

29th International Conference on Types for Proofs and Programs

TYPES 2023, June 12-16, 2023, ETSInf, Universitat Politècnica
de València, Spain

Edited by

Delia Kesner

Eduardo Hermo Reyes

Benno van den Berg



Editors

Delia Kesner 

Université Paris Cité, France
kesner@irif.fr

Eduardo Hermo Reyes 

Formal Vindications, Barcelona, Spain
ehermo.reyes@formalv.com

Benno van den Berg 

University of Amsterdam, The Netherlands
b.vandenberg3@uva.nl

ACM Classification 2012

Theory of computation → Type theory; Theory of computation → Type structures; Computing methodologies → Representation of mathematical objects; Theory of computation → Interactive proof systems; Theory of computation → Logic; Theory of computation → Logic and verification; Theory of computation → Proof theory; Theory of computation → Constructive mathematics; Theory of computation → Linear logic; Theory of computation → Process calculi; Software and its engineering → Formal software verification; Security and privacy → Systems security

ISBN 978-3-95977-332-4

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/978-3-95977-332-4>.

Publication date

August, 2024

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication 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.2023.0

ISBN 978-3-95977-332-4

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

- Luca Aceto (Reykjavik University, IS and Gran Sasso Science Institute, IT)
- 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)
- Daniel Král' (Masaryk University, Brno, CZ)
- Meena Mahajan (*Chair*, Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (Nanyang Technological University, SG)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)
- Pierre Senellart (ENS, Université PSL, Paris, FR)

ISSN 1868-8969

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

■ Contents

Preface	
<i>Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg</i>	0:vii
Regular Papers	
Classification of Covering Spaces and Canonical Change of Basepoint	
<i>Jelle Wemmenhove, Cosmin Manea, and Jim Portegies</i>	1:1–1:23
Finite Combinatory Logic with Predicates	
<i>Andrej Dudenhefner, Christoph Stahl, Constantin Chaumet, Felix Laarmann, and Jakob Rehof</i>	2:1–2:22
Categorical Models of Subtyping	
<i>Greta Coraglia and Jacopo Emmenegger</i>	3:1–3:19
A Sound and Complete Substitution Algorithm for Multimode Type Theory	
<i>Joris Ceulemans, Andreas Nuyts, and Dominique Devriese</i>	4:1–4:23
Consistent Ultrafinitist Logic	
<i>Michal J. Gajda</i>	5:1–5:20
A Reflection Principle for Potential Infinite Models of Type Theory	
<i>Matthias Eberl</i>	6:1–6:20



■ 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 29th International Conference on Types for Proofs and Programs, TYPES 2023, that was held at Universitat Politècnica de València (UPV), from 12 to 15 June 2023.

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). 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 and constructive mathematics; Homotopy type theory; Applications of type theory; 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; Type theory in linguistics.

The TYPES conferences are all based on contributed talks based on short abstracts, reporting work in progress and work presented or published elsewhere. After the conference, a post-proceedings volume is edited, comprising papers presenting original work that has not previously been published. Papers submitted to the post-proceedings are subject to a full peer-review process.

The conference program of TYPES 23 consisted of 61 contributed talks, and four invited talks by Yannick Forster (Inria Nantes, France), Marie Kerjean (CNRS, Université Sorbonne Paris Nord, France), Simona Ronchi Della Rocca (Università di Torino, Italy) and Andrej Bauer (University of Ljubljana, Slovenia). The conference was a successful event with 92 registered participants. All the details of the conference can be found at <https://types2023.webs.upv.es/>.

Concerning the post-proceedings, 10 papers were initially submitted, out of which 6 were accepted. We thank all the authors and reviewers for their hard work to make this possible!

Delia Kesner, Eduardo Hermo Reyes and Benno van den Berg, June 2024.



Classification of Covering Spaces and Canonical Change of Basepoint

Jelle Wemmenhove¹  

Department of Mathematics and Computer Science, Eindhoven University of Technology,
The Netherlands

Cosmin Manea 

Department of Mathematics and Computer Science, Eindhoven University of Technology,
The Netherlands

Jim Portegies 

Department of Mathematics and Computer Science, Eindhoven University of Technology,
The Netherlands

Abstract

Using the language of homotopy type theory (HoTT), we 1) prove a synthetic version of the classification theorem for covering spaces, and 2) explore the existence of canonical change-of-basepoint isomorphisms between homotopy groups. There is some freedom in choosing how to translate concepts from classical algebraic topology into HoTT. The final translations we ended up with are easier to work with than the ones we started with. We discuss some earlier attempts to shed light on this translation process. The proofs are mechanized using the COQ proof assistant and closely follow classical treatments like those by Hatcher [6].

2012 ACM Subject Classification Mathematics of computing → Algebraic topology; Theory of computation → Type theory; Theory of computation → Constructive mathematics

Keywords and phrases Synthetic Homotopy Theory, Homotopy Type Theory, Covering Spaces, Change-of-Basepoint Isomorphism

Digital Object Identifier 10.4230/LIPIcs.TYPES.2023.1

Supplementary Material *Software (Source Code)*: <https://gitlab.tue.nl/computer-verified-proofs/covering-spaces> [18]

archived at `swh:1:dir:bea0c0af55e3ec9869679f3a5611cc5154a9ddb`

1 Introduction

Homotopy type theory (HoTT) is a variant of Martin-Löf type theory (MLTT) that can be used as a synthetic language for developing the theory of algebraic topology, specifically the subfield of homotopy theory. This means that the types in MLTT are given topological interpretations. The main example is the identity type $a =_X b$ which in HoTT is interpreted as the type of paths from a to b in a space X . Since identity types are primitive objects in MLTT, one obtains a low-level encoding of the mathematical theory: definitions are simple and you can quickly go on to prove interesting theorems.

Many classical results from algebraic topology and homotopy theory have already been developed synthetically within HoTT, e.g. computations of homotopy groups of spheres (Brunerie and Licata [2, 13], and Ljungström and Mörtberg [14]), the Blakers-Massey theorem (Hou, Finster, Licata, and Lumsdaine [8]), and Van Kampen's theorem (Hou and Shulman [10]). Not every result, however, can be translated into homotopy type theory directly: to prove a synthetic version of Brouwer's fixed point theorem, for example, requires

¹ Corresponding author



extending the base theory to so-called *real-cohesive* homotopy type theory [15]. In this article, we build on the development of covering spaces in HoTT by Hou and Harper [9], and Buchholtz and Hou’s work on cellular cohomology [3].

We prove a synthetic version of the *classification of covering spaces* and synthetically explore the existence of *canonical change-of-basepoint isomorphisms* between homotopy groups. Although these topics seem quite disparate, the motivation to develop these results in HoTT was unitary: we had little experience in using HoTT as a synthetic language, so to increase our knowledge we set out to prove an exercise from Hatcher’s *Algebraic Topology* [6] (Exercise 3.3.11) in HoTT. The solution to this exercise required the development of the two topics presented here. The results have been mechanized² in the COQ proof assistant using the COQ-HoTT library [1]. We were able to closely mirror the classical arguments used by Hatcher, making extensive use of core HoTT concepts like “transport” and “truncations”. With the aim of serving as an entry point for other newcomers to HoTT, we tried to keep our proofs explicit.

Our low-level approach might suggest that HoTT is merely a formal language for re-expressing existing concepts and proofs from algebraic topology. Yet for experts, HoTT does provide new ways to think about their subjects and allows them to express new concepts in high-level conceptual arguments, such as the development of so-called “higher groups” by Buchholtz, Van Doorn, and Rijke [4].

The synthetic version of the *classification of covering spaces* is shown in Section 3, as well as intermediate results like the *lifting criterion*. We use Hou and Harper’s definition of covering spaces [9] and proof techniques like “extension by weak constancy” [5] to stay close to the classical treatment by Hatcher [6]. In contrast, Buchholtz et al. prove the Galois correspondence of covering spaces from a more abstract perspective using their theory of higher groups [4, Thm. 7]. Finding the “right” translations of classical statement into HoTT was an iterative process: after having proven a specific formulation of some theorem, we would realize that the proof could strongly be simplified if we used a different, yet equivalent, formulation. To shed some light on the translation process, Section 3 also discusses some of our earlier attempts.

In Section 4, we prove conditions for the existence of *canonical change-of-basepoint isomorphisms* between homotopy groups by looking at the triviality of the π_1 -action on these groups. We were interested in the existence of such isomorphisms because we needed them to define the degree of non-pointed maps between spheres. Hou [7] defines the degree of such maps in a different way: on the level of sets, non-pointed maps between spheres are equivalent to pointed ones, and for these it is easy to define a degree. Although this approach seems quite different from ours, we prove a classical result relating basepoint-preserving and free homotopy classes of maps, which illustrates that Hou’s approach also relies on the triviality of the π_1 -actions for spheres.

Let us here also quickly mention a third way to turn non-pointed maps between spheres into pointed ones. As part of recent work, Buchholtz et al.³ constructed a family of sphere reparameterizations that can map any point on the sphere to the basepoint. As such any non-pointed map between spheres can be transformed into a pointed one by post-composition with the appropriate reparameterization.

Before moving on to the main results in Sections 3 and 4, we recall the homotopical interpretation of types and discuss some techniques used throughout this article.

² Code repository: <https://gitlab.tue.nl/computer-verified-proofs/covering-spaces>. The content of this article corresponds to version 0.3.

³ See Meyers’s talk *The tangent bundles of spheres*, available at <https://www.youtube.com/watch?v=9T9B9XBjVpk> (timestamp 9:30 minutes).

2 Background

2.1 Topological interpretation

We recall the basic notions from homotopy type theory necessary for understanding the results in this article. For a full explanation of homotopy type theory, we refer to the HoTT book [17]. It starts with a friendly introduction to type theory and its homotopical interpretation, discusses the differences with MLTT, such as higher inductive types and the univalence axiom, and provides notes on the historical development of these ideas. In this section, we do highlight some nuances that might be missed on an initial reading of the book.

2.1.1 Spaces, points, paths, and higher paths

The key to reading type theoretical statements as topological statements is the *identification-as-paths* interpretation: a type X is interpreted as a *space*, terms $x : X$ are interpreted as *points* in this space, and identity types $a =_X b$ are interpreted as the *type of paths* from point a to b . Viewing the identity type $a =_X b$ as a space in itself, we obtain a type $p =_{a=_X b} q$ of *paths-between-paths* with $p, q : a =_X b$. These paths-between-paths are called *homotopies* in topology. Continuing this construction we get an infinite tower of higher-and-higher paths, endowing every type X with the structure of an ∞ -groupoid.

► **Remark.** Coming from a math background, it is easy to mistake “a type X is interpreted as a space” as meaning that X is a *topological* space. This is incorrect! It is meant that X is a space precisely in the sense that it has an ∞ -groupoid structure. ∞ -groupoids turn out to be the right data structure for describing the (higher order) path-structure of genuine topological spaces: from a topological space X' one can construct the fundamental ∞ -groupoid $\Pi_\infty(X')$ and this ∞ -groupoid is enough to prove many theorems in algebraic topology, specifically the theorems belonging to the subfield of homotopy theory. Words like *path* and *homotopy* are reused for elements of ∞ -groupoids both to prevent the need for a new vocabulary and to more easily tap into our spatial intuition. For an excellent discussion on the differences between ∞ -groupoids and topological spaces, see the introduction of Shulman’s work [15].

Even though identity types $a =_X b$ are interpreted as paths, they are also still thought of as equalities. In calculations, it feels more natural to read a chain like $a = \dots = z$ as a list of equalities than as a composition of paths. The usual rules for equalities also still apply: for example, an equality $p : a =_X b$ implies an equality $\mathbf{ap}_f(p) : f(a) =_Y f(b)$, where $f : X \rightarrow Y$. In HoTT, this statement is interpreted as saying that all functions are *continuous*, in the sense that they preserve paths between points.

2.1.2 Type families, transport, and dependent paths

A type family $P : X \rightarrow \mathbf{Type}$ is interpreted as a collection of spaces that “lie over” a *base space* X . the space $P(x)$ is also referred to as the *fiber* over $x : X$. The sigma type $\sum_{x:X} P(x)$ is called the *total space* of P and the first projection $\mathbf{pr}_1 : (\sum_{x:X} P(x)) \rightarrow X$ is a special map called a *fibration*. These maps play an important role in classical homotopy theory, but in HoTT it is easier to work with type families directly.

Given a path $p : a =_X b$, points $u : P(a)$ in the fiber over a can be “transported” along p to the fiber $P(b)$, the result is denoted as $\mathbf{transport}^P(p, u) : P(b)$. Transporting along paths behaves as expected: transport along a constant path $\mathbf{refl}_x : x =_X x$ leaves the points in $P(x)$ unchanged, $\mathbf{transport}^P(\mathbf{refl}_x, u) \equiv u$, transport along a composite path $p \cdot q : a =_X b =_X c$ is the same as first doing transport along p and then along q ,

$$\mathbf{transport}^P(p \cdot q, u) =_{P(c)} \mathbf{transport}^P(q, \mathbf{transport}^P(p, u)) ,$$

and transport along the reverse path $p^{-1} : b =_X a$ is the inverse of transport along p . Chapter 2 of the HoTT book [17] contains a number of useful characterizations of transport in different kinds of type families. Of special interest to us are the formulas for transport in loop spaces, e.g. transport of a loop $q : a =_X a$ along $p : a =_X b$ equals conjugation with p ,

$$\text{transport}^{x \mapsto x =_X x}(p, q) =_{(b =_X b)} p^{-1} \cdot q \cdot p . \quad (1)$$

Transport also provides a convenient way to reason about paths in the total space. In general, a path $\tilde{p} : (a, u) = (b, v)$ in the total space is equivalent to the combination of a path $p : a =_X b$ in the base space and a path $w_p : (\text{transport}^P(p, u) =_{P(b)} v)$ in the fiber $P(b)$, see [17, Thm. 2.7.2]. A path of the form $(\text{transport}^P(p, u) =_{P(b)} v)$ is called a *dependent* path; it is interpreted as a path from u to v that lies *over* the path p . For example, the dependent path (1) is interpreted as a path (i.e. a homotopy) from loop $q : a =_X a$ to loop $p^{-1} \cdot q \cdot p : b =_X b$ that lies over p . The interpretation of dependent paths as lying over paths in the base space is justified by the equivalence with paths in the total space. In fact, to specify the paths that lie over some path p , it is easier to use dependent paths than paths in the total space. This is because the equality $\text{ap}_{\text{pr}_1}(\tilde{p}) = p$ which picks out the paths $\tilde{p} : (a, u) = (b, v)$ that lie over p is propositional and not judgemental.

Finally, we have the following result which says that transport commutes with operations on the fibers of the type family.

► **Lemma 1.** *Let $P, Q : X \rightarrow \text{Type}$ be two type families and let $f_{(-)} : \prod_x (P(x) \rightarrow Q(x))$ be a family of maps, then for all paths $p : a =_X b$ and points $u : P(a)$ it holds that*

$$\text{transport}^Q(p, f_a(u)) = f_b(\text{transport}^P(p, u)) .$$

Proof. Let $p : a =_X b$, then there exists a dependent equality between f_a and f_b over p ,

$$\text{ap}_{f_{(-)}}(p) : \text{transport}^{x \mapsto (P(x) \rightarrow Q(x))}(p, f_a) = f_b ,$$

and by Lemma 2.9.6 in the HoTT book [17], this is equivalent to what we need to show. ◀

2.1.3 Truncations and path-connectedness

In classical homotopy theory, the infinite structure of ∞ -groupoids can be “truncated” to make them easier to study. Such truncation operators are also available in HoTT, but here their main use is as modalities that increase the logical expressiveness of the theory.

For example, propositional truncation is needed to accurately capture the concept of path-connected spaces in homotopy type theory. A space X is called path-connected (or just *connected* in HoTT, as there is no analogy to the topological notion of connectedness) if for every two points $a, b : X$ there *merely* exists a path between them; this is expressed as the (propositionally) truncated type $\|a =_X b\|$ being inhabited. An *explicit* witness $p : a =_X b$ would imply having a canonical choice of path, a form of constructive existence which is stronger than mere existence.

Some types already have a truncated higher-order ∞ -groupoid structure of their own, truncating them again has no effect. A type Z for which $\|Z\|_n \simeq Z$ is called an n -type. In this article we will mainly encounter (-1) -types and 0 -types. These are called *propositions* and *sets* respectively, the types of propositions and sets are denoted by Prop and Set . Two terms of a proposition are always equal, e.g. for $p, q : \|a =_X b\|$ it holds that $p = q$, and sets are homotopy equivalent to a collection of points.

2.1.4 Loop spaces and homotopy groups

We briefly recall the definitions of loop spaces, homotopy groups, and induced maps on these.

The term refl_x^n denotes the constant n -dimensional path (also called an n -cell) from $x : X$ to itself. It is defined recursively by

$$\text{refl}_x^{n+1} := \text{refl}_{\text{refl}_x^n} : (\text{refl}_x^n = \text{refl}_x^n) \quad \text{with} \quad \text{refl}_x^0 := x ,$$

so $\text{refl}_x^1 \equiv \text{refl}_x : (x =_X x)$ denoted the constant path and $\text{refl}_x^2 : (\text{refl}_x =_X \text{refl}_x)$ denotes the constant homotopy between the constant path refl_x and itself.

Given a pointed type (X, x_0) , the space $\Omega^n(X, x_0) := (\text{refl}_{x_0}^{n-1} = \text{refl}_{x_0}^{n-1})$ of n -cells is called the n -th *loop space* of (X, x_0) . It is a pointed type in itself, with the constant n -cell $\text{refl}_{x_0}^n : \Omega^n(X, x_0)$ as the designated point. Thus, we have $\Omega^{n+1}(X, x_0) := \Omega(\Omega^n(X, x_0))$. The *homotopy groups* $\pi_n(X, x_0)$ are defined as the *sets* of n -dimensional loops, meaning that $\pi_n(X, x_0) := \|\Omega^n(X, x_0)\|_0$.

Given a pointed map $f : (X, x_0) \rightarrow (Y, y_0)$ with $w_f : f(x_0) = y_0$ as its proof of pointedness (which alternatively can also be denoted as a pair $(f, w_f) : (X, x_0) \rightarrow (Y, y_0)$), there are induced maps $f_* : \Omega(X, x_0) \rightarrow \Omega(Y, y_0)$ and $f_* : \pi_1(X, x_0) \rightarrow \pi_1(Y, y_0)$ given by

$$f_*(p) := w_f^{-1} \cdot \text{ap}_f(p) \cdot w_f \quad \text{and} \quad f_*(|p|_0) := |w_f^{-1} \cdot \text{ap}_f(p) \cdot w_f|_0 \quad \text{with} \quad p : (x_0 =_X x_0) .$$

Whichever version of f_* is meant should be clear from context. Note that it suffices to define $f_* : \pi_1(X, x_0) \rightarrow \pi_1(Y, y_0)$ for terms of the form $|p|_0$ since the codomain is a set, see the next section.

2.2 Dealing with truncations

The topological results in this article are about connected spaces, so we are constantly confronted with the fact that we only have truncated paths $\|a =_X b\|$. Although they come from non-truncated paths, we cannot freely use them as such.

The basic way to deal with truncated types is by using their induction principle: if the goal is to prove an n -type, we may “strip” the truncation-bars from a truncated type $\|Z\|_n$ and use it as if it were the non-truncated type. In practice, this means that an n -truncated term $z : \|Z\|_n$ can be assumed to be of the form $z \equiv |z'|_n$ with $z' : Z$. So, if W is an n -type, we can define maps $\|Z\|_n \rightarrow W$ by only specifying the output on terms of the form $|z|_n$.

The absence of canonical paths is also an issue in the classical algebraic topology when doing constructions. There, the solution is to take an arbitrary path for the construction and to then show that final result does not depend of the specific path chosen. We can do something similar in HoTT, with the caveat that the constructed object’s type has to be a *set*. The magic ingredient is a technique called *extension by weak constancy*.

► **Lemma 2** (Extension by weak constancy, cf. generalization [5, Thm. 1]). *Let Z be a type and W a set. If $f : Z \rightarrow W$ satisfies $f(z_1) =_W f(z_2)$ for all $z_1, z_2 : Z$, it can be extended to a map $g : \|Z\| \rightarrow W$ such that $g(|z|) \equiv f(z)$ for all $z : Z$.*

Extension by weak constancy is used as follows to construct objects from truncated paths. First, give the construction for an arbitrary, non-truncated path $p : a =_X b$, i.e. define a map $w : (a =_X b) \rightarrow W$, where W denotes the type of object to construct. Provided that (i) W is a set, and (ii) $w(p) = w(q)$ for all $p, q : a =_X b$, the construction w can be extended to a map $\bar{w} : \|a =_X b\| \rightarrow W$. The inhabitant of the truncated path type, $*$: $\|a =_X b\|$, is

inserted to obtain the final object, $w^* := \overline{w}(*): W$. Additionally, it holds that $w^* = w(p)$ for any explicit path $p: a =_X b$. This is because the equality is a proposition, so the truncation from $*$ can be stripped, from which it follows that

$$w^* \equiv \overline{w}(*)\equiv \overline{w}(|q|)\equiv w(q) = w(p) ,$$

where $* \equiv |q|$ for some $q: a =_X b$.

2.3 Notation

We stick to the notation of the HoTT book, with some exceptions: we use `Type` instead of \mathcal{U} to denote the universe of types and we use f_* to denote the induced maps on loop spaces and fundamental groups instead of $\Omega(f)$ and $\pi_1(f)$ in order to stay closer to the notation used by Hatcher [6]. As not to confuse f_* with the shorthand notation for transport, like $p_*u = v$, we prefer to write out transport in full, namely as $(\text{transport}^P(p, u) = v)$.

3 Classification of covering spaces

In this section we prove synthetic versions of the *lifting criterion* and the *classification of covering spaces*. With the right translations into HoTT, we were able to obtain low-level proofs that closely follow their classical counterparts, e.g. those used by Hatcher [6]. Figuring out what the “right” translations were, however, took multiple attempts. Besides giving the final versions of definitions and statements, we also discuss some earlier versions and what trouble they caused us, so that others may learn from our experience.

► **Note.** There is an official notion of “correctness” for translation of classical statements into HoTT. Homotopy type theory can be modeled in the topos of simplicial sets [12], and simplicial sets have a geometric realization as CW complexes, i.e. actual topological spaces. Statements in HoTT can thus be interpreted as statements about CW complexes, and so it can be checked that the translation of statements into HoTT is equivalent to the original statements under this interpretation. Checking the correctness of translations in this way is not the purpose of this article. We rely on the homotopical intuition build up over time by the community and the *a posteriori* justification provided by proofs of classical results.

3.1 Covering spaces in HoTT

The study of covering spaces in HoTT was initiated by Hou and Harper [9]. They prove that every covering space of a pointed space (X, x_0) corresponds to a set with a $\pi_1(X, x_0)$ -action and they construct the universal covering space. We take their definition of a pointed covering space as a starting point:

► **Definition 3** (cf. [9, Def. 1&7]). *Let (X, x_0) be a pointed type. A **covering space** of X is a set-valued type family $F: X \rightarrow \text{Set}$. If F is equipped with a designated point $u_0: F(x_0)$, the pair (F, u_0) is called a **pointed covering space**.*

Recall that type families naturally correspond to fibrations in HoTT. The requirement that F takes values in `Set` guarantees the “sheetedness” of the covering space as it implies that each fiber is homotopic to a discrete collection of points.

Since types automatically inherit a notion of equality, we can also check a definition's correctness by inspecting its identity type.⁴ The classical classification theorem only distinguishes covering spaces up to fiberwise, base-point preserving homeomorphism, and, indeed, the lemma below shows that Hou and Harper's definition gives rise to the same notion of equality. (Note that the notion of homeomorphism can only be expressed as homotopy equivalence in HoTT.)

► **Lemma 4** (Characterization of equality between pointed covering spaces). *Let (F_1, u_1) and (F_2, u_2) be pointed covering spaces over a pointed type (X, x_0) . Then there is an equivalence*

$$(F_1, u_1) = (F_2, u_2) \quad \simeq \quad \sum_{h: \prod_x F_1(x) \simeq F_2(x)} h(x_0, u_1) =_{F_2(x_0)} u_2 .$$

This implies that two pointed covering spaces are equal if and only if there exists a basepoint-preserving, fiberwise equivalence between them.

Proof. A term $w : (F_1, u_1) = (F_2, u_2)$ is a path in the total space $\sum_X F$, so it is equivalent to a path between functions $h' : F_1 = F_2$ that satisfies

$$\text{transport}^{F \rightarrow F(x_0)}(h', u_1) = u_2 .$$

By functional extensionality, h' corresponds to the family $\text{happly}(h') : \prod_x F_1(x) = F_2(x)$. Each $\text{happly}(h', x) : F_1(x) = F_2(x)$ is a path between types (technically, between types with a proof that they are sets, but since being a set is a proposition in itself, this can be ignored), so by univalence $\text{happly}(h', x)$ corresponds to an equivalence

$$\text{idtoeqv}(\text{happly}(h', x)) : F_1(x) \simeq F_2(x) .$$

Thus h' corresponds 1-to-1 to the fiberwise equivalence $h : \prod_{x:X} F_1(x) \simeq F_2(x)$ given by $h(x, -) \equiv \text{idtoeqv}(\text{happly}(h', x))$. By path induction, it holds that

$$h(x, -) \equiv \text{idtoeqv}(\text{happly}(h', x)) = \text{transport}^{F \rightarrow F(x)}(h', -) ,$$

so the condition $(\text{transport}^{F \rightarrow F(x_0)}(h', u_1) = u_2)$ satisfied by h' is equivalent to the statement $h(x_0, u_1) = u_2$, giving us an equivalence on the level of sigma types. ◀

3.2 Lifting criterion

The lifting criterion serves as a nice stepping stone towards the classification of covering spaces. Not only is it used in the classical proof of the classification theorem, it also allows us to practice with translating classical statements into homotopy type theory. First, we obtain a direct translation of the lifting criterion, but it turns out that this translation can be simplified whilst more closely reflecting the geometric ideas.

We define the lift of a map in homotopy type theory as follows.

► **Definition 5.** *Let $f : (Y, y_0) \rightarrow (X, x_0)$ be a pointed map with $w_f : f(y_0) =_X x_0$ its proof of pointedness. A **pointed lift** of f to the covering space (F, u_0) over (X, x_0) is a dependent map $\tilde{f} : \prod_{y:Y} F(f(y))$ such that*

$$\text{transport}^F(w_f, \tilde{f}(y_0)) =_{F(x_0)} u_0 .$$

⁴ This method of validating definitions in HoTT by checking if they produce the right type of equality was suggested by Egbert Rijke at the 2023 HoTT/UF workshop.

Instead of defining the lift as a dependent map $\tilde{f} : \prod_{y:Y} F(f(y))$ – meaning that \tilde{f} maps each point $y : Y$ to a point in the fiber over $f(y)$ – we could have stayed closer to the classical definition and defined the lift as a map $\tilde{f} : Y \rightarrow \sum_X F$ such that $w_{\tilde{f}} : \text{pr}_1 \circ \tilde{f} = f$. These formulations are equivalent and both types deserve to be called lifts, but we prefer to use the dependent map formulation because, in our experience, it was easier to work with. We tried to use the classical formulation in an earlier attempt, but this required us to use transport along $w_{\tilde{f}}$ every time we had to compare terms in the fibers over $\text{pr}_1(\tilde{f}(y))$ and $f(y)$. This does not happen when defining the lift as a dependent map, because then the equality $\text{pr}_1 \circ \tilde{f} \equiv f$ is a judgemental one.

Using Definition 5, the lifting criterion can be formulated directly in homotopy type theory.

► **Theorem 6** (direct translation, cf. [6, Prop.1.33]). *Let (F, u_0) be a pointed covering space over a pointed type (X, x_0) . A pointed map $f : (Y, y_0) \rightarrow (X, x_0)$, with Y a connected type, can be lifted to pointed lift $\tilde{f} : \prod_{y:Y} F(f(y))$ if and only if*

$$f_*(\pi_1(Y, y_0)) \subset \text{pr}_{1*}(\pi_1(\sum_X F, (x_0, u_0))) . \quad (2)$$

In proving Lemma 6, we found criterion (2) inconvenient to work with. The notation $f_*(\pi_1(Y, y_0))$ conceals multiple truncations – a propositional-truncation to define the image of a map and a set-truncation for π_1 – which hinder access to the paths themselves. Furthermore, Hou and Harper’s definition of covering spaces does not involve the (pointed) total space $(\sum_X F, (x_0, u_0))$, so we would prefer to have a version of criterion (2) that does not use it either.

Upon closer inspection, criterion (2) can be expressed more succinctly as a statement involving dependent paths. The criterion basically says that for any loop in type $x_0 =_X x_0$ of the form $f_*(p)$ with $p : y_0 =_Y y_0$, there exists a loop of type $(x_0, u_0) =_{(x_0, u_0)} (x_0, u_0)$ in the total space lying over $f_*(p)$. In HoTT, the latter is interpreted as the existence of a dependent loop

$$\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0 .$$

This gives us an equivalent alternative to criterion (2) as stated in the lemma below. Note that the set-truncated π_1 ’s are also replaced by untruncated path types. This is possible because both statements are propositions.

► **Lemma 7.** *Let (F, u_0) be a pointed covering space over a pointed type (X, x_0) and let $f : (Y, y_0) \rightarrow (X, x_0)$ be a pointed map. The following propositions are equivalent:*

- (i) $f_*(\pi_1(Y, y_0)) \subset \text{pr}_{1*}(\pi_1(\sum_X F, (x_0, u_0)))$;
- (ii) for all loops $p : y_0 =_Y y_0$ there exists a dependent loop of type $(\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0)$.

The proof is a bit technical and involves a lot of truncations. We deal with these now so we do not have to deal with them when proving the lifting criterion.

Proof. Since both conditions are propositions, it suffices to show that both statements imply each other.

Assume that $f_*(\pi_1(Y, y_0)) \subset \text{pr}_{1*}(\pi_1(\sum_X F, (x_0, u_0)))$ and let $p : y_0 =_Y y_0$. Naturally, the truncated loop $f_*(|p|_0)$ lies in $f_*(\pi_1(Y, y_0))$, so by assumption it also lies in $\text{pr}_{1*}(\pi_1(\sum_X F, (x_0, u_0)))$, meaning that there *merely* exists a $q' : \pi_1(\sum_X F, (x_0, u_0))$ such that $\text{pr}_{1*}(q') = f_*(|p|_0)$. The goal is to show that $(\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0)$, and

since $F(x_0)$ is a set, this is a proposition. Hence, we can strip the “merely” in “merely exists” and we can strip the truncation from $\pi_1(\sum_X F, (x_0, u_0)) \equiv \|(x_0, u_0) = (x_0, u_0)\|_0$ to obtain an explicit loop $q : (x_0, u_0) = (x_0, u_0)$ which satisfies $\text{pr}_{1*}(|q|_0) = f_*(|p|_0)$. As q is a loop in the total space, it can be split into its projection onto the base space $\text{ap}_{\text{pr}_1}(q) : x_0 =_X x_0$ and a dependent loop of type $(\text{transport}^F(\text{ap}_{\text{pr}_1}(q), u_0) =_{F(x_0)} u_0)$ in the fiber $F(x_0)$. Therefore, to show that $(\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0)$, it suffices to show that

$$\text{ap}_{\text{pr}_1}(q) = f_*(p) .$$

This follows from the fact that q satisfies $\text{pr}_{1*}(|q|_0) = f_*(|p|_0)$: using the definitions of the induced maps, this equality can be rewritten as $|\text{pr}_{1*}(q)|_0 = |f_*(p)|_0$, which by Theorem 7.3.12 in [17], is equivalent to $\|\text{pr}_{1*}(q) = f_*(p)\|$. The truncation can be stripped, and the result implies that

$$\text{ap}_{\text{pr}_1}(q) = \text{refl}_{x_0}^{-1} \cdot \text{ap}_{\text{pr}_1}(q) \cdot \text{refl}_{x_0} \equiv \text{pr}_{1*}(q) = f_*(p) .$$

Conversely, assume that for all loops $p : y_0 =_Y y_0$ there exists a dependent path of the form $(\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0)$. Let $r : \pi_1(X, x_0)$ and assume that r lies in the image $f_*(\pi_1(Y, y_0))$, meaning that there merely exists a $p' : \pi_1(Y, y_0)$ such that $f_*(p') = r$. The goal is to show the proposition that r lies in $\text{pr}_{1*}(\pi_1(\sum_X F, (x_0, u_0)))$, so we can again strip the truncations in “the mere existence of p' ” to obtain an explicit loop $p : y_0 =_Y y_0$ that satisfies $f_*(|p|_0) = r$. To show that r lies in $\text{pr}_{1*}(\pi_1(\sum_X F, (x_0, u_0)))$, it suffices to construct a loop $q : (x_0, u_0) = (x_0, u_0)$ that satisfies $\text{pr}_{1*}(|q|_0) = r$. Since $r = f_*(|p|_0)$, this latter property can be replaced by $\text{pr}_{1*}(|q|_0) = f_*(|p|_0)$. From here we can ignore the loop r , making this part of the proof more similar to the previous part. To construct a loop $q : (x_0, u_0) = (x_0, u_0)$ it suffices to provide a loop $q_1 : x_0 =_X x_0$ in the base space and a dependent loop of type $(\text{transport}^F(q_1, u_0) =_{F(x_0)} u_0)$ over q_1 . Choose $q_1 \equiv f_*(p)$, then the dependent loop of type $(\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0)$ exists by assumption. Furthermore, by construction it holds that $\text{ap}_{\text{pr}_1}(q) \equiv q_1 \equiv f_*(p)$, so it holds that

$$\text{pr}_{1*}(|q|_0) \equiv |\text{refl}_{x_0}^{-1} \cdot \text{ap}_{\text{pr}_1}(q) \cdot \text{refl}_{x_0}|_0 = |\text{ap}_{\text{pr}_1}(q)|_0 = |f_*(p)|_0 \equiv f_*(|p|_0) . \quad \blacktriangleleft$$

Replacing criterion (2) in the direct translation of the lifting criterion (Theorem 6) with proposition (ii) from Lemma 7, we obtain a version of the lifting criterion that is better suited to the language of homotopy type theory. We prove this version and, hence, also the direct translation. The proof closely follows the argument used by Hatcher [6].

► **Theorem 8** (Lifting criterion, cf. [6, Prop. 1.33]). *Let (F, u_0) be a pointed covering space over a pointed type (X, x_0) . A pointed map $f : (Y, y_0) \rightarrow (X, x_0)$, with Y a connected type, can be lifted to a pointed lift $\tilde{f} : \prod_{y:Y} F(f(y_0))$ if and only if for all loops $p : y_0 =_Y y_0$ there exists dependent loop*

$$\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0 ,$$

meaning that there exists a loop from u_0 to u_0 over $f_*(p)$ in F .

Proof. Before we start, note that the point $x_0 : X$ is arbitrary, and hence we can perform path induction on the path $w_f : f(y_0) =_X x_0$ that encodes f 's pointedness. So assume that we have a judgmental equality $x_0 \equiv f(y_0)$. This lets us replace every occurrence of x_0 in the theorem's statement by $f(y_0)$. Moreover, we also get to assume that the path w_f itself is judgmentally equal to the constant path, i.e. $w_f \equiv \text{refl}_{f(y_0)}$.

1:10 Classification of Covering Spaces and Canonical Change of Basepoint

Assume that there exists a dependent loop of type $(\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0)$ for every loop $p : y_0 =_Y y_0$. Let $y : Y$. Since Y is connected, the mere path type $\|y_0 =_Y y\|$ is inhabited, which we use to define the lift $\tilde{f}(y)$ like Hatcher: first, we take an arbitrary path $q : y_0 =_Y y$ to construct $\tilde{f}(y)$ by transporting the point $u_0 : F(x_0)$ to the fiber $F(f(y))$ along the path $\text{ap}_f(q) : f(y_0) = f(y)$; we then show that $\tilde{f}(y)$ is well-defined, i.e. that the resulting construction did not depend on the specific choice of path $q : y_0 =_Y y$. This way of constructing $\tilde{f}(y)$ is justified because of extension by weak constancy (Lemma 2) as explained in Section 2.2.

Let $q : y_0 =_Y y$ be an arbitrary path; this gives a path $\text{ap}_f(q) : f(y_0) =_X f(y)$. We then define the lift $\tilde{f}(y)$ as

$$\tilde{f}(y) := \text{transport}^F(\text{ap}_f(q), u_0) : F(f(y)) .$$

It remains to show that this construction does not depend on the choice of path of type $y_0 =_Y y$. Let $q_1, q_2 : y_0 =_Y y$, we need to show that

$$\text{transport}^F(\text{ap}_f(q_1), u_0) = \text{transport}^F(\text{ap}_f(q_2), u_0) .$$

We again follow Hatcher's proof. Remark that $\text{ap}_f(q_1) \cdot \text{ap}_f(q_2)^{-1}$ is a loop in the type $f(y_0) =_X f(y_0)$ of the form $f_*(p)$ for some $p : y_0 =_Y y_0$, namely

$$\text{ap}_f(q_1) \cdot \text{ap}_f(q_2)^{-1} = \text{ap}_f(q_1 \cdot q_2^{-1}) = \text{refl}_{f(y_0)}^{-1} \cdot \text{ap}_f(q_1 \cdot q_2^{-1}) \cdot \text{refl}_{f(y_0)} \equiv f_*(q_1 \cdot q_2^{-1}) .$$

Hence, by assumption we have a dependent loop over $\text{ap}_f(q_1) \cdot \text{ap}_f(q_2)^{-1}$,

$$\text{transport}^F(\text{ap}_f(q_1) \cdot \text{ap}_f(q_2)^{-1}, u_0) =_{F(x_0)} u_0 .$$

Transporting both sides back along $\text{ap}_f(q_2)$, we see that $\text{ap}_f(q_1)$ and $\text{ap}_f(q_2)$ both send u_0 to the same point in $F(f(y))$:

$$\text{transport}^F(\text{ap}_f(q_1), u_0) = \text{transport}^F(\text{ap}_f(q_2), u_0) .$$

This concludes the construction of the lift $\tilde{f}(y)$.

Since $w_f \equiv \text{refl}_{f(y_0)}$, proving that \tilde{f} is a pointed lift reduces to showing that $\tilde{f}(y_0) = u_0$. To compute $\tilde{f}(y_0)$, we utilize that we have access to an explicit loop $\text{refl}_{y_0} : y_0 =_Y y_0$. This gives us that

$$\tilde{f}(y_0) = \text{transport}^F(\text{ap}_f(\text{refl}_{y_0}), u_0) \equiv \text{transport}^F(\text{refl}_{f(y_0)}, u_0) \equiv u_0 .$$

Conversely, assume that a pointed lift \tilde{f} exists. Let $p : y_0 =_Y y_0$, then we need to show that there exists a dependent loop of type $(\text{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0)$. The path p gives rise to a dependent equality between $\tilde{f}(y_0)$ and itself over p in the family $F \circ f$, namely

$$\text{apd}_{\tilde{f}}(p) : \text{transport}^{F \circ f}(p, \tilde{f}(y_0)) =_{F(f(y_0))} \tilde{f}(y_0) .$$

By Lemma 2.3.10 in the HoTT book [17], transport along p in $F \circ f$ equals transport along $\text{ap}_f(p)$ in F , so we get a dependent loop at $\tilde{f}(y_0)$ in F ,

$$\text{transport}^F(\text{ap}_f(p), \tilde{f}(y_0)) =_{F(f(y_0))} \text{transport}^{F \circ f}(p, \tilde{f}(y_0)) =_{F(f(y_0))} \tilde{f}(y_0) .$$

Since \tilde{f} is pointed, i.e. $\tilde{f}(y_0) = u_0$, this gives the dependent loop over $f_*(p)$ that was sought:

$$\begin{aligned} \text{transport}^F(f_*(p), u_0) &\equiv \text{transport}^F(\text{refl}_{f(y_0)}^{-1} \cdot \text{ap}_f(p) \cdot \text{refl}_{f(y_0)}, u_0) \\ &= \text{transport}^F(\text{ap}_f(p), u_0) = \text{transport}^F(\text{ap}_f(p), \tilde{f}(y_0)) = \tilde{f}(y_0) = u_0 . \end{aligned} \quad \blacktriangleleft$$

3.3 Classification theorem

After translating the lifting criterion, we are ready to state and prove the classification of covering spaces in homotopy type theory. The theorem is given below. Note how the subgroup associated to a covering space is not defined as the image of the covering map, but is given in terms of dependent loops, like with the reformulated lifting criterion in Theorem 8.

► **Theorem 9** (Classification, cf. [6, first half of Thm. 1.38]). *Let (X, x_0) be a pointed type, then there exists an equivalence between pointed, connected covering spaces (F, u_0) over (X, x_0) and subgroups of $\pi_1(X, x_0)$, obtained by associating to (F, u_0) the subgroup $H_{(F, u_0)}$ given by*

$$H_{(F, u_0)}(|p|_0) := (\text{transport}^F(p, u_0) =_{F(x_0)} u_0) ,$$

meaning that $|p|_0 : \pi_1(X, x_0)$ belongs to $H_{(F, u_0)} : \pi_1(X, x_0) \rightarrow \mathbf{Prop}$ if there exists a loop from u_0 to u_0 lying over p in F .

The theorem classifies *connected* covering spaces. Again, we follow Hou and Harper [9]’s definition:

► **Definition 10.** *A covering space $F : X \rightarrow \mathbf{Set}$ is called **connected** if its total space $\sum_X F$ is connected.*

Both the HoTT proof and the classical proof consists of two parts: showing that the association in Theorem 9 is injective and surjective. Surjectivity is shown using the universal covering space P constructed by Hou and Harper [9]. To show injectivity, the classical proof uses the lifting criterion to construct maps between covering spaces \widetilde{X}_1 and \widetilde{X}_2 by lifting the respective projections:

$$\begin{array}{ccccc} \widetilde{X}_1 & \overset{\widetilde{p}_1}{\dashrightarrow} & \widetilde{X}_2 & \overset{\widetilde{p}_2}{\dashrightarrow} & \widetilde{X}_1 \\ & \searrow p_1 & \downarrow p_2 & \swarrow p_1 & \\ & & X & & \end{array}$$

Unfortunately, we cannot use the lifting criterion for the same purpose in the HoTT-setting, as the codomains of the maps we wish to construct are no longer spaces like \widetilde{X}_i , but families $F_i : X \rightarrow \mathbf{Set}$. Therefore, we prove two new lemmas that give conditions for the existence and uniqueness of maps between covering spaces defined as families of sets. They correspond to the lifting criterion and the unique lifting property in the classical theory.

► **Lemma 11.** *Let (F_1, u_1) and (F_2, u_2) be pointed covering spaces over a pointed type (X, x_0) with F_1 connected. Then there exists a fiberwise map $h : \prod_{x:X} F_1(x) \rightarrow F_2(x)$ that preserves the basepoint, meaning that $h(x_0, u_1) = u_2$, if and only if for all loops $p : x_0 =_X x_0$ we have*

$$(\text{transport}^{F_1}(p, u_1) =_{F_1(x_0)} u_1) \longrightarrow (\text{transport}^{F_2}(p, u_2) =_{F_2(x_0)} u_2) ,$$

meaning that existence of a loop from u_1 to u_1 over p in F_1 implies existence of a loop from u_2 to u_2 over p in F_2 .

► **Lemma 12.** *Let (F_1, u_1) and (F_2, u_2) be pointed covering spaces over a pointed type (X, x_0) with F_1 connected. Then any two fiberwise maps $h_1, h_2 : \prod_{x:X} F_1(x) \rightarrow F_2(x)$ are equal if they coincide in a single point, e.g. if $h_1(x_0, u_1) = h_2(x_0, u_1)$.*

Since Lemma 11 serves as a “family of sets”-based version of the lifting criterion, its proof is also similar to that of Theorem 8.

1:12 Classification of Covering Spaces and Canonical Change of Basepoint

Proof (Lemma 11). Assume that for all for all loops $p : x_0 =_X x_0$, existence of a dependent loop of type $(\text{transport}^{F_1}(p, u_1) =_{F_1(x_0)} u_1)$ in F_1 implies existence of a dependent loop of type $(\text{transport}^{F_2}(p, u_2) =_{F_2(x_0)} u_2)$ in F_2 . Let $x : X$ and $u : F_1(x)$. We construct $h(x, u)$ in the same way as the lift in the proof of the lifting criterion (Theorem 8), namely as the transport of the designated point $u_2 : F_2(x_0)$ along a path to the fiber $F_2(x)$. The path which u_2 is transported along is given in terms of a path in the total space of F_1 . Since F_1 is connected, we only have the mere existence of such paths, so we need to show that the construction of $h(x, u)$ is path-independent. Like in the lifting criterion, we again use extension by weak constancy (Lemma 2) to replicate this construction method in HoTT.

Take an arbitrary path of type $(x_0, u_1) = (x, u)$ in the total space of F_1 . This is equivalent to an arbitrary path $q : x_0 =_X x$ such that $(\text{transport}^{F_1}(q, u_1) =_{F_1(x)} u)$. Since transporting u_1 along q yields u , it seems reasonable that transport of u_2 along q should give $h(x, u)$. So, define $h(x, u)$ as:

$$h(x, u) := \text{transport}^{F_2}(q, u_2) : F_2(x) .$$

It remains to show that any two paths in $(x_0, u_1) = (x, u)$ produce the same point $h(x, u)$. It suffices to show that any two paths $q_1, q_2 : x_0 =_X x$ satisfying $(\text{transport}^{F_1}(q_i, u_1) =_{F_1(x)} u)$ yield the same value for $h(x, u)$. Let q_1, q_2 be such paths. Like in the proof of the lifting criterion (Theorem 8), $(q_1 \cdot q_2^{-1})$ is a loop of type $x_0 =_X x_0$ and it holds that

$$\text{transport}^{F_1}(q_1 \cdot q_2^{-1}, u_1) =_{F_1(x_0)} u_1 , \quad \text{and so} \quad \text{transport}^{F_2}(q_1 \cdot q_2^{-1}, u_2) =_{F_2(x_0)} u_2 .$$

It follows that $(\text{transport}^{F_2}(q_1, u_2) =_{F_2(x_0)} \text{transport}^{F_2}(q_2, u_2))$, so $h(x, u)$ is well-defined.

To prove pointedness, we utilize that we have access to an explicit loop $\text{refl}_{x_0} : x_0 =_X x_0$ which satisfies $(\text{transport}^{F_1}(\text{refl}_{x_0}, u_1) \equiv u_1)$. This allows us to compute $h(x_0, u_1)$:

$$h(x_0, u_1) = \text{transport}^{F_2}(\text{refl}_{x_0}, u_2) \equiv u_2 .$$

Conversely, let $h : \prod_{x:X} F_1(x) \rightarrow F_2(x)$ be a basepoint-preserving, fiberwise map between covering spaces, let $p : x_0 =_X x_0$ be a loop and assume there exists a dependent loop of type $(\text{transport}^{F_1}(p, u_1) =_{F_1(x_0)} u_1)$. By Lemma 1, the family $h(x, -)$ commutes with transport, which implies the existence of a dependent loop at $h(x_0, u_1)$ over p in F_2 , namely

$$\text{transport}^{F_2}(p, h(x_0, u_1)) = h(x_0, \text{transport}^{F_1}(p, u_1)) = h(x_0, u_1) .$$

Since $h(x_0, u_1) = u_2$, we thus have a dependent loop of type $(\text{transport}^{F_2}(p, u_2) =_{F_2(x_0)} u_2)$, which is what we needed. \blacktriangleleft

Proof (Lemma 12). Let $h_1, h_2 : \prod_{x:X} F_1(x) \rightarrow F_2(x)$ be fiberwise maps and w.l.o.g. assume they coincide in u_1 , so $h_1(x_0, u_1) = h_2(x_0, u_2)$. Let $x : X$ and $u : F_1(x)$ be arbitrary, the goal is to prove the proposition $h_1(x, u) =_{F_2(x)} h_2(x, u)$. F_1 is connected, so the mere path type $\|(x_0, u_1) = (x, u)\|$ is inhabited. As the goal is a proposition, the truncation can be stripped, leaving an explicit path. This path is equivalent to a path $p : x_0 =_X x$ that satisfies $(\text{transport}^{F_1}(p, u_1) =_{F_1(x_0)} u)$. Lemma 1 implies that the families $h_i(x, -)$ commute with transport, so

$$h_i(x, u) = h_i(x, \text{transport}^{F_1}(p, u_1)) = \text{transport}^{F_2}(p, h_i(x_0, u_1))$$

for $i = 1, 2$. By the assumption $h_1(x_0, u_1) = h_2(x_0, u_1)$, we thus have $h_1(x, u) = h_2(x, u)$ for arbitrary x and u . \blacktriangleleft

Using Lemma 11 and 12 as replacements for the lifting criterion and the unique lifting property, we give a proof for the classification theorem in HoTT. Again, the proof itself closely follows the argument used by Hatcher [6].

Proof (Theorem 9). We need to show that the association $(F, u_0) \mapsto H_{(F, u_0)}$ is an equivalence, meaning that it is both injective and surjective.

The proof of surjectivity uses the universal covering space $P : X \rightarrow \mathbf{Set}$ constructed by Hou and Harper [9]; the fibers $P(x)$ are defined as the sets of paths from x_0 to x

$$P(x) := \|x_0 =_X x\|_0 .$$

Let $H : \pi_1(X, x_0) \rightarrow \mathbf{Prop}$ be a subgroup. We follow Hatcher [6] for the construction of a covering space F_H . Let \sim_x denote the relation on $\|x_0 =_X x\|_0$ defined by

$$q_1 \sim_x q_2 := H(q_1 \cdot q_2^{-1}) \quad \text{with } q_1, q_2 : \|x_0 =_X x\|_0 .$$

Since H is closed under group operations, \sim_x is an equivalence relation. The covering space $F_H : X \rightarrow \mathbf{Set}$ is then defined as the set-quotient of P w.r.t \sim :

$$F_H(x) := P(x) / \sim_x \quad \text{with designated point } u_H := [|\mathbf{refl}_{x_0}|_0] : F_H(x_0) .$$

Connectedness of F_H follows along similar lines as Lemma 3.11.8 in the HoTT book [17]. In order to reason about the terms $v : F_H(x)$ as being given by representatives $v \equiv [p']$ with $p' : \|x_0 =_X x\|$, we do need to perform induction on the quotient type. The (higher) coherence conditions required are satisfied because a type being connected is a proposition.

We need to show that the subgroup $H_{(F_H, u_H)}$ is equal to H . This is equivalent to showing that for all loops $p : x_0 =_X x_0$ there is an equivalence

$$H(|p|_0) \simeq (\mathbf{transport}^{F_H}(p, u_H) =_{F_H(x_0)} u_H) \quad (\equiv H_{(F_H, u_H)}(|p|_0)) .$$

(Being an equivalence is a proposition, so the truncation from loops in $\pi_1(X, x_0) \equiv \|x_0 =_X x_0\|_0$ can be stripped.) Let $p : x_0 =_X x_0$. By definition of (F_H, u_H) , the dependent loop type $(\mathbf{transport}^{F_H}(p, u_H) =_{F_H(x_0)} u_H)$ can be rewritten as $[|p|_0] =_{F_H(x_0)} [|\mathbf{refl}_{x_0}|_0]$ because $u_H \equiv [|\mathbf{refl}_{x_0}|_0]$ and

$$\mathbf{transport}^{F_H}(p, u_H) \equiv \mathbf{transport}^{F_H}(p, [|\mathbf{refl}_{x_0}|_0]) = [|\mathbf{transport}^{x \mapsto x_0 =_X x}(p, \mathbf{refl}_{x_0})|_0] = [|p|_0] ,$$

where we use that the family of maps $[|-|_0] : (x_0 =_X x) \rightarrow F_H(x)$ commutes with $\mathbf{transport}$ (Lemma 1). Since \sim_{x_0} is an equivalence relation, $[|p|_0] =_{F_H(x_0)} [|\mathbf{refl}_{x_0}|_0]$ is equivalent to $|p|_0 \sim_{x_0} |\mathbf{refl}_{x_0}|_0$, which, by definition, is equivalent to $H(|p|_0 \cdot |\mathbf{refl}_{x_0}|_0^{-1})$ and hence $H(|p|_0)$. Thus, the subgroups $H_{(F_H, u_H)}$ and H are equal.

We now show that the association $(F, u_0) \mapsto H_{(F, u_0)}$ is injective. Assume that for two connected, pointed covering spaces (F_1, u_1) and (F_2, u_2) , it holds that $H_{(F_1, u_1)} = H_{(F_2, u_2)}$. By definition of these subgroups, this implies that for all loops $p : x_0 =_X x_0$ we have

$$(\mathbf{transport}^{F_1}(p, u_1) =_{F_1(x_0)} u_1) \simeq (\mathbf{transport}^{F_2}(p, u_2) =_{F_2(x_0)} u_2) .$$

(Again, truncations can be stripped from loops in $\pi_1(X, x_0)$ as the goal $(F_1, u_1) = (F_2, u_2)$ is a proposition.) By Lemma 11 there exist fiberwise maps $h_{12} : \prod_{x:X} F_1(x) \rightarrow F_2(x)$ and $h_{21} : \prod_{x:X} F_2(x) \rightarrow F_1(x)$ such that $h_{12}(x_0, u_1) = u_2$ and $h_{21}(x_0, u_2) = u_1$. By Lemma 12, it holds that $h_{21} \circ h_{12} = \mathbf{id}_{F_1}$ and $h_{12} \circ h_{21} = \mathbf{id}_{F_2}$, because these maps coincide in the designated points (x_0, u_1) and (x_0, u_2) , namely

$$h_{21}(x_0, h_{12}(x_0, u_1)) = u_1 \equiv \mathbf{id}_{F_1}(x_0, u_1) \quad \text{and} \quad h_{12}(x_0, h_{21}(x_0, u_2)) = u_2 \equiv \mathbf{id}_{F_2}(x_0, u_2) .$$

Hence, h_{12} and h_{21} are each other's inverses, and so h_{12} is a basepoint-preserving, fiberwise equivalence from (F_1, u_1) to (F_2, u_2) . By Lemma 4 this means the pointed covering spaces are equal, thus proving the classification theorem. ◀

4 Canonical change of basepoint

To develop the necessary theory to prove Exercise 3.3.11 from Hatcher's *Algebraic Topology* [6] – our initial goal – we needed the existence of change-of-basepoint isomorphisms between homotopy groups of spheres, $\pi_n(S^n, x) \cong \pi_n(S^n, y)$. Such isomorphisms were needed to define the degree of a non-pointed map $S^n \rightarrow S^n$. The degree is easy to define for pointed maps, but for non-pointed maps you need a consistent way to associate $\pi_n(S^n, f(\text{base}))$ with $\pi_n(S^n, \text{base})$. In this section we prove some classical results on the existence of change-of-basepoint isomorphisms for connected spaces in general.

In classical homotopy theory, any path p from a to b in a topological space X induces a change-of-basepoint isomorphism between homotopy groups $\pi_n(X, a) \cong \pi_n(X, b)$. The isomorphism depends on the homotopy class of the path p . In the case that X is simply-connected, the isomorphism can be considered canonical – there is only one class of paths from a to b .

In homotopy type theory, this change-of-basepoint isomorphism is given by transport along a path $p : a =_X b$, but usually we do not have access to such a path explicitly. If X is connected, all we have is that the mere path type $\|a =_X b\|$ is inhabited. However, if transport along any specific path of type $a =_X b$ yields the same isomorphism, we can still obtain an explicit isomorphism from $\|a =_X b\|$ via extension by weak constancy (Lemma 2). The result we call a **canonical** change-of-basepoint isomorphism, since it is independent on the path $p : a =_X b$.

► **Note.** Because extension by weak constancy requires the constructed object's type to be a set, we cannot do away with the set-truncations in $\pi_n(X, x) \equiv \|\Omega^n(X, x)\|_0$ as we often could in Section 3.

The condition that transport along every path in $a =_X b$ results in the same map $\pi_n(X, a) \rightarrow \pi_n(X, b)$ is equivalent to the condition that the fundamental group $\pi_1(X, a)$ acts trivially on the higher homotopy groups $\pi_n(X, a)$. The latter is a well-studied property.

► **Definition 13** (π_1 -action). *Let (X, x_0) be a pointed type. The action of $\pi_1(X, x_0)$ on the higher homotopy groups $\pi_n(X, x_0)$ is defined on truncated loops $|p|_0 : \pi_1(X, x_0)$ by*

$$|p|_0 \cdot u := \text{transport}^{\pi_n(X, -)}(p^{-1}, u),$$

with $u : \pi_n(X, x_0)$. The inversion p^{-1} is to obtain a left action. The action is called trivial if multiplication by any term $p' : \pi_1(X, x_0)$ leaves $u : \pi_n(X, x_0)$ unchanged, i.e. $p' \cdot u = u$.

- **Lemma 14.** *Let X be a type with points $a, b : X$. Then the following propositions hold:*
- (i) *for all paths $q_1, q_2 : a =_X b$ we have $(\text{transport}^{\pi_n(X, -)}(q_1, -) = \text{transport}^{\pi_n(X, -)}(q_2, -))$ if the $\pi_1(X, a)$ -action on $\pi_n(X, a)$ is trivial.*
 - (ii) *the $\pi_1(X, a)$ -action on $\pi_n(X, a)$ is trivial if the mere path type $\|a =_X b\|$ is inhabited and for all $q_1, q_2 : a =_X b$ it holds that $(\text{transport}^{\pi_n(X, -)}(q_1, -) = \text{transport}^{\pi_n(X, -)}(q_2, -))$.*

► **Corollary 15.** *Let X be a type with points $a, b : X$. There exists a change-of-basepoint isomorphism $\varphi : \pi_n(X, a) \cong \pi_n(X, b)$ which is canonical, in the sense that for all $p : a =_X b$ it holds that $\varphi = \text{transport}^{\pi_n(X, -)}(p, -)$, if and only if the $\pi_1(X, a)$ -action on $\pi_n(X, a)$ is trivial.*

Proof (Lemma 14). We first show part (i). Assume that the $\pi_1(X, a)$ action on $\pi_n(X, a)$ is trivial. Since the goal is to show a proposition, this assumption can be stated as

$$\text{transport}^{\pi_n(X, -)}(p^{-1}, u) = u ,$$

for any loop $p : a =_X a$ and $u : \pi_n(X, a)$. Let $q_1, q_2 : a =_X b$, then $(q_1 \cdot q_2^{-1})$ is a loop of type $a =_X a$, so by assumption it holds that

$$\text{transport}^{\pi_n(X, -)}(q_2^{-1}, \text{transport}^{\pi_n(X, -)}(q_1, u)) = \text{transport}^{\pi_n(X, -)}(q_1 \cdot q_2^{-1}, u) = u ,$$

for all $u : \pi_n(X, a)$. The result follows by applying $\text{transport}^{\pi_n(X, -)}(q_2, -)$ to both sides.

Now for part (ii). Assume the mere path type $\|a =_X b\|$ is inhabited and that for all paths $q_1, q_2 : a =_X b$ it holds that $(\text{transport}^{\pi_n(X, -)}(q_1, -) = \text{transport}^{\pi_n(X, -)}(q_2, -))$. Now let $p' : \pi_1(X, a)$ and $u : \pi_n(X, a)$. Since the goal is to show a proposition, we may assume that $p' \equiv |p|_0$ for some loop $p : a =_X a$. We also strip the truncation from $\|a =_X b\|$ to obtain an explicit path $p_{ab} : a =_X b$. By assumption, transport along both paths $(p^{-1} \cdot p_{ab}) : a =_X b$ and $p_{ab} : a =_X b$ yields the same function, so

$$\begin{aligned} \text{transport}^{\pi_n(X, -)}(p_{ab}, \text{transport}^{\pi_n(X, -)}(p^{-1}, u)) &= \text{transport}^{\pi_n(X, -)}(p^{-1} \cdot p_{ab}, u) \\ &= \text{transport}^{\pi_n(X, -)}(p_{ab}, u) , \end{aligned}$$

for all $u : \pi_n(X, a)$. Applying $\text{transport}^{\pi_n(X, -)}(p_{ab}^{-1}, -)$ to both sides yield the desired result that $(\text{transport}^{\pi_n(X, -)}(p^{-1}, u) = u)$. ◀

The following theorem collects some results on triviality of the π_1 -action in homotopy type theory. Results (i) and (ii) cover the cases where $(X, x_0) \equiv (S^n, \text{base})$, giving us the canonical change-of-basepoint isomorphisms $\pi_n(S^n, x) \cong \pi_n(S^n, y)$ we needed to define the degree for non-pointed maps. In working on these results, we also managed to prove third, more complicated result, for which we were unable to find a reference in the classical theory.

► **Theorem 16.** *Let (X, x_0) be a pointed type.*

- (i) *If X is simply-connected, then the action of $\pi_1(X, x_0)$ on $\pi_n(X, x_0)$ is trivial for all $n \geq 1$;*
- (ii) *The fundamental group $\pi_1(X, x_0)$ is abelian if and only if the action of $\pi_1(X, x_0)$ on itself is trivial;*
- (iii) *If merely for all loops $p, q : \Omega(X, x_0)$ it holds that $p \cdot q = q \cdot p$, then the action of $\pi_1(X, x_0)$ on $\pi_n(X, x_0)$ is trivial for all $n \geq 1$.*

Results (i) and (ii) follow quickly in both the synthetic and classical setting.

Proof (Theorem 16, result (i)). Let X be a simply-connected space. We need to show that $(p' \cdot u =_{\pi_n(X, x_0)} u)$ for all $p' : \pi_1(X, x_0)$ and $u : \pi_n(X, x_0)$; this is a proposition, so it suffices to show this claim for truncated loops $p' \equiv |p|_0$ only. Let $p : x_0 =_X x_0$, then since X is simply-connected, there exists a mere homotopy $\|p^{-1} = \text{refl}_{x_0}\|$ between p^{-1} and the constant path refl_{x_0} . Again, we can strip the truncation to obtain an explicit homotopy $h : p^{-1} = \text{refl}_{x_0}$, which implies that

$$|p|_0 \cdot u \equiv \text{transport}^{\pi_n(X, -)}(p^{-1}, u) = \text{transport}^{\pi_n(X, -)}(\text{refl}_{x_0}, u) \equiv u ,$$

for all $u : \pi_n(X, x_0)$. ◀

1:16 Classification of Covering Spaces and Canonical Change of Basepoint

Proof (Theorem 16, result (ii)). First, note that since truncation $|-|_0 : (x = x) \rightarrow \pi_1(X, x)$ commutes with transport (Lemma 1), and transport in the loop space $(x_0 = x_0)$ equals conjugation (Lemma 2.11.2 in [17]), we have that for all loops $p, q : x_0 =_X x_0$,

$$\text{transport}^{\pi_1(X, -)}(p^{-1}, |q|_0) = |\text{transport}^{x \mapsto x =_X x}(p^{-1}, q)|_0 = |p \bullet q \bullet p^{-1}|_0 \equiv |p|_0 \cdot |q|_0 \cdot |p|_0^{-1},$$

where the final equality is because of the definition of the group operations on $\pi_1(X, x)$.

Now, assume that $\pi_1(X, x_0)$ is abelian and let $p', q' : \pi_1(X, x_0)$. Since the goal is to show a proposition, we may assume that $p' \equiv |p|_0$ and $q' \equiv |q|_0$ for some loops $p, q : x_0 =_X x_0$, so by commutativity of $\pi_1(X, x_0)$ we have that

$$\text{transport}^{\pi_1(X, -)}(p^{-1}, |q|_0) = |p|_0 \cdot |q|_0 \cdot |p|_0^{-1} = |q|_0.$$

Conversely, assume that the $\pi_1(X, x_0)$ -action on itself is trivial and let $p', q' : \pi_1(X, x_0)$. Since the goal is to show a proposition, we may again assume that $p' \equiv |p|_0$ and $q' \equiv |q|_0$ for some loops $p, q : x_0 =_X x_0$. Then, using the same equation as before, it holds that

$$|p|_0 \cdot |q|_0 \cdot |p|_0^{-1} = \text{transport}^{\pi_1(X, -)}(p^{-1}, |q|_0) = |q|_0,$$

so $p' \cdot q' \equiv |p|_0 \cdot |q|_0 = |q|_0 \cdot |p|_0 \equiv q' \cdot p'$. ◀

Before proving result (iii) of Theorem 16, let us discuss its assumption, namely that

$$\text{“merely for all loops } p, q : \Omega(X, x_0) \text{ it holds that } p \bullet q = q \bullet p\text{”}. \quad (3)$$

On the surface, this seems to just say that the fundamental group $\pi_1(X, x_0)$ is abelian, but statement (3) is stronger. Consider, for example, the space $S^1 \vee S^2$. Its fundamental group is abelian, but the action of π_1 on π_2 is not trivial. If (3) could be weakened to just demanding that the fundamental group $\pi_1(X, x_0)$ is abelian, then $S^1 \vee S^2$ would serve as a counterexample to result (iii).

