau_f) \vdash_S \vartheta$.

As u is no longer free in $\forall x \in X. \vartheta_{f^{-1}}^\sharp$, since – by Lemma 12 – $\vartheta_{\langle x_1, \dots, x_{n-1}, u \rangle, f^{-1}, x}^\sharp$ binds u , it follows (by [1, 3.4.2]) that $\forall x \in X. \vartheta_{f^{-1}}^\sharp \vdash_S \forall u \vartheta$, and thus $\vdash_S \forall u \vartheta = \forall x \in X. \vartheta_{f^{-1}}^\sharp$.

Finally, by [1, 3.7.1], $\vartheta \vdash_S \exists u \vartheta$. Thus

$$x \in X, \vartheta_{f^{-1}}^\sharp \vdash_S (\exists u \vartheta)_{f^{-1}}^\sharp \quad (\text{by Theorem 16})$$

$$\vdash_S \exists u \vartheta \wedge x \in X \quad (\text{by Corollary 14})$$

$$\vdash_S \exists u \vartheta.$$

Therefore, by [1, 3.7.2], $\exists x \in X. \vartheta_{f^{-1}}^\sharp \vdash_S \exists u \vartheta$.

Conversely, by [1, 3.7.1], $x \in X \wedge \vartheta_{f^{-1}}^{\natural} \vdash_S \exists x \in X. \vartheta_{f^{-1}}^{\natural}$. Now, by Theorem 16, $\vartheta \vdash_S (\exists x \in X. \vartheta_{f^{-1}}^{\natural})(\tau_f)$, thus $\vartheta \vdash_S \exists x \in X. \vartheta_{f^{-1}}^{\natural}$.

Since u is not free in $\exists x \in X. \vartheta_{f^{-1}}^{\natural}$, by [1, 3.7.2], it follows that $\exists u \vartheta \vdash_S \exists x \in X. \vartheta_{f^{-1}}^{\natural}$ and $\vdash_S \exists u \vartheta = \exists x \in X. \vartheta_{f^{-1}}^{\natural}$ $\blacktriangleleft$

4 Functional representability

The main purpose of this section is to provide proofs for Theorems A and C. This section is a more *set theoretical* counterpart to the previous section which analyzed the logical aspects of the reparameterizations studied therein.

► **Theorem A.** *Let $X = \{x_{\mathbf{A}} : \alpha\}$ and $Y = \{y_{\mathbf{B}} : \beta\}$ be objects in $\mathcal{T}(S)$ and let i_X and i_Y be the inclusions $(x_{\mathbf{A}} \mapsto x_{\mathbf{A}}) : X \rightarrow U_{\mathbf{A}}$ and $(y_{\mathbf{B}} \mapsto y_{\mathbf{B}}) : Y \rightarrow U_{\mathbf{B}}$, respectively. Then,*

$$\vdash_{\text{Th}(\mathcal{T}(S))} \langle i_X(u), i_Y(\text{Symb}(f)(u)) \rangle \in |\mathcal{T}(\eta_S)f|. \quad (8)$$

That is, f itself is the only syntactic function for which the following diagram commutes in $\mathcal{T}(\text{Th}(\mathcal{T}(S)))$:

$$\begin{array}{ccc} U_X & \xrightarrow{r_X} & \mathcal{T}(\eta_S)X \\ \downarrow (x \mapsto \text{Symb}(f)(x)) & & \downarrow \mathcal{T}(\eta_S)f \\ U_Y & \xrightarrow{r_Y} & \mathcal{T}(\eta_S)Y, \end{array} \quad (9)$$

where r_X is the corestriction of $(u \mapsto i_X(u))$ to $\mathcal{T}(\eta_S)X$.

Moreover, the inclusion functor $\iota_{\mathcal{T}(S)}$ and the logical functor induced by the canonical translation η_S are isomorphic:

$$\mathcal{T}(\eta_S) \cong \iota_{\mathcal{T}(S)}. \quad (10)$$

► **Remark 19.** Since $\vdash_{\text{Th}(\mathcal{T}(S))} \exists! v. \langle i_X(u), i_Y(v) \rangle \in |f|$, by [1, 3.36], f is the unique S -function that satisfies (8).

The following results will be needed in the proof of Theorem A. Their conclusions are the expected.

► **Lemma 20.** *Let $\mathcal{E}$ be a topos and α be a formula in $\text{Th}(\mathcal{E})$ with a single free variable. Then, $\vdash_{\text{Th}(\mathcal{E})} x \in \{x : \alpha\} \Leftrightarrow \exists u. x = i(u)$, where i is $\overline{\llbracket \alpha \rrbracket}_x$.*

Proof. Since $i = \overline{\llbracket \alpha \rrbracket}_x$, it follows that $\llbracket \alpha \rrbracket_x = \chi(i)$. From which,

$$\llbracket \chi(i)(x) \rrbracket_x = \chi(i) \circ \llbracket x \rrbracket_x = \chi(i) = \llbracket \alpha \rrbracket_x = \llbracket x \in \{x : \alpha\} \rrbracket_x \quad (\text{by (25)});$$

wherefrom, by [1, 3.23.3], $\models_{H_{\mathcal{E}}} \chi(i)(x) \Leftrightarrow x \in \{x : \alpha\}$. Then, $\vdash_{\text{Th}(\mathcal{E})} \chi(i)(x) \Leftrightarrow x \in \{x : \alpha\}$, and thus, $\vdash_{\text{Th}(\mathcal{E})} \exists u. x = i(u) \Leftrightarrow x \in \{x : \alpha\}$. $\blacktriangleleft$

► **Lemma 21.** *Let S be a local set theory in a local language $\mathcal{L}$. Let X be an S -set determined by the formula α . Then,*

$$\vdash_{\text{Th}(\mathcal{T}(S))} \alpha(x/i_X(u)), \quad (32)$$

with u a variable of type $\mathbf{X}$; and thus

$$\vdash_{\text{Th}(\mathcal{T}(S))} i_X(u) \in X. \quad (33)$$

14:18 Functional Representability in Local Set Theories

Furthermore, one has that the arrow $r_X : U_{\mathbf{X}} \rightarrow X$, the corestriction of $(u \mapsto \mathbf{i}_{\mathbf{X}}(u))$ to X , is an isomorphism. (Recall that $\mathbf{X}$ is the type symbol associated with X in $\mathcal{L}(\mathcal{T}(S))$ and that $\mathbf{i}_{\mathbf{X}}$ is a function symbol not present in $\mathcal{L}$).

Proof. By Lemma 4, the diagram (24) is a pullback diagram in $\mathcal{T}(S)$, and hence in $\mathcal{T}(\text{Th}(\mathcal{T}(S)))$ (where x is a variable of type $\mathbf{A}$, since the following translation is being used: $\eta_S x_{\mathbf{A}} = x_{\eta_S \mathbf{A}} = x_{\mathbf{A}}$, and $\llbracket \alpha \rrbracket_{K_S, x} = \llbracket \alpha \rrbracket_{H_{\mathcal{T}(S)}, x}$, by (27)).

Now, since $\llbracket \alpha \rrbracket_{H_{\mathcal{T}(S)}, x} \circ r = i_X = \llbracket \mathbf{i}_{\mathbf{X}}(u) \rrbracket_{H_{\mathcal{T}(S)}, u}$ for some isomorphism r in $\mathcal{T}(S)$, (32) and (33) follow from [1, 3.34]. Lastly, to see that r_X is an isomorphism, define $|h|$ as the $\text{Th}(\mathcal{T}(S))$ -set $\{\langle x, u \rangle : \mathbf{i}_{\mathbf{X}}(u) = x\}$. As the notation suggests, h is a function and, thus it is clearly the inverse of r_X . Indeed, by Lemma 20 and the fact that i_X is monic, $\vdash_{\text{Th}(\mathcal{T}(S))} x \in \{x : \alpha\} \Leftrightarrow \exists ! u. x = \mathbf{i}_{\mathbf{X}}(u)$; wherefrom, $\vdash_{\text{Th}(\mathcal{T}(S))} x \in X \Rightarrow \exists ! u. x = \mathbf{i}_{\mathbf{X}}(u)$, and thus h is a $\text{Th}(\mathcal{T}(S))$ -function $X \rightarrow U_{\mathbf{X}}$. $\blacktriangleleft$

Proof of Theorem A. Under the canonical translation η_S , one can regard S as a subtheory of $\text{Th}(\mathcal{T}(S))$ and the associated logical functor $\mathcal{T}(\eta_S) : \mathcal{T}(S) \hookrightarrow \mathcal{T}(\text{Th}(\mathcal{T}(S)))$ as the inclusion. Let $f : X \rightarrow Y$ be an arbitrary S -function and denote by $\mathbf{f}$ its corresponding function symbol **Symb**(f) in $\text{Th}(\mathcal{T}(S))$.

The first step of the proof is to verify that the following diagram in $\mathcal{T}(\text{Th}(\mathcal{T}(S)))$ is commutative:

$$\begin{array}{ccc}
 X & \xrightarrow{f} & Y \\
 \downarrow (x \mapsto x) & & \downarrow (y \mapsto y) \\
 & & B \\
 & & \downarrow (z \mapsto \{z\}) \\
 A & \xrightarrow{(x \mapsto \{y : \langle x, y \rangle \in |f|\})} & PB.
 \end{array} \tag{34}$$

In other words, this amounts to $\vdash_{\text{Th}(\mathcal{T}(S))} |(x \mapsto \{y : \langle x, y \rangle \in |f|\})| = |(y \mapsto \{y\}) \circ f|$. To see this, first observe that

$$\begin{aligned}
 \langle x, z \rangle \in |f|, u \in \{z : \langle x, z \rangle \in |f|\} &\vdash_{\text{Th}(\mathcal{T}(S))} \langle x, z \rangle \in |f| \wedge \langle x, u \rangle \in |f| \\
 &\vdash_{\text{Th}(\mathcal{T}(S))} u = z && \text{(by Proposition 1)} \\
 &\vdash_{\text{Th}(\mathcal{T}(S))} u \in \{z\},
 \end{aligned}$$

and that

$$\begin{aligned}
 \langle x, z \rangle \in |f|, u \in \{z\} &\vdash_{\text{Th}(\mathcal{T}(S))} \langle x, z \rangle \in |f| \wedge u = z \\
 &\vdash_{\text{Th}(\mathcal{T}(S))} \langle x, u \rangle \in |f| \\
 &\vdash_{\text{Th}(\mathcal{T}(S))} u \in \{z : \langle x, z \rangle \in |f|\};
 \end{aligned}$$

so that, by the equivalence and extensionality rules,

$$\langle x, z \rangle \in |f| \vdash_{\text{Th}(\mathcal{T}(S))} \{z : \langle x, z \rangle \in |f|\} = \{z\}. \tag{35}$$

Since

$$\langle z, b \rangle \in |(y \mapsto \{y\})| \vdash_{\text{Th}(\mathcal{T}(S))} b = \{z\}, \tag{36}$$

it follows from (36) and (35) that

$$\langle x, z \rangle \in |f| \wedge \langle z, b \rangle \in |(y \mapsto \{y\})| \vdash_{\text{Th}(\mathcal{T}(S))} b = \{z : \langle x, z \rangle \in |f|\};$$

thus, by [1, 3.7.2], $\exists z(\langle x, z \rangle \in |f| \wedge \langle z, b \rangle \in |(y \mapsto \{y\})|) \vdash_{\text{Th}(\mathcal{T}(S))} b = \{z : \langle x, z \rangle \in |f|\}$. By (17) and the cut rule,

$$\langle x, b \rangle \in |(y \mapsto \{y\}) \circ f| \vdash_{\text{Th}(\mathcal{T}(S))} \langle x, b \rangle = \langle x, \{z : \langle x, z \rangle \in |f|\} \rangle.$$

So that, by

$$\begin{aligned} \langle x, b \rangle \in |(y \mapsto \{y\}) \circ f| \vdash_{\text{Th}(\mathcal{T}(S))} x \in X \\ \vdash_{\text{Th}(\mathcal{T}(S))} \langle x, \{z : \langle x, z \rangle \in |f|\} \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})|, \end{aligned}$$

it follows that

$$\langle x, b \rangle \in |(y \mapsto \{y\}) \circ f| \vdash_{\text{Th}(\mathcal{T}(S))} \langle x, b \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})|. \quad (37)$$

To see the converse of (37), notice that

$$\langle x, z \rangle \in |f| \vdash_{\text{Th}(\mathcal{T}(S))} z \in Y \vdash_{\text{Th}(\mathcal{T}(S))} \langle z, \{z\} \rangle \in |(y \mapsto \{y\})|$$

yields

$$\langle x, z \rangle \in |f| \vdash_{\text{Th}(\mathcal{T}(S))} \langle x, z \rangle \in |f| \wedge \langle z, \{z\} \rangle \in |(y \mapsto \{y\})|. \quad (38)$$

Now, since $\langle x, b \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})| \vdash_{\text{Th}(\mathcal{T}(S))} b = \{z : \langle x, z \rangle \in |f|\}$, it follows from (35) and (38) that

$$\langle x, b \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})|, \langle x, z \rangle \in |f| \vdash_{\text{Th}(\mathcal{T}(S))} \langle x, z \rangle \in |f| \wedge \langle z, b \rangle \in |(y \mapsto \{y\})|;$$

wherefrom, by [1, 3.7.1],

$$\begin{aligned} \langle x, b \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})|, \langle x, z \rangle \in |f| \vdash_{\text{Th}(\mathcal{T}(S))} \\ \exists z(\langle x, z \rangle \in |f| \wedge \langle z, b \rangle \in |(y \mapsto \{y\})|), \end{aligned}$$

and thus $\langle x, b \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})|, \langle x, z \rangle \in |f| \vdash_{\text{Th}(\mathcal{T}(S))} \langle x, b \rangle \in |(y \mapsto \{y\}) \circ f|$, so that, by [1, 3.7.2],

$$\langle x, b \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})|, \exists z(\langle x, z \rangle \in |f|) \vdash_{\text{Th}(\mathcal{T}(S))} \langle x, b \rangle \in |(y \mapsto \{y\}) \circ f|.$$

However,

$$\begin{aligned} \langle x, b \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})| \vdash_{\text{Th}(\mathcal{T}(S))} x \in X \\ \vdash_{\text{Th}(\mathcal{T}(S))} \exists! z(\langle x, z \rangle \in |f|) \\ \vdash_{\text{Th}(\mathcal{T}(S))} \exists z(\langle x, z \rangle \in |f|), \end{aligned}$$

and thus

$$\langle x, b \rangle \in |(x \mapsto \{z : \langle x, z \rangle \in |f|\})| \vdash_{\text{Th}(\mathcal{T}(S))} \langle x, b \rangle \in |(y \mapsto \{y\}) \circ f|. \quad (39)$$

Now, by the equivalence rule applied to (37) and (39), it follows that (34) commutes.

