



# Implementing a Type Theory with Observational Equality, Using Normalisation by Evaluation

Matthew Sirman

University of Cambridge, UK

Meven Lennon-Bertrand  

University of Cambridge, UK

Neel Krishnaswami  

University of Cambridge, UK

---

## Abstract

We report on an experimental implementation in Haskell of a dependent type theory featuring an observational equality type, based on Pujet et al.’s  $CC^{obs}$ . We use normalisation by evaluation to produce an efficient normalisation function, which is used to implement a bidirectional type checker. To allow for greater expressivity, we extend the core  $CC^{obs}$  calculus with quotient types and inductive types. To make the system usable, we explore various proof-assistant features, notably a rudimentary version of a “hole” system similar to Agda’s. While rather crude, this experience should inform other, more substantial implementation efforts of observational equality.

**2012 ACM Subject Classification** Software and its engineering → Functional languages; Theory of computation → Type theory; Theory of computation → Denotational semantics

**Keywords and phrases** Dependent type theory, Bidirectional typing, Observational equality, Normalisation by evaluation

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2024.5

**Related Version** *Dissertation*: <https://www.cl.cam.ac.uk/~nk480/sirman-dissertation.pdf> [30]

## Supplementary Material

*Software*: <https://github.com/matthew-sirman/observational-type-theory> [31]

archived at `swh:1:dir:a55c73294775921cf98631f7da69b9a9a3dabf82`

## 1 Introduction

Since the inception of logics based on (dependent) type theories, *propositional equality*, i.e. the one manipulated in proofs, has been a thorn in the side of users. In recent years, an alternative has been proposed by Altenkirch et al. [5, 6], and developed by Pujet and Tabareau [27, 26, 28]: *observational equality*. It has two main characteristics. First, it is *definitionally proof-irrelevant*: any two proofs of equality are identified in the type theory, drastically simplifying its behaviour. Second, rather than being defined uniformly, observational equality has a specific behaviour at each type. Equality between functions is pointwise equality (*function extensionality*), equality between propositions is logical equivalence (*propositional extensionality*), and equality at quotient types is the relation by which the quotient was taken. Together, these aspects make equality closer to what mathematicians are used to, and allow seamless support for quotient types. Lean’s mathematics library [32], a leading effort in formalized mathematics, relies on such a definitionally irrelevant equality with function extensionality and quotient, although their approach is type-theoretically somewhat ill-behaved compared to observational equality.

Pujet and Tabareau’s work comes with extensive meta-theory, but no implementation. We attack this unexplored aspect with an experimental implementation of  $CC^{obs}$ , based on *normalisation by evaluation* (NbE) [1]. NbE is a modern technique to decide *definitional*



© Matthew Sirman, Meven Lennon-Bertrand, and Neel Krishnaswami;  
licensed under Creative Commons License CC-BY 4.0

30th International Conference on Types for Proofs and Programs (TYPES 2024).

Editors: Rasmus Ejlers Møgelberg and Benno van den Berg; Article No. 5; pp. 5:1–5:22

Leibniz International Proceedings in Informatics



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

*equality*, the equations that the type-checker is able to automatically enforce – in contrast with propositional equality, which require explicit proofs. To decide definitional equality, NbE follows the naïve strategy of computing normal forms for the terms/types under scrutiny. NbE shines, in that it very efficiently computes these normal forms, by instrumenting the evaluation mechanism of the host language.<sup>1</sup>

**Contributions.** In our implementation, we extend standard NbE techniques, as presented by e.g. Abel [1] and in Kovács’ *elaboration-zoo* [19], to an extension of Pujet and Tabareau’s  $CC^{obs}$  [26]. In addition to the constructions already handled by Abel and Kovács, our type theory features a sort of definitionally irrelevant (strict) propositions  $\Omega$  [13], an observational equality valued in that sort, and quotient types by  $\Omega$ -valued relations. We also explore inductive types, as first-class constructs equipped with a Mendler-style recursion [23].

Experimental implementations of  $CC^{obs}$  and of NbE for strict propositions already exist [7, 9], but to the best of our knowledge we are the first to describe one in print. The latter was also theoretically studied [2], but not implemented, and we observe (in Sec. 3.3) that their approach is actually problematic. Our dependently-typed adaptation of Mendler-style induction is also novel, although it would deserve a theoretical investigation we lack.

A last contribution is in some sense a non-contribution: an experience report that, apart from subtleties around strict propositions, NbE mostly *just worked*. This is not our achievement, but we believe it is nonetheless important to stress. The same can also be said of other techniques, such as bidirectional typing and pattern unification, which readily adapted to our setting.