The subtle difference between these statements is caused by where the propositional truncation, indicated by the word *merely*, is placed. In HoTT, there exists a map

$$\left\| \prod_{p, q : \Omega(X, x_0)} p \bullet q = q \bullet p \right\| \longrightarrow \prod_{p, q : \Omega(X, x_0)} \left\| p \bullet q = q \bullet p \right\|,$$

but, in general, this is not an equivalence. The type on the left-hand side expresses statement (3), the type on the right-hand side is equivalent, via Theorem 7.3.12 in [17], to the statement that the fundamental group $\pi_1(X, x_0)$ is abelian. According to Remark 3.8.4 in the HoTT book [17], it is admissible to assume that an inverse exists – which is equivalent to assuming the axiom of choice – if the loop space $\Omega(X, x_0)$ is a set. In general this is not the case, as with the space $S^1 \vee S^2$.

What are the kind of spaces that satisfy the hypothesis in result (iii)? Classically, loop spaces in which composition is commutative are called *homotopy commutative*. The commutativity is allowed to only hold up to homotopy, which is the default setting for equality between paths in homotopy type theory. By the Eckmann-Hilton argument, every H-space has a commutative loop space, but not every commutative loop space arises in this way, e.g. James Stasheff [16, Thm. 1.18] proved that $\mathbb{C}P^3$ has a commutative loop space. Hideyuki Kachi [11] also presents a plethora of finite CW complexes with commutative loop spaces, their focus is on simply-connected spaces – for which triviality of the π_1 -action is already covered by result (i) – but they also briefly discuss non simply-connected CW complexes.

Proving result (iii) takes more work than results (i) and (ii).

Proof (Theorem 16, result (iii)). Assume that merely for all loops $p, q : \Omega(X, x_0)$ it holds that $p \cdot q = q \cdot p$. We claim that it suffices to merely show that transport in Ω^n is trivial, i.e. that the type

$$\left\| \prod_{p : \Omega(X, x_0)} \prod_{u : \Omega^n(X, x_0)} \text{transport}^{\Omega^n(X, -)}(p^{-1}, u) =_{\Omega^n(X, x_0)} u \right\| \quad (4)$$

is inhabited. This type implies that for all loops $p : x_0 =_X x_0$ and $u : \Omega^n(X, x_0)$ the mere path type $\|\text{transport}^{\Omega^n(X, -)}(p^{-1}, u) =_{\Omega^n(X, x_0)} u\|$ is inhabited, which by Theorem 7.3.12 in [17], is equivalent to an equality between truncated loops, $|\text{transport}^{\Omega^n(X, -)}(p^{-1}, u)|_0 =_{\pi_n(X, x_0)} |u|_0$. As $|-|_0 : \Omega^n(X, x) \rightarrow \pi_n(X, x)$ commutes with transport by Lemma 1, this implies that

$$|p|_0 \cdot |u|_0 \equiv \text{transport}^{\pi_n(X, -)}(p^{-1}, |u|_0) = |\text{transport}^{\Omega^n(X, -)}(p^{-1}, u)|_0 = |u|_0,$$

which says that the action of $\pi_1(X, x_0)$ on $\pi_n(X, x_0)$ is trivial. (Since triviality of the $\pi_1(X, x_0)$ -action is a proposition, it suffices to only show the triviality for truncated loops of the form $|p|_0 : \pi_1(X, x_0)$ and $|u|_0 : \pi_n(X, x_0)$.)

We proceed to show statement (4) by induction on n . The base case $n = 1$ follows directly from the assumption. Since (4) is a proposition, the “merely” part in the initial assumption can be stripped, leaving us with the fact that composition in the loop space is commutative. Together with Lemma 2.11.2 in [17], which says that transport in $\Omega(X, x) \equiv (x =_X x)$ equals conjugation, it holds that

$$\text{transport}^{\Omega(X, -)}(p^{-1}, q) = p \cdot q \cdot p^{-1} = q \quad \text{for all loops } p, q : (x_0 =_X x_0).$$

For the inductive step, assume that (4) holds for n , we need to show it holds for $n + 1$. Since both the induction hypothesis and the goal are propositionally truncated, we can strip both truncations. The resulting induction hypothesis says that transport in Ω^n is trivial, i.e.

$$\text{transport}^{\Omega^n(X, -)}(p^{-1}, u) =_{\Omega^n(X, x_0)} u \quad \text{for all } p : x_0 =_X x_0 \text{ and } u : \Omega^n(X, x_0).$$

By functional extensionality, this gives an equality between maps $\Omega^n(X, x_0) \rightarrow \Omega^n(X, x_0)$,

$$h_p : \text{transport}^{\Omega^n(X, -)}(p^{-1}, -) = \text{id}_{\Omega^n(X, x_0)}. \quad (5)$$

Now, let $p : x_0 =_X x_0$ and $v : \Omega^{n+1}(X, x_0)$. The goal is to show that transport in Ω^{n+1} is trivial, i.e.

$$\text{transport}^{\Omega^{n+1}(X, -)}(p^{-1}, v) =_{\Omega^{n+1}(X, x_0)} v.$$

The next part involves a lot of complicated equations. We use a theorem from the HoTT book [17] to relate transport in Ω^{n+1} to transport in Ω^n ; from there we use the induction hypothesis, namely the path h_p , to relate transport in Ω^n to the identity on Ω^n ; using the previous two steps, we obtain an expression of transport in Ω^{n+1} as a conjugation of the original $(n + 1)$ -dimensional loop v in Ω^{n+1} ; using that composition in Ω^{n+1} commutes, we get that transport in Ω^{n+1} is trivial.

By Theorem 2.11.4 [17], transport in Ω^{n+1} can be related to transport in Ω^n . Recall that $v : \Omega^{n+1}(X, x_0) \equiv (\text{refl}_{x_0}^n = \text{refl}_{x_0}^n)$, so

$$\begin{aligned} & \text{transport}^{\Omega^{n+1}(X, -)}(p^{-1}, v) \\ &= (\text{apd}_{\text{refl}_{(-)}^n}(p^{-1}))^{-1} \cdot (\text{ap}_{\text{transport}^{\Omega^n(X, -)}(p^{-1}, -)}(v)) \cdot (\text{apd}_{\text{refl}_{(-)}^n}(p^{-1})), \end{aligned} \quad (6)$$

where $\text{apd}_{\text{refl}_{(-)}^n}(p^{-1}) : \text{transport}^{\Omega^n(X, -)}(p^{-1}, \text{refl}_{x_0}^n) =_{\Omega^n(X, x_0)} \text{refl}_{x_0}^n$ is a dependent equality between $\text{refl}_{x_0}^n$ and itself.

1:18 Classification of Covering Spaces and Canonical Change of Basepoint

Using the path h_p from the induction hypothesis (5), the middle term in (6) can be rewritten as

$$\begin{aligned} \mathbf{ap}_{\text{transport}^{\Omega^n(X, -)}(p^{-1}, -)}(v) &= \text{transport}^{\varphi \mapsto \varphi(\text{refl}_{x_0}^n) = \varphi(\text{refl}_{x_0}^n)}(h_p^{-1}, \mathbf{ap}_{\text{id}_{\Omega^n}}(v)) \\ &= (\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1}))^{-1} \cdot \mathbf{ap}_{\text{id}_{\Omega^n}}(v) \cdot (\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1})) \\ &\equiv (\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1}))^{-1} \cdot v \cdot (\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1})), \end{aligned} \quad (7)$$

where the second equality follows from Theorem 2.11.3 in [17]. The term $\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1})$ is of type $\text{refl}_{x_0}^n \equiv \text{id}_{\Omega^n}(\text{refl}_{x_0}^n) = \text{transport}^{\Omega^n(X, -)}(p^{-1}, \text{refl}_{x_0}^n)$, the reverse equality of the term $\mathbf{apd}_{\text{refl}_{x_0}^n}^n(p^{-1})$ in (6).

Combining equations (6) and (7), we have that $\text{transport}^{\Omega^{n+1}(X, -)}(p^{-1}, v)$ equals

$$(\mathbf{apd}_{\text{refl}_{x_0}^n}^n(p^{-1}))^{-1} \cdot (\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1}))^{-1} \cdot v \cdot (\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1})) \cdot (\mathbf{apd}_{\text{refl}_{x_0}^n}^n(p^{-1})).$$

Note that $(\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1})) \cdot (\mathbf{apd}_{\text{refl}_{x_0}^n}^n(p^{-1}))$ is a $(n+1)$ -dimensional loop of the type $\text{refl}_{x_0}^n = \text{refl}_{x_0}^n$, just like v . Transportation in Ω^{n+1} thus comes down to conjugation with some $(n+1)$ -cell, just like in Ω^1 . Since $n+1 \geq 2$, composition of $(n+1)$ -cells in Ω^{n+1} is commutative, which gives us that

$$\text{transport}^{\Omega^{n+1}(X, -)}(p^{-1}, v) = \tilde{p}^{-1} \cdot v \cdot \tilde{p} = v,$$

with $\tilde{p} \equiv (\mathbf{ap}_{\varphi \mapsto \varphi(\text{refl}_{x_0}^n)}(h_p^{-1})) \cdot (\mathbf{apd}_{\text{refl}_{x_0}^n}^n(p^{-1}))$, i.e. transport in Ω^{n+1} is trivial. \blacktriangleleft

Relation to free pointedness

Whereas we used change-of-basepoint isomorphisms $\pi_n(S^n, x) \cong \pi_n(S^n, y)$ to define the degree for non-pointed maps $S^n \rightarrow S^n$, Hou [7] takes a different approach: they observe that the degree-map $(S^n \rightarrow S^n) \rightarrow \mathbb{Z}$ takes values in a set, so it suffices to define the map on the set-truncation $\|S^n \rightarrow S^n\|_0$. They then use that, on the level of sets, pointedness is free, meaning that for $n \geq 1$ the projection that forgets about pointedness is an equivalence, i.e.

$$\|(S^n, \text{base}) \cdot \rightarrow (S^n, \text{base})\|_0 \simeq \|S^n \rightarrow S^n\|_0.$$

In [3], Buchholtz and Hou provide an intuition as to why this map is an equivalence: for any map $f : S^n \rightarrow S^n$, the suspension $\text{susp}(f) : S^{n+1} \rightarrow S^{n+1}$ is automatically pointed and by the Freudenthal suspension theorem all maps $S^{n+1} \rightarrow S^{n+1}$ are of this form.

Hou's approach to defining the degree map and the approach in this article are more similar than they appear on first sight. In this section, we show a classical statement relating basepoint-preserving and free homotopy classes of functions. In HoTT, these are encoded as the sets of pointed maps $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$ and non-pointed maps $\|Z \rightarrow X\|_0$. From this classical statement, it follows that the forgetful map

$$\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0 \longrightarrow \|Z \rightarrow X\|_0$$

is an equivalence precisely when the $\pi_1(X, x_0)$ -action on pointed maps $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$ is trivial. Moreover, for $(Z, z_0) \equiv (S^n, \text{base})$, the π_1 -action on $\|(S^n, \text{base}) \cdot \rightarrow (X, x_0)\|_0$ coincides with the π_1 -action on higher homotopy groups π_n under the equivalence

$$\|(S^n, \text{base}) \cdot \rightarrow (X, x_0)\|_0 \cong \pi_n(X, x_0)$$

from Lemma 6.5.4 in the HoTT book [17]. Thus, in the end both approaches rest on the triviality of the π_1 -action on the higher homotopy groups of spheres.

Similar to the π_1 -action on higher homotopy groups (Definition 13), the π_1 -action on the set of pointed maps is defined using transport. Following Hatcher [6], this action is defined as a right action, as opposed to the action on higher homotopy groups which is a left action.

► **Definition 17** (π_1 -action on pointed maps). *Let (Z, z_0) and (X, x_0) be pointed types. The action of $\pi_1(X, x_0)$ on the set of pointed maps $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$ is defined on truncated loops $|p|_0 : \pi_1(X, x_0)$ as*

$$f \cdot |p|_0 \equiv \text{transport}^{\|(Z, z_0) \cdot \rightarrow (X, -)\|_0}(p, f)$$

with $f : \|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$.

Alternatively, the π_1 -action on pointed maps can be expressed as follows.

► **Lemma 18.** *Let (Z, z_0) and (X, x_0) be pointed types. For all loops $p : x_0 =_X x_0$ and truncated, pointed maps $|(f, w_f)|_0 : \|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$, it holds that*

$$|(f, w_f)|_0 \cdot |p|_0 = |(f, w_f \cdot p)|_0 ,$$

i.e. the π_1 -action only acts on the proof of pointedness $w_f : f(z_0) =_X x_0$.

Proof. Let $|p|_0 : \pi_1(X, x_0)$ and $|(f, w_f)|_0 : \|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$. Transport in the type family $((Z, z_0) \cdot \rightarrow (X, -)) \equiv \sum_{(g:Z \rightarrow X)} g(z_0) =_X (-)$ can be rewritten as

$$\text{transport}^{(Z, z_0) \cdot \rightarrow (X, -)}(p, (f, w_f)) = (f, \text{transport}^{f(z_0) = (-)}(p, w_f)) . \quad (8)$$

This follows from a more general statement where p is a free path (meaning that we also have universal quantification over its endpoints), which holds by path induction. Using equation (8) and that truncation $|-|_0$ commutes with transport by Lemma 1, it holds that

$$\begin{aligned} \text{transport}^{\|(Z, z_0) \cdot \rightarrow (X, -)\|_0}(p, |(f, w_f)|_0) &= |\text{transport}^{(Z, z_0) \cdot \rightarrow (X, -)}(p, (f, w_f))|_0 \\ &= |(f, \text{transport}^{f(z_0) = (-)}(p, w_f))|_0 = |(f, w_f \cdot p)|_0 , \end{aligned}$$

where the final equality is by Theorem 2.11.2 in [17]. ◀

The π_1 -action on pointed maps coincides with the π_1 -action on higher homotopy groups, modulo an inversion to account for the difference in left and right action.

► **Lemma 19.** *Let (X, x_0) be a pointed type, then there exists an equivalence*

$$\varphi : \|(S^n, \text{base}) \cdot \rightarrow (X, x_0)\|_0 \cong \pi_n(X, x_0) ,$$

and $\varphi(f \cdot p) = p^{-1} \cdot \varphi(f)$ for loops $p : \pi_1(X, x_0)$ and maps $f : \|(S^n, \text{base}) \cdot \rightarrow (X, x_0)\|_0$.

Proof. The equivalence φ comes from Lemma 6.5.4 in [17]. Moreover, we actually have a family of equivalences $\varphi_x : \|(S^n, \text{base}) \cdot \rightarrow (X, x)\|_0 \cong \pi_n(X, x)$. Let $p' : \pi_1(X, x_0)$ and $f : \|(S^n, \text{base}) \cdot \rightarrow (X, x_0)\|_0$. The goal is to show a proposition, namely an equality in the set $\pi_n(X, x_0)$, so we can assume that $p' \equiv |p|_0$ for some $p : x_0 =_X x_0$. By Lemma 1, the family of equivalences commutes with transport, which gives us that

$$\varphi(f \cdot |p|_0) \equiv \varphi(\text{transport}^{\|(S^n, \text{base}) \cdot \rightarrow (X, -)\|_0}(p, f)) = \text{transport}^{\pi_n(X, -)}(p, \varphi(f)) = |p|_0^{-1} \cdot \varphi(f) . \quad \blacktriangleleft$$

We now show the classical statement relating basepoint-preserving homotopy classes $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$ and free homotopy classes $\|Z \rightarrow X\|_0$ in homotopy type theory. Note that the classical formulation requires Z to be restricted to the class of CW complexes, but this is not necessary in HoTT since ∞ -groupoids already behave well enough.

► **Lemma 20** (cf. [6, Prop. 4A.2]). *Let (Z, z_0) and (X, x_0) be pointed types with X connected. The map that forgets pointedness induces an equivalence between the orbit set $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0 / \pi_1(X, x_0)$ and $\|Z \rightarrow X\|_0$.*

The proof is kind of similar to the classical proof, but one needs to be able to read through the homotopical interpretation to see the correspondence.

Proof. First, note that $\mathbf{pr}_1 : ((Z, z_0) \cdot \rightarrow (X, x_0)) \rightarrow (Z \rightarrow X)$ indeed induces a map

$$\psi : \|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0 / \pi_1(X, x_0) \longrightarrow \|Z \rightarrow X\|_0 ,$$

which is given by $\psi([\!(f, w_f)\!|_0]) := |f|_0$. To show that ψ is well-defined, we need to prove that for all maps $f'_1, f'_2 : \|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$ in the same orbit it holds that $\psi(f'_1) = \psi(f'_2)$. (The higher coherence conditions are automatically satisfied because $\|Z \rightarrow X\|_0$ is a set.) If f'_1 and f'_2 are in the same orbit, this means there exists a truncated loop $p' : \pi_1(X, x_0)$ and a path $w_{p'} : (f'_1 \cdot p') = f'_2$. (Technically, we have only mere existence of p' and $w_{p'}$, but this can be stripped since $\psi(f'_1) = \psi(f'_2)$ is a proposition). Since one endpoint of w_p is free, namely f'_2 , we can perform path induction on $w_{p'}$, meaning that we may assume $f'_2 \equiv (f'_1 \cdot p')$. The goal is then to show that $\psi(f'_1) = \psi(f'_1 \cdot p')$. We may also assume that $p' \equiv |p|_0$ and $f'_1 \equiv |(f_1, w_{f_1})|_0$ for some $p : x_0 =_X x_0$ and $(f_1, w_{f_1}) : (Z, z_0) \cdot \rightarrow (X, x_0)$. By the alternative expression for the $\pi_1(X, x_0)$ -action (Lemma 18), it holds that

$$f'_1 \cdot p' \equiv |p|_0 \cdot |(f_1, w_{f_1})|_0 = |(f_1, w_{f_1} \cdot p)|_0 ,$$

from which it follows that

$$\psi(f'_1) \equiv \psi(|(f_1, w_{f_1})|_0) \equiv |f_1|_0 \equiv \psi(|(f_1, w_{f_1} \cdot p)|_0) = \psi(f'_1 \cdot p') .$$

Next, we show that ψ is surjective. Let $f : Z \rightarrow X$, the goal is to *merely* construct a term $f_\bullet : \|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0 / \pi_1(X, x_0)$ such that $\psi(f_\bullet) = |f|_0$. (We may strip the truncation from $\|Z \rightarrow X\|_0$ because the goal is a proposition). Since X is connected, $\|f(z_0) =_X x_0\|$ is inhabited. Note that the goal is to merely construct a term in the preimage of f , so we can strip this truncation, which yields an explicit path $p : f(z_0) =_X x_0$. We use this path to define f_\bullet , namely

$$f_\bullet := [|(f, p)|_0] : \|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0 / \pi_1(X, x_0) ,$$

and we have that $\psi(f_\bullet) \equiv \psi([|(f, p)|_0]) \equiv |f|_0$.

To show injectivity, consider two terms in $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0 / \pi_1(X, x_0)$ of the form $[|(f_1, w_{f_1})|_0]$ and $[|(f_2, w_{f_2})|_0]$ which satisfy $\psi([|(f_1, w_{f_1})|_0]) = \psi([|(f_2, w_{f_2})|_0])$, meaning $|f_1|_0 = |f_2|_0$. The goal is to show that $[|(f_1, w_{f_1})|_0] = [|(f_2, w_{f_2})|_0]$. (We may assume the terms $[|(f_i, w_{f_i})|_0]$ to be of this form because the goal is a proposition, which not only allows us to strip the truncation from $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$, but also to consider specific representatives of the equivalence classes.) By Theorem 7.3.12 in [17], the assumption $|f_1|_0 = |f_2|_0$ is equivalent to a mere equality $\|f_1 = f_2\|$. Stripping the truncation gives an explicit path $h' : f_1 = f_2$, and thus also a homotopy $h : \prod_z f_1(z) = f_2(z)$. The goal is equivalent to showing that $[|(f_1, w_{f_1})|_0]$ and $[|(f_2, w_{f_2})|_0]$ belong to the same orbit, i.e. showing there (merely) exists a loop $p : \pi_1(X, x_0)$ such that $[|(f_1, w_{f_1})|_0] \cdot p = [|(f_2, w_{f_2})|_0]$. Choose $p := |w_{f_1}^{-1} \cdot h(z_0) \cdot w_{f_2}|_0$, then by Lemma 18 it holds that

$$[|(f_1, w_{f_1})|_0] \cdot |w_{f_1}^{-1} \cdot h(z_0) \cdot w_{f_2}|_0 = [|(f_1, w_{f_1} \cdot (w_{f_1}^{-1} \cdot h(z_0) \cdot w_{f_2}))|_0] = [|(f_1, h(z_0) \cdot w_{f_2})|_0] .$$

It thus suffices to show that $(f_1, h(z_0) \cdot w_{f_2}) = (f_2, w_{f_2})$. To turn the equality $h' : f_1 = f_2$ into an equality of pointed maps $(f_1, h(z_0) \cdot w_{f_2}) = (f_2, w_{f_2})$, we need to show that $(\text{transport}^{f \mapsto f(z_0)=x_0}(h', h(z_0) \cdot w_{f_2}) = w_{f_2})$, which holds since

$$\text{transport}^{f \mapsto f(z_0)=x_0}(h', h(z_0) \cdot w_{f_2}) = h(z_0)^{-1} \cdot (h(z_0) \cdot w_{f_2}) = w_{f_2} ,$$

the first equality follows from path induction on h' . This concludes the proof that ψ is a well-defined equivalence. \blacktriangleleft

By the previous lemma, the forgetful map $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0 \rightarrow \|Z \rightarrow X\|_0$ is an equivalence precisely when the π_1 -action on pointed maps is trivial. Hence, both the free pointedness of maps $\|(Z, z_0) \cdot \rightarrow (X, x_0)\|_0$, and the existence of canonical change-of-basepoint isomorphisms $\pi_n(X, x) \cong \pi_n(X, y)$, are consequences of trivial π_1 -actions, the same π_1 -action, in fact, when $(Z, z_0) := (S^n, \text{base})$.

5 Discussion and Conclusion

In this article, we developed parts of algebraic topology in the synthetic language provided by homotopy type theory (HoTT). We proved a synthetic version of the classification of covering spaces, and synthetically explored the existence of canonical change-of-basepoint isomorphisms between homotopy groups.

Developing this theory synthetically required translating the classical definitions and statements into HoTT and there is some freedom in picking what translation to use. Some translations are easier to work with than others. For those interested in formalizing some topics from algebraic topology in HoTT themselves, we recommend the following:

1. Use type families $P : X \rightarrow \mathbf{Type}$ instead of the total space $\sum_X P$. Although this perspective takes some getting used to (especially when coming from classical mathematics), using type families directly allows more relations to be encoded as judgmental equalities instead of propositional ones, saving you from having to carry these around using transport.
2. Some direct translations using concepts from the HoTT book [17] can be simplified. In the case of the lifting criterion, we found an alternative condition that saved us from a lot of truncations (see Lemma 7).

We were able to closely mirror the classical proofs found in a standard reference like Hatcher’s *Algebraic Topology* [6]. Extension by weak constancy (Lemma 2) proved vital to reproduce classical constructions that rely on the mere existence of paths. The transport operation does feature more prominently in HoTT proofs than in the classical ones: it is used to construct both homotopy lifts, homotopy extensions, and to define dependent paths. The notions of transport and dependent paths do faithfully represent the geometrical reasoning in the classical proofs. (The abundance of transport operations did make us appreciate all the lemmas in the HoTT book characterizing transport in different type families [17, Ch. 2].)

We see both advantages and disadvantages with the synthetic approach to algebraic topology. The low-level encoding of mathematical concepts allows one to be more explicit about the objects involved without being overwhelmed by notation, and the theory can be mechanized in proof assistants much more easily. Truncations are a nice way to make explicit the difference between constructive and platonic existence statements in mathematics, as with the mere existence of a paths in a path-connected spaces. On the other hand, they are a nuisance to deal with. In Section 3, we managed to avoid a lot of truncations since the fibers $F(x)$ of a covering space are sets, but we were not so lucky in Section 4. By the end we had gotten used to working with them – and hopefully the reader with us.

At the start of this project, we only had a passing knowledge of what made HoTT special compared to the standard flavor of Martin-Löf type theory. Having formalized some classical results from algebraic topology ourselves, we have gained a better understanding and deeper appreciation for concepts like transport, truncations, and univalence.

References


- 1 Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017*, pages 164–172, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3018610.3018615.
- 2 Guillaume Brunerie. *On the homotopy groups of spheres in homotopy type theory*. PhD thesis, Laboratoire J.A. Dieudonné, August 2016. doi:10.48550/arXiv.1606.05916.
- 3 Ulrik Buchholtz and Kuen-Bang Hou. Cellular Cohomology in Homotopy Type Theory. *Logical Methods in Computer Science*, Volume 16, Issue 2, June 2020. doi:10.23638/LMCS-16(2:7)2020.
- 4 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.
- 5 Paolo Capriotti, Nicolai Kraus, and Andrea Vezzosi. Functions out of Higher Truncations. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, volume 41 of *LIPICs*, pages 359–373, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICs.CSL.2015.359.
- 6 Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002. URL: <https://pi.math.cornell.edu/~hatcher/AT/ATpage.html>.
- 7 Kuen-Bang Hou (Favonia). *Higher-Dimensional Types in the Mechanization of Homotopy Theory*. PhD thesis, Carnegie Mellon University, February 2017. URL: <https://favonia.org/thesis>.
- 8 Kuen-Bang Hou (Favonia), Eric Finster, Daniel R. Licata, and Peter LeFanu Lumsdaine. A Mechanization of the Blakers–Massey Connectivity Theorem in Homotopy Type Theory. 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 565–574, 2016. doi:10.1145/2933575.2934545.
- 9 Kuen-Bang Hou (Favonia) and Robert Harper. Covering Spaces in Homotopy Type Theory. In Silvia Ghilezan, Herman Geuvers, and Jelena Ivetić, editors, *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*, volume 97 of *Leibniz International Proceedings in Informatics (LIPICs)*, pages 11:1–11:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICs.TYPES.2016.11.
- 10 Kuen-Bang Hou (Favonia) and Michael Shulman. The Seifert-van Kampen Theorem in Homotopy Type Theory. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, volume 62 of *LIPICs*, pages 22:1–22:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICs.CSL.2016.22.
- 11 Hideyuki Kachi. Homotopy commutativity of the loop space of a finite CW-complex. *Hiroshima Mathematical Journal*, 20(2):365–384, 1990. doi:10.32917/hmj/1206129185.
- 12 Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of Univalent Foundations (after Voevodsky). *Journal of European Mathematical Society*, 23(6):2071–2126, March 2021. doi:10.4171/JEMS/1050.

- 13 Daniel R. Licata and Guillaume Brunerie. $\pi_n(S^n)$ in Homotopy Type Theory. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, pages 1–16, Cham, 2013. Springer International Publishing. doi:10.1007/978-3-319-03545-1_1.
- 14 Axel Ljungstrom and Anders Mörtberg. Formalizing $\pi_4(S^3) \cong \mathbb{Z}/2\mathbb{Z}$ and Computing a Brunerie Number in Cubical Agda. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, Los Alamitos, CA, USA, June 2023. IEEE Computer Society. doi:10.1109/LICS56636.2023.10175833.
- 15 Michael Shulman. Brouwer’s fixed-point theorem in real-cohesive homotopy type theory. *Mathematical Structures in Computer Science*, 28(6):856–941, 2018. doi:10.1017/S0960129517000147.
- 16 James Stasheff. On homotopy Abelian H-spaces. *Mathematical Proceedings of the Cambridge Philosophical Society*, 57(4):734–745, 1961. doi:10.1017/S0305004100035878.
- 17 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- 18 Jelle Wemmenhove, Cosmin Manea, and Jim Portegies. Formalization of Classification of Covering Spaces and Canonical Change of Basepoint in HoTT. Software, version 0.3., sw-hId: swh:1:dir:bea0c0af55e3ec9869679f3a5611cc5154a9ddb (visited on 2024-07-19). URL: <https://gitlab.tue.nl/computer-verified-proofs/covering-spaces>.

Finite Combinatory Logic with Predicates

Andrej Dudenhefner ✉ 


TU Dortmund University, Germany

Christoph Stahl ✉ 

TU Dortmund University, Germany

Constantin Chaumet ✉ 

TU Dortmund University, Germany

Felix Laarmann ✉ 

TU Dortmund University, Germany

Jakob Rehof ✉ 

TU Dortmund University, Germany

Lamarr Institute for Machine Learning and Artificial Intelligence, Dortmund, Germany

Abstract

Type inhabitation in extensions of Finite Combinatory Logic (FCL) is the mechanism underlying various component-oriented synthesis frameworks. In FCL inhabitant sets correspond to regular tree languages and vice versa. Therefore, it is not possible to specify non-regular properties of inhabitants, such as (dis)equality of subterms. Additionally, the monomorphic nature of FCL oftentimes hinders concise specification of components.

We propose a conservative extension to FCL by quantifiers and predicates, introducing a restricted form of polymorphism. In the proposed type system (FCLP) inhabitant sets correspond to decidable term languages and vice versa. As a consequence, type inhabitation in FCLP is undecidable. Based on results in tree automata theory, we identify a fragment of FCLP with the following two properties. First, the fragment enjoys decidable type inhabitation. Second, it allows for specification of local (dis)equality constraints for subterms of inhabitants.

For empirical evaluation, we implement a semi-decision procedure for type inhabitation in FCLP. We compare specification capabilities, scalability, and performance of the implementation to existing FCL-based approaches. Finally, we evaluate practical applicability via a case study, synthesizing mechanically sound robotic arms.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases combinatory logic, inhabitation, intersection types, program synthesis

Digital Object Identifier 10.4230/LIPIcs.TYPES.2023.2

Supplementary Material

Software (Source Code): <https://github.com/tudo-seal/clsp-python> [21]
archived at `swh:1:dir:8cb1770ba578c4d7e9a562f6513d82792e49e726`

Funding *Constantin Chaumet*: Deutsche Forschungsgemeinschaft (DFG, German Research Foundation – Project Number: 276879186).

1 Introduction

Type inhabitation in a type assignment system is the following problem: Given a type environment Γ and a type τ , is there a term M which can be assigned the type τ in the type environment Γ ? Type inhabitation can be understood as the search for a program (term M) which satisfies a given specification (type τ) under given assumptions (type environment Γ). For (polymorphic) λ -calculi type inhabitation corresponds to program synthesis from scratch [29, 30]. In comparison, for (variants of) combinatory logic type inhabitation corresponds to program synthesis from given domain-specific components [31].



© Andrej Dudenhefner, Christoph Stahl, Constantin Chaumet, Felix Laarmann, and Jakob Rehof; licensed under Creative Commons License CC-BY 4.0

29th International Conference on Types for Proofs and Programs (TYPES 2023).

Editors: Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg; Article No. 2; pp. 2:1–2:22

Leibniz International Proceedings in Informatics



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

Finite Combinatory Logic with Intersection Types $FCL(\cap, \leq)$ [32] is a monomorphic variant of combinatory logic with intersection types [17] relativized to arbitrary bases. Type inhabitation in $FCL(\cap, \leq)$ is $EXPTIME$ -complete [32, Theorem 12], which provides the basis for the Combinatory Logic Synthesizer (CLS) [8, 4]. CLS is a domain-agnostic program synthesis framework, and it has been applied in the following areas: Object-oriented program composition [7, 6], software product line design [25, 24], factory planning [37], motion planning [34], simulation model construction [28], and cyber-physical systems [14].

Previously, two extensions of $FCL(\cap, \leq)$ have been studied for which type inhabitation is decidable. First, Bounded Combinatory Logic [18] relaxes monomorphism to bounded schematism. Second, Finite Combinatory Logic with Boolean Queries [20] adds Boolean connectives \wedge , \vee , and \neg atop the type language. A major limitation of $FCL(\cap, \leq)$ and both aforementioned extensions is that inhabitant sets correspond to regular tree languages and vice versa [32, Corollary 11]. Therefore, neither theory allows for specification of non-regular properties¹ of inhabitants, such as (dis)equality of subterms. Additionally, monomorphism (and also bounded schematism) oftentimes hinders concise specification of domain-knowledge.

In the present work, we propose *Finite Combinatory Logic with Predicates* (FCLP) as a conservative extension to $FCL(\cap, \leq)$, addressing the above shortcomings. The main distinguishing property of FCLP is that inhabitant sets correspond to decidable term languages and vice versa. The type language of FCLP encompasses intersection types with the following three additions:

Literals: Literals are nullary type constructors, which may also occur in argument position in combinatory terms (facilitating a restricted form of dependent types [1]).

Quantifiers and Variables: Quantifiers bind variables in types, which allows for a restricted form of polymorphism, improving conciseness of specification.

Predicates: Decidable predicates reference variables, and allow for specification of non-regular properties of inhabitants.

The inclusion of decidable predicates entails undecidability of type inhabitation. The main contribution of the present work is the identification of an expressive fragment of FCLP which strictly includes $FCL(\cap, \leq)$, and in which type inhabitation is decidable. We formally describe a decision procedure for type inhabitation in the identified fragment. The decision procedure is similar to the existing decision procedure for type inhabitation in $FCL(\cap, \leq)$ [4, Definition 29]. However, instead of representing an inhabitant set by a regular tree grammar (or, a regular tree automaton), we represent an inhabitant set by means of the minimal Herbrand model of a logic program² (or, a tree automaton with term constraints [33]).

For empirical evaluation, an algorithm for the construction of inhabitants is implemented in the Python programming language. The algorithm is evaluated on the basis of a case study, undertaken for both $FCL(\cap, \leq)$ and the identified fragment of FCLP.

Synopsis. The present work is structured as follows:

Section 2: Definition of FCLP (Definition 8), decidability of type checking (Theorem 18), and undecidability of type inhabitation (Theorem 20).

Section 3: Fragment of FCLP (Problem 31) with decidable type inhabitation (Theorem 42).

Section 4: Implementation of FCLP in the Python programming language.

Section 5: Empirical evaluation of FCLP at the basis of a case study.

Section 6: Conclusion and remarks on future work.

¹ Czajka et.al. [5] specify non-regular properties as external restrictions via term rewriting systems.

² Kallat et.al. [27] combine regular tree grammars with SMT constraints in a logic program.

2 FCLP

In this section we present the type assignment system *Finite Combinatory Logic with Predicates* (FCLP). The system FCLP adds three new constructs to the existing *Finite Combinatory Logic with Intersection Types* $\text{FCL}(\cap, \leq)$ [32, Figure 3]. First, *literals* are both types and term arguments, and facilitate a restricted form of dependent types. Second, *quantifiers* bind literal variables and term variables in types, and allow for polymorphic specification. Third, decidable *predicates* reference literal variables and term variables in types, and describe term properties, which are difficult to specify otherwise.

Intersection types with covariant constructors [7, Definition 15] extended with literals (Definition 1) constitute the core of the type language of FCLP.

► **Definition 1** (Intersection Types with Covariant Constructors and Literals).

$$\text{INTERSECTION TYPES } \mathbb{T} \ni \sigma, \tau, \rho ::= \omega \mid \sigma \rightarrow \tau \mid \sigma \cap \tau \mid c(\sigma) \mid l$$

where ω is the universal type, c ranges over an enumerable set of unary type constructors, and l ranges over an enumerable set of literals.

The intersection type constructor (\cap) is considered associative, commutative, and idempotent. Additionally, standard rules of intersection type subtyping [17, Definition 1.3] are extended to covariant constructors in the following Definition 2.

► **Definition 2** (Intersection Type Subtyping). *The relation (\leq) is the least preorder on intersection types closed under the following rules:*

$$\begin{array}{lll} \sigma \leq \omega & \omega \leq \omega \rightarrow \omega & \\ \sigma \cap \tau \leq \sigma & \sigma \cap \tau \leq \tau & \text{if } \sigma \leq \tau_1 \text{ and } \sigma \leq \tau_2 \text{ then } \sigma \leq \tau_1 \cap \tau_2 \\ (\sigma \rightarrow \tau) \cap (\sigma \rightarrow \rho) \leq \sigma \rightarrow (\tau \cap \rho) & & \text{if } \sigma \leq \sigma' \text{ and } \tau \leq \tau' \text{ then } \sigma' \rightarrow \tau \leq \sigma \rightarrow \tau' \\ c(\sigma) \cap c(\tau) \leq c(\sigma \cap \tau) & & \text{if } \sigma \leq \sigma' \text{ then } c(\sigma) \leq c(\sigma') \end{array}$$

If $\sigma \leq \tau$ and $\tau \leq \sigma$, then we identify σ and τ , writing $\sigma = \tau$.

► **Remark 3.** Some existing extensions of Finite Combinatory Logic [4, Chapter 3] contain the binary product type constructor $\sigma \times \tau$. Observing the equivalence $\sigma \times \tau = (\sigma \times \omega) \cap (\omega \times \tau)$ [4, Definition 5], we omit an explicit product type constructor and represent products as intersections $\pi_1(\sigma) \cap \pi_2(\tau)$, where π_1 and π_2 are unary type constructors. In general, an n -ary constructor applied to $\sigma_1, \dots, \sigma_n$ is represented as the intersection $\bigcap_{i=1}^n c_i(\sigma_i)$ using unary type constructors c_1, \dots, c_n .

In practice, literals are partitioned in collections such as integers, floating point numbers, or character strings. We write $l : t$ to signify that the literal l belongs to a collection with the collection identifier t .

We tacitly extend intersection types by literal variables, ranged over by α, β, γ , and define *parameterized types* (the type language of FCLP) as follows.

► **Definition 4** (Parameterized Types).

$$\text{PARAMETERIZED TYPES } \varphi, \psi ::= \sigma \mid \langle \alpha : t \rangle \Rightarrow \varphi \mid \langle x : \sigma \rangle \Rightarrow \varphi \mid P \Rightarrow \varphi$$

where t ranges over collection identifiers and P ranges over decidable predicates³, possibly containing literal variables and term variables, ranged over by x, y, z . A literal variable α is bound in $\langle \alpha : t \rangle \Rightarrow \varphi$. A term variable x is bound in $\langle x : \sigma \rangle \Rightarrow \varphi$.

³ A predicate P is decidable, if there exists an effective procedure deciding whether P holds for given arguments. Syntactically, a predicate can be consider a first-order logic formula.

2:4 Finite Combinatory Logic with Predicates

Combinatory terms which may contain literals in argument position (Definition 5) constitute the term language of FCLP.

► **Definition 5** (Combinatory Terms and Arguments).

$$\begin{array}{l} \text{COMBINATORY TERMS} \quad \mathbb{C} \ni M, N ::= A \mid MT \\ \text{COMBINATORY ARGUMENTS} \quad T ::= M \mid l \end{array}$$

where A, B, C range over an enumerable set of combinators.

► **Definition 6** (Closed and Open Types, Substitutions). A parameterized type is closed, if every occurrence of a literal variable and every occurrence of a term variable is bound; otherwise the parameterized type is open. Literal variable substitution is denoted $\varphi[\alpha := l]$ and term variable substitution is denoted $\varphi[x := M]$.

In addition to *type environments* (finite sets of typed combinators), we introduce *literal environments* which contain pairs $l : t$, signifying that the literal l is associated with the collection identifier t .

► **Definition 7** (Type and Literal Environments).

$$\begin{array}{l} \text{TYPE ENVIRONMENT} \quad \Gamma ::= \{A_1 : \varphi_1, \dots, A_n : \varphi_n\} \text{ where } \varphi_1, \dots, \varphi_n \text{ are closed} \\ \text{LITERAL ENVIRONMENT} \quad \Delta ::= \{l_1 : t_1, \dots, l_n : t_n\} \\ \\ \text{DOMAIN} \quad \text{dom}(\{A_1 : \varphi_1, \dots, A_n : \varphi_n\}) = \{A_1, \dots, A_n\} \\ \quad \text{dom}(\{l_1 : t_1, \dots, l_n : t_n\}) = \{l_1, \dots, l_n\} \\ \text{RANGE} \quad \text{ran}(\{A_1 : \varphi_1, \dots, A_n : \varphi_n\}) = \{\varphi_1, \dots, \varphi_n\} \\ \quad \text{ran}(\{l_1 : t_1, \dots, l_n : t_n\}) = \{t_1, \dots, t_n\} \end{array}$$

Finally, we give the rules of Finite Combinatory Logic with Predicates (FCLP), deriving *judgments* $\Gamma; \Delta \vdash M : \varphi$, where φ is closed.

► **Definition 8** (Finite Combinatory Logic with Predicates (FCLP)).

$$\begin{array}{l} \frac{(A : \varphi) \in \Gamma}{\Gamma; \Delta \vdash A : \varphi} \text{ (Var)} \qquad \frac{\Gamma; \Delta \vdash M : P \Rightarrow \varphi \quad P \text{ holds}}{\Gamma; \Delta \vdash M : \varphi} \text{ (PE)} \\ \\ \frac{\Gamma; \Delta \vdash M : \langle \alpha : t \rangle \Rightarrow \varphi \quad (l : t) \in \Delta}{\Gamma; \Delta \vdash Ml : \varphi[\alpha := l]} \text{ (}\langle \rangle\text{E)} \quad \frac{\Gamma; \Delta \vdash M : \langle \langle x : \sigma \rangle \rangle \Rightarrow \varphi \quad \Gamma; \Delta \vdash N : \sigma}{\Gamma; \Delta \vdash MN : \varphi[x := N]} \text{ (}\langle \langle \rangle\text{)E)} \\ \\ \frac{\Gamma; \Delta \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma; \Delta \vdash M : \tau} (\leq) \qquad \frac{\Gamma; \Delta \vdash M : \sigma \rightarrow \tau \quad \Gamma; \Delta \vdash N : \sigma}{\Gamma; \Delta \vdash MN : \tau} (\rightarrow\text{E)} \end{array}$$

► **Remark 9.** The above rules (Var), (\leq), (\rightarrow E), together with the derivable intersection introduction rule (Lemma 14) constitute the original FCL(\cap, \leq) type system [32, Figure 3]. The additional rules $\langle \rangle$ E, $\langle \langle \rangle$ E, and PE mimic the pure type system application rule [1], where proofs that a predicate holds are irrelevant.

The notion of *paths* [18] is an essential ingredient in the algorithmic treatment of intersection type subtyping. Algebraically, paths are *prime factors* [4, Definition 10] into which each intersection type factorizes uniquely. In the following Definition 10 and Lemma 12 we recollect the notion and corresponding key property of paths.

► **Definition 10** (Path Decomposition [18, Lemma 1]).

$$\begin{array}{l} \mathbb{P}(\omega) = \emptyset \\ \mathbb{P}(\sigma \rightarrow \tau) = \{\sigma \rightarrow \tau' \mid \tau' \in \mathbb{P}(\tau)\} \\ \mathbb{P}(\sigma \cap \tau) = \mathbb{P}(\sigma) \cup \mathbb{P}(\tau) \\ \mathbb{P}(c(\sigma)) = \{c(\tau) \mid \tau \in \mathbb{P}(\sigma)\} \cup \{c(\omega)\} \\ \mathbb{P}(l) = \{l\} \end{array}$$

► **Remark 11.** If $\sigma \leq \tau$, then for all $\tau' \in \mathbb{P}(\tau)$ there exists $\sigma' \in \mathbb{P}(\sigma)$ such that $\sigma' \leq \tau'$ (easily shown for each rule in Definition 2). Therefore, $\sigma = \tau$ implies $\mathbb{P}(\sigma) = \mathbb{P}(\tau)$.

► **Lemma 12.** We have $\rho \leq \sigma_1 \rightarrow \dots \rightarrow \sigma_k \rightarrow \tau$ iff there exists a (possibly empty) set $\{\sigma_1^1 \rightarrow \dots \rightarrow \sigma_k^1 \rightarrow \tau^1, \dots, \sigma_1^m \rightarrow \dots \rightarrow \sigma_k^m \rightarrow \tau^m\} \subseteq \mathbb{P}(\rho)$ such that

1. $\sigma_j \leq \bigcap_{i=1}^m \sigma_j^i$ for $j = 1 \dots k$
2. $\bigcap_{i=1}^m \tau^i \leq \tau$

where the empty intersection denotes the universal type ω .

Proof. Immediate consequence of beta-soundness [2, Lemma 2.4] (shown inductively using the definition of intersection type subtyping). ◀

The following Lemma 13 (cf. [32, Lemma 4]) characterizes derivable judgments in FCLP.

► **Lemma 13 (Path Lemma).** We have $\Gamma; \Delta \vdash AT_1 \dots T_n : \tau$ iff for some typed combinator $(A : e_1 \Rightarrow \dots \Rightarrow e_m \Rightarrow \rho) \in \Gamma$ there exists a literal substitution θ such that

1. There exists a term substitution ξ such that for $i = 1, \dots, m$ we have
 - a. if $e_i = \langle \alpha : t \rangle$, then $\theta(\alpha) = T_i$ is a literal and $(T_i : t) \in \Delta$
 - b. if $e_i = \langle\langle x : \sigma \rangle\rangle$, then $\xi(x) = T_i \in \mathbb{C}$ and $\Gamma; \Delta \vdash T_i : \theta(\sigma)$
 - c. if $e_i = P$, then $\xi(\theta(P))$ holds
2. Let $k = n - m + p$ where p is the number of predicates in $\{e_1, \dots, e_m\}$, there exists a (possibly empty) set $\{\sigma_1^1 \rightarrow \dots \rightarrow \sigma_k^1 \rightarrow \tau^1, \dots, \sigma_1^q \rightarrow \dots \rightarrow \sigma_k^q \rightarrow \tau^q\} \subseteq \mathbb{P}(\theta(\rho))$ such that
 - a. $\Gamma; \Delta \vdash T_{m-p+j} : \bigcap_{i=1}^q \sigma_j^i$ for $j = 1 \dots k$
 - b. $\bigcap_{i=1}^q \tau^i \leq \tau$

Proof. The direction from right to left is obvious, observing that by Lemma 12 we have $\theta(\rho) \leq (\bigcap_{i=1}^q \sigma_1^i) \rightarrow \dots \rightarrow (\bigcap_{i=1}^q \sigma_k^i) \rightarrow \tau$. For the converse, we assume $\Gamma; \Delta \vdash AT_1 \dots T_n : \tau$. Necessarily, there is some $(A : e_1 \Rightarrow \dots \Rightarrow e_m \Rightarrow \rho) \in \Gamma$ such that each e_i is addressed by either $(\langle \rangle E)$, $(\langle\langle \rangle \rangle E)$, or (PE) . We collect the according literal and term instances in substitutions θ and ξ and obtain properties (1.a)–(1.c). Let $k = n - m + p$ where p is the number of predicates in $\{e_1, \dots, e_m\}$. We have that $\Gamma; \Delta \vdash AT_1 \dots T_{n-k} : \theta(\rho)$ such that the remaining k arguments are addressed by the rules $(\rightarrow E)$ and (\leq) where (shown by reordering) rule (\leq) never follows rule $(\rightarrow E)$. Finally, by Lemma 12 we obtain properties (2.a)–(2.b). ◀

At first glance, the intersection introduction rule $(\cap I)$ of $FCL(\cap, \leq)$ is missing from FCLP. However, using the above Lemma 13 the rule $(\cap I)$ is derivable (cf. [4, Lemma 11]).

► **Lemma 14.** The following rule is derivable:
$$\frac{\Gamma; \Delta \vdash M : \sigma \quad \Gamma; \Delta \vdash M : \tau}{\Gamma; \Delta \vdash M : \sigma \cap \tau} (\cap I) .$$

Proof. We assume $\Gamma; \Delta \vdash M : \sigma$ and $\Gamma; \Delta \vdash M : \tau$ and proceed by induction on the term $M = AT_1 \dots T_n$. W.l.o.g. $\sigma \neq \omega \neq \tau$. We have $(A : e_1 \Rightarrow \dots \Rightarrow e_m \Rightarrow \rho) \in \Gamma$ and substitutions $\theta_1, \theta_2, \xi_1, \xi_2$ which satisfy Lemma 13.1 and agree on the free variables in ρ . Let $k = n - m + p$ where p is the number of predicates in $\{e_1, \dots, e_m\}$. We have subsets $S_1, S_2 \subseteq \mathbb{P}(\theta_1(\rho)) = \mathbb{P}(\theta_2(\rho))$ such that $S_1 \cup S_2 = \{\sigma_1^1 \rightarrow \dots \rightarrow \sigma_k^1 \rightarrow \tau^1, \dots, \sigma_1^q \rightarrow \dots \rightarrow \sigma_k^q \rightarrow \tau^q\}$ which satisfy Lemma 13.2. By the induction hypothesis, we have $\Gamma; \Delta \vdash T_{m-p+j} : \bigcap_{i=1}^q \sigma_j^i$ for $j = 1 \dots k$, and we have $\bigcap_{i=1}^q \tau^i \leq \sigma \cap \tau$. By Lemma 13 we obtain $\Gamma; \Delta \vdash M : \sigma \cap \tau$. ◀

Since intersection introduction is derivable and the additional rules $(\langle \rangle E)$, $(\langle\langle \rangle \rangle E)$, and (PE) refer to new type constructors, FCLP is a conservative extension of $FCL(\cap, \leq)$.

► **Corollary 15.** Let Γ be a type environment such that $\text{ran}(\Gamma) \subseteq \mathbb{T}$, let M be a combinatory term which does not contain literals, and let τ be an intersection type. We have $\Gamma; \emptyset \vdash M : \tau$ iff $\Gamma \vdash M : \tau$ is a derivable judgment in the type assignment system $FCL(\cap, \leq)$.

Finally, we state the key decision problems: *Intersection type checking* (Problem 16) and *intersection type inhabitation* (Problem 17).

► **Problem 16** (Intersection Type Checking). Given a type environment Γ , a literal environment Δ , a combinatory term M , and an intersection type τ , does $\Gamma; \Delta \vdash M : \tau$ hold?

► **Problem 17** (Intersection Type Inhabitation). Given a type environment Γ , a literal environment Δ , and an intersection type τ , is there a combinatory term M such that $\Gamma; \Delta \vdash M : \tau$ holds?

Viewing the judgment $\Gamma; \Delta \vdash M : \tau$ in the context of component-oriented program synthesis, the type environment Γ contains domain-specific components specified by corresponding parameterized types, the literal environment Δ contains possible parameters, and the intersection type τ specifies desired programs.

While decidability of intersection type checking (Theorem 18) follows from Lemma 13, intersection type inhabitation is undecidable (Theorem 20).

► **Theorem 18.** *Intersection type checking $\Gamma; \Delta \vdash M : \tau$ is decidable.*

Proof. By induction on M , and as immediate consequence of Lemma 13 and decidability of each predicate occurring in parameterized types in $\text{ran}(\Gamma)$. ◀

► **Theorem 19.** *Intersection type inhabitation (Problem 17) is semi-decidable.*

Proof. Since intersection type checking is decidable (Theorem 18), a semi-decision procedure may enumerate and type check each combinatory term as a potential inhabitant. ◀

► **Theorem 20.** *Intersection type inhabitation (Problem 17) is undecidable.*

Proof. We reduce the halting problem to intersection type inhabitation. Let \mathcal{T} be a Turing machine and let $P(x)$ be a predicate on combinatory terms stating that \mathcal{T} halts on the empty word after exactly $\text{size}(x)$ steps (where $\text{size}(M)$ is the number of combinator occurrences in M). For any combinatory term M we have that $P(M)$ is decidable. Let

$$\Gamma = \{A : c(\omega) \cap (c(\omega) \rightarrow c(\omega)), B : \langle\langle x : c(\omega) \rangle\rangle \Rightarrow P(x) \Rightarrow d(\omega)\}$$

If $\Gamma; \emptyset \vdash M : d(\omega)$ for some combinatory term M , then M is of shape BN and \mathcal{T} halts on the empty word after $\text{size}(N)$ steps. Complementarily, if \mathcal{T} halts on the empty word after exactly k steps, then we have $\Gamma; \emptyset \vdash BN : d(\omega)$ where $N = A(\dots(AA)\dots)$ is of size k . Therefore, \mathcal{T} halts on the empty word iff there exists a term M such that $\Gamma; \emptyset \vdash M : d(\omega)$. ◀

► **Remark 21.** Emptiness and finiteness of the set $\{M \mid \Gamma; \Delta \vdash M : \tau\}$ of inhabitants are orthogonal problems. The proof of the above Theorem 20 gives a finite set of inhabitants with an undecidable emptiness question. Complementarily, let us consider the predicate $P'(x)$ on combinatory terms stating that $x = A$ or a given Turing machine \mathcal{T} halts on the empty word after at most $\text{size}(x)$ steps. The corresponding set of inhabitants is non-empty, and infinite iff \mathcal{T} halts on the empty word.

Since intersection type checking in FCLP is decidable (Theorem 18), the set of inhabitants $\{M \mid \Gamma; \Delta \vdash M : \tau\}$ is decidable. Complementarily, the following Theorem 22 shows that each decidable set of combinatory terms can be described by some set of inhabitants.

► **Theorem 22.** *Let \mathcal{M} be a decidable set of combinatory terms containing combinators drawn from a finite set \mathcal{A} . There exists a combinator B and a type environment Γ such that $\mathcal{M} = \{M \mid \Gamma; \emptyset \vdash BM : \omega\}$.*

Proof. Let $\Gamma' = \{A : \omega \mid A \in \mathcal{A}\}$ and $\Gamma = \Gamma' \cup \{B : \langle\langle x : \omega \rangle\rangle \Rightarrow (x \in \mathcal{M}) \Rightarrow \omega\}$ for a fresh combinator $B \notin \mathcal{A}$. Since $\mathcal{M} \subseteq \{M \mid \Gamma'; \emptyset \vdash M : \omega\}$, we have $\Gamma; \emptyset \vdash BM : \omega$ iff $M \in \mathcal{M}$. ◀

3 Decidable Inhabitation Fragment

While intersection type inhabitation (Problem 17) is undecidable in general (Theorem 20), FCLP contains fragments which enjoy decidable inhabitation. By Corollary 15 one such fragment is $\text{FCL}(\cap, \leq)$, for which inhabitation is EXPTIME -complete [32, Theorem 12]. In the remainder of this section we present a fragment of FCLP which strictly includes $\text{FCL}(\cap, \leq)$ and enjoys decidable inhabitation. The key idea is a reduction from intersection type inhabitation (under certain restrictions on predicates) to emptiness of bottom-up tree automata with term constraints [33].

The following Definition 23 specifies the *arity* of a parameterized type. If a combinator is applied to a number of arguments exceeding the arity of its type, then the only type assigned to such an application is the universal type ω .

► **Definition 23** (Arity).

$$\begin{aligned} \text{ar}(\langle \alpha : t \rangle \Rightarrow \varphi) &= \text{ar}(\langle x : \sigma \rangle \Rightarrow \varphi) = 1 + \text{ar}(\varphi) \\ \text{ar}(P \Rightarrow \varphi) &= \text{ar}(\varphi) \\ \text{ar}(\omega) &= \text{ar}(c(\sigma)) = \text{ar}(l) = \text{ar}(\alpha) = 0 \\ \text{ar}(\sigma \rightarrow \tau) &= 1 + \text{ar}(\tau) \quad \text{where } \tau \neq \omega \\ \text{ar}(\sigma \cap \tau) &= \max\{\text{ar}(\sigma), \text{ar}(\tau)\} \end{aligned}$$

► **Lemma 24** (Maximal Arity). *Given environments Γ, Δ , for a typed combinator $(A : \varphi) \in \Gamma$, an $n > \text{ar}(\varphi)$, combinatory arguments T_1, \dots, T_n , and an intersection type τ , we have that if $\Gamma; \Delta \vdash AT_1 \dots T_n : \tau$, then $\tau = \omega$.*

Proof. For Γ, Δ , and $(A : e_1 \Rightarrow \dots \Rightarrow e_m \Rightarrow \rho) \in \Gamma$ let $n > \text{ar}(e_1 \Rightarrow \dots \Rightarrow e_m \Rightarrow \rho)$ and let $n' = n - \text{ar}(e_1 \Rightarrow \dots \Rightarrow e_m \Rightarrow \omega)$. By induction on ρ for any literal substitution θ we have that $\sigma'_1 \rightarrow \dots \rightarrow \sigma'_{n'} \rightarrow \tau' \notin \mathbb{P}(\theta(\rho))$ for any types $\sigma'_1, \dots, \sigma'_{n'}, \tau'$. Assuming $\Gamma; \Delta \vdash AT_1 \dots T_n : \tau$, by Lemma 13.2.b we obtain $\omega \leq \tau$, showing the claim. ◀

Combinatory terms can be naively represented as binary trees, having combinators and literals as leaves and binary term application as inner nodes. However, the naive representation is inappropriate for certain tree constraints. For example, brother equality and disequality constraints [10] compare the terms M and N in the application MN for the naive representation, which is of little interest in practice. In the following, we represent a combinatory term $AT_1 \dots T_n$ as the tree with root $A_{(n)}$ with n children T_1, \dots, T_n .

► **Definition 25** (Tree Representation).

$$\begin{aligned} \text{tree}(l) &= l \\ \text{tree}(AT_1 \dots T_n) &= A_{(n)}(\text{tree}(T_1), \dots, \text{tree}(T_n)) \end{aligned}$$

In general, for a combinator A we have infinitely many symbols $A_{(0)}, A_{(1)}, A_{(2)}, \dots$, which is unsatisfactory for tree languages over a *finite* signature. Fortunately, relying on Lemma 24 for a typed combinator $A : \varphi$ we can reasonably bound the arity by $\text{ar}(\varphi)$ in the following Definition 26.

► **Definition 26** (Arity Respect). *We say that a combinatory term $AT_1 \dots T_n$ respects arities in Γ , if $(A : \varphi) \in \Gamma$, $n \leq \text{ar}(\varphi)$, and each T_i which is not a literal respects arities in Γ .*

The set of combinatory terms M which contain literals from $\text{dom}(\Delta)$ and respect arities in Γ is denoted $\mathbb{C}(\Gamma, \Delta)$.

The following Lemma 27 shows that for intersection type inhabitation in the fragment corresponding to $\text{FCL}(\cap, \leq)$ (cf. Corollary 15) it suffices to consider inhabitants which respect arities in Γ .

► **Lemma 27.** *Let Γ be a type environment such that $\text{ran}(\Gamma) \subseteq \mathbb{T}$, let M be a combinatory term which does not contain literals, and let τ be an intersection type. If $\Gamma; \emptyset \vdash M : \tau$, then there exists a combinatory term $N \in \mathbb{C}(\Gamma, \emptyset)$ such that $\Gamma; \emptyset \vdash N : \tau$.*

Proof. Assume $\Gamma; \emptyset \vdash M : \tau$ where $\text{ran}(\Gamma) \subseteq \mathbb{T}$; we proceed by induction on the number of combinator occurrences in M . If $M \in \mathbb{C}(\Gamma, \emptyset)$ we obtain the claim, otherwise the derivation of $\Gamma; \emptyset \vdash M : \tau$ contains a subderivation $\Gamma; \emptyset \vdash A M_1 \dots M_{\text{ar}(\sigma)} M' : \rho$ for some $(A : \sigma) \in \Gamma$ and an intersection type ρ . By Lemma 24 we have $\rho = \omega$. Therefore, we can derive $\Gamma; \emptyset \vdash A M_1 \dots M_{\text{ar}(\sigma)} : \rho$, replace the original subderivation, and obtain the claim by the induction hypothesis. ◀

In general, for intersection type inhabitation the following Remark 28 shows that inhabitants which respect arities in Γ do not suffice.

► **Remark 28.** For the type environment $\Gamma = \{A : \omega, B : \langle\langle x : \omega \rangle\rangle \Rightarrow (x \neq A) \Rightarrow l\}$ we have $\Gamma; \emptyset \vdash B(AA) : l$. Since $\text{ar}(\omega) = 0$ we have $B(AA) \notin \mathbb{C}(\Gamma, \emptyset)$. The only term $N \in \mathbb{C}(\Gamma, \emptyset)$ such that $\Gamma; \emptyset \vdash N : \omega$ is the term $N = A$. Therefore, there is no combinatory term $M \in \mathbb{C}(\Gamma, \emptyset)$ such that $\Gamma; \emptyset \vdash M : l$.

We consider a fragment of FCLP for which any predicate occurring in parameterized types in $\text{ran}(\Gamma)$ is of specific shape (Definition 29 and Definition 30), motivated by tree automata with term constraints [33].

► **Definition 29 (Term Constraint).** *For a type environment Γ , a predicate occurring in a parameterized type in $\text{ran}(\Gamma)$ is a term constraint, if it is either $x = M$ or $x \neq M$ where M is (abusing notation) an open term (possibly containing variables) which respects arities in Γ .*

► **Definition 30 (Literal Constraint).** *A predicate P is a literal constraint if every free variable occurring in P is a literal variable.*

Restricting predicates to literal and term constraints we define the following inhabitation problem.

► **Problem 31 (Intersection Type Inhabitation with Literal and Term Constraints).** Given a type environment Γ such that each predicate occurring in a parameterized type in $\text{ran}(\Gamma)$ is a literal constraint or a term constraint, a literal environment Δ , and an intersection type τ , is there a combinatory term $M \in \mathbb{C}(\Gamma, \Delta)$ such that $\Gamma; \Delta \vdash M : \tau$ holds?

We follow the approach initiated by Frühwirth [23] (an overview is given by Jacquemard [26, Section I.2]) and describe tree languages via finite sets of Horn clauses over certain first-order signatures. The following Definition 32 establishes suitable first-order signatures, based on environments and a set of intersection types.

► **Definition 32 (Signature).** *For environments Γ, Δ and a finite set of intersection types Ξ the signature $\Sigma(\Gamma, \Delta, \Xi)$ contains the following:*

- nullary function symbols l for each $l \in \text{dom}(\Delta)$
- n -ary function symbols $A_{(n)}$ for each $(A : \varphi) \in \Gamma$ and $n \leq \text{ar}(\varphi)$
- unary predicates Q_τ for each $\tau \in \Xi$
- unary predicates Q_t for each $t \in \text{ran}(\Delta)$
- a binary equality ($=$) predicate and a binary disequality (\neq) predicate

The Herbrand universe over the signature $\Sigma(\Gamma, \Delta, \Xi)$ is $\{\text{tree}(T) \mid T \in \mathbb{C}(\Gamma, \Delta) \cup \text{dom}(\Delta)\}$. Horn clauses over the signature $\Sigma(\Gamma, \Delta, \Xi)$ are of shape $H \leftarrow H_1, \dots, H_m$, where H is the *head* of the clause and H_1, \dots, H_m are *antecedents* of the clause. Specifically, we consider Horn clauses with heads of shape either $Q_t(l)$ such that $(l : t) \in \Delta$, or $Q_\tau(A_{(n)}(X_1, \dots, X_n))$ where X_1, \dots, X_n are free first-order variables. Let us recall in the following Definition 33 the standard Herbrand semantics [36] for signatures $\Sigma(\Gamma, \Delta, \Xi)$.

► **Definition 33 (Model).** *Let Γ, Δ be environments, let $M \in \mathbb{C}(\Gamma, \Delta)$ be a combinatory term, let Ξ be a finite set of intersection types, let \mathcal{H} be a finite set of Horn clauses over the signature $\Sigma(\Gamma, \Delta, \Xi)$, and let $\tau \in \Xi$ be a type. We write $\mathcal{H} \Vdash Q_\tau(\text{tree}(M))$ if $Q_\tau(\text{tree}(M))$ is true in the smallest Herbrand model in which every Horn clause from \mathcal{H} is true.*

Next, given environments Γ (restricted to literal and term constraints) and Δ , and an intersection type τ , we present a terminating algorithm INH which computes a set \mathcal{H} of Horn clauses such that for any $M \in \mathbb{C}(\Gamma, \Delta)$ we have $\Gamma; \Delta \vdash M : \tau$ iff $\mathcal{H} \Vdash Q_\tau(\text{tree}(M))$.