14:20 Functional Representability in Local Set Theories

Under the natural interpretation of $\text{Th}(\mathcal{T}(S))$ in $\mathcal{T}(S)$ – denoted by $\llbracket \tau \rrbracket_{\mathbf{x}} = \llbracket \tau \rrbracket_{H_{\mathcal{T}(S)}, \mathbf{x}}$, where τ is a term of $\text{Th}(\mathcal{T}(S))$ – it follows from (23) and (27) that the following diagram is the same as (34) and that it also commutes:

$$\begin{array}{ccc}
 X & \xrightarrow{f} & Y \\
 \downarrow 1 & & \downarrow \llbracket \{\mathbf{i}_{\mathbf{Y}}(v)\} \rrbracket_v \\
 X & \xrightarrow{\llbracket \{y : \langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f|\} \rrbracket_u} & PB; \\
 \searrow \llbracket \mathbf{i}_{\mathbf{X}}(u) \rrbracket_u & & \nearrow \llbracket \{y : \langle x, y \rangle \in |f|\} \rrbracket_x \\
 & A &
 \end{array}$$

Since $\llbracket \mathbf{f}(u) \rrbracket_u = f$, it follows that $\llbracket \{y : \langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f|\} \rrbracket_u = \llbracket \{\mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u))\} \rrbracket_u$; and, by [1, 3.23.3],

$$\models_{H_{\mathcal{T}(S)}} \{y : \langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f|\} = \{\mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u))\}.$$

Therefore, by (25), $\vdash_{\text{Th}(\mathcal{T}(S))} \{y : \langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f|\} = \{\mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u))\}$. Thus, since $\{\mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u))\}$ is by definition $\{y : y = \mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u))\}$, by [1, 3.4.3],

$$\vdash_{\text{Th}(\mathcal{T}(S))} y = \mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u)) \Leftrightarrow \langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f|. \quad (40)$$

From this,

$$\begin{aligned}
 \langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f| &\vdash_{\text{Th}(\mathcal{T}(S))} y = \mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u)) \wedge \langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f| \\
 &\vdash_{\text{Th}(\mathcal{T}(S))} \langle \mathbf{i}_{\mathbf{X}}(u), \mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u)) \rangle \in |f|.
 \end{aligned}$$

So that, by [1, 3.7.2], $\exists y(\langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f|) \vdash_{\text{Th}(\mathcal{T}(S))} \langle \mathbf{i}_{\mathbf{X}}(u), \mathbf{i}_{\mathbf{Y}}(\mathbf{f}(u)) \rangle \in |f|$.

On the other hand, by (33),

$$\vdash_{\text{Th}(\mathcal{T}(S))} \mathbf{i}_{\mathbf{X}}(u) \in X \vdash_{\text{Th}(\mathcal{T}(S))} \exists! y(\langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f|) \vdash_{\text{Th}(\mathcal{T}(S))} \exists y(\langle \mathbf{i}_{\mathbf{X}}(u), y \rangle \in |f|).$$

wherefrom, by the cut rule, (8) follows.

Equation (8) is equivalent to the commutativity of (9) in $\mathcal{T}(\text{Th}(\mathcal{T}(S)))$. The horizontal isomorphisms are clearly natural. $\blacktriangleleft$

Lastly, the proof of Theorem C is divided in the next two theorems.

► **Theorem 22.** *Let S be a local set theory in a local language $\mathcal{L}$. Let $f : X \rightarrow Y$ be an S -function, and let γ be a formula that determines $|f|$. Then*

$$\vdash_{\text{Th}(\mathcal{T}(S))} (\gamma(x/\mathbf{i}_{\mathbf{X}}(u), y/\mathbf{i}_{\mathbf{Y}}(v)))_{r_X^{-1} \times r_Y^{-1}}^{\natural} = \gamma. \quad (41)$$

Furthermore,

$$(\{\langle u, v \rangle : \gamma(x/\mathbf{i}_{\mathbf{X}}(u), y/\mathbf{i}_{\mathbf{Y}}(v))\}, U_{\mathbf{X}}, U_{\mathbf{Y}})$$

is a $\text{Th}(\mathcal{T}(S))$ -function henceforth denoted by f^* .

Proof. Recall from Lemma 21 that r_X and r_Y are the corestrictions of $(u \mapsto \mathbf{i}_{\mathbf{X}}(u))$ and $(v \mapsto \mathbf{i}_{\mathbf{Y}}(v))$ to X and Y , respectively. Denote by $|f^*|$ the set $\{\langle u, v \rangle : \gamma(x/\mathbf{i}_{\mathbf{X}}(u), y/\mathbf{i}_{\mathbf{Y}}(v))\}$.

The equation (41) follows from the fact that the following is a pullback of pullbacks:

$$\begin{array}{ccccc}
 |f^*| & \xrightarrow{\cong} & |f| & \xrightarrow{\quad} & 1 \\
 \downarrow i_{|f^*|} & & \downarrow j_{|f|} & & \downarrow \top \\
 U_X \times U_Y & \xrightarrow[r_X \times r_Y]{\cong} & X \times Y & \xrightarrow{i_X \times i_Y} & A \times B \xrightarrow{(\langle x, y \rangle \mapsto \gamma)} \Omega, \\
 & \searrow & \nearrow & \searrow & \nearrow \\
 & & & \langle (u, v) \mapsto \gamma(i_X(u), i_Y(v)) \rangle &
 \end{array}$$

$\langle (u, v) \mapsto \gamma(i_X(u), i_Y(v)) \rangle$
 $\langle (u, v) \mapsto \gamma(i_X(u), i_Y(v)) \rangle$

the proof of which is *mutatis mutandis* already within the proof of Theorem 16. A direct application of Theorem 18 proves that f^* is a function. ◀

► **Theorem 23.** *Let S be a local set theory in a local language $\mathcal{L}$. Let X and Y be S -sets, with α and β the formulae determining them, resp., and let $f : X \rightarrow Y$ be an S -function with γ the formula that determines $|f|$. Let f^* be the function obtained in Theorem 22. Then,*

$$\vdash_{Th(\mathcal{T}(S))} \langle u, \mathbf{f}(u) \rangle \in |f^*|. \quad (42)$$

In other words, that $f^* = (u \mapsto \mathbf{f}(u))$.

Proof. It is immediate that $\langle i_X(u), i_Y(\mathbf{f}(u)) \rangle \in |f| \vdash_{Th(\mathcal{T}(S))} \langle u, \mathbf{f}(u) \rangle \in |f^*|$, thus (42) follows by Theorem A. Finally, since by Lemma 22 both are functions, the conclusion now follows. ◀

References

- 1 J. L. Bell. *Toposes and local set theories*, volume 14 of *Oxford Logic Guides*. The Clarendon Press, Oxford University Press, New York, 1988. An introduction, Oxford Science Publications.
- 2 P. T. Johnstone. *Topos theory*. Academic Press [Harcourt Brace Jovanovich, Publishers], London-New York, 1977. London Mathematical Society Monographs, Vol. 10.
- 3 Anders Kock. *Synthetic differential geometry*, volume 333 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, second edition, 2006. doi:10.1017/CB09780511550812.
- 4 J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1986.
- 5 René Lavendhomme. *Basic concepts of synthetic differential geometry*, volume 13 of *Kluwer Texts in the Mathematical Sciences*. Kluwer Academic Publishers Group, Dordrecht, 1996. Translated from the 1987 French original, Revised by the author. doi:10.1007/978-1-4757-4588-7.
- 6 F. William Lawvere. Axiomatic cohesion. *Theory Appl. Categ.*, 19:No. 3, 41–49, 2007.
- 7 Colin McLarty. Elementary axioms for canonical points of toposes. *J. Symbolic Logic*, 52(1):202–204, 1987. doi:10.2307/2273873.
- 8 Colin McLarty. *Elementary categories, elementary toposes*, volume 21 of *Oxford Logic Guides*. The Clarendon Press, Oxford University Press, New York, 1992. Oxford Science Publications. doi:10.1093/oso/9780198533924.001.0001.
- 9 Enrique Ruiz Hernández and Pedro Solórzano. On logical parameterizations and functional representability in local set theories, 2021. arXiv:2104.07405.
- 10 Enrique Ruiz-Hernández and Pedro Solórzano. Connectedness through decidable quotients. *Cah. Topol. Géom. Différ. Catég.*, 66(3):65–78, 2025.

Mendler Dialgebras and Recursion Schemes of Mixed Variance

Stephan Alexander Spahn  

Department of Computer Science, Tallinn University of Technology, Estonia

Abstract

We introduce the notion of *Mendler dialgebra* and provide a categorical semantics of recursion schemes of mixed variance in its terms. The Mendler-style approach – which employs second-order inference rules – to recursion schemes of mixed variance was championed by Uustalu and Vene in [48] using dinatural transformations, and we generalize their methods to include a variety of new recursion schemes including those presented by Ahn-Sheard in [3], and Stump et al. in [45]. We give sufficient criteria for reducibility of Mendler dialgebras to Lambek algebras [27] which correspond to first-order inference rules. A similar reduction in elementary terms for the special case of the systems studied in [48] had been given by the authors, but our approach differs in that we use the language of two-sided fibrations [43] to express the reduction for our generalization. This reveals that the “diagonal” of every two-sided fibration with a fibered initial object is isomorphic to a category of Lambek algebras; a dual version concerning reduction to Lambek coalgebras, as well as to bialgebras (inserters), [28] is given. We also discuss various properties and examples of Mendler dialgebras, including higher-order abstract syntax which is paradigmatic for definitions of mixed variance. Generally, we regard the paper as a contribution to a more systematic understanding of the relation between higher-order and first-order inference rules.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning

Keywords and phrases Mendler Algebra, Dinatural Transformation, Structured Recursion Scheme, Grothendieck Fibration, Higher-Order Abstract Syntax

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.15

Acknowledgements I would like to thank Aaron Stump, Christopher Jenkins, Fredrik Nordvall Forsberg, Tarmo Uustalu, and Pawel Sobocinski for discussions about Mendler algebras as well as comments on the draft.

1 Introduction

Mendler algebras were introduced in context of second-order lambda calculus in [35] as part of a recursion scheme for iteration on inductively defined types. While there are many specialized forms of recursion (e.g. iteration, primitive recursion, recursion allowing for certain nestings in the recursive calls), and assorted notions of *algebra*, one can distinguish two major styles of recursive definitions; speaking categorically¹ these are: recursion where the signature f is explicitly required to be functorial, the *Lambek style* [27], and recursion where functoriality is no explicit requirement, the *Mendler style* [35].

To explain the difference, let $f : \mathcal{C} \rightarrow \mathcal{C}$ be an endofunctor on some category $\mathcal{C}$, and recall that a *Lambek algebra* is a pair (X, α) consisting of an object X (the carrier) of $\mathcal{C}$, and a morphism $\alpha : f(X) \rightarrow X$. By contrast, a *Mendler algebra* is a pair (X, ϕ) where again $X : \mathcal{C}$, but $\phi : [-, X] \rightarrow [f-, X]$ is a natural transformation between the hom functors $[-, X]$ and $[f-, X]$. A morphism between such Lambek, respectively Mendler, algebras is a morphism

¹ The same distinction can be discussed in set theory as we will briefly do in Section 2, and in syntactic deduction calculi.



© Stephan Alexander Spahn;

licensed under Creative Commons License CC-BY 4.0

31st International Conference on Types for Proofs and Programs (TYPES 2025).

Editors: Fredrik Nordvall Forsberg and James McKinna; Article No. 15; pp. 15:1–15:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

between the carriers commuting with the algebra structures (respectively given by α and ϕ) in the obvious way. Speaking 2-categorically, the difference is thus between algebra-structures as 1-morphisms vs. algebra-structures as 2-morphisms.

These notions of algebra give rise to different forms of recursion equations: For Lambek algebras, it has the form $h \circ in = \alpha \circ f(h)$ asking – for fixed (X, α) – for a unique solution h ; this is expressed diagrammatically on the right below as the unique morphism from the initial Lambek f -algebra $(\mu f, in)$ (whenever it exists) to (X, α) .

For Mendler algebras we again consider the unique morphism from the initial Mendler algebra $(\mu f, in)$ to some Mendler algebra (X, ϕ) , and thus obtain a priori a family of sets of equations (for each A , one for each $g : A \rightarrow \mu f$), but by choosing the component μf (diagram on the left) and evaluating in the identity on μf (diagram in the middle), we obtain a canonical single equation where $h : \mu f \rightarrow X$ is the unique solution of the recursion equation $h \circ in_{\mu f}(id) = \phi_{\mu f}(h)$.

$$\begin{array}{ccc}
 [\mu f, \mu f] & \xrightarrow{[\mu f, h]} & [\mu f, X] \\
 \downarrow in_{\mu f} & & \downarrow \phi_{\mu f} \\
 [f(\mu f), \mu f] & \xrightarrow{[f(\mu f), h]} & [f(\mu f), X]
 \end{array}
 \qquad
 \begin{array}{ccc}
 f(\mu f) & \xrightarrow{in_{\mu f}(id)} & \mu f \\
 \searrow \phi_{\mu f}(h) & & \downarrow h \\
 & & X
 \end{array}
 \qquad
 \begin{array}{ccc}
 f(\mu f) & \xrightarrow{in} & \mu f \\
 \downarrow f(h) & & \downarrow h \\
 f(X) & \xrightarrow{\alpha} & X
 \end{array}$$

In the parlance of natural-deduction calculi, commutation of the middle, respectively of the right, diagram corresponds to the *computation rule*. In particular we see that in case of Lambek algebras the signature f needs to be functorial (at least in the morphism h it is applied to) while in the Mendler case this is not a requirement since f is not applied to any morphism. This shows an advantage of the Mendler style at least if one is working with a syntactic deduction calculus since in such a calculus there is a priori no rule asserting functoriality of the signature, and for higher-order signatures such as in System F_ω there can be fundamental obstacles to establish functoriality at all (see “syntactic restriction” and “monotonicity witness” in [2, §3.4, p.19; §5, p.31]). While in category theory, the Yoneda lemma² tells us that the categories of Mendler f -algebras and that of Lambek f -algebras are isomorphic, it was recognized in [48] that one can adapt the definition of Mendler algebra to difunctors $f : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$. A Mendler algebra for such an “endo-difunctor” is now a pair (X, ϕ) where again $X : \mathcal{C}$, but $\phi : [Idd(-, =), X] \rightarrow [f(-, =), X]$ is a *dinatural transformation* [9] (see also Definition 1), and $Idd : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ is the difunctor ignoring the first argument and acting as the identity on the second. This mixed-variant notion of Mendler algebra for $f : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ is not in general equivalent to that of a Lambek algebra³ for an endofunctor on $\mathcal{C}$. In [48], a sufficient criterion was also given for when Mendler algebras of the variety considered there can be reduced to algebras for an endofunctor on $\mathcal{C}$.

In Section 2 we give a motivation for the notion of Mendler algebra in terms of set-theoretic induction, illustrative of the difference between the two styles of inductive definition. In Section 3 we recall necessary definitions and give examples of other types of Mendler algebras. In Section 4 we define a category of *diindexed dinatural transformations* (*ddts*) and show that it is the total category of a two-sided fibration; this structure will be the key for the reduction

² There does not seem to be an original textual reference for this “arguably [...] most important result in category theory” [41], but according to [31] it was proved by Nobuo Yoneda in 1954; the obituary [30] relates the story how it became known. The lemma and its variants (see Example 4) play an important rôle in the theory of Mendler algebras.

³ Although there is a notion of *Freyd dialgebra* [14] (see also Definition 23) for f that (under some conditions) can be reduced to a Lambek algebra for an endofunctor on $\mathcal{C}^{op} \times \mathcal{C}$.

of *Mendler dialgebras* (see Definition 16) – which will by definition lie in the “diagonal” of the category of ddt’s – to Lambek algebras as we will explain in Section 5. In Section 6 we will give some more examples of Mendler dialgebras – in particular for modelling higher-order abstract syntax – and for reduction to Lambek algebras. In Section 7 we point to some differences regarding initial algebra semantics for Mendler dialgebras compared to Lambek algebras. In Section 8 we make concluding remarks and formulate some ideas for future work related to Mendler dialgebras.

The contributions of our paper are: We introduce the notion of Mendler dialgebra, give a general categorical semantics of recursion schemes of mixed variance in its terms, and establish a correspondence between Mendler dialgebras and Lambek algebras under a certain sufficient criterion. From the mathematical point of view, we observe that the same arguments more generally establish a relation between categories of Lambek algebras and “diagonals” of two-sided fibrations with fibered initial object.

2 Background: Relation To Induction In Set Theory

Uustalu and Vene [50] ask: “What are the origins of the Mendler style and why the name? It seems that the technique was invented by N. P. Mendler, in type theory.”

The origin of the idea might appear less mysterious when realizing that the difference between a Lambek style and a Mendler style is present in set-theoretic induction as well:

An *inductive definition* (see e.g. [36], [40, p.83–84]) is defined in the following way with respect to a fixed set A .

1. A *clause* is a pair (X, x) where $X \subseteq A$, and $x \in A$.
2. A signature σ is a set of clauses.
3. For a signature σ we define a predicate $cl(\sigma, -)$ on the powerset of A by (for $Y \subseteq A$)

$$cl(\sigma, Y) :\Leftrightarrow (\forall X. (X \subseteq Y) \rightarrow (\{x \in A \mid (X, x) \in \sigma\} \subseteq Y))$$

In words: ‘ Y is closed under σ ’, and we write $\Sigma(X) := \{x \mid (X, x) \in \sigma\}$.

4. The *inductive set* defined by the signature σ is defined by $Ind(\sigma) := \bigcap \{Y \mid Y \subseteq A, cl(\sigma, Y)\}$.

If we assume proof terms, this is to say that $\tilde{\sigma} : cl(\sigma, Y)$ is a proof that Y is closed under σ . The notation

$$\tilde{\sigma} : \forall X. (X \subseteq Y) \rightarrow (\Sigma(X) \subseteq Y)$$

expresses then essentially (in particular if “ $\subseteq$ ” is generalized to any preorder or morphism “ $\rightarrow$ ” in a category) that the pair $(Y, \tilde{\sigma})$ is a *Mendler algebra* with *carrier* Y , and $Ind(\sigma)$ is (the carrier of) the initial such Mendler algebra.

It is then usually shown that a signature σ can equivalently be expressed as a monotone (with respect to inclusion) endofunction ϕ_σ on the powerset of A , namely $\phi_\sigma : Y \mapsto \{x \mid x \in A, \exists X \subseteq Y. x \in \Sigma(X)\}$ and its least fixpoint as the inductive set it determines; conversely for any monotone endofunction $\phi : P(A) \rightarrow P(A)$ of the powerset of A , we obtain a signature as the set $\sigma_\phi := \{(X, y) \mid X \subseteq A, y \in \phi(X)\}$.

A comparison in the situation of a deduction calculus can be found in [48].

3 Preliminaries

We assume the reader has some familiarity with category theory even though we include those definitions from categorical logic [21] essential for the main argument. We let throughout $\mathcal{V}$ denote the category of sets and (total) functions. For hom-sets we use angular brackets $[,]$ if

no confusion can arise. We call a functor of the form $f : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{D}$ for some categories $\mathcal{C}$ and $\mathcal{D}$ a $(\mathcal{C}, \mathcal{D})$ -bifunctor. We sometimes denote the identity morphism on an object A by A itself.

► **Definition 1** (Dinatural transformation [9]). *For bifunctors $F, G : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{D}$, a dinatural transformation from F to G consists of a $\mathcal{C}$ -indexed family τ of morphisms in $\mathcal{D}$ such that for all $h : b \rightarrow a$ diagrams of the following form commute.*

$$\begin{array}{ccccc}
 & & F(a, a) & \xrightarrow{\tau_a} & G(a, a) \\
 & \nearrow^{F(a, h)} & & & \searrow^{G(h, a)} \\
 F(a, b) & & & & G(b, a) \\
 & \searrow_{F(h, b)} & & & \nearrow_{G(b, h)} \\
 & & F(b, b) & \xrightarrow{\tau_b} & G(b, b)
 \end{array}$$

Natural transformations are examples of dinatural transformations between bifunctors that both ignore the contravariant variable. But unlike natural transformations which are morphisms in functor categories, dinatural transformations cannot in general constitute the class of morphisms of any category⁴ since they cannot (generally) be composed – a “defect” that is rectified by the following more restrictive definition due to Barr [38].

► **Definition 2** (Strong dinatural transformation). *For bifunctors $F, G : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{D}$, a strong dinatural transformation from F to G consists of a $\mathcal{C}$ -indexed family σ of morphisms in $\mathcal{D}$ satisfying the following condition: for every $a : \mathcal{C}$ it holds: for every $h : b \rightarrow a$, and every triple (Q, p_1, p_2) , if the left square commutes, so does the hexagon.*

$$\begin{array}{ccccc}
 & & F(a, a) & \xrightarrow{\sigma_a} & G(a, a) \\
 & \nearrow^{p_1} & \searrow^{F(h, a)} & & \searrow^{G(h, a)} \\
 Q & & F(b, a) & & G(b, a) \\
 & \searrow^{p_2} & \nearrow^{F(b, h)} & & \nearrow^{G(b, h)} \\
 & & F(b, b) & \xrightarrow{\sigma_b} & G(b, b)
 \end{array}$$

In case $\mathcal{D}$ has pullbacks, for a given $h : b \rightarrow a$, the premise in the defining condition, that for all triples (Q, p_1, p_2) the left square commutes, can of course be replaced by the premise that the left square is a pullback. Every strong dinatural transformation is also a dinatural transformation.

Adjacent strong dinatural transformations can always be composed and since the ensuing category is interesting and important, we introduce a notation for it:

► **Definition 3** ($sd_{(\mathcal{C}, \mathcal{D})}$). *For categories $\mathcal{C}, \mathcal{D}$, the category $sd_{(\mathcal{C}, \mathcal{D})}$ of strong dinatural transformations has $(\mathcal{C}, \mathcal{D})$ -bifunctors as objects, strong dinatural transformations as morphisms, and the identity dinatural transformation – which is strong – as identity. The composition operation is given by pointwise composition of the families.*

⁴ Bifunctors and dinatural transformations form, however, a paracategory in the sense of [19, 20].

Compositionality of dinatural transformations (see [42] for a comprehensive discussion) is e.g. of interest when bifunctors serve as models for types in a type theory and terms are modelled by dinatural transformations (see [8]). To have a category of (certain) dinatural transformations will also simplify some of our arguments later on (see Proposition 12). Beyond their virtue of compositionality, strong dinatural transformations are interesting for a variety of other reasons. Interesting for us is the connection to (initial) Lambek algebras, for, if the domain F is of the form $[G-, =]$ for some endofunctor G , the premise in the condition for strong dinaturality expresses exactly that the morphisms $h : b \rightarrow a$ figuring in the dinaturality condition (see Definition 1) are homomorphisms of G -Lambek algebras; strong dinaturality has also been used to define a notion of *strongly weighted limit* [46] to prove a generalization of Tarski's "non-constructive" fixpoint theorem. We have the following Yoneda-like statements:

► **Example 4** ([47]). For $\mathcal{V}$ -enriched $\mathcal{C}$ we have:

1. $sd_{(\mathcal{C}, \mathcal{V})}([F-, =], [C, =]) \simeq Cone_{U_F}(C)$ for the set of cones with vertex C of the forgetful functor U_F from the category of Lambek algebras for $F : \mathcal{C} \rightarrow \mathcal{C}$.
2. $sd_{(\mathcal{C}, \mathcal{V})}([H-, =], K-) \simeq K(\mu H)$ for $H : \mathcal{C} \rightarrow \mathcal{C}$, $K : \mathcal{C} \rightarrow \mathcal{V}$.
3. $sd_{(\mathcal{C}, \mathcal{V})}([- , H], K-) \simeq K(\nu H)$ for $H : \mathcal{C} \rightarrow \mathcal{C}$, $K : \mathcal{C}^{op} \rightarrow \mathcal{V}$.
4. $sd_{(\mathcal{C}, \mathcal{V})}([F-, =], Idd(-, =)) \simeq \mu F^5$ for $F : \mathcal{C} \rightarrow \mathcal{C}$.

In [47] the question was asked whether the statements in the previous example had already been observed elsewhere. We find the following:

► **Remark 5** (Kelly's strong Yoneda lemma). Let $K : \mathcal{C} \rightarrow \mathcal{V}$ and define $\tilde{K} : Alg(T)^{op} \times \mathcal{C} \rightarrow \mathcal{V}$ by $\tilde{K}((X, a), Y) := K(Y)$. Then Kelly's formula [25, (2.38)], $\int_{B : \mathcal{B}} K(B, JB) \simeq [\mathcal{B}^{op} \times \mathcal{A}, \mathcal{V}](\mathcal{A}(J-, =), K(-, =))$, can be instantiated by taking $\mathcal{B} := Alg(T)$, $\mathcal{A} := \mathcal{C}$, $J := U : Alg(T) \rightarrow \mathcal{C}$ the functor forgetting the algebra structure, and $K := \tilde{K}$. With this choice, we obtain

$$K(\mu T) \simeq \int_{(X, a) : Alg(T)} K(X) \simeq [Alg(T)^{op} \times \mathcal{C}, \mathcal{V}]([U-, =], \tilde{K}(-, =)).$$

Indeed, Kelly's formula gives the right-hand isomorphism, while the left-hand isomorphism is the composite

$$K(\mu T) \simeq [Alg(T), \mathcal{V}](\Delta_1, K \circ U) \simeq \int_{(X, a) : Alg(T)} K(X),$$

where $\Delta_1 : Alg(T) \rightarrow \mathcal{V}$ is the constant functor at the singleton set.

Another justification for the significance of the definition of strong dinatural transformations is that the Church numerals, i.e. the set of dinatural transformations from a hom functor to itself sending an endomorphism to its n -fold iteration for some natural number n are the only strong dinatural transformations between these functors (see [17]).

We officially give the central definition of Mendler (co)algebras and their morphisms that we already mentioned in the introduction.

► **Definition 6** (Classical (covariant) Mendler (co)algebra [35]). *A Mendler algebra for an endofunctor f on a category $\mathcal{C}$ is a pair (X, ϕ) where X is an object of $\mathcal{C}$, and $\phi : [-, X] \rightarrow [f-, X]$ is a natural transformation. A morphism between such Mendler f -algebras from*

⁵ Where here again Idd is the identity functor supplemented by an ignored contravariant argument.