► **Definition 34 (Algorithm $\text{INH}_{\Gamma, \Delta}(\tau, \Xi)$).** *Let Γ be a type environment such that each predicate occurring in a parameterized type in $\text{ran}(\Gamma)$ is a literal constraint or a term constraint and let Δ be a literal environment. For an intersection type τ and a set Ξ of intersection types let*

$$\text{INH}_{\Gamma, \Delta}(\tau, \Xi) = \begin{cases} \emptyset & \text{if } \tau \in \Xi \\ \bigcup_{(A:\varphi) \in \Gamma} \text{REC}_{\Gamma, \Delta, \tau, (\Xi \cup \tau)}^A((\cdot), \varphi, (\cdot), \emptyset) & \text{otherwise} \end{cases}$$

where Algorithm $\text{REC}_{\Gamma, \Delta, \tau, \Xi}^A$ is defined as follows. The arguments of $\text{REC}_{\Gamma, \Delta, \tau, \Xi}^A$ are

- a list \vec{X} of distinct first-order variables
- a parameterized type φ
- a list \vec{H} of antecedents
- a finite set \mathcal{H} of Horn clauses

The result of $\text{REC}_{\Gamma, \Delta, \tau, \Xi}^A(\vec{X}, \varphi, \vec{H}, \mathcal{H})$ is a set of Horn clauses computed as follows. Consider the shape of φ :

Case φ is $\langle \alpha : t \rangle \Rightarrow \psi$: let Y be a fresh first-order variable and return

$$\bigcup_{(l:t) \in \Delta} \text{REC}_{\Gamma, \Delta, \tau, \Xi}^A((\vec{X}, Y), \psi[\alpha := l], (\vec{H}, Q_t(Y), Y = l), \mathcal{H} \cup \{Q_t(l) \leftarrow\})$$

Case φ is $\langle\langle x : \sigma \rangle\rangle \Rightarrow \psi$: let Y be a fresh first-order variable and return

$$\text{REC}_{\Gamma, \Delta, \tau, \Xi}^A((\vec{X}, Y), \psi[x := Y], (\vec{H}, Q_\sigma(Y)), \mathcal{H} \cup \text{INH}_{\Gamma, \Delta}(\sigma, \Xi))$$

Case φ is $P \Rightarrow \psi$ such that P is closed: if P does not hold return \emptyset , otherwise return

$$\text{REC}_{\Gamma, \Delta, \tau, \Xi}^A(\vec{X}, \psi, \vec{H}, \mathcal{H})$$

Case φ is $(X = M) \Rightarrow \psi$ where M may contain free first-order variables: return

$$\text{REC}_{\Gamma, \Delta, \tau, \Xi}^A(\vec{X}, \psi, (\vec{H}, X = \text{tree}(M)), \mathcal{H})$$

Case φ is $(X \neq M) \Rightarrow \psi$ where M may contain free first-order variables: return

$$\text{REC}_{\Gamma, \Delta, \tau, \Xi}^A(\vec{X}, \psi, (\vec{H}, X \neq \text{tree}(M)), \mathcal{H})$$

Case φ is ρ for some intersection type ρ : return

$$\mathcal{H} \cup \bigcup_{k=0}^{\text{ar}(\rho)} \bigcup_{S \subseteq \mathbb{P}(\rho)} \mathcal{H}_{k,S} \text{ where } \mathcal{H}_{k,S} \text{ is defined as follows.}$$

If $S = \{\sigma_1^1 \rightarrow \dots \rightarrow \sigma_k^1 \rightarrow \tau^1, \dots, \sigma_1^q \rightarrow \dots \rightarrow \sigma_k^q \rightarrow \tau^q\}$ and $\bigcap_{i=1}^q \tau^i \leq \tau$, then let Y_1, \dots, Y_k be fresh first-order variables, let $\sigma_j = \bigcap_{i=1}^q \sigma_j^i$ for $j = 1 \dots k$, and let n be the length of the list $(\vec{X}, Y_1, \dots, Y_k)$ in

$$\mathcal{H}_{k,S} = \{Q_\tau(A_{(n)}(\vec{X}, Y_1, \dots, Y_k)) \leftarrow \vec{H}, Q_{\sigma_1}(Y_1), \dots, Q_{\sigma_k}(Y_k)\} \cup \bigcup_{j=1}^k \text{INH}_{\Gamma, \Delta}(\sigma_j, \Xi)$$

Otherwise, $\mathcal{H}_{k,S} = \emptyset$.

The following Example 35 illustrates an invocation of Algorithm INH for the type environment from Remark 28.

► **Example 35.** Consider the type environment $\Gamma = \{A : \omega, B : \langle x : \omega \rangle \Rightarrow (x \neq A) \Rightarrow l\}$ from Remark 28 where l is some literal. We have

$$\begin{aligned} & \text{INH}_{\Gamma, \emptyset}(l, \emptyset) \\ &= \text{REC}_{\Gamma, \emptyset, l, \{l\}}^A((\cdot), \omega, (\cdot), \emptyset) \cup \text{REC}_{\Gamma, \emptyset, l, \{l\}}^B((\cdot), \langle x : \omega \rangle \Rightarrow (x \neq A) \Rightarrow l, (\cdot), \emptyset) \\ &= \emptyset \cup \text{REC}_{\Gamma, \emptyset, l, \{l\}}^B((Y), (Y \neq A) \Rightarrow l, (Q_\omega(Y)), \emptyset \cup \text{INH}_{\Gamma, \emptyset}(\omega, \{l\})) \\ &= \text{REC}_{\Gamma, \emptyset, l, \{l\}}^B((Y), l, (Q_\omega(Y), (Y \neq A_{(0)})), \text{INH}_{\Gamma, \emptyset}(\omega, \{l\})) \\ &= \text{INH}_{\Gamma, \emptyset}(\omega, \{l\}) \cup \{Q_l(B_{(1)}(Y)) \leftarrow Q_\omega(Y), (Y \neq A_{(0)})\} \\ &= \dots \\ &= \{Q_\omega(A_{(0)}) \leftarrow, \\ &\quad Q_\omega(B_{(1)}(Z)) \leftarrow Q_\omega(Z), (Z \neq A_{(0)}), \\ &\quad Q_l(B_{(1)}(Y)) \leftarrow Q_\omega(Y), (Y \neq A_{(0)})\} \end{aligned}$$

In accordance with Remark 28, in the smallest Herbrand model in which every Horn clause from the above set $\text{INH}_{\Gamma, \emptyset}(l, \emptyset)$ is true we have

- $Q_\omega = \{A_{(0)}\} = \{\text{tree}(M) \mid M \in \mathbb{C}(\Gamma, \emptyset) \text{ such that } \Gamma; \emptyset \vdash M : \omega\}$
- $Q_l = \emptyset = \{\text{tree}(M) \mid M \in \mathbb{C}(\Gamma, \emptyset) \text{ such that } \Gamma; \emptyset \vdash M : l\}$

The following Example 36 illustrates the result of Algorithm INH in the presence of literal constraints (cf. Section 5).

► **Example 36.** Consider the literal environment $\Delta = \{0 : \text{int}, 1 : \text{int}, 2 : \text{int}, 3 : \text{int}\}$ and the type environment $\Gamma = \{A : 0, B : \langle \alpha : \text{int} \rangle \Rightarrow \langle \beta : \text{int} \rangle \Rightarrow (\beta = \alpha + 1) \Rightarrow \alpha \rightarrow \beta\}$. We have

$$\begin{aligned} & \text{INH}_{\Gamma, \Delta}(2, \emptyset) \\ &= \{Q_0(A_{(0)}) \leftarrow, \\ &\quad Q_1(B_{(3)}(X_1, X_2, X_3)) \leftarrow Q_{\text{int}}(X_1), (X_1 = 0), Q_{\text{int}}(X_2), (X_2 = 1), Q_0(X_3), \\ &\quad Q_2(B_{(3)}(Y_1, Y_2, Y_3)) \leftarrow Q_{\text{int}}(Y_1), (Y_1 = 1), Q_{\text{int}}(Y_2), (Y_2 = 2), Q_1(Y_3), \\ &\quad Q_{\text{int}}(0) \leftarrow, Q_{\text{int}}(1) \leftarrow, Q_{\text{int}}(2) \leftarrow\} \end{aligned}$$

In the smallest Herbrand model in which every Horn clause from the above set $\text{INH}_{\Gamma, \Delta}(2, \emptyset)$ is true we have $Q_0 = \{A_{(0)}\}$, $Q_1 = \{B_{(3)}(0, 1, A_{(0)})\}$, $Q_2 = \{B_{(3)}(1, 2, B_{(3)}(0, 1, A_{(0)}))\}$, and $Q_{\text{int}} = \{0, 1, 2\}$. Specifically, we have $Q_i = \{\text{tree}(M) \mid M \in \mathbb{C}(\Gamma, \Delta) \text{ such that } \Gamma; \Delta \vdash M : i\}$ for $i \in \{0, 1, 2\}$. The literal $3 \in \text{dom}(\Delta)$ does not occur in Horn clauses in $\text{INH}_{\Gamma, \Delta}(2, \emptyset)$.

Termination of $\text{INH}_{\Gamma, \Delta}(\tau, \Xi)$ is shown using an upper bound on the set Ξ of types and the fact that Ξ strictly increases in recursive invocations of $\text{INH}_{\Gamma, \Delta}$.

► **Lemma 37.** *For any type environment Γ , literal environment Δ , intersection type τ , and set Ξ of intersection types we have that Algorithm $\text{INH}_{\Gamma,\Delta}(\tau, \Xi)$ terminates.*

Proof. Recursive invocations of $\text{INH}_{\Gamma,\Delta}$ increase the set Ξ by the considered type τ , such that for some literal substitution θ with range $\text{dom}(\Delta)$ one of the following conditions holds:

- $\tau = \theta(\sigma)$ such that $\langle\langle x : \sigma \rangle\rangle$ is a binder occurring in a parameterized type in $\text{ran}(\Gamma)$
- $\tau = \bigcap_{i=1}^q \sigma_j^i$ such that $\{\sigma_1^1 \rightarrow \dots \rightarrow \sigma_k^1 \rightarrow \tau^1, \dots, \sigma_1^q \rightarrow \dots \rightarrow \sigma_k^q \rightarrow \tau^q\} \subseteq \mathbb{P}(\theta(\rho))$ for some ρ occurring in a parameterized type in $\text{ran}(\Gamma)$ and $k \leq \text{ar}(\theta(\rho))$

Since Γ , Δ , the number of literal substitution θ with range $\text{dom}(\Delta)$, and the number of distinct subsets of $\mathbb{P}(\theta(\rho))$ are finite, the number of types τ obeying the above restriction is finite. Therefore, the number of recursive invocations of $\text{INH}_{\Gamma,\Delta}$ is finite. ◀

The following Theorem 38 shows that $\text{INH}_{\Gamma,\Delta}(\tau, \emptyset)$ computes Horn clauses which characterize inhabitants (respecting arities in Γ) of type τ .

► **Theorem 38 (Correctness).** *Let Γ, Δ be environments such that each predicate occurring in a parameterized type in $\text{ran}(\Gamma)$ is a literal constraint or a term constraint, let Ξ be a set of intersection types, let $\tau \in \Xi$, and let \mathcal{H} be the set $\text{INH}_{\Gamma,\Delta}(\tau, \emptyset)$ of Horn clauses over the signature $\Sigma(\Gamma, \Delta, \Xi)$. We have $\Gamma; \Delta \vdash M : \tau$ iff $\mathcal{H} \Vdash Q_\tau(\text{tree}(M))$ for any $M \in \mathbb{C}(\Gamma, \Delta)$.*

Proof. W.l.o.g. we assume that distinct bound variables have distinct names and there is a bijection μ between term variables and first-order variables such that in case $\langle\langle x : \sigma \rangle\rangle \Rightarrow \psi$ of Algorithm REC the chosen fresh first-order variable is $\mu(x)$.

For the implication from left to right, we assume $\Gamma; \Delta \vdash M : \tau$ and proceed by induction on M . We have $M = AT_1 \dots T_n$ such that $(A : \varphi) \in \Gamma$, $n \leq \text{ar}(\varphi)$, and there exists a literal substitution θ and a term substitution ξ satisfying properties of Lemma 13.

We have $\text{REC}_{\Gamma,\Delta,\tau,\{\tau\}}^A((\cdot), \varphi, (\cdot), \emptyset) \subseteq \mathcal{H}$ containing the clause $Q_\tau(A_{(n)}(X_1, \dots, X_n)) \leftarrow \vec{H}$ such that for $1 \leq i \leq n$ the following properties hold.

- If T_i is a literal, then $Q_t(X_i), (X_i = T_i) \in \vec{H}$, introduced by case $\langle\langle \alpha : t \rangle\rangle \Rightarrow \psi$ such that $\theta(\alpha) = T_i$ and $(T_i : t) \in \Delta$. Additionally, \mathcal{H} contains the clause $Q_t(T_i) \leftarrow$.
- If T_i is not a literal, then $Q_\sigma(X_i) \in \vec{H}$ for some type σ such that $\Gamma; \Delta \vdash T_i : \sigma$ by either Lemma 13.1.b or Lemma 13.2.a, and by the induction hypothesis $\mathcal{H} \Vdash Q_\sigma(\text{tree}(T_i))$.
- If a literal constraint P occurs in φ , then $\theta(P)$ holds by Lemma 13.1.c.
- If a term constraint P occurs in φ , then $\mu(\theta(P))$ occurs in \vec{H} and $\xi(\theta(P))$ holds by Lemma 13.1.c.

Using the substitution which maps X_i to $\text{tree}(T_i)$ for $i = 1, \dots, n$ each antecedent in \vec{H} is true in the considered smallest Herbrand model, and we obtain $\mathcal{H} \Vdash Q_\tau(\text{tree}(AT_1 \dots T_n))$.

For the implication from right to left, we assume $\mathcal{H} \Vdash Q_\tau(\text{tree}(M))$ and proceed by induction on M . We have $M = AT_1 \dots T_n$ and \mathcal{H} contains the clause $Q_\tau(A_{(n)}(X_1, \dots, X_n)) \leftarrow \vec{H}$, constructed by $\text{REC}_{\Gamma,\Delta,\tau,\Xi}^A$ for some set Ξ . Additionally, for some literal substitution θ and the substitution which maps X_i to $\text{tree}(T_i)$ for $i = 1, \dots, n$ each antecedent in \vec{H} is true in the considered smallest Herbrand model, and the following properties hold.

- If T_i is a literal, then $(T_i : t) \in \Delta$ for some t . Additionally, $(X_i = T_i) \in \vec{H}$, introduced by case $\langle\langle \alpha : t \rangle\rangle \Rightarrow \psi$ such that $\theta(\alpha) = T_i$.
- If T_i is not a literal, then $Q_\sigma(X_i) \in \vec{H}$ for some type σ and $\mathcal{H} \Vdash Q_\sigma(\text{tree}(T_i))$. By the induction hypothesis we have $\Gamma; \Delta \vdash T_i : \sigma$.
- For any literal constraint P occurring in φ we have that $\theta(P)$ holds.
- Let ξ be a term substitution such that $\xi(x) = T_i$ if $\mu(x) = X_i$. For any term constraint P occurring in φ , we have $\mu(\theta(P)) \in \vec{H}$ and $\xi(\theta(P))$ holds.

By Lemma 13 we obtain $\Gamma; \Delta \vdash AT_1 \dots T_n : \tau$. ◀

The following Definition 39 gives the tree language over a signature $\Sigma(\Gamma, \Delta, \Xi)$ for an intersection type $\tau \in \Xi$ described by a set of Horn clauses constructed in Algorithm $\text{INH}_{\Gamma, \Delta}(\tau, \emptyset)$.

► **Definition 39.** *Let \mathcal{H} be a set of Horn clauses over the signature $\Sigma(\Gamma, \Delta, \Xi)$ and let $\tau \in \Xi$, we call $\mathcal{L}_{\mathcal{H}}(\tau) = \{\text{tree}(T) \mid T \in \mathbb{C}(\Gamma, \Delta) \cup \text{dom}(\Delta) \text{ such that } \mathcal{H} \Vdash Q_{\tau}(\text{tree}(T))\}$ the tree language of τ in \mathcal{H} .*

We recall the shape of *automata clauses* by Reuß and Seidl in the following Definition 40, for which emptiness of the corresponding tree language is decidable (Theorem 41).

► **Definition 40 (Automata Clauses [33, Section 2]).** *An automata clause over the signature $\Sigma(\Gamma, \Delta, \Xi)$ is a Horn clause of the form*

$$Q_0(A_{(n)}(X_1, \dots, X_n)) \leftarrow Q_1(X_1), \dots, Q_n(X_n), X_{i_1} = u_1, \dots, X_{i_k} = u_k, X_{j_1} \neq v_1, \dots, X_{j_m} \neq v_m$$

where $A_{(n)} \in \Sigma(\Gamma, \Delta, \Xi)$ is an n -ary function symbol, $Q_0, \dots, Q_n \in \Sigma(\Gamma, \Delta, \Xi)$ are unary predicates, X_1, \dots, X_n are distinct first-order variables, $u_1, \dots, u_k, v_1, \dots, v_m$ are trees over $\Sigma(\Gamma, \Delta, \Xi)$ containing variables from $\{X_1, \dots, X_n\}$, and $i_1, \dots, i_k, j_1, \dots, j_m \in \{1, \dots, n\}$.

The tree language $\mathcal{L}_{\mathcal{H}}(\tau)$ corresponds to the language of a bottom-up tree automaton with term constraints [33] described by automata clauses \mathcal{H} and having the accepting states $\{Q_{\tau}\}$. Therefore, emptiness of $\mathcal{L}_{\mathcal{H}}(\tau)$ is decidable.

► **Theorem 41 ([33, Theorem 14]).** *Given a set \mathcal{H} of automata clauses over the signature $\Sigma(\Gamma, \Delta, \Xi)$ and $\tau \in \Xi$, emptiness of the tree language $\mathcal{L}_{\mathcal{H}}(\tau)$ is decidable.*

Finally, we show decidability of intersection type inhabitation with literal and term constraints by reduction to emptiness of bottom-up tree automata with term constraints.

► **Theorem 42.** *Intersection type inhabitation with literal and term constraints (Problem 31) is decidable.*

Proof. Due to Theorem 38 and Theorem 41, it suffices to show that the set of Horn clauses $\mathcal{H} = \text{INH}_{\Gamma, \Delta}(\tau, \emptyset)$ over the signature $\Sigma(\Gamma, \Delta, \Xi)$ for some set Ξ of intersection types contains only automata clauses.

Heads of clauses in \mathcal{H} are either $Q_t(l)$ for some $(l : t) \in \Delta$ (constructed in case $\langle \alpha : t \rangle \Rightarrow \psi$) or $Q_{\tau}(A_{(n)}(\vec{X}, Y_1, \dots, Y_k))$ where n is the length of the list $(\vec{X}, Y_1, \dots, Y_k)$ (constructed in the intersection type case), which are both of proper shape. It remains to show that any antecedent in clauses in \mathcal{H} is of proper shape. We consider the individual cases in which antecedents are constructed in Algorithm REC.

Case $\langle \alpha : t \rangle \Rightarrow \psi$: The constructed antecedents are $Q_t(Y)$ and $Y = l$ for some fresh first-order variable Y and $(l : t) \in \Delta$.

Case $\langle\langle x : \sigma \rangle\rangle \Rightarrow \psi$: The constructed antecedent is $Q_{\sigma}(Y)$ for some fresh first-order variable Y and $\sigma \in \Xi$.

Case $P \Rightarrow \psi$ such that P is closed: No antecedents are constructed.

Case $(X = M) \Rightarrow \psi$ where M may contain free variables: The constructed antecedent is $(X = \text{tree}(M))$. Since Γ contains only closed parameterized types, any literal variable in M is substituted by some literal in $\text{dom}(\Delta)$ and any term variable in M is substituted by some first-order variable. Therefore, $(X = \text{tree}(M))$ is of proper shape.

Case $(X \neq M) \Rightarrow \psi$ where M may contain free variables: The constructed antecedent is $(X \neq \text{tree}(M))$, which analogously to the above case is of proper shape.

Case ρ : The constructed antecedents are $Q_{\sigma_1}(Y_1), \dots, Q_{\sigma_k}(Y_k)$ for some fresh first-order variables Y_1, \dots, Y_k and $\sigma_1, \dots, \sigma_k \in \Xi$.

Finally, we need to ensure that each first-order variable Z occurring in the head of the constructed clause $Q_\tau(A_{(n)}(\vec{X}, Y_1, \dots, Y_k)) \leftarrow \vec{H}, Q_{\sigma_1}(Y_1), \dots, Q_{\sigma_k}(Y_k)$ in the last case occurs in exactly one antecedent $Q_\sigma(Z)$ for some $\sigma \in \Xi$ or $Q_t(Z)$ for some $t \in \text{ran}(\Delta)$. This trivially holds for the above fresh first-order variables Y_1, \dots, Y_k . The remaining first-order variables $Y \in \vec{X}$ are introduced in case $\langle \alpha : t \rangle \Rightarrow \psi$ (with the corresponding antecedent $Q_t(Y)$) and in case $\langle\langle x : \sigma \rangle\rangle \Rightarrow \psi$ (with the corresponding antecedent $Q_\sigma(Y)$). ◀

Concluding the presentation of the decidable fragment of FCLP, we give remarks on its complexity bounds (Remark 43), extensions (Remark 44), and alternatives (Remark 45).

► **Remark 43.** Complexity bounds for emptiness of bottom-up tree automata with term constraints are not known [33, Section 6]. Therefore, we cannot give complexity bounds for intersection type inhabitation with literal and term constraints. Additionally, we do not impose complexity bounds on predicate evaluation (besides decidability).

► **Remark 44.** The class of bottom-up tree automata with term constraints is closed under Boolean operations [33, Proposition 6]. Therefore, existing techniques extending $\text{FCL}(\cap, \leq)$ by a Boolean query language [20] are applicable.

► **Remark 45.** There are other classes of constrained tree automata [26] which could be used to obtain a decidable fragment of FCLP. One example are generalized encompassment automata [11, Definition 1], which are more expressive than bottom-up tree automata with term constraints. However, the presentation of such automata as sets of Horn clauses resulting from an inhabitation algorithm appears challenging.

4 Implementation

In practical applications one is rarely interested in solving just the inhabitation *decision problem*, but rather in computing one, several, or all inhabitants. Therefore, we consider the following *synthesis problem*, which is a slightly modified version of Problem 17.

► **Problem 46 (Synthesis).** Given a type environment Γ , a literal environment Δ , and an intersection type τ , enumerate combinatory terms M for which $\Gamma; \Delta \vdash M : \tau$ holds.

Furthermore, when solving particular problems via synthesis, we want to *interpret* the resulting combinatory terms as solutions for those problems.

A framework [21] addressing Problem 46 was implemented on the basis of the existing Combinatory Logic Synthesizer (CLS)⁴ using the Python programming language.

In this section we discuss the implementation, and evaluate how the added features of FCLP help modeling, and improve performance compared to $\text{FCL}(\cap, \leq)$, using maze solving (Problem 52) as a benchmark example.

► **Remark 47.** The presented framework does not make use of Python’s built-in type system for synthesis, and implements types as Python classes.

4.1 Usage

The framework is implemented as a Python library, requiring Python version 3.10 or later. It does not rely on any additional libraries. For the sake of brevity, an embedded domain-specific language (eDSL), shown in Figure 1, was created for writing parameterized types.

⁴ <https://github.com/cls-python/cls-python>

Constructor	Python	Constructor	Python
ω	<code>Omega()</code>	$\langle x : t \rangle$	<code>Use('x', 't')</code>
$\sigma \rightarrow \tau$	<code>σ ** τ</code>	$\langle\langle \alpha : \tau \rangle\rangle$	<code>Use('alpha', tau)</code>
$\sigma \cap \tau$	<code>σ & τ</code>	Predicate P using variables v_0, \dots, v_n	<code>With(lambda v0, ..., vn: P)</code>
$c(\sigma)$	<code>'c'@sigma</code>	$\dots \Rightarrow \tau$	<code>DSL().In(tau)</code>
Literal $l : t$	<code>Literal(l,t)</code>		
Variable α	<code>LVar('alpha')</code>		

■ **Figure 1** Embedded DSL for parameterized and intersection types in Python.

► **Example 48 (eDSL).** The parameterized type $\langle \alpha : \text{int} \rangle \Rightarrow \langle x : \sigma \rangle \Rightarrow \langle y : \sigma \rangle \Rightarrow (x = y) \Rightarrow \langle \beta : \text{int} \rangle \Rightarrow (\beta = \alpha + 1) \Rightarrow c(\alpha) \rightarrow c(\beta) \rightarrow (c(4) \cap c(\omega))$ corresponds to the following eDSL term:

```
DSL().Use('alpha', 'int').Use('x', sigma).Use('y', sigma)
    .With(lambda x y: x == y).Use(beta, 'int').With(lambda alpha beta: beta = alpha + 1)
    .In(('c'@LVar(alpha)) ** ('c'@LVar(beta)) **
        ('c'@Literal(4, 'int') & 'c'@(Omega())))
```

► **Remark 49.** The operator `**` was chosen to represent the arrow type constructor, since it is the only right associative operator available in Python.

In order to synthesize inhabitants, we need to define a type environment, a literal environment, and an intersection type as a *query*. A type environment is a `dict`, mapping combinators to their types, where combinators can be any `Hashable Python object`. Types can be formed via the eDSL or by instantiating appropriate subclasses of the class `Type`. A literal environment is a `dict`, mapping collection identifiers (represented as `str`) to literals, which can be any `Python objects`. The three main operations of the framework are:

- `FiniteCombinatoryLogic(...).inhabit(...)` to initialize the synthesis procedure and compute an intermediate result representation,
- `enumerate_terms` to extract combinatory terms from the intermediate representation,
- `interpret_term` to interpret a combinatory term as a solution in the problem domain.

Given a type environment Γ , literal environment Δ , and a type τ , we can use the above operations to enumerate elements of the set $\{M \mid \Gamma, \Delta \vdash M : \tau\}$ by the following steps. First, we generate an intermediate representation of the synthesis results.

```
results = FiniteCombinatoryLogic(repository = Gamma, literals = Delta).inhabit(tau)
```

Second, we enumerate up to n distinct terms.

```
terms = enumerate_terms(tau, results, n)
```

A term is represented as a tuple, such that its first projection is the associated combinator, and the following projections are representations of the arguments. Finally, we interpret these terms to obtain solutions in the problem domain.

```
solutions = [interpret_term(term) for term in terms]
```

Each combinator can be equipped with a computational component, realized by implementing the `Callable` protocol. In this step each callable combinator is interpreted by calling it on its interpreted arguments.

4.2 Synthesis Procedure

In contrast to Algorithm INH (Definition 34), the implemented synthesis procedure is not limited to the decidable fragment. While this makes inhabitation undecidable, in most practical applications the advantages of unrestricted predicates outweigh potential lock-ups.

Given a type environment Γ , literal environment Δ , and intersection type τ , the synthesis process is structured as follows.

Preprocessing: For each typed combinator $(C : e_1 \Rightarrow \dots \Rightarrow e_m \Rightarrow \rho) \in \Gamma$, we first generate the set of substitutions determined by Δ and the literal quantifiers in e_1, \dots, e_m . Next, we use these substitutions to evaluate all literal constraints in e_1, \dots, e_m , discarding substitutions which violate at least one constraint. The remaining substitutions are stored alongside the combinator. Afterward, we remove all literal quantifiers and literal constraints from e_1, \dots, e_m . Finally, for each arity k up to $\text{ar}(\rho)$, we decompose ρ into possible pairs of k argument types and a return type.

Generating Horn clauses: For each typed combinator $(C : e_1 \Rightarrow \dots \Rightarrow e_m \Rightarrow \rho) \in \Gamma$, each literal substitution θ for C , and each arity k , if the intersection of the return types is a subtype of τ , a Horn clause is created analogous to the last case of Algorithm INH. Further Horn clauses are generated by recursion on each argument type, as well as on each type occurring in term quantifiers in e_1, \dots, e_m .

Enumeration: Given the above set of Horn clauses and a number n , we enumerate up to n inhabitants in a *bottom-up* manner. In this step we resolve term quantifiers by enumerating inhabitants of the type quantified over, placing them at the respective argument position in a given combinator and substituting the respective term variables in the remaining constraints by those inhabitants. If this leads to violated constraints, we discard those terms. Similarly, literals are placed at positions corresponding to their quantifier position.

4.3 Solutions in a Maze

The additions of FCLP compared to $\text{FCL}(\cap, \leq)$ improve upon the expressiveness of specification, and it was observed that more concise modeling can lead to performance improvements. Consider the following example of finding solutions in a maze.

► **Definition 50 (Maze).** Let $n \in \mathbb{N}$, an $n \times n$ -maze is a function $\mathcal{M} : \{0, \dots, n-1\}^2 \rightarrow \mathbb{B}$ indicating whether a position is free or blocked.

► **Definition 51 (Maze Solution).** A solution to an $n \times n$ -maze \mathcal{M} is a finite sequence $((x_0, y_0), \dots, (x_l, y_l))$ such that:

- $(x_0, y_0) = (0, 0)$ and $(x_l, y_l) = (n-1, n-1)$,
- for each $i \in \{0, \dots, l\}$ we have $(x_i, y_i) \in \text{dom}(\mathcal{M})$ and $\mathcal{M}(x_i, y_i) = \text{true}$,
- for each $i \in \{0, \dots, l-1\}$ we have $|x_i - x_{i+1}| + |y_i - y_{i+1}| = 1$.

► **Problem 52 (Maze Solving).** Given an $n \times n$ -maze \mathcal{M} , enumerate solutions to \mathcal{M} .

Variants of maze solving are common, miniature benchmark examples⁵ for component-oriented synthesis [9, 4, 19]. Domain-specific components for maze solving encompass movement directions and the maze layout. Specification capabilities, scalability, and performance of the framework in the case of maze solving translate well to software product line design [25], factory planning [37], and cyber-physical system design [14].

⁵ Of course, if one is solely interested in maze solving, a domain-specific algorithm using dynamic programming is recommended instead of domain-agnostic component-oriented synthesis.

2:16 Finite Combinatory Logic with Predicates

Let us explore an approach to maze solving in $\text{FCL}(\cap, \leq)$. Given an $n \times n$ -maze \mathcal{M} we construct the following type environment $\Gamma_{\text{FCL}}^{\mathcal{M}}$.

$$\begin{aligned} \Gamma_{\text{FCL}}^{\mathcal{M}} = & \{\text{FREE}_{x,y} : \text{isfree}(x, y) \mid (x, y) \in \text{dom}(\mathcal{M}) \text{ such that } \mathcal{M}(x, y) = \text{true}\} \cup \\ & \{\text{START} : \text{pos}(0(\omega), 0(\omega)), \\ & \text{UP} : \bigcap_{(x,y) \in \text{dom}(\mathcal{M})} (\text{isfree}(x, y) \rightarrow \text{pos}(x, y+1) \rightarrow \text{pos}(x, y)), \\ & \text{DOWN} : \bigcap_{(x,y) \in \text{dom}(\mathcal{M})} (\text{isfree}(x, y) \rightarrow \text{pos}(x, y-1) \rightarrow \text{pos}(x, y)), \\ & \text{LEFT} : \bigcap_{(x,y) \in \text{dom}(\mathcal{M})} (\text{isfree}(x, y) \rightarrow \text{pos}(x+1, y) \rightarrow \text{pos}(x, y)), \\ & \text{RIGHT} : \bigcap_{(x,y) \in \text{dom}(\mathcal{M})} (\text{isfree}(x, y) \rightarrow \text{pos}(x-1, y) \rightarrow \text{pos}(x, y))\} \end{aligned}$$

For better legibility, we allow for binary constructors as described in Remark 3. Combinators $\text{FREE}_{x,y}$ denote witnesses that the space at coordinates x and y is free. Combinators UP, DOWN, LEFT and RIGHT denote movement in the corresponding direction. Since $\text{FCL}(\cap, \leq)$ does not allow for literals, numbers need to be encoded by constructors $(0(\omega), 1(\omega), \dots, n(\omega))$ and the position shifts need to be computed beforehand.

► **Example 53.** Consider a 2×2 maze \mathcal{M} in which exactly the position $(1, 0)$ is blocked. In order to synthesize (not necessarily loop-free) solutions in \mathcal{M} , we enumerate elements of the set $\{M \mid \Gamma_{\text{FCL}}^{\mathcal{M}} \vdash M : \text{pos}(1(\omega), 1(\omega))\}$, resulting in combinatory terms such as: $(\text{RIGHT FREE}_{1,1} (\text{DOWN FREE}_{0,1} \text{START}))$. Given the appropriate interpretation for the movement combinators, we can interpret the term as the solution $((0, 0), (0, 1), (1, 1))$, shown in the below Figure 2.



■ **Figure 2** A 2×2 maze with a solution in red. Position $(0, 0)$ is in the top-left corner.

While the above shows that $\text{FCL}(\cap, \leq)$ can model Problem 52, we identify three improvements that can be made using FCLP.

- Quantifiers can be used to avoid intersections spanning all positions in the maze.
- Since we expect each combinator to be fully applied (have exactly as many arguments as the arity of its type), we model positions as parameters.
- The combinators $\text{FREE}_{x,y}$ act as predicates for the movement combinators. Using FCLP, we can use predicates directly, removing those combinators.

Applying the above improvements, we construct the type environment $\Gamma_{\text{FCLP}}^{\mathcal{M}}$ together with the literal environment $\Delta = \{0 : \text{int}, \dots, n-1 : \text{int}\}$.

$$\begin{aligned} \Gamma_{\text{FCLP}}^{\mathcal{M}} = & \{\text{START} : \text{pos}(0, 0), \\ \text{UP} : & \langle \alpha : \text{int} \rangle \Rightarrow \langle \beta : \text{int} \rangle \Rightarrow \langle \gamma : \text{int} \rangle \Rightarrow (\gamma = \beta + 1) \Rightarrow \mathcal{M}(\alpha, \beta) \Rightarrow \langle \text{p} : \text{pos}(\alpha, \gamma) \rangle \Rightarrow \text{pos}(\alpha, \beta), \\ \text{DOWN} : & \langle \alpha : \text{int} \rangle \Rightarrow \langle \beta : \text{int} \rangle \Rightarrow \langle \gamma : \text{int} \rangle \Rightarrow (\gamma = \beta - 1) \Rightarrow \mathcal{M}(\alpha, \beta) \Rightarrow \langle \text{p} : \text{pos}(\alpha, \gamma) \rangle \Rightarrow \text{pos}(\alpha, \beta), \\ \text{LEFT} : & \langle \alpha : \text{int} \rangle \Rightarrow \langle \beta : \text{int} \rangle \Rightarrow \langle \gamma : \text{int} \rangle \Rightarrow (\gamma = \alpha + 1) \Rightarrow \mathcal{M}(\alpha, \beta) \Rightarrow \langle \text{p} : \text{pos}(\gamma, \beta) \rangle \Rightarrow \text{pos}(\alpha, \beta), \\ \text{RIGHT} : & \langle \alpha : \text{int} \rangle \Rightarrow \langle \beta : \text{int} \rangle \Rightarrow \langle \gamma : \text{int} \rangle \Rightarrow (\gamma = \alpha - 1) \Rightarrow \mathcal{M}(\alpha, \beta) \Rightarrow \langle \text{p} : \text{pos}(\gamma, \beta) \rangle \Rightarrow \text{pos}(\alpha, \beta) \} \end{aligned}$$

Clearly, the above type environment $\Gamma_{\text{FCLP}}^{\mathcal{M}}$ is more concise compared to the type environment $\Gamma_{\text{FCL}}^{\mathcal{M}}$. Additionally, we do not need to manually compute the position shifts to construct $\Gamma_{\text{FCLP}}^{\mathcal{M}}$ beforehand, as was needed for $\Gamma_{\text{FCL}}^{\mathcal{M}}$.

Performance Evaluation

Using the modeling techniques introduced with FCLP, we observe improved performance. Figure 3 shows synthesis execution time to find all solutions using different type environments, up to a maze size of 70×70 . All benchmarks were performed on the same machine⁶ using the implementation at hand. Environments $\Gamma_{\text{FCLP(lit)}}^{\mathcal{M}}$, $\Gamma_{\text{FCLP(pos)}}^{\mathcal{M}}$, $\Gamma_{\text{FCLP(pred)}}^{\mathcal{M}}$ each correspond to an improvement identified above, namely using literals for coordinates, using term quantifiers for the position, and using a predicate for free positions respectively.

Size	$\Gamma_{\text{FCL}}^{\mathcal{M}}$	$\Gamma_{\text{FCLP(lit)}}^{\mathcal{M}}$	$\Gamma_{\text{FCLP(pos)}}^{\mathcal{M}}$	$\Gamma_{\text{FCLP(pred)}}^{\mathcal{M}}$	$\Gamma_{\text{FCLP}}^{\mathcal{M}}$
10×10	1.3 s	0.5 s	0.3 s	0.1 s	0.1 s
20×20	21.9 s	8.0 s	6.4 s	2.4 s	1.9 s
30×30	125.2 s	41.0 s	30.7 s	12.8 s	9.8 s
40×40	464.7 s	130.2 s	97.9 s	42.3 s	32.4 s
50×50	1279.8 s	322.2 s	239.5 s	103.2 s	78.5 s
60×60	3038.5 s	645.4 s	486.3 s	214.2 s	160.2 s
70×70	> 5000	1195.6 s	893.5 s	384.9 s	299.4 s

■ **Figure 3** Benchmarks for different maze sizes and different type environments.

While the data shows that all approaches scale at an exponential rate given the size, using $\Gamma_{\text{FCLP}}^{\mathcal{M}}$ leads to a performance increase of one order of magnitude compared to $\Gamma_{\text{FCL}}^{\mathcal{M}}$, with each modeling technique contributing to the speed-up. The performance increase of $\Gamma_{\text{FCLP}}^{\mathcal{M}}$ compared to $\Gamma_{\text{FCL}}^{\mathcal{M}}$ can be attributed to the following three factors:

1. $\Gamma_{\text{FCLP(lit)}}^{\mathcal{M}}$ reduces the size of the type of each movement combinator.
2. $\Gamma_{\text{FCLP(pos)}}^{\mathcal{M}}$ induces fewer subtype checks due to restricted term shape.
3. $\Gamma_{\text{FCLP(pred)}}^{\mathcal{M}}$ reduces the number of combinators.

A benchmark using $\Gamma_{\text{FCL}}^{\mathcal{M}}$ and the latest version of the prior implementation of CLS in Python was conducted, leading to a time of 276 s for a 10×10 maze. The difference to the prior implementation stems from the fact that the implementation at hand focuses on performance, while the previous focuses on formal verification [4].

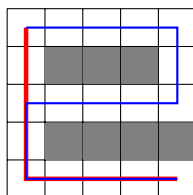
For performance evaluation, frameworks based on Bounded Combinatory Logic [18] or based on Finite Combinatory Logic with Boolean Queries [20] are of no consequence. Neither bounded polymorphism nor Boolean connectives are suited to model maze solving. Therefore, the resulting performance is close to the prior implementation of CLS.

Loop-free solutions

As observed above, utilizing quantifiers and predicates can lead to significant speed-ups in certain use-cases. Interestingly, predicates also model specifications, for which an effective model in $\text{FCL}(\cap, \leq)$ is unclear.

Consider the maze presented in Figure 4 and solutions, that do not visit any position more than once (*loop-free*). In each movement combinator a predicate can require each visited position to be unique in a given term, thereby only allowing for loop-free solutions. During enumeration, terms containing at least one loop are discarded and the procedure will return

⁶ AMD Ryzen 7 5800X (3.8 GHz), 16 GB DDR4 RAM



■ **Figure 4** A 5×5 maze with exactly two loop-free solutions (in red and blue).

exactly two solutions, after which it halts. In contrast, for $\text{FCL}(\cap, \leq)$ we need to rely on a generate-and-test approach. Such an approach would enumerate infinitely many candidates (including those, which contain arbitrary many loops) and filter out those containing loops. In particular, the procedure as a whole would search indefinitely for a third loop-free solution. Furthermore, adjusting the size of the maze in Figure 4, there are arbitrary many looping solutions whose length lie between the two loop-free solutions. The FCLP approach discards solutions containing at least one loop early, and thereby never considers solutions with multiple loops. In comparison, the $\text{FCL}(\cap, \leq)$ approach has no such mechanism, leading to exponentially more candidates to be checked and therefor an arbitrary long time between the two loop-free solutions.

It is possible to model loop-free solutions in the synthesis framework based on Finite Combinatory Logic with Boolean Queries [20]. In particular, negation is suitable to express that a position is not yet visited. However, a performance evaluation has shown that such an approach is infeasible for mazes beyond size 5×5 .

5 Case Study: Robotic Arms

We evaluate practical applicability of FCLP by means of a case study in which robotic arms are synthesized from a set of 28 modular components [15]. The individual components are modeled as typed combinators such that inhabitants of specific types can be interpreted as assembly instructions for robotic arms. The assembly instructions are executed in CAD software, assembling 3D models of robotic arms. Analysis tools, which are part of the CAD software, confirm that the assembled robotic arms are mechanically sound, individual components do not interfere with each other, and mechanical joints kinematically work as intended. These properties hold for all robotic arms up to six degrees of freedom (six moving joints) synthesized in the case study, a total of 364 arms containing on average 140 components each. Due to the chain-like nature of robotic arms we argue that higher degrees of freedom also exhibit these properties.

In previous work [14] conducting the same case study utilizing $\text{FCL}(\cap, \leq)$, numerical constraints necessitate an exponential number of combinators. There are a number of common numerical constraints of high importance, such as the degrees of freedom, the total drawn current, total weight of the assembly, or the torque of motors. In previous work such constraints are modeled as families of the following typed combinators, which specify individual cases.

$$\begin{aligned}
 C_{i_1, i_2, i_3, i_4} &: \text{Assembly}(i_1(\omega)) \rightarrow \text{Assembly}(i_2(\omega)) \rightarrow \text{Assembly}(i_3(\omega)) \rightarrow \text{Assembly}(i_4(\omega)) \\
 &\text{such that } 0, \dots, n \text{ are unary type constructors,} \\
 &i_1, i_2, i_3, i_4 \in \{0, \dots, n\}, \\
 &\text{and } i_4 = i_1 + i_2 + i_3 + k
 \end{aligned}$$

Each of the above typed combinators C_{i_1, i_2, i_3, i_4} refers to individual numbers i_1, i_2, i_3 of specific parts in each connected assembly, and the accumulated number i_4 of specific parts increased by a constant k . The number of such combinators is exponential in the number of connected assemblies (the arity of the type) multiplied by the number of distinct constraints of interest. The number of connected assemblies depends on the granularity of the model, but even for a case study of this scale a typical number is five, and requests usually employ at least three constraints. The value range n for the constraints is usually no more than ten. This leads to repositories of enormous size, containing tens of thousands of combinators for typical requests. In practice, this large number of essentially redundant combinators impairs debugging and deteriorates performance of the inhabitation algorithm.

In the later case study [15] the described issues are tackled using FCLP. The above family of typed combinators is condensed to the following single typed combinator:

$$C : \langle \alpha_1 : \text{int} \rangle \Rightarrow \langle \alpha_2 : \text{int} \rangle \Rightarrow \langle \alpha_3 : \text{int} \rangle \Rightarrow \langle \alpha_4 : \text{int} \rangle \Rightarrow (\alpha_4 = \alpha_1 + \alpha_2 + \alpha_3 + k) \Rightarrow \\ \langle \langle x_1 : \text{Assembly}(\alpha_1) \rangle \rangle \Rightarrow \langle \langle x_2 : \text{Assembly}(\alpha_2) \rangle \rangle \Rightarrow \langle \langle x_3 : \text{Assembly}(\alpha_3) \rangle \rangle \Rightarrow \text{Assembly}(\alpha_4) \\ \text{with } \Delta \supseteq \{0 : \text{int}, \dots, n : \text{int}\}$$

The above combinator C concisely expresses the described numeric constraint for the particular assembly. Literal variables $\alpha_1, \alpha_2, \alpha_3$ refer to individual numbers of specific parts in each connected assembly. The literal variable α_4 refers to the accumulated number of specific parts increased by a constant k , which is described by the literal constraint $\alpha_4 = \alpha_1 + \alpha_2 + \alpha_3 + k$. As a result, such combinators closely represent the individual component in the actual use-case [15]. The number of required combinators per component is constant, and independent from specified constraints. Since each predicate is a literal constraint (Definition 30), the corresponding inhabitation problem is decidable (Theorem 42).

The ability to cleanly handle constraints allows leveraging combinatory logic synthesis to explore the robotic arm design space efficiently and glean information about it.

6 Conclusion

The present work conservatively extends the type system $\text{FCL}(\cap, \leq)$ [32] by literals, quantifiers, and predicates. While the inhabitation problem in the resulting type system FCLP (Definition 8) is undecidable (Theorem 20), we give an expressive fragment of FCLP for which inhabitation is decidable (Theorem 42). The particular fragment is based on results for tree automata with term constraints by Reuß and Seidl [33], and allows for specification of certain local (dis)equality constraints (Definition 29) for subterms of inhabitants. The main contribution of the present work is a terminating algorithm INH which given a type environment, a literal environment, and an intersection type computes a logic program (set of Horn clauses) which represents all inhabitants.

For empirical evaluation, an algorithm for inhabitant enumeration (Problem 46) is implemented in the programming language Python. The implementation, as part of a larger synthesis framework [21], is shown superior to an existing $\text{FCL}(\cap, \leq)$ -based framework CLS [4] with respect to specification capabilities, scalability, and performance. Finally, practical applicability is demonstrated via a case study in the area of cyber-physical systems.

There are several directions for further research.

First, it is worth investigating type inhabitation for more expressive type languages in the setting of combinatory logic. Polymorphic set-theoretic types [12, 13] constitute a promising candidate for type-based component-oriented program synthesis.

Second, there is room for exploration of more expressive fragments of FCLP with decidable type inhabitation. A promising candidate could be obtained by using generalized encompassment automata [11, Definition 1] instead of (the less expressive) bottom-up tree

automata with term constraints. Another candidate could rely on automata with disequality constraints [16, Definition 1]. Such automata are used by Czajka et.al. [5] to externally restrict sets of inhabitants via term rewriting systems. It appears appealing to internalize such restrictions as part of the specification language.

Third, the present work focuses on inhabitants which respect arities in the given type environment. This restriction is based on the observation that in practice not every domain-specific component is a function. However, the subtyping rule $\omega \leq \omega \rightarrow \omega$ is in conflict with this observation. It is intriguing to explore semantics of combinatory logic with intersection types [17] without the subtyping rule $\omega \leq \omega \rightarrow \omega$.

Fourth, satisfiability of literal constraints, such as $\beta = \alpha + 1$, could be addressed in algorithm INH by a principled approach, for example based on SMT. Besides potential performance improvements, such an approach may allow for a countably infinite parameter space (literal environment).

Fifth, efficient enumeration procedures [35, 22] for tree languages focus on regular structures. Besides the naive generate-and-test approach in the present work, there is no practical enumeration procedure for trees accepted by bottom-up tree automata with term constraints. It is unclear whether methods known from logic programming, such as sideways information passing [3], are applicable.

References

- 1 Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991. doi:10.1017/S0956796800020025.
- 2 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983. URL: <http://www.jstor.org/stable/2273659>.
- 3 Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Log. Program.*, 10(3&4):255–299, 1991. doi:10.1016/0743-1066(91)90038-Q.
- 4 Jan Bessai. *A type-theoretic framework for software component synthesis*. PhD thesis, TU Dortmund University, 2019.
- 5 Jan Bessai, Lukasz Czajka, Felix Laarmann, and Jakob Rehof. Restricting tree grammars with term rewriting. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, volume 228 of *LIPICs*, pages 14:1–14:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.FSCD.2022.14.
- 6 Jan Bessai, Boris Döder, George Heineman, and Jakob Rehof. Combinatory synthesis of classes using feature grammars. In *Formal Aspects of Component Software: 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers 12*, pages 123–140. Springer, 2016.
- 7 Jan Bessai, Andrej Dudenhefner, Boris Döder, Tzu-Chun Chen, Ugo de’Liguoro, and Jakob Rehof. Mixin composition synthesis based on intersection types. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPICs)*, pages 76–91, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICs.TLCA.2015.76.
- 8 Jan Bessai, Andrej Dudenhefner, Boris Döder, Moritz Martens, and Jakob Rehof. Combinatory Logic Synthesizer. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2014. doi:10.1007/978-3-662-45234-9_3.

- 9 Jan Bessai and Anna Vasileva. User support for the combinator logic synthesizer framework. In Paolo Masci, Rosemary Monahan, and Virgile Prevosto, editors, *Proceedings 4th Workshop on Formal Integrated Development Environment, F-IDE@FLoC 2018, Oxford, England, 14 July 2018*, volume 284 of *EPTCS*, pages 16–25, 2018. doi:10.4204/EPTCS.284.2.
- 10 Bruno Bogaert, Franck Seynhaeve, and Sophie Tison. The recognizability problem for tree automata with comparisons between brothers. In Wolfgang Thomas, editor, *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1578 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 1999. doi:10.1007/3-540-49019-1_11.
- 11 Anne-Cécile Caron, Hubert Comon, Jean-Luc Coquidé, Max Dauchet, and Florent Jacquemard. Pumping, cleaning and symbolic constraints solving. In Serge Abiteboul and Eli Shamir, editors, *Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings*, volume 820 of *Lecture Notes in Computer Science*, pages 436–449. Springer, 1994. doi:10.1007/3-540-58201-0_88.
- 12 Giuseppe Castagna. Programming with union, intersection, and negation types. *CoRR*, abs/2111.03354, 2021. arXiv:2111.03354.
- 13 Giuseppe Castagna, Mickaël Laurent, and Kim Nguyen. Polymorphic type inference for dynamic languages. *Proc. ACM Program. Lang.*, 8(POPL):1179–1210, 2024. doi:10.1145/3632882.
- 14 Constantin Chaumet and Jakob Rehof. CLS-CAD: Synthesizing CAD Assemblies in Fusion 360, 2023. arXiv:2311.18492.
- 15 Constantin Chaumet, Jakob Rehof, and Thomas Schuster. A knowledge-driven framework for synthesizing designs from modular components, 2023. arXiv:2311.18533.
- 16 Hubert Comon and Florent Jacquemard. Ground reducibility is EXPTIME-complete. *Inf. Comput.*, 187(1):123–153, 2003. doi:10.1016/S0890-5401(03)00134-2.
- 17 Mariangiola Dezani-Ciancaglini and J. Roger Hindley. Intersection types for combinatory logic. *Theor. Comput. Sci.*, 100(2):303–324, 1992. doi:10.1016/0304-3975(92)90306-Z.
- 18 Boris Döder, Moritz Martens, Jakob Rehof, and Paweł Urzyczyn. Bounded combinatory logic. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPICs*, pages 243–258. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICs.CSL.2012.243.
- 19 Andrej Dudenhefner. *Algorithmic aspects of type-based program synthesis*. PhD thesis, TU Dortmund University, 2019.
- 20 Andrej Dudenhefner, Felix Laarmann, Jakob Rehof, and Christoph Stahl. Finite combinatory logic extended by a Boolean query language for composition synthesis. In *29th International Conference on Types for Proofs and Programs TYPES 2023-Abstracts*, page 105, 2023.
- 21 Andrej Dudenhefner, Christoph Stahl, and Jan Bessai. Combinatory Logic Synthesizer with Predicates. URL: <https://github.com/tudo-seal/clsp-python>.
- 22 Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: functional enumeration of algebraic types. In Janis Voigtländer, editor, *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 61–72. ACM, 2012. doi:10.1145/2364506.2364515.
- 23 Thom W. Frühwirth, Ehud Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 300–309. IEEE Computer Society, 1991. doi:10.1109/LICS.1991.151654.
- 24 George Heineman, Jan Bessai, Boris Döder, and Jakob Rehof. A long and winding road towards modular synthesis. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I 7*, pages 303–317. Springer, 2016.

- 25 George Heineman, Armend Hoxha, Boris Döder, and Jakob Rehof. Towards migrating object-oriented frameworks to enable synthesis of product line members. In *Proceedings of the 19th International Conference on Software Product Line*, pages 56–60, 2015.
- 26 Florent Jacquemard. *Extended Tree Automata Models for the Verification of Infinite State Systems*. PhD thesis, École normale supérieure de Cachan-ENS Cachan, 2011.
- 27 Fadil Kallat, Tristan Schäfer, and Anna Vasileva. CLS-SMT: bringing together combinatory logic synthesis and satisfiability modulo theories. In Giselle Reis and Haniel Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*, volume 301 of *EPTCS*, pages 51–65, 2019. doi:10.4204/EPTCS.301.7.
- 28 Alexander Mages, Carina Mieth, Jens Hetzler, Fadil Kallat, Jakob Rehof, Christian Riest, and Tristan Schäfer. Automatic component-based synthesis of user-configured manufacturing simulation models. In *2022 Winter Simulation Conference (WSC)*, pages 1841–1852. IEEE, 2022.
- 29 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630. ACM, 2015. doi:10.1145/2737924.2738007.
- 30 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 522–538. ACM, 2016. doi:10.1145/2908080.2908093.
- 31 Jakob Rehof. Towards combinatory logic synthesis. In *1st International Workshop on Behavioural Types, BEAT*, 2013.
- 32 Jakob Rehof and Paweł Urzyczyn. Finite combinatory logic with intersection types. In C.-H. Luke Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011. doi:10.1007/978-3-642-21691-6_15.
- 33 Andreas Reuß and Helmut Seidl. Bottom-up tree automata with term constraints. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 581–593. Springer, 2010. doi:10.1007/978-3-642-16242-8_41.
- 34 Tristan Schäfer, Jim Bergmann, Rafael Garcia Carballo, Jakob Rehof, and Petra Wiederkehr. A synthesis-based tool path planning approach for machining operations. *Procedia CIRP*, 104:918–923, 2021.
- 35 Cas van der Rest and Wouter Swierstra. A completely unique account of enumeration. *Proc. ACM Program. Lang.*, 6(ICFP):411–437, 2022. doi:10.1145/3547636.
- 36 Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976. doi:10.1145/321978.321991.
- 37 Jan Winkels, Julian Graefenstein, Tristan Schäfer, David Scholz, Jakob Rehof, and Michael Henke. Automatic composition of rough solution possibilities in the target planning of factory planning projects by means of combinatory logic. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV 8*, pages 487–503. Springer, 2018.

Categorical Models of Subtyping

Greta Coraglia   

Dipartimento di Filosofia (LUCI Lab), Dipartimento di Eccellenza 2023-2027,
Università degli Studi di Milano, Italy

Jacopo Emmenegger   

Dipartimento di Matematica (DIMA), Dipartimento di Eccellenza 2023-2027,
Università degli Studi di Genova, Italy

Abstract

Most categorical models for dependent types have traditionally been heavily *set* based: contexts form a category, and for each we have a set of types in said context – and for each type a set of terms of said type. This is the case for categories with families, categories with attributes, and natural models; in particular, all of them can be traced back to certain discrete Grothendieck fibrations. We extend this intuition to the case of general, not necessarily discrete, fibrations, so that over a given context one has not only a set but a *category* of types.

We argue that the added structure can be attributed to a notion of subtyping that shares many features with that of *coercive* subtyping, in the sense that it is the product of thinking about subtyping as an abbreviation mechanism: we say that a given type A' is a subtype of A if there is a unique coercion from A' to A . Whenever we need a term of type A , then, it suffices to have a term of type A' , which we can “plug-in” into A .

For this version of subtyping we provide rules, coherences, and explicit models, and we compare and contrast it to coercive subtyping as introduced by Z. Luo and others. We conclude by suggesting how the tools we present can be employed in finding appropriate rules relating subtyping and certain type constructors.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Categorical semantics

Keywords and phrases dependent types, subtyping, coercive subtyping, categorical semantics, categories with families, monad

Digital Object Identifier 10.4230/LIPIcs.TYPES.2023.3

Related Version *Previous Version*: <https://arxiv.org/abs/2312.14600>

Funding *Greta Coraglia*: The research has been partially funded by the Project PRIN2020 “BRIO” (2020SSKZ7R) awarded by the Italian Ministry of University and Research.

Jacopo Emmenegger: The research has been partially funded by the the Italian Ministry of University and Research under the scheme “Young Researchers” with agreement number SOE_0000071.

Acknowledgements The first author would like to thank the University of Genoa, where some of this research was completed. We are deeply grateful to Francesco Dagnino, who suggested coercive subtyping in the first place, for his immense clarity of thought. This work benefitted from a lovely visit to LAMA at Université Savoie Mont Blanc in November 2023, and the first author is for it indebted to Tom Hirschowitz and Peio Borthelle, and to the whole group there. Finally, we would like to thank Luca Mesiti for encouraging we explore the monad underlying our structure.

Introduction

The notion of subtyping is often quite tricky because of its double nature: on the one hand, it is meant to represent a relation so simple that one would like to not be bothered to look too much into it but, on the other hand, programming languages do not do well with things that are left unsaid. In [27, §15.1], subtyping $A \leq B$ is defined as a relation between two



© Greta Coraglia and Jacopo Emmenegger;
licensed under Creative Commons License CC-BY 4.0

29th International Conference on Types for Proofs and Programs (TYPES 2023).

Editors: Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg; Article No. 3; pp. 3:1–3:19

Leibniz International Proceedings in Informatics



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

3:2 Categorical Models of Subtyping

types A and B such that, if a term of type A is provided, than it can be “safely” used in a context where B is expected, and the program should not falter if this happens. This is often called the *principle of safe substitution*, and is encoded via a new typing rule that goes by the name of *subsumption*.

$$\text{(Sub)} \frac{\Gamma \vdash a : A \quad \Gamma \vdash A \leq B}{\Gamma \vdash a : B} \quad (1)$$

The problem with subsumption is that it threatens to break certain structures and properties such as canonicity or induction principles. Many variants of subtyping have been proposed, and one in particular has proved itself to be well-behaved with respect to such issues, and that is *coercive subtyping* [21].

In the present work we tackle the problem of subtyping from the point of view of its categorical semantics: we extend a known model to naturally include a notion of subtyping (Section 1), and show how this turns out to be in fact quite closely related to coercive subtyping (Section 2). We then study some examples and applications (Section 3), and conclude by observing some properties that that this new notion intrinsically shows (Section 4).

1 Categorical models of dependent types

The relation between type theory and category theory is one which has been widely studied, and has in the years produced a large variety of models and structures. The interest of the present work is in those that strongly employ objects and techniques coming from the theory of Grothendieck fibrations.

1.1 Grothendieck fibrations

Fibrations were introduced by A. Grothendieck [9] for purposes pertaining to algebraic geometry, but were soon found extremely useful to describe certain phenomena in logic [16]. We will recall key results and definitions when needed, but we refer to [3, Chapter 8] for a detailed introduction.

► **Definition 1** (Cartesian morphism). *Let $p: \mathcal{E} \rightarrow \mathcal{B}$ a functor and $s: B \rightarrow A$ a morphism in \mathcal{E} . We say that s is p -cartesian or cartesian over $\sigma: \Theta \rightarrow \Gamma$ if $p(s) = \sigma$ and for any other $r: C \rightarrow A$ and τ such that $p(r) = \sigma \circ \tau$ there is a unique $t: C \rightarrow B$ in \mathcal{E} such that $p(t) = \tau$ and $s \circ t = r$.*

$$\begin{array}{ccc} \begin{array}{ccc} C & \xrightarrow{r} & A \\ & \searrow \text{dashed} & \downarrow s \\ & & B \end{array} & & \mathcal{E} \\ & & \downarrow p \\ \begin{array}{ccc} \Xi & \xrightarrow{\sigma \circ \tau} & \Gamma \\ & \searrow \tau & \downarrow \sigma \\ & & \Theta \end{array} & & \mathcal{B} \end{array} \quad (2)$$

We say that s is a (p -)cartesian lifting of σ , and it is often denoted $s_{A,\sigma}: \sigma^*A \rightarrow A$.

► **Definition 2** (Fibration). *A functor $p: \mathcal{E} \rightarrow \mathcal{B}$ is a fibration if for all A in \mathcal{E} , each $\sigma: \Theta \rightarrow pA$ has a cartesian lifting. We also say that \mathcal{E} is fibered over \mathcal{B} or that \mathcal{E} is over \mathcal{B} . Oftentimes \mathcal{B} is called the base category and \mathcal{E} the total category of p .*

From the point of view of the logic, we can consider \mathcal{B} to be a category of contexts and substitutions/terms, and \mathcal{E} to be a category of formulae/types, so that each formula is sent to its corresponding context: using fibrations in this way is very much in the spirit of classical categorical logic (see for instance [17, 24, 12, 23]). Asking that for each term and formula there exists a lifting, then, amounts to asking that substitution can be computed and that it indeed produces a formula in then new context. A classical example is that of the Lindenbaum-Tarski algebra for a given first-order theory (Section 3.1).

A particular feature of fibrations is that they induce a factorization system on their total category, because for any given $r: C \rightarrow A$, we can always instantiate the diagram in (2) as follows,

$$\begin{array}{ccc}
 C & & \\
 \downarrow & \searrow r & \\
 \sigma^* A & \xrightarrow{s_{A,\sigma}} & A \\
 \\
 \Theta & & \mathcal{E} \\
 \downarrow \text{id} & \searrow \sigma & \downarrow p \\
 \Theta & \xrightarrow{\sigma} & \Gamma \\
 & & \mathcal{B}
 \end{array}$$

producing a factorization of r . We refer to [26] for a review of its main features. Maps in the left class of such factorization system, meaning maps that are sent to identities, are called *vertical*.

► **Proposition 3** (Vertical/cartesian factorization system). *Consider $p: \mathcal{E} \rightarrow \mathcal{B}$ a fibration. The classes of vertical and cartesian morphisms form an orthogonal factorization system on \mathcal{E} . It additionally has the following properties:*

1. *if g and gf are vertical, then so is f ;*
2. *pullbacks of vertical maps along cartesian ones exist and are vertical.*

If cartesian maps can be thought of as substitutions, vertical maps relate formulae/types in the same context precisely in a way that “does nothing” to the underlying terms.

Not only are vertical maps part of a factorization system, but they can be shown to form a category: more precisely, for each context Γ in \mathcal{B} , one can define a category \mathcal{E}_Γ having for objects those in \mathcal{E} that are sent to Γ , and for morphisms those in \mathcal{E} that are vertical over id_Γ . This intuition is part of one of the most meaningful results of the theory, again due to Grothendieck.

► **Theorem 4** ([9]). *There exists a 2-equivalence*

$$\mathbf{Fib}(\mathcal{B}) \cong \mathbf{Psd}[\mathcal{B}^{\text{op}}, \mathbf{Cat}]$$

between the 2-category of fibrations (with base \mathcal{B}), functors preserving cartesian maps, and natural transformations, and that of contravariant pseudofunctors (from \mathcal{B}), pseudonatural transformations, and modifications.

From left to right, a fibration is sent to a (pseudo)functor that computes for each Γ its *fiber* \mathcal{E}_Γ . From right to left, it performs what is called the *Grothendieck construction* of a (pseudo)functor, which to an $F: \mathcal{B}^{\text{op}} \rightsquigarrow \mathbf{Cat}$, maps the fibration $p: \int F \rightarrow \mathcal{B}$, where objects of $\int F$ are pairs (Γ, A) with Γ in \mathcal{B} and A in $F(\Gamma)$.

Relevant classes of fibrations are those that correspond to functors, which are called *split*, those whose fibers are preorders, which are called *faithful*, and those whose fibers are sets, which are called *discrete*.