(X, ϕ) to (Y, ψ) consists of a morphism $m : X \rightarrow Y$ such that for all A squares of the form on the left commute. Dually, a Mendler coalgebra is a pair (X, ζ) where $X : \text{ob}(\mathcal{C})$ is an object, and $\zeta : [X, -] \rightarrow [X, f-]$ is a natural transformation; a morphism $(X, \zeta) \rightarrow (Y, \gamma)$ consists of a morphism $m : X \rightarrow Y$ such that for all A the squares of the form on the right commute.

$$\begin{array}{ccc} [A, X] & \xrightarrow{[A, m]} & [A, Y] \\ \downarrow \phi_A & & \downarrow \psi_A \\ [fA, X] & \xrightarrow{[fA, m]} & [fA, Y] \end{array} \quad \begin{array}{ccc} [Y, A] & \xrightarrow{[m, A]} & [X, A] \\ \downarrow \gamma_A & & \downarrow \zeta_A \\ [Y, fA] & \xrightarrow{[m, fA]} & [X, fA] \end{array}$$

Even though, as mentioned in the introduction, the categories of classical Mendler f -(co)algebras are isomorphic to the categories of Lambek f -algebras, their syntactic difference matters in practice as one could discuss by writing out an appropriate natural deduction calculus with fixpoint operator.

In [48] it was remarked that the definition of Mendler algebra also makes sense for endobifunctors, i.e. functors of mixed variance of the form $f : \mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathcal{C}$ and Mendler f -algebras of the form $(X, \phi : [\text{Id}(-, =), X] \rightarrow [f(-, =), X])$ where ϕ is a dinatural transformation. In this case it is however not true anymore that the categories of Mendler f -algebras are always equivalent to the category of Lambek f -algebras – not even for Lambek g -algebras for a possibly different endofunctor g . Nevertheless it was given – under the condition that certain initial objects exist – a construction of an endofunctor g on $\mathcal{C}$ such that the category of Mendler f -algebras is isomorphic to the category of Lambek g -algebras.

In this paper we will generalize the results of [48] to Mendler-style algebras of a more general form that we call *Mendler dialgebras*, and characterize the reduction of Mendler dialgebras to Lambek algebras in terms of universal properties of a category of *diindexed dinatural transformations* which contains the category of Mendler dialgebras as a subcategory (the “diagonal”). Our main tool for this reduction will be (Grothendieck) (op)fibrations.

► **Definition 7** ((Grothendieck) (op)fibration [44][21]). Let $p : \mathcal{A} \rightarrow \mathcal{J}$ be a functor.

1. a. A morphism t is cartesian over $p(t)$ if for all $j : A \rightarrow C$ for which there exists a morphism $\text{lift}_{p(t)}p(j)$ such that $p(t) \circ \text{lift}_{p(t)}p(j) = p(j)$ (i.e. a lift through its image), there exists a unique $\text{Lift}_t j$ such that $t \circ \text{Lift}_t j = j$, and $p(\text{Lift}_t j) = \text{lift}_{p(t)}p(j)$

$$\begin{array}{ccc} & B & \\ \text{Lift}_t j \nearrow & \downarrow t & \\ A & \xrightarrow{j} & C \end{array} \quad \begin{array}{ccc} & p(B) & \\ \text{lift}_{p(t)}p(j) \nearrow & \downarrow p(t) & \\ p(A) & \xrightarrow{p(j)} & p(C) \end{array}$$

- b. p is called a fibration if every morphism of the form $u : J \rightarrow p(C)$ has a cartesian lift, i.e. there is a cartesian morphism t such that $p(t) = u$. A choice, for every morphism $u : J \rightarrow p(C)$, of a cartesian morphism t with codomain C and $p(t) = u$ is called a cleavage, and a fibration is called split if this choice preserves identities and composition.
2. a. A morphism t is opcartesian over $p(t)$ if for all j for which there exists some $\text{ext}_{p(t)}p(j) : p(B) \rightarrow p(A)$ with $\text{ext}_{p(t)}p(j) \circ p(t) = p(j)$, there exists a unique $\text{Ext}_t j$ such that $p(\text{Ext}_t j) = \text{ext}_{p(t)}p(j)$, and $\text{Ext}_t j \circ t = j$.

$$\begin{array}{ccc} C & \xrightarrow{j} & A \\ t \downarrow & \nearrow \text{Ext}_t j & \\ B & & \end{array} \quad \begin{array}{ccc} p(C) & \xrightarrow{p(j)} & p(A) \\ p(t) \downarrow & \nearrow \text{ext}_{p(t)}p(j) & \\ p(B) & & \end{array}$$

- b. p is called an *opfibration* if every morphism of the form $u : p(C) \rightarrow I$ has an *opcarteresian* lift, i.e. there is an *opcarteresian* morphism t such that $p(t) = u$. A choice, for every morphism $u : p(C) \rightarrow I$, of an *opcarteresian* morphism t with domain C and $p(t) = u$ is called an *opcleavage*, and an *opfibration* is called *opsplit* if this choice preserves identities and composition.

A morphism is called *p-vertical* (for an (op)fibration p) if it is sent to an identity by p .

Examples of Mendler Algebras

The main part of this paper will be dedicated to the reduction of Mendler-style algebras to Lambek (co)algebras and properties of this reduction. To give the reader nevertheless some more sense of the utility of the concept of Mendler algebra, we review some well-known examples of Mendler algebras; we will see some more examples later in Section 6. For now, a Mendler-style algebra is a pair of a carrier and a dinatural transformation; all these variant Mendler algebras will be instances of *Mendler dialgebras* that we will define later on.

► **Example 8** (Primitive (co)recursion [34]). A Mendler-style algebra for *primitive recursion* has algebras of the form (X, ϕ) , where the natural transformation is of the form $\phi : [-, X] \times [-, \mu f] \rightarrow [f-, X]$ for a covariant endofunctor f and the carrier μf of its initial (Lambek or conventional Mendler) algebra. The corresponding scheme for primitive corecursion is obtained by switching the arguments of the hom-functors and taking the carrier νf of the terminal f -coalgebra instead of μf .

An example illustrating that Mendler algebras give some legitimacy to the “hate-love” relationship between algebra and coalgebra [23, p.x] by subsuming both aspects is *course-of-value iteration*.

► **Example 9** (Mendler-style course-of-value iteration (aka histomorphism)[49, 50, 51]). A Mendler-style algebra for *course-of-value iteration* with carrier X is a pair $(X, \phi : [-, X \times F =] \rightarrow [F-, X])$ where ϕ is required to be strong in [50]⁶ for reasons that will be explained shortly.

One can read off the terminal sequence $1 \leftarrow X \times F(1) \leftarrow X \times F(X \times F(1)) \leftarrow X \times F(X \times F(X \times F(1))) \leftarrow \dots$ (whenever it exists) of the functor $F^X(A) := X \times F(A)$ how all “earlier” arguments – the “course of values of F ” – are made available for reference in the recursion defined by the initial Mendler algebra of this kind.

To a course of values $\zeta_C : F(\nu F^C) \rightarrow C$ corresponds via Example 4.3, a strong dinatural transformation with components $\eta_A^{\zeta_C} : (h : A \rightarrow C \times FA) \mapsto \mathcal{C}(F(\llbracket h \rrbracket), C)(\zeta_C)$ where $\llbracket h \rrbracket$ is the coiterative function defined by the F^C -coalgebra h . The cofree comonad on F has the underlying functor $C \mapsto \nu F^C$. Denoting its counit by ϵ , we obtain a course of values as the composite $\zeta_{\mu F} : F(\nu F^{\mu F}) \xrightarrow{F(\epsilon_{\mu F})} F(\mu F) \xrightarrow{in} \mu F$ on μF (where in is the unique isomorphism provided by Lambek’s lemma [27]) which we can thus regard as canonical. Hence, there is a unique morphism (induced by an $i_{(\mu F, \eta^{\zeta_{\mu F}})} : C_0 \rightarrow \mu F$) of course-of-value Mendler algebras $(C_0, In) \rightarrow (\mu F, \eta^{\zeta_{\mu F}})$, from the initial such Mendler algebra. We can furthermore for any course of values $\zeta_C : F(\mu F^C) \rightarrow C$ derive recursion equations by asking for unique morphisms $(\mu F, \eta^{\zeta_{\mu F}}) \rightarrow (C, \eta^{\zeta_C})$.

⁶ It is not explicitly required to be strong in [3] but there is an implicit assumption of parametricity, which implies (under conditions) strong dinaturality (see [10, 11]); for the implication to dinaturality see [39].

Examples of recursive functions definable as histomorphisms on the natural numbers are Fibonacci numbers, and the even-number predicate where the “next” value can be computed from the previous two, respectively the immediately preceding, value.

There is a wealth of other examples of Mendler-style algebras (some of which we will discuss later) and in all these cases it is of interest whether they can be reduced to Lambek algebras. Thus, the main concern of this paper is firstly to determine a general form of Mendler-style algebra, and secondly to give a sufficient criterion for a reduction of Mendler-style algebras to Lambek algebras, as well as to clarify the mathematics underlying this reduction.

4 The Two-Sided Fibration Of Diindexed Dinatural Transformations

The results in this section will be used in the next section to provide a sufficient criterion under which Mendler dialgebras (defined in the next section) can be reduced to Lambek algebras. This criterion will be formulated in terms of certain (Grothendieck) (op)fibrations. We begin with the definition of the appropriate total category which will contain the category of Mendler dialgebras as a subcategory.

► **Definition 10** (Diindexed (strong) dinatural transformation). *For functors $F, G : \mathcal{C}^{op} \times \mathcal{C} \times \mathcal{J} \rightarrow \mathcal{V}$, we define the category $\mathcal{W}$ of diindexed dinatural transformations (ddts):*

1. *An object of $\mathcal{W}$ is a pair $((X, Y), \delta)$ where $X, Y : \mathcal{J}$, and $\delta : F(-, =, X) \rightarrow G(-, =, Y)$ is a dinatural transformation.*
2. *A morphism $h : ((X, Y), \delta) \rightarrow ((X', Y'), \delta')$ is a pair of morphisms $h_0 : X \rightarrow X'$, and $h_1 : Y \rightarrow Y'$ such that for each $a : \mathcal{C}$ the following diagram commutes*

$$\begin{array}{ccc} F(a, a, X) & \xrightarrow{F(a, a, h_0)} & F(a, a, X') \\ \delta_a \downarrow & & \downarrow \delta'_a \\ G(a, a, Y) & \xrightarrow{G(a, a, h_1)} & G(a, a, Y') \end{array}$$

The definition of the category $\mathcal{W}^s$ of diindexed strong dinatural transformations is as above except that we additionally require the participating dinatural transformations to be strong.

We could make the above definition more general by allowing for functors of more than two variables (see [32]) and/or use polycategories (see [7]) to cover models of sequent calculi with fixpoint operators, but the examples we presently seek to cover do not imply the necessity to do so.

The following generalization of Grothendieck fibrations is appropriate for our situation of mixed variance. Two-sided fibrations are also called fibered spans in [43] [21]. Compared to Definition 7 we use from now on the usual notations “ $f^*(e)$ ” for domains of cartesian lifts, and “ $g_!(e)$ ” for codomains of opcartesian ones.

► **Definition 11** (Two-sided fibration [43][29]). *A span $\mathcal{A} \xleftarrow{q} \mathcal{M} \xrightarrow{p} \mathcal{B}$ in the 2-category $\mathcal{Cat}$ of categories (or any 2-category) is a two-sided fibration if:*

1. *Any morphism of the form $f : b \rightarrow p(e)$ has a cartesian lift with codomain e that is q -vertical.*
2. *Any morphism of the form $g : q(e) \rightarrow a$ has an opcartesian lift with domain e that is p -vertical.*

3. Given a cartesian lift $f^*(e) \rightarrow e$ of f , and an opcartesian lift $e \rightarrow g_!(e)$ of g , it follows from the previous two conditions that the composite $f^*(e) \rightarrow e \rightarrow g_!(e)$ lies over f via p , and over g via q . In this situation, the induced morphism $g_!f^*e \rightarrow f^*g_!e$ is required to be an isomorphism.

► **Proposition 12.** *Let $\mathcal{W}$ be the category defined in Definition 10, $p((X, Y), \delta) := X$, $q((X, Y), \delta) := Y$, $q(h_0, h_1) := h_1$. Then $\mathcal{J} \xleftarrow{q} \mathcal{W} \xrightarrow{p} \mathcal{J}$ is a two-sided fibration. p as well as q are split.*

Proof. By [43] (see also [53, Example 2.1][29]) every lax pullback in the category of (small) categories carries the structure of a fibered span and there is an obvious way to exhibit $\mathcal{W}$ as a lax pullback (every such fibered span is a discrete fibered span but we will not need to make use of this additional fact). In Appendix A we also spelled out an explicit proof. ◀

We denote the p -fiber, respectively the q -fiber, over an object X by $\mathcal{W}_{X-}$, respectively $\mathcal{W}_{-X}$; the simultaneous (p, q) -fiber over X and Y we denote by $\mathcal{W}_{XY}$, and analogous definitions for $\mathcal{W}^s$.

We move on to the last ingredient we still need for the reduction of Mendler dialgebras to Lambek algebras.

(Op)fibered initial (terminal) objects

Provided what we have seen so far, a sufficient criterion for the reduction to Lambek (co)algebras will be that p have a fibered initial object (respectively that q has an opfibered terminal object). The following characterization will be useful for us (for the terminal case, it can be found in [21, Lemma 1.8.8, p.85]).

► **Remark 13 ((Op)fibered initial (terminal) object).** 1. (Fibered initial object) Let $p : \mathcal{T} \rightarrow \mathcal{B}$ denote a fibration, and for $X : \mathcal{B}$ let $0_{\mathcal{T}_X}$ denote the initial object in the p -fiber over X . The following are equivalent:

- a. For each X , $0_{\mathcal{T}_X}$ exists and is preserved by reindexing; i.e. for the p -cartesian lifting situation

$$\frac{Cart(h, 0_{\mathcal{T}_{X'}}) : h^*(0_{\mathcal{T}_{X'}}) \rightarrow 0_{\mathcal{T}_{X'}}}{h : X \rightarrow X'} \downarrow p$$

there is an isomorphism $i : 0_{\mathcal{T}_X} \rightarrow h^*(0_{\mathcal{T}_{X'}})$. In particular, composing the isomorphism $i : 0_{\mathcal{T}_X} \rightarrow h^*(0_{\mathcal{T}_{X'}})$ with the chosen p -cartesian lift $h^*(0_{\mathcal{T}_{X'}}) \rightarrow 0_{\mathcal{T}_{X'}}$ yields a morphism $0_{\mathcal{T}_X} \rightarrow 0_{\mathcal{T}_{X'}}$, which is p -cartesian.

- b. There is a fibered left adjoint $\mathbf{0} \dashv p$ from the terminal fibration over $\mathcal{B}$ to p .

$$\begin{array}{ccc} & \mathbf{0} & \\ \mathcal{T} & \xleftarrow{\quad} & \mathcal{B} \\ & \searrow p & \swarrow id \\ & \mathcal{B} & \end{array}$$

$t(=p)$

The $\mathbf{0} \dashv p$ -counit has as components the canonical morphisms $\iota_w : \mathbf{0}(p(w)) \rightarrow w$ and these are p -vertical.

2. (Opfibered terminal object) Let $q : \mathcal{T} \rightarrow \mathcal{B}$ denote an opfibration, and for $X : \mathcal{B}$, let $1_{\mathcal{T}_X}$ denote the terminal object in the q -fiber over X . The following are equivalent:

- a. For each X , $1_{\mathcal{T}_X}$ exists and is preserved by opreindexing; i.e. for the q -opcartesian lifting situation

$$\frac{opCart(h, 1_{\mathcal{T}_X}) : 1_{\mathcal{T}_X} \rightarrow h_!(1_{\mathcal{T}_X})}{h : X \rightarrow X'} \downarrow q$$

15:10 Mendler Dialgebras

there is an isomorphism $i : q_!(1_{\mathcal{T}_X}) \rightarrow 1_{\mathcal{T}_{X'}}$. In particular, composing the chosen q -opcartesian lift $1_{\mathcal{T}_X} \rightarrow h_!(1_{\mathcal{T}_X})$ with the isomorphism $i : q_!(1_{\mathcal{T}_X}) \rightarrow 1_{\mathcal{T}_{X'}}$ yields a morphism $1_{\mathcal{T}_h} : 1_{\mathcal{T}_X} \rightarrow 1_{\mathcal{T}_{X'}}$, which is q -opcartesian.

- b. There is an opfibered right adjoint $q \dashv \mathbf{1}$ from the terminal fibration over $\mathcal{B}$ to q

$$\begin{array}{ccc} E & \xrightleftharpoons[t(=q)]{} & B \\ & \searrow q & \swarrow id \\ & B & \end{array}$$

The $q \dashv \mathbf{1}$ -unit has as components the canonical morphisms $\pi : w \rightarrow \mathbf{1}(q(w))$ and these are q -vertical.

For simplicity, we sometimes identify $\mathbf{1}(h : X \rightarrow X')$ with the opcartesian lift of h with respect to $1_{\mathcal{T}_X}$.

► **Remark 14.** For ease of later reference we summarize the adjunctions $\mathcal{J} \xrightleftharpoons[p]{\mathbf{0}} \mathcal{W} \xrightleftharpoons[q]{\mathbf{1}} \mathcal{J}$.

1. For the $\mathbf{0} \dashv p$ adjunction
 - a. The unit $\eta : id \rightarrow p\mathbf{0}$ is the identity natural transformation.
 - b. The counit $\iota : \mathbf{0}p \rightarrow id$ is not (generally) the identity, but it is p -vertical.
2. For the $q \dashv \mathbf{1}$ adjunction
 - a. The unit $\pi : id \rightarrow \mathbf{1}q$ is not (generally) the identity, but it is q -vertical.
 - b. The counit $\epsilon : q\mathbf{1} \rightarrow id$ is the identity.

Recall that each canonical map ι_w is p -vertical, and each canonical map π_w is q -vertical. (It is neither generally true that ι_w is q -vertical, nor that π_w is p -vertical.)

For later reference, we spell out:

► **Lemma 15.**

1. Assuming p -fibered initial objects, for all $h : ((X, X), \delta) \rightarrow ((X', X'), \delta')$ the diagrams of the form on the left below commute.
2. Assuming q -opfibered terminal objects, for all $h : ((X, X), \delta) \rightarrow ((X', X'), \delta')$ the diagrams of the form on the right below commute.

$$\begin{array}{ccc} q(0_{\mathcal{W}_{X-}}) & \xrightarrow{q(\iota_{((X, X), \delta)})} & X \\ \downarrow q(0_{\mathcal{W}_{h_0-}}) & & \downarrow h_0 \\ q(0_{\mathcal{W}_{X'-}}) & \xrightarrow{q(\iota_{((X', X'), \delta')})} & X' \end{array} \quad \begin{array}{ccc} X & \xrightarrow{p(\pi_{((X, X), \delta)})} & p(1_{\mathcal{W}_{X-}}) \\ \downarrow h_0 & & \downarrow p(0_{\mathcal{W}_{h_0-}}) \\ X' & \xrightarrow{p(\pi_{((X', X'), \delta')})} & p(1_{\mathcal{W}_{X'-}}) \end{array}$$

Proof. (1) Apply q to

$$\begin{array}{ccc} \mathbf{0}(p((X, X), \delta)) & \xrightarrow{\iota_{((X, X), \delta)}} & ((X, X), \delta) \\ \downarrow \mathbf{0}(p(h)) & & \downarrow h \\ \mathbf{0}(p((X', X'), \delta')) & \xrightarrow{\iota_{((X', X'), \delta')}} & ((X', X'), \delta') \end{array}$$

The dual statement (2) about counits can be seen analogously. ◀

5 Reduction Of Mendler Dialgebras To Lambek (Co)algebras

We are now ready to explain the reducibility of Mendler dialgebras to Lambek algebras; this generalizes and clarifies results from [48].

► **Definition 16** (Mendler Dialgebra). *We denote the “diagonal” of $\mathcal{W}$, i.e. the subcategory of $\mathcal{W}$ with objects w satisfying $p(w) = q(w)$, and morphisms h satisfying $p(h) = q(h)$, by $\text{Alg}^{\mathcal{W}}$, and call this the category of Mendler dialgebras.*

For the following to be well-defined, we need to assume that (op)fibered initial (terminal) objects are chosen (as opposed to be merely existent). For the reduction of Mendler dialgebras to

Lambek algebras, we assume that p has a chosen fibered initial object. (5.1)

Lambek coalgebras, we assume that q has a chosen opfibered terminal object. (5.2)

the endofunctors for which Mendler dialgebras (under the given assumptions) will be Lambek algebras are defined as follows:

► **Definition 17.**

1. *Provided (5.1), we define a functor*

$$\begin{aligned}\underline{w} : \mathcal{J} &\rightarrow \mathcal{J} \\ \underline{w}(X) &:= q(0_{\mathcal{W}_{X-}}) \\ \underline{w}(f : X \rightarrow Z) &:= q(\iota_{f*0_{\mathcal{W}_{Z-}}})\end{aligned}$$

This is well defined because $q(f*0_{\mathcal{W}_{Z-}}) = q(0_{\mathcal{W}_{Z-}})$.

2. *Provided (5.2), we define a functor*

$$\begin{aligned}\underline{m} : \mathcal{J} &\rightarrow \mathcal{J} \\ \underline{m}(X) &:= p(1_{\mathcal{W}_{-X}}) \\ \underline{m}(g : X \rightarrow Z) &:= p(\pi_{g!1_{\mathcal{W}_{-X}}})\end{aligned}$$

This is well defined because $p(g!1_{\mathcal{W}_{-X}}) = p(1_{\mathcal{W}_{-X}})$.

The previous definition can be understood in terms of the category-of-elements form of representability, cf. [21, Section 1.8], as follows.

► **Remark 18** (Fibred initial objects as representability). For fixed $a \in \mathcal{J}$, write $P_a(b) := \text{Dinat}(F(-, =, a), G(-, =, b))$ where on the right dinatural transformations between the respective difunctors are meant, with functoriality in b given, for $h : b \rightarrow b'$, by post-composition along $G(-, =, h)$. Then the p -fiber $\mathcal{W}_{a-}$ is the category of elements of P_a ; hence $\mathcal{W}_{a-}$ has an initial object iff P_a is representable, i.e. iff there is an object $H(a) \in \mathcal{J}$ and a natural isomorphism $P_a(b) \cong \mathcal{J}(H(a), b)$. In that case, if $0_{\mathcal{W}_{a-}} = ((a, H(a)), \eta_a)$ is the universal element corresponding to $\text{id}_{H(a)}$, then the functor of Definition 17 satisfies $\underline{w}(a) = q(0_{\mathcal{W}_{a-}}) = H(a)$. Equivalently, the $\mathcal{V}$ -valued profunctor $P : \mathcal{J}^{\text{op}} \times \mathcal{J} \rightarrow \mathcal{V}$, defined by $P(a, b) := \text{Dinat}(F(-, =, a), G(-, =, b))$, is represented in its second variable by the endofunctor $\underline{w} : \mathcal{J} \rightarrow \mathcal{J}$.

The operations sending Mendler dialgebras to Lambek (co)algebras and vice versa are given by:

► **Definition 19.**

1. Let $((X, X), \delta) \in \text{Alg}^{\mathcal{W}}$.
 - a. Assuming that p has a fibered initial object, (5.1), we define a $\underline{w}$ -algebra by $\mathbb{A}((X, X), \delta) := (X, q(\iota_{((X, X), \delta)}))$.
 - b. Assuming that q has an opfibered terminal object, (5.2), we define a $\underline{m}$ -coalgebra by $\mathbb{C}((X, X), \delta) := (X, p(\pi_{((X, X), \delta)}))$.
2. a. Assuming that p has a fibered initial object, (5.1), we define $\mathbb{M}^{\text{alg}}(X, \alpha : \underline{w}(X) \rightarrow X) := \alpha_!(0_{\mathcal{W}_X})$.
- b. Assuming that q has an opfibered terminal object, (5.2), we define $\mathbb{M}^{\text{coalg}}(X, \alpha : X \rightarrow \underline{m}(X)) := \alpha^*(1_{\mathcal{W}_X})$.

The operations just defined all extend to functors; for 1a and 1b, the action on morphisms is spelled out in Lemma 15.

► **Remark 20 (Reduction to, and lifting of a bialgebra).** Definition 19 1a and 1b taken together, we can define a so-called $(\underline{w}, \underline{m})$ -bialgebra

$$\mathbb{D}((X, X), \delta) := (X, q(\iota_{((X, X), \delta)}), p(\pi_{((X, X), \delta)})).$$

That is, the carrier X is equipped with a $\underline{w}$ -algebra structure $q(\iota_{((X, X), \delta)}) : \underline{w}X \rightarrow X$ and a $\underline{m}$ -coalgebra structure $p(\pi_{((X, X), \delta)}) : X \rightarrow \underline{m}X$. Generally, a bialgebra consists of a carrier that is an algebra for one functor, and a coalgebra for another functor (and as such is a special form of *dialgebra* [28][18]) such that there is a natural transformation – in this context called a *distributive law* [5] – between the composite functors. In the present case, this natural transformation $d : \underline{w}\underline{m} \rightarrow \underline{m}\underline{w}$ is the composite $d = p\pi\mathbf{0} \circ \eta \circ \eta \circ q\iota\mathbf{1}$ where $\iota, \pi, \epsilon, \eta$ were defined in Remark 14 (in particular ϵ and η are both the identity). Usually, one of the functors is a monad or comonad and the transformation is additionally required to be compatible with this (co)monad structure (see [5], or [22] for a more recent reference). But in our case neither $\underline{w}$ nor $\underline{m}$ generally carry such additional structure. One can also lift a bialgebra to a Mendler dialgebra, see Appendix C.

► **Corollary 21.** *The pairs of functors defined in Definition 19 are inverses.*

Proof. See Appendix B. ◀

The previous corollary does not depend on the specifics of the category $\mathcal{W}$ but only on the two-sided fibered structure.

► **Theorem 22.** *Let $\mathcal{J} \xleftarrow{p} \mathcal{T} \xrightarrow{q} \mathcal{J}$ be any two-sided split fibration where p has a fibered initial object, respectively q has an opfibered terminal object. Then there is an endofunctor w respectively m on $\mathcal{J}$ such that the “diagonal”⁷ $\text{Alg}^{\mathcal{T}}$ of $\mathcal{T}$ is isomorphic to the category of algebras for w , respectively to the category of coalgebras for m .*

Proof. Inspection of Definition 17, Definition 19, and the proof of Corollary 21. ◀

6 Examples

All recursion schemes from [3] are examples of Mendler dialgebras. Apart from classical (covariant) Mendler algebras (as in Definition 6), the authors list the histomorphism (see Section 3), the so-called “abstract inverse” which we will explain shortly, “higher-order”

⁷ Of course defined in analogy to Definition 16 as the subcategory of $\mathcal{T}$ on which p and q coincide.

versions of all systems which are categorically simply obtained by replacing the base category by categories of endofunctors, as well as combinations of these systems which can be obtained by taking products of hom-sets (see Example 27). Since the “abstract inverse”, a Mendler variant for defining higher-order abstract syntax, is a paradigmatic example for recursion schemes of mixed variance, we will focus on this example.

Freyd dialgebras and higher-order abstract syntax

The significance of *higher-order abstract syntax*, i.e. the definition of a datatype whose terms are to be interpreted as lambda terms of some object language is to use meta-language variables in place of object-language variables which simplifies certain routines such as capture-avoiding substitution [52, p.1]. A motivating example is the untyped lambda calculus where the signature functor $T : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ is of mixed variance and defined by $T(X, Y) = (X \rightarrow Y) + (Y \times Y) + V$ for some fixed set of variables V . Since it is not possible to define a category of Lambek algebras of the form $(X, \alpha : T(X, X) \rightarrow X)$ as there is no appropriate notion of morphism, Freyd [14] (see also [13]) defined the following notion which served as a basis for further developments in encoding higher-order abstract syntax in [33], [52] and [3].