1.2 Categories with families

- **Definition 5** (Category with families, [7]). A category with families (cwf) is the data of
- a small category \mathcal{B} with terminal object \top ;
 - a functor $Ty: \mathcal{B}^{\text{op}} \rightarrow \mathbf{Set}$;
 - a functor $Tm: \int Ty^{\text{op}} \rightarrow \mathbf{Set}$
 - for each Γ in \mathcal{B} and A in $Ty(\Gamma)$ an object $\Gamma.A$ in \mathcal{B} , together with two projections $\rho_A: \Gamma.A \rightarrow \Gamma$ and $\nu_A \in Tm(\Gamma.A, Ty_{\rho_A}(A))$ such that for each $\sigma: \Theta \rightarrow \Gamma$ and $a \in Tm(Ty_{\sigma}(A))$ there exists a unique morphism $\Theta \rightarrow \Gamma.A$ making the obvious triangles commute.

In particular, $Ty(\Gamma)$ is a set, the set of types in context Γ , and $Tm(\Gamma, A)$ is again a set, the set of terms of type A in context Γ . The terminal object is meant to model the empty context, while the last condition is what provides context extension – since we are about to give a simpler, equivalent form of it, we do not dwell on it any longer.

Following Theorem 4, we can turn Definition 5 upside down. What we get is the (equivalent, see [2, Prop. 1.2]) notion of natural model.

- **Definition 6** (Natural model, [2]). A natural model is the data of
- a small category \mathcal{B} with terminal object \top ;
 - a discrete fibration $u: \mathcal{U} \rightarrow \mathcal{B}$;
 - a discrete fibration $\dot{u}: \dot{\mathcal{U}} \rightarrow \mathcal{B}$;
 - a fibration morphism $\Sigma: \dot{u} \rightarrow u$ with a right adjoint functor.

In this case u collects types and \dot{u} collects terms, as both are fibered on contexts.

$$\begin{array}{ccc} \Gamma \vdash A \text{ Type} & & \Gamma \vdash a : A \\ u(A) = \Gamma & & \dot{u}(a) = \Gamma, \Sigma(a) = A \\ u \text{ in } \mathbf{Fib}^{\text{disc}}(\mathcal{B}) & & \dot{u} \text{ in } \mathbf{Fib}^{\text{disc}}(\mathcal{B}) \end{array}$$

The fibration morphism (*i.e.* a functor making the diagram below commute) Σ maps to each term its type, while Δ computes for each A what in Definition 5 is called ν_A , meaning a variable in A “transported” to the context $\Gamma.A$.

$$\begin{array}{ccc} \dot{\mathcal{U}} & \begin{array}{c} \Delta \\ \top \\ \Sigma \end{array} & \mathcal{U} \\ \dot{u} \searrow & & \swarrow u \\ & \mathcal{B} & \end{array} \quad (\Sigma) \frac{\Gamma \vdash a : A}{\Gamma \vdash A \text{ Type}} \quad (\Delta) \frac{\Gamma \vdash A \text{ Type}}{\Gamma.A \vdash \nu_A : A} \quad (3)$$

1.3 Generalized categories with families

Our intuition, now, is that we want to use the theory of fibrations to generalize categories with families (or natural models) to the case where fibers over a given context are no longer a set, but a (small) category. If we manage to do this swiftly, we will have found a model for dependent types that introduces an additional relation between types in the same context, so that this relation does imply absolutely no substitution. This leads us to the following definition.

- **Definition 7** (Generalized category with families, [5, 4]). A generalized category with families (gcwf) is the data of
- a small category \mathcal{B} (with terminal object \top)¹;

¹ Existence of the terminal object is needed in case one wants to model the empty context – which one often wants to do. We only put its existence as conditional because we want to begin comparing gcwfs with comprehension categories, and they in turn do not require it.

- a fibration $u: \mathcal{U} \rightarrow \mathcal{B}$;
- a fibration $\dot{u}: \dot{\mathcal{U}} \rightarrow \mathcal{B}$;
- a fibration morphism $\Sigma: \dot{u} \rightarrow u$ with a right adjoint functor, and unit and counit with cartesian components.

Notice that the adjoint pair in Definition 7 is *not* fibered, as Δ does not make the desired triangle commute and unit and counit have cartesian components.

Of course a gcwf with discrete u, \dot{u} is a regular cwf, so that a gcwf is simply a generalization of a well-known model. We choose the name as to remark that this new structure falls into a long tradition of models, which categories with families is perhaps one of the most prominent exponents of, but these could be very easily called “generalized natural models”, or something entirely different (cf. [5]). We conclude this section with one last result relating gcwfs to other notable structures, namely comprehension categories [11], as to show that our path did not stray much away from known territory.

► **Theorem 8** ([6, 4]). *There is a biequivalence between (the 2-category of) gcwfs and (the 2-category of) comprehension categories.*

A generalized category with families, then, is precisely *as good a model as* a comprehension category. The purpose of introducing the new structure, though, lies in the clarity of its use, and we hope that the rest of the present work will be a witness to that.

► **Remark 9.** In our exposition we put aside two elements in the discussion on categorical models for dependent types. The first is of course the issue of coherence: since fibrations involve *pseudofunctors*, equations for identities and, especially, associativity only hold up to vertical isomorphism, while it is usually preferable to have substitution “on the nose”. If one so wishes, the reader is invited to only use *split* fibrations, meaning those that correspond to strict functors, and morphisms that preserve the splitting. A thorough discussion on the topic can be found in [30, §3].

Another sensitive topic is that of definitional equality. For the moment, having other purposes in mind, we settle for interpreting it as identity of objects (in the category over each context) and, therefore, rarely make explicit coherence rules involving it, as they are all trivial from our definitions. This is of course not unprecedented both for the literature on categorical semantics of type theory at large (cf. the largely influential [12] and [7]) and for semantics at large, as it lies in the path of the approach better known as *denotational semantics* (cf. [29]).

2 Vertical maps induce a notion of subtyping

The idea that vertical maps could nicely relate types in the same context is of course not new. The most closely related precedent to this is perhaps [25], followed by the notes in [32], where, quite radically, functors were interpreted to *be* type refinement systems. Our work is similar in spirit – though one should really be careful of the difference between “type refinement” and “subtyping” [32, §2.2] – but of course based on fibrations instead on functors, hence with a special focus on substitution.

2.1 Two new judgements

Now that we are all set, let us describe what it is that we can say in a gcwf, that was not already available in the discrete case. To the two judgements pertaining to types and terms, we add two new ones involving subtyping, and collect them all in Table 1. We read

3:6 Categorical Models of Subtyping

■ **Table 1** Judgements of the theory.

$$\begin{array}{cccc} \Gamma \vdash A \text{Type} & \Gamma \vdash a : A & \Gamma \vdash A' \leq_f A & \Gamma \vdash a :_g A \\ u(A) = \Gamma & \dot{u}(a) = \Gamma, \Sigma(a) = A & f : A' \rightarrow A, u(f) = \text{id}_\Gamma & g : \Sigma a \rightarrow A, u(g) = \text{id}_\Gamma \end{array}$$

the new judgements, respectively, as A' is a subtype of A , as witnessed by f and a is a term of type a subtype of A , as witnessed by g . Recall that vertical maps enjoy some nice properties (Proposition 3), and their combination provides us with structural rules for the new judgements.

► **Proposition 10** (Structural subtyping rules). *The following rules are satisfied by a gcwf.*

$$\begin{array}{cc} (Sbsm) \frac{\Gamma \vdash a :_g A' \quad \Gamma \vdash A' \leq_f A}{\Gamma \vdash a :_{fg} A} & (Trans) \frac{\Gamma \vdash A' \leq_f A \quad \Gamma \vdash A'' \leq_g A'}{\Gamma \vdash A'' \leq_{fg} A} \\ \\ (Sbst) \frac{\Gamma.A \vdash B' \leq_f B \quad \Gamma \vdash a :_g A}{\Gamma \vdash B'[a] \leq_{\dot{u}(\Delta g \eta_a)^* f} B[a]} & (Wkn) \frac{\Gamma \vdash A' \leq_f A \quad \Gamma \vdash B \text{Type}}{\Gamma.B \vdash A' \leq_{(u \in_B)^* f} A} \end{array}$$

Intuitively, Subsumption and Transitivity are both due to the fact that vertical maps compose to vertical maps, while Substitution and Weakening make use of the substitution structure due to the fibration part of the system. We refer to the appendix for further details.

Note that a gcwf is also equipped with a notion of “sub-typing” for terms as well. However, the hom-set $\dot{U}_\Gamma(a, b)$ of vertical arrows over Γ is in bijection with the set of vertical arrows $f : \Sigma a \rightarrow \Sigma b$ such that $(\dot{u}\Delta f)\dot{u}\eta_a = \dot{u}\eta_b$ (see [6, Lemma 3.16]). Therefore a term a is a sub-term of b precisely when the type of a is a sub-type of the type of b , and substituting b (i.e. reindexing along $\dot{u}\eta_b$) is the same as substituting a , modulo the fact that Σa is a sub-type of Σb .

2.2 Comparison with coercive subtyping

As we said in the introduction, it turns out that the calculus resulting in this generalization comes close to coercive subtyping. We hope to convince the reader that, despite all the technical differences, they actually entertain the same spirit.

The basic idea of coercive subtyping is that subtyping is modelled as an abbreviation mechanism: A is a subtype of B , if there is a unique coercion c from A to B , written as $A <_c B$. Then, if a hole in a context requires an object of type B , it is legal to supply an object a of type A – it is equivalent to supplying the object $c(a)$. [22]

A coercion c from A to B is technically a term of the function type $A \rightarrow B$, hence computing $c(a)$ amounts to function application. Coercions are added “manually” to a given type theory, as they are *a priori* not part of the calculus, and the resulting system is shown both to be a conservative extension of the original one and to act well with respect to canonicity.

Let us now collect in a table the main similarities and differences between coercive subtyping and subtyping via vertical maps. We then will consider each point in detail.

<i>coercive subtyping</i>	<i>categorical subtyping</i>
$\Gamma \vdash f : A' \rightarrow A$	$f : A' \rightarrow A$ vertical over Γ
judgements added to the calculus	judgements “added” to the classical model
no witnesses for typing judgements	witnesses for typing judgements
f is unique	f is not necessarily unique
(Sbsm) via substitution	(Sbsm) via composition
satisfies (Trans)	satisfies (Trans)
satisfies (Sbst)	satisfies (Sbst)
satisfies (Wkn)	satisfies (Wkn)
satisfies congruence	satisfies congruence

The main technical difference is of course what coercions *are*, as on one side they are function terms, on the other they are (particular) morphisms in the total category. In some sense this difference is unavoidable: if one starts from the traditional syntax of a type theory, syntactic objects are all that is available, while if one looks at a very general model as if it *was* a syntax (and one would have some merit in doing so, see Section 3.1), then they have more objects at hand. In both cases, though, subtyping is a notion that is somehow independent of the calculus, because in one case it is added by selecting a choice of witnesses (and adding one new judgement for each), while on the other it is literally *orthogonal* to the rest of the structure.

While we will deal with the matter of uniqueness of coercions in Section 2.3, a difference that is unbridgeable is that of dealing with typing judgements: in coercive subtyping, a term can indeed have more than one type, so that a typing judgment is in some sense ambiguous, while in subtyping with vertical maps one can always unambiguously know *by which means* a term is of a certain type, but for this it pays the price of having a whole new set of annotated judgements.

Finally, validity with respect to rules appearing in the table above is common to both systems – by “congruence”, in particular, we mean congruence of the subtyping relation with respect to definitional equality, see Remark 9. It should be remarked that rules appearing on the “coercive subtyping” side are not all rules required, for example, in [22], but they are in a sense the structural ones, as the others aim to discipline either how coercions (as terms of function type) interact with the system, while in our case that is granted due to the fact that they come a categorical setting, which automatically ensures a certain degree of “coherence” (e.g. \leq is automatically reflexive, transitive, and functorial).

2.3 On uniqueness of coercions

As we suggested in presenting this new perspective on subtyping, uniqueness of coercions is not really an issue. In particular, we have a whole theory of fibrations whose fibers are preorders, we actually call them *faithful* fibrations (cf. Section 1.1).

Not only that, but we can show that, given a gwf, if its type fibration is faithful, then so is its term fibration.

► **Proposition 11.** *Let $(u, \dot{u}, \Sigma \dashv \Delta)$ be a gwf. Then Σ is a faithful functor.*

► **Corollary 12.** *Let $(u, \dot{u}, \Sigma \dashv \Delta)$ be a gwf. If u is faithful, then so is \dot{u} .*

In this sense, faithfulness, hence uniqueness of coercions, can indeed be modeled by simply looking at faithful fibrations, and such a burden is in fact not laid on the choice of the fibration collecting terms.

■ **Table 2** Judgements of the theory – faithful case.

$$\Gamma \vdash A \text{Type} \quad \Gamma \vdash a : A \quad \Gamma \vdash A' \leq A \quad \Gamma \vdash a :_{\leq} A$$

From the point of view of the syntax, then, judgements in Table 1 only have one witness and become as in Table 2, with $a :_{\leq} A$ denoting that $\Sigma a \leq A$, and rules in Proposition 10 become the (possibly) more familiar following.

$$\begin{array}{l} \text{(Sbsm)} \frac{\Gamma \vdash a :_{\leq} A' \quad \Gamma \vdash A' \leq A}{\Gamma \vdash a :_{\leq} A} \quad \text{(Trans)} \frac{\Gamma \vdash A' \leq A \quad \Gamma \vdash A'' \leq A'}{\Gamma \vdash A'' \leq A} \\ \text{(Sbst)} \frac{\Gamma.A \vdash B' \leq B \quad \Gamma \vdash a :_{\leq} A}{\Gamma \vdash B'[a] \leq B[a]} \quad \text{(Wkn)} \frac{\Gamma \vdash A' \leq A \quad \Gamma \vdash B \text{Type}}{\Gamma.B \vdash A' \leq A} \end{array}$$

► **Remark 13 (A case against uniqueness).** Though it is a key feature of coercive subtyping, avoiding uniqueness might have its merits. Consider for example the case of sum types: it seems like one should have two different witnesses for the judgement $\Gamma \vdash A \leq A + A$, one per coproduct injection.

2.4 Comparison with related variants of type theory

Directed type theory

Extending the paradigm connecting types, ω -groupoids, and homotopy theory (cf. [31]), to the directed case – meaning involving ω -categories and directed homotopy theory, one encounters the notion of directed type theory. The underlying intuition wishes to add, for each pair of terms of the same type (and substitutions), a new *asymmetric* “identity” type of transformations from one to the other, possibly stopping at some given height/iteration [19]. Though vertical in some sense, this notion of directed-ness is considered only for terms, while in our case the main focus is for types (though one could regard types as terms of a given universe, in DTT transformations are introduced for *all* pairs of terms). Being based on (ω -)categories, they share some properties of morphisms, such as composition and identity, but our vertical maps are a lot simpler, and in no way higher-dimensional – though the monad in Theorem 25 suggests possible extensions in this direction.

Observational type theory

Observational type theory was introduced in [1] to combine the nice computational features of intensional type theory, such as termination of reductions, and propositional equality of extensional type theory, so that two functions are equal if they are equal point-wise, or “if all observations about them agree” [1, §1]. The fact that conversion rules allow to pass implicitly between definitionally equal types is extended to a mechanism that allows to pass implicitly between *provably* equal types. This process is explicit and its agents are called *coercions*. Coercions for a OTT work in a way that is much similar to our vertical maps, but their being originated by proofs of equality of types ensures that for a given coercion, one can always find one associated to it and going in the opposite direction.

Practical subtyping

Another relevant syntactic approach is that of “practical subtyping” as introduced in [18]. There, the new subtyping judgements are *ternary* relations as $t \in A \subset B$, which are encoded as sorts of implications: “if t is a term of type A , then it is a term of type B ”. In particular,

this notion of subtyping most notably (and “practically”) describes subtyping from the perspective of terms, and not types, as in our case. In particular, subtyping between types, meaning intrinsic judgements such as $A \subseteq B$, are encoded by means of choice operators, such as Hilbert’s ϵ [18, p. 26].

We can compare our categorical subtyping with the practical one as follows: again, consider the case where all fibrations involved are faithful (cf. Section 2.3), so that there is at most *one* subtyping judgement for each pair of types (or for a type and a term). In particular, judgements in [18, p. 22] seem to match our structural ones, when one replaces “ $t : A$ ” with our “ $t :_{\leq} A$ ”. The treatment of type constructors, too, seems quite keen in spirit to ours, but we postpone an in-depth analysis of said comparison to future work.

3 Examples and applications

3.1 Gcwfs from Lindenbaum-Tarski

Consider the Lindenbaum-Tarski algebra of a given first-order theory \mathcal{T} in a language \mathcal{L} . We take \mathbf{ctx} to be the category where

- objects are lists of distinct variables $x = (x_1, \dots, x_n)$,
- arrows are lists of substitution for variables, meaning $[t_1/y_1, \dots, t_m/y_m] = [t/y] : x \rightarrow y$, with t_j ’s being \mathcal{L} -terms that are built on variables x_1, \dots, x_n ,

and composition is defined by simultaneous substitution. The product of two lists x and y is a list w with length the sum of the lengths of x and y , and projections on x and y are substitutions with, respectively, the first n and the last m variables in w . Categorically and in the sense of [16], this is the free Lawvere theory on the language \mathcal{L} .

One can define the functor $LT_{\mathcal{T}} : \mathbf{ctx}^{\text{op}} \rightarrow \mathbf{InfSL}$ so that to each list x , the category $LT_{\mathcal{T}}(x)$ has for objects equivalence classes of well-formed formulae in \mathcal{L} with free variables at most those that are in x , and with respect to the equivalence relation induced by reciprocal deducibility in \mathcal{T} , $\phi \dashv\vdash_{\mathcal{T}} \phi'$. Notice that this makes our treatment *proof-irrelevant*. Maps in $LT_{\mathcal{T}}(x)$ are provable consequences in \mathcal{T} . Composition is given by the cut rule of the calculus, and identities are tautologies. Since substitution preserves provability, $LT_{\mathcal{T}}$ can be suitably extended to a functor, and its correspondent under the Grothendieck construction Theorem 4 is a faithful fibration $p : \int LT_{\mathcal{T}} \rightarrow \mathbf{ctx}$.

Such a construction was first introduced with the name of (*hyper*)*doctrine* in [17] and is thoroughly explained in [14] and [23]. We here show that it underlies the structure of a gcwf where terms are entailments.

► **Example 14.** Call \mathcal{E} the category of p -vertical maps and commutative squares, which is again fibered over \mathbf{ctx} – call e this fibration, $\text{Cod} : \mathcal{E} \rightarrow \int LT_{\mathcal{T}}$ the codomain functor, Diag the functor mapping each formula to its identity. The triple $(p, e, \text{Cod} \dashv \text{Diag})$ is a gcwf.

This gcwf is in fact *not* a cwf. Here, types are formulae and terms are (unique) witnesses to entailment, meaning triples $x; \phi \vdash \psi$ where ϕ and ψ are both formulae in the fiber over x . The underlying notion of subtyping actually coincides with terms.

► **Remark 15.** This is an example of a more general instance, which will thoroughly be discussed in Section 4.

3.2 Gcwfs from topos theory

Let \mathcal{E} be an elementary topos and $\top: 1 \rightarrow \Omega$ its subobject classifier, then consider

$$\begin{array}{ccc}
 \mathcal{E}/1 & \begin{array}{c} \xleftarrow{\Delta_{\top}} \\ \xrightarrow{\Sigma_{\top}} \end{array} & \mathcal{E}/\Omega \\
 & \searrow \sim & \swarrow \text{dom} \\
 & & \mathcal{E}
 \end{array}$$

with $\Sigma_{\top} = \top \circ -$ and $\Delta_{\top}(\phi: X \rightarrow \Omega) = (\text{canonical p.b. of } \phi \text{ along } \top)$.

► **Example 16.** The triple $(\sim, \text{dom}, \Sigma_{\top} \dashv \Delta_{\top})$ is a gcwf.

Compatibly with the Mitchell-Bénabou interpretation, types are (proof irrelevant) propositions, Δ_{\top} computes the comprehension $\{x \mid \phi(x)\}$, and so on – we refer to [13, Part D], [15, Chap. II] for an extensive treatment of the topic, and to [20] for a quick overview. In this case, the fibrations involved are discrete, hence this is really is a cwf, and there is no subtyping.

A topos is quite a general structure, so let us break down a couple of examples for it. The prototypical example of an elementary topos is **Set**, the categories of sets and functions, with subobject classifier $\top: 1 \rightarrow 2$, $* \mapsto 1$, which classifies subsets via their characteristic function.

$$\begin{array}{ccc}
 A_{\phi} & \longrightarrow & 1 \\
 i_{\phi} \downarrow & \lrcorner & \downarrow \top \\
 A & \xrightarrow{\phi} & 2
 \end{array}$$

In this case terms are sets, types are functions $\phi: A \rightarrow 2$, which are equivalently subsets of A – and the reason why we usually call them *propositions*. The context for each proposition ϕ is its domain set A . Let us now break down rules as in (3),

$$(\Sigma_{\top}) \frac{A \vdash A}{A \vdash i_{\text{id}_A} \text{Type}} \quad (\Delta_{\top}) \frac{A \vdash \phi \text{Type}}{A_{\phi} \vdash A_{\phi}} \tag{4}$$

where we use the fact that $\Delta_{\top}(\phi) = \text{dom}i_{\phi} = \{a \mid \phi(a)\} \subseteq A$ by definition. Notice how, again, Δ does not make the triangle commute, as, in principle, $A \neq A_{\phi}$.

Let us now look at a more complicated case, namely that of **Eff**, the effective topos [10, 28], whose objects are sets A with an \mathbb{N} -valued equality predicate $=_A \in \mathcal{P}(\mathbb{N})^{A \times A}$, and whose morphisms are $\mathcal{P}(\mathbb{N})$ -valued functional relations: we think of each subset of \mathbb{N} as the *realizers* (in the sense of Kleene) for a given proposition. One can show that **Eff** is a topos with subobject classifier $\top: (\{*\}, \mathbb{N}) \rightarrow (\mathcal{P}(\mathbb{N}), \leftrightarrow)$, where $A \leftrightarrow B := (A \rightarrow B) \wedge (B \rightarrow A)$ is a pair of recursive functions mimicking bi-implication [10, §1], and $\top(*, A) = [A \leftrightarrow \mathbb{N}]$. In this case, terms are sets with an \mathbb{N} -valued equality predicate, and types are subobjects of them, which can be shown to be sort of subsets with a compatible equality predicates and an additional “membership” unary predicate compatible with it.

3.3 Gcwfs from pullbacks

For \mathcal{C} with pullbacks we can define $\mathcal{C}^{\rightarrow}$ the category of arrows in \mathcal{C} and **Sec**(\mathcal{C}) of sections of arrows in \mathcal{C} , meaning of pairs (s, f) with $f, s \in \mathcal{C}^{\rightarrow}$ and $f \circ s = \text{id}$. Therefore a type in context A is a map $f: B \rightarrow A$ and a term of type f is one of its sections, s . For a given type

f (map) context extension can be performed using what is usually called its “kernel pair” construction, meaning computing the following pullback,

$$\begin{array}{ccc}
 B & \xrightarrow{\text{id}} & B \\
 \downarrow d(f) & \searrow & \downarrow f \\
 K_f & \xrightarrow{\quad} & B \\
 \downarrow f^+ & \swarrow & \downarrow f \\
 B & \xrightarrow{\quad} & A
 \end{array}$$

which we can interpret as the universal f -induced congruence on B . One can always compute the section $B \rightarrow K_f$, which can be extended to a functor $K: \mathcal{C}^{\rightarrow} \rightarrow \mathbf{Sec}(\mathcal{C})$.

► **Example 17.** Call $U: \mathbf{Sec}(\mathcal{C}) \rightarrow \mathcal{C}^{\rightarrow}$ the functor mapping each pair (s, f) , with s section of f , to f . The triple $(\text{cod}, \text{cod}U, U \dashv K)$ is a gcwf.

Actually, by Theorem 8, the same can be said for any \mathcal{D} subcategory of $\mathcal{C}^{\rightarrow}$ closed for pullbacks, for example we can consider monomorphisms, and their sections. Notice that this is *not* a cwf, as vertical maps are commutative triangles.

Let us give a couple of examples of this case, as well. We look at **Set** again, for a function $f: B \rightarrow A$, we find that K_f is the set of pairs of elements of B with the same image through f , and $B \rightarrow K_f$ is the diagonal. We again try to break down rules as in (3), and get the following.

$$(U) \frac{A \vdash f: B \hookrightarrow A: s}{A \vdash f: B \rightarrow A \text{ Type}} \quad (K) \frac{A \vdash f: B \rightarrow A \text{ Type}}{B \vdash f^+: K_f \hookrightarrow B: d(f)} \quad (5)$$

Again, it is important that the right adjoint K , the one performing context extension, does not make the triangle commute, as B is not (necessarily) equal to A . Let us now look at subtyping judgements: morphisms in $\mathcal{C}^{\rightarrow}$ are commutative squares, vertical ones are squares such that the codomain is the identity, hence commutative triangles. The whole set of structural judgements available in this model (cf. Section 2.1) is then the following.

$$\begin{array}{ccc}
 \begin{array}{c} A \vdash f \text{ Type} \\ \text{dom } f \\ \downarrow f \\ A \end{array} & \begin{array}{c} A \vdash s: f \\ \text{dom } f \\ \begin{array}{c} \uparrow s \\ \downarrow f \end{array} \\ A \end{array} & \begin{array}{c} A \vdash f' \leq_h f \\ \text{dom } f' \xrightarrow{h} \text{dom } f \\ \begin{array}{ccc} & \searrow & \swarrow \\ & f' & f \\ & \searrow & \swarrow \\ & A & \end{array} \end{array} & \begin{array}{c} A \vdash s: k f \\ \text{dom } f' \xrightarrow{k} \text{dom } f \\ \begin{array}{ccc} & \searrow & \swarrow \\ & f' & f \\ & \searrow & \swarrow \\ & A & \end{array} \end{array}
 \end{array}$$

Of course one could consider a great multitude of other categories, such as **Pos**, **Top**, **Vect_K**, **Grp**, all topoi, and many others.

3.4 On type constructors

We begin by considering function types of the form $\text{Fun}(A, B)$, where A and B are in the same context (in particular, B does not depend on A). In this case, given a subtype A' of A and a subtype B of B' (note the different order), one would like to conclude that $\text{Fun}(A, B)$ is a subtype of $\text{Fun}(A', B')$. In our proposed framework, this can be expressed by requiring an action of Fun on vertical arrows, which is contravariant in the first component.

3:12 Categorical Models of Subtyping

In the case of a discrete gcwf (that is, in the fibrational formulation of a natural model/cwf) the existence of function types can be expressed by requiring the existence of functors Fun and abs making the right-hand square below a pullback [2]

$$\begin{array}{ccccc}
 \mathcal{U}_{\dot{u}\Delta \times_{\dot{u}} \dot{\mathcal{U}}} & \longleftarrow & \mathbf{W}^*(\mathcal{U}_{\dot{u}\Delta \times_{\dot{u}} \dot{\mathcal{U}}}) & \xrightarrow{\text{abs}} & \dot{\mathcal{U}} \\
 \text{id} \times \Sigma \downarrow & & \downarrow & & \downarrow \Sigma \\
 \mathcal{U}_{\dot{u}\Delta \times_u \mathcal{U}} & \xleftarrow{\mathbf{W}} & \mathcal{U}_{u \times_u \mathcal{U}} & \xrightarrow{\text{Fun}} & \mathcal{U}
 \end{array}$$

where the left-hand square is a pullback. The lower functor Fun simply takes two types A and B in the same context Γ to the type $\text{Fun}(A, B)$. The functor \mathbf{W} is weakening of the second type with the first one: it takes a pair (A, B) of two types A and B in the same context Γ to the pair $(A, (u\epsilon_A)^*B)$ where $(u\epsilon_A)^*B$ is the type B weakened to the context $\Gamma.A$. As the left-hand square is a pullback, the domain of the functor abs consists of pairs (A, b) of a type A in context Γ and a term b of the weakened type $(u\epsilon_A)^*B$ in context $\Gamma.A$. The functor abs maps such a pair to the term $\text{abs}(A, b)$. Commutativity of the square ensures that the type of $\text{abs}(A, b)$ is $\text{Fun}(A, B)$. As it is not relevant for our discussion, we refer to [2] for details on how the pullback property of the right-hand square validates both the elimination rule and the η -rule.

To extend this setting to include subtyping in the form of vertical arrows, we need to take into account that the action of Fun (and abs) should be contravariant *only on the vertical arrows* of the first component. Contravariant actions of functors on some category \mathcal{B} are rendered by considering functors on the opposite category \mathcal{B}^{op} , but here we want to take the opposite only of vertical arrows, otherwise substitution, *i.e.* the action of Fun on cartesian arrows, would go in the wrong direction. We need to consider what is known as the *fiberwise opposite* $u^\circ: \mathcal{U}^\circ \rightarrow \mathcal{B}$ of u , see [30, Section 5]. The construction of u° uses the orthogonal factorisation system on \mathcal{U} given by vertical and cartesian arrows to construct a bicategory on the same objects of \mathcal{U} whose 1-cells $B \rightarrow B'$ are spans $B \leftarrow \bar{B}' \rightarrow B'$ of a vertical arrow $B \leftarrow \bar{B}'$ and a cartesian one $\bar{B}' \rightarrow B'$. 2-cells are morphisms of spans, that is, arrows between the vertices making the two triangles commute. It turns out that 2-cells are unique (given two spans) and always invertible. The category \mathcal{U}° induced by identifying two isomorphic 1-cells is again fibered over \mathcal{B} , where an arrow $[g, \bar{\sigma}]$ is cartesian (over $\sigma = \dot{u}\bar{\sigma}$) if g is invertible (equivalently, if there is a representative of the form $(\text{id}, \bar{\sigma}')$). It is easy to see that, for every Γ , there is an isomorphism $(\mathcal{U}^\circ)_\Gamma \cong (\mathcal{U}_\Gamma)^{\text{op}}$ natural in Γ . In fact, it is well-known (see *e.g.* [26, Theorem 3.7]) that the fibered category u° is equivalent (over \mathcal{B}) to the fibered category $\mathbf{F}((-)^{\text{op}} \circ \mathbf{P}(u))$, where $\mathbf{P}: \mathbf{Fib}(\mathcal{B}) \rightarrow \mathbf{Psd}[\mathcal{B}^{\text{op}}, \mathbf{Cat}]$ and $\mathbf{F}: \mathbf{Psd}[\mathcal{B}^{\text{op}}, \mathbf{Cat}] \rightarrow \mathbf{Fib}(\mathcal{B})$ are the two 2-functors realizing the 2-equivalence in Theorem 4.

► **Remark 18.** If we start with a fibration equipped with a (split) cleavage, then it is easy to see that u° can also be equipped with a (split) cleavage. Indeed, we can pick $(A, \sigma) \mapsto [\text{id}, s(A, \sigma)]$ as cleavage for u° , where s is a cleavage for u . If s is split, then the above choice is clearly normal, and $(\text{id}, s(A, \sigma \circ \rho))$ is a representative for $[\text{id}, s(s_0(A, \sigma), \rho)] \circ [\text{id}, s(A, \sigma)]$. Furthermore, a cleavage s for u provides us with a choice of a span $(g, s(A, \sigma))$ in each equivalence class. These choices compose (*i.e.* form a category) as soon as the cleavage is split. Therefore, in the case of a split fibration u , an arrow $A \rightarrow B$ in \mathcal{U}° over some σ in \mathcal{B} is actually just a vertical arrow $s_0(B, \sigma) \rightarrow A$.

► **Proposition 19.** *Let $(u, \dot{u}, \Sigma, \Delta)$ be a gcwf over \mathcal{B} equipped with two morphisms of fibrations Fun and abs making the right hand square below a pullback over \mathcal{B} .*

$$\begin{array}{ccccc}
\mathcal{U}^{\circ}_{\dot{u} \circ \Delta \circ \times \dot{u}} \dot{\mathcal{U}} & \longleftarrow & \mathbb{W}^*(\mathcal{U}^{\circ}_{\dot{u} \circ \Delta \circ \times \dot{u}} \dot{\mathcal{U}}) & \xrightarrow{\text{abs}} & \dot{\mathcal{U}} \\
\text{id} \times \Sigma \downarrow & & \downarrow & & \downarrow \Sigma \\
\mathcal{U}^{\circ}_{\dot{u} \circ \Delta \circ \times u} \mathcal{U} & \longleftarrow & \mathbb{W}_{u \circ \times u} \mathcal{U} & \xrightarrow{\text{Fun}} & \mathcal{U}
\end{array}$$

Then, in addition to the usual formation, introduction, elimination and computation rules for function types, the following rules are satisfied.

$$\frac{\Gamma \vdash A' \leq_f A \quad \Gamma \vdash B \leq_g B'}{\Gamma \vdash \text{Fun}(A, B) \leq_{\text{Fun}(f, g)} \text{Fun}(A', B')} \quad \frac{\Gamma.A \vdash b :_f B}{\Gamma \vdash \text{abs}(A, b) :_{\text{Fun}(\text{id}_A, f)} \text{Fun}(A, B)}$$

In order to extend Proposition 19 to the case of dependent function types, *i.e.* Π -types, we need to take into account two changes in variance. One is due, as in the non-dependent case, to the fact that Π acts contravariantly on the first argument. The second change in variance happens because the second argument depends on the first one, but the subtyping relation on second arguments goes in the opposite direction of the subtyping between on first arguments. These two changes in variance can be described as follows. Consider first a pullback of Σ as in the left-hand diagram below. Then take its fiberwise opposite, resulting in the morphism of fibrations in the middle below. Finally, compose that triangle with the fibration $u : \mathcal{U} \rightarrow \mathcal{B}$ and take again the fiberwise opposite of the result, obtaining the morphism of fibrations Σ_{Π} on the right-hand below, from $\dot{u}_{\Pi} := (u \circ \dot{u}_1^{\circ})^{\circ}$ to $u_{\Pi} := (u \circ u_1^{\circ})^{\circ}$.

$$\begin{array}{ccc}
\begin{array}{ccccc}
\dot{\mathcal{U}} & \longleftarrow & \dot{\mathcal{U}}_1 & & \\
\downarrow \Sigma & & \downarrow \Sigma_1 & & \\
\mathcal{U} & \longleftarrow & \mathcal{U}_1 & & \\
\downarrow \dot{u} & & \downarrow \dot{u}_1 & & \\
\mathcal{B} & \longleftarrow & \mathcal{U} & & \\
& & \downarrow \dot{u} \Delta & & \\
& & \mathcal{U} & &
\end{array} &
\begin{array}{ccc}
\dot{\mathcal{U}}_1^{\circ} & \xrightarrow{\Sigma_1^{\circ}} & \mathcal{U}_1^{\circ} \\
\downarrow \dot{u}_1^{\circ} & & \downarrow u_1^{\circ} \\
\mathcal{U} & & \mathcal{U}
\end{array} &
\begin{array}{ccc}
\dot{\mathcal{U}}_{\Pi} & \xrightarrow{\Sigma_{\Pi}} & \mathcal{U}_{\Pi} \\
\downarrow \dot{u}_{\Pi} & & \downarrow u_{\Pi} \\
\mathcal{B} & & \mathcal{B}
\end{array} \quad (6)
\end{array}$$

Unfolding the constructions involved, we see that an arrow $(f, [g, \bar{f}]) : (A, B) \rightarrow (A', B')$ in \mathcal{U}_{Π} vertical over Γ consists of an arrow $f : A' \rightarrow A$ vertical over Γ together with a 1-cell $[g, \bar{f}] : B' \rightarrow B$ in \mathcal{U}_1° with $g : \bar{B} \rightarrow B'$ vertical over $\Gamma.A' = \dot{u} \Delta A'$ and $\bar{f} : \bar{B} \rightarrow B$ cartesian over $\dot{u} \Delta f : \Gamma.A' \rightarrow \Gamma.A$. Similarly, an arrow $(A, b) \rightarrow (A', b')$ in $\dot{\mathcal{U}}_{\Pi}$ vertical over Γ consists of an arrow $f : A' \rightarrow A$ as before, together with a 1-cell $[\dot{g}, \bar{f}]$ in $\dot{\mathcal{U}}_1^{\circ}$ with $\dot{g} : \bar{b} \rightarrow b'$ vertical over $\Gamma.A'$ and $\bar{f} : \bar{b} \rightarrow b$ cartesian over $\dot{u} \Delta f$. The following result follows.

► **Proposition 20.** *Let $(u, \dot{u}, \Sigma, \Delta)$ be a gcwf equipped with two morphisms of fibrations Π and abs making the square below a pullback over \mathcal{B} .*

$$\begin{array}{ccc}
\dot{\mathcal{U}}_{\Pi} & \xrightarrow{\text{abs}} & \dot{\mathcal{U}} \\
\Sigma_{\Pi} \downarrow & & \downarrow \Sigma \\
\mathcal{U}_{\Pi} & \xrightarrow{\Pi} & \mathcal{U}
\end{array}$$

Then, in addition to the usual formation, introduction, elimination and computation rules for dependent function types, as well as the η -rule, the following rules are satisfied.

$$\frac{\Gamma \vdash A' \leq_f A \quad \Gamma.A' \vdash B[\dot{u} \Delta f] \leq_g B'}{\Gamma \vdash \Pi(A, B) \leq_{\Pi(f, g)} \Pi(A', B')} \quad \frac{\Gamma.A \vdash b :_g B}{\Gamma \vdash \text{abs}(A, b) :_{\Pi(\text{id}_A, g)} \Pi(A, B)}$$

A type constructor like the dependent sum (*i.e.* Σ -types), on the other hand, requires no change in variance. Therefore its formulation for a gcwf is a straightforward generalization of the discrete case. The fibration classifying the premises is what is called u_1 in diagram (6)

above: the objects, as expected, are the same of u_{Π} , but the vertical arrows $(A, B) \rightarrow (A', B')$ now are just pairs of a vertical arrow $f: A \rightarrow A'$ and $g: B \rightarrow B'$ over $\dot{u}\Delta f$. The construction of the fibration \dot{u}_{Σ} classifying the premises of the introduction rule is slightly more involved, and we refer to [5, Example 3.7.4] for a complete description.

► **Proposition 21.** *Let $(u, \dot{u}, \Sigma, \Delta)$ be a gcwf equipped with two morphisms of fibrations Σ and pair making the square below a pullback over \mathcal{B} .*

$$\begin{array}{ccc} \dot{U}_{\Sigma} & \xrightarrow{\text{pair}} & \dot{U} \\ \Sigma_{\Sigma} \downarrow & & \downarrow \Sigma \\ \mathcal{U}_1 & \xrightarrow{\Sigma} & \mathcal{U} \end{array}$$

Then, in addition to the usual formation, introduction, elimination and computation rules for dependent sum types, as well as the η -rule, the following rules are satisfied.

$$\frac{\Gamma \vdash A \leq_f A' \quad \Gamma.A \vdash B \leq_g B'[\dot{u}\Delta f]}{\Gamma \vdash \Sigma(A, B) \leq_{\Sigma(f,g)} \Sigma(A', B')} \quad \frac{\Gamma \vdash a :_f A \quad \Gamma.A \vdash b :_g B[a]}{\Gamma \vdash \text{pair}(a, b) :_{\Sigma(f,g)} \Sigma(A, B)}$$

In writing the rules above in Propositions 20 and 21, we have written the action of reindexing as if the fibrations involved were split. This allows us to simplify notation in the rules, which would otherwise look quite cumbersome. In the case of non-split fibrations, a judgement like $\Gamma.A \vdash B \leq_g B'[\dot{u}\Delta f]$ in a premise should be read as “for all pairs of a cartesian $\bar{f}: \bar{B}' \rightarrow B'$ over $\dot{u}\Delta f$ and a vertical $g: B \rightarrow \bar{B}'$ ”, and similarly for the other judgements. In particular, in the non-split case, the cartesian arrow \bar{f} should appear in the conclusion as well, as one of the arguments of $\Sigma(f, g)$.

4 The “subtyping” monad

We now deepen our intuition for both the properties and the features of this new construction. In particular, the process of *taking into consideration* vertical maps turns out to amount to the result of the action of a particular monad. Grothendieck fibrations are known to be strongly linked to certain (co)monads, to the point of them being classified as pseudo-algebras of a given one (cf. [8] for a recent review and extension). We here present a simple monad that has the nice feature of collecting vertical maps, and show that it actually produces gcwfs out of gcwfs.

This section is perhaps best intended not as a *development*, but more of an additional perspective of our theory of subtyping: though it does not provide new results, it puts its emerging in context, and can hopefully be used in future works to combine different notions with this one – for example, what would a distributive law combining the monad here and that producing split fibrations [30, §3] mean, and what would it entail?

We start from the endofunctor $(\text{Id}_-/\text{Id}_-): \mathbf{Fib}(\mathcal{B}) \rightarrow \mathbf{Fib}(\mathcal{B})$ computing for each fibration $p: \mathcal{E} \rightarrow \mathcal{B}$ the *comma object* $(\text{Id}_p/\text{Id}_p)$, meaning the following (very boring) 2-limit below.

$$\begin{array}{ccc} (\text{Id}_p/\text{Id}_p) & \longrightarrow & p \\ \downarrow & \swarrow & \downarrow \text{Id}_p \\ p & \xrightarrow{\text{Id}_p} & p \end{array}$$

To make the reding a bit smoother, we denote $(\text{Id}_p/\text{Id}_p)$ by (p/p) . Unpacking this categorical construction, one finds that the new fibration $(p/p): \mathcal{E} \downarrow^v \mathcal{E} \rightarrow \mathcal{B}$ has in the domain triples (f, A', A) such that $f: A' \rightarrow A$ is a p -vertical map, while morphisms in $\mathcal{E} \downarrow^v \mathcal{E}$ are simply commutative squares.

► **Lemma 22.** *There is a monad on $\mathbf{Fib}(\mathcal{B})$ with endofunctor $(\text{Id}_-/ \text{Id}_-)$.*

Its associated unit has components $\eta: p \rightarrow Tp$ sending each object to its identity, while the multiplication $\mu_p: TTp \rightarrow Tp$ is fiber-wise composition – as objects in the total category of TTp are squares of vertical maps, and vertical maps compose.

This simple construction we can actually extend to a gcwf, meaning we can use it to build out of a gcwf with type fibration u , a new gcwf with type fibration Tu .

► **Proposition 23.** *Let $(u, \dot{u}, \Sigma \dashv \Delta)$ a gcwf. Then there are adjoint functors $\overline{\Sigma} \dashv \overline{\Delta}$ such that $(u/u, \Sigma/u, \overline{\Sigma} \dashv \overline{\Delta})$ is a gcwf.*

If we look at the basic judgments this second gcwf is describing, we will find precisely the two introduced in Section 2.1, so that we can reformulate our old perspective:

$$\begin{array}{cccc} \Gamma \vdash A \text{Type} & \Gamma \vdash a : A & \Gamma \vdash A' \leq_f A & \Gamma \vdash a :_g A \\ u(A) = \Gamma & \dot{u}(a) = \Gamma, \Sigma(a) = A & f: A' \rightarrow A, u(f) = \text{id}_\Gamma & g: \Sigma a \rightarrow A, u(g) = \text{id}_\Gamma \end{array}$$

into the newer following notation.

$$\begin{array}{cccc} \Gamma \vdash A \text{Type} & \Gamma \vdash a : A & \Gamma \vdash A' \leq_f A & \Gamma \vdash a :_g A \\ u(A) = \Gamma & \dot{u}(a) = \Gamma, \Sigma(a) = A & (u/u)(f, A', A) = \Gamma & (\Sigma/u)(g, a, A) = \Gamma, p_2 \circ \overline{\Sigma}(g, a, A) = A \end{array}$$

Notice now the symmetry of the first and the second two judgments: the two new ones can be regarded as “classifiers” for new types and terms themselves, in a way such that it is always possible to recover the original theory – one simply has to look at those vertical maps that, in particular, are identities.

► **Definition 24** (The category of gcwfs). *A gcwf morphism $(u, \dot{u}, \Sigma \dashv \Delta) \rightarrow (v, \dot{v}, \Sigma' \dashv \Delta')$ is the data of a pair (H, \dot{H}) , with $H: u \rightarrow v$, $\dot{H}: \dot{u} \rightarrow \dot{v}$ fibration morphisms such that $H\Sigma = \Sigma'\dot{H}$. We denote \mathbf{GCwF} the category of gcwfs and gcwf morphisms, and $\mathbf{GCwF}(\mathcal{B})$ its subcategory with fixed context category \mathcal{B} .*

Again, it should be noted that gcwfs can be equipped with more interesting (and higher dimensional) structure [4, §3.2], but for the purposes of the present paper we are not interested in that.

► **Theorem 25.** *The monad in Lemma 22 can be lifted to a monad over $\mathbf{GCwF}(\mathcal{B})$.*

If a monad is an object for abstract computation, this one in particular computes “subtyping judgements”. An interesting question is now to check what objects does iterating this computation produce: the first iteration collects vertical maps, and morphisms are squares; in particular, vertical morphisms are those squares having vertical (with respect to the original type fibration) components, therefore the second iteration produced the fibration of said “all vertical” squares; here morphisms are cubes, and the vertical ones are those where all “connecting” maps are vertical (again, with respect to the original type fibration), and so on.

5 Open problems

Of course the applications to constructors can be taken much further than what is done in Section 3.4, and considering more complex ones can lead to a new extension of the present theory. Our intention was to introduce the “vertical maps” perspective, so that whoever gets interested in categorical models for type theory is motivated to consider the benefit of including non-discrete fibrations – usually at very little additional cost.

We conclude this exposition with a take that was suggested to the first author by F. Dagnino, and which we believe might be of some interest: as a fibration can be regarded as an internal category in the category of discrete fibrations, it seems like the process of going from a theory *without* a notion of subtyping to a theory *with* subtyping is close to that of considering a category internal to another one. In this sense, our work is perhaps not best interpreted as a generalization of some known models, but as a structure internal to them.

References

- 1 Thorsten Altenkirch and Conor McBride. Towards Observational Type Theory. Manuscript, available at <https://www.cs.nott.ac.uk/~psztxa/publ/ott-conf.pdf>, 2006.
- 2 Steve Awodey. Natural models of homotopy type theory. *Mathematical Structures in Computer Science*, 28(2):241–286, 2018. doi:10.1017/S0960129516000268.
- 3 Francis Borceux. *Handbook of Categorical Algebra*, volume 2 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994. doi:10.1017/CB09780511525865.
- 4 Greta Coraglia. *Categorical structures for deduction*. PhD thesis, Università degli Studi di Genova, 2023. URL: <https://hdl.handle.net/11567/1134175>.
- 5 Greta Coraglia and Ivan Di Liberti. Context, judgement, deduction, 2021. arXiv:2111.09438.
- 6 Greta Coraglia and Jacopo Emmenegger. A 2-categorical analysis of context comprehension, 2024. arXiv:2403.03085.
- 7 Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 120–134, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. doi:10.1007/3-540-61780-9_66.
- 8 Jacopo Emmenegger, Luca Mesiti, Giuseppe Rosolini, and Thomas Streicher. A comonad for Grothendieck fibrations. *Theory and Applications of Categories*, 40:371–389, 2023.
- 9 Alexander Grothendieck. Catégories fibrées et descente (Exposé VI). *Revêtements étales et groupe fondamental - SGA1*, 1960-61.
- 10 J. Martin E. Hyland. The Effective Topos. In A.S. Troelstra and D. van Dalen, editors, *The L. E. J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 165–216. Elsevier, 1982. doi:10.1016/S0049-237X(09)70129-6.
- 11 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.
- 12 Bart Jacobs. *Categorical logic and type theory*. Elsevier, 1999.
- 13 Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium: Volume 1*. Oxford University Press, 2002.
- 14 Anders Kock and Gonzalo E. Reyes. Doctrines in categorical logic. In *Studies in Logic and the Foundations of Mathematics*, volume 90, pages 283–313. Elsevier, 1977. doi:10.1016/S0049-237X(08)71104-2.
- 15 Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.
- 16 F. William Lawvere. *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the Context of Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963. Available as Reprints in *Theory and Applications of Categories*, No. 5 (2004) pp 1-121 at <http://www.tac.mta.ca/tac/reprints/articles/5/tr5abs.html>.
- 17 F.W. Lawvere. Equality in hyperdoctrines and comprehension schema as an adjoint functor. In A. Heller, editor, *Proc. New York Symposium on Application of Categorical Algebra*, pages 1–14. Amer.Math.Soc., 1970.
- 18 Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for Curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1):5:1–5:58, 2019. doi:10.1145/3285955.
- 19 Daniel R. Licata and Robert Harper. 2-dimensional directed type theory. *Electronic Notes in Theoretical Computer Science*, 276:263–289, 2011. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII). doi:10.1016/j.entcs.2011.09.026.

- 20 Zhen Lin Low. Logic in a topos, 2013. Manuscript, available at <http://z1122.user.srcf.net/talks/2013-01-24-InternalLogic.pdf>.
- 21 Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999. doi:10.1093/logcom/9.1.105.
- 22 Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: Theory and implementation. *Information and Computation*, 223:18–42, 2013. doi:10.1016/j.ic.2012.10.020.
- 23 Maria E. Maietti and Giuseppe Rosolini. Quotient completion for the foundation of constructive mathematics. *Logica Universalis*, 7(3):371–402, 2013. doi:10.1007/s11787-013-0080-2.
- 24 Michael Makkai. The fibrational formulation of intuitionistic predicate logic I: completeness according to Gödel, Kripke, and Läuchli, part 2. *Notre Dame J. Formal Log.*, 34:471–498, 1993. doi:10.1305/ndjfl/1093633902.
- 25 Paul-André Mellies and Noam Zeilberger. Functors are type refinement systems. *SIGPLAN Not.*, 50(1):3–16, January 2015. doi:10.1145/2775051.2676970.
- 26 David Jaz Myers. Cartesian factorization systems and Grothendieck fibrations, 2020. arXiv:2006.14022.
- 27 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- 28 Edmund Robinson and Giuseppe Rosolini. Colimit completions and the effective topos. *The Journal of Symbolic Logic*, 55(2):678–699, 1990. doi:10.2307/2274658.
- 29 Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Technical Report PRG06, OUCL, August 1971. Available at <https://www.cs.ox.ac.uk/publications/publication3723-abstract.html>.
- 30 Thomas Streicher. Fibred categories à la Jean Bénabou, 2022. arXiv:1801.02927.
- 31 Benno van den Berg and Richard Garner. Types are weak ω -groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, October 2010. doi:10.1112/plms/pdq026.
- 32 Noam Zeilberger. Principles of type refinement, 2016. Notes of lectures at the Oregon Programming Language Summer School 2016, available at <http://noamz.org/oplss16/refinements-notes.pdf>.

A Proofs

Proof of Proposition 10. The first two rules, Subsumption (*Sbsm*) and Transitivity (*Trans*), simply follow from composition of vertical arrows. Substitution (*Sbst*) and Weakening (*Wkn*) are a bit trickier and make use of, precisely, the substitutional part of the structure, namely it being a fibration. We begin with Weakening, as it is a bit easier: consider that the premise of the rule is given by maps and objects as below.

$$\begin{array}{ccc}
 & A' & \\
 & \downarrow f & \\
 & A & \\
 \Gamma.B \xrightarrow{u\epsilon_B} \Gamma & & \begin{array}{c} U \\ \downarrow u \\ B \end{array}
 \end{array}$$

Since B is a type in context Γ , we can compute the counit $\epsilon: \Sigma\Delta B \rightarrow B$, which we know by (3) that acts as weakening, in particular producing the context extension $u\epsilon_B: \Gamma.B \rightarrow \Gamma$. In the fiber over Γ , not only do we have B , but we also have a vertical map $f: A' \rightarrow A$. Applying the reindexing $(u\epsilon_B)^*$ to the three of them yields a new vertical map over $\Gamma.B$, hence the judgement $\Gamma.B \vdash (u\epsilon_B)^* A' \leq_{(u\epsilon_B)^* f} (u\epsilon_B)^* A$. In Proposition 10 we have actually written A for $(u\epsilon_B)^* A$, at it is customary for weakening rules – and as, in fact, it appears in (3). All in all, the Weakening rule only tells us that subtyping is preserved through weakening.

Now let us get to Substitution. In order to prove it, we first need to recall how $B[a]$ is computed in a traditional natural model (cf. [2, §2.1]): for given $\Gamma \vdash a : A$ and $\Gamma.A \vdash B$, $B[a]$ is the result of the reindexing B along the (image of the) unit η_a . (This is not precisely how the original paper presents it, though it is equivalent to that: see [4, §2.3.4] for a detailed discussion on the matter.) In our setting, the premises of *(Sbst)* start from the following.

$$\begin{array}{ccccc}
 \Sigma a & \longrightarrow & \Sigma \Delta \Sigma a & \longrightarrow & \Sigma a \\
 \downarrow & & \searrow & & \downarrow g \\
 A & \dashrightarrow & \Sigma \Delta A & \longrightarrow & A \\
 & & & & \mathcal{U} \\
 & & & & \downarrow u \\
 & & & & \mathcal{B} \\
 \Gamma & \xrightarrow{\dot{u}\eta_a} & \Gamma.\Sigma a & \xrightarrow{\dot{u}\Delta g} & \Gamma.A & \xrightarrow{u\epsilon_{\Sigma a}} & \Gamma
 \end{array}$$

Notice that, since g is vertical, and the top horizontal compositions in the top diagram is an identity due to triangle identities of $\Sigma \dashv \Delta$, by cartesianness of the counit $\epsilon_A : \Sigma \Delta A \rightarrow A$ there is a unique map making the top diagram commute. Now, each “external” side in the top diagram is mapped to the identity through u , meaning we can repeat Awodey’s argument, and the required vertical arrow is the result of reindexing along the (image of the) unit – post-composed with $\dot{u}\Delta g$. Notice that a is only “allowed” to interact with B, B' through g , which in turn does nothing to contexts, so that we are in our right to write, as customary, $B[a]$ for $(\dot{u}(\Delta g \circ \eta_a))^* B$. ◀

► **Remark 26.** It might not seem like it, but this last proof makes heavy use of the universal property of the comma object (u/u) , and it could be in fact entirely proved using that profusely. We point the interested reader to [4, §3.5] for the corresponding alternative proof.

Proof of Proposition 11. Each component of the unit in a gcwf is a monic arrow. Indeed, let $f, g : a \rightarrow b$ in \mathcal{U} be such that $\eta_b f = \eta_b g$. It follows that

$$\dot{u}f = (u\epsilon_{\Sigma b})(\dot{u}\eta_b)(\dot{u}f) = (u\epsilon_{\Sigma b})(\dot{u}\eta_b)(\dot{u}g) = \dot{u}g$$

and, in turn, that $f = g$ since η_b is cartesian. The left adjoint Σ is then faithful. ◀

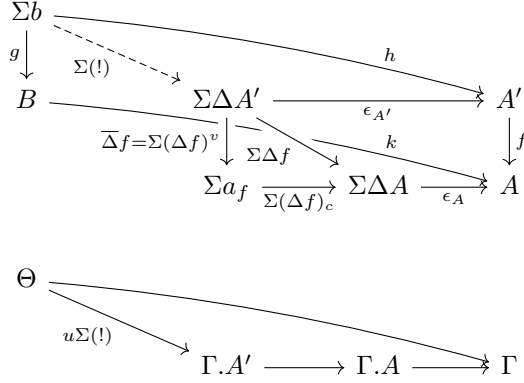
Proofs of Propositions 19–21. This is straightforward, by reading the action of the morphisms involved (**Fun**, Π , **abs**, Σ , **pair**) on vertical arrows and unfolding the definitions of the sub-typing judgements as in Table 1. ◀

Proof of Proposition 23. We begin by describing what the fibration of terms (Σ/u) does: it simply collects pairs (g, a, A) such that $g : \Sigma a \rightarrow A$ is a u -vertical map, and sends them to their underlying context. Its vertical maps are pairs of vertical maps (respectively \dot{u} - and u -vertical) fitting in appropriate squares, and its cartesian maps are pairs of cartesian maps (again, respectively, with reference to \dot{u} and u).

The required typing functor, then, is simply $\bar{\Sigma}$ sending each triple (g, a, A) to $(g, \Sigma a, A)$, which is cartesian because Σ is. Describing its right adjoint requires a little more effort: for a triple (f, A', A) in (u/u) one considers its image through Δ , then its vertical-cartesian factorization system (cf. Proposition 3), then the image of the vertical portion of it through Σ to get back to \mathcal{U} .

$$\begin{array}{ccc}
 \begin{array}{c} A' \\ \downarrow f \\ A \end{array} & \begin{array}{ccc} \Delta A' & & \\ (\Delta f)^v \downarrow & \searrow \Delta f & \\ a_f & \xrightarrow{(\Delta f)_c} & \Delta A \end{array} & \begin{array}{ccc} \Sigma \Delta A' & \xrightarrow{\epsilon_{A'}} & A' \\ \bar{\Delta}f = \Sigma(\Delta f)^v \downarrow & \searrow \Sigma \Delta f & \downarrow f \\ \Sigma a_f & \xrightarrow{\Sigma(\Delta f)_c} & \Sigma \Delta A \xrightarrow{\epsilon_A} A \end{array}
 \end{array}$$

Cartesianness of $(\Delta f)_c$ can be used to provide functoriality. Let us now show how the two functors form an adjoint pair by means of the universal property of its counit, which is actually depicted on the right square above. For any triple (g, b, B) in (Σ/u) and morphism $(h, k): \bar{\Sigma}(g, b, B) \rightarrow (f, A', A)$ we must show that there is a unique pair (m, n) such that $\bar{\Sigma}(m, n): \bar{\Sigma}(g, b, B) \rightarrow \bar{\Sigma}\bar{\Delta}(f, A', A)$ makes the obvious triangle commute. The universal property of the counit ϵ'_A yields a unique $m: b \rightarrow \Delta A'$ so that $h = \epsilon_{A'} \circ m$,



hence a map filling the bottom diagram to a commuting triangle. By cartesianness of $\epsilon_A \circ \Sigma(\Delta f)_c$ there is a unique map $n: B \rightarrow \Sigma a_f$ making the “other” triangle commute as well. ◀

Proof of Theorem 25. Let us denote (T, η, μ) the desired monad. On objects it of course acts as $T(u, \dot{u}, \Sigma \vdash \Delta) = ((u/u), (\Sigma/u), \bar{\Sigma} \dashv \bar{\Delta})$ as in Proposition 23. The image of a morphism (H, \dot{H}) is actually simply the action of H , and it provides a commutative square because $H\Sigma = \Sigma' \dot{H}$. Unit is again induced by identity, and multiplication by composition. ◀

A Sound and Complete Substitution Algorithm for Multimode Type Theory

Joris Ceulemans   

DistriNet, KU Leuven, Belgium

Andreas Nuyts   

DistriNet, KU Leuven, Belgium

Dominique Devriese   

DistriNet, KU Leuven, Belgium

Abstract

Multimode Type Theory (MTT) is a generic type theory that can be instantiated with an arbitrary mode theory to model features like parametricity, cohesion and guarded recursion. However, the presence of modalities in MTT significantly complicates the substitution calculus of this system. Moreover, MTT’s syntax has explicit substitutions with an axiomatic system – not an algorithm – governing the connection between an explicitly substituted term and the resulting term in which variables have actually been replaced. So far, the only results on eliminating explicit substitutions in MTT rely on normalisation by evaluation and hence also immediately normalise a term. In this paper, we present a substitution algorithm for MTT that is completely separated from normalisation. To this end, we introduce Substitution-Free Multimode Type Theory (SFMTT): a formulation of MTT without explicit substitutions, but for which we are able to give a structurally recursive substitution algorithm, suitable for implementation in a total programming language or proof assistant. On the usual formulation of MTT, we consider σ -equality, the congruence generated solely by equality rules for explicit substitutions. There is a trivial embedding from SFMTT to MTT, and a converse translation that eliminates the explicit substitutions. We prove soundness and completeness of our algorithm with respect to σ -equivalence and thus establish that MTT with σ -equality has computable σ -normal forms, given by the terms of SFMTT.

2012 ACM Subject Classification Theory of computation \rightarrow Type theory; Theory of computation \rightarrow Modal and temporal logics; Software and its engineering \rightarrow Syntax

Keywords and phrases dependent type theory, modalities, multimode type theory, explicit substitutions, substitution algorithm

Digital Object Identifier 10.4230/LIPIcs.TYPES.2023.4

Supplementary Material *Text (Technical report)*: <https://arxiv.org/abs/2406.13622> [13]

Funding *Joris Ceulemans*: Held a PhD fellowship (1184122N) of the Research Foundation – Flanders (FWO) while working on this research. This research is partially funded by the Research Fund KU Leuven and by the Research Foundation - Flanders (FWO; G030320N).

Andreas Nuyts: Holds a Postdoctoral fellowship (1247922N) of the Research Foundation – Flanders (FWO).

1 Introduction

Substitution is the operation that replaces variables in a term with other terms. It is a key part in defining the semantics of many programming languages. In a dependent type system, it is even necessary in order to formulate the typing rules, such as the one for dependent function application. However, defining substitution is not as simple as it intuitively may seem.



© Joris Ceulemans, Andreas Nuyts, and Dominique Devriese;
licensed under Creative Commons License CC-BY 4.0

29th International Conference on Types for Proofs and Programs (TYPES 2023).

Editors: Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg; Article No. 4; pp. 4:1–4:23



Leibniz International Proceedings in Informatics

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

1.1 Renaming and Substitution in the Simply Typed Lambda Calculus

For example, consider the well-known simply typed lambda calculus. We call $\mathsf{Tm}^{\text{STLC}}(\Gamma \vdash T)$ the set of terms of type T with free variables in context Γ and $\mathsf{Sub}^{\text{STLC}}(\Gamma \rightarrow \Delta)$ the set of well-formed (simultaneous) substitutions from Γ to Δ . These substitutions are lists of terms: they contain a term of type T in context Γ for every variable of type T in context Δ . In other words, STLC substitutions are constructed in two ways: $!_{\Gamma} \in \mathsf{Sub}^{\text{STLC}}(\Gamma \rightarrow \cdot)$ representing the empty list and $\sigma.t \in \mathsf{Sub}^{\text{STLC}}(\Gamma \rightarrow (\Delta, x : T))$ which substitutes variables in Δ according to $\sigma \in \mathsf{Sub}^{\text{STLC}}(\Gamma \rightarrow \Delta)$ and substitutes $t \in \mathsf{Tm}^{\text{STLC}}(\Gamma \vdash T)$ for the variable $x : T$.

Applying a substitution $\sigma \in \mathsf{Sub}^{\text{STLC}}(\Gamma \rightarrow \Delta)$ to a term $t \in \mathsf{Tm}^{\text{STLC}}(\Delta \vdash T)$ should produce a term $t[\sigma] \in \mathsf{Tm}^{\text{STLC}}(\Gamma \vdash T)$. This can be defined via recursion on the term t . Some cases are very simple: for variables x the corresponding term is found in σ and for applications $(fs)[\sigma]$ we recurse on the subterms $(f[\sigma])(s[\sigma])$. However, difficulty arises when binders are involved. For lambda terms $\lambda x.s \in \mathsf{Tm}^{\text{STLC}}(\Delta \vdash T \rightarrow S)$ with $s \in \mathsf{Tm}^{\text{STLC}}(\Delta, x : T \vdash S)$, the substitution $(\lambda x.s)[\sigma]$ is defined as $\lambda x.(s[\sigma^+])$ where $\sigma^+ \in \mathsf{Sub}^{\text{STLC}}((\Gamma, x : T) \rightarrow (\Delta, x : T))$ is a version of σ that is lifted to the contexts extended with x . We can construct σ^+ as $\mathsf{weaken}(\sigma).x$, where $\mathsf{weaken}(\sigma)$ contains the same terms as σ , but weakened to live in the extended context $\Gamma, x : T$. A naive definition might implement this weakening of terms $t \in \mathsf{Tm}^{\text{STLC}}(\Gamma \vdash A)$ to $\mathsf{Tm}^{\text{STLC}}(\Gamma, x : B \vdash A)$ by applying a substitution from $\Gamma, x : B$ to Γ , but this makes the story cyclic.