► **Definition 23** (Freyd dialgebra). *A Freyd dialgebra for a difunctor $T : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ is a pair $(X, \alpha : TXX \rightarrow X, \alpha' : X \rightarrow TXX)$. A homomorphism of Freyd dialgebras is a pair $(m : X \rightarrow Y, m' : Y \rightarrow X) : (X, \alpha, \alpha') \rightarrow (Y, \beta : TYY \rightarrow Y, \beta' : Y \rightarrow TYY)$ such that $(\beta \circ Tm'm = m \circ \alpha) \wedge (Tmm' \circ \beta' = \alpha' \circ m')$.*

► **Remark 24** (Freyd Dialgebras vs. Mendler Dialgebras). Freyd dialgebras (α, α') give rise to (strong) Mendler dialgebras of the form $[\alpha', -] \times [-, \alpha] : [T(X, X), -] \times [-, T(X, X)] \rightarrow [X, -] \times [-, X]$, and morphisms of Mendler dialgebras induced by a morphism $(m, m') : (X, X) \rightarrow (Y, Y)$ in $\mathcal{C}^{op} \times \mathcal{C}$ are exactly those pairs (m, m') constituting morphisms of Freyd dialgebras. We display the correspondence for morphisms in the following diagrams showing how the morphism (m, m') gives rise to a family of maps of the form $(\psi, \psi') \mapsto (\psi \circ T(m, m'), T(m', m) \circ \psi', (h, h') \mapsto (h \circ m', m \circ h')$ natural in A constituting the vertical maps in the upper square where $\delta_A := [\alpha', -] \times [-, \alpha]$ and $\gamma_A := [\beta', -] \times [-, \beta]$ (and (h, h') which is not in the picture factors for verification of commutation as $(\psi \circ \alpha', \alpha \circ \psi')$).

$$\begin{array}{ccc}
 [TXX, A] \times [A, TXX] & \xrightarrow{\delta_A} & [X, A] \times [A, X] \\
 \downarrow & & \downarrow \\
 [TYY, A] \times [A, TYY] & \xrightarrow{\gamma_A} & [Y, A] \times [A, Y]
 \end{array}$$

$$\begin{array}{ccc}
 \forall A \forall \psi'. \quad \begin{array}{ccc} TXX & \xleftarrow{\psi'} A & \xrightarrow{\psi'} TXX \\ \downarrow \alpha & & \downarrow Tm'm \\ X & \xrightarrow{m} Y & \xleftarrow{\beta} TYY \end{array} & iff & \begin{array}{ccc} TXX & \xrightarrow{Tm'm} & TYY \\ \alpha \downarrow & & \downarrow \beta \\ X & \xrightarrow{m} & Y \end{array} \\
 \forall A \forall \psi. \quad \begin{array}{ccc} TYY & \xleftarrow{\beta'} Y & \xrightarrow{m'} X \\ Tmm' \downarrow & & \downarrow \alpha' \\ TXX & \xrightarrow{\psi} A & \xleftarrow{\psi} TXX \end{array} & iff & \begin{array}{ccc} Y & \xrightarrow{m'} & X \\ \beta' \downarrow & & \downarrow \alpha' \\ TYY & \xrightarrow{Tmm'} & TXX \end{array}
 \end{array}$$

Meijer and Hutton proposed in [33] an implementation of Freyd dialgebras as a realization of higher-order abstract syntax. Given the involutive nature of Freyd dialgebras (see [13]), defining a recursive function out of the “initial Freyd dialgebra” (also called a minimal invariant object) amounts to simultaneously defining a corecursive function into it. In applications to higher-order abstract syntax this is not the desired formulation: the corecursive contribution is intended to be determined by the recursive part.

Fegaras and Sheard [12], and later Washburn and Weirich [52], address this issue by using a form of abstract inverse, relying on parametricity to control the additional structure. Categorically, this should be read against the standard relation between relational parametricity, naturality, and dinaturality: dinaturality follows from relational parametricity in the sense of Plotkin–Abadi [39], while the categorical role of natural and dinatural transformations in polymorphism goes back at least to Bainbridge–Freyd–Scedrov–Scott [4]. For the relation with strong dinaturality, see also [10].

Three issues meet in this example. The first is higher-order abstract syntax itself, which introduces mixed-variance signatures. The second is the use of relational parametricity in the programming-language literature to control the abstract inverse. The third is the categorical distinction between dinatural and ordinary natural transformations. We will not develop the relational parametric semantics here. The point relevant for the present paper is instead that the schemes in [3] are written in a polymorphic naturality-style form, while the basic mixed-variance Mendler algebra is naturally formulated as a dinatural transformation. We therefore recall the Dubuc–Street comparison between dinaturality and ordinary naturality.

► **Remark 25 (Dinaturality vs. naturality).**

1. We recall the Dubuc–Street comparison between dinaturality and ordinary naturality. Let $H, K : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{V}$ be bifunctors, with $\mathcal{C}$ small and with the required ends existing in $\mathcal{V}$. Dubuc and Street associate to K a bifunctor $\bar{K} : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{V}$ defined by $\bar{K}(A, B) := \int_C (\mathcal{C}(B, C) \times \mathcal{C}(C, A)) \multimap K(C, C)$. In the present paper $\mathcal{V}$ is the category of sets, so the cotensor $\multimap$ is simply the function-set construction $(-) \rightarrow (-)$. Thus, in our notation,

$$\bar{K}(A, B) = \int_C (\mathcal{C}(B, C) \times \mathcal{C}(C, A) \rightarrow K(C, C)).$$

The Dubuc–Street theorem [9, Theorem 2] gives a natural isomorphism $dn_{(\mathcal{C}, \mathcal{V})}(H, K) \simeq n(H, \bar{K})$, where dn denotes dinatural transformations and n ordinary natural transformations. Hence a dinatural transformation into K may equivalently be presented as an ordinary natural transformation into the Dubuc–Street transform $\bar{K}$.

2. The most basic mixed-variance Mendler algebra is obtained by taking $f : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ and, for a carrier $X \in \mathcal{C}$, considering a dinatural transformation $[Idd(-, =), X] \rightarrow [f(-, =), X]$, i.e. an element of $dn_{(\mathcal{C}, \mathcal{V})}([Idd(-, =), X], [f(-, =), X])$. This is the elementary instance of the Mendler-dialgebra form considered in this paper.

Applying the Dubuc–Street comparison to $K(A, B) := [f(A, B), X]$ rewrites this dinatural datum as an ordinary natural transformation $[Idd(-, =), X] \rightarrow \bar{K}$, where

$$\bar{K}(A, B) = \int_C (\mathcal{C}(B, C) \times \mathcal{C}(C, A) \rightarrow [f(C, C), X]).$$

This rewrite into naturality form is to be borne in mind when reading the scheme $\Pi_a [X, aX] \times [aX, X] \rightarrow [f(aX), X]$ found in [3] (where a is an additional parameter that can act on the carrier). That scheme is not literally the basic mixed-variance Mendler algebra above, but a variant of the basic scheme with an additional parameter. In the

framework of the present paper, such variants are expressed as Mendler dialgebras by separating positive and negative occurrences into the appropriate domain and codomain data. The point of the Dubuc–Street comparison here is only to explain why the dinatural formulation used in this paper may appear in [3] in naturality-style, polymorphic notation.

In context of higher-order abstract syntax one finds the terminology of the “abstract inverse” which can be explained as follows.

► **Example 26** (Abstract inverse [3, 12]). For $f : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$, $F, G : (\mathcal{C} \rightarrow \mathcal{C})^{op} \times (\mathcal{C} \rightarrow \mathcal{C}) \times \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{V}$ are defined by $F(a, a', X, X') := [X, a'X'] \times [aX, X']$, and $G(a, a', X, X') := [f(a'X', aX), X']$. The action on morphisms is induced by the hom-functors.

Evaluating the initial Mendler dialgebra of this type over the identity we obtain the diagrammatic recursion equation on the left, i.e. the canonical morphism from the initial Mendler dialgebra of this signature to an arbitrary Mendler dialgebra ϕ of this signature (shown is the component over (h', h)).

$$\begin{array}{ccccc}
 f(\mu f, \mu f) & \xrightarrow{in_{id}(id, id)} & \mu f & & Exp \times Exp \xrightarrow{ev \times ev} Val \times Val & & [Exp, Exp] \xrightarrow{[ev, un]} [Val, Val] \\
 \downarrow f(h', h) & & \downarrow h & & \downarrow a & & \downarrow a' \\
 f(X, X) & \xrightarrow{\phi_{(h \circ h', h \circ h')}} & X & & Exp \xrightarrow{ev} Val & & Exp \xrightarrow{ev} Val \\
 & & & & & & \downarrow l' \\
 & & & & & & Val
 \end{array}$$

Instantiating with the motivating example, i.e. with the signature $f(X', X) := (X' \rightarrow X) + (X \times X)$, we obtain (by taking the coproduct apart) the familiar recursion equations for an interpreter ev of the syntax of untyped lambda calculus with application and abstraction (displayed in the middle and right square above) where we denoted the constructors of Exp by a and l , the algebra structure of Val by a' and l' , and ev denotes evaluation and un the corresponding map in the opposite direction, as customary in the context of the abstract inverse.

For the sake of completeness, we mention that we can obtain all schemes in [3] by combining schemes we already discussed by taking products of hom functors; e.g. combining the histomorphism (which, to recall, was the Mendler dialgebra defined by $F(a, b, X) := [a, fb] \times [a, X]$ and $G(a, b, X) := [fa, X]$) with the abstract inverse:

► **Example 27** (“Histomorphism $\times$ abstract inverse”). For $f : \mathcal{C} \rightarrow \mathcal{C}$, $F, G : (\mathcal{C} \rightarrow \mathcal{C})^{op} \times (\mathcal{C} \rightarrow \mathcal{C}) \times \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{V}$ are defined by $F(a, a', X, X') := [aX, f(a'X)] \times [X, a'X'] \times [aX, X']$, and $G(a, a', X, X') := [f(aX), X']$.

An example for reduction of a scheme

For clarity and emphasis we spell out how the reduction of a scheme looks like in the basic case of a Mendler dialgebra arising from a Freyd dialgebra.

► **Remark 28** (Freyd dialgebras and the reduction to first-order algebras). We spell out the endofunctor obtained from Definition 17 in the Freyd case considered in Remark 24. Let $T : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ be a mixed-variance bifunctor and put $\mathcal{I} := \mathcal{C}^{op} \times \mathcal{C}$. We consider the following representable subcategory of the diindexed dinatural transformations of Definition 10. Its objects are triples $((X, Y), (U, V), \delta)$, where $(X, Y), (U, V) \in \mathcal{I} = \mathcal{C}^{op} \times \mathcal{C}$ and δ is a dinatural transformation

$$\delta : [T(Y, X), -] \times [-, T(X, Y)] \rightarrow [U, -] \times [-, V]$$

required to be uniquely of the form $\delta = [s', -] \times [-, s]$, for maps $s' : U \rightarrow T(Y, X)$ and $s : T(X, Y) \rightarrow V$ in $\mathcal{C}$. A morphism $((X, Y), (U, V), \delta) \rightarrow ((X', Y'), (U', V'), \delta')$ is a pair $h = (h_0, h_1)$, with $h_0 : (X, Y) \rightarrow (X', Y')$ and $h_1 : (U, V) \rightarrow (U', V')$ in $\mathcal{J}$, such that the square of Definition 10 commutes. The projections are $p((X, Y), (U, V), \delta) = (X, Y)$ and $q((X, Y), (U, V), \delta) = (U, V)$. The diagonal objects in this ambient category are the Mendler dialgebras of Definition 16; the Freyd dialgebras of Definition 23 give such diagonal objects by Remark 24.

Fix $(X, Y) \in \mathcal{J}$. The object $0_{(X, Y)} = (((X, Y), (T(Y, X), T(X, Y))), \text{id})$ lies in the ambient p -fiber over (X, Y) , since its p -projection is (X, Y) . Here id is the identity transformation

$$\text{id} : [T(Y, X), -] \times [-, T(X, Y)] \longrightarrow [T(Y, X), -] \times [-, T(X, Y)].$$

This object is not required to be a Mendler dialgebra; it is an object of the ambient p -fiber.

We claim that $0_{(X, Y)}$ is initial in this p -fiber. Let $((X, Y), (U, V), \delta)$ be another object in the same p -fiber, with $\delta = [s', -] \times [-, s]$, where $s' : U \rightarrow T(Y, X)$ and $s : T(X, Y) \rightarrow V$ in $\mathcal{C}$. A morphism $0_{(X, Y)} \rightarrow ((X, Y), (U, V), \delta)$ in the p -fiber is a morphism in the total category whose p -component is $\text{id}_{(X, Y)}$. Hence it is determined by a morphism $h' = (s', s) : (T(Y, X), T(X, Y)) \rightarrow (U, V)$ in $\mathcal{J} = \mathcal{C}^{op} \times \mathcal{C}$, i.e. by maps $s' : U \rightarrow T(Y, X)$ and $s : T(X, Y) \rightarrow V$ in $\mathcal{C}$, and the square of Definition 10 is $([s', -] \times [-, s]) \circ \text{id} = \delta$. This holds for the displayed h' , while uniqueness follows from the assumed unique carrier representation of δ .

Therefore $0_{(X, Y)}$ is initial in the ambient p -fiber over (X, Y) . Consequently, the general formula for $\underline{w} : I \rightarrow I$, of Definition 17 gives

$$\underline{w}((X, Y)) = q(0_{(X, Y)}) = (T(Y, X), T(X, Y)).$$

The action on morphisms u is given by

$$\underline{w}(u) := q(\iota u * 0_{\mathcal{W}_{(X', Y')-}}) = (T(u_2, u_1), T(u_1, u_2)) : (T(Y, X), T(X, Y)) \rightarrow (T(Y', X'), T(X', Y'))$$

where $u = (u_1, u_2) : (X, Y) \rightarrow (X', Y')$ in $\mathcal{J} = \mathcal{C}^{op} \times \mathcal{C}$.