An elegant solution to avoid this cycle, standard in the literature, is to separately consider renamings and substitutions. Whereas a substitution maps variables to terms, a renaming from Γ to Δ maps every variable in Δ to a variable in Γ of the same type. Weakening, in particular, is a renaming.¹ Thus, the terms listed in a substitution can be weakened by applying a weakening *renaming*, and the variables listed in a renaming – represented as De Bruijn indices – can be weakened by incrementation. So we can break the cycle by defining first how to rename and then how to substitute in a term, each time by induction on the term. Going further, code duplication between the two term traversals can be avoided with a shared generic implementation [23, 4].

1.2 Multimode Type Theory

This paper is concerned with substitution in modal type theory, more specifically in the system MTT (Multimode Type Theory²) by Gratzer et al. [20]. MTT is a type theory that can be instantiated with a mode theory that specifies, among others, a collection of modes and modalities. Modes m index typing judgements and qualify their meaning: judgements in one mode may represent, for example, regular values, while judgements in other modes may represent time-indexed values or pairs of values satisfying a certain relation [12]. Modalities $\mu : m_1 \rightarrow m_2$ represent ways to transport terms and types from mode m_1 to mode m_2 . We postpone a more extensive introduction to MTT to Section 2, but we will already explain why (algorithmic) substitution in modal type theory is significantly more complicated.

First, modes and modalities complicate the context structure in MTT. For every modality μ , MTT has a new primitive context operation $_.\blacksquare_{\mu}$ which also extends to substitutions: if $\sigma \in \mathsf{Sub}^{\text{MTT}}(\Gamma \rightarrow \Delta)$, then we get a new substitution $\sigma.\blacksquare_{\mu} \in \mathsf{Sub}^{\text{MTT}}(\Gamma.\blacksquare_{\mu} \rightarrow \Delta.\blacksquare_{\mu})$.³ Furthermore, all variables in a context are annotated with a modality. This also impacts

¹ Note that our notion of renaming has little to do with α -conversion. Rather, a renaming will map free variables to possibly different free variables. It is also a useful concept in an unnamed setting.

² The names Multimode and Multimodal Type Theory are used interchangeably for the same system MTT which supports both multiple modes and multiple modalities.

³ The operation $_.\blacksquare_{\mu}$ can be seen as some sort of left adjoint to μ . See Section 2.1 for more details.

how substitutions are defined: to produce a substitution from Γ to $\Delta . (\mu \mid x : T)$ (i.e. Δ extended with a variable x of type T annotated with modality μ), we need to provide a $\sigma \in \text{Sub}^{\text{MTT}}(\Gamma \rightarrow \Delta)$ and a term $t \in \text{Tm}^{\text{MTT}}(\Gamma . \blacksquare_{\mu} \vdash T)$ in a locked context. Complicating things further, mode theories can define 2-cells $\alpha \in \mu \Rightarrow \rho$ between modalities μ and ρ . For every 2-cell $\alpha \in \mu \Rightarrow \rho$ from μ to ρ and every context Γ we get a new primitive *key substitution* $\mathfrak{R}_{\Gamma}^{\alpha}$ from $\Gamma . \blacksquare_{\rho}$ to $\Gamma . \blacksquare_{\mu}$ and we have to specify how these act on variables and terms. We conclude that MTT substitutions are not mere lists of terms and applying substitutions to variables is not just a lookup operation.

In the original presentation of MTT [20, 19], these difficulties are circumvented by using explicit substitutions [1]: the syntax for terms has a constructor for applying a substitution to a term. A system of judgemental equality axioms then allows us to rewrite the explicitly substituted terms. However, this axiom system does not provide an algorithm to compute substitutions away. A priori, it is not even clear if every MTT term is judgementally equal to a term in which no explicit substitutions occur.

In this paper we give a positive answer to the last question by constructing a substitution algorithm for MTT. Moreover, we want this algorithm to be structurally recursive so that it can be implemented in a proof assistant. This requirement makes the construction even more complicated: in a non-modal setting such as STLC, composition of substitutions can be a defined operation. However, in MTT the additional primitive substitutions make this impossible (we refer to Theorem 2 for more details). For that reason, MTT includes a primitive constructor $\tau \circ \sigma$ for substitution composition. However, in an algorithm for computing $t[\sigma]$ we first traverse t until we reach a variable. During this phase, the substitution σ can grow, for instance the lifting operation $+$ is applied when going under a binder. To then compute $x[\sigma]$ for a variable x , we perform a case split and recursion on σ . In the case where σ is a composite of the form $\tau \circ \psi$, we would like to define $x[\tau \circ \psi]$ as $(x[\tau])[\psi]$. However, $x[\tau]$ is again an arbitrary term so that $(x[\tau])[\psi]$ may trigger another arbitrary term traversal. Thus, this naïve definition of the substitution algorithm is not structurally recursive, and restructuring the algorithm to restore structural recursion is one of the main contributions of the current paper (Section 3).

1.3 Contributions and Overview

In this paper, we define substitution for MTT, resolving the above problems by identifying the equivalent of renamings and substitutions in MTT and building a structurally recursive substitution algorithm in terms of them. Specifically, we contribute the following.

- We define WSMTT: an intrinsically and modally scoped untyped syntax for MTT. One can see MTT as an extrinsic typing discipline over WSMTT and as such, our substitution results for WSMTT carry over to MTT. Moreover, we define σ -equivalence for WSMTT: the congruence relation generated by substitution-related equality rules, but not β - and η -rules.
- We define SFMTT: a variant of WSMTT without explicit substitutions in terms or types. Moreover, we define a notion of SFMTT renamings and substitutions and implement a structurally recursive algorithm to apply those to types and terms.
- We provide a translation $\llbracket _ \rrbracket$ from WSMTT to SFMTT, which translates every WSMTT term and type to an expression without substitutions. In the other direction, there is an almost trivial embedding function $\text{embed}(_)$ from SFMTT to WSMTT.
- We prove the soundness and completeness of our algorithm. Soundness means that the substitution-free WSMTT term obtained as $\text{embed}(\llbracket t \rrbracket)$ is σ -equivalent to the original term t . Completeness states that $\llbracket _ \rrbracket$ maps σ -equivalent WSMTT terms to equal SFMTT terms. Both results combined show that SFMTT terms are the σ -normal forms of WSMTT terms.

Section 2 will provide the necessary background and details about the multimode type theory MTT and introduce WSMTT. We continue in Section 3 to describe the SFMTT syntax and the algorithm for renaming and substitution in that setting. The translation $\llbracket _ \rrbracket$ from WSMTT to SFMTT is also discussed there. Section 4 then covers the soundness and completeness results. We conclude in Section 5 with related and future work. A technical report accompanying this paper contains all details of the soundness and completeness proofs, as well as full descriptions of the systems WSMTT and SFMTT [13].

2 Multimode Type Theory (MTT)

In this section we introduce the type system MTT as developed by Gratzer et al. [20]. We start in Section 2.1 with the necessary background and continue in Section 2.2 with our own presentation of MTT that we call WSMTT, including a discussion of the differences with the original formulation. In this section we also discuss (WS)MTT's substitution calculus. Section 2.3 concludes with a discussion on an equivalence relation on terms and substitutions called σ -equivalence.

2.1 Background on the MTT Type System

MTT can be seen as a *framework* for modal type theory: it is parametrised by a mode theory which specifies the modalities and how they interact. More concretely, a mode theory in MTT is a strict 2-category of which the 0-cells (objects) are called modes and the 1-cells (morphisms) are called modalities. This already makes it clear that we have a unit modality $\mathbb{1}_m$ for every mode m (sometimes just written $\mathbb{1}$ when the mode is clear) and that compatible modalities can be composed. Moreover, we also have a notion of 2-cells between modalities, which will be denoted as $\alpha \in \mu \Rightarrow \nu$ for a 2-cell α from μ to ν . Such 2-cells can be composed vertically (which we write as $\beta \circ \alpha$) and horizontally (written as $\beta \star \alpha$). For every modality $\mu : m \rightarrow n$ there is a unit 2-cell $1_\mu \in \mu \Rightarrow \mu$.

In MTT, every judgement (so every context, type and term) lives at a particular mode of the mode theory. This is made clear by adding $@m$ to a judgement at mode m . We can think of every mode as containing a copy of Martin-Löf Type Theory (MLTT [22]) with natural numbers, products, etc. As they are confined to a single mode and do not really interact with modalities, we will not discuss these rules in the paper (as an illustration we do include a type of Booleans in the technical report though [13]). The connection between the different modes is made via the modalities, as explained in the following paragraphs.

A selection of the rules for constructing contexts, types and terms in MTT can be found in Figure 1. Contexts consist of variables (CTX-EXTEND), each annotated with a modality, and locks (CTX-LOCK), which play an important role in determining when a variable can be used to construct a term. Note that a lock goes in the opposite direction of its modality: the lock operation for a modality $\mu : m \rightarrow n$ takes a context from mode n to mode m .

A variable can be used as a term whenever there is a 2-cell from its annotation to the composition of all locks to the right of that variable (TM-VAR). Note that the 2-cell α to access a variable is an integral part of the term and consequently the terms x^α and x^β are not considered equal when the 2-cells α and β are distinct. Furthermore, an operation $_^\alpha$ is applied to the type T of the variable in order to bridge the gap between the context in the conclusion of TM-VAR and $\Gamma . \mathbb{1}_\mu$ (in which T is well-formed). We refer to [20] for more details about this operation.

$$\begin{array}{c}
\text{CTX-EMPTY} \\
\frac{}{\cdot \text{ctx} @ m} \\
\text{locks}(\cdot) = \mathbb{1} \\
\text{TY-ARROW} \\
\frac{\mu : m \rightarrow n \quad \Gamma . \mathbf{\blacklozenge}_\mu \vdash T \text{ ty} @ m \quad \Gamma . (\mu \mid x : T) \vdash S \text{ ty} @ n}{\Gamma \vdash (\mu \mid T) \rightarrow S \text{ ty} @ n} \\
\text{TM-LAM} \\
\frac{\mu : m \rightarrow n \quad \Gamma . (\mu \mid x : T) \vdash s : S @ n}{\Gamma \vdash \lambda(\mu \mid x).s : (\mu \mid T) \rightarrow S @ n} \\
\text{CTX-LOCK} \\
\frac{\Gamma \text{ ctx} @ n \quad \mu : m \rightarrow n}{\Gamma . \mathbf{\blacklozenge}_\mu \text{ ctx} @ m} \\
\text{locks}(\Gamma . \mathbf{\blacklozenge}_\mu) = \text{locks}(\Gamma) \circ \mu \\
\text{TM-VAR} \\
\frac{\mu : m \rightarrow n \quad \alpha \in \mu \Rightarrow \text{locks}(\Delta)}{\Gamma . (\mu \mid x : T) . \Delta \vdash x^\alpha : T^\alpha @ m} \\
\text{TM-APP} \\
\frac{\mu : m \rightarrow n \quad \Gamma \vdash f : (\mu \mid T) \rightarrow S @ n \quad \Gamma . \mathbf{\blacklozenge}_\mu \vdash t : T @ m}{\Gamma \vdash \text{app}_\mu(f; t) : S [\text{id}.t] @ n} \\
\text{CTX-EXTEND} \\
\frac{\Gamma \text{ ctx} @ n \quad \mu : m \rightarrow n \quad \Gamma . \mathbf{\blacklozenge}_\mu \vdash T \text{ ty} @ m}{\Gamma . (\mu \mid x : T) \text{ ctx} @ n} \\
\text{locks}(\Gamma . (\mu \mid x : T)) = \text{locks}(\Gamma) \\
\text{TM-MOD} \\
\frac{\mu : m \rightarrow n \quad \Gamma . \mathbf{\blacklozenge}_\mu \vdash t : T @ m}{\Gamma \vdash \text{mod}_\mu(t) : \langle \mu \mid T \rangle @ n} \\
\text{TY-MOD} \\
\frac{\mu : m \rightarrow n \quad \Gamma . \mathbf{\blacklozenge}_\mu \vdash T \text{ ty} @ m}{\Gamma \vdash \langle \mu \mid T \rangle \text{ ty} @ n}
\end{array}$$

■ **Figure 1** Selection of rules that define MTT contexts, types, and terms.

$$\begin{array}{c}
\rho \\
\begin{array}{ccc}
m & \xrightarrow{\quad} & n \\
\mu & \xleftarrow{\quad} &
\end{array}
\end{array}
\quad
\begin{array}{l}
\eta \in \mathbb{1}_n \Rightarrow \rho \circ \mu \\
\varepsilon \in \mu \circ \rho \Rightarrow \mathbb{1}_m
\end{array}$$

■ **Figure 2** The mode theory for Examples 1 and 8 is the strict 2-category freely generated by the depicted modalities and 2-cells (triangle identities for η and ε have been omitted).

Every modality μ gives rise to a modal type former $\langle \mu \mid _ \rangle$ which can be seen as a (weak) dependent right adjoint [10] to $_ . \mathbf{\blacklozenge}_\mu$ (TY-MOD). One direction of transposition for this dependent adjunction is given by TM-MOD: to construct a term of type $\langle \mu \mid T \rangle$, we must construct a term of type T after locking the context with μ . We do not discuss the MTT elimination principle for modal types here.

Finally, we can also consider modal function types (TY-ARROW). Their values can be constructed via lambda abstraction (TM-LAM), which adds an annotated variable to the context. Eliminating functions is done via application (TM-APP) where the argument should type check in a locked context. Note that we are using a substitution in this rule to accommodate for dependent types, but we postpone the discussion about substitution in MTT to Section 2.2.1.

► **Example 1.** To illustrate MTT, we look at an example program in a concrete mode theory as depicted in Figure 2. This mode theory consists of an adjunction of modalities $\mu \dashv \rho$, as witnessed by the unit 2-cell η and the counit 2-cell ε . For such a mode theory, Gratzer et al. [19, 20] already showed that the type formers $\langle \mu \mid _ \rangle$ and $\langle \rho \mid _ \rangle$ can also be seen as adjoint. For example, the unit function f of type $(\mathbb{1} \mid A) \rightarrow \langle \rho \mid \langle \mu \mid A^\eta \rangle \rangle$ can be constructed as follows: $f = \lambda(\mathbb{1} \mid x). \text{mod}_\rho(\text{mod}_\mu(x^\eta))$. The variable x gets bound under the unit modality $\mathbb{1}$ by lambda abstraction and subsequently the modal constructors mod_ρ and mod_μ add $\mathbf{\blacklozenge}_\rho$ and $\mathbf{\blacklozenge}_\mu$ to the context. In such a context, the variable x can be used, since the 2-cell η has the proper domain and codomain to access it according to the rule TM-VAR.

2.2 Alternative Presentation: Extrinsically Typed, Intrinsically Scoped

The way the MTT syntax is presented in the previous section, which is also how it is originally presented in [20], can be called *intrinsically typed*. This means that we see the typing rules from Figure 1 as the way types and terms are introduced. In other words, we cannot even talk about ill-typed terms or ill-formed types.

$$\begin{array}{c}
\text{SCTX-EMPTY} \\
\frac{}{\cdot \text{ sctx } @ m} \\
\\
\text{LOCKTELE-EMPTY} \\
\frac{}{\cdot : \text{ LockTele}(m \rightarrow m)} \\
\text{locks}(\cdot) = \mathbb{1}
\end{array}
\qquad
\begin{array}{c}
\text{SCTX-LOCK} \\
\frac{\hat{\Gamma} \text{ sctx } @ n \quad \mu : m \rightarrow n}{\hat{\Gamma} . \mathbf{\mu} \text{ sctx } @ m} \\
\\
\text{LOCKTELE-LOCK} \\
\frac{\Lambda : \text{ LockTele}(o \rightarrow n) \quad \mu : m \rightarrow n}{\Lambda . \mathbf{\mu} : \text{ LockTele}(o \rightarrow m)} \\
\text{locks}(\Lambda . \mathbf{\mu}) = \text{locks}(\Lambda) \circ \mu
\end{array}
\qquad
\begin{array}{c}
\text{SCTX-EXTEND} \\
\frac{\hat{\Gamma} \text{ sctx } @ n \quad \mu : m \rightarrow n}{\hat{\Gamma} . \mu \text{ sctx } @ n}
\end{array}$$

■ **Figure 3** Definition of scoping contexts and lock telescopes.

For the purposes of this paper, it will be more useful to work with extrinsically typed (one could say raw) syntax. In that way, our substitution algorithm can work on pure syntax without having to take typing derivations into account. Moreover, substitution is necessary to formulate some typing rules (such as TM-APP). In MTT, this does not lead to circularity thanks to the use of explicit substitutions (see further) but it would make a substitution algorithm problematically cyclic if it works with intrinsically typed syntax.

However, in order to conveniently develop our substitution algorithm, we will use *intrinsically scoped* syntax, defined in this section. In order to distinguish between our system and the original presentation of MTT, we call the intrinsically scoped syntax WSMTT (for well-scoped MTT). Apart from the change from an intrinsically-typed to an extrinsically-typed presentation, this reformulation does not modify the MTT type theory. Specifically, it does not modify MTT’s treatment of substitution; that will only happen in Section 3, in a different system called SFMTT.

For defining the intrinsically scoped syntax, we introduce scoping contexts in Figure 3. They are essentially MTT contexts from Figure 1 where all type information has been removed. We note that in the rule SCTX-EXTEND only the modality annotation of a variable is added to a scoping context. Indeed, in the rest of the paper we will not use named variables but a form of De Bruijn indices. This allows us to ignore α -equivalence and variable capture when implementing substitution.

The WSMTT syntax is now introduced via a judgement $\hat{\Gamma} \vdash_{\text{ws}} t \text{ expr } @ m$, meaning that t is a WSMTT expression in scoping context $\hat{\Gamma}$ at mode m . Intrinsic scoping means that the inference rules for such a judgement do not define a relation between scoping contexts and some predefined notion of raw syntax; they rather *construct* a WSMTT expression, which can not be seen outside of its scoping context. Put differently, in a proof assistant one would formalise WSMTT expressions as a dependent type indexed by scoping contexts. Note that since we are not specifying typing rules, the distinction between types and terms has disappeared and we talk about WSMTT *expressions*.

Some examples of rules that introduce WSMTT syntax (in other words, WSMTT constructors) can be found in the first two rows of Figure 4. In order to construct a modal function type in scoping context $\hat{\Gamma}$, we need a domain type in the locked scoping context $\hat{\Gamma} . \mathbf{\mu}$ and a codomain type where we extend the scoping context with a variable annotated with μ (WSMTT-EXPR-ARROW). The rule for introducing lambda abstraction is similar (WSMTT-EXPR-LAM). Note that we can obtain all these constructors by removing the typing information from the typing rules in Figure 1. The WSMTT variable rule WSMTT-EXPR-VAR has changed somewhat with respect to Figure 1: it only allows us to access the last variable added to a scoping context and only if it is locked behind the same modality as its annotation. It is standard, in formulations of type theory with explicit substitutions [1], to only allow access to the last variable which has De Bruijn index zero, since the De Bruijn index can then be

$$\begin{array}{c}
\text{WSMTT-EXPR-ARROW} \\
\frac{\mu : m \rightarrow n \quad \hat{\Gamma} . \mathbf{\mu} \vdash_{\text{ws}} T \text{ expr} @ m \quad \hat{\Gamma} . \mu \vdash_{\text{ws}} S \text{ expr} @ n}{\hat{\Gamma} \vdash_{\text{ws}} (\mu \uparrow T) \rightarrow S \text{ expr} @ n} \\
\\
\text{WSMTT-EXPR-LAM} \\
\frac{\mu : m \rightarrow n \quad \hat{\Gamma} . \mu \vdash_{\text{ws}} t \text{ expr} @ n}{\hat{\Gamma} \vdash_{\text{ws}} \lambda^\mu (t) \text{ expr} @ n} \\
\\
\text{WSMTT-EXPR-VAR} \\
\frac{\hat{\Gamma} \text{ sctx} @ n \quad \mu : m \rightarrow n}{\hat{\Gamma} . \mu . \mathbf{\mu} \vdash_{\text{ws}} \mathbf{v}_0 \text{ expr} @ m} \\
\\
\text{WSMTT-EXPR-SUB} \\
\frac{\hat{\Delta} \vdash_{\text{ws}} t \text{ expr} @ m \quad \vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m}{\hat{\Gamma} \vdash_{\text{ws}} t [\sigma]_{\text{ws}} \text{ expr} @ m} \\
\\
\text{WSMTT-SUB-EMPTY} \\
\frac{}{\vdash_{\text{ws}} ! \text{ sub}(\hat{\Gamma} \rightarrow \cdot) @ m} \\
\\
\text{WSMTT-SUB-ID} \\
\frac{}{\vdash_{\text{ws}} \text{id sub}(\hat{\Gamma} \rightarrow \hat{\Gamma}) @ m} \\
\\
\text{WSMTT-SUB-WEAKEN} \\
\frac{\mu : m \rightarrow n \quad \hat{\Gamma} \text{ sctx} @ n}{\vdash_{\text{ws}} \pi \text{ sub}(\hat{\Gamma} . \mu \rightarrow \hat{\Gamma}) @ n} \\
\\
\text{WSMTT-SUB-COMPOSE} \\
\frac{\vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Delta} \rightarrow \hat{\Xi}) @ m \quad \vdash_{\text{ws}} \tau \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m}{\vdash_{\text{ws}} \sigma \circ \tau \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Xi}) @ m} \\
\\
\text{WSMTT-SUB-LOCK} \\
\frac{\vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n \quad \mu : m \rightarrow n}{\vdash_{\text{ws}} \sigma . \mathbf{\mu} \text{ sub}(\hat{\Gamma} . \mathbf{\mu} \rightarrow \hat{\Delta} . \mathbf{\mu}) @ m} \\
\\
\text{WSMTT-SUB-KEY} \\
\frac{\Theta, \Psi : \text{LockTele}(n \rightarrow m) \quad \alpha \in \text{locks}(\Theta) \Rightarrow \text{locks}(\Psi)}{\vdash_{\text{ws}} \mathbf{Q}_{\hat{\Gamma}}^{\alpha \in \Theta \Rightarrow \Psi} \text{ sub}(\hat{\Gamma} . \Psi \rightarrow \hat{\Gamma} . \Theta) @ m} \\
\\
\text{WSMTT-SUB-EXTEND} \\
\frac{\mu : m \rightarrow n \quad \vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n \quad \hat{\Gamma} . \mathbf{\mu} \vdash_{\text{ws}} t \text{ expr} @ m}{\vdash_{\text{ws}} \sigma . t \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta} . \mu) @ n}
\end{array}$$

■ **Figure 4** Constructors for intrinsically well-scoped WSMTT expressions (defined using the judgement $\hat{\Gamma} \vdash_{\text{ws}} t \text{ expr} @ m$) and substitutions (defined using the judgement $\vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$).

incremented using a weakening substitution π (WSMTT-SUB-WEAKEN). This is similar to the representation of variables in a CwF. In the technical report on MTT [19], this standard practice is adapted to MTT with a variable rule that is a typed version of WSMTT-EXPR-VAR. The general variable rule TM-VAR (or its intrinsically scoped counterpart) remains derivable by explicitly substituting \mathbf{v}_0 with substitutions constructed via π , \mathbf{Q}^α (WSMTT-SUB-KEY) and $\cdot . \mathbf{\mu}$ (WSMTT-SUB-LOCK).

2.2.1 Substitution Calculus

In both [20, 19] and our presentation, MTT is a system with explicit substitution: applying a substitution to an expression is viewed as a syntax constructor (WSMTT-EXPR-SUB). This also means that expressions are defined mutually inductively with substitutions. For the latter, we introduce a judgement form $\vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ expressing that σ is a substitution from scoping context $\hat{\Gamma}$ to $\hat{\Delta}$ at mode m (again, the inference rules for this judgement *construct* WSMTT substitutions, rather than defining a well-scopedness relation over them).

Figure 4 shows all WSMTT substitution constructors. There is a unique substitution to the empty context (WSMTT-SUB-EMPTY) and identity (WSMTT-SUB-ID) and weakening (WSMTT-SUB-WEAKEN) substitutions. We can compose substitutions (WSMTT-SUB-COMPOSE, note that this is a constructor), lock them (WSMTT-SUB-LOCK) and extend them with a term to extend the codomain with a new variable (WSMTT-SUB-EXTEND). Note that this term has to live in a locked scoping context. Finally, every 2-cell in the mode theory gives rise to a key substitution (WSMTT-SUB-KEY). This last rule introduces the concept of lock telescopes: sequences of zero or more locks that have the right domain and codomain modes to be composed. A lock telescope $\Theta : \text{LockTele}(n \rightarrow m)$ can be applied to a scoping context at mode n to obtain a scoping context at mode m ; and similarly to a well-scoped substitution by iteratively applying WSMTT-SUB-LOCK. We can also compose all modalities in Θ to obtain a modality $\text{locks}(\Theta) : m \rightarrow n$. Precise definitions are given in Figure 3.

$$\begin{array}{c}
\text{SCTX-LOCK-ID} \\
\frac{\hat{\Gamma} \text{ sctx } @ m}{\hat{\Gamma} . \mathbf{\hat{\mu}}_{\mathbb{1}} = \hat{\Gamma} \text{ sctx } @ m}
\end{array}
\qquad
\begin{array}{c}
\text{SCTX-LOCK-COMP} \\
\frac{\hat{\Gamma} \text{ sctx } @ o \quad \mu : m \rightarrow n \quad \nu : n \rightarrow o}{\hat{\Gamma} . \mathbf{\hat{\mu}}_{\nu \circ \mu} = \hat{\Gamma} . \mathbf{\hat{\mu}}_{\nu} . \mathbf{\hat{\mu}}_{\mu} \text{ sctx } @ m}
\end{array}$$

■ **Figure 5** Strict functoriality of the lock operation on scoping contexts (optional).

2.2.2 Lock Telescopes vs. Strict Functoriality of Locks

The original presentation of MTT [20, 19] makes no mention of lock telescopes. Instead, it features strict functoriality rules for the lock operation on contexts, of which we give counterparts for scoping contexts in Figure 5. A consequence of these rules is that any lock telescope can be fused into a single lock.

It is however quite unusual to have a non-trivial equational theory on contexts and early explorations of a *lock calculus* for MTT [25] suggest that it may be advantageous to drop the functoriality rules; by `WSMTT-SUB-KEY` for the identity 2-cell, they automatically hold up to isomorphism. During the development of the current paper, we had a formulation of MTT in mind *without* these functoriality rules. However, nowhere in our constructions and proofs do we case distinguish on the number of locks in a given part of the context, or read off the modality annotation of a specific lock, so our results remain valid when we extend raw `WSMTT` with the rules in Figure 5.

► **Theorem 2** (Non-definability of composition). *Even with the rules in Figure 5, it is still not true that any `WSMTT` substitution can be alternatively constructed without using `WSMTT-SUB-COMPOSE` directly (i.e. not through a generalised rule). This remains impossible even if we use composition to generalise the rules `WSMTT-SUB-WEAKEN` and (jointly) `WSMTT-SUB-KEY` and `WSMTT-SUB-LOCK`, so that we would get $\pi(\sigma) := \sigma \circ \pi$ and $\sigma . \mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \Theta \Rightarrow \Psi} := \mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \Theta \Rightarrow \Psi} \circ (\sigma . \Psi)$ each by a single rule.*

Proof. Consider a mode theory with three modes p, q, r , three modalities $p \xrightarrow{\nu} q \xrightarrow{\mu} r \xleftarrow{\rho} p$ and a 2-cell $\alpha \in \mu \circ \nu \Rightarrow \rho$. Then we can consider the key substitution $\vdash_{\text{ws}} \mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \Theta \Rightarrow \Psi} \text{ sub}(\hat{\Gamma} . \mathbf{\hat{\mu}}_{\rho} \rightarrow \hat{\Gamma} . \mathbf{\hat{\mu}}_{\mu} . \mathbf{\hat{\mu}}_{\nu}) @ p$. Furthermore, given an expression t (e.g. `true`) in $\hat{\Gamma} . \mathbf{\hat{\mu}}_{\mu} . \mathbf{\hat{\mu}}_{\mathbb{1}}$ we can construct $\vdash_{\text{ws}} (\text{id}.t) . \mathbf{\hat{\mu}}_{\nu} \text{ sub}(\hat{\Gamma} . \mathbf{\hat{\mu}}_{\mu} . \mathbf{\hat{\mu}}_{\nu} \rightarrow \hat{\Gamma} . \mathbf{\hat{\mu}}_{\mu} . \mathbb{1} . \mathbf{\hat{\mu}}_{\nu}) @ p$. The composite of these two is a substitution from $\hat{\Gamma} . \mathbf{\hat{\mu}}_{\rho}$ to $\hat{\Gamma} . \mathbf{\hat{\mu}}_{\mu} . \mathbb{1} . \mathbf{\hat{\mu}}_{\nu}$, which both splits ρ into $\mu \circ \nu$ and extends the codomain with a variable. Assume we have an alternative substitution τ of the same domain and codomain, constructed without using `WSMTT-SUB-COMPOSE` directly. Since all remaining substitution constructors cause domain and codomain to be extended with locks and variables, travelling the derivation tree of τ upwards (constructed using the possibly generalised rules in Figure 4), we somehow need to peel off $\mathbf{\hat{\mu}}_{\nu}$ from the codomain, using the operation $\sigma . \mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \Theta \Rightarrow \Psi}$. This is impossible since $\rho : p \rightarrow r$ has no decomposition $p \rightarrow q \rightarrow r$.

An alternative proof can be given in the mode theory with α reversed, where the composite substitution combines α with a weakening in between the locks. ◀

2.3 σ -equivalence

Since substitution in `WSMTT` expressions is an explicit constructor, it does not compute (as will be the case in `SFMTT` in Section 3). This means that there are a lot of distinct `WSMTT` expressions that we would actually like to consider equivalent. For example, from the perspective of the rules in Figure 4 the expressions $t [\sigma]_{\text{ws}} [\tau]_{\text{ws}}$ and $t [\sigma \circ \tau]_{\text{ws}}$ have nothing to do with each other. For this reason, we add an axiomatic system to the intrinsically

$$\begin{array}{c}
\frac{\hat{\Xi} \vdash_{\text{ws}} t \text{ expr} @ m \quad \frac{\vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Delta} \rightarrow \hat{\Xi}) @ m \quad \vdash_{\text{ws}} \tau \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m}{\hat{\Gamma} \vdash_{\text{ws}} t [\sigma \circ \tau]_{\text{ws}} \equiv^{\sigma} t [\sigma]_{\text{ws}} [\tau]_{\text{ws}} \text{ expr} @ m} \quad \frac{\hat{\Delta} \vdash_{\text{ws}} t \equiv^{\sigma} s \text{ expr} @ m \quad \vdash_{\text{ws}} \tau \equiv^{\sigma} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m}{\hat{\Gamma} \vdash_{\text{ws}} t [\tau]_{\text{ws}} \equiv^{\sigma} s [\sigma]_{\text{ws}} \text{ expr} @ m}}{\hat{\Gamma} \vdash_{\text{ws}} t [\sigma \circ \tau]_{\text{ws}} \equiv^{\sigma} t [\sigma]_{\text{ws}} [\tau]_{\text{ws}} \text{ expr} @ m} \quad \frac{\mu : m \rightarrow n \quad \hat{\Delta} . \mu \vdash_{\text{ws}} t \text{ expr} @ n \quad \vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n}{\hat{\Gamma} \vdash_{\text{ws}} (\lambda^{\mu}(t)) [\sigma]_{\text{ws}} \equiv^{\sigma} \lambda^{\mu}(t [\sigma^+]_{\text{ws}}) \text{ expr} @ n} \quad \text{with } \sigma^+ = (\sigma \circ \pi) . \mathbf{v}_0 \\
\frac{\frac{\frac{\vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Delta} \rightarrow \hat{\Xi}) @ m \quad \vdash_{\text{ws}} \tau \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m}{\vdash_{\text{ws}} (\sigma \circ \tau) . \mathbf{\mu} \equiv^{\sigma} (\sigma . \mathbf{\mu}) \circ (\tau . \mathbf{\mu}) \text{ sub}(\hat{\Gamma} . \mathbf{\mu} \rightarrow \hat{\Xi} . \mathbf{\mu}) @ n} \quad \frac{\hat{\Gamma} \text{ sctx} @ n \quad \Lambda : \text{LockTele}(n \rightarrow m)}{\vdash_{\text{ws}} \mathbf{q}_{\hat{\Gamma}}^{1_{\text{locks}(\Lambda)} \in \Lambda \Rightarrow \Lambda} \equiv^{\sigma} \text{id sub}(\hat{\Gamma} . \Lambda \rightarrow \hat{\Gamma} . \Lambda) @ m}}{\vdash_{\text{ws}} \mathbf{q}_{\hat{\Gamma}}^{\beta \circ \alpha \in \Lambda \Rightarrow \Psi} \equiv^{\sigma} \mathbf{q}_{\hat{\Gamma}}^{\alpha \in \Lambda \Rightarrow \Theta} \circ \mathbf{q}_{\hat{\Gamma}}^{\beta \in \Theta \Rightarrow \Psi} \text{ sub}(\hat{\Gamma} . \Psi \rightarrow \hat{\Gamma} . \Lambda) @ m}}{\vdash_{\text{ws}} \mathbf{q}_{\hat{\Delta}}^{\alpha \in \Lambda \Rightarrow \Theta} \circ (\sigma . \Theta) \equiv^{\sigma} (\sigma . \Lambda) \circ \mathbf{q}_{\hat{\Gamma}}^{\alpha \in \Lambda \Rightarrow \Theta} \text{ sub}(\hat{\Gamma} . \Theta \rightarrow \hat{\Delta} . \Lambda) @ m} \\
\frac{\alpha \in \text{locks}(\Lambda) \Rightarrow \text{locks}(\Theta) \quad \beta \in \text{locks}(\Theta) \Rightarrow \text{locks}(\Psi)}{\vdash_{\text{ws}} \mathbf{q}_{\hat{\Gamma}}^{\beta \circ \alpha \in \Lambda \Rightarrow \Psi} \equiv^{\sigma} \mathbf{q}_{\hat{\Gamma}}^{\alpha \in \Lambda \Rightarrow \Theta} \circ \mathbf{q}_{\hat{\Gamma}}^{\beta \in \Theta \Rightarrow \Psi} \text{ sub}(\hat{\Gamma} . \Psi \rightarrow \hat{\Gamma} . \Lambda) @ m} \\
\frac{\alpha \in \text{locks}(\Lambda) \Rightarrow \text{locks}(\Theta) \quad \vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n}{\vdash_{\text{ws}} \mathbf{q}_{\hat{\Delta}}^{\alpha \in \Lambda \Rightarrow \Theta} \circ (\sigma . \Theta) \equiv^{\sigma} (\sigma . \Lambda) \circ \mathbf{q}_{\hat{\Gamma}}^{\alpha \in \Lambda \Rightarrow \Theta} \text{ sub}(\hat{\Gamma} . \Theta \rightarrow \hat{\Delta} . \Lambda) @ m}
\end{array}$$

■ **Figure 6** Selected rules for σ -equivalence in WSMTT.

scoped WSMTT syntax that specifies when two expressions or substitutions are σ -equivalent (note that we do not add β - or η -equivalence to this system yet, those should be covered in the type system that would be defined on top of the syntax described in this paper).

Some of the rules for σ -equivalence can be found in Figure 6. We make use of a judgement $\hat{\Gamma} \vdash_{\text{ws}} t \equiv^{\sigma} s \text{ expr} @ m$ for expressions and $\vdash_{\text{ws}} \sigma \equiv^{\sigma} \tau \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ for substitutions. We find rules expressing the connection between applying a composed substitution and consecutively applying both substitutions, expressing how to push a substitution through expression constructors such as λ^{μ} (here σ^+ is the lifting of σ defined as $\sigma^+ = (\sigma \circ \pi) . \mathbf{v}_0$) and expressing functoriality of locks on substitutions. There are also quite some rules that express properties of key substitutions: their naturality and their behaviour with respect to the unit 2-cell and composition of 2-cells. The full definition of σ -equivalence for WSMTT can be found in the technical report [13].

3 Substitution Algorithm

In this section we describe our substitution algorithm for MTT. For this purpose we introduce a new language called SFMTT (for substitution-free MTT), which has no expression constructor for substitutions like `wsmtt-expr-sub` in Figure 4. We also introduce renamings and substitutions for SFMTT. All of this is included in Section 3.1. We then proceed in Section 3.2 to the core part of the substitution algorithm: applying SFMTT renamings and substitutions to SFMTT expressions. Finally, using this functionality we can translate WSMTT expressions to SFMTT expressions.

3.1 Substitution-free Multimode Type Theory (SFMTT)

3.1.1 SFMTT Expressions

Exactly like our presentation of WSMTT, the expressions in SFMTT will be extrinsically typed but intrinsically scoped. We can reuse the same notion of scoping context and lock telescope from Figure 3. However, as indicated in Section 2.2, the WSMTT representation

$$\begin{array}{c}
\text{SF-VAR-ZERO} \\
\frac{\Theta : \text{LockTele}(n \rightarrow m) \quad \mu : m \rightarrow n \quad \hat{\Gamma} \text{ sctx} @ n}{\hat{\Gamma} . \mu . \Theta \vdash_{\text{sf}} \mathbf{v}_0^\alpha \text{ var} @ m}
\end{array}
\qquad
\begin{array}{c}
\text{SF-VAR-SUC} \\
\frac{\Theta : \text{LockTele}(n \rightarrow m) \quad \mu : o \rightarrow n \quad \hat{\Gamma} . \Theta \vdash_{\text{sf}} v \text{ var} @ m}{\hat{\Gamma} . \mu . \Theta \vdash_{\text{sf}} \text{ suc}(v) \text{ var} @ m}
\end{array}$$

■ **Figure 7** Constructors for well-scoped SFMTT variables.

of variables makes use of explicit substitutions, which we do not have in SFMTT. For this reason, SFMTT has a dedicated variable judgement $\hat{\Gamma} \vdash_{\text{sf}} v \text{ var} @ m$ introducing the syntactic category of accessible variables v in scoping context $\hat{\Gamma}$ at mode m . The inference rules for this judgement can be found in Figure 7. Either we access the last variable in the scoping context, in which case we have to provide an appropriate 2-cell (SF-VAR-ZERO), or we skip the last variable in the scoping context, which may be located under a lock telescope (SF-VAR-SUC). As a conclusion, an SFMTT variable is of the form $\text{suc}^n(\mathbf{v}_0^\alpha)$, so it is just a De Bruijn index with a 2-cell annotation.

Similar to WSMTT, SFMTT expressions can now be introduced via a judgement $\hat{\Gamma} \vdash_{\text{sf}} t \text{ expr} @ m$. The constructors are the same as those for WSMTT in Figure 4, except for WSMTT-EXPR-VAR and WSMTT-EXPR-SUB, which are not included. Instead, there is a constructor promoting any variable $\hat{\Gamma} \vdash_{\text{sf}} v \text{ var} @ m$ to an SFMTT expression in $\hat{\Gamma}$. We emphasize that SFMTT expressions cannot contain substitutions.

3.1.2 SFMTT Renamings and Substitutions

We can also define substitutions for the SFMTT syntax, which will be required in the next section. As in our intrinsically scoped presentation of WSMTT, every SFMTT renaming and substitution has a domain and a codomain scoping context. This ensures that applying a renaming or substitution to an SFMTT expression is a total (always defined) operation.

Similar to McBride [23] and Allais et al. [4], we define an action of renaming on expressions before we discuss the action of substitutions. Such a renaming does not only allow us to lift a substitution when pushing it under a binder, but also to perform some modal operations. Of course, we have to take into account that we want a structurally recursive substitution algorithm, which is impossible when substitution composition is added as a constructor. We solve this problem by first defining atomic renamings and substitutions, which are not closed under composition but which can be applied to SFMTT expressions in a structurally recursive way. Regular renamings and substitutions (from now on also referred to as rensubs) will be defined in terms of these atomic rensubs.

Just like substitutions in WSMTT, atomic renamings and substitutions are defined using a judgement $\vdash_{\text{sf}} \sigma \text{ aren/asub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ (much of the structure between renamings and substitutions is shared). There is a similar judgement $\vdash_{\text{sf}} \sigma \text{ ren/sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ for regular rensubs. The constructors for atomic rensubs can be found in Figure 8. Many of them are similar to the ones for WSMTT substitutions, such as the empty atomic rensb (SF-ARENSUB-EMPTY), locking (SF-ARENSUB-LOCK) and keys (SF-ARENSUB-KEY). As explained, we purposely omit a constructor for composition of atomic rensubs. As a consequence, we need a constructor for weakening rensubs (SF-ARENSUB-WEAKEN) which in WSMTT would have been accomplished by precomposing with π . Also note that we have an atomic identity rensb id^a (SF-ARENSUB-ID). We could have alternatively implemented id^a in terms of the other constructors but taking it as a constructor will make the rest of the paper easier because we can *define* its action on expressions to be trivial, whereas otherwise that would require a

$$\begin{array}{c}
\text{SF-ARENSUB-EMPTY} \\
\hline
\vdash_{\text{sf}} ! \text{aren/asub}(\hat{\Gamma} \rightarrow \cdot) @ m \\
\\
\text{SF-ARENSUB-WEAKEN} \\
\hline
\vdash_{\text{sf}} \sigma \text{aren/asub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m \\
\vdash_{\text{sf}} \text{weaken}(\sigma) \text{aren/asub}(\hat{\Gamma} . \mu \rightarrow \hat{\Delta}) @ m \\
\\
\text{SF-ARENSUB-KEY} \\
\hline
\Theta, \Psi : \text{LockTele}(n \rightarrow m) \quad \alpha \in \text{locks}(\Theta) \Rightarrow \text{locks}(\Psi) \\
\vdash_{\text{sf}} \mathfrak{Q}_{\hat{\Gamma}}^{\alpha \in \Theta \Rightarrow \Psi} \text{aren/asub}(\hat{\Gamma} . \Psi \rightarrow \hat{\Gamma} . \Theta) @ m \\
\\
\text{SF-AREN-EXTEND} \\
\hline
\mu : m \rightarrow n \quad \vdash_{\text{sf}} \sigma \text{aren}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n \\
\hat{\Gamma} . \mathfrak{L}_{\mu} \vdash_{\text{sf}} v \text{var} @ m \\
\hline
\vdash_{\text{sf}} \sigma.v \text{aren}(\hat{\Gamma} \rightarrow \hat{\Delta} . \mu) @ n \\
\\
\text{SF-ASUB-EXTEND} \\
\hline
\mu : m \rightarrow n \quad \vdash_{\text{sf}} \sigma \text{asub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n \\
\hat{\Gamma} . \mathfrak{L}_{\mu} \vdash_{\text{sf}} t \text{expr} @ m \\
\hline
\vdash_{\text{sf}} \sigma.t \text{asub}(\hat{\Gamma} \rightarrow \hat{\Delta} . \mu) @ n
\end{array}$$

■ **Figure 8** Constructors for atomic SFMTT renamings and substitutions.

$$\begin{array}{c}
\text{SF-RENSUB-ID} \\
\hline
\hat{\Gamma} \text{ sctx} @ m \\
\hline
\vdash_{\text{sf}} \text{id} \text{ren/sub}(\hat{\Gamma} \rightarrow \hat{\Gamma}) @ m \\
\\
\text{SF-RENSUB-SNOC} \\
\hline
\vdash_{\text{sf}} \sigma \text{ren/sub}(\hat{\Delta} \rightarrow \hat{\Xi}) @ m \quad \vdash_{\text{sf}} \tau \text{aren/asub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m \\
\hline
\vdash_{\text{sf}} \sigma \otimes \tau \text{ren/sub}(\hat{\Gamma} \rightarrow \hat{\Xi}) @ m
\end{array}$$

■ **Figure 9** Constructors for regular SFMTT renamings and substitutions.

non-trivial proof. The only difference between atomic renamings and substitutions is the way they can be extended: a renaming is extended with a variable (SF-AREN-EXTEND) whereas a substitution can be extended with an arbitrary SFMTT expression (SF-ASUB-EXTEND).

Figure 9 shows the full definition of regular rensups. In essence, a rensup is well-scoped snoc-lists of atomic substitutions. It can be empty, so it is actually the identity (SF-RENSUB-ID), or it consists of an atomic rensup postcomposed with a regular rensup (SF-RENSUB-SNOC).

One operation that we will need in the next section, is the lifting of atomic rensups. Given an atomic rensup σ from $\hat{\Gamma}$ to $\hat{\Delta}$, we can construct a new, lifted atomic rensup $\sigma^+ := \text{weaken}(\sigma).v_0^{1\mu}$ from $\hat{\Gamma} . \mu$ to $\hat{\Delta} . \mu$ (here $v_0^{1\mu}$ is interpreted as a variable in the case of renamings and as an expression in the case of substitutions).⁴ Moreover, for any scoping context $\hat{\Gamma}$ and modality μ , we have a weakening atomic rensup $\pi := \text{weaken}(\text{id}^a)$ from $\hat{\Gamma} . \mu$ to $\hat{\Gamma}$. The lift and lock operations can be extended to regular rensups by applying those operations to all constituent atomic rensups. In other words, we have $\text{id}^+ = \text{id}$, $(\sigma \otimes \tau)^+ = \sigma^+ \otimes \tau^+$, $\text{id} . \mathfrak{L}_{\mu} = \text{id}$ and $(\sigma \otimes \tau) . \mathfrak{L}_{\mu} = (\sigma . \mathfrak{L}_{\mu}) \otimes (\tau . \mathfrak{L}_{\mu})$.

3.2 Renaming and Substitution Algorithm for SFMTT

We are now ready to describe one of the core parts of the paper: the algorithm for applying an SFMTT substitution to an SFMTT expression. The definition is built up in 4 steps, each defining the action of another class of syntactic objects on SFMTT expressions: 1. atomic renamings, 2. regular renamings, 3. atomic substitutions, and 4. regular substitutions.

⁴ It might be surprising that this works for substitutions too, since we explained in the introduction for STLC that defining weakening for substitutions requires recursively applying a substitution (or renaming) to terms in the context. However, substituting a variable with $\text{weaken}(\sigma)$ will involve the application of a renaming, as we will see in the next section.

However, there is considerable overlap between some of these steps. For this reason, we will treat steps 2 and 4 together as well as large parts of steps 1 and 3.⁵ All operations take an (atomic) rensusb from $\hat{\Gamma}$ to $\hat{\Delta}$ and an SFMTT expression in scoping context $\hat{\Delta}$ to produce an SFMTT expression in scoping context $\hat{\Gamma}$.

3.2.1 Atomic rensubs acting on non-variable expressions

We first discuss the application of atomic rensubs on SFMTT expressions other than variables. In general, if we have an atomic rensusb $\vdash_{\text{sf}} \sigma \text{ aren/asub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ and an SFMTT expression $\hat{\Delta} \vdash_{\text{sf}} t \text{ expr} @ m$ (other than a variable), we describe how to construct an expression $\hat{\Gamma} \vdash_{\text{sf}} t [\sigma]_{\text{aren/asub}} \text{ expr} @ m$. Note that in order for this construction to be well-defined on intrinsically-scoped syntax, one must verify that it does indeed preserve well-scopedness.

$$\begin{aligned} \langle \mu \mid A \rangle [\sigma]_{\text{aren/asub}} &= \langle \mu \mid A [\sigma \cdot \mathbf{\mu}]_{\text{aren/asub}} \rangle \\ \text{mod}_{\mu}(t) [\sigma]_{\text{aren/asub}} &= \text{mod}_{\mu}(t [\sigma \cdot \mathbf{\mu}]_{\text{aren/asub}}) \\ ((\mu \mid A) \rightarrow B) [\sigma]_{\text{aren/asub}} &= (\mu \mid A [\sigma \cdot \mathbf{\mu}]_{\text{aren/asub}}) \rightarrow B [\sigma^+]_{\text{aren/asub}} \\ (\lambda^{\mu}(t)) [\sigma]_{\text{aren/asub}} &= \lambda^{\mu}(t [\sigma^+]_{\text{aren/asub}}) \\ \text{app}_{\mu}(f; t) [\sigma]_{\text{aren/asub}} &= \text{app}_{\mu}(f [\sigma]_{\text{aren/asub}}; t [\sigma \cdot \mathbf{\mu}]_{\text{aren/asub}}) \end{aligned}$$

3.2.2 Atomic renamings acting on variables

We now turn to the case for variables. This is where we distinguish between atomic renamings and atomic substitutions. We first discuss the action of an atomic renaming on a variable, producing another variable. The intuitive “type signature” of this operation is too weak to make recursion work. In particular, it does not allow us to go under locks in renamings. Therefore, we have a result that generalises over a lock telescope Λ , but we can recover the desired result by taking the empty lock telescope for Λ . Note that for the remainder of Section 3 we will state the signatures of the different parts of the substitution algorithm in lemmas and theorems. The actual description of the algorithm can be found in the corresponding proofs (which we will call constructions to make it clear that they contain computationally interesting content).

► **Lemma 3.** *If we have an SFMTT atomic renaming $\vdash_{\text{sf}} \sigma \text{ aren}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n$, a lock telescope $\Lambda : \text{LockTele}(n \rightarrow m)$ and an SFMTT variable $\hat{\Delta} \cdot \Lambda \vdash_{\text{sf}} v \text{ var} @ m$, then we can construct a variable $\hat{\Gamma} \cdot \Lambda \vdash_{\text{sf}} v [\sigma]_{\text{aren,var}}^{\Lambda} \text{ var} @ m$*

Lemma 3 is a core lemma for this paper. Our substitution algorithm crucially relies on identifying a notion of renamings that can be recursively applied to MTT terms. It is this lemma that establishes that our choices achieve this and we include the construction below because it clarifies well why atomic renamings should be defined as they are.

In the construction for Lemma 3 we will make use of the following result.

► **Lemma 4.** *Let $\Theta, \Psi : \text{LockTele}(n \rightarrow m)$ be two lock telescopes and $\alpha \in \text{locks}(\Theta) \Rightarrow \text{locks}(\Psi)$ a 2-cell. Then we can transform a variable $\hat{\Gamma} \cdot \Theta \vdash_{\text{sf}} v \text{ var} @ m$ to a variable $\hat{\Gamma} \cdot \Psi \vdash_{\text{sf}} v [\alpha]_{2\text{-cell}}^{\Theta \Rightarrow \Psi} \text{ var} @ m$.*

⁵ In fact, the action of regular renamings is not really used anywhere. Only atomic renamings will be important. However, as already mentioned the treatment of regular renamings and regular substitutions is entirely the same.

Construction. We proceed by structural recursion on the variable v (i.e. the annotated De Bruijn index).

- CASE $\hat{\Gamma} . \Theta \vdash_{\text{sf}} \mathbf{v}_0^\beta \text{ var } @ m$ with $\hat{\Gamma} = \hat{\Delta} . \mu . \Lambda$ (SF-VAR-ZERO, Λ is a lock telescope so it only contains locks)
 We know that $\hat{\Delta} . \mu . \Lambda . \Theta \vdash_{\text{sf}} \mathbf{v}_0^\beta \text{ var } @ m$, so $\beta \in \mu \Rightarrow \text{locks}(\Lambda . \Theta) = \text{locks}(\Lambda) \circ \text{locks}(\Theta)$. Using the horizontal composition \star , we can construct a 2-cell $1_{\text{locks}(\Lambda)} \star \alpha \in \text{locks}(\Lambda) \circ \text{locks}(\Theta) \Rightarrow \text{locks}(\Lambda) \circ \text{locks}(\Psi)$. Hence we use the rule SF-VAR-ZERO again to obtain $\mathbf{v}_0^\beta [\alpha]_{2\text{-cell}}^{\Theta \Rightarrow \Psi} = \mathbf{v}_0^{(1_{\text{locks}(\Lambda)} \star \alpha) \circ \beta}$.⁶
- CASE $\hat{\Gamma} . \Theta \vdash_{\text{sf}} \text{ suc } (v) \text{ var } @ m$ with $\hat{\Gamma} = \hat{\Delta} . \mu . \Lambda$ (SF-VAR-SUC, Λ is a lock telescope)
 In this case we have that $\hat{\Delta} . \Lambda . \Theta \vdash_{\text{sf}} v \text{ var } @ m$. By recursion we then obtain a variable $\hat{\Delta} . \Lambda . \Psi \vdash_{\text{sf}} v [\alpha]_{2\text{-cell}}^{\Theta \Rightarrow \Psi} \text{ var } @ m$. Applying the rule SF-VAR-SUC again to this result gives us the desired variable, so $\text{ suc } (v) [\alpha]_{2\text{-cell}}^{\Theta \Rightarrow \Psi} = \text{ suc } \left(v [\alpha]_{2\text{-cell}}^{\Theta \Rightarrow \Psi} \right)$. ◀

Construction for Lemma 3. We proceed by structural recursion on σ .

- CASE $\vdash_{\text{sf}} ! \text{ aren}(\hat{\Gamma} \rightarrow \cdot) @ n$
 In this case, $\hat{\Delta}$ is the empty scoping context. We can see from Figure 7 that there can be no variables in the empty scoping context (the scoping contexts in conclusions of both inference rules both contain at least a variable annotation). Hence we do not have to deal with this case further.⁷
- CASE $\vdash_{\text{sf}} \text{ id}^a \text{ aren}(\hat{\Gamma} \rightarrow \hat{\Gamma}) @ n$
 Now $\hat{\Gamma} . \Lambda \vdash_{\text{sf}} v \text{ var } @ m$, so we can just say $v [\text{ id}^a]_{\text{aren, var}}^\Lambda = v$.
- CASE $\vdash_{\text{sf}} \text{ weaken}(\sigma) \text{ aren}(\hat{\Gamma} . \mu \rightarrow \hat{\Delta}) @ n$
 We know that $\hat{\Delta} . \Lambda \vdash_{\text{sf}} v \text{ var } @ m$, so we can use recursion for σ and obtain a variable $\hat{\Gamma} . \Lambda \vdash_{\text{sf}} v [\sigma]_{\text{aren, var}}^\Lambda \text{ var } @ m$. Since Λ is a lock telescope not containing variable annotations, we can then apply the rule SF-VAR-SUC from Figure 7 with $\Theta = \Lambda$ to obtain a variable in $\hat{\Gamma} . \mu . \Lambda$ as required. In other words, $v [\text{ weaken}(\sigma)]_{\text{aren, var}}^\Lambda = \text{ suc } \left(v [\sigma]_{\text{aren, var}}^\Lambda \right)$.
- CASE $\vdash_{\text{sf}} \sigma . \blacksquare_\mu \text{ aren}(\hat{\Gamma} . \blacksquare_\mu \rightarrow \hat{\Delta} . \blacksquare_\mu) @ n$
 Adding the \blacksquare_μ to the left of the lock telescope Λ , we get $v [\sigma . \blacksquare_\mu]_{\text{aren, var}}^\Lambda = v [\sigma]_{\text{aren, var}}^{\blacksquare_\mu . \Lambda}$.
- CASE $\vdash_{\text{sf}} \mathcal{Q}_{\hat{\Gamma}}^{\beta \in \Theta \Rightarrow \Psi} \text{ aren}(\hat{\Gamma} . \Psi \rightarrow \hat{\Gamma} . \Theta) @ n$
 We have that $\hat{\Gamma} . \Theta . \Lambda \vdash_{\text{sf}} v \text{ var } @ m$ and that $\beta \in \text{locks}(\Theta) \Rightarrow \text{locks}(\Psi)$. This means that $\beta \star 1_{\text{locks}(\Lambda)} \in \text{locks}(\Theta . \Lambda) \Rightarrow \text{locks}(\Psi . \Lambda)$. Using Lemma 4, we can use this 2-cell to obtain a variable in $\hat{\Gamma} . \Psi . \Lambda$, so $v [\mathcal{Q}_{\hat{\Gamma}}^{\beta \in \Theta \Rightarrow \Psi}]_{\text{aren, var}}^\Lambda = v [\beta \star 1_{\text{locks}(\Lambda)}]_{2\text{-cell}}^{\Theta . \Lambda \Rightarrow \Psi . \Lambda}$.
- CASE $\vdash_{\text{sf}} \sigma . w \text{ aren}(\hat{\Gamma} \rightarrow \hat{\Delta} . \mu) @ n$
 We know that $\hat{\Delta} . \mu . \Lambda \vdash_{\text{sf}} v \text{ var } @ m$ (where Λ contains only locks) and perform a case split on v .
 - CASE $\hat{\Delta} . \mu . \Lambda \vdash_{\text{sf}} \mathbf{v}_0^\alpha \text{ var } @ m$
 In this case we have a 2-cell $\alpha \in \mu \Rightarrow \text{locks}(\Lambda)$. Moreover, as one of the premises of SF-AREN-EXTEND we know that $\hat{\Gamma} . \blacksquare_\mu \vdash_{\text{sf}} w \text{ var } @ m$. We can then use Lemma 4 with lock telescopes $\Theta = \blacksquare_\mu$ and $\Psi = \Lambda$ to transform w to a variable in $\hat{\Gamma} . \Lambda$. In other words, $\mathbf{v}_0^\alpha [\sigma . w]_{\text{aren, var}}^\Lambda = w [\alpha]_{2\text{-cell}}^{\blacksquare_\mu \Rightarrow \Lambda}$.

⁶ This definition seems to imply a dependency of $\mathbf{v}_0^\beta [\alpha]_{2\text{-cell}}^{\Theta \Rightarrow \Psi}$ on Λ , but note that Λ is completely determined by the scoping context and the variable.

⁷ This case illustrates why it is advantageous to use intrinsically scoped syntax. It makes sure that the codomain of the renaming and the scoping context of the expression match, so we do not have to cover insensible cases.

- CASE $\hat{\Delta}. \mu. \Lambda \vdash_{\text{sf}} \text{suc}(v) \text{ var} @ m$
 Now we know that $\hat{\Delta}. \Lambda \vdash_{\text{sf}} v \text{ var} @ m$ and that $\vdash_{\text{sf}} \sigma \text{ aren}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n$. Consequently, we can recursively obtain a variable in $\hat{\Gamma}. \Lambda$, so $\text{suc}(v) [\sigma.w]_{\text{aren}, \text{var}}^{\Lambda} = v [\sigma]_{\text{aren}, \text{var}}^{\Lambda}$. ◀

Note that the algorithm presented in the construction for Lemma 3 is indeed structurally recursive: in every recursive call the renaming gets structurally smaller (and moreover the algorithm in the construction for Lemma 4 does not depend on that of Lemma 3).

Together with the equations from Section 3.2.1, we get the following result.

► **Lemma 5** (Atomic renaming for SFMTT expressions). *If $\vdash_{\text{sf}} \sigma \text{ aren}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ and $\hat{\Delta} \vdash_{\text{sf}} t \text{ expr} @ m$, then we can construct $\hat{\Gamma} \vdash_{\text{sf}} t [\sigma]_{\text{aren}} \text{ expr} @ m$.*

3.2.3 Atomic substitutions acting on variables

We now describe the action of atomic substitutions on variables. This will produce an SFMTT expression that is not necessarily a variable anymore (as was the case for atomic renamings). We have a result very similar to Lemma 3.

► **Lemma 6.** *If we have an SFMTT atomic substitution $\vdash_{\text{sf}} \sigma \text{ asub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n$, a lock telescope $\Lambda : \text{LockTele}(n \rightarrow m)$ and an SFMTT variable $\hat{\Delta}. \Lambda \vdash_{\text{sf}} v \text{ var} @ m$, then we can construct an expression $\hat{\Gamma}. \Lambda \vdash_{\text{sf}} v [\sigma]_{\text{asub}, \text{var}}^{\Lambda} \text{ expr} @ m$.*

Construction. Again we proceed by case distinction and recursion over σ . The cases for $!$, id^a , σ , \blacksquare_{μ} and $\mathfrak{Q}_{\hat{\Gamma}}^{\beta \in \Theta \Rightarrow \Psi}$ are similar to the construction for Lemma 3 so we omit them.

- CASE $\vdash_{\text{sf}} \text{weaken}(\sigma) \text{ asub}(\hat{\Gamma}. \mu \rightarrow \hat{\Delta}) @ n$
 We have that $\hat{\Delta}. \Lambda \vdash_{\text{sf}} v \text{ var} @ m$ and $\vdash_{\text{sf}} \sigma \text{ asub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n$, so we can use recursion to obtain an expression in $\hat{\Gamma}. \Lambda$. Then we can apply Lemma 5 with the atomic renaming $\pi. \Lambda$ (i.e. applying all the locks from Λ to π) to obtain an expression in $\hat{\Gamma}. \mu. \Lambda$ as required. Consequently, we have $v [\text{weaken}(\sigma)]_{\text{asub}, \text{var}}^{\Lambda} = \left(v [\sigma]_{\text{asub}, \text{var}}^{\Lambda} \right) [\pi. \Lambda]_{\text{aren}}$.
- CASE $\vdash_{\text{sf}} \sigma.t \text{ asub}(\hat{\Gamma} \rightarrow \hat{\Delta}. \mu) @ n$
 We know that $\hat{\Delta}. \mu. \Lambda \vdash_{\text{sf}} v \text{ var} @ m$ and perform a case split on v .
 - CASE $\hat{\Delta}. \mu. \Lambda \vdash_{\text{sf}} \mathbf{v}_0^{\alpha} \text{ var} @ m$
 In this case $\alpha \in \mu \Rightarrow \text{locks}(\Lambda)$ and $\hat{\Gamma}. \blacksquare_{\mu} \vdash_{\text{sf}} t \text{ expr} @ m$. Therefore, we can apply Lemma 5 with the renaming $\mathfrak{Q}_{\hat{\Gamma}}^{\alpha \in \blacksquare_{\mu} \Rightarrow \Lambda}$ and the expression t to obtain an expression in $\hat{\Gamma}. \Lambda$. In other words $\mathbf{v}_0^{\alpha} [\sigma.t]_{\text{asub}, \text{var}}^{\Lambda} = t \left[\mathfrak{Q}_{\hat{\Gamma}}^{\alpha \in \blacksquare_{\mu} \Rightarrow \Lambda} \right]_{\text{aren}}$.
 - CASE $\hat{\Delta}. \mu. \Lambda \vdash_{\text{sf}} \text{suc}(v) \text{ var} @ m$
 Now $\hat{\Delta}. \Lambda \vdash_{\text{sf}} v \text{ var} @ m$, so we can apply recursion (with both the variable and the substitution getting structurally smaller) to get $\text{suc}(v) [\sigma.t]_{\text{asub}, \text{var}}^{\Lambda} = v [\sigma]_{\text{asub}, \text{var}}^{\Lambda}$. ◀

Note that all of the cases in the previous construction are similar to the corresponding cases in the construction for Lemma 3. The most important difference is that the result of applying a substitution is an expression and not a variable. To transform the results from recursive calls, we therefore make use of the fact that atomic renamings act on expressions as shown in Lemma 5 (unlike the direct manipulation of variables as in the construction for Lemma 3). This is reminiscent of how renaming gets used in the definition of substitution in [23, 4].

Combining the previous result with the equations in Section 3.2.1, we get the following.

► **Lemma 7** (Atomic substitution for SFMTT expressions). *If $\vdash_{\text{sf}} \sigma \text{ asub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ and $\hat{\Delta} \vdash_{\text{sf}} t \text{ expr} @ m$, then we can construct $\hat{\Gamma} \vdash_{\text{sf}} t [\sigma]_{\text{asub}} \text{ expr} @ m$.*

► **Example 8.** In order to illustrate the substitution algorithm described in this section, we reconsider the MTT function f from Example 1. In the MTT context $\Gamma . (\mathbb{1} \mid y : (\rho \mid B) \rightarrow A) . (\rho \mid z : B) . \mathbf{a}_{\mathbb{1}}$ we can construct a term $t = \text{app}_{\rho} (y^{1_1}; z^{1_{\rho}})$ of type A to which f can be applied. Although neither SFMTT nor WSMTT have built-in β -equivalence, we can still construct the term to which this application of f should reduce. First of all, the SFMTT version of the function f is $\lambda^1 (\text{mod}_{\rho} (\text{mod}_{\mu} (\mathbf{v}_0^{\eta})))$ and t becomes $\text{app}_{\rho} (\text{suc} (\mathbf{v}_0^{1_1}); \mathbf{v}_0^{1_{\rho}})$. If we write π^2 for $\text{weaken}(\pi)$, then we can compute the desired term as follows

$$\begin{aligned}
& (\text{mod}_{\rho} (\text{mod}_{\mu} (\mathbf{v}_0^{\eta}))) \left[\pi^2 . \text{app}_{\rho} (\text{suc} (\mathbf{v}_0^{1_1}); \mathbf{v}_0^{1_{\rho}}) \right]_{\text{asub}} \\
&= \text{mod}_{\rho} \left(\text{mod}_{\mu} \left(\mathbf{v}_0^{\eta} \left[\left(\pi^2 . \text{app}_{\rho} (\text{suc} (\mathbf{v}_0^{1_1}); \mathbf{v}_0^{1_{\rho}}) \right) . \mathbf{a}_{\rho} . \mathbf{a}_{\mu} \right]_{\text{asub, var}} \right) \right) \\
&= \text{mod}_{\rho} \left(\text{mod}_{\mu} \left(\mathbf{v}_0^{\eta} \left[\pi^2 . \text{app}_{\rho} (\text{suc} (\mathbf{v}_0^{1_1}); \mathbf{v}_0^{1_{\rho}}) \right]_{\text{asub, var}}^{\mathbf{a}_{\rho} . \mathbf{a}_{\mu}} \right) \right) \\
&= \text{mod}_{\rho} \left(\text{mod}_{\mu} \left(\left(\text{app}_{\rho} (\text{suc} (\mathbf{v}_0^{1_1}); \mathbf{v}_0^{1_{\rho}}) \right) \left[\mathbf{a}_{\hat{\Gamma} . \mathbb{1} . \rho}^{\eta \in \mathbf{a}_{\mathbb{1}} \Rightarrow \mathbf{a}_{\rho} . \mathbf{a}_{\mu}} \right]_{\text{aren}} \right) \right) \\
&= \text{mod}_{\rho} \left(\text{mod}_{\mu} \left(\text{app}_{\rho} (\text{suc} (\mathbf{v}_0^{1_1} \left[\mathbf{a}_{\hat{\Gamma} . \mathbb{1} . \rho}^{\eta \in \mathbf{a}_{\mathbb{1}} \Rightarrow \mathbf{a}_{\rho} . \mathbf{a}_{\mu}} \right]_{\text{aren, var}}); \mathbf{v}_0^{1_{\rho}} \left[\mathbf{a}_{\hat{\Gamma} . \mathbb{1} . \rho}^{\eta \in \mathbf{a}_{\mathbb{1}} \Rightarrow \mathbf{a}_{\rho} . \mathbf{a}_{\mu}} \right]_{\text{aren, var}}^{\mathbf{a}_{\rho}} \right) \right) \right) \\
&= \text{mod}_{\rho} \left(\text{mod}_{\mu} \left(\text{app}_{\rho} (\text{suc} (\mathbf{v}_0^{\eta}); \mathbf{v}_0^{\eta * 1_{\rho}}) \right) \right).
\end{aligned}$$

3.2.4 Regular renamings/substitutions

We now turn to regular renamings and substitutions. There is no need to distinguish between these two as the procedure for renamings and substitutions will be exactly the same. Since a regular rensb is a sequence of atomic rensbs, we can just sequentially apply the results from the previous sections. We therefore get the following.

$$t [\text{id}]_{\text{ren/sub}} = t \qquad t [\sigma \circledast \tau]_{\text{ren/sub}} = \left(t [\sigma]_{\text{ren/sub}} \right) [\tau]_{\text{aren/asub}}$$

As a conclusion, we have finished the algorithm for renaming and substitution in SFMTT.

► **Theorem 9** (Renaming and substitution for SFMTT expressions). *Given a renaming or substitution $\vdash_{\text{sf}} \sigma \text{ ren/sub} (\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ and an SFMTT expression $\hat{\Delta} \vdash_{\text{sf}} t \text{ expr} @ m$, we can construct an expression $\hat{\Gamma} \vdash_{\text{sf}} t [\sigma]_{\text{ren/sub}} \text{ expr} @ m$.*

Note that we do not actually need the action of full renamings on SFMTT expressions in order to define the action of atomic substitutions, atomic renamings suffice for that purpose.

Although we are not really concerned with performance in this paper, we note that optimisations are certainly possible. For example, as it is currently described, the algorithm will, when applying a regular substitution consisting of n atomic ones to an expression t , perform n traversals of t , one for every atomic substitution. This could be reduced by traversing the expression just once and applying lifting (+) or locks to all atomic substitutions simultaneously when required.

3.3 Interpretation of WSMTT Expressions in SFMTT

We now turn to the relation between WSMTT and SFMTT. Using the substitution algorithm just defined, we will show that WSMTT expressions can be translated to SFMTT expressions, essentially proving that explicit substitutions can be computed away. The reverse direction is easier: apart from variables, every SFMTT expression constructor also appears in WSMTT

so we can almost trivially embed the former system into the latter. We define the two translations here and consider their meta-theoretical properties (particularly soundness and completeness) in the next sections.

3.3.1 Translation from WSMTT to SFMTT

The translation from WSMTT to SFMTT is defined mutually recursively for both expressions and substitutions. In other words, for any WSMTT expression $\hat{\Gamma} \vdash_{\text{ws}} t \text{ expr} @ m$ we get an SFMTT expression $\hat{\Gamma} \vdash_{\text{sf}} \llbracket t \rrbracket \text{ expr} @ m$ and for any WSMTT substitution $\vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ we get an SFMTT (regular) substitution $\vdash_{\text{sf}} \llbracket \sigma \rrbracket \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$. We only show some of the cases for the different expression constructors. Again, in order for $\llbracket _ \rrbracket$ to be well-defined, we should check that the definition below preserves well-scopedness.

$$\begin{array}{ll}
\llbracket \mathbf{v}_0 \rrbracket = \mathbf{v}_0^{1\mu} & \llbracket \pi \rrbracket = \text{id} @ \text{weaken}(\text{id}^a) \\
\llbracket (\mu \vdash A) \rightarrow B \rrbracket = (\mu \vdash \llbracket A \rrbracket) \rightarrow \llbracket B \rrbracket & \llbracket \sigma \circ \tau \rrbracket = \llbracket \sigma \rrbracket ++ \llbracket \tau \rrbracket \\
\llbracket t [\sigma]_{\text{ws}} \rrbracket = \llbracket t \rrbracket \llbracket \llbracket \sigma \rrbracket \rrbracket_{\text{sub}} & \llbracket \sigma . \mathbf{a}_\mu \rrbracket = \llbracket \sigma \rrbracket . \mathbf{a}_\mu \\
\llbracket ! \rrbracket = \text{id} @ ! & \llbracket \mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \Theta \Rightarrow \Psi} \rrbracket = \text{id} @ \mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \Theta \Rightarrow \Psi} \\
\llbracket \text{id} \rrbracket = \text{id} & \llbracket \sigma . t \rrbracket = \llbracket \sigma \rrbracket^+ @ (\text{id}^a . \llbracket t \rrbracket)
\end{array}$$

When translating an (explicitly) substituted WSMTT expression $t [\sigma]_{\text{ws}}$, we translate both the expression t and the substitution σ and then apply Theorem 9 (i.e. the algorithm from the previous section). Translation of a composite substitution involves the concatenation of the two translated substitutions, which are regular SFMTT substitutions so sequences of atomic SFMTT substitutions. Recall that the operations $_ . \mathbf{a}_\mu$ and $+$ for regular SFMTT substitutions are defined at the end of Section 3.1. Finally, one could wonder why in the translation of $\sigma . t$ we first add $\llbracket t \rrbracket$ to the identity atomic substitution and then apply the lifted version of $\llbracket \sigma \rrbracket$ where it would seem easier to first apply (the non-lifted) $\llbracket \sigma \rrbracket$ and then extend id^a with $\llbracket t \rrbracket$. The answer is that in that case $\llbracket t \rrbracket$ would live in the wrong scoping context: if $\llbracket \sigma \rrbracket$ goes from $\hat{\Gamma}$ to $\hat{\Delta}$, then $\llbracket t \rrbracket$ lives in $\hat{\Gamma} . \mathbf{a}_\mu$ but if we want the translation of $\sigma . t$ to be of the form $(\text{id}^a . ?) @ \llbracket \sigma \rrbracket$, then we need some term in scoping context $\hat{\Delta} . \mathbf{a}_\mu$ at the place of the question mark.

3.3.2 Embedding of SFMTT into WSMTT

We only provide an embedding of SFMTT expressions to WSMTT expressions (so not for substitutions). Apart from the constructor for variable expressions, all SFMTT expression constructors also occur in WSMTT. We therefore only specify how to embed variables.

$$\text{embed}(\mathbf{v}_0^\alpha) = \mathbf{v}_0 \left[\mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \mathbf{a}_\mu \Rightarrow \Theta} \right]_{\text{ws}} \quad \text{embed}(\text{suc}(v)) = \text{embed}(v) [\pi . \Theta]_{\text{ws}}$$

The lock telescopes Θ in both cases are inferred from the scoping context (recall that we consider SFMTT expressions to be intrinsically scoped).

As a result, for every SFMTT expression $\hat{\Gamma} \vdash_{\text{sf}} t \text{ expr} @ m$ we get a corresponding WSMTT expression $\hat{\Gamma} \vdash_{\text{ws}} \text{embed}(t) \text{ expr} @ m$.

4 Soundness & Completeness

In the previous section, we introduced a translation from WSMTT to SFMTT that uses our substitution algorithm to translate away WSMTT's explicit substitution. In this section, we establish the translation's key properties: soundness and completeness with respect to σ -equivalence in WSMTT.

4.1 Soundness

In our setting, soundness is the property that starting from a WSMTT expression, applying the translation where all explicit substitutions are computed away, and then embedding the result back into WSMTT, we get a result that is σ -equivalent to the original expression. In order to prove this, we will make use of an embedding of SFMTT substitutions into WSMTT, which was not provided in Section 3.3.2. We therefore define the following for both atomic and regular SFMTT substitutions.

$$\begin{array}{ll}
 \text{embed}(!) = ! & \text{embed}\left(\mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \Lambda \Rightarrow \Theta}\right) = \mathcal{Q}_{\hat{\Gamma}}^{\alpha \in \Lambda \Rightarrow \Theta} \\
 \text{embed}(\text{id}^a) = \text{id} & \text{embed}(\sigma.t) = \text{embed}(\sigma) . \text{embed}(t) \\
 \text{embed}(\text{weaken}(\sigma)) = \text{embed}(\sigma) \circ \pi & \text{embed}(\text{id}) = \text{id} \\
 \text{embed}(\sigma . \mathbf{\mu}_{\mu}) = \text{embed}(\sigma) . \mathbf{\mu}_{\mu} & \text{embed}(\sigma \circ \tau) = \text{embed}(\sigma) \circ \text{embed}(\tau)
 \end{array}$$

The crucial case in the proof of the soundness theorem is when the WSMTT expression is of the form $t [\sigma]_{\text{ws}}$. In that case we use the following lemma.

► **Lemma 10.** *Given an SFMTT expression $\hat{\Delta} \vdash_{\text{sf}} t \text{ expr} @ m$ and substitution $\vdash_{\text{sf}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$, we have that $\hat{\Gamma} \vdash_{\text{ws}} \text{embed}(t [\sigma]_{\text{sub}}) \equiv^{\sigma} \text{embed}(t) [\text{embed}(\sigma)]_{\text{ws}} \text{ expr} @ m$.*

Lemma 10 tells us that computing away a substitution in SFMTT and embedding the result in WSMTT gives an expression that is σ -equivalent to the result of explicitly applying the embedded substitution in WSMTT. The proof of this lemma is technically quite involved (it proceeds by induction on t and σ , the most difficult cases being weakening and key substitutions) and can therefore be found in the technical report [13].

► **Theorem 11 (Soundness).** *For every WSMTT expression $\hat{\Gamma} \vdash_{\text{ws}} t \text{ expr} @ m$ we have $\hat{\Gamma} \vdash_{\text{ws}} \text{embed}(\llbracket t \rrbracket) \equiv^{\sigma} t \text{ expr} @ m$ and for every WSMTT substitution $\vdash_{\text{ws}} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ we have $\vdash_{\text{ws}} \text{embed}(\llbracket \sigma \rrbracket) \equiv^{\sigma} \sigma \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$.*

Idea of proof. This proof proceeds by induction on the expression t and the substitution σ . We only show one case for t , the other cases can be found in the technical report [13].

■ CASE $\hat{\Gamma} \vdash_{\text{ws}} t [\sigma]_{\text{ws}} \text{ expr} @ m$

We have $\text{embed}(\llbracket t [\sigma]_{\text{ws}} \rrbracket) = \text{embed}(\llbracket t \rrbracket [\llbracket \sigma \rrbracket]_{\text{sub}}) \equiv^{\sigma} \text{embed}(\llbracket t \rrbracket) [\text{embed}(\llbracket \sigma \rrbracket)]_{\text{ws}}$ where the last σ -equivalence holds because of Lemma 10. The induction hypothesis for t and σ gives us that $\text{embed}(\llbracket t \rrbracket) \equiv^{\sigma} t$ and $\text{embed}(\llbracket \sigma \rrbracket) \equiv^{\sigma} \sigma$, which proves the desired result. ◀

4.2 Completeness

Completeness of our algorithm with respect to σ -equivalence means that whenever two WSMTT expressions are σ -equivalent, the results when computing away all substitutions via $\llbracket _ \rrbracket$ are the same. Recall that σ -equivalence for WSMTT expressions is defined mutually recursively with σ -equivalence for WSMTT substitutions (see Figure 6). Therefore, to prove completeness we will simultaneously need to prove a similar result about σ -equivalent WSMTT substitutions. However, in SFMTT, syntactic equality of substitutions is not a good notion of equivalence. Instead, we will use the following.

► **Definition 12 (Observational equivalence).** *We say that two SFMTT substitutions $\vdash_{\text{sf}} \sigma, \tau \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$ are observationally equivalent when $t [\sigma]_{\text{sub}} = t [\tau]_{\text{sub}}$ for every expression $\hat{\Delta} \vdash_{\text{sf}} t \text{ expr} @ m$. We will write this as $\sigma \approx^{\text{obs}} \tau$.*

This notion of observational equivalence is actually quite strong because it quantifies over all possible SFMTT expressions. That means that both substitutions might get pushed under a lot of expression constructors, with locks or lifts added along the way. The technical report [13] shows the following lemma, which makes it easier to prove observational equivalence.

► **Lemma 13.** *Let $\vdash_{\text{sf}} \sigma, \tau \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ n$ be two SFMTT substitutions and suppose that $v [\sigma \cdot \Lambda]_{\text{sub}} = v [\tau \cdot \Lambda]_{\text{sub}}$ for every lock telescope $\Lambda : \text{LockTele}(n \rightarrow m)$ and every variable $\hat{\Delta} \cdot \Lambda \vdash_{\text{sf}} v \text{ var} @ m$. Then $\sigma \approx^{\text{obs}} \tau$.*

► **Remark 14.** Lemma 13 essentially says that a substitution is uniquely determined, up to observational equivalence, by its action on De Bruijn indices. In plain dependent type theory, substitutions are often *defined* as mappings from variables to terms, or at least it is clear that they can be uniquely represented in this way. The technical report [13] provides an example that this is impossible for SFMTT: not every such mapping arises from an SFMTT substitution. In other words, the structure of substitutions in modal type theory is fundamentally more complex than in plain dependent type theory.

► **Theorem 15 (Completeness).** *If we have two σ -equivalent WSMTT expressions $\hat{\Gamma} \vdash_{\text{ws}} t \equiv^{\sigma} s \text{ expr} @ m$, then $\llbracket t \rrbracket = \llbracket s \rrbracket$. Furthermore, given two σ -equivalent WSMTT substitutions $\vdash_{\text{ws}} \sigma \equiv^{\sigma} \tau \text{ sub}(\hat{\Gamma} \rightarrow \hat{\Delta}) @ m$, we have that $\llbracket \sigma \rrbracket \approx^{\text{obs}} \llbracket \tau \rrbracket$.*

Idea of proof. We proceed by induction on a derivation of the σ -equivalence judgement, going over all inference rules from Figure 6. Only one case is presented here, the others can be found in the technical report [13].

■ CASE $\hat{\Gamma} \vdash_{\text{ws}} t [\tau]_{\text{ws}} \equiv^{\sigma} s [\sigma]_{\text{ws}} \text{ expr} @ m$
 The premises of this inference rule are $\hat{\Gamma} \vdash_{\text{ws}} t \equiv^{\sigma} s \text{ expr} @ m$ and $\vdash_{\text{ws}} \tau \equiv^{\sigma} \sigma \text{ sub}(\hat{\Delta} \rightarrow \hat{\Gamma}) @ m$, so by the induction hypothesis we have $\llbracket t \rrbracket = \llbracket s \rrbracket$ and $\llbracket \tau \rrbracket \approx^{\text{obs}} \llbracket \sigma \rrbracket$. Using the definition of \approx^{obs} we then get that $\llbracket t [\tau]_{\text{ws}} \rrbracket = \llbracket t \rrbracket [\llbracket \tau \rrbracket]_{\text{sub}} = \llbracket s \rrbracket [\llbracket \sigma \rrbracket]_{\text{sub}} = \llbracket s [\sigma]_{\text{ws}} \rrbracket$. ◀

A consequence of soundness and completeness is the following result.

► **Theorem 16.** *Given two WSMTT expressions $\hat{\Gamma} \vdash_{\text{ws}} t, s \text{ expr} @ m$, then $\hat{\Gamma} \vdash_{\text{ws}} t \equiv^{\sigma} s \text{ expr} @ m$ if and only if $\llbracket t \rrbracket = \llbracket s \rrbracket$. From this it follows that SFMTT expressions are the σ -normal forms of WSMTT expressions, and $\llbracket - \rrbracket$ is the normalisation function.*

Proof. The direction from left to right is exactly Theorem 15. Conversely, suppose that $\llbracket t \rrbracket = \llbracket s \rrbracket$. Then we know that $t \equiv^{\sigma} \text{embed}(\llbracket t \rrbracket) = \text{embed}(\llbracket s \rrbracket) \equiv^{\sigma} s$. To show that SFMTT expressions are the σ -normal forms of WSMTT expressions, we only need to prove that every SFMTT expression is in the image of the $\llbracket - \rrbracket$ function. This is indeed the case since $\llbracket \text{embed}(t) \rrbracket = t$ for all $\hat{\Gamma} \vdash_{\text{sf}} t \text{ expr} @ m$ (which is provable via a trivial induction on t). ◀

5 Related and Future Work

5.1 Normalisation by Evaluation for MTT

Normalisation of MTT with respect to $\sigma\beta\eta$ -equality had already been proven by Gratzner [17] [18, ch. 8]. He uses a normalisation by evaluation (NbE) argument [5, 3], structured using the more recent technique of Synthetic Tait Computability (STC) [31][18, ch. 4]. We compare Gratzner's work with ours both in terms of approach and of implications.

Implications. An NbE algorithm will take as input a term $\Gamma \vdash t : T$ (considered up to $\sigma\beta\eta$ -equality) and a *value environment* $\rho : \text{env}(\Delta \rightarrow \Gamma)$ and return a $\sigma\beta\eta$ -normal form $\Delta \vdash \text{nbe}(t, \rho) : T[\rho]$. When we instantiate ρ with the identity environment, which substitutes every variable with itself or its η -expansion, then we are really just normalising t . When instead we are only interested in syntax up to $\sigma\beta\eta$ -equality, and thus not in $\sigma\beta\eta$ -normalisation which is inobservable up to $\sigma\beta\eta$ -equality, then the algorithm really just applies the substitution ρ to the term t . So in this sense an NbE algorithm already allows for substitution and indeed this is sufficient for a proof-of-concept implementation of MTT [30].

However, for conceptual, didactical and practical reasons, we see a role for a substitution and σ -normalisation algorithm unreliant on $\beta\eta$ -equality as presented in the current paper. Conceptually, there is the fact that substitution originates as a find-replace operation that replaces every occurrence of a given variable with a term of the same type. While the definition of such an operation becomes more difficult with the introduction of variable binding, dependent types, \dots , it is still a reasonable expectation and indeed a sanity check to ask that this operation be definable, without referring to computation or $\beta\eta$ -equality. It ensures that, even before considering computation, variables can be thought of as placeholders, and that programs are not permanently tied to the context in which they are defined, but merely use the context as an interface. Didactically, since computation relies on substitution, it is desirable to be able to explain substitution *first*, and especially without having to introduce NbE. Practically, when working in a dependently typed proof assistant, we want to get type goals that are *not* in $\sigma\beta\eta$ -normal form. For example, an η -normal form of an advanced algebraic structure will typically be a big nested record type listing all carriers and implementing all available operations, which may not be quite as readable as the more intensional way in which the algebra was constructed. A proof assistant that relies on NbE for substitution, will not be able to type a function application $f a$ of a dependently typed function f without normalising the codomain of f . Our algorithm, on the other hand, will cleanly push substitutions through all non-substitution-related syntax constructors and merely find and replace variables.

Approach. A first stark difference between NbE and the current work is that NbE considers a type system’s syntax up to $\sigma\beta\eta$ -equality, i.e. it considers the type system’s initial model in which important type formers can be characterised by their universal properties. In order to speak about σ -equality, we need to distinguish $\beta\eta$ -equal terms and lose some of the categorical tooling. In particular, the category of models of a type system is of little use and most type formers do not satisfy their universal properties up to σ - or syntactic equality.

Similarly, because typing relies on $\beta\eta$ -equality and we want to get the complications of substitution out of the way *before* considering $\beta\eta$ -equality (e.g. because of the conceptual and didactical reasons above), we work with intrinsically scoped untyped syntax, whereas NbE generally works with intrinsically typed syntax.

NbE arguments generally feature at least five “collections of program representations”: variables, neutrals, normal forms, values, and $\sigma\beta\eta$ -equivalence classes⁸ of terms. An NbE proof involves several operations on and between these collections, and each of them is stable under renaming, which is necessary to deal with λ -abstraction and application. In the current work, we do not ever need to construct or eliminate functions, so while we do need to apply lock telescopes to renamings and substitutions, it turns out there is no need to prove that

⁸ When formalising type theory in type theory, one would not use set-theory-style quotients based on equivalence classes, but instead use quotient-inductive-inductive types [6].

every operation featured in the construction is stable under renaming. Furthermore, while MTT and SFMTT can be regarded as the collections of terms and normal forms respectively, and we also have a definition of SFMTT variables, we do not need to distinguish between values and normal forms (which in NbE has mostly to do with η -equality) and we do not need a separate collection of neutrals (as σ -reduction, unlike β -reduction, is never stuck on a variable).

5.2 Second-order Algebraic Theories

Allais, Atkey, Chapman, McBride and McKinna [4] implement renaming and substitution (among many other things) at once for a large class of languages, which Fiore and Szamozvancev [16] identify to be what is often called second-order multisorted algebraic theories (SOMATs). Here, multisorted means simply-typed, and second-order means that they accommodate variable-binding, but no other context features, i.e. it is assumed that contexts, renamings and substitutions are lists of types, variables and terms respectively. More recently and in a more categorical perspective, Uemura has defined the corresponding class of dependently typed languages, which in the larger naming scheme would be called second-order generalised algebraic theories (SOGATs). A similar general substitution result should be possible for SOGATs, and in any case it is very well understood (but considered tedious) how to define substitution for specific SOGATs, which is why there is nowadays little attention for this problem in the metatheory of specific *non-modal* languages. The necessity of the current work arises from the fact that, due to the presence of modal locks, MTT is not a SOGAT (it is a generalised algebraic theory or GAT [11], as are WSMTT and SFMTT). A generalisation of second-order algebraic theories that would subsume MTT or at least Multimode Simple Type Theory (MSTT) [12] is work in progress [24] and will be informed by our current findings.

5.3 Other Approaches to Modal Contexts and/or Substitution

Lock calculi. Bahr, Grathwohl and Møgelberg [7] introduce Clocked Type Theory (CloTT), a system for guarded type theory which features a *later* modality \triangleright for every clock listed in the clock context. If we keep the clock context fixed, then to a large extent CloTT can be regarded as an instance of MTT,⁹ but the “lock” operation for each later modality is *named*. To clarify, we put the introduction rules for the later types for a clock κ in MTT and CloTT side by side:

$$\frac{\Gamma, \mathbf{lock}_{\triangleright\kappa} \vdash t : T}{\Gamma \vdash \mathbf{mod}_{\triangleright\kappa}(t) : \langle \triangleright^{\kappa} \mid T \rangle} \quad \frac{\Gamma, \alpha : \kappa \vdash t : T}{\Gamma \vdash \lambda(\alpha : \kappa).t : \triangleright(\alpha : \kappa).T}$$

The variable α is called a tick of the clock κ , but we can more generally call it a lock variable. The specific mode theory for CloTT is enforced by requiring that α be used substructurally. This slightly complicates the type system but on the bright side, substitutions in CloTT are simply variable and tick replacement operations and do not have the complex categorical structure they have in MTT, facilitating implementation in Agda [33]. Dependent quantification over an affine or cartesian interval variable in cubical homotopy or parametric type theory [9, 15, 8] can also be regarded as an instance of this approach, with the interval

⁹ Alternatively, we could regard the clock context as the mode, in which case we have an instance of MTT where clock substitution and quantification are also modalities. However, our discussion about lock calculi does not apply if we take that perspective.

variable being analogous to the tick. We could similarly try to assign a lock variable to every lock in MTT and extend MTT with a substructural lock calculus [25]. This is challenging however, as we need to deal with arbitrarily complex mode theories and the lock calculus admits in general neither weakening, exchange nor contraction.

2-posetal MTT. If MTT is instantiated on a mode theory that is a 2-poset, meaning that the 2-cell of a given domain and codomain is unique if it exists, and if moreover this existence is decidable, then rather than listing 2-cell information on variables and in substitutions, the unique existence of the necessary 2-cells can be checked. Then all the remaining *information* in a substitution is a list of terms, and the substitution operation is again merely a find-replace operation. In the implementation of the proof assistant Mitten [30], this fact is used to optimise the NbE algorithm (Section 5.1) for implementation.

Left division. MTT is based on a line of work on type systems using a *left division* operation [2, 28, 27, 26], which in turn can be regarded as a generalisation of a dual-context approach [29]. Rather than having a context *constructor* $\mathbf{\mu}$ which is semantically left adjoint to the modality, it is assumed that there exists a left division operation $\mu \setminus _$ left adjoint to $\mu \circ _$ on modalities, and this operation extends to contexts by applying it to the modal annotation of every variable. In systems based on left division, contexts are lists of modality-annotated types, and substitutions are lists of terms. The difficult question there is not whether substitution is definable, but whether left division of contexts is functorial. This question has to our knowledge never been properly studied for general mode theories. Moreover, left division of contexts is itself a defined operation on syntax and, unlike substitution, typically does not have clean denotational semantics.

Fitch-style calculi. Logics and type systems that feature typically a single modality \square and a left adjoint context constructor $\mathbf{\mu}$, but no modal annotations on variables, are referred to as Fitch-style calculi [14]. Given the presence of only a single modality, the proof of Theorem 2 only stands if there is a non-trivial and non-horizontally-decomposable 2-cell between powers of \square , e.g. the duplication $\delta \in \square \Rightarrow \square \square$ of a comonad. Gratzer, Sterling and Birkedal [21] implement type theory with an S4-style \square -modality (i.e. an applicative comonad) and indeed our counterexample applies. They do not define a substitution operation and instead use NbE. Valliappan, Ruch and Cortiñas [32] prove NbE for four modal systems, where \square is an applicative functor with optionally a co-unit $\varepsilon \in \square \Rightarrow \mathbb{1}$ and/or a duplication δ . Each time, they define a *modal accessibility relation* $\Delta \triangleleft \Gamma$ on contexts which entails the existence of a substitution $\Gamma \rightarrow \Delta . \mathbf{\mu}$ involving only weakening and 2-cells. As such, unlike MTT, their system has a composition-free substitution $\Gamma . \mathbf{\mu} . A . \mathbf{\mu} \rightarrow \Gamma . \mathbf{\mu}$ that forgets the variable of type A and fuses the locks all at once. Still, they do not claim definability of composition of substitution (only identity), nor do they define substitution, instead using NbE. For pointed modalities and monads, on the other hand, we refer back to the lock calculi discussed above, with the later modality and interval quantification as examples.

References

- 1 M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90*, pages 31–46, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/96709.96712.

- 2 Andreas Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 18(5):797–822, 2008. doi:10.1017/S0960129508006853.
- 3 Andreas Abel. *Normalization by evaluation: Dependent types and impredicativity*. Habilitation thesis, Ludwig-Maximilians-Universität München, Germany, 2013.
- 4 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type- and scope-safe universe of syntaxes with binding: their semantics and proofs. *Journal of Functional Programming*, 31:e22, 2021. doi:10.1017/S0956796820000076.
- 5 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter T. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 1995. doi:10.1007/3-540-60164-3_27.
- 6 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016. doi:10.1145/2837614.2837638.
- 7 Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The clocks are ticking: No more delays! In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005097.
- 8 Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electron. Notes in Theor. Comput. Sci.*, 319:67–82, 2015. doi:10.1016/j.entcs.2015.12.006.
- 9 Marc Bezem, Thierry Coquand, and Simon Huber. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, Dagstuhl, Germany, 2014. doi:10.4230/LIPIcs.TYPES.2013.107.
- 10 Lars Birkedal, Ranald Clouston, Bassel Manna, 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.
- 11 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.
- 12 Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. Sikkel: Multimode simple type theory as an agda library. In Jeremy Gibbons and Max S. New, editors, *Proceedings Ninth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2022, Munich, Germany, 2nd April 2022*, volume 360 of *EPTCS*, pages 93–112, 2022. doi:10.4204/EPTCS.360.5.
- 13 Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. A sound and complete substitution algorithm for multimode type theory: Technical report. 2024. doi:10.48550/arXiv.2406.13622.
- 14 Ranald Clouston. Fitch-style modal lambda calculi. In Christel Baier and Ugo Dal Lago, editors, *FOSSACS 2018, Held as Part of ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2018. doi:10.1007/978-3-319-89366-2_14.
- 15 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, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.TYPES.2015.5.
- 16 Marcelo Fiore and Dmitriy Szamozvancev. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022. doi:10.1145/3498715.

- 17 Daniel Gratzer. Normalization for multimodal type theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '22*, 2022. doi:10.1145/3531130.3532398.
- 18 Daniel Gratzer. *Syntax and semantics of modal type theory*. PhD thesis, Aarhus University, Denmark, 2023. URL: [Syntaxandsemanticsofmodaltypetheory](https://syntaxandsemanticsofmodaltypetheory).
- 19 Daniel Gratzer, Alex Kavvos, Andreas Nuyts, and Lars Birkedal. Type theory à la mode. Pre-print, 2020. URL: <https://lirias.kuleuven.be/retrieve/635295>.
- 20 Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal Dependent Type Theory. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021. doi:10.46298/lmcs-17(3:11)2021.
- 21 Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *Proc. ACM Program. Lang.*, pages 107:1–107:29, 2019. doi:10.1145/3341711.
- 22 Per Martin-Löf. Intuitionistic type theory. In *Studies in proof theory*. Bibliopolis, 1984.
- 23 Conor McBride. Type-preserving renaming and substitution. Unpublished note, 2005. URL: <http://strictlypositive.org/ren-sub.pdf>.
- 24 Andreas Nuyts. Contextual algebraic theories: Generic boilerplate beyond abstraction (extended abstract). In *Workshop on Type-Driven Development (TyDe)*, September 2022. URL: <https://anuyts.github.io/files/cmat-tyde22-abstract.pdf>.
- 25 Andreas Nuyts. A lock calculus for multimode type theory. In *29th International Conference on Types for Proofs and Programs (TYPES)*, June 2023. URL: <https://lirias.kuleuven.be/retrieve/720873>.
- 26 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 *Logic in Computer Science (LICS) 2018, Oxford, UK, July 09-12, 2018*, pages 779–788, 2018. doi:10.1145/3209108.3209119.
- 27 Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *PACMPL*, 1(ICFP):32:1–32:29, 2017. doi:10.1145/3110276.
- 28 Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS '01*, pages 221–230, 2001. doi:10.1109/LICS.2001.932499.
- 29 Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. doi:10.1017/S0960129501003322.
- 30 Philipp Stassen, Daniel Gratzer, and Lars Birkedal. mitten: A Flexible Multimodal Proof Assistant. In Delia Kesner and Pierre-Marie Pédro, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:23, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.TYPES.2022.6.
- 31 Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University, USA, 2022. doi:10.1184/R1/19632681.V1.
- 32 Nachiappan Valliappan, Fabian Ruch, and Carlos Tomé Cortiñas. Normalization for fitch-style modal calculi. *Proc. ACM Program. Lang.*, 6(ICFP), August 2022. doi:10.1145/3547649.
- 33 Niccolò Veltri and Andrea Vezzosi. Formalizing π -calculus in guarded cubical agda. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 270–283. ACM, 2020. doi:10.1145/3372885.3373814.

Consistent Ultrafinitist Logic

Michał J. Gajda   

Migamake Pte Ltd, Singapore

Abstract

Ultrafinitism postulates that we can only compute on relatively short objects, and numbers beyond a certain value are not available. This approach would also forbid many forms of infinitary reasoning and allow removing certain paradoxes stemming from enumeration theorems. For a computational application of ultrafinitist logic, we need more than a proof system, but a logical framework to express both proofs, programs, and theorems in a single framework. We present its inference rules, reduction relation, and self-encoding to allow direct proving of the properties of ultrafinitist logic within itself. We also provide a justification why it can express all bounded Turing programs, and thus serve as a “logic of computability”.

2012 ACM Subject Classification Theory of computation → Constructive mathematics

Keywords and phrases ultrafinitism, bounded Turing completeness, logic of computability, decidable logic, explicit complexity, strict finitism

Digital Object Identifier 10.4230/LIPIcs.TYPES.2023.5

Related Version Presented on TYPES2022, CiE2022, LAP2021, CLMPST2023, CLA2022

Preprint: <https://arxiv.org/abs/2106.13309>

Acknowledgements We thank Vendran Čačić, Seth Chaiken, Bhupinder Singh Anand, Orestis Melkonian, Anton Setzer for their invaluable feedback. Thanks to Daniel Schwartz for sparking my interest in Kleene normal form.

Background

Ultrafinitism [42, 61, 79, 22, 47] postulates that we can only reason and compute relatively short objects¹

Tighter limit can be established using petahertz frequency [9] as a quantum limit for light-based systems giving 10^{33} serial cycles during the lifetime of Earth. [49, 28, 66, 48, 44, 46], and numbers beyond certain value are not available [79, 66]. Some philosophers also question the physical existence of real numbers beyond a certain level of accuracy [24]. This approach would also forbid many forms of infinitary reasoning and allow removing many from paradoxes stemming from a countable enumeration.

However, philosophers² still disagree on whether such a ultrafinitist logic could be consistent [17, 51], while constructivist mathematicians claim that “*no satisfactory developments exist*” [74]. We present a proof system based on the Curry-Howard isomorphism [35] and explicit bounds for computational complexity that answers the question.

This approach invalidates logical paradoxes that stem from a profligate use of transfinite reasoning [6, 55, 67], and assures that we only state problems that are decidable by the limit on input size, proof size, and the number of steps. This explicitly excludes phenomena of undecidability by excluding them from our realm of valid statements [73]. Our approach allows to express all Turing Machine programs that are bounded [34] by proof terms of the logic³.

¹ For example, a computation run by computer the size of Earth within the lifespan of Earth so far. Of the order of 10^{93} as described by [28].

² We cite physical and metaphysical arguments from previous work equally.

³ Up to fixed emulation overhead, see Emulation Complexity below in section 4.



© Michał J. Gajda;

licensed under Creative Commons License CC-BY 4.0

29th International Conference on Types for Proofs and Programs (TYPES 2023).

Editors: Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg; Article No. 5; pp. 5:1–5:20

Leibniz International Proceedings in Informatics



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

Explicitly bounding computational complexity also prevents a famous *paradox of inference*. This paradox of classical theory of semantic information [4, 19] unjustly labels all mathematical proofs as “trivial information”, because it can be inferred from the axioms.

1 Introduction

By *finitism* we understand the mathematical logic that tries to absolve us from transfinite inductions [42]. *Ultrafinitism*⁴ goes even further by postulating a definite limit for the complexity of objects that we can compute with [48, 44, 66, 49, 28, 20]. We assume these without committing ourselves to adopt a fixed number as a limit.

In order to permit only *ultrafinitist* inferences, we postulate *ultraconstructivism*: we permit only *constructive* proofs with a *deadline*. That is constructions that are not just strictly computable, but for which there is a *upper bound* on the amount of computation that is needed to resolve them. That means that we forbid proofs that go for an arbitrarily long time and require *totality* for any proof or computation.

For the sake of generality, we will attach this *deadline* in the form of *bounding function* that takes as arguments *size variables* (*depths of input terms*), and outputs the upper bound on the number of steps that the proof is permitted to make (along with upper bound on the size of the output). *Depths of input terms* are a convenient upper bound on the complexity of normalized proof terms. (Normalized proof terms are those with opportunity for *cut* or β -reduction.)

Our approach is inspired by the *Curry-Howard isomorphism* – the fact that the constructive proofs always correspond to executable programs. It also follows *inverse Curry-Howard isomorphism*: the philosophy that rejects logical inference which do not correspond to programs computable in our universe⁵.

The philosophy of *ultraconstructivism* would similarly purport that while transfinite logics may be consistent, they are correspond to objects „*out-of-this-world*’⁶, since our observable universe is inherently finitary [49].

Contributions

To enumerate chief contributions of this paper:

1. First consistent **ultrafinitist** logic to the knowledge of the author. It allows bounding by any and arbitrarily large computational limit (section 2.1), and consistent reasoning resolving Wang’s Paradox[17]. Thus this logic is first formal theory to claim a purely philosophical legacy of ultrafinitism [79, 66, 51, 10].
2. A decidable logic having **meta-theory expressible in itself** (see section 3.4).
3. Clear and comprehensive demonstration that **assumptions of Gödel are too strong** [25, 26] when considering bounded logics. This is because decidability of bounded term can be demonstrated by simple enumeration with a more generous bound. Most proofs are elementary by enumeration. Proof of consistency is done by reduction to widely known intuitionistic logic to make the paper accessible to second year students of computer science or first year graduates in constructive mathematics (section 4).

⁴ Also called *strict finitism* by [51].

⁵ Given that all formal proofs in mathematically strict formal systems can be considered finite numbers of connected steps in computation or hypercomputation.

⁶ Extra-universal.

4. Candidate for a most expressive logic that allows explicitly bounded computable functions. Ideal **logic of computability** must forbid all reasoning about uncomputable, and only allow computable statements (with proofs corresponding to *bounded Turing Machine* programs.) For all bounds that can be computed within the framework, we can also compute the function bounded by these (section 4.3).
5. Placing **ultrafinitism** and **ultraconstructivism** as candidate for realization of *computable foundations for mathematics* programme (discussion in section 6.3).
6. All statements with bounds having a proof without bounds have a proof with bounds too⁷ (see section 4.6 theorem 9).

We will further abbreviate the “Consistent Ultra-Finitist Logic” proposed in here as “UFL” when speaking about higher order variant (with dependent Π, Σ types for quantification).

2 Syntax and inference

Due to size bounds and clarity of this paper, we first introduce propositional ultrafinitist logic, and then describe interpretation of universal quantifier in a separate section.

2.1 Bounds

We express bounds as polyvariate functions of the natural numbers, called *sizes*. These explicitly bound our proofs, depending on the size of input terms. While subtraction within natural domain is permitted, only positive results of computation are permitted. All bound functions are increasing with respect to all arguments (monotonic).

The bounds⁸ will be standing on one of two roles: as an upper bound on the proof complexity, and there we will use symbol α as a placeholder, or to state an upper bound on the depth of the normal form of the proof indicated by the symbol β . That is because the number of constructors may sometimes bound a recursive examination of the proof of a proposition.

Here ρ^ρ is an exponentiation, and $iter(\lambda v. \rho_1, \rho_2, \rho_3)$ is an iterated composition of function described by expression ρ_1 with respect to an argument variable v ; that iteration happens ρ_2 times, and the function is applied to initial argument ρ_3 . The $\rho_1 \llbracket \rho_2/v \rrbracket$ describes substitution of bound variable v by ρ_2 , inside expression ρ_1 .

Any total function $f(\dots)$ over naturals has a bounds $b(\dots)$ function $\forall x \in \text{Nat} . x \leq n \implies f(x) \leq b(n)$: given a range limited⁹ by n , we can compute $b(n)$ that is $\max_{x \leq n} f(x)$.

Conjecture Bounds terminate Since all iterations in bounds quantification terminate, all bounds terminate.

2.2 Terms

All terms are explicitly limited, but we avoid labelling terms for which bounds can be easily inferred (see below).

⁷ Thanks to anonymous reviewer for pointing importance of this result.

⁸ Using a bound on cost and depth of the term for each inference, we independently developed a very similar approach to that used for cost bounding in higher-order rewriting [41].

⁹ This is not true for traditional real numbers \mathbb{R} : hyperbola $y = \frac{1}{x}$ is unbounded around 0 because $\lim_{x \rightarrow 0} \frac{1}{x} = -\infty$.

5:4 Ultrafinitist Logic

Size variables:	$v \in V$
Size values	$n = \mathbf{1} \mid \mathbf{S}(n)$
Term variables:	$x \in X$
Positive naturals:	$i \in \mathbb{N} \setminus \{0\}$
Upper bounds:	$\rho ::= v \mid i \mid \mathbf{S}(\rho) \mid$ $\mid \text{iter}(\lambda v. \rho, \rho, \rho) \mid \rho \llbracket \rho/v \rrbracket \mid \text{max}(\rho, \rho)$

Iteration is defined as:

$$\begin{aligned} \text{iter}(\lambda v. e, \mathbf{1}, a) &= e \llbracket a/v \rrbracket \\ \text{iter}(\lambda v. e, \mathbf{S}(n), a) &= \text{iter}(\lambda v. e, n, e \llbracket a/v \rrbracket) \end{aligned}$$

Later we will explain how bounds expressions can be encoded in the same language as the proof terms. At the level of basic logic we do not need this, but it will become useful when we consider meta-reasoning (and encode entirety of the logic within its own proof terms.)

Data size bounds:	$\alpha ::= \rho$
Computation bounds:	$\beta ::= \rho$
Types:	$\tau ::= v \mid \tau \wedge \tau \mid \tau \vee \tau \mid \tau_v \rightarrow_{\beta}^{\alpha} \tau \mid \perp \mid \circ \mid \mathbf{Type}$
Terms:	$E ::= x \mid \lambda x. E \mid \text{in}_r(E) \mid \text{in}_l(E) \mid (E, E) \mid$ $\mid \text{case } E \text{ of } \begin{cases} \text{in}_l(x) \rightarrow E; \\ \text{in}_r(x) \rightarrow E; \end{cases}$
Environments:	$\Gamma ::= v_1 : \tau_{\beta_1}^{\alpha_1}, \dots, v_n : \tau_{\beta_n}^{\alpha_n}$
Judgements:	$J ::= \Gamma \vdash_{\beta}^{\alpha} E : \tau$

There is a special expression **Type** which is syntactically in τ . It is later used when introducing dependent types, since **Type** lives both as a type and as a term. Translation between proofs and types is later described in table 6.

Notation $A_v \rightarrow_{\beta(v)}^{\alpha(v)} B$ binds proof variable x with type of A and size variable v , and then bound in bounds $\alpha(v)$ for complexity and $\beta(v)$ for *depth* of the *normalized* term. We use notation $\alpha(v)$ instead of α to emphasize that both $\alpha(v)$ and $\beta(v)$ are functions of size variable v .

We could attach a pair of bounds to each proposition and judgement $A_{(\alpha, \beta)}$ that would describe both complexity α of computing the proof and a maximum depth β of the resulting (normalized) witness. However, in most cases, one of these would be 1 or could be inferred from the remaining information.

The *occurs* (x, E) is a count of free occurrences of variable x in term E . Free variables of E are computed by $\text{free}(E)$.

2.3 Inference rules

With any term variable x we need to introduce an associated bound variable v .

$$\frac{\Gamma \vdash_{\beta}^{\alpha} A \text{ type} \quad x \in X \quad v \in V}{\Gamma, x_v : A \vdash_v^1 x : A} \text{ var}$$

Sometimes we might want to overestimate proof complexity for the sake of simplicity:

$$\frac{\Gamma \vdash_{\beta_1}^{\alpha_1} e : A \quad \alpha_1 \leq \alpha_2 \quad \beta_1 \leq \beta_2}{\Gamma \vdash_{\beta_2}^{\alpha_2} \text{subsume}(e, \alpha_2, \beta_2) : A} \text{ subsume}$$

■ **Table 1** Encoding arithmetic operations, hyperoperation, and Ackermann function

The addition, multiplication, and exponentiation can be defined on bounds using $\mathbf{S}()$ and $iter(\lambda x.\rho_1, \rho_2, \rho_2)$:

$hyper(a, b, \mathbf{1})$	\equiv	$iter(\lambda x.\mathbf{S}(x), a, b)$
$hyper(a, b, \mathbf{S}(\mathbf{1}))$	\equiv	$iter(\lambda y.iter(\lambda x.\mathbf{S}(x), y, b), \mathbf{1}, \mathbf{1})$ <i>Argument y is ignored in this special case.</i>
$hyper(a, b, \mathbf{S}(\mathbf{S}(n)))$	$=$	$iter(\lambda x.hyper(x, a, n), b, n)$
h	$=$	$\lambda g.\lambda a.\lambda b. case\ b\ of\ \begin{cases} \mathbf{1} \rightarrow & a; \\ \mathbf{S}(c) \rightarrow & iter(\lambda x.fxa, c, a); \end{cases}$
$hyper(a, b, n)$	$=$	$iter(\lambda f.h(f), n, \lambda g.g)$
$a + b$	$=$	$iter(\lambda x.\mathbf{S}(x), a, b)$
$a * \mathbf{1}$	$=$	a
$a * \mathbf{S}(b)$	$=$	$iter(\lambda x.x + a, b, a)$
a^1	$=$	a
$a^{\mathbf{S}(b)}$	$=$	$iter(\lambda x.x * a, b, a)$
$a[\mathbf{S}(n)]b$	$=$	$case\ b\ of\ \begin{cases} \mathbf{1} \rightarrow & a; \\ \mathbf{S}(c) \rightarrow & iter(\lambda x.x[n]b, c, a); \end{cases}$
$a + b$	\equiv	$hyper(a, b, \mathbf{1})$
$a * b$	\equiv	$hyper(a, b, \mathbf{S}(\mathbf{1}))$
a^b	\equiv	$hyper(a, b, \mathbf{S}(\mathbf{S}(\mathbf{1})))$
$\mathbf{S}(\mathbf{S}(\mathbf{S}(ack(m, \mathbf{S}(\mathbf{S}(\mathbf{S}(n)))))))$	\equiv	$2[m](\mathbf{S}(\mathbf{S}(\mathbf{S}(n))))$

Here $=$ is for definition, and \equiv states equivalence of expressions. To avoid definition of predecessor function, we use equivalence to express Ackermann function.

$hyper(a, b, n)$, and $a[n]c$ are alternative ways of introducing hyperoperations.

We use hyperoperations for clarity, showing that we can indeed express Ackermann function as bounded iteration of function compositions.

Note that subsumption is necessary for *case*-expressions. Below we have typical rules for construction and destruction of basic types:

$$\frac{}{\Gamma \vdash_{\beta}^1 \cdot : \circ} \textit{unit}$$

$$\frac{\Gamma \vdash_{\beta}^{\alpha} e : A}{\Gamma \vdash_{\beta+1}^{\alpha+1} in_l(e) : A \vee B} \textit{inl} \quad \frac{\Gamma \vdash_{\beta}^{\alpha} e : B}{\Gamma \vdash_{\beta+1}^{\alpha+1} in_r(e) : A \vee B} \textit{inr}$$

$$\frac{\Gamma \vdash_{\beta_v+1}^{\alpha_v} a : L \vee R \quad \Gamma, x_{\beta_v} : L \vdash_{\beta_l}^{\alpha_l} l : B \quad \Gamma, y_{\beta_v} : R \vdash_{\beta_r}^{\alpha_r} r : B}{\Gamma \vdash_{\max(\beta_l, \beta_r)}^{\alpha_v + \max(\alpha_l, \alpha_r) + 1} case\ a\ of\ \begin{cases} in_l(x) \rightarrow & l; \\ in_r(y) \rightarrow & r; \end{cases} : B} \textit{case}$$

$$\frac{\Gamma \vdash_{\beta_a}^{\alpha_a} a : A \quad \Gamma \vdash_{\beta_b}^{\alpha_b} b : B}{\Gamma \vdash_{\max(\beta_a, \beta_b)+1}^{\alpha_a + \alpha_b + 1} (a, b) : A \wedge B} \textit{pair}$$

$$\frac{\Gamma \vdash_{\beta+1}^{\alpha} e : A \wedge B \quad i \in \{l, r\}}{\Gamma \vdash_{\beta}^{\alpha+1} prj_i e : A} \textit{prl} \quad \frac{\Gamma \vdash_{\beta+1}^{\alpha} e : A \wedge B \quad i \in \{l, r\}}{\Gamma \vdash_{\beta}^{\alpha+1} prj_r e : B} \textit{prr}$$

Please note that notation $A_v \rightarrow_{\beta}^{\alpha} B$ has a size variable v declared as a depth of *normal form proof term* having type A , and then bounds α and β apply to the computation of the result.

$$\frac{\Gamma, x_v : A \vdash_{\beta}^{\alpha} e : B \quad x, v \notin \Gamma}{\Gamma \vdash_{\beta \llbracket 1/v \rrbracket + 1}^{\alpha \llbracket 1/v \rrbracket + 1} \lambda x. e : A_v \rightarrow_{\beta}^{\alpha} B} \text{ abs}$$

Note that abstraction increases term depth by one, and application decreases it by one¹⁰. All introduction rules (*abs*, *pair*, *inl*, *inr*) increase β by at least one¹¹. Likewise all *non-functional* (data) elimination rules (*case*, *prl*, *prrr*) decrease depth expected from the resulting normal form β by one.

$$\frac{\Gamma \vdash_{\beta_1}^{\alpha_1} e : A_v \rightarrow_{\beta_2}^{\alpha_2} B \quad \Gamma \vdash_{\beta_3}^{\alpha_3} a : A}{\Gamma \vdash_{\beta_2 \llbracket \beta_3/v \rrbracket}^{\alpha_1 + \alpha_2 \llbracket \beta_3/v \rrbracket + \alpha_3} e a : B} \text{ app}$$

Please note that these rules all maintain bounded depth with no unbounded recursion. We add an explicit bounded recursive definition (like the definition of the closure) with this rule:

$$\frac{\Gamma \vdash_{\beta_1}^{\alpha_1} f : A_v \rightarrow_{\beta_2}^{\alpha_2} A \quad \Gamma \vdash_{\beta_3}^{\alpha_3} k : B \quad \Gamma \vdash_{\beta_4}^{\alpha_4} a : A}{\Gamma \vdash_{\beta_1 \llbracket iter(\lambda v. \beta_2, \beta_3, \beta_4)/v \rrbracket}^{\alpha_1 + iter(\lambda v. \alpha_2, \beta_3, \beta_4) + \alpha_3 + \alpha_4} rec(f, k, a) : B} \text{ rec}$$

Here the depth of the term must decrease at each step of the recursion. With the exception of *subsume*, and *rec* these are all reinterpretations of rules for intuitionistic logic [11, 75, 72], enriched with bounds on the proof length α and normalized depth of term t namely $|t|_d$ as depth expression β . The rule *rec* allows for explicitly *bounded recursion*, as opposed to traditional approaches that rely on an unbounded fixpoint¹².

Note that we may quantify on higher order values, but we cannot recurse indefinitely: there is always a limit to a number of function compositions allowed. Power of bounded function composition gives an explicit limit to Peano Arithmetic induction[38]: any computational application of Peano induction is unbounded. At the same time, we can use multiple recursions over bounded number of functions, terms, not just natural numbers. Wired-in explicit bounding also allows us to prove termination of arbitrary “towers” of function compositions, like hyperoperations [7, 33, 71, 62]¹³, including Goodstein functions that cannot be proven *within* PA itself[27].

2.3.1 Implicit universal quantification

In the propositional logic above, provability allows us to confirm statements with \forall for all variables on top. Given that statement of existence of bounded proof term x for witness bounded by result size v can be interpreted in the following way in unbound logic $\exists v \in \mathbb{N}^+. \exists x. |x|_d \leq v$.

¹⁰This allows us to correctly treat Church encoding.

¹¹For *unit*, the inner proof term would have null depth, since there is no term there. Thus depth is $0 + 1$ instead of $\beta + 1$.

¹²Fixpoint may not exist, thus leading not only to arbitrarily long computation, but also to undecidability in cases where computation may never end.

¹³See table 1 to see how hyperoperations and Ackermann function can be encoded using *iter* for bounded iteration of function composition.

So $a_v \rightarrow_{\beta}^{\alpha} b$ becomes the following statement in unbounded logics $\forall v. \forall a. (|a|_d < v) \rightarrow a \rightarrow b \wedge |b|_d < \beta \wedge c(b) < \alpha$. That is: we can infer that fact for all a below an arbitrarily large depth, and bound the depth and computational complexity of the resulting witness.

This concludes our treatment of Ultrafinitist Propositional Logic (UFPL).

2.3.2 Quantification with dependent types

It is customary in constructive mathematics and theorem proving to use dependent types instead of usual universal and existential quantifiers [52].

Please note that just like one can define intuitionistic propositional logic with just implication and then encode both sum and product types[36], so Π type can express both universal quantification and plain implication, while Σ type can express both existential quantification and product type. Since implication can already express sum and product types in polymorphic calculus, we will only show how to modify rules for implication and lambda to make the Π type that corresponds to universal quantification.

While it is usual to introduce universal quantification directly in calculi without proof terms, we will introduce them with Π types, like is now customary in dependently typed languages.

First we need a rule to introduce a type variable:

$$\frac{\Gamma \vdash_{\beta}^{\alpha} t \text{ type}}{\Gamma, v < \beta, x_v : t \vdash_v^1 x : t} \text{ tyvar}$$

This rule allows us to use variables at type level, and together with Π and Σ types allow to express quantification.

For the inequalities, it suffices to ensure that they are not cyclic and thus unsatisfiable. Note that inequality stems from the fact that value is always no longer than its encoding as a type.

$$\frac{\Gamma \vdash_{\beta_1}^{\alpha_1} A \text{ type} \quad \Gamma, x_v : A \vdash_{\beta_2(v)}^{\alpha_2(v)} B \text{ type} \quad x, v \notin \Gamma}{\Gamma \vdash_{\max(\beta_1, \beta_2 \llbracket 1/v \rrbracket) + 1}^{\alpha_1 + \alpha_2 \llbracket 1/v \rrbracket} \Pi(x_v : A) \rightarrow_{\beta_2}^{\alpha_2} B \text{ type}} \text{ forall-form}$$

Please note that similarly to the treatment of lambda abstraction as proof of implication, we estimate the computational cost of dependent product by substituting free variables with 1, but now we still need to consider the same substitution in the resulting Π type.

Treatment of universal quantifier bears usual similarity[52] to *abs* and *app* rules:

$$\frac{\Gamma, x_v : A \vdash_{\beta}^{\alpha} e : B}{\Gamma \vdash_{\beta \llbracket 1/v \rrbracket + 1}^{\alpha \llbracket 1/v \rrbracket + 1} \lambda(x : A). e : \Pi(x_v : A) \rightarrow_{\beta}^{\alpha} B} \text{ forall-intro}$$

Elimination works the same way as for usual application, since computation works after type erasure.

$$\frac{\Gamma \vdash_{\beta_1}^{\alpha_1} e : A_v \rightarrow_{\beta_2}^{\alpha_2} B \quad \Gamma \vdash_{\beta_3}^{\alpha_3} a : A}{\Gamma \vdash_{\beta_2 \llbracket \beta_3/v \rrbracket}^{\alpha_1 + \alpha_2 \llbracket \beta_3/v \rrbracket + \alpha_3} e a : B \llbracket A/x \rrbracket} \text{ forall-app}$$

This allows us to replace implication and by extension, all UFPL types. It also allows for quantification of higher order values.

We leave introducing Σ -types to interested students of type theory, since they are not essential to our argument that we may have a decidable higher order logic.

Since introduction creates the proof term in the same way, the proof terms can be enumerated in the same way as shown in section 4.4 on page 13.

■ **Table 2** Simplification of bounds. May rewrite left to right.

(1)	$a * x^e + b * x^f \leq (a + b) * x^f$
(2)	$a * x^e * y^g \leq a * x^f * y^h$
(3)	$iter(\lambda v.e, g, x) \leq iter(\lambda w.f, h, x)$
(4)	$iter(\lambda v.v * a, e, x) = a^e * x$
(5)	$iter(\lambda v.v + a, e, x) = x + a * e$
(6)	$iter(\lambda v.v^e, g, x) = x^{e^g}$
Assumptions:	
■	$x, y \geq 1$ are data size variables in the environment,
■	$1 \leq e \leq f$ and $1 \leq g \leq h$ are arbitrary positive and increasing expressions,
■	$a, b, c... \geq 1$ are constants.

3 Application of the logic

3.1 Using proofs

Each proof ultimately leads to a judgment $\Gamma \vdash_{\beta}^{\alpha} e : A$. We may resolve all upper bound variables v_1, v_2, \dots, v_n in the α to get an upper bound on computational complexity of the statement, and in β to get an upper bound on normalized term resulting from the proof. This way all proofs are ultra-finitary statements: Only as long as α is less than our assumed limit, we will consider the proof valid and proof computation to be available within the given time.

3.2 Simplifying upper bounds

Our inference rules rely on computing upper bounds and their inequality. Here we note a few inequalities that simplify reasoning about these bounds, albeit at the cost of making them somewhat looser.

First, we note that all variables are positive naturals because they represent the data of non-zero size: $x \geq 1$.

That means that the following laws are true, assuming that $x, y, \dots \geq 1$ are data size variables in the environment, $1 \leq e \leq f$ and $1 \leq g \leq h$ are arbitrary positive expressions, and $a, b, c... \geq 1$ are constants. For easier use, the rules are presented in left-to-right order, just like conventional rewrite rules.

We may thus use these rules to loosen the bound in such a way as to reduce the size of the bound expression and make it a sum of a single term in all variables and an additional constant term. This reduction may be delayed until we have bound to verify.

We may use inference rules leaving “type holes”[57] instead of bounds, which could be named “bound holes”, and let them be filled by the framework interpreter.

3.3 Reduction

Reduction relation is defined as small step semantics [60] in order to preserve number of computational steps made over the course of evaluation. See table 3 on page 9.

When performing application, we expect substitution to take work proportional to the number of occurrences of the variable, like changing links on directed acyclic graph of the term.

■ **Table 3** Reduction rules.

$\frac{e \Downarrow_k e'}{\text{case } e \text{ of } \begin{cases} \text{in}_l(x) \rightarrow b; \\ \text{in}_r(y) \rightarrow c; \end{cases} \Downarrow_k \text{case } e' \text{ of } \begin{cases} \text{in}_l(x) \rightarrow b; \\ \text{in}_r(y) \rightarrow c; \end{cases}} \text{eval-case-arg}$
$\frac{k = \text{occurs}(x, b)}{\text{case } \text{in}_l(a) \text{ of } \begin{cases} \text{in}_l(x) \rightarrow b; \\ \text{in}_r(y) \rightarrow c; \end{cases} \Downarrow_k b \llbracket a/x \rrbracket} \text{eval-case-left}$
$\frac{k = \text{occurs}(y, c)}{\text{case } \text{in}_r(a) \text{ of } \begin{cases} \text{in}_l(x) \rightarrow b; \\ \text{in}_r(y) \rightarrow c; \end{cases} \Downarrow_k c \llbracket a/y \rrbracket} \text{eval-case-right}$
$\frac{k = \text{occurs}(x, e)}{(\lambda x. e) f \Downarrow_k e \llbracket f/x \rrbracket} \text{eval-app}$
$\frac{e \Downarrow_n e' \quad i \in \{l, r\}}{\text{in}_i(e) \Downarrow_n \text{in}_i(e')} \text{eval-sum}$
$\frac{}{\text{prj}_l(a, b) \Downarrow_1 a} \text{eval-prl} \qquad \frac{}{\text{prj}_r(a, b) \Downarrow_1 b} \text{eval-prr}$
$\frac{a_l \Downarrow_n a'_l}{(a_l, a_r) \Downarrow_n (a'_l, a_r)} \text{eval-pairleft}$
$\frac{a_r \Downarrow_n a'_r}{(a_l, a_r) \Downarrow_n (a_l, a'_r)} \text{eval-pairright}$

For discussion of efficient reduction of lambda terms please read [45, 5], since here we focus on demonstration with a simplified cost model.

3.4 Self-encoding

3.4.1 Natural numbers

In this section we will encode bounds, propositions (types) and proof terms as proof terms within UFPL. Thus $\llbracket \cdot \rrbracket$ corresponds to LISP quote.

Below we use notation $\mathbb{B}(v)$ for de Bruijn index of the variable [15].

■ **Table 4** Encoding natural numbers.

Nat_β	$=$	$(\text{rec}(x, [\circ \vee x], \beta) [\circ])$	
zero	$=$	$\text{in}_l(\cdot)$	$:\!:\!_1^1 \quad \text{Nat}_1$
succ	$=$	$\lambda x_v \rightarrow_{v+1}^1 \text{in}_r(x)$	$:\!:\!_{v+1}^1 \quad \text{Nat}_v \rightarrow \text{Nat}_{v+1}$

 ■ **Table 5** Encoding bounds.

Var_β	$=$	Nat_β
$\text{Bound}_{\beta+1}$	$=$	$\text{Var} \vee \text{Nat}_\beta \vee \circ \vee (\text{Bound}_\beta, \text{Bound}_\beta)$
		$\vee (\text{Bound}_\beta, \text{Bound}_\beta)$
		$\vee (\text{Bound}_\beta, \text{Bound}_\beta)$
		$\vee (\text{Bound}_\beta, (\text{Bound}_\beta, \text{Var}))$
		$\vee (\text{Bound}_\beta, (\text{Var}, \text{Bound}_\beta))$
$\llbracket v \rrbracket$	$=$	$\text{in}_l(\text{in}_l(\text{in}_l(\mathbb{B}(v))))$
$\llbracket i \rrbracket$	$=$	$\text{in}_l(\text{in}_l(\text{in}_r(i)))$
$\llbracket \cdot \rrbracket$	$=$	$\text{in}_l(\text{in}_l(\text{in}_r(\cdot)))$
$\llbracket \rho_1 + \rho_2 \rrbracket$	$=$	$\text{in}_l(\text{in}_r(\text{in}_r(\llbracket \rho_1 \rrbracket, \llbracket \rho_2 \rrbracket))))$
$\llbracket \rho_1 * \rho_2 \rrbracket$	$=$	$\text{in}_r(\text{in}_l(\text{in}_l(\llbracket \rho_1 \rrbracket, \llbracket \rho_2 \rrbracket))))$
$\llbracket \rho_1^{\rho_2} \rrbracket$	$=$	$\text{in}_r(\text{in}_l(\text{in}_r(\llbracket \rho_1 \rrbracket, \llbracket \rho_2 \rrbracket))))$
$\llbracket \text{iter}(\lambda v. \rho_1, \rho_2, \rho_3) \rrbracket$	$=$	$\text{in}_r(\text{in}_r(\text{in}_l(\mathbb{B}(v), \llbracket \rho_1 \rrbracket, (\llbracket \rho_2 \rrbracket, \llbracket \rho_3 \rrbracket))))$
$\llbracket \rho_1 \llbracket \rho/v \rrbracket \rrbracket$	$=$	$\text{in}_r(\text{in}_r(\text{in}_r(\llbracket \rho_1 \rrbracket, (\llbracket \rho_2 \rrbracket, \mathbb{B}(v))))))$

3.4.2 Encoding bounds

Now we may encode bounds (table 5), types (table 6), and proof terms (table 7).

This encoding allows us to make operations on types akin to generic programming in Haskell [50].