► **Remark 29 (Freyd's diagonal category and Fiore's doubling trick).** The preceding remark shows that, in the Freyd case, the abstract functor $\underline{w}$ of Definition 17 is the endofunctor $\underline{w} : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}^{op} \times \mathcal{C}$ given by $\underline{w}((X, Y)) = (T(Y, X), T(X, Y))$. This is the same first-order endofunctor which appears in the classical reductions of mixed-variance recursive types, i.e. types specified by a difunctor of shape $T : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ to ordinary inductive types, i.e. types defined by an endofunctor.

Freyd's route [14] starts from his custom notion of dialgebra Definition 23. For a bifunctor $T : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$, a T -dialgebra is an object X with maps $f : T(X, X) \rightarrow X$ and $f' : X \rightarrow T(X, X)$. Freyd calls (E, f, f') free if, for every T -dialgebra (A, a, a') , there is a unique pair $x : E \rightarrow A$ and $x' : A \rightarrow E$ satisfying the two dialgebra morphism equations $xf = aT(x', x)$ and $f'x' = T(x, x')a'$. Thus the universal property is already two-sided at the level of morphisms. Since, however, Freyd's carriers are single objects $X \in \mathcal{C}$, he packages this two-sided morphism data by passing to the diagonal category: it has the same objects as $\mathcal{C}$, but a morphism $X \rightarrow Y$ is a pair $\mathcal{C}(X, Y) \times \mathcal{C}(Y, X)$. Equivalently, up to swapping the two components, this is the full subcategory of $\mathcal{C}^{op} \times \mathcal{C}$ on the diagonal objects (X, X) . Under this identification, the endofunctor induced by T is the diagonal restriction of the doubled functor, sending (X, X) to $(T(X, X), T(X, X))$. Freyd proves the corresponding di-Lambek lemma and relates regular free dialgebras to minimal invariant objects.

Fiore's route avoids this diagonal category and instead works on the doubled category. An involution on a category is a self-duality $D : \mathcal{D}^{op} \rightarrow \mathcal{D}$ with $D^2 = \text{id}$; the example relevant here is $\mathcal{D} = \mathcal{C}^{op} \times \mathcal{C}$, with involution exchanging the two factors. An endofunctor $S : \mathcal{D} \rightarrow \mathcal{D}$ is

symmetric if it commutes with this involution. The bifunctor $T : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ determines the symmetric endofunctor $S_T : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}^{op} \times \mathcal{C}$ defined by $S_T(X, Y) = (T(Y, X), T(X, Y))$. Fiore’s doubling trick is precisely the passage from the mixed-variance bifunctor T , which is not itself an endofunctor, to this symmetric endofunctor S_T on the involutory doubled category [13, Section 6.3].

The point of the construction is that it identifies the recursive fixed-point problem for T with an ordinary initial-algebra problem for S_T . A fixed point of T is an object X with an isomorphism $T(X, X) \cong X$. Fiore calls such a fixed point minimal if it is a retract of every other fixed point of T ; this is a categorical minimality condition, not an order-theoretic least-element condition. If $\mathcal{C}$ is parameterised algebraically compact, meaning in particular that initial algebras are free in the sense that their inverses are final coalgebras, then $\mathcal{C}^{op} \times \mathcal{C}$ is again algebraically compact and the initial S_T -algebra yields the canonical minimal fixed point of T [13, Corollary 6.2.7 and Theorem 6.4.1]. Thus Fiore does not define Freyd dialgebras and then compare them with ordinary algebras; rather, he replaces T by the doubled endofunctor S_T , and proves that the initial algebra of S_T supplies the desired canonical/minimal recursive type for T .

Our construction of $\underline{w}$ generalises Freyd’s and Fiore’s in the sense that it is not limited to signatures of the form $T : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{C}$ and does not require the ambient category $\mathcal{C}$ to be algebraically compact, but still recovers their endofunctor in that special case, namely $\underline{w}((X, Y)) = (T(Y, X), T(X, Y))$.

Example 27 was introduced in [45] to realize a nested inductive type. Generally⁸, a nested inductive type is defined in two steps: firstly, a signature functor X_0 for the nested type is itself defined by induction, and secondly the nested type is obtained as the initial algebra of the signature functor X_0 . Usually (see [6, 24]), the signature X_0 is obtained as the initial Lambek algebra for a functor H between functor categories (a “higher-order functor”), and the same can equivalently be achieved by instantiating the basic covariant Mendler scheme Definition 6 with H . The following scheme defines X_0 without explicit use of a “higher-order functor”; the signature f for the scheme has the same type as the carriers of the scheme.

► **Example 30** (Component-indexed Mendler algebra [45]). For $f : \mathcal{C} \rightarrow \mathcal{C}$, $F, G : \mathcal{C}^{op} \times \mathcal{C} \times (\mathcal{C} \rightarrow \mathcal{C}) \rightarrow \mathcal{V}$ are defined by $F(a, b, X) := [a, Xb]$ and $G(a, b, X) := [fa, Xb]$.

Assuming impredicative quantification, we can now for instance compute the functor $\underline{w}$ induced by this system where the action on objects is given by $\underline{w}(X) = \prod_{Y : \mathcal{C} \rightarrow \mathcal{C}} \prod_{\phi : [-, X] \rightarrow [f-, Y]} Y$. Indeed $((X, \prod_{Y : \mathcal{C} \rightarrow \mathcal{C}} \prod_{\phi : [-, X] \rightarrow [f-, Y]} Y), in^X)$, where $in_a^X(\alpha)(x) = (Y \mapsto \phi_a \mapsto (\phi_a(\alpha)(x)))$ is initial over X : given $((X, Y), \Psi)$, the unique morphism displayed below is induced by $i_{((X, Y), \Psi)} : \eta \mapsto \eta Y \Psi$.

$$\begin{array}{ccc} [a, Xa] & \xrightarrow{id} & [a, Xa] \\ \downarrow in_a^X & & \downarrow \Psi_a \\ [fa, \prod_{Y : \mathcal{C} \rightarrow \mathcal{C}} \prod_{\phi_a : [a, Xa] \rightarrow [fa, Ya]} Ya] & \xrightarrow{[fa, i_{((X, Y), \Psi)}]} & [fa, Ya] \end{array}$$

i.e. $\underline{w}$ is the “higher-order functor” implicit in the system and one can show that Definition 6 instantiated with $\underline{w}$ defines the same nested type as the above system.

⁸ There is a spectrum of terminologies in use and there are of course different ways in which a type can be nested: for example the type of “rose trees” and that of ordinal notations [1] fit our nomenclature here, but sometimes also initial algebras of certain higher-order functors themselves, i.e. (certain) inductive families such as “List” or “Tree”, are called “nested types” (cf. [2] and [15]).

7 Cones Revisited

This section can be contextualized in the so-called “extended initial-algebra semantics” of [24]. On the level of generality of Mendler dialgebras we can (so far) say little in the direction of constructing initial objects in this category following [26]; nevertheless we can make some observations related to the variant Yoneda lemmas in Example 4. Recall that the “Church encoding” $\Pi_X(F(X) \rightarrow X) \rightarrow X = \mu F$ of the initial Lambek F -algebra can be interpreted by restriction of the Π -type to strong dinatural⁹ transformations $sd_{(e,v)}([F-, =], Id(-, =)) \simeq \mu F$. Also recall that we generally have:

► **Remark 31.** Let $p : \mathcal{A} \rightarrow \mathcal{J}$ be a functor, let $\mathcal{A}$ have an initial object $0_{\mathcal{A}}$, and let i be the assignment, to any object w , of the canonical morphism $i_w : 0_{\mathcal{A}} \rightarrow w$. Then $(p(0_{\mathcal{A}}), p(i))$ is a limiting cone for p .

For p being the forgetful functor U_F from a category of Lambek F -algebras, the converse is true as well [16]. This converse fails¹⁰ already for dialgebras. But we still can consider the analog of $sd_{(e,v)}([F-, =], [C, =]) \simeq Cone_{U_F}(C)$, and observe the following for $p : Alg^{\mathcal{W}} \rightarrow \mathcal{J}$: Write $C/ : \mathcal{J}^{op} \times \mathcal{J} \rightarrow Cat$ for the (degenerate) pseudofunctor defined by $(X, Y) \mapsto [C, Y]$. It follows from Proposition 12 via the Grothendieck construction for two-sided fibrations [29] that $\mathcal{J} \xleftarrow{p} \mathcal{W} \xrightarrow{q} \mathcal{J}$ defines a pseudofunctor $\underline{\mathcal{W}} : \mathcal{J}^{op} \times \mathcal{J} \rightarrow Cat$.

► **Proposition 32.** A cone for the functor $p : Alg^{\mathcal{W}} \rightarrow \mathcal{J}$ with vertex C is a strong (pseudo¹¹) dinatural transformation $\Theta : \underline{\mathcal{W}} \rightarrow C/$.

The proof follows from p being *elementary* (i.e. isomorphism-reflecting and faithful). There is an obvious dual statement for cocones.

For $p, q : \mathcal{W} \rightarrow \mathcal{J}$ not restricted to the “diagonal”, the above characterization of (co)cones does not hold. An observation in this case is that assuming (op)fibered initial (terminal) objects, we can exchange (co)cones for p with (co)cones for q :

► **Remark 33.**

1. If (L, λ) is a cone for p , then $(q0(L), q(\iota) \circ \lambda)$ is a cone for q .
2. If (R, ρ) is a cocone for q , then $(p1(R), \rho \circ p(\pi))$ is a cocone for p .

8 Conclusion

We introduced the notion of Mendler dialgebra providing an extension of the existing semantics of Mendler-style recursion schemes to cover a variety of examples of schemes of mixed variance using diindexed dinatural transformations. We explained the reduction of Mendler dialgebras to Lambek (co)algebras via universal properties of two-sided fibrations with (op)fibered initial (terminal) objects, and observed that this reduction depends only on these universal properties and not on any other specifics of the total category.

We discussed the example of higher-order abstract syntax which is paradigmatic for mixed variance recursion schemes. In future we would like to explore how definitions via Mendler dialgebras relate to inductive-inductive definitions [37] which can e.g. define the syntax of type theory by a mutual definition of types and contexts, and whose semantics is given by (certain) dialgebras (aka subequalizers or inserters).

⁹ For a syntactic discussion of dinatural terms in second-order lambda calculus, see [8].

¹⁰ Still, one can always obtain a weakly initial object from a free object on this limit.

¹¹ By “pseudo dinatural transformation” we mean the appropriate notion of dinatural transformation of pseudofunctors.

We also made some observations about the forgetful functor from the category of Mendler dialgebras related to free objects in this category and would like to say more about constructing initial Mendler dialgebras in future.

References

- 1 Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005. doi:10.1016/j.tcs.2005.06.002.
- 2 Andreas Abel, Ralph Matthes, and Tarmo Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.*, 333(1-2):3–66, 2005. doi:10.1016/j.tcs.2004.10.017.
- 3 Ki Yung Ahn and Tim Sheard. A hierarchy of mendler style recursion combinators: Taming inductive datatypes with negative occurrences. *SIGPLAN Not.*, 46(9):234–246, September 2011. doi:10.1145/2034574.2034807.
- 4 E. S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990. doi:10.1016/0304-3975(90)90151-7.
- 5 Jon Beck. Distributive laws. In B. Eckmann, editor, *Seminar on Triples and Categorical Homology Theory*, pages 119–140. Springer Berlin Heidelberg, 1969.
- 6 Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction*, pages 52–67. Springer Berlin Heidelberg, 1998. doi:10.1007/BFB0054285.
- 7 Nicolas Blanco and Noam Zeilberger. Bifibrations of polycategories and classical linear logic. In *36th International Conference on Mathematical Foundations of Programming Semantics — MFPS 2020*, 2020.
- 8 Joachim de Lataillade. Dinatural terms in system F. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic in Computer Science*, pages 267–276. IEEE Computer Society, 2009. doi:10.1109/LICS.2009.30.
- 9 Eduardo Dubuc and Ross Street. Dinatural transformations. In S. MacLane, H. Applegate, M. Barr, B. Day, E. Dubuc, Phreilambud, A. Pultr, R. Street, M. Tierney, and S. Swierczkowski, editors, *Reports of the Midwest Category Seminar IV*, pages 126–137. Springer Berlin Heidelberg, 1970.
- 10 Adam Eppendahl. Parametricity and Mulry’s strong dinaturality. Technical Report 768, Department of Computer Science, Queen Mary and Westfield College, University of London, 1999.
- 11 Adam Eppendahl. Cospan diparametricity, (unpublished manuscript), 2005.
- 12 Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’96, pages 284–294, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237721.237792.
- 13 Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Cambridge University Press, USA, 1996.
- 14 Peter Freyd. Recursive types reduced to inductive types. In *Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 498–507, Los Alamitos, CA, USA, June 1990. IEEE Computer Society. doi:10.1109/LICS.1990.113772.
- 15 Neil Ghani, Lorenzo Malatesta, and Fredrik Nordvall Forsberg. Positive inductive-recursive definitions. *Log. Methods Comput. Sci.*, 11(1), 2015. doi:10.2168/LMCS-11(1:13)2015.
- 16 Neil Ghani, Tarmo Uustalu, and Varmo Vene. Build, augment and destroy, universally. In Wei-Ngan Chin, editor, *Programming Languages and Systems*, pages 327–347. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-30477-7_22.

- 17 John W. Gray. The integration of logical and algebraic types. In H. Ehrig, H. Herrlich, H. J. Kreowski, and G. Preuß, editors, *Categorical Methods in Computer Science With Aspects from Topology*, pages 16–35. Springer Berlin Heidelberg, 1989.
- 18 Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.
- 19 Claudio Hermida and Paulo Mateus. Paracategories I: internal paracategories and saturated partial algebras. *Theor. Comput. Sci.*, 309(1-3):125–156, 2003. doi:10.1016/S0304-3975(03)00135-X.
- 20 Claudio Hermida and Paulo Mateus. Paracategories II: adjunctions, fibrations and examples from probabilistic automata theory. *Theor. Comput. Sci.*, 311(1-3):71–103, 2004. doi:10.1016/S0304-3975(03)00317-7.
- 21 Bart Jacobs. *Categorical Logic and Type Theory*. Studies in logic and the foundations of mathematics. Elsevier Science, 1999.
- 22 Bart Jacobs. A bialgebraic review of deterministic automata, regular expressions and languages. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation: Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 375–404. Springer Berlin Heidelberg, 2006. doi:10.1007/11780274_20.
- 23 Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge tracts in theoretical computer science. Cambridge University Press, 2016. doi:10.1017/CBO9781316823187.
- 24 Patricia Johann and Neil Ghani. Initial algebra semantics is enough! In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 207–222. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-73228-0_16.
- 25 G. M. Kelly. Basic concepts of enriched category theory. *Reprints in Theory and Applications of Categories*, 10:vi+137, 2005. Reprint of the 1982 original.
- 26 G.M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. *Bulletin of the Australian Mathematical Society*, 22(1):1–83, 1980. doi:10.1017/S0004972700006353.
- 27 Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.
- 28 Joachim Lambek. Subequalizers. *J-Can-Math-Bull*, 13:337–350, 1970. doi:10.4153/CMB-1970-065-6.
- 29 Fosco Loregian and Emily Riehl. Categorical notions of fibration. *Expositiones Mathematicae*, 38(4):496–514, 2020. doi:10.1016/j.exmath.2019.02.004.
- 30 Saunders MacLane. The Yoneda lemma. *Mathematica Japonicae*, 47:156, 1998.
- 31 Jean-Pierre Marquis. From a Geometrical Point of View: A Study of the History and Philosophy of Category Theory. *Philosophia Mathematica*, 18(2):227–234, March 2010. doi:10.1093/philmat/nkq006.
- 32 Guy McCusker and Alessio Santamaria. On compositionality of dinatural transformations. In Dan R. Ghica and Achim Jung, editors, *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, volume 119 of *LIPICs*, pages 33:1–33:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.CSL.2018.33.
- 33 Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 324–333, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224164.224225.
- 34 N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *LICS*, 1987.
- 35 Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159–172, 1991. doi:10.1016/0168-0072(91)90069-X.

- 36 Yiannis N. Moschovakis. *Elementary Induction on Abstract Structures*. Dover Publications, 1974.
- 37 Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013. URL: <https://personal.cis.strath.ac.uk/fredrik.nordvall-forsberg/thesis/thesis.pdf>.
- 38 Robert Paré and Leopoldo Román. Dinatural numbers. *Journal of Pure and Applied Algebra*, 128:33–92, June 1998. doi:10.1016/S0022-4049(97)00036-4.
- 39 Gordon D. Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 1993. doi:10.1007/BFb0037118.
- 40 Wolfram Pohlers. *Proof theory. The first step into impredicativity. 2nd printing*. Springer, January 2009. doi:10.1007/978-3-540-69319-2.
- 41 Emily Riehl. *Category theory in context*. Dover Modern Math Originals, Courier Dover Publications, 2017.
- 42 Alessio Santamaria. *Towards a Godement Calculus for Dinatural Transformations*. PhD thesis, University of Bath, 2019.
- 43 Ross Street. Fibrations and Yoneda’s lemma in a 2-category. In Gregory M. Kelly, editor, *Category Seminar*, pages 104–133. Springer Berlin Heidelberg, 1974.
- 44 Thomas Streicher. Fibred Categories à la Jean Bénabou. *arXiv e-prints*, January 2018. arXiv:1801.02927.
- 45 Aaron Stump, Christopher Jenkins, Stephan Spahn, and Colin McDonald. Strong functional pearl: Harper’s regular-expression matcher in Cedille. In *25th ACM SIGPLAN International Conference on Functional Programming - ICFP*, 2020.
- 46 Tarmo Uustalu. Strong dinaturality and initial algebras. Presented at the 12th Nordic Workshop on Programming Theory, October 11–13, 2000, Bergen, Norway, 2000.
- 47 Tarmo Uustalu. A note on strong dinaturality, initial algebras and uniform parameterized fixpoint operators. In Luigi Santocanale, editor, *7th Workshop on Fixed Points in Computer Science, FICS 2010, Brno, Czech Republic, August 21-22, 2010*, pages 77–82. Laboratoire d’Informatique Fondamentale de Marseille, 2010. URL: <https://hal.archives-ouvertes.fr/hal-00512377/document#page=78>.
- 48 Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of Computing*, 6(3):343–361, September 1999.
- 49 Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10(1):5–26, 1999. doi:10.3233/INF-1999-10102.
- 50 Tarmo Uustalu and Varmo Vene. Coding recursion a la Mendler (extended abstract). Technical report, Department Of Computer Science, Utrecht University, 2000.
- 51 Tarmo Uustalu and Varmo Vene. The recursion scheme from the cofree recursive comonad. *Electron. Notes Theor. Comput. Sci.*, 229(5):135–157, 2011. doi:10.1016/j.entcs.2011.02.020.
- 52 Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008. doi:10.1017/S0956796807006557.
- 53 Mark Weber. Yoneda structures from 2-toposes. *Applied Categorical Structures*, 15:259–323, 2007. doi:10.1007/S10485-007-9079-2.

A Proof of Proposition 12

After observing what the (op)cartesian lifts are, the proof of Proposition 12 is a simple verification of the conditions; the interest of including it is that some of its details are needed in practice when reducing particular Mendler dialgebras to Lambek algebras (see Example 30).

1. A cleavage $Cart$ is given as follows:

for $f : Z \rightarrow p((X, Y), \delta)$, we obtain a lift $Cart(f, ((X, Y), \delta)) : f^*((X, Y), \delta) \rightarrow ((X, Y), \delta)$ by $(\delta \circ F(-, -, f), id) : ((Z, Y), \delta \circ F(-, -, f)) \rightarrow ((X, Y), \delta)$; diagrammatically:

$$\begin{array}{ccccc}
 & F(a, b, Z) & & & F(a, b, X) \\
 & \downarrow F(a, b, f) & \searrow & & \swarrow \\
 & F(a, b, X) & & F(a, a, Z) \xrightarrow{F(a, a, f)} F(a, a, X) & \\
 & \searrow & \downarrow F(a, a, f) & \downarrow \delta_a & \\
 & & F(a, a, X) & & \\
 & & \downarrow \delta_a & & \\
 & & G(a, a, Y) \xrightarrow{id} G(a, a, Y) & & \\
 & \swarrow & & \searrow & \\
 & G(b, a, Y) & & & G(b, a, Y)
 \end{array}$$

This lift is cartesian because the lifting problem

$$\begin{array}{ccc}
 p((W, Q), \delta') & & \\
 \downarrow g & \searrow p(\eta) & \\
 Z & \xrightarrow{f} & p((X, Y), \delta)
 \end{array}$$

is (uniquely) solved by

$$\begin{array}{ccccc}
 & F(a, a, W) & & & \\
 & \downarrow \delta'_a & & & \\
 & G(a, a, Q) & & & \\
 \swarrow F(a, a, g) & & \searrow \eta_a^p & & \\
 & & & & \\
 F(a, a, Z) & \xrightarrow{F(a, a, f)} & F(a, a, X) & & \\
 \downarrow \delta_a \circ F(a, a, f) & & \downarrow \delta_a & & \\
 G(a, a, Y) & \xrightarrow{id} & G(a, a, Y) & &
 \end{array}$$

2. Dually, one obtains an opcleavage by lifting $g : q((X, Y), \delta) \rightarrow Z$ to

$$(id, G(-, -, g) \circ \delta) : ((X, Y), \delta) \rightarrow ((X, Z), G(-, -, g) \circ \delta)$$

3. a. The composite morphism in question is

$$\begin{array}{ccccc}
 F(a, a, Z) & \xrightarrow{F(a, a, f)} & F(a, a, X) & \xrightarrow{id} & F(a, a, X) \\
 \downarrow \delta_a \circ F(a, a, f) & & \downarrow \delta_a & & \downarrow G(a, a, g) \circ \delta_a \\
 G(a, a, Y) & \xrightarrow{id} & G(a, a, Y) & \xrightarrow{G(a, a, g)} & G(a, a, Z)
 \end{array}$$

and we readily observe that the verticality conditions are satisfied.

b. The comparison map is even an equality:

$$\begin{aligned} g_!(f^*((X, Y), \delta)) &= ((Z, Z), (G(-, -, g) \circ \delta) \circ F(-, -, f)) \\ &= ((Z, Z), G(-, -, g) \circ (\delta \circ F(-, -, f))) = f^*(g_!(((X, Y), \delta))) \end{aligned}$$

Regarding the splitness, one verifies that

1. $Cart(id, ((X, Y), \delta)) = id_{((X, Y), \delta)}$
2. $Cart(f \circ g, (((X, Y), \delta))) = Cart(f, ((X, Y), \delta)) \circ Cart(g, f^*((X, Y), \delta))$.

The corresponding *op*-ed statement about $OpCart$ is verified analogously.

B Proof of Corollary 21

To prove Corollary 21, we show:

1. $\mathbb{M}^{alg}(\mathbb{A}(-)) \simeq id$
2. $\mathbb{A}(\mathbb{M}^{alg}(-)) \simeq id$
3. $\mathbb{M}^{coalg}(\mathbb{C}(-)) \simeq id$
4. $\mathbb{C}(\mathbb{M}^{coalg}(-)) \simeq id$

1.

$$\begin{aligned} \mathbb{M}^{alg}(\mathbb{A}((X, X), \delta)) &= \mathbb{M}^{alg}(X, q(\iota_{((X, X), \delta)})) \\ &= (q(\iota_{((X, X), \delta)}))_!(0_{\mathcal{W}_{X-}}) \\ &= ((X, X), \delta) \end{aligned}$$

The last identity holds because the lifting problem

$$\begin{array}{ccc} & q(q(\iota_{((X, X), \delta)}))_!(0_{\mathcal{W}_{X-}}) & \\ q(\iota_{((X, X), \delta)}) \nearrow & \uparrow id & \\ q(0_{\mathcal{W}_{X-}}) \xrightarrow{q(\iota_{((X, X), \delta)})} & q((X, X), \delta) & \end{array}$$

for $q(\iota_{((X, X), \delta)})$ is solved by

$$OpCart(id, ((X, X), \delta)) : ((X, X), \delta) \rightarrow q(\iota_{((X, X), \delta)}))_!(0_{\mathcal{W}_{X-}})$$

which is the identity because q is split.

2.

$$\begin{aligned} \mathbb{A}(\mathbb{M}^{alg}(X, \alpha : \underline{w}(X) \rightarrow X)) &= \mathbb{A}(\alpha_!(0_{\mathcal{W}_{X-}})) \\ &= (X, q(\iota_{\alpha_!(0_{\mathcal{W}_{X-}})})) \\ &= (X, \alpha : \underline{w}(X) \rightarrow X) \end{aligned}$$

where the last equation holds because $\alpha_!(0_{\mathcal{W}_{X-}}) : \mathcal{W}_{X,X}$ gives us an isomorphism, and q is split, whence this isomorphism is the identity.

3.

$$\begin{aligned} \mathbb{M}^{coalg}(\mathbb{C}((X, X), \delta)) &= \mathbb{M}^{coalg}(X, p(\pi_{((X, X), \delta)})) \\ &= (p(\pi_{((X, X), \delta)}))^*(1_{\mathcal{W}_{X-}}) \\ &= ((X, X), \delta) \end{aligned}$$

where the last equation holds by the dual of the corresponding statement about algebras.

4.

$$\begin{aligned}
\mathbb{C}(\mathbb{M}^{coalg}(X, \alpha : X \rightarrow \underline{m}(X))) &= \mathbb{C}(\alpha^*(1_{\mathcal{W}_{-X}})) \\
&= (X, p(\pi_{\alpha^*(1_{\mathcal{W}_{-X}})})) \\
&= (X, \alpha : X \rightarrow \underline{m}(X))
\end{aligned}$$

where we use for the last equation that $\alpha^*(1_{\mathcal{W}_{-X}}) : \mathcal{W}_{XX}$ provides an isomorphism, and splitness of p implies that this isomorphism is an identity. The equations for the actions on morphisms follow from the universal properties involved.

C Lifting Bialgebras

In the opposite direction of Definition 19, i.e. lifting a bialgebra $\underline{w}(X) \xrightarrow{\alpha} X \xrightarrow{\zeta} \underline{m}(X)$ by lifting α , and ζ separately as above, we obtain two objects that we can relate by lifting the composite $\zeta \circ \alpha$ to two morphisms (denoted l and r). l is obtained by lifting through the opfibred terminal object, and r by lifting through the fibred initial object as shown in the following diagram:

$$\begin{array}{ccc}
\alpha^* \mathbf{0}X & & \\
\downarrow \text{Cart}(\alpha, \mathbf{0}X) & & \downarrow l \\
\mathbf{0}X & \xrightarrow{\iota_X} & \zeta^* \mathbf{1}X \\
\downarrow o & & \downarrow c \\
\alpha_! \mathbf{0}X & \xrightarrow{\pi_X} & \mathbf{1}X \\
\downarrow r & & \downarrow \text{OpCart}(\zeta, \mathbf{1}X) \\
& & \zeta_! \mathbf{1}X
\end{array}$$

where $o = \text{OpCart}(\alpha, \mathbf{0}X)$ and $c = \text{Cart}(\zeta, \mathbf{1}X)$. We have thus: $p(l) = \zeta \circ \alpha$, $q(l) = q(\iota_X)$, and $p(r) = p(\pi_X)$, $q(r) = \zeta \circ \alpha$; in other words, considering “round trips”, both possible lifts lie over the bialgebra with which we started and this bialgebra can be recovered by either applying p or q to the lifts.