Our inference rules rely on computing bounds and their inequalities. Given that all variables are positive naturals because they represent the data of non-zero size: $x \geq 1$, we may simplify these bounds with a set of simple inequalities.

3.4.3 Encoding proof terms

Note that every type term in *normal form* is longer than its own type.

► **Theorem 1** (Encoding). *All bound, type, proof, or proposition of UFPL can be encoded as a proof term of UFPL.*

Details are visible in the tables 4-7.

 ■ **Table 6** Encoding types.

$\llbracket A \vee B \rrbracket$	$=$	$\text{in}_l(\text{in}_l(\llbracket A \rrbracket, \llbracket B \rrbracket))$
$\llbracket A \wedge B \rrbracket$	$=$	$\text{in}_l(\text{in}_r(\llbracket A \rrbracket, \llbracket B \rrbracket))$
$\llbracket A_v \rightarrow_\beta^\alpha B \rrbracket$	$=$	$\text{in}_r(\text{in}_l(\llbracket A \rrbracket, (\lambda x : A. \llbracket B \rrbracket, (\lambda v : \text{Nat}_v. \llbracket \alpha \rrbracket, \lambda v : \text{Nat}_v. \llbracket \beta \rrbracket))))$
$\llbracket \circ \rrbracket$	$=$	$\text{in}_r(\text{in}_r(\cdot))$

■ **Table 7** Encoding terms.

$\llbracket x_v \rrbracket$	$=$	$in_l (in_l (in_l (in_l (\mathbb{B}(x), v))))$
$\llbracket subsume(A, B) \rrbracket$	$=$	$in_l (in_l (in_l (in_r (\llbracket B \rrbracket_{\text{Bound}}, \llbracket A \rrbracket))))$
$\llbracket unit \rrbracket$	$=$	$in_l (in_l (in_r (in_l (\cdot))))$
$\llbracket in_l(A) \rrbracket$	$=$	$in_l (in_l (in_r (in_r (A))))$
$\llbracket in_r(A) \rrbracket$	$=$	$in_l (in_r (in_l (in_l (A))))$
$\llbracket prj_l A \rrbracket$	$=$	$in_l (in_r (in_l (in_r (A))))$
$\llbracket prj_r A \rrbracket$	$=$	$in_l (in_r (in_r (in_l (A))))$
$\llbracket (A, B) \rrbracket$	$=$	$in_l (in_r (in_r (in_r (\llbracket A \rrbracket, \llbracket B \rrbracket))))$
$\llbracket AB \rrbracket$	$=$	$in_r (in_l (in_l (in_l (\llbracket A \rrbracket, \llbracket B \rrbracket))))$
$\llbracket \lambda x_v. A \rrbracket$	$=$	$in_r (in_l (in_l (in_r (\mathbb{B}(x), \mathbb{B}(v)), \llbracket A \rrbracket))))$
$\llbracket rec(v, A, B)C \rrbracket$	$=$	$in_r (in_l (in_r (in_l (\mathbb{B}(v), \llbracket A \rrbracket), (\llbracket B \rrbracket, \llbracket C \rrbracket))))$

4 Properties of the logic

When implementing the computation seems straightforward, we will just establish the finite limit for the computation that should be taken as a proof. That is what we describe as *problem is decidable by the limit of a given complexity*. This approach explicitly describes undecidable problems as those that require an infinite number of steps to solve.

4.1 Consistency

Here we will only use well-known proof of consistency of intuitionistic logic [11, 75, 72]¹⁴. We do not use the self-encoding presented in section 3.4.

► **Theorem 2** (Consistency of UFL). *UFPL is consistent, if intuitionistic propositional logic is consistent.*

Proof. After elision of bounds¹⁵, we interpret the rule *subsume* as $id = \lambda x.x$. Then we see the standard proof rules for intuitionistic logic. The consistency follows from the consistency of intuitionistic logic. ◀

4.2 Expressivity

► **Theorem 3** (At least as expressive as PRA.). *UFL can express all Primitive Recursive programs.*

Proof. It is easy to show that our logic can emulate bounded loop programs[53] which has power equivalent to primitive recursive functions[64]. Every bounded *loop* can be encoded by *iter*($\lambda v.loop, x, n$), then every flat logical statement can be encoded with a tuple containing states of the variables. ◀

One could muse that this class does not cover all Bounded Turing Machine[34] programs. In order to support these, we would need to define more general bounding functions.

¹⁴The proof above is totally independent of previous conjectures.

¹⁵Elision of bounds is only used once to prove consistency.

One can replace upper bound expressions with arbitrary bounding functions expressed in simply typed lambda calculus (see section 3.4). These are the operations used in inference rules. However, such functions are more difficult to bound and compute themselves.

It has been proven that any function whose complexity is *bounded* by primitive recursive function is also primitive recursive[16], which means that estimating our complexities could become an impossibly long endeavour, but logically consistent one.

To give an example of simplified Ackermann function which is the best known example of function beyond PRA [70, 3], evaluation takes $A(5) = 2^{2^{2^{2^{16}}}} - 3$ [40]. That means that these evaluations quickly get out of hand and indeed outside of any reasonable limits.

The encoding of Ackermann function is through hyperoperation in table 1.

4.3 Bounded Turing completeness

An evidence of stronger expressivity may be found by encoding bounded Turing Machine programs in UFPL. This proof uses encoding similar to 3.4, but for a Turing Machine. For a reference on encoding of Turing machines in lambda calculus see [2].

For any complexity bound $f(x)$ expressible in the language of ultrafinitist logic, and an algorithm that satisfies it and emulation function with complexity of $e(f)(x)$ – that is an encoding $e(f)$ of f , applied to the argument x of f – which we can encode this emulation as a bound.

► **Theorem 4** (Emulation complexity). *Assume a time complexity $c(x)$ for program (or proof) s that can be encoded as UFL bounds. If we can emulate (encode evaluation) of $f(x)$ with an overhead e for each step, then we can prove that complexity of evaluating s is $e * c(x) + cc(x)$.*

Where $cc(x)$ is complexity of evaluating complexity bounds for the encoding $c(x)$.

Proof. Given each step of emulation encoded as $s(x)$, where x is a current state, emulation with a complexity function encoded as $f(x)$ can be executed by $iter(\lambda v.s, f(x), x)$.

Assuming that $e(f)$ is function emulation in UFPL, we can write proof expression $iter(\lambda x.e(f)(x), e(c)(x), x)$. This expression evaluated encoded s and has exactly the assumed complexity ◀

The most complex part of the proof may be logically inferring the right complexity $c(x)$ and totality of the function f within this number of steps.

► **Theorem 5** (Bounded Turing Machine emulation). *For programs of Bounded Turing Machine f over alphabet size $|a|$ and number of states $|s|$ with complexity that can be encoded in UFL, we can prove time complexity of $\lg_2(|a|) + \lg_2(|s|) * |c| + |cc|$ with UFPL. Note that $|cc|$ is cost to evaluate complexity function itself.*

Proof. We use emulation argument for Bounded Turing machines that may be limited by bounds described above, it is too with $\mathcal{O}(\log^2(a) * f(x))$, where a is the bound on the size of alphabet and number of states of the machine.

For the Bounded Turing machine we encode tape as pair of lists, with current position at the top of both lists.

Then we encode the following steps:

- examine the alphabet character: $\mathcal{O}(\log(|a|))$
- examine finite state machine for a character: $\mathcal{O}(\log(|s|))$
- move one step up or down the tape by moving the top from one line list to the other: $\mathcal{O}(1)$;

- if we want to write at the current position, we take the top element from the right list, and put the new one.

Together they make a single step of the Turing machine at the cost of $\mathcal{O}(\log(|a|) + \log(|s|))$.

◀

We encode variable bindings as a dictionary with cost of $\mathcal{O}(\lg_2 |\text{Var}|)$, where Var is number of variables used. All operations not involving substitution should remain at $\mathcal{O}(1)$ complexity within emulation.

► **Lemma 6** (Self-emulation). *ULF self-emulation of function with integral bound $|cc|$ is feasible within $\mathcal{O}(|cc| * \lg_2 |\text{Var}|)$.*

Overall we can infer that for each algorithm of bounded complexity B that we may encode in ULF, we may use ULF self-emulation to find a proof with complexity of at most $e * B * \lg_2 |\text{Var}|$. All steps of ULF are $\mathcal{O}(1)$ with respect to inferred bounds on computational complexity, with the exception of function application and variable substitution which are $\mathcal{O}(\lg_2 |\text{Var}|)$

► **Theorem 7** (Emulation completeness). *If the bounds that can be encoded within the bounds function, the UFPL is complete for proving its own bounds up to the cost of self-emulation e .*

Since we can encode any statement in UFPL in UFPL itself, this likely would mean the proof of emulation completeness can be written in the UFPL itself.

There are complex ways of proving completeness that apply in the realm of non-idempotent intersection type systems, but they use a more abstract notion of complexity[1].

4.4 Decidability of bounded statements

► **Theorem 8** (Decidability). *Every valid proposition with a fixed bound on input n can be checked by enumerating inputs, and is thus decidable.*

This comes at the cost of complexity that increases by $\alpha(n) * a^n$, where n is the depth of input, since we need to enumerate all inputs of depth n . Proof follows directly from enumeration, and bounds.

Proof. Let's try to enumerate the terms that can be constructed for a given bounds, without using *subsume* rule:

Bounds function will contain a given number n of successor functions, and *iter* expressions. Each time we make a single inference rule, and construct a slightly more complex term, we add a successor function or *iter* expression, the number of different proof tree shapes equals to the number of ternary trees constructible with n nodes. This number is defined as OEIS A001764 [39] and given by the algebraic term $\frac{\binom{3n}{n}}{2n+1}$. Given that for each of n nodes we can

choose one of the 12 rules (12^n different choices), we have at most $\mathcal{O}\left(\frac{12^n \binom{3n}{n}}{2n+1}\right)$ different

proofs with the complexity given by a term of n nodes.

With the inclusion of subsumption rule, we can only decrease the n , so at most:

$$\sum_{i=1..n} \mathcal{O}\left(\frac{12^n}{2n+1} \binom{3n}{n}\right) = \mathcal{O}\left(12^n \binom{3n}{n}\right)$$

Since number of proofs is finite, we can decide the provability after they are exhausted. ◀

Thus all judgements with bounds are decidable. This property is shared with some other *resource bounded logics* [1].

Given that exponential lower bound has been established for implicational intuitionistic logic [37], we expect that lower bound for ultrafinitist logic will also be exponential and thus the proposed bound is asymptotically tight.

4.5 Paradox of undecidability

Expressing any statements about undecidability implicitly requires unbounded computational effort. Since all our proofs and arguments are explicitly bounded, there is no room to state undecidability. Thus we conclude that this paradox is removed from ultrafinitist logic: statement of undecidability is *invalid* as a proposition. All *valid propositions* are decidable.

This is not as outrageous as it superficially seems, since we already know that computation models that would allow transfinite number of steps would also make all functions computable [30].

4.6 Finitary completeness

Let's assume we have upper bounds on all variables within an intuitionistic theorem.

Can we prove it with UFL?

► **Theorem 9** (Preservation of bounded intuitionistic theorems). *Any intuitionistic theorem bounded by definite integers in UFL can be proven in UFL.*

Proof. Let's enumerate complexities of computing intuitionistic proof for a given set of inputs bounded by given value. We may enumerate these proofs, and thus take maximum length of the computation. This maximum length will be upper bound on all proofs. ◀

This proof uses 4.4, and 3.4. Naturally this means that all statements with bounds but proof without bounds will also have proof with bounds.

5 Related work

The philosophical problem with transfinite arguments has been spotted long before [42, 61, 79, 22, 47]. Automatic theorem provers like Coq require a monotonically decreasing bounding function in the ordinal domain for each inductive definition [56, 58, 8]. This makes all recursive definitions *well-founded* [56], but since transfinite ordinals are permitted, it also allows theories outside computable universe.

The computation of a bounding function may turn out to take unfeasible amount of time. Cost calculi for functional languages attempt to assign cost to certain operations in order to reason about time and space complexity [65]. But these approaches do not require all proofs and propositions to carry the cost as we do.

Philosophers have postulated distinction between feasible computations and unfeasible ones [79], however it was considered unclear whether it is possible to realize this distinction on the basis of a logic [74], with some claiming that such a logic could not be consistent [17, 51].

There exist logics that implicitly constrain computational complexity of the proofs, for example Bounded Arithmetic [12, 43, 14] that is restricted to computations in polynomial time. However, most of them are significantly weaker than class of primitive recursive functions, which is widely considered to contain most useful programs. This would put the logician in a position of trying to state a widely known facts about objects that are inexpressible within the logic.

6 Discussion

6.1 Explicit bounds versus implicit structural recursion

It is long known that unbounded logics may give rise to paradoxes [23, 13], and the use of implicit techniques [14], including bounded recursion [43], structural recursion [59], well-founded sets [56], or predicative bounding [21] were developed.

Using explicit bounds provides a more obvious solution, which is easier to prove correct, and parallels development of an explicit mathematical limit [69, 29], starting from Eudoxus' method of exhaustion [18], through implicit notion of terminus [76] to a modern concept of a mathematical limit of a function [78]¹⁶.

6.2 Open problem of directly proving bounded Turing completeness

Note that the proof above mentions Turing completeness, if we can prove that all bounds can be expressed by the bounds functions defined above. While the usual examples of fast growing functions like Ackermann [70, 3, 40], or Goodstein [27, 38] are expressible by bounded composition of functions, the clarity is still elusive. (Of course such fast growing functions would quickly surpass any reasonable limit.)

We still search for a proof of bounded Turing completeness that would not use a recursive argument, where we replace bound functions with arbitrary bounded lambda expressions. That is because our induction principle would have to be more complex to include the latter.

Interested student may prove Turing completeness by encoding to Kleene normal form with iteration on top [31]. In this logic, Kleene normal form may be made explicit.

6.3 Computability as foundation of mathematics

Finite descriptions of the proofs and their objects are most rational foundations of mathematics. These objects are all definable by bounded Turing computability.

Attempts to define *hypercomputation* beyond bounded Turing machine immediately lead to physical impossibilities [54]. At the same time computability or bounded Turing machine and total computable functions have been translated between multiple mathematical models. Hence we conjecture that the only mathematical proof principle that is immune to rational doubt is the bounded Turing machine, and ultrafinitist logic.

A logic that allows expression of any bounded Turing function and nothing else could be rightly called a logic of computable functions, and a best candidate for encoding foundations of mathematics. Alternative attempts to narrow set theory by predicativism [21, 68] are subject to critique[77] that motivates further search.

That is because we can encode axioms that are incomputable as function parameters with assumed types, and use these to prove or disprove theorems of traditional axiomatic theories without endangering consistency of the underlying logical framework.

¹⁶Interestingly delta-epsilon definition is formalizable for computable functions, by assuming that as n approaches the limit the smaller computable environment is taken. Of course both delta and epsilon would have to be a finite expansions (approximations) instead of possibly transcendental value. This would give a definition of "computable limit".

6.4 Automated theorem proving

Interesting avenue for future work would be to define a full type theory, dependently typed language and an automatic prover for these inference rules. Improving on the bound of $\mathcal{O}\left(\binom{3n}{n}\right)$ for deciding subtheorems would be possible, since we only need to consider normal forms. It would be exciting to prove metatheoretic results about the UFL in itself, and verify it with an automatic theorem prover.

Since meta-reasoning always results in longer proofs than original theorems, the UFL may also allow us to prove consistency of ultrafinitist arithmetic, enabling to second Hilbert problem [32], and potentially allowing self-verifiable formalization of mathematics.

Theories for uncomputable are only indirectly formalisable within such framework as functions taking uncomputable actions (like infinite recursor of Peano arithmetic) as arguments. Previously created theories are prone to high complexity and errors due to difficulty at maintaining expressivity and consistency together. Simplicity of proving the hierarchy of universes as hierarchy of complexities, and expanding ultrafinitist logic with strongly normalizable dependent types gives us hope that such automated theorem proving framework would be simpler.

Since the logic includes upper bounds for all functions, we may use these and proof irrelevance to automatically and safely optimize proofs as well. For example, we could automatically replace computation of naturals defined by successor function with computation defined on positional binary numbers.

6.5 Proving decidability in strictly finite domain

The explicit bounding of all objects, including proofs in this work is used to prevent undecidability within finite domain [63].

7 Conclusion

We have shown a possible consistent logic for inference with a strictly bounded number of steps. This allows us to limit our statements by the length of acceptable proof, and thus define statements that are both true, and computable within Bremermann-Gorelik limit [28] This inference system explicitly bounds both the length of the resulting proof, and the bounds on the depth of the normalized result term. This allows avoiding inconsistencies suggested by philosophical work, and at the same time steers away from relatively weak logics with implicit complexity like Bounded Arithmetic [43], which capture polynomial time hierarchy. It also shows how much we gain by making explicit bounds, since these may be tighter than with implicit complexity approaches. While Emulation Complexity is the powerful approach to proving expressivity of this logic, it would be nice to see a proof with tighter bounds on what we may prove with it¹⁷.

We strive to prove that all bounded computable functions are expressible within this framework, and thus we propose this logic as a “logic of practical computability”.

¹⁷A promising avenue of work would be proving that *amortized complexity* by replacing single bound variable by a vector of monotonic bound variables. Another approach would be attempt to obtain tighter bounds directly by separately counting beta-reduction steps and substitutions, instead of all reduction steps and substitutions together [1].

References

- 1 Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018. doi:10.1145/3236789.
- 2 Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. A log-sensitive encoding of turing machines in the λ -calculus, 2023. arXiv:2301.12556.
- 3 Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, December 1928. doi:10.1007/BF01459088.
- 4 Yehoshua Bar-Hillel and Rudolf Carnap. Semantic information. *British Journal for the Philosophy of Science*, 4(14):147–157, 1953. doi:10.1093/bjps/IV.14.147.
- 5 Pablo Barenbaum and Eduardo Bonelli. Optimality and the Linear Substitution Calculus. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2017.9.
- 6 Jose Benardete. *Infinity: An Essay in Metaphysics*. Clarendon Press, 1964.
- 7 Albert A. Bennett. Note on an operation of the third grade. *Annals of Mathematics. Second Series*, 17 (2):74–75, December 1915. doi:10.2307/2007124.
- 8 Yves Bertot and Vladimir Komendantsky. Fixed point semantics and partial recursion in Coq. In *MPC 2008*, Marseille, France, July 2008. URL: <https://hal.inria.fr/inria-00190975>.
- 9 Tobias Boolakee, Christian Heide, Antonio Garzon-Ramirez, Heiko B. Weber, Ignacio Franco, and Peter Hommelhoff. Light-field control of real and virtual charge carriers. *Nature*, 605(7909):251–255, May 2022. doi:10.1038/s41586-022-04565-9.
- 10 Manuel Bremer. Varieties of finitism. *Metaphysica*, 8:131–148, October 2007. doi:10.1007/s12133-007-0012-9.
- 11 l. e. j. Brouwer. *Over de grondslagen der wiskunde*, volume 1 of *mc varia*. Mathematisch Centrum, Amsterdam, 1981. including unpublished fragments, correspondence with D. J. Korteweg and reviews by G. Mannoury, edited and with an introduction by D. Van Dalen.
- 12 Samuel R. Buss. The polynomial hierarchy and intuitionistic bounded arithmetic. In Alan L. Selman, editor, *Structure in Complexity Theory, Proceedings of the Conference hold at the University of California, Berkeley, California, USA, June 2-5, 1986*, volume 223 of *Lecture Notes in Computer Science*, pages 77–103. Springer, 1986. doi:10.1007/3-540-16486-3_91.
- 13 Thierry Coquand. An analysis of girard’s paradox. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*, pages 227–236. IEEE Computer Society Press, June 1986.
- 14 Ugo Dal Lago. Implicit computation complexity in higher-order programming languages: A survey in memory of martin hofmann. *Math. Struct. Comput. Sci.*, 32(6):760–776, 2022. doi:10.1017/S0960129521000505.
- 15 Nicolaas Govert De Bruijn. *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*, volume 75, pages 381–392. Elsevier, 1972.
- 16 P. J. Denning, J. B. Dennis, and J. E. Qualitz. *Machines, Languages, and Computation*. Prentice-Hall, 1978.
- 17 Michael Dummett. Wang’s paradox. *Synthese*, 30(3/4):301–324, 1975. URL: <https://link.springer.com/article/10.1007/BF00485048>.
- 18 Liu Dun. A comparison of archimedes’ and liu hui’s studies of circles. In *Chinese Studies in the History and Philosophy of Science and Technology 179*, pages 279–287. Kluwer Academic Publishers, 1966.
- 19 Marie Duzi. The paradox of inference and the non-triviality of analytic information. *Journal of Philosophical Logic*, 39:473–510, October 2010. doi:10.1007/s10992-010-9127-5.
- 20 Lisa Dyson, Matthew Kleban, and Leonard Susskind. Disturbing implications of a cosmological constant. *Journal of High Energy Physics*, 2002(10):011, November 2002. doi:10.1088/1126-6708/2002/10/011.

- 21 Solomon Feferman. Systems of predicative analysis. *Journal of Symbolic Logic*, 29:1–30, 1964. URL: <https://api.semanticscholar.org/CorpusID:35126801>.
- 22 Amanda Geffer. Mind-bending mathematics: Why infinity has to go. *New Scientist*, 219(2930):32–35, 2013. doi:10.1016/S0262-4079(13)62043-6.
- 23 Jean-Yves Girard. *Interprétation fonctionnelle et Élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris Diderot - Paris 7, 1972. URL: <https://www.cs.cmu.edu/~kw/scans/girard72thesis.pdf>.
- 24 Nicolas Gisin. Indeterminism in physics, classical chaos and bohmian mechanics. are real numbers really real?, 2019. arXiv:1803.06824.
- 25 Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, December 1931. doi:10.1007/BF01700692.
- 26 Kurt Gödel, S. Feferman, J.W. Dawson, S.C. Kleene, G. Moore, R. Solovay, and J. van Heijenoort. *Kurt Gödel: Collected Works: Volume I: Publications 1929-1936*. Collected Works of Kurt Godel. OUP USA, 1986. URL: <https://books.google.pl/books?id=5ya4A0w62skC>.
- 27 R. L. Goodstein. On the restricted ordinal theorem. *The Journal of Symbolic Logic*, 9(2):33–41, 1944. doi:10.2307/2268019.
- 28 Gennady Gorelik. Bremermann's limit and cgh-physics, 2010. arXiv:0910.3424.
- 29 Judith V. Grabiner. Who gave you the epsilon? cauchy and the origins of rigorous calculus. *The American Mathematical Monthly*, 90(3):185–194, 1983. URL: <http://www.jstor.org/stable/2975545>.
- 30 Joel David Hamkins. Every function can be computable! Blog post., March 2016. URL: <http://jdh.hamkins.org/every-function-can-be-computable/>.
- 31 Hans Hermes. *Enumerability, decidability, computability - an introduction to the theory of recursive functions*. Springer, 1969.
- 32 David Hilbert. Mathematical problems. *Bulletin of the American Mathematical Society*, 8(10):437–479, 1902.
- 33 David Hilbert. Über das unendliche. *Mathematische Annalen*, 95:161–190, 1926. doi:10.1007/BF01206605.
- 34 John E. Hopcroft and Jeffrey D. Ullman. Relations between time and tape complexities. *J. ACM*, 15(3):414–427, July 1968. doi:10.1145/321466.321474.
- 35 William A. Howard. The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, λ -calculus and Formalism*, pages 479–490. Academic Press, 1980.
- 36 Jan Martin Jansen. *Programming in the λ -Calculus: From Church to Scott and Back*, pages 168–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-40355-2_12.
- 37 Emil Jeřábek. A simplified lower bound for implicational logic, 2023. arXiv:2303.15090.
- 38 Laurie Kirby and Jeff Paris. Accessible independence results for peano arithmetic. *Bulletin of the London Mathematical Society*, 14(4):285–293, 1982. doi:10.1112/blms/14.4.285.
- 39 Don Knuth, Frank Ellerman, and Natan Arie Consigli. A001764 - $a(n) = \text{binomial}(3*n, n) / (2*n + 1)$ (enumerates ternary trees and also noncrossing trees). (formerly m2926 n1174), 2016. URL: <https://oeis.org/A001764>.
- 40 Don Knuth, Frank Ellerman, and Natan Arie Consigli. A046859: Simplified ackermann function (main diagonal of ackermann-péter function), 2016. URL: <https://oeis.org/A046859>.
- 41 Cynthia Kop and Deivid Vale. Tuple Interpretations for Higher-Order Complexity. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, volume 195 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2021.31.
- 42 Andras Kornai. Explicit finitism. *International Journal of Theoretical Physics*, 42:301–307, February 2003. doi:10.1023/A:1024451401255.

- 43 Jan Krajčec. *Bounded Arithmetic, Propositional Logic and Complexity Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1995. doi:10.1017/CB09780511529948.
- 44 Lawrence M. Krauss and Glenn D. Starkman. Universal limits on computation. Preprint arXiv., 2004. arXiv:astro-ph/0404510.
- 45 John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '90*, pages 16–30, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/96709.96711.
- 46 Jean-Luc Lehnens and Jerome Quintin. A small universe, 2023. arXiv:2309.03272.
- 47 Jonathan Lenchner. A finitist's manifesto: Do we need to reformulate the foundations of mathematics?, 2020. arXiv:2009.06485.
- 48 Seth Lloyd. Ultimate physical limits to computation. *Nature*, 406(6799):1047–1054, August 2000. doi:10.1038/35023282.
- 49 Seth Lloyd. Computational capacity of the universe. *Physical review letters*, 88 23:237901, 2002.
- 50 José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löb. A generic deriving mechanism for haskell. *SIGPLAN Not.*, 45(11):37–48, September 2010. doi:10.1145/2088456.1863529.
- 51 Ofra Magidor. Strict finitism refuted? *Proceedings of the Aristotelian Society*, 107(1pt3):403–411, 2007. doi:10.1111/j.1467-9264.2007.00230.x.
- 52 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.
- 53 Albert R. Meyer and Dennis M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd National Conference*, ACM '67, pages 465–469, New York, NY, USA, 1967. Association for Computing Machinery. doi:10.1145/800196.806014.
- 54 Aran Nayebi. Practical intractability: A critique of the hypercomputation movement. *Minds and Machines*, 24:275–305, 2012. URL: <https://api.semanticscholar.org/CorpusID:9328713>.
- 55 Daniel Nolan. Lessons from infinite clowns (1st edition). In Karen Bennett and Dean Zimmerman, editors, *Oxford Studies in Metaphysics Vol. 14*. Oxford University Press, forthcoming.
- 56 B. Nordström. Terminating general recursion. *BIT*, 28(3):605–619, July 1988. doi:10.1007/BF01941137.
- 57 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290327.
- 58 Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. 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 328–345. Springer, 1993. doi:10.1007/BFb0037116.
- 59 Rózsa Péter. Über die verallgemeinerung der theorie der rekursiven funktionen für abstrakte mengen geeigneter struktur als definitionsbereiche. *Acta Mathematica Academiae Scientiarum Hungaricae*, 12(3):271–314, May 1964. doi:10.1007/BF02023919.
- 60 Gordon D. Plotkin. A structural approach to operational semantics, 2004.
- 61 Karlis Podnieks. Towards a real finitism? Web page published online., December 2005. URL: <http://www.ltn.lv/~podnieks/finitism.htm>.
- 62 António G. Porto and Armando B. Matos. Ackermann and the superpowers. *SIGACT News*, 12(3):90–95, September 1980. doi:10.1145/1008861.1008872.
- 63 Pavel Pudlák. Incompleteness in the finite domain. *Bulletin of Symbolic Logic*, 23(4):405–441, 2017. doi:10.1017/bsl.2017.32.
- 64 Raphael M. Robinson. Primitive recursive functions. *Bulletin of the American Mathematical Society*, 53(10):p. 925–942, 1947. doi:bams/1183511140.

- 65 David Sands. *Calculi for time analysis of functional programs*. PhD thesis, University of London, 1990.
- 66 Vladimir Yu. Sazonov. On feasible numbers. In Daniel Leivant, editor, *Logic and Computational Complexity*, pages 30–51, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- 67 Matthias Schirn and Karl-Georg Niebergall. Finitism = pra? on a thesis of w. w. tait. *Reports on Mathematical Logic*, January 2005.
- 68 Kurt Schütte. Predicative well-orderings. *Studies in logic and the foundations of mathematics*, 40:280–303, 1965. URL: <https://api.semanticscholar.org/CorpusID:117343586>.
- 69 Galina Iwanowna Sinkiewicz. On history of epsilonotics. *Antiquitates Mathematicae*, 10(0):183–204, 2017. doi:10.14708/am.v10i0.805.
- 70 G. Sudan. Sur le nombre transfini ω^ω . *Bulletin Mathématique de la Société Roumaine des Sciences*, 30:11–30, 1927.
- 71 Yngve Sundblad. The ackermann function. a theoretical, computational, and formula manipulative study. *BIT Numerical Mathematics*, 11(1):107–119, March 1971. doi:10.1007/BF01935330.
- 72 Morten Heine b. Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Studies in Logic and Foundations of Mathematics. Elsevier, 1998. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7385>.
- 73 Alfred Tarski, Andrzej Mostowski, and Raphael M. Robinson. Undecidable theories. *Philosophy*, 30(114):278–279, 1955.
- 74 A. S. Troelstra. *Constructivism in Mathematics: An Introduction*. Elsevier, 1988. URL: <https://www.sciencedirect.com/bookseries/studies-in-logic-and-the-foundations-of-mathematics/vol/121/suppl/C>.
- 75 Dirk Van Dalen. *Intuitionistic Logic*, pages 225–339. Springer Netherlands, Dordrecht, 1986. doi:10.1007/978-94-009-5203-4_4.
- 76 Herman Van Looy. A chronology and historical analysis of the mathematical manuscripts of gregorius a sancto vincentio (1584–1667). *Historia Mathematica*, 11(1):57–75, 1984. doi:10.1016/0315-0860(84)90005-3.
- 77 Nik Weaver. What is predicativism? Unpublished manuscript online., 2013. URL: <https://api.semanticscholar.org/CorpusID:40352024>.
- 78 Karl Weierstraß. *Rechnen mit komplexen Zahlen*, pages 25–44. Vieweg+Teubner Verlag, Wiesbaden, 1861-1862. doi:10.1007/978-3-663-06846-4_3.
- 79 Aleksandr S Yessenin-Volpin. The ultra-intuitionistic criticism and the antitraditional program for foundations of mathematics. In *Studies in Logic and the Foundations of Mathematics*, volume 60, pages 3–45. Elsevier, 1970.

A Reflection Principle for Potential Infinite Models of Type Theory

Matthias Eberl  

LMU Munich, Germany

Abstract

Denotational models of type theory, such as set-theoretic, domain-theoretic, or category-theoretic models use (actual) infinite sets of objects in one way or another. The potential infinite, seen as an extensible finite, requires a dynamic understanding of the infinite sets of objects. It follows that the type *nat* cannot be interpreted as a set of all natural numbers, $\llbracket nat \rrbracket = \mathbb{N}$, but as an increasing family of finite sets $\mathbb{N}_i = \{0, \dots, i - 1\}$. Any reference to $\llbracket nat \rrbracket$, either by the formal syntax or by meta-level concepts, must be a reference to a (sufficiently large) set \mathbb{N}_i .

We present the basic concepts for interpreting a fragment of the simply typed λ -calculus within such a dynamic model. A type ϱ is thereby interpreted as a process, which is formally a factor system together with a limit of it. A factor system is very similar to a direct or an inverse system, and its limit is also defined by a universal property. It is crucial to recognize that a limit is not necessarily an unreachable end beyond the process. Rather, it can be regarded as an intermediate state within the factor system, which can still be extended.

The logical type *bool* plays an important role, which we interpret classically as the set $\{true, false\}$. We provide an interpretation of simply typed λ -terms in these factor systems and limits. The main result is a reflection principle, which states that an element in the limit has a “full representative” at a sufficiently large stage within the factor system. For propositions, that is, terms of type *bool*, this implies that statements about the limit are true if and only if they are true at that sufficiently large stage.

2012 ACM Subject Classification Theory of computation \rightarrow Lambda calculus; Theory of computation \rightarrow Higher order logic; Theory of computation \rightarrow Type theory; Theory of computation \rightarrow Denotational semantics

Keywords and phrases Indefinite extensibility, Potential infinite, Reflection principle

Digital Object Identifier 10.4230/LIPIcs.TYPES.2023.6

1 Introduction

In set theory, infinite sets are given by the dictum of the axiom of infinity. There is no idea of “construction” or “approximation” involved – it is a *static* concept where only existence is required, without any way to get to these sets. In contrast, consider the constructions of infinite sets as limits of direct and inverse systems. These sets are approximated and so can be understood from the perspective of a potential infinite. Moreover, they possess the structure of their approximating parts. Finally, and most importantly from a finitistic perspective, there is no necessity to “jump over” to an absolute, actual infinite limit set if all states of the system are finite. Instead, if one takes care of all the stages of one’s investigation, then a sufficiently large state within the system is sufficient and is a full substitute for an infinite limit set.

From a consequent potentialist’s point of view, it is actually a misuse of language to call a set infinite. Since the potentialist’s view is a form of finitism, and since sets are given by their extension, every set is finite in this regard. Therefore, what can be considered infinite is the type, say *nat*, whereas the set \mathbb{N} would be more accurately described as *indefinitely extensible*. Thus, the usual terminology, saying that a set is infinite, means that its type is



© Matthias Eberl;

licensed under Creative Commons License CC-BY 4.0

29th International Conference on Types for Proofs and Programs (TYPES 2023).

Editors: Delia Kesner, Eduardo Herme Reyes, and Benno van den Berg; Article No. 6; pp. 6:1–6:20

Leibniz International Proceedings in Informatics



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

infinite and that the extension of the type, i.e., the set of elements of that type, is given by some *indefinitely large* state (or *sufficiently large* state) in an indefinitely extensible system of finite sets.

A system, introduced in Section 2.1, formalizes a changing totality of objects, an “open” and potentially unending process in which both the collection and its elements expand simultaneously. This process allows a (relative) completion or compactification by constructing a limit, which temporarily ends or “closes” the process.

The concept of partiality becomes relevant in extensible systems. It is a consequence of the fact that objects can be generated, or if you prefer a less constructive language, detected, so that they do not exist from the beginning. Partiality first appears in direct systems, where objects do not exist at the initial stages. In this situation, partiality is not difficult to deal with, but it becomes demanding in the function space construction, especially in the presence of higher types.

A few words about the notation. The term “iff” is an abbreviation for “if and only if”. \mathbb{N} refers to the set of natural numbers $\{0, 1, 2, \dots\}$, $\mathbb{N}^+ := \mathbb{N} \setminus \{0\}$ and $\mathbb{N}_i := \{0, \dots, i - 1\}$. We write $[A \rightarrow B]$ for the function space of all functions with domain A and codomain B , and $\mathcal{P}(\mathcal{M})$ for the power set of \mathcal{M} .

1.1 Extensibility, Coinduction and Domain Theory

A fundamental concept in the context of the potential infinite is *extensibility*. The main modes of extensibility are the creation of new objects and the creation of new knowledge about existing objects. However, we understand the latter as a differentiation of an object or an identification of several objects. In the case of differentiation, there may be multiple versions of an object at a later stage. When considering limit constructions, this is different from the understanding of the accumulation of information. To illustrate this point, let us consider inductive and coinductive definitions. These are related to adding and differentiating objects respectively, but they are not the only way.

1. The dynamic reading of an inductive definition leads to an infinite process of creating objects, related to direct systems.
2. The dynamic reading of a coinductive definition leads to an infinite process of differentiating objects, related to inverse systems.

To give an example, let Seq denote finite 0-1-sequences and Seq^∞ stand for infinite 0-1-sequences. Then the algebra with constructors $nil : \{*\} \rightarrow Seq$ and $append : Seq \times \mathbb{N}_2 \rightarrow Seq$ inductively defines the structure of Seq , whereby $\mathbb{N}_2 = \{0, 1\}$. The coinductive definition of Seq^∞ has one destructor-pair $(head, tail) : Seq^\infty \rightarrow \mathbb{N}_2 \times Seq^\infty$ and defines a coalgebra. To give these structural definitions a dynamic reading requires an index set, which will be \mathbb{N}^+ and \mathbb{N} .

In case of inductive definitions one starts with the element $*$. The states of Seq are thus $Seq_0 = \{*\}$, $Seq_1 = \{()\}$, $Seq_2 = \{(), 0, 1\}$, $Seq_3 = \{(), 0, 1, 00, 01, 10, 11\}$ and so on. The inductive definition of Seq gives rise to a direct system $(Seq_i)_{i \in \mathbb{N}^+}$ with subset inclusion as embedding. A direct system is more general than this construction by an inductive definition, as it allows for the possibility of non-injective embeddings. This corresponds to the addition and identification of objects in a single process and is relevant for a construction of quotients.

For a dynamic understanding of coinductive definitions, start with a “generic” element, say s , so $Seq_0^\infty = \{s\}$. The destructor-pair $(head, tail)$ gives $Seq_1^\infty = \{0s, 1s\}$, $Seq_2^\infty = \{00s, 01s, 10s, 11s\}$, and so on. The coinductive definition of Seq^∞ allows one to increase the knowledge about the sequence, which we understand as a process of differentiation. The

projections $proj_i^{i'} : Seq_i^\infty \rightarrow Seq_{i'}^\infty$ then remove this information gained by differentiation, e.g., $proj_1^2(01s) = 0s$. We read $01s \in Seq_2^\infty$ and $0s \in Seq_1^\infty$ as states of the same (infinite) object, differing only in the amount of information that we have about that object. The inverse system $(Seq^\infty, proj)$ corresponds to subsets $A \subseteq \mathbb{N}$ if we take $A = \{i \in \mathbb{N} \mid s_i = 1\}$ for $(s_i)_{i \in \mathbb{N}} \in Seq^\infty$.

This idea of differentiating objects is also related to Brouwer's concept of (lawlike and lawless) choice sequences within a spread – the standard text on this subject is [13]. However, our approach does not involve the constructive reasoning that is inherent in the definition of a spread.

Differentiating infinite objects differs from domain theory [1], where the ideal, infinite elements are ideals (being infinite sets) of their approximating parts, which are the compact elements. In domain theory, each approximation is seen as a different object, not as states of one object, and the ideal completion uses the idea that sets are *actual* infinite. In other words, the idea of differentiating infinite objects, as formulated in coinductive definitions and inverse systems, is not present in domain theory. Domain theory uses inductive definitions and least fixed point constructions.

Another way to get these ideal elements in domains is to think of an infinite domain as a bilimit of finite domains. A bilimit is a limit construction where direct and inverse limits coincide. One could interpret this coincidence as a reduction of inverse systems to direct systems. In the approach here, the limit construction is, roughly speaking, between the direct and the inverse limit, since the predecessor relation $\overset{p}{\mapsto}$ (which will be introduced soon) is, again roughly speaking, between the embedding *emb* and the projection *proj*. In Section 2.1 we will introduce such an embedding-projection pair, associated with the predecessor relation $\overset{p}{\mapsto}$. Direct and inverse systems can then be seen as extreme situations of adding and differentiating objects, while the system based on the predecessor relation $\overset{p}{\mapsto}$ is in general a combination of both.

1.2 Formalizing the Potential Infinite

A potential infinite set \mathcal{M} is a dynamic, finite set. To formalize this consequently, any reference to \mathcal{M} can only be made by reference to some finite state \mathcal{M}_i . A completed totality of all elements or all states of this set does not exist. For example, there exist states \mathbb{N}_i of the set of natural numbers, but there is neither a complete set \mathbb{N} nor the complete family of all sets \mathbb{N}_i . The latter has to do with the fact that, for a consequent reading, this finitistic view applies to meta-level concepts as well, in particular to the index set from which the indices i are taken; see Section 1.4 for more on this.

In a first step we introduce indices i , given by a directed index set \mathcal{I} , i.e., a set \mathcal{I} of stages together with a binary, reflexive and transitive relation \leq so that any finite set of indices has an upper bound. A potential infinite set is thus a family $\mathcal{M}_{\mathcal{I}}$ of states \mathcal{M}_i , $i \in \mathcal{I}$, where all sets are finite.

We want to express that two elements $a_i \in \mathcal{M}_i$ and $a_{i'} \in \mathcal{M}_{i'}$ at different stages i and i' are equal. This equality is not given by an equivalence relation, but by a family of reflexive relations $\overset{p}{\mapsto}_{i',i} \subseteq \mathcal{M}_{i'} \times \mathcal{M}_i$ for $i' \geq i$. We write $a_{i'} \overset{p}{\mapsto} a_i$ for $(a_{i'}, a_i) \in \overset{p}{\mapsto}_{i',i}$, saying that a_i is a predecessor of $a_{i'}$, and we use $\overset{p}{\mapsto}$ as an abbreviation for $\overset{p}{\mapsto}_{i',i}$. Reflexivity means that $a_i \overset{p}{\mapsto} a_i$ for all $i \in \mathcal{I}$ and $a_i \in \mathcal{M}_i$. We do not require $\overset{p}{\mapsto}$ to be transitive.¹ A *system* $(\mathcal{M}_{\mathcal{I}}, \overset{p}{\mapsto})$

¹ The reason is this: The index set of the function space is $\mathcal{I} \times \mathcal{J}$ with pairs written as $i \rightarrow j$ (c.f. Section 2.2). Call an extension of the index from $i \rightarrow j$ to $i \rightarrow j'$, $j' \geq j$, *covariant* and from $i \rightarrow j$ to $i' \rightarrow j$, $i' \geq i$, *contravariant*. Then a combination of two different kind of extensions may fail to be a correct extension. For an example, see [5].

consists of a family $\mathcal{M}_{\mathcal{I}}$ together with reflexive relations $\vdash^{\mathcal{I}}$. Later we will introduce further properties in order to allow a function space construction, leading to the notion of a *factor system* [5].

Let types $\varrho \in Typ$ be given. In this paper they consist of some base types ι , including type *bool* for propositions, which are interpreted classically as the Boolean values *true* and *false*, and a type constructor \rightarrow . So types are $\varrho ::= \iota \mid \varrho \rightarrow \varrho$. A (typing) context Γ is a list of types $(\varrho_0, \dots, \varrho_{n-1})$. More explicitly we can write $(x_0 : \varrho_0, \dots, x_{n-1} : \varrho_{n-1})$ for the context Γ , since we use a fixed list of variables x_0, x_1, x_2, \dots in Section 3. The empty context is $()$ and $\Gamma.\varrho$ denotes the context Γ , extended by ϱ . We assume that for each type ϱ there is an index set $(\mathcal{I}_{\varrho}, \leq)$, so each type comes with its own set of stages. For instance, the index set for type *nat* is (\mathbb{N}^+, \leq) with $\mathcal{M}_i = \mathbb{N}_i$ for $i \in \mathbb{N}^+$.² Another example is type *bool* with the singleton set $(\{\text{bool}\}, =)$ of one index *bool* and $\mathcal{M}_{\text{bool}} := \mathbb{B} := \{\text{true}, \text{false}\}$. For a typing context $\Gamma = (\varrho_0, \dots, \varrho_{n-1})$ define $\mathcal{I}_{\Gamma} := \mathcal{I}_{\varrho_0} \times \dots \times \mathcal{I}_{\varrho_{n-1}}$ and endow \mathcal{I}_{Γ} with the product order.

Since we want to consider sets, relations and functions as objects, i.e., as elements of further potential infinite sets, we have to consider elements as dynamic entities as well. A dynamic object a of type ϱ is given by its states a_i , $i \in \mathcal{I}_{\varrho}$. As with sets, any reference to a can only be a reference to one of its states a_i .

A dynamic object needs not be defined on all stages $i \in \mathcal{I}_{\varrho}$, so the index set \mathcal{I}_a , for which a state a_i of a exists, is a subset of \mathcal{I}_{ϱ} , i.e., $\mathcal{I}_a \subseteq \mathcal{I}_{\varrho}$. One of the basic requirements is that every object has “sufficiently many” indices \mathcal{I}_a . For natural numbers, or more generally for base type objects, the situation is simple: If a number n occurs at stage $n + 1$, i.e., $n \in \mathbb{N}_{n+1}$, it will be there for all future stages $m > n + 1$, i.e., $n \in \mathbb{N}_m$, so the index set $\mathcal{I}_n = \{n + 1, n + 2, \dots\}$ is an up-set. By *up-set* we mean a non-empty, upward closed subset, in this case a subset of \mathbb{N}^+ . For higher-order functions, however, the situation is less straightforward and is one of the main challenges of this approach.

The concept of a potential infinite has two aspects, a cardinal aspect \mathfrak{D} and an ordinal aspect \ll . Let $C = (i_0, \dots, i_{n-1}) \in \mathcal{I}_{\Gamma}$ be a list of indices $i_0 \in \mathcal{I}_{\varrho_0}, \dots, i_{n-1} \in \mathcal{I}_{\varrho_{n-1}}$, for $\Gamma = (\varrho_0, \dots, \varrho_{n-1})$, called *state context*. Write $C.i$ for the extension of C by i .

1. The cardinal aspect is given by a set $\mathfrak{D}_{\Gamma} \subseteq \mathcal{P}(\mathcal{I}_{\Gamma})$ for each context Γ . $\mathcal{H} \in \mathfrak{D}_{\Gamma}$ says that there are *indefinitely many*, or *sufficiently many* contexts in \mathcal{H} .
2. The ordinal aspect is given by relations $\ll^{\Gamma.e} \in \mathfrak{D}_{\Gamma.e}$. Let $C \ll i$ stand for $C.i \in \ll^{\Gamma.e}$, meaning that a stage i is *indefinitely large*, or *sufficiently large* relative to the state context C .

The sets \mathfrak{D}_{Γ} satisfy the following properties: Each set $\mathcal{H} \in \mathfrak{D}_{\Gamma}$ is cofinal – recall that cofinality of \mathcal{H} means $\forall C \in \mathcal{I}_{\Gamma} \exists C' \geq C$ with $C' \in \mathcal{H}$. This minimal requirement simply states that we will always find a context in \mathcal{H} beyond any bound. \mathfrak{D}_{Γ} is closed under supersets, so “more than indefinitely many is indefinitely many”. Furthermore, an up-set on \mathcal{I}_{Γ} always has indefinitely many indices. Finally, the main restriction is that \mathfrak{D}_{Γ} is closed under intersections. This is necessary in order to guarantee that a relation between two objects can be established on indefinitely many indices. This amounts to saying:

\mathfrak{D}_{Γ} is a proper filter on $\{\mathcal{H} \subseteq \mathfrak{D}_{\Gamma} \mid \mathcal{H} \text{ is cofinal}\}$ and \mathfrak{D}_{Γ} contains all up-sets. (Filter)

² The use of \mathbb{N}^+ instead of \mathbb{N} as index set is done for technical reasons: A factor system has projections between different states \mathbb{N}_i , and there is no projection from \mathbb{N}_i , $i > 0$ into $\mathbb{N}_0 = \emptyset$.

We will use the locution \mathfrak{D} -many indices, which refers to a set in \mathfrak{D}_Γ , and *cofinal many indices*, which refers to a cofinal index set. The interpretation in a potential infinite structure, which we introduce in Section 3, will be relative to \mathfrak{D} (and later also relative to \ll , when we introduce the universal quantifier in a subsequent paper).

A basic theme of the potential infinite is *dependency*. In particular, there are no fixed sets \mathfrak{D} and \ll , but these are parameters that depend on factors of the concrete mathematical investigation and the state of it. Another way of expressing this is to say that \mathfrak{D} and \ll are intensional notions whose extension depends on the context of investigation. In the same way that \mathfrak{D} and \ll depend on the investigation, some concepts, in turn, depend on \mathfrak{D} and \ll ; the notion of continuity depends on \mathfrak{D} and the interpretation of the universal quantifier depends on \ll . It is possible to define continuity on limit sets without reference to the underlying system. The basic relation is then a family of PERs and we will explore this structure in more detail in a separate paper. Moreover, in this paper we deal only with the cardinal aspect, i.e., \ll is not considered here.

1.3 Relation to Constructive Approaches

We are only investigating the idea of a potential infinite, not that of constructivity, decidability, complexity or knowledge about existence, which are important concepts in intuitionism [14] and in theories about computability [8]. The common models of intuitionistic logic, such as Kripke models, or more generally topos-theoretic models [9], use unbounded universal quantification. For instance, in a Kripke model, the validity of a universal quantified formula uses a reference to all “future” nodes – there could be infinitely many of them and at each such node the carrier set could be an infinite set as well. In our approach, only finite sets are used, and in a consequent finitistic view only finitely many of them.

We use classical logic. This is because it has a simpler model than constructive models and is more widely used. Furthermore, the results presented here rely on the fact that classical logic has a finite number of truth values. This is an important difference from intuitionistic logic, which has, if truth values are used at all, infinitely many of them.

The novel aspect of this potential infinite model is the introduction of *state judgements*, which are refinements of typing judgements. If $\Gamma \vdash r : \varrho$ is such a typing judgement, then a state judgement has the form $C \mid r : i$, where C is a stage of the context Γ and i is a stage of the type ϱ . The idea that we can only refer to infinite objects via a specific state is reflected in the fact that the primary object of interpretations is a state judgement $C \mid r : i$. It is interpreted in a family of factor systems, which have only “local” application functions given as

$$App_{i,j} : [\mathcal{M}_i \rightarrow \mathcal{N}_j] \times \mathcal{M}_i \rightarrow \mathcal{N}_j, (f_{i \rightarrow j}, a_i) \mapsto f_{i \rightarrow j}(a_i).$$

Additionally, the typing judgement $\Gamma \vdash r : \varrho$ has an interpretation in the limit set. However, the notion of a limit of a factor system depends on the notion of “indefinitely many stages”, which we formalize as sets \mathfrak{D}_ϱ . This is in particular relevant for the function space and the possibility to define a “global” application function $App : [\mathcal{M} \rightarrow \mathcal{N}] \times \mathcal{M} \rightarrow \mathcal{N}, (f, a) \mapsto f(a)$ on limit sets, leading to a common *extensional type structure* [2]. The question which higher-order functions exist and whether all local applications together yield a global application depends on the properties of \mathfrak{D}_ϱ . A global application function is available for all first-order functions, however, this may not be the case for higher-order functions. In contrast, in a type structure, also referred to as (*typed*) *applicative structure*, a global application is available for all types. This holds analogously for *Kripke applicative structures* [10].

1.4 The Meta-Theory

The meta-theory in which the concepts are developed is classical higher-order logic, as formalized in Church’s simple type theory [3]. This theory will also serve as the investigated theory. At the object-level, we develop a potential infinite model in order to interpret typed λ -terms. At the meta-level these two views of infinity are relevant:

1. One accepts actual infinity at the meta-level. In this case, a limit can be seen as the usual actual infinite set beyond the system. This view allows a comparison of an actual infinite model, given as limit structure, with the potential infinite part, i.e., the system.
2. One uses the view that infinity is an extensible finite. Then a limit is an intermediate state of the system. This is the consistent realization of the finitistic approach.

The reflection principle, which is the main theorem of this paper, states in the first case that all objects in the limit, including propositions, have a counterpart in the factor system. So it says something like this: Whatever exists and holds under the assumption that actual infinite sets exist, already exists and holds at a sufficiently large stage in the system. For infinite objects, these can be seen as approximations. For type *bool*, which is interpreted in classical logic as a finite set of truth values $\{true, false\}$, the values are the same in the actual infinite limit set and at a finite stage of the system. This is because these values are not approximated.

In the second case, the reflection principle is only a means to show the correctness of the interpretation. However, a consistent realization requires that an infinite set on meta-level, like the index set \mathcal{I} , is only available at a stage j . A consequent realization in type theory moreover uses a type in place of the index set, together with a term \leq , which is then shown to be reflexive, transitive, and directed.

The use of a classical meta-theory instead of a constructive one is not essential here. We could also take an intuitionistic type theory at meta-level and develop most of the model theory in a pure constructive way. So we expect that the model construction can be formalized in common proof assistants such as Coq, Lean or Agda. We need, however, a bit of classical reasoning, at least when introducing the universal quantifier. To prove the reflection principle with universal quantifier as an extension of Theorem 24, one has to do a kind of Löwenheim-Skolem construction. This requires that a universal quantified formula is either true (at all stages), or it is false at some stage and is false at all later stages, too. It is of greater significance that, for a consequent realization, in which only potential infinities are used at meta-level, the proof assistant must implement state judgments.

1.5 Structure of the Paper

We already started in Section 1.2 to formalize the potential infinite as a dynamic concept, which replaces an actual infinite set with a *factor system*, which is, roughly, a generalization of a direct and inverse system. The concept of factor systems was first introduced in [5]. In Section 2 we reiterate the definition of a factor system and add further definitions that are necessary for an interpretation. These are primarily the notions of a direct and inverse factor system, which are required to interpret variables. We show the construction of the function space between two factor systems and elucidate the notions of a target and a limit of a factor system. As with direct and inverse systems, limits are targets that satisfy a universal property and have a concrete construction. In Section 2.5 we demonstrate how to introduce an application that makes limit sets a type structure.

Section 3 introduces a judgement for states, parallel to judgements for types. These are defined on a fragment of the simply typed λ -calculus, which we call *core fragment*. Based on these state judgements we give a first version of an interpretation of λ -terms in the core

fragment, not including any constants. In particular, the present paper does not yet include logic. We give an interpretation of types and terms with two parts, one is within the system, the other is in its limit. Based on this interpretation we show a first version of a reflection principle, which says that an element a of type ϱ in the limit set is reflected by an element a_i at some stage $i \in \mathcal{I}_\varrho$ in the system. This element a_i is an approximation of a , and at the same time it fully represents a . If logic is included, then the representation includes all propositions about these elements, so anything we can say about a is true if and only if it is true for a_i .

2 Factor Systems and their Limits

Factor systems and their limits have been introduced in [5]. In this paper, we summarize their properties. The main concept is that of a factor system, those of a prefactor system (with embeddings/projections) are afferent notions. In addition to that, we introduce the notion of a *stable* system, which is a natural notion to prove stronger properties, although these are not necessary here. We prove that stability is closed under both the function space construction and the limit construction.

Relevant for the interpretation in Section 3 are the specific forms of a *direct* and *inverse* factor system. A direct factor system is similar to a direct system, it is more specific in the sense that the embedding is part of an embedding-projection pair. On the other hand it is more general in the sense that equations hold only up to an equivalence relation. The same holds for inverse factor systems and inverse systems. Moreover, we introduce the concept of a *homomorphism* between two systems.

The subsequent Lemmata 9 and 10 are extended versions of corresponding lemmata in [5], which use the property (Filter), not only cofinality. These versions are necessary to prove Corollary 15, which states that the function space construction commutes with the limit construction. This is a prerequisite for the definition of a model. Proposition 7 and Corollary 12 describe how direct and inverse factor systems extend to the function space and to the limit. Both are essential to prove the reflection principle for variables.

2.1 Factor Systems

\mathcal{I} will always denote a non-empty directed index set with preorder \leq . A *system* is a pair $(\mathcal{M}_\mathcal{I}, \overset{p}{\rightrightarrows})$ consisting of a family $\mathcal{M}_\mathcal{I} := (\mathcal{M}_i)_{i \in \mathcal{I}}$ and reflexive (for $i = i'$) relations $\overset{p}{\rightrightarrows}$ on $\mathcal{M}_{i'} \times \mathcal{M}_i$ for $i' \geq i$. Two elements $a_i \in \mathcal{M}_i$ and $b_j \in \mathcal{M}_j$ are *consistent*, written as $a_i \asymp b_j$, iff there is an index $i' \geq i, j$ and some $a_{i'} \in \mathcal{M}_{i'}$ such that $a_{i'} \overset{p}{\rightrightarrows} a_i$ and $a_{i'} \overset{p}{\rightrightarrows} b_j$. As a convention, whenever we use a suffix $i \in \mathcal{I}$ for some element, this refers to the state, e.g. $a_i \in \mathcal{M}_i$. An important special case is that the relations $\overset{p}{\rightrightarrows}$ are partial functions, which is equivalent to:

$$a_i \asymp b_i \iff a_i \overset{p}{\rightrightarrows} b_i \iff a_i = b_i \quad (\text{Fun})$$

for all $a_i, b_i \in \mathcal{M}_i$ and all $i \in \mathcal{I}$. A system that satisfies (Fun) is called *standard*. $(\mathcal{M}_\mathcal{I}, \overset{p}{\rightrightarrows})$ is a *prefactor system* iff it is a system satisfying

$$a_{i'} \asymp a_i \iff a_{i'} \overset{p}{\rightrightarrows} a_i \quad (\text{Factor})$$

for all $a_i \in \mathcal{M}_i$ and $a_{i'} \in \mathcal{M}_{i'}$ with $i \leq i'$. The relation \asymp is then an equivalence relation on a single set \mathcal{M}_i with $a_i \asymp b_i \iff a_i \overset{p}{\rightrightarrows} b_i \iff b_i \overset{p}{\rightrightarrows} a_i$ for $a_i, b_i \in \mathcal{M}_i$. In a prefactor system $b_{i'} \asymp a_{i'} \overset{p}{\rightrightarrows} a_i$ implies $b_{i'} \overset{p}{\rightrightarrows} a_i$, but sometimes we want to have the “dual” property

as well, which we call *stability*. Although we can do without stability for most properties, it is an obvious requirement, and all natural examples satisfy this property. Note that a system that satisfies (Fun) is automatically stable.

► **Definition 1.** A system $(\mathcal{M}_{\mathcal{I}}, \overset{P}{\mapsto})$, and in particular the relation $\overset{P}{\mapsto}$, is called *stable* iff for all $i' \geq i$, all $a_{i'} \in \mathcal{M}_{i'}$ and $a_i, b_i \in \mathcal{M}_i$

$$a_{i'} \overset{P}{\mapsto} a_i \asymp b_i \Rightarrow a_{i'} \overset{P}{\mapsto} b_i. \quad (\text{Stab})$$

A family $emb = (emb_i^{i'})_{i \leq i'}$ of \asymp -embeddings consists of \asymp -preserving maps $emb_i^{i'} : \mathcal{M}_i \rightarrow \mathcal{M}_{i'}$ satisfying $emb_i^{i'}(a_i) \asymp a_i$ and $emb_{i'}^{i''}(emb_i^{i'}(a_i)) \asymp emb_i^{i''}(a_i)$. The requirement \asymp -preserving means that $emb_i^{i'}(a_i) \asymp emb_i^{i'}(b_i)$ holds for $a_i \asymp b_i$, for all $a_i, b_i \in \mathcal{M}_i$. Similar to a family of \asymp -embeddings, \asymp -projections $proj = (proj_i^{i'})_{i \leq i'}$ consist of \asymp -preserving maps $proj_i^{i'} : \mathcal{M}_{i'} \rightarrow \mathcal{M}_i$ satisfying $proj_i^{i'}(a_i) \asymp a_i$ and $proj_i^{i'}(proj_{i'}^{i''}(a_{i''})) \asymp proj_i^{i''}(a_{i''})$. Moreover, \asymp -embeddings emb together with \asymp -projections $proj$ form an \asymp -embedding-projection pair iff $proj_i^{i'}(emb_i^{i'}(a_i)) \asymp a_i$ holds for all $a_i \in \mathcal{M}_i$ and all $i \leq i'$.

The \asymp -embeddings emb and \asymp -projections $proj$ are *coherent* if they satisfy for all indices $i \leq i' \leq i''$

$$a_{i'} \overset{P}{\mapsto} a_i \Rightarrow emb_{i'}^{i''}(a_{i'}) \overset{P}{\mapsto} a_i \text{ and} \quad (\text{Emb})$$

$$a_{i''} \overset{P}{\mapsto} a_i \Rightarrow proj_{i'}^{i''}(a_{i''}) \overset{P}{\mapsto} a_i \text{ resp.} \quad (\text{Proj})$$

A \asymp -embedding-projection pair $(emb, proj)$ is coherent if emb and $proj$ are both coherent. Property (Emb) implies that $emb_i^{i'}(a_i) \overset{P}{\mapsto} a_i$ holds for all $a_i \in \mathcal{M}_i$, and in case that Property (Fun) holds, $\overset{P}{\mapsto}$ is a *partial surjection*.

► **Definition 2.** A prefactor system with embeddings is a *prefactor system* $(\mathcal{M}_{\mathcal{I}}, \overset{P}{\mapsto})$ with coherent \asymp -embeddings emb . A prefactor system with projections is a *prefactor system* $(\mathcal{M}_{\mathcal{I}}, \overset{P}{\mapsto})$ with coherent \asymp -projections $proj$. A factor system is a *prefactor system* $(\mathcal{M}_{\mathcal{I}}, \overset{P}{\mapsto})$ with a coherent \asymp -embedding-projection pair $(emb, proj)$. A *prefactor system* is *direct* iff it has coherent \asymp -embeddings which satisfy

$$a_{i'} \overset{P}{\mapsto} a_i \iff a_{i'} \asymp emb_i^{i'}(a_i). \quad (\text{Dir})$$

A *prefactor system* is *inverse* iff it has coherent \asymp -projections which satisfy

$$a_{i'} \overset{P}{\mapsto} a_i \iff proj_i^{i'}(a_{i'}) \asymp a_i. \quad (\text{Inv})$$

If $\overset{P}{\mapsto}$, emb and $proj$ are known, we often call $\mathcal{M}_{\mathcal{I}}$ a *factor system* (and likewise with prefactor systems with embeddings/projections).

► **Example 3.** The embeddings emb in a direct system $(\mathcal{M}_{\mathcal{I}}, emb)$, with $a_{i'} \overset{P}{\mapsto} a_i : \iff emb_i^{i''}(a_i) = emb_{i'}^{i''}(a_{i'})$ for some $i'' \geq i, i'$, are automatically \asymp -embeddings, since they preserve \asymp . If there are \asymp -projections $proj$, such that emb and $proj$ form a \asymp -embedding-projection pair, then $(\mathcal{M}_{\mathcal{I}}, \overset{P}{\mapsto}, emb, proj)$ is a direct factor system.

An inverse system $(\mathcal{M}_{\mathcal{I}}, proj)$ satisfies (Fun), so the projections $proj$ are automatically \asymp -projections. If there are \asymp -embeddings emb , such that emb and $proj$ form a \asymp -embedding-projection pair, then $(\mathcal{M}_{\mathcal{I}}, \overset{P}{\mapsto}, emb, proj)$, with $a_{i'} \overset{P}{\mapsto} a_i : \iff proj_i^{i'}(a_{i'}) = a_i$, is an inverse factor system.

► **Example 4.** Consider $(\mathbb{N}_i)_{i \in \mathbb{N}^+}$ with the embedding-projection pair $emb_i^{i'} : \mathbb{N}_i \rightarrow \mathbb{N}_{i'}$, $n \mapsto n$ and $proj_i^{i'} : \mathbb{N}_{i'} \rightarrow \mathbb{N}_i$ with $n \mapsto \min(n, i - 1)$. In all three cases for $\overset{P}{\mapsto}$ they form an \asymp -embedding-projection pair and $((\mathbb{N}_i)_{i \in \mathbb{N}^+}, \overset{P}{\mapsto}, emb, proj)$ is a factor system:

1. The standard model of the natural numbers has $\mathbb{N}_{i'} \ni n \xrightarrow{p} n \in \mathbb{N}_i$ for all $n < i$, which is a direct factor system.
2. The definition $n' \xrightarrow{p} n : \iff \text{proj}_i^{i'}(n') = n$ makes it an inverse factor system.
3. With $n' \xrightarrow{p} n$ for all $n' \in \mathbb{N}_{i'}$ and $n \in \mathbb{N}_i$, the factor system $(\mathbb{N}_i)_{i \in \mathbb{N}^+}$ is direct and inverse.

The second example basically adds an infinite number to \mathbb{N} , while the last example is artificial, but shows the difference to direct and inverse limits when we pick up this factor system again in Example 8. One can easily check that $(\mathbb{N}_i)_{i \in \mathbb{N}^+}$ is stable in all three cases, and in the first two cases Property (Fun) is also satisfied.

► **Lemma 5.** *Let $\mathcal{M}_{\mathcal{I}}$ be a prefactor system and given indices $i'' \geq i' \geq i$.*

1. *If $\mathcal{M}_{\mathcal{I}}$ is direct, then $a_{i''} \xrightarrow{p} a_i$ implies $a_{i''} \xrightarrow{p} \text{emb}_i^{i'}(a_i)$.*
2. *If $\mathcal{M}_{\mathcal{I}}$ is inverse, then $a_{i''} \xrightarrow{p} a_{i'}$ implies $a_{i''} \xrightarrow{p} \text{proj}_i^{i'}(a_{i'})$.*

Proof. First, $a_{i''} \xrightarrow{p} a_i$ implies $\text{emb}_i^{i''}(\text{emb}_i^{i'}(a_i)) \simeq \text{emb}_i^{i''}(a_i) \simeq a_{i''}$ by (Dir), hence $\text{emb}_i^{i''}(\text{emb}_i^{i'}(a_i)) \xrightarrow{p} a_{i''}$. From (Emb) we deduce $\text{emb}_i^{i''}(\text{emb}_i^{i'}(a_i)) \xrightarrow{p} \text{emb}_i^{i'}(a_i)$ and thus $a_{i''} \xrightarrow{p} \text{emb}_i^{i'}(a_i)$ by (Factor). For the second clause let $a_{i''} \xrightarrow{p} a_{i'}$, then $\text{proj}_i^{i''}(a_{i''}) \simeq a_{i'}$ by (Inv), hence $\text{proj}_i^{i''}(a_{i''}) \simeq \text{proj}_i^{i'}(\text{proj}_i^{i''}(a_{i''})) \simeq \text{proj}_i^{i'}(a_{i'})$. Consequently $a_{i''} \xrightarrow{p} \text{proj}_i^{i'}(a_{i'})$ by (Inv) again, as claimed. ◀

Compare these properties with (Emb) and (Proj), which hold in any prefactor system with embeddings/projections.

► **Definition 6.** *A homomorphism $\Phi = (\Phi_0, (\Phi^i)_{i \in \mathcal{I}})$ between two systems $(\mathcal{M}_{\mathcal{I}}, \xrightarrow{p})$ and $(\mathcal{N}_{\mathcal{J}}, \xrightarrow{p})$ consists of maps $\Phi_0 : \mathcal{I} \rightarrow \mathcal{J}$ and $\Phi^i : \mathcal{M}_i \rightarrow \mathcal{N}_{\Phi_0(i)}$ such that Φ_0 is monotone and for all $i \leq i'$*

$$a_{i'} \xrightarrow{p} a_i \iff \Phi^{i'}(a_{i'}) \xrightarrow{p} \Phi^i(a_i). \quad (1)$$

If (1) is an equivalence, then Φ is said to be strong. A homomorphism between two prefactor systems with embeddings $(\mathcal{M}_{\mathcal{I}}, \xrightarrow{p}, \text{emb})$ additionally satisfies $\Phi^{i'}(\text{emb}_i^{i'}(a_i)) = \text{emb}_j^{j'}(\Phi^i(a_i))$, $j := \Phi_0(i)$, $j' := \Phi_0(i')$, a homomorphism between two prefactor systems with projections $(\mathcal{M}_{\mathcal{I}}, \xrightarrow{p}, \text{proj})$ satisfies $\Phi^i(\text{proj}_i^{i'}(a_{i'})) = \text{proj}_j^{j'}(\Phi^{i'}(a_{i'}))$, and a homomorphism between two factor systems satisfies both equations.

We call Φ injective (surjective) iff every map in Φ is injective (surjective resp.). An isomorphism Φ between two systems (prefactor systems with embeddings/projections, factor systems) is a homomorphism with a further homomorphism Ψ as its inverse, i.e., each part of Ψ is inverse to that of Φ . So an isomorphism is automatically strong. We write $\mathcal{M}_{\mathcal{I}} \simeq \mathcal{N}_{\mathcal{J}}$ if an isomorphism between $\mathcal{M}_{\mathcal{I}}$ and $\mathcal{N}_{\mathcal{J}}$ exists.

In the following we will use homomorphisms Φ only for the special situation that $\mathcal{I} = \mathcal{J}$ and Φ_0 is the identity map. In that case we do not mention Φ_0 .

2.2 The Function Space

The function space of two factor systems $\mathcal{M}_{\mathcal{I}}$ and $\mathcal{N}_{\mathcal{J}}$, denoted as $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$, is a family of (finite) sets $[\mathcal{M}_i \rightarrow \mathcal{N}_j]$ indexed by pairs $(i, j) \in \mathcal{I} \times \mathcal{J}$ with product order, whereby we write $i \rightarrow j$ for such an index in $\mathcal{I} \times \mathcal{J}$. The set $[\mathcal{M}_i \rightarrow \mathcal{N}_j]$ consists of all (total) functions $f : \mathcal{M}_i \rightarrow \mathcal{N}_j$ which preserve \simeq , i.e., which satisfy $f(a_i) \simeq f(b_i)$ for $a_i \simeq b_i$. If the relations \xrightarrow{p} are partial functions on $\mathcal{M}_{\mathcal{I}}$ and $\mathcal{N}_{\mathcal{J}}$, then \simeq is the identity on \mathcal{M}_i and \mathcal{N}_j and $[\mathcal{M}_i \rightarrow \mathcal{N}_j]$ simply consists of all functions from \mathcal{M}_i to \mathcal{N}_j .

6:10 Potential Infinite Models of Type Theory

Let $f \in [\mathcal{M}_i \rightarrow \mathcal{N}_j]$, $f' \in [\mathcal{M}_{i'} \rightarrow \mathcal{N}_{j'}]$ and $i \rightarrow j \leq i' \rightarrow j'$, i.e., $i \leq i'$ and $j \leq j'$. The basic relation \xrightarrow{P} on the function space is a logical relation [12]. It is thus defined as

$$f' \xrightarrow{P} f : \iff a_{i'} \xrightarrow{P} a_i \text{ implies } f'(a_{i'}) \xrightarrow{P} f(a_i)$$

for all $a_{i'} \in \mathcal{M}_{i'}$ and $a_i \in \mathcal{M}_i$. The embedding-projection pair for the function space is defined in the usual way:

$$\begin{aligned} \text{emb}_{i \rightarrow j}^{i' \rightarrow j'} : [\mathcal{M}_i \rightarrow \mathcal{N}_j] &\rightarrow [\mathcal{M}_{i'} \rightarrow \mathcal{N}_{j'}] & f &\mapsto \text{emb}_j^{j'} \circ f \circ \text{proj}_i^{i'}, \\ \text{proj}_{i \rightarrow j}^{i' \rightarrow j'} : [\mathcal{M}_{i'} \rightarrow \mathcal{N}_{j'}] &\rightarrow [\mathcal{M}_i \rightarrow \mathcal{N}_j] & f' &\mapsto \text{proj}_j^{j'} \circ f' \circ \text{emb}_i^{i'}. \end{aligned}$$

► **Proposition 7.** *If $\mathcal{M}_{\mathcal{I}}$ and $\mathcal{N}_{\mathcal{J}}$ are both factor systems, so is their function space. Moreover,*

1. *If \xrightarrow{P} satisfies (Fun) or (Stab) on $\mathcal{N}_{\mathcal{J}}$, so does \xrightarrow{P} on $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$.*
2. *If $\mathcal{M}_{\mathcal{I}}$ is inverse and $\mathcal{N}_{\mathcal{J}}$ direct, then $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$ is direct.*
3. *If $\mathcal{M}_{\mathcal{I}}$ is direct and $\mathcal{N}_{\mathcal{J}}$ inverse, then $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$ is inverse.*

Proof. This has been proven in [5], except for the statements about (Stab) and about direct and inverse factor systems. Assume $f' \xrightarrow{P} f \asymp g$ for $f' \in [\mathcal{M}_{i'} \rightarrow \mathcal{N}_{j'}]$ and $f, g \in [\mathcal{M}_i \rightarrow \mathcal{N}_j]$, $i \rightarrow j \leq i' \rightarrow j'$, so that we have to show $f' \xrightarrow{P} g$. Let $a_{i'} \xrightarrow{P} a_i$, and to confirm that $f'(a_{i'}) \xrightarrow{P} g(a_i)$, use the stability condition on $\mathcal{N}_{\mathcal{J}}$ applied to $f'(a_{i'}) \xrightarrow{P} f(a_i) \asymp g(a_i)$.

Next, let $\mathcal{M}_{\mathcal{I}}$ be inverse and $\mathcal{N}_{\mathcal{J}}$ direct. Assume first that $f' \xrightarrow{P} f$ with $f' \in [\mathcal{M}_{i'} \rightarrow \mathcal{N}_{j'}]$, $f \in [\mathcal{M}_i \rightarrow \mathcal{N}_j]$ and $i \rightarrow j \leq i' \rightarrow j'$. We wish to show that $f' \asymp \text{emb}_{i \rightarrow j}^{i' \rightarrow j'}(f)$, which is the same as $f' \asymp \text{emb}_j^{j'} \circ f \circ \text{proj}_i^{i'}$. Let $a_{i'} \xrightarrow{P} b_{i'}$ and define $b_i := \text{proj}_i^{i'}(b_{i'})$. Then $\text{proj}_i^{i'}(a_{i'}) \asymp b_i$ since projections preserve \asymp . It follows that $a_{i'} \xrightarrow{P} b_{i'}$ since $\mathcal{M}_{\mathcal{I}}$ is inverse, and thus $f'(a_{i'}) \xrightarrow{P} f(b_{i'})$. Consequently, $f'(a_{i'}) \asymp \text{emb}_j^{j'}(f(b_{i'})) = \text{emb}_j^{j'} \circ f \circ \text{proj}_i^{i'}(b_{i'})$, since $\mathcal{N}_{\mathcal{J}}$ is direct.

For the other direction assume $f' \asymp \text{emb}_{i \rightarrow j}^{i' \rightarrow j'} \circ f \circ \text{proj}_i^{i'}$ and let $a_{i'} \xrightarrow{P} a_i$. We shall prove that $f'(a_{i'}) \xrightarrow{P} f(a_i)$. Then $a_{i'} \xrightarrow{P} a_i$ implies $\text{proj}_i^{i'}(a_{i'}) \asymp a_i$ since $\mathcal{M}_{\mathcal{I}}$ is inverse. Now f and the embeddings preserve \asymp , so $\text{emb}_j^{j'}(f(\text{proj}_i^{i'}(a_{i'}))) \asymp \text{emb}_j^{j'}(f(a_i))$. Certainly $f'(a_{i'}) \asymp \text{emb}_j^{j'}(f(a_i))$. This shows $f'(a_{i'}) \xrightarrow{P} f(a_i)$, since $\mathcal{N}_{\mathcal{J}}$ is direct, as claimed. The proof of the last statement is verified in a similar way. ◀

2.3 Targets and Limits

The subsequent concepts require sets $\mathfrak{D}(\mathcal{I})$ from Section 1.2 with Property (Filter), whereby we write \mathfrak{D}_ρ for $\mathfrak{D}(\mathcal{I}_\rho)$ and \mathfrak{D}_Γ for $\mathfrak{D}(\mathcal{I}_\Gamma)$. A *target* $(\mathcal{M}, \xrightarrow{P})$ for a system $\mathcal{M}_{\mathcal{I}}$ extends the system “at the top”, i.e., the extension leads to a system $\mathcal{M}_{\bar{\mathcal{I}}}$, called *compactification* of $\mathcal{M}_{\mathcal{I}}$, with index set $\bar{\mathcal{I}} := \mathcal{I} \cup \{\text{top}\}$, *top* as greatest index, and $\mathcal{M}_{\text{top}} = \mathcal{M}$. Let $a \xrightarrow{P} a_i$ denote $a \xrightarrow{P} a_i$ provided that $a \in \mathcal{M}$, and we also write Emb_i for $\text{emb}_i^{\text{top}}$ and Proj_i for $\text{proj}_i^{\text{top}}$. Relation \xrightarrow{P} on a target \mathcal{M} , and consequently relation \asymp on \mathcal{M} as well, is by definition the identity. The *extension* of an element $a \in \mathcal{M}$ is $\text{Ext}(a) := \{a_i \in \bigcup_{i \in \mathcal{I}} \mathcal{M}_i \mid a \xrightarrow{P} a_i\}$. A target \mathcal{M} for a system $\mathcal{M}_{\mathcal{I}}$ satisfies by definition

$$\mathcal{I}_a := \{i \in \mathcal{I} \mid \exists a_i \in \mathcal{M}_i \ a \xrightarrow{P} a_i\} \in \mathfrak{D}(\mathcal{I})$$

for all objects $a \in \mathcal{M}$. Moreover, if the system is a prefactor system, a prefactor system with embeddings/projections or a factor system, then the compactification $\mathcal{M}_{\bar{\mathcal{I}}}$ must have this additional structure with its properties as well. Whereas the compactification of a system is automatically a system, the compactification of a *prefactor system* requires for all $a \in \mathcal{M}$, $a_i \in \mathcal{M}_i$, $a_{i'} \in \mathcal{M}_{i'}$ and $i \leq i'$

$$a \xrightarrow{P} a_{i'}, a \xrightarrow{P} a_i \Rightarrow a_{i'} \xrightarrow{P} a_i. \quad (2)$$

If \mathcal{M} is a target for a *prefactor system with embeddings* $\mathcal{M}_{\mathcal{I}}$, then this implies the existence of \asymp -embeddings $Emb_i : \mathcal{M}_i \rightarrow \mathcal{M}$, satisfying $Emb_{i'}(emb_i^{i'}(a_i)) = Emb_i(a_i)$ and

$$a_{i'} \xrightarrow{p} a_i \Rightarrow Emb_{i'}(a_{i'}) \xrightarrow{p} a_i \text{ for all } i \leq i', \quad (3)$$

$a_i \in \mathcal{M}_i$, $a_{i'} \in \mathcal{M}_{i'}$ and $a \in \mathcal{M}$. If \mathcal{M} is a target for a *prefactor system with projections* $\mathcal{M}_{\mathcal{I}}$, then there are moreover \asymp -projections $Proj_i : \mathcal{M} \rightarrow \mathcal{M}_i$ such that $proj_i^{i'}(Proj_{i'}(a)) \asymp Proj_i(a)$ and

$$a \xrightarrow{p} a_i \Rightarrow Proj_{i'}(a) \xrightarrow{p} a_i \text{ for all } i \leq i'. \quad (4)$$

If \mathcal{M} is a target for a *factor system*, then Emb and $Proj$ with these properties exist and both form a \asymp -embedding-projection pair. We write $(\mathcal{M}, \xrightarrow{p})$, $(\mathcal{M}, \xrightarrow{p}, Emb)$, $(\mathcal{M}, \xrightarrow{p}, Proj)$ and $(\mathcal{M}, \xrightarrow{p}, Emb, Proj)$, respectively, for these targets. If \mathcal{M} is a target for a prefactor system with projections $\mathcal{M}_{\mathcal{I}}$, then the projections $Proj_i$ can be defined by

$$Proj_i(a) := proj_i^{i'}(a_{i'}) \text{ for some } i' \geq i \text{ with } a \xrightarrow{p} a_{i'} \quad (5)$$

for $a \in \mathcal{M}$. It follows from the properties of a prefactor system that $Proj_i(a)$ is unique modulo \asymp . Since $Emb_{i'}(a_{i'}) \xrightarrow{p} a_{i'}$ for $a_{i'} \in \mathcal{M}_{i'}$ by (3), we have for $i \leq i'$

$$Proj_i(Emb_{i'}(a_{i'})) \asymp proj_i^{i'}(a_{i'}). \quad (6)$$

A target $(\mathcal{M}, \xrightarrow{p})$ for a system $\mathcal{M}_{\mathcal{I}}$ is a *limit* of $\mathcal{M}_{\mathcal{I}}$ iff for every further target $(\mathcal{N}, \xrightarrow{q})$ for $\mathcal{M}_{\mathcal{I}}$ there is a unique map $\Phi : \mathcal{N} \rightarrow \mathcal{M}$ such that $a \xrightarrow{q} a_i$ implies $\Phi(a) \xrightarrow{p} a_i$. If the underlying system is a factor system or a prefactor system, then we call the limit *factor limit* and *prefactor limit*, resp. It turns out, however, that a factor limit is the same as the prefactor limit, and that this limit $\underline{\lim}(\mathcal{M}_{\mathcal{I}})$ is unique modulo isomorphism. Therefore we simply speak of a *limit*, or a *limit set*, if we want to distinguish it from a *limit element* in this limit set.

► **Example 8.** Recall Example 3. If a direct system is also a direct factor system and $a_{i'} \xrightarrow{p} a_i : \iff emb_i^{i''}(a_i) = emb_{i'}^{i''}(a_{i'})$ for some $i'' \geq i, i'$, then the factor limit is the direct limit, i.e., $\underline{\lim}(\mathcal{M}_{\mathcal{I}}) = \underline{\lim}(\mathcal{M}_{\mathcal{I}})$. If an inverse system is additionally an inverse factor system and $a_{i'} \xrightarrow{p} a_i : \iff proj_i^{i'}(a_{i'}) = a_i$, then the factor limit $\underline{\lim}(\mathcal{M}_{\mathcal{I}})$ is the inverse limit $\overleftarrow{\lim}(\mathcal{M}_{\mathcal{I}})$.

In the first case of Example 4 the limit is \mathbb{N} (which is the direct limit as well as the factor limit). In the second case the limit (inverse limit and factor limit) is $\mathbb{N}_{\infty} := \mathbb{N} \cup \{\infty\}$ with $\infty = (0, 1, 2, \dots)$. In the third case the factor limit is a singleton set, which is neither the direct nor the inverse limit.

2.4 Consistent Sets and Dynamic Elements

It is possible to define concrete targets and limits in the form of sets of states: A set $\alpha \subseteq \bigcup_{i \in \mathcal{I}} \mathcal{M}_i$ in a system $\mathcal{M}_{\mathcal{I}}$ is called a *consistent set* iff $a_{i'} \xrightarrow{p} a_i$ holds for all $a_{i'}, a_i \in \alpha$ with $i' \geq i$, and $\mathcal{I}_{\alpha} := \{i \in \mathcal{I} \mid \alpha \cap \mathcal{M}_i \neq \emptyset\} \in \mathfrak{D}(\mathcal{I})$. If we already have a target \mathcal{M} for a prefactor system, then the set $Ext(a)$ is such a consistent set for all $a \in \mathcal{M}$. The set of all consistent sets in a (pre)factor system $\mathcal{M}_{\mathcal{I}}$ is itself a target for $\mathcal{M}_{\mathcal{I}}$ with $\alpha \xrightarrow{p} a_i : \iff a_i \in \alpha$. In other words, if \mathcal{M} denotes the set of all consistent sets, then the target is (\mathcal{M}, \exists) .

A *dynamic element* is a maximal (w.r.t. subset inclusion) consistent set, and for each consistent set α in a prefactor system there is exactly one dynamic element α^m such that $\alpha \subseteq \alpha^m$. Let $\mathcal{EL}(\mathcal{M}_{\mathcal{I}})$ denote the set of all of these dynamic elements, then

$$(\mathcal{EL}(\mathcal{M}_{\mathcal{I}}), \exists) \quad (7)$$

6:12 Potential Infinite Models of Type Theory

is a prefactor limit of $\mathcal{M}_{\mathcal{I}}$. For the next two lemmata, recall that it is assumed that $\mathfrak{D}(\mathcal{I})$ is a filter as defined in (Filter).

► **Lemma 9.** *Let α be a consistent set in a prefactor system $(\mathcal{M}_{\mathcal{I}}, \overset{p}{\mapsto})$ and $b_i \in \mathcal{M}_i$, then the following are equivalent:*

1. $b_i \in \alpha^m$.
2. $\alpha \cup \{b_i\}$ is a consistent set.
3. There are cofinal many $i' \in \mathcal{I}$ with $a_{i'} \overset{p}{\mapsto} b_i$ for some $a_{i'} \in \alpha$.
4. $a_{i'} \overset{p}{\mapsto} b_i$ for all $a_{i'} \in \alpha$ with $i' \geq i$.
5. There are \mathfrak{D} -many $i' \in \mathcal{I}$ with $a_{i'} \overset{p}{\mapsto} b_i$ for some $a_{i'} \in \alpha$.

Proof. This has been shown in [5], except the last clause. For its equivalence to the other statements notice that it follows from Clause 4. since $\mathcal{I}_\alpha \cap \uparrow i \in \mathfrak{D}(\mathcal{I})$, and it implies Clause 3. since each set in $\mathfrak{D}(\mathcal{I})$ is cofinal. ◀

► **Lemma 10.** *Let α and β be consistent sets in a prefactor system $(\mathcal{M}_{\mathcal{I}}, \overset{p}{\mapsto})$, then the following are equivalent:*

1. $\alpha^m = \beta^m$.
2. $a_i \asymp b_i$ for all $i \in \mathcal{I}$ with $a_i \in \alpha$ and $b_i \in \beta$.
3. $a_i \asymp b_j$ for all $i, j \in \mathcal{I}$ with $a_i \in \alpha$ and $b_j \in \beta$.

Proof. The equivalence of 1. and 3. has been shown in [5], so it suffices to prove that 2. implies $b_i \in \alpha^m$ for each $b_i \in \beta$. We wish to find cofinal many $a_{i'} \in \alpha$ with $a_{i'} \overset{p}{\mapsto} b_i$ and apply Lemma 9. There are \mathfrak{D} -many $i' \geq i$ with $i' \in \mathcal{I}_\alpha \cap \mathcal{I}_\beta$. For all $a_{i'} \in \alpha$, $b_{i'} \in \beta$ we have $a_{i'} \asymp b_{i'}$ by assumption, hence $a_{i'} \overset{p}{\mapsto} b_i$, since $b_{i'} \overset{p}{\mapsto} b_i$. ◀

If one, and hence all, of the conditions in Lemma 10 are true, then we write $\alpha \sim \beta$. Call an element $a \in \mathcal{M}$ of a target \mathcal{M} for a prefactor system $\mathcal{M}_{\mathcal{I}}$ a *limit (element)* of a consistent set α iff $\alpha \sim \text{Ext}(a)$. Obviously, a is a limit of its extension $\text{Ext}(a)$. If there is only one limit element, we denote it as $\text{lim}(\alpha)$.

Given a factor system $\mathcal{M}_{\mathcal{I}}$. Then $\mathcal{EL}(\mathcal{M}_{\mathcal{I}})$ is, up to isomorphism, the limit $\underline{\text{lim}}(\mathcal{M}_{\mathcal{I}})$, whereby projections have been defined by (5) and embeddings are

$$\text{Emb}_i(a_i) := \{ \text{emb}_i^{i'}(a_i) \in \bigcup_{j \in \mathcal{I}} \mathcal{M}_j \mid i' \geq i \}^m. \quad (8)$$

Let \mathcal{M} be a target for a system $\mathcal{M}_{\mathcal{I}}$, then \mathcal{M} is *maximal (over $\mathcal{M}_{\mathcal{I}}$)* iff $\text{Ext}(a) \in \mathcal{EL}(\mathcal{M}_{\mathcal{I}})$ for all $a \in \mathcal{M}$. \mathcal{M} is *extensional (over $\mathcal{M}_{\mathcal{I}}$)* iff $\text{Ext}(a) \sim \text{Ext}(b)$ implies $a = b$ for all $a, b \in \mathcal{M}$. \mathcal{M} is *complete (over $\mathcal{M}_{\mathcal{I}}$)* iff for all consistent sets α there is a limit element a , i.e., some $a \in \mathcal{M}$ with $\text{Ext}(a) \sim \alpha$. One can characterize a limit $\underline{\text{lim}}(\mathcal{M}_{\mathcal{I}})$ also as a target that is maximal, extensional and complete over $\mathcal{M}_{\mathcal{I}}$.

► **Proposition 11.** *Assume a prefactor system $\mathcal{M}_{\mathcal{I}}$ has been compactified with a target \mathcal{M} , yielding the extended prefactor system $\mathcal{M}_{\bar{\mathcal{I}}}$.*

1. If $\mathcal{M}_{\mathcal{I}}$ is stable and \mathcal{M} maximal over $\mathcal{M}_{\mathcal{I}}$, then $\mathcal{M}_{\bar{\mathcal{I}}}$ is stable, too.
2. If $\mathcal{M}_{\mathcal{I}}$ is direct and \mathcal{M} extensional over $\mathcal{M}_{\mathcal{I}}$, then $\mathcal{M}_{\bar{\mathcal{I}}}$ is direct and \mathcal{I}_a contains an up-set for all $a \in \mathcal{M}$.
3. If $\mathcal{M}_{\mathcal{I}}$ is inverse and \mathcal{M} maximal over $\mathcal{M}_{\mathcal{I}}$, then $\mathcal{M}_{\bar{\mathcal{I}}}$ is inverse and $\mathcal{I}_a = \mathcal{I}$ for all $a \in \mathcal{M}$.

Proof.

1. In order to show stability of $\mathcal{M}_{\bar{\mathcal{I}}}$ it suffices to prove that $a \dashv^p a_i \asymp b_i$ implies $a \dashv^p b_i$ for all $a \in \mathcal{M}$. Indeed, there are cofinal many indices $i' \geq i$ with $a \dashv^p a_{i'}$, and consequently $a_{i'} \xrightarrow{p} a_i$. Applying (Stab) yields $a_{i'} \xrightarrow{p} b_i$ for cofinal many indices $i' \in \mathcal{I}$. By Lemma 9 $a \dashv^p b_i$ follows, since $Ext(a)$ is a maximal consistent set.
2. We claim that $a \dashv^p a_i \iff a = Emb_i(a_i)$ for $a \in \mathcal{M}$. So assume $a \dashv^p a_i$ and we first establish that $a \dashv^p a_j$ implies $Emb_i(a_i) \dashv^p a_j$ for all $j \in \mathcal{I}$. There are cofinal many $i' \geq i, j$ with $a \dashv^p a_{i'}$, $a_{i'} \xrightarrow{p} a_i$ and $a_{i'} \dashv^p a_j$, hence $a_{i'} \asymp emb_i^{i'}(a_i)$, because $\mathcal{M}_{\mathcal{I}}$ is direct. It follows that $emb_i^{i'}(a_i) \dashv^p a_j$ for cofinal many indices $i' \in \mathcal{I}$ and thus $Emb_i(a_i) \dashv^p a_j$. This proves $Ext(a) \sim Ext(Emb_i(a_i))$, so a and $Emb_i(a_i)$ are equal since \mathcal{M} is extensional over $\mathcal{M}_{\mathcal{I}}$. The inverse implication $a = Emb_i(a_i) \Rightarrow a \dashv^p a_i$ holds trivially. Moreover, $a = Emb_i(a_i)$ for $a \in \mathcal{M}$ implies that \mathcal{I}_a contains the up-set $\uparrow i$.
3. We shall prove that $a \dashv^p a_i \iff Proj_i(a) \asymp a_i$. For the forward implication assume $Proj_i(a) = proj_i^{i'}(a_{i'})$ for some $a_{i'}$ with $a \dashv^p a_{i'}$. We establish that $proj_i^{i'}(a_{i'}) \asymp a_i$ for $a \dashv^p a_i$ by using the fact that $\mathcal{M}_{\mathcal{I}}$ is inverse and that $a \dashv^p a_{i'}$ and $a \dashv^p a_i$ implies $a_{i'} \xrightarrow{p} a_i$.

For the backward implication assume $Proj_i(a) = proj_i^{i'}(a_{i'}) \asymp a_i$. To show $a \dashv^p a_i$ it suffices to find cofinal many indices $i'' \geq i'$ with $a \dashv^p a_{i''}$ and $a_{i''} \xrightarrow{p} a_i$. To this aim, we use maximality of \mathcal{M} . Indeed, there are cofinal many indices $i'' \geq i'$ with $a \dashv^p a_{i''}$, and to confirm that $a_{i''} \xrightarrow{p} a_i$ it suffices to prove $proj_i^{i''}(a_{i''}) \asymp a_i$. Now $a_{i''} \dashv^p a_{i'}$ is a consequence of $a \dashv^p a_{i''}$ and $a \dashv^p a_{i'}$, hence $proj_i^{i''}(a_{i''}) \asymp a_{i'}$. Because \asymp -projections preserve \asymp , we deduce $proj_i^{i''}(a_{i''}) \asymp proj_i^{i'}(proj_i^{i''}(a_{i''})) \asymp proj_i^{i'}(a_{i'}) \asymp a_i$, as claimed. For all $i \in \mathcal{I}$ there is some a_i with $a \dashv^p a_i \asymp Proj_i(a)$, hence $\mathcal{I}_\alpha = \mathcal{I}$. \blacktriangleleft

► **Corollary 12.** *Assume a prefactor system $\mathcal{M}_{\mathcal{I}}$ has been compactified with a limit \mathcal{M} , yielding the extended prefactor system $\mathcal{M}_{\bar{\mathcal{I}}}$. If $\mathcal{M}_{\mathcal{I}}$ is stable (direct, inverse), then $\mathcal{M}_{\bar{\mathcal{I}}}$ is also stable (direct, inverse resp.).*

2.5 Targets and Limits on the Function Space

Consider two factor systems $\mathcal{M}_{\mathcal{I}}$ and $\mathcal{N}_{\mathcal{J}}$ with limits \mathcal{M} and \mathcal{N} resp. Then

$$[\mathcal{M} \rightarrow_{\mathfrak{D}} \mathcal{N}] := \{f : \mathcal{M} \rightarrow \mathcal{N} \mid \mathcal{I}_f \in \mathfrak{D}(\mathcal{I} \times \mathcal{J})\}$$

is a target for $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$, but not necessarily a limit. We now state a condition which guarantees that $[\mathcal{M} \rightarrow_{\mathfrak{D}} \mathcal{N}]$ is indeed a limit. So let ζ be a consistent set on the factor system $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$, and α a consistent set on the factor system $\mathcal{M}_{\mathcal{I}}$. Define

$$App(\zeta, \alpha) := \zeta(\alpha) := \{f_{i \rightarrow j}(a_i) \mid f_{i \rightarrow j} \in \zeta \text{ and } a_i \in \alpha\}. \quad (\text{Appl})$$

It is easy to see that this set satisfies $b_{j'} \xrightarrow{p} b_j$ for all $j \leq j'$ and $b_j, b_{j'} \in \zeta(\alpha)$. However, $\zeta(\alpha)$ is not necessarily a consistent set. For that, $\mathcal{I}_f[\mathcal{I}_a] \in \mathfrak{D}(\mathcal{J})$ must be the case, whereby $\mathcal{H}[\mathcal{I}'] := \{j \in \mathcal{J} \mid \exists i \in \mathcal{I}' \ i \rightarrow j \in \mathcal{H}\}$ for $\mathcal{H} \subseteq \mathcal{I} \times \mathcal{J}$ and $\mathcal{I}' \subseteq \mathcal{I}$. This is ensured when the following condition for \mathfrak{D} is met for all sets \mathcal{I} and \mathcal{J} :

$$\mathcal{H} \in \mathfrak{D}(\mathcal{I} \times \mathcal{J}) \text{ and } \mathcal{I}' \in \mathfrak{D}(\mathcal{I}) \text{ implies } \mathcal{H}[\mathcal{I}'] \in \mathfrak{D}(\mathcal{J}). \quad (\text{D})$$

We will require Condition (D) for the rest of this paper.

► **Lemma 13.** *The application defined by (Appl) is extensional, i.e., for all $\zeta, \zeta' \in \mathcal{EL}([\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}])$ we have:*

$$\zeta(\alpha) = \zeta'(\alpha) \text{ for all } \alpha \in \mathcal{EL}(\mathcal{M}_{\mathcal{I}}) \text{ implies } \zeta = \zeta'.$$

Proof. Assume $\zeta(\alpha) = \zeta'(\alpha)$ for all $\alpha \in \mathcal{EL}(\mathcal{M}_{\mathcal{I}})$ and let $f_{i \rightarrow j} \in \zeta$. We will show that $f_{i \rightarrow j} \in \zeta'$, which proves the lemma. Choose indices $i' \rightarrow j' \in \mathcal{I}_{\zeta} \cap \mathcal{I}_{\zeta'} \cap \uparrow(i \rightarrow j) \in \mathfrak{D}(\mathcal{I} \times \mathcal{J})$ and let $f'_{i' \rightarrow j'} \in \zeta'$. We wish to show $f'_{i' \rightarrow j'} \xrightarrow{p} f_{i \rightarrow j}$. Since this is the case for \mathfrak{D} -many indices $i' \rightarrow j'$, we have shown $f_{i \rightarrow j} \in \zeta'$ by Lemma 9, which suffices.

Now assume $a_{i'} \xrightarrow{p} a_i$ for $a_{i'} \in \mathcal{M}_{i'}$ and $a_i \in \mathcal{M}_i$, so that we have to prove $f'_{i' \rightarrow j'}(a_{i'}) \xrightarrow{p} f_{i \rightarrow j}(a_i)$. For $\alpha := \text{Emb}_{i'}(a_{i'}) \in \mathcal{EL}(\mathcal{M}_{\mathcal{I}})$ we know that $a_i \in \alpha$, by maximality and (3). Further, $f'_{i' \rightarrow j'}(a_{i'}) \in \zeta'(\alpha) = \zeta(\alpha)$ and $f_{i \rightarrow j}(a_i) \in \zeta(\alpha)$ implies $f'_{i' \rightarrow j'}(a_{i'}) \xrightarrow{p} f_{i \rightarrow j}(a_i)$, because $\zeta(\alpha)$ is a consistent set. \blacktriangleleft

► **Proposition 14.** *Let \mathcal{M} and \mathcal{N} be a targets for the factor systems $\mathcal{M}_{\mathcal{I}}$ and $\mathcal{N}_{\mathcal{J}}$ resp. Assume Condition (D) holds. If \mathcal{N} is maximal (extensional, complete) over $\mathcal{N}_{\mathcal{J}}$, so is $[\mathcal{M} \rightarrow_{\mathfrak{D}} \mathcal{N}]$ over $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$.*

Proof. First we shall prove the claim about maximality, so let $g_{i \rightarrow j} \in \mathcal{M}_{i \rightarrow j}$ and we apply Lemma 9. Assume there is a set $\mathcal{H} \in \mathfrak{D}(\mathcal{I} \times \mathcal{J})$ with $f \xrightarrow{p} f_{i' \rightarrow j'} \xrightarrow{p} g_{i \rightarrow j}$ for $i' \rightarrow j' \in \mathcal{H}$, then it suffices to show $f \xrightarrow{p} g_{i \rightarrow j}$. By the definition of \xrightarrow{p} it must be checked that

$$f(a) \xrightarrow{p} g_{i \rightarrow j}(a_i) \text{ for } a \xrightarrow{p} a_i.$$

There are \mathfrak{D} -many indices $i' \geq i$ with $a \xrightarrow{p} a_{i'}$, since $\mathcal{I}_a \cap \uparrow i \in \mathfrak{D}(\mathcal{I})$. They satisfy $a_{i'} \xrightarrow{p} a_i$, because $a \xrightarrow{p} a_{i'}$ and $a \xrightarrow{p} a_i$. Applying Condition (D) give us \mathfrak{D} -many indices $j' \in \mathcal{H}[\mathcal{I}_a \cap \uparrow i]$, which all satisfy $f(a) \xrightarrow{p} f_{i' \rightarrow j'}(a_{i'}) \xrightarrow{p} g_{i \rightarrow j}(a_i)$. So maximality of \mathcal{N} yields $f(a) \xrightarrow{p} g_{i \rightarrow j}(a_i)$, showing the maximality of $[\mathcal{M} \rightarrow_{\mathfrak{D}} \mathcal{N}]$.

For extensionality we prove that $f = g$, provided $\text{Ext}(f) \sim \text{Ext}(g)$. So let $a \in \mathcal{M}$ and we claim that $f(a) = g(a)$. For all $i \rightarrow j \in \mathcal{I}_f \cap \mathcal{I}_g$ and $f_{i \rightarrow j} \in \text{Ext}(f)$, $g_{i \rightarrow j} \in \text{Ext}(g)$ we have $f_{i \rightarrow j} \asymp g_{i \rightarrow j}$ by definition. Condition (D) shows that

$$\mathcal{J}' := (\mathcal{I}_f \cap \mathcal{I}_g)[\mathcal{I}_a] \in \mathfrak{D}(\mathcal{J}),$$

and for all these indices $i \in \mathcal{I}_a$ and $j \in \mathcal{J}'$ we have $a \xrightarrow{p} a_i$ for some $a_i \in \mathcal{M}_i$, $f(a) \xrightarrow{p} f_{i \rightarrow j}(a_i)$, $g(a) \xrightarrow{p} g_{i \rightarrow j}(a_i)$ and $f_{i \rightarrow j}(a_i) \asymp g_{i \rightarrow j}(a_i)$. This yields $\text{Ext}(f(a)) \sim \text{Ext}(g(a))$, since the elements of $\text{Ext}(f(a))$ and $\text{Ext}(g(a))$ are consistent (i.e., in \asymp -relation to each other) for cofinal many indices $j \in \mathcal{J}$. We now appeal to extensionality of \mathcal{N} to get $f(a) = g(a)$.

To show completeness of $[\mathcal{M} \rightarrow_{\mathfrak{D}} \mathcal{N}]$ over $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$ let ζ be a consistent set in $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$. We define a limit element $f : \mathcal{M} \rightarrow \mathcal{N}$ of ζ by

$$f(a) := \text{limit element of } \beta := \{g_{i \rightarrow j}(a_i) \mid g_{i \rightarrow j} \in \zeta \cap [\mathcal{M}_i \rightarrow \mathcal{N}_j] \text{ and } a \xrightarrow{p} a_i\}$$

for all $a \in \mathcal{M}$. This function is well-defined: If $a \xrightarrow{p} a_i$, $a \xrightarrow{p} b_i$ and $f_{i \rightarrow j}, g_{i \rightarrow j} \in \zeta \cap [\mathcal{M}_i \rightarrow \mathcal{N}_j]$, then $a_i \asymp b_i$ follows and consequently $f_{i \rightarrow j}(a_i) \asymp g_{i \rightarrow j}(b_i)$; note that $a_i \asymp b_i \iff a_i \xrightarrow{p} b_i$ and the same for $f_{i \rightarrow j}(a_i) \asymp g_{i \rightarrow j}(b_i)$. This implies that any two elements in β are consistent. Condition (D) guarantees that $\mathcal{I}_f = \mathcal{I}_{\zeta}[\mathcal{I}_a] \in \mathfrak{D}(\mathcal{J})$. The set β is thus a consistent set and completeness of \mathcal{N} ensures that a limit element of β exists.

We claim that f is a limit of ζ . Let $f \xrightarrow{p} f_{i \rightarrow j}$ and $g_{i \rightarrow j} \in \zeta$ with $f_{i \rightarrow j}, g_{i \rightarrow j} \in \mathcal{M}_{i \rightarrow j}$ and we have to show $f_{i \rightarrow j} \xrightarrow{p} g_{i \rightarrow j}$. Given $b_i \xrightarrow{p} a_i$, both in \mathcal{M}_i , so it suffices to show that $f_{i \rightarrow j}(b_i) \xrightarrow{p} g_{i \rightarrow j}(a_i)$. Take $a := \text{Emb}_i(a_i) = \text{Emb}_i(b_i)$, then $a \xrightarrow{p} a_i$ and $a \xrightarrow{p} b_i$, hence $f(a) \xrightarrow{p} f_{i \rightarrow j}(b_i)$ and $g_{i \rightarrow j}(a_i) \in \beta$. Since $f(a)$ is a limit element of β , it follows that $\beta \sim \text{Ext}(f(a))$, and thus $f_{i \rightarrow j}(b_i) \asymp g_{i \rightarrow j}(a_i)$ by Lemma 10, so we are done. \blacktriangleleft

Recall that limits are targets which are maximal, extensional and complete. Recall also that $\mathcal{EL}([\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}])$ is a limit of $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$ and that limits are unique modulo isomorphism.

► **Corollary 15.** *Given two factor systems $\mathcal{M}_{\mathcal{I}}$ and $\mathcal{N}_{\mathcal{J}}$ with limits \mathcal{M} and \mathcal{N} resp. Assume Condition (D) holds, then $[\mathcal{M} \rightarrow_{\mathfrak{D}} \mathcal{N}]$ is a limit of $[\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}]$. Equivalently stated, there is an isomorphism $[\mathcal{EL}(\mathcal{M}_{\mathcal{I}}) \rightarrow_{\mathfrak{D}} \mathcal{EL}(\mathcal{N}_{\mathcal{J}})] \simeq \mathcal{EL}([\mathcal{M}_{\mathcal{I}} \rightarrow \mathcal{N}_{\mathcal{J}}])$.*

3 Towards Models and Interpretations

This section contains the first steps towards an interpretation of a typed λ -term in a factor system and a limit of it. Let types $\varrho \in Typ$ be given by $\varrho ::= \iota \mid \varrho \rightarrow \varrho$ for base types ι . We assume that type *bool* is one of these base types. For each type ϱ let a non-empty, directed set \mathcal{I}_{ϱ} of indices be given and let $\mathfrak{D}_{\Gamma} := \mathfrak{D}(\mathcal{I}_{\Gamma})$ be a filter which satisfies (Filter). Maps on indices, such as Emb_i , extend in an obvious way to state contexts $C = (i_0, \dots, i_{n-1})$ as products, e.g. $Emb_C = Emb_{i_0} \times \dots \times Emb_{i_{n-1}}$.

Given also a basic version of a simply typed λ -calculus (without constants) and for convenience we use a fixed list of variables x_0, x_1, x_2, \dots within the Church-style λ -terms $r ::= x_k \mid rr \mid \lambda x_k^{\varrho} r$. We apply the usual conventions for λ -terms [11].

3.1 Judgements for Types and States

Typing judgements $\Gamma \vdash r : \varrho$ with $\Gamma = (\varrho_0, \dots, \varrho_{n-1})$ are defined recursively as follows:

$$\text{(VAR)} \frac{k < n}{\Gamma \vdash x_k : \varrho_k} \quad \text{(APP)} \frac{\Gamma \vdash r : \varrho \rightarrow \sigma \quad \Gamma \vdash s : \varrho}{\Gamma \vdash rs : \sigma} \quad \text{(ABS)} \frac{\Gamma.\varrho \vdash r : \sigma}{\Gamma \vdash \lambda x_n^{\varrho} r : \varrho \rightarrow \sigma}$$

We can write more explicitly $(x_0 : \varrho_0, \dots, x_{n-1} : \varrho_{n-1}, x_n : \varrho_n)$ for the context $\Gamma.\varrho_n$, so λ -abstractions binds the last variable of the context.³ When we speak of typed terms this refers to this typing judgement. It is well known that each term r in the simply typed λ -calculus has a unique type for a given context Γ .

► **Definition 16.** *Let ι denote a base type. The positive and negative types are defined as*

$$Typ^+ \ni \varrho^+ ::= \iota \mid \varrho^- \rightarrow \varrho^+ \quad \text{and} \quad Typ^- \ni \varrho^- ::= \text{bool} \mid \varrho^+ \rightarrow \varrho^-.$$

For an index $i \in \mathcal{I}_{\varrho}$ with $\varrho \in Typ^+$ we write $i \in Idx^+$, and similar for $i \in Idx^-$. Moreover, $Typ^c \ni \varrho^c ::= \iota \mid \varrho^+ \rightarrow \varrho^c \mid \varrho^- \rightarrow \varrho^c$.

For example, *nat* is a positive type, $\text{nat} \rightarrow \text{nat} \rightarrow \text{bool}$ is a negative type, $\text{bool} \rightarrow \text{bool}$ is both and $\text{nat} \rightarrow \text{nat}$ is neither. Obviously $Typ^+ \cup Typ^- \subseteq Typ^c$ and $\varrho \in Typ^+ \cap Typ^- \iff \varrho ::= \text{bool} \mid \varrho \rightarrow \varrho$. In order to formulate the rules for states, we use a fragment of the simply typed λ -calculus, which we refer to as the *core fragment*. This fragment is constrained in such a way that contexts $\Gamma = (\varrho_0, \dots, \varrho_{n-1})$ contain only positive or negative types, that is,

$$\varrho_k \in Typ^+ \cup Typ^- \text{ for all } \varrho_k \text{ in } \Gamma. \tag{9}$$

A more general definition would permit the use of further base types in place of the negative type *bool*. This requires that the base type in question has a finite set of objects and an interpretation analogous to that of type *bool* in Definition 18. The primary motivation for this definition is the existence of suitable rules for state judgements of variables and the fact that positive types can be interpreted as direct factor systems and negative types as inverse factor systems.

► **Lemma 17.** *The typed term $\Gamma \vdash r : \varrho$ is in the core fragment iff $\Gamma \rightarrow \varrho \in Typ^c$.*

³ This style is basically the de Bruijn level notation of λ -terms, see e.g. [6].

6:16 Potential Infinite Models of Type Theory

This lemma follows easily by induction on $\Gamma \vdash r : \varrho$. Therein $\Gamma \rightarrow \sigma$ is defined by $() \rightarrow \sigma := \sigma$ and $\Gamma.\varrho \rightarrow \sigma := \Gamma \rightarrow (\varrho \rightarrow \sigma)$; the index $C \rightarrow i$ is defined in the same way as $\Gamma \rightarrow \sigma$. We now introduce judgements for states of the form $C \mid r : i$. They require a typing judgement $\Gamma \vdash r : \varrho$ such that $C \rightarrow i \in \mathcal{I}_{\Gamma \rightarrow \varrho}$. The rules are based on the typing rules⁴, whereby $C = (i_0, \dots, i_{n-1})$.

$$\begin{array}{c} \text{(VAR+)} \frac{j \geq i_k \in Idx^+}{C \mid x_k : j} \\ \text{(APP)} \frac{C \mid r : i \rightarrow j \quad C \mid s : i}{C \mid rs : j} \end{array} \qquad \begin{array}{c} \text{(VAR-)} \frac{j \leq i_k \in Idx^-}{C \mid x_k : j} \\ \text{(ABS)} \frac{C.i \mid r : j}{C \mid \lambda x_n^{\varrho} r : i \rightarrow j} \end{array}$$

The condition $j \geq i_k$ in Rule (VAR+) is a consequence of the fact that objects a_{i_k} of positive type exist at all later stages j as $emb_{i_k}^j(a_{i_k})$. The condition $j \leq i_k$ in Rule (VAR-) is related to the fact that we find unique restrictions of a relation, but not unique extensions. Note that for an index $i_k \in Idx^+ \cap Idx^-$ both of the rules for variables apply. In this case, however, there is only one index, so both rules are the same.

A term usually has several state judgement, not only one. Indeed, we expect that there are \mathfrak{D} -many of them and it seems necessary to prove this stronger property in order to guarantee that at least one judgement exists. However, we have not shown that all terms of the core fragment have a state judgement. It is possible to define sets \mathfrak{D}_Γ , similar as in [4], but with a reorder of the context Γ that takes the positive types first, and afterward the negative ones. In a subsequent paper we will present such a definition of sets \mathfrak{D}_Γ satisfying Condition (D) and a proof that each term in the above mentioned fragment has indeed \mathfrak{D} -many state judgements.

3.2 Interpretation of Types

An interpretation of a type ϱ has a static and a dynamic part. Although the limit is uniquely defined from a structural perspective, the extension, i.e., the set of elements in the limit, depends on the stage of the meta-level investigation, assuming a consequent finitistic view. In this case, the limit is not the absolute end of the extensible system. As the investigation progresses, both the system and the limit increase. The main part of the interpretation is the factor system – the limit is necessary to prove that the definition in the factor system is correct.

Let $(\llbracket \varrho \rrbracket_i)_{i \in \mathcal{I}_\varrho}$ be the factor system that interprets type ϱ , which consists of finite sets $\llbracket \varrho \rrbracket_i$. The limit, up to isomorphism, is then $\llbracket \varrho \rrbracket := \mathcal{EL}((\llbracket \varrho \rrbracket_i)_{i \in \mathcal{I}_\varrho})$, see Section 2.4. These limit sets give rise to an extensional type structure if we use (Appl) as application. Recall Lemma 13 and our assumption of Condition (D).

► **Definition 18.** *An interpretation $\llbracket \cdot \rrbracket$ of types assigns to a type ϱ a factor system $(\llbracket \varrho \rrbracket_i)_{i \in \mathcal{I}_\varrho}$ and a limit $\llbracket \varrho \rrbracket$ of it. So the pair $((\llbracket \varrho \rrbracket_i)_{i \in \mathcal{I}_\varrho}, \llbracket \varrho \rrbracket)$ interprets the type ϱ . This interpretation shall satisfy the following properties:*

1. *For a base type ι , $(\llbracket \iota \rrbracket_i)_{i \in \mathcal{I}_\iota}$ is a direct factor system.*
2. *Type $bool$ is interpreted by the factor system $\mathbb{B}_{\{bool\}}$ with $\mathbb{B}_{bool} = \mathbb{B} = \{true, false\}$ and limit \mathbb{B} .*
3. *The interpretation of $\varrho \rightarrow \sigma$ is the factor system $[(\llbracket \varrho \rrbracket_i)_{i \in \mathcal{I}_\varrho} \rightarrow (\llbracket \sigma \rrbracket_j)_{j \in \mathcal{I}_\sigma}]$ together with the limit $\llbracket \varrho \rrbracket \rightarrow_{\mathfrak{D}} \llbracket \sigma \rrbracket$.*

⁴ One may also combine both rules into one. The rules are however more readable if one keeps them separated.

Since Condition (D) holds, it follows from Corollary 15 that the function space $\llbracket \varrho \rrbracket \rightarrow \mathfrak{D} \llbracket \sigma \rrbracket$ is indeed the limit of the underlying factor system $[(\llbracket \varrho \rrbracket_i)_{i \in \mathcal{I}_\varrho} \rightarrow (\llbracket \sigma \rrbracket_j)_{j \in \mathcal{I}_\sigma}]$. Definition 18 extends to contexts Γ in the usual way by taking products.

► **Example 19.** The standard interpretation of the natural numbers $\llbracket nat \rrbracket = \mathbb{N}$ is the limit of the direct factor system $(\llbracket nat \rrbracket_i)_{i \in \mathbb{N}^+}$ with $\llbracket nat \rrbracket_i = \mathbb{N}_i$.⁵ Of course, there are other non-standard models of *nat* with non-standard natural numbers as well. These non-standard numbers do not appear at the limit step, but at some stage within the system.

The next lemma is required for the reflection principle stated in Theorem 24. Its proof proceeds by induction on Definition 16 and uses Proposition 7 and Corollary 12.

- **Lemma 20.** *Given an interpretation of types $\llbracket \cdot \rrbracket$ and let $i \leq i'$ with $i, i' \in \mathcal{I}_\varrho$ for a type ϱ .*
1. *If $\varrho \in \text{Typ}^+$, then the factor system $(\llbracket \varrho \rrbracket_i)_{i \in \mathcal{I}_\varrho}$ is direct and $a \mapsto^p a_i$ implies $a \mapsto^p \text{emb}_i^{i'}(a_i)$.*
 2. *If $\varrho \in \text{Typ}^-$, then the factor system is inverse and $a \mapsto^p a_{i'}$ implies $a \mapsto^p \text{proj}_i^{i'}(a_{i'})$.*

3.3 Interpretation of Terms and a First Version of the Reflection Principle

The interpretation of terms in the type structure of the limit sets is the common interpretation. What is new is the interpretation in the factor system. This part of the interpretation requires a state judgement, so it is defined only for the core fragment, defined in Section 3.1.

► **Definition 21.** *A Γ -environment is a list of elements $\mathbf{A} \in \llbracket \Gamma \rrbracket := \llbracket \varrho_0 \rrbracket \times \cdots \times \llbracket \varrho_{n-1} \rrbracket$ for $\Gamma = (\varrho_0, \dots, \varrho_{n-1})$. The value $\llbracket r \rrbracket_{\mathbf{A}} \in \llbracket \varrho \rrbracket$ of a typed term $\Gamma \vdash r : \varrho$ is defined recursively on the derivation of the judgement $\Gamma \vdash r : \varrho$ relative to a Γ -environment $\mathbf{A} \in \llbracket \Gamma \rrbracket$:*

$$\begin{aligned} \llbracket x_k \rrbracket_{\mathbf{A}} &:= A_k && \text{for } \Gamma \vdash x_k : \varrho_k, \\ \llbracket rs \rrbracket_{\mathbf{A}} &:= \llbracket r \rrbracket_{\mathbf{A}}(\llbracket s \rrbracket_{\mathbf{A}}) && \text{for } \Gamma \vdash r : \varrho \rightarrow \sigma \text{ and } \Gamma \vdash s : \varrho, \\ \llbracket \lambda x_n^{\varrho} r \rrbracket_{\mathbf{A}}(B) &:= \llbracket r \rrbracket_{\mathbf{A}.B} && \text{for } \Gamma.\varrho \vdash r : \sigma \text{ and } B \in \llbracket \varrho \rrbracket. \end{aligned}$$

► **Definition 22.** *A C -environment is a list of elements $\mathbf{a} \in \llbracket \Gamma \rrbracket_C := \llbracket \varrho_0 \rrbracket_{i_0} \times \cdots \times \llbracket \varrho_{n-1} \rrbracket_{i_{n-1}}$ for $C = (i_0, \dots, i_{n-1})$. The value $\llbracket r \rrbracket_{\mathbf{a}.C}^i \in \llbracket \varrho \rrbracket_i$ of a term with $C \mid r : i$ relative to a C -environment $\mathbf{a} \in \llbracket \Gamma \rrbracket_C$ is defined recursively on the derivation of the state judgement $C \mid r : i$:*

$$\begin{aligned} \llbracket x_k \rrbracket_{\mathbf{a}.C}^j &:= \text{emb}_{i_k}^j(a_k) && \text{for } C \mid x_k : i_k \text{ and } \varrho_k \in \text{Typ}^+, \\ \llbracket x_k \rrbracket_{\mathbf{a}.C}^j &:= \text{proj}_j^{i_k}(a_k) && \text{for } C \mid x_k : i_k \text{ and } \varrho_k \in \text{Typ}^-, \\ \llbracket rs \rrbracket_{\mathbf{a}.C}^j &:= \llbracket r \rrbracket_{\mathbf{a}.C}^{i \rightarrow j}(\llbracket s \rrbracket_{\mathbf{a}.C}^i) && \text{for } C \mid r : i \rightarrow j \text{ and } C \mid s : i, \\ \llbracket \lambda x_n^{\varrho} r \rrbracket_{\mathbf{a}.C}^{i \rightarrow j}(b) &:= \llbracket r \rrbracket_{\mathbf{a}.b.C.i}^j && \text{for } C.i \mid r : j \text{ and } b \in \llbracket \varrho \rrbracket_i. \end{aligned}$$

By definition, the value $\llbracket r \rrbracket_{\mathbf{a}.C}^i$ depends on the way the judgement $C \mid r : i$ has been derived. So different derivations could lead to different values, making this definition incorrect. The value $\llbracket rs \rrbracket_{\mathbf{a}.C}^j$ seemingly depend on the chosen index i used in $\llbracket s \rrbracket_{\mathbf{a}.C}^i$. A consequence of the main Theorem 24 is that the value $\llbracket r \rrbracket_{\mathbf{a}.C}^i$ is indeed independent (modulo \simeq) of the derivation of the state judgement $C \mid r : i$.

⁵ Here we assume that the interpretation of type *nat* is the set \mathbb{N} of all natural numbers, which is an actual infinite set if one accepts actual infinities at meta-level. If not, \mathbb{N} is an arbitrary large finite set, which corresponds to a set in the implicitly given factor system at meta-level, i.e., \mathbb{N} is \mathbb{N}_j for some sufficiently large index j depending on the stages of the meta-level investigation.

► **Definition 23.** Given a pair (\mathbf{A}, \mathbf{a}) with a Γ -environment \mathbf{A} and a C -environment \mathbf{a} such that $\mathbf{A} \vdash^p \mathbf{a}$. The interpretation of a typed term $\Gamma \vdash r : \varrho$ with state judgement $C \mid r : i$ relative to (\mathbf{A}, \mathbf{a}) is the pair $(\llbracket r \rrbracket_{\mathbf{A}}, \llbracket r \rrbracket_{\mathbf{a}:C}^i) \in \llbracket \varrho \rrbracket \times \llbracket \varrho \rrbracket_i$.

One would expect that $\mathbf{A} \vdash^p \mathbf{a}$ implies $\llbracket r \rrbracket_{\mathbf{A}} \vdash^p \llbracket r \rrbracket_{\mathbf{a}:C}^i$. Indeed, this is the content of the next theorem, which is also the main theorem.

► **Theorem 24.** Given a typed term $\Gamma \vdash r : \varrho$ with $C \mid r : i$. If $\mathbf{A} \in \llbracket \Gamma \rrbracket$, $\mathbf{a} \in \llbracket \Gamma \rrbracket_C$ are variable assignments with $\mathbf{A} \vdash^p \mathbf{a}$, then

$$\llbracket r \rrbracket_{\mathbf{A}} \vdash^p \llbracket r \rrbracket_{\mathbf{a}:C}^i.$$

Proof. The proof proceeds by induction on the derivation of $C \mid r : i$. For a variable x_k of a positive type we apply Lemma 20. This yields

$$\llbracket x_k \rrbracket_{\mathbf{A}} = A_k \vdash^p \text{emb}_{i_k}^j(a_k) = \llbracket x_k \rrbracket_{\mathbf{a}:C}^j,$$

since $A_k \vdash^p a_k$. A similar consideration holds for negative types. For application and abstraction the claim follows from the fact that \vdash^p is a logical relation. ◀

Consider the case of a closed term $r : \varrho$. Let us assume that ϱ is some data type, then the theorem states that $\llbracket r \rrbracket \vdash^p \llbracket r \rrbracket^i$. Recall that the predecessor relation \vdash^p is seen as a directed equality between two states of the same object. We can thus read \vdash^p , which is nothing more than \vdash^p applied limits, as an equality between $\llbracket r \rrbracket$ and $\llbracket r \rrbracket^i$. As indicated in Section 1.1, $\llbracket r \rrbracket^i$ is a strong form of an approximation of $\llbracket r \rrbracket$.

Due to the use of logical relations on higher types, application and λ -abstraction respect this equality. Since the type *bool* of propositions is part of the calculus, the equality holds also for truth values. The relation $\llbracket r \rrbracket \vdash^p \llbracket r \rrbracket^i$ for type *bool*, with the only values *true* and *false* in classical logic, is the identity. Consequently, truth in the limit and truth in a sufficiently large stage of the factor system coincide. Theorem 24 is the main result, and at the same time this theorem is necessary to show that the basic definition of the interpretation of a term, as introduced in Definition 22, is correct.

► **Corollary 25.** Let $\Gamma \vdash r : \varrho$ be a typed term with $i \leq i'$, $C \mid r : i$ and $C \mid r : i'$. Then $\llbracket r \rrbracket_{\mathbf{a}:C}^{i'} \vdash^p \llbracket r \rrbracket_{\mathbf{a}:C}^i$ holds for all C -environments $\mathbf{a} \in \llbracket \Gamma \rrbracket_C$.

Proof. Define $\mathbf{A} := \text{Emb}_C(\mathbf{a})$. Then $\mathbf{A} \vdash^p \mathbf{a}$ follows from Condition (Emb), hence $\llbracket r \rrbracket_{\mathbf{A}} \vdash^p \llbracket r \rrbracket_{\mathbf{a}:C}^{i'}$ and $\llbracket r \rrbracket_{\mathbf{A}} \vdash^p \llbracket r \rrbracket_{\mathbf{a}:C}^i$ by Theorem 24, and therefore $\llbracket r \rrbracket_{\mathbf{a}:C}^{i'} \vdash^p \llbracket r \rrbracket_{\mathbf{a}:C}^i$ by Condition (Factor). ◀

The independence of the value $\llbracket r \rrbracket_{\mathbf{a}:C}^i$ (modulo \asymp) from the state judgement follows from Corollary 25 if we take $i = i'$.

4 Conclusion and Further Work

We presented a model that can be used to interpret a fragment of the simply typed λ -calculus (which we called *core fragment*) based on the assumption that infinite sets are potential infinite. We gave a formalization of the potential infinite based on the filters \mathfrak{D} and sets \ll within these filters. This allows one to avoid any notion of actual infinity. The function space in this model is a family of finite function spaces and the model already has an interpretation of the logical type *bool* with the usual two truth values.

We introduced an interpretation of λ -terms from the core fragment which has a dynamic part, the factor system, and a static part, the limit of the factor system. Nevertheless, both parts are necessary, because from a dynamic point of view, the construction of a limit is not the end of the process, but an intermediate state.

The next step is to extend the λ -calculus to constants, in particular to the logical constants of implication and universal quantifier. The challenge here is that the universal quantifier is not continuous and cannot be interpreted as a higher-order functional. The solution is to introduce an additional rule with a side condition $C \ll i$ on the state judgments. The correctness of the interpretation uses the fact that propositions have “stable truth values” in the sense that for each proposition there is a stage in the model where the truth value of the proposition does not change anymore during further extensions. A corresponding interpretation for first-order logic has been given in [4].

The reflection principle from Theorem 24 then states that all objects and propositions (propositions are specific terms of type *bool*) in the limit are reflected in a sufficiently large state in the system. The interpretation is possible on the core fragment of simple type theory, not for all terms. However, this fragment includes a version of a classical higher-order logic. Unlike other models, such as domain theory [1], it does not require (actual) infinite sets at all, and it includes logic.⁶

We do not expect that the theorem about the reflection principle can be extended from the core fragment to all terms of simply typed λ -calculus without some kind of further restrictions. This is because variables of function type correspond to arbitrary functions, which can also be used to define higher-order functions by λ -abstraction. From the perspective of extensibility, this necessitates the consideration of both covariant (with respect to the codomain) and contravariant (with respect to the domain) extensions simultaneously. For instance, the totality of higher-order functions requires that, for each argument, which could be a function f of type $nat \rightarrow nat$, there must be a value. However, f is not given as a single entity. So one must identify a property that guarantees totality and conditions that ensure invariance, that is, existing properties must be preserved by future extensions of f . This is a significant challenge for higher-order functions and it is related to properties of the filter \mathfrak{D} . It is also the reason that for arbitrary functions, there are no simple state judgements as formulated in the rules (VAR+) and (VAR-). A future task will be to identify conditions under which the core fragment can be extended to include function variables.

References

- 1 Samson Abramsky and Achim Jung. Domain theory. In *Handbook of logic in computer science*. Oxford University Press, 1994.
- 2 Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, Amsterdam, 1985.
- 3 Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. doi:10.2307/2266170.
- 4 Matthias Eberl. A model theory for the potential infinite. *Reports on Mathematical Logic*, 57:3–30, 2022. doi:10.4467/20842589rm.22.001.16658.
- 5 Matthias Eberl. Higher-order concepts for the potential infinite. *Theoretical Computer Science*, 945:113667, 2023. doi:10.1016/j.tcs.2022.12.017.

⁶ The idea of a logic-enriched type theory [7] separates propositions from the underlying type theoretic framework. In our approach the logic is inside type theory, but the interpretation of the universal quantifier has an extra treatment.

- 6 Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proc. 14th LICS Conf.*, pages 193–202. IEEE, Computer Society Press, 1999. doi:10.1109/LICS.1999.782615.
- 7 Nicola Gambino and Peter Aczel. The generalised type-theoretic interpretation of constructive set theory. *The Journal of Symbolic Logic*, 71(1):67–103, 2006. doi:10.2178/js1/1140641163.
- 8 John Longley and Dag Normann. *Higher-order computability*, volume 100. Springer, 2015. doi:10.1007/978-3-662-47992-6.
- 9 Saunders MacLane and Ieke Moerdijk. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer Science & Business Media, 2012. doi:10.1007/978-1-4612-0927-0.
- 10 John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):99–124, 1991. doi:10.1016/0168-0072(91)90067-V.
- 11 Rob Nederpelt and Herman Geuvers. *Type theory and formal proof: an introduction*. Cambridge University Press, 2014.
- 12 Gordon Plotkin. *Lambda-definability and logical relations*. Edinburgh University, 1973.
- 13 Anne S. Troelstra. *Choice sequences. A chapter of intuitionistic mathematics. Oxford logic guides*, volume 44. Clarendon Press, 1979.
- 14 Anne S. Troelstra and Dirk van Dalen. Constructivism in mathematics. vol. i, volume 121 of *Studies in Logic and the Foundations of Mathematics*, 26, 1988.