In Section 2 we present the type theory we implement. As our base NbE algorithm is very close to Abel’s [1], we refer the reader to that work for background. In Section 3, we tackle the core of our implementation, NbE and type-checking for  $CC^{obs}$ . Section 4 presents extensions: quotient and inductive types, and a lightweight feature similar to Agda’s holes.

The Haskell code for the implementation is freely available on GitHub [31]. This article is based on the first author’s Part III dissertation [30].

## 2 Background

Our type theory is an extension of  $CC^{obs}$  [26, 27, 28], itself based on Martin-Löf Type Theory (MLTT) [21], the staple dependently-typed theory. In this section, we first quickly sum up the additions made by  $CC^{obs}$  compared to MLTT. Our version of  $CC^{obs}$  is very close to that of Pujet [26], to which we refer for an extensive discussion. We then present the main point where we depart from it: the addition of inductive type as a first-class construct in the language, featuring Mendler-style recursion.

### 2.1 Observational type theory

**Martin-Löf Type Theory.** MLTT is a dependent type theory presented by five mutually defined judgements, characterizing well-formed context  $\vdash \Gamma$ , types  $\Gamma \vdash A$  and terms  $\Gamma \vdash t : A$ , and asserting that two terms (resp. types) are *convertible* or *definitionally equal*  $\Gamma \vdash A \equiv A'$  (resp.  $\Gamma \vdash t \equiv t' : A$ ). During type-checking we need to compare (dependent) types, which can contain terms. Thus, equations between the latter can appear when comparing the former, meaning we have to decide conversion between arbitrary terms.

<sup>1</sup> We somewhat depart from this by implementing defunctionalized NbE [1], where closures are used instead of meta-level functions.

MLTT is a language with binders, represented with names in the text for readability. In the implementation, the front language has names, but internally we use de Bruijn indices for terms and de Bruijn levels for semantic values, as is standard for NbE [1].

We use a single universe  $\mathcal{U}$  as the type of all types, with  $\mathcal{U} : \mathcal{U}$ . This is known to break termination of the system [14], making the type theory undecidable and inconsistent. Yet, as this is orthogonal to our focus, we go with the simple albeit inconsistent approach in our prototype. Apart from this, our MLTT is standard, featuring dependent function ( $\Pi$ ) and pair ( $\Sigma$ ) types with their  $\eta$ -laws, natural numbers with large elimination, and a unit type.

**Proof irrelevance.** The first extension of  $\text{CC}^{\text{obs}}$  compared to MLTT are *proof-irrelevant propositions*, given by a universe  $\Omega$  with the following conversion rule:

$$\frac{\Gamma \vdash P : \Omega \quad \Gamma \vdash t : P \quad \Gamma \vdash u : P}{\Gamma \vdash t \equiv u : P}$$

We use the symbol  $\mathfrak{s}$  for an arbitrary sort,  $\mathcal{U}$  or  $\Omega$ . Propositions include the false and true propositions  $\perp$  and  $\top$ , and existential quantification  $\exists(x :_{\mathfrak{s}} A). B$ . A  $\Pi$  type with a propositional codomain is again a proposition, representing universal quantification if the domain is relevant, or (dependent) implication if it is not.

**Observational equality.** Observational equality is a family of types, representing identifications between two inhabitants of the same type.

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash t : A \quad \Gamma \vdash u : A}{\Gamma \vdash t \sim_A u : \Omega}$$

Equality is proven by reflexivity  $\mathbf{refl}(t) : t \sim_A t$ , and can be used in two different ways: transport  $\mathbf{transp}(t, x \ p. C, u, t', e)$  lets us use the proof  $e : t \sim_A t'$  to turn the proof  $u : C[t/x, \mathbf{refl} \ t/p] : \Omega$  into a proof of  $C[t'/x, e/p]$ ;<sup>2</sup> with cast  $\mathbf{cast}(A, B, e, t)$ , we can use a proof  $e : A \sim_U B$  to construct an inhabitant of  $B$  one of  $A$ . The difference is that the latter applies to *relevant* types, while the former proves a proposition. Thus, only  $\mathbf{cast}$  needs to be endowed with computational content, as all propositions are convertible.

Beyond  $\mathbf{refl}$ , we add constants for symmetry ( $\mathbf{sym}$ ) and transitivity ( $\mathbf{trans}$ ). These are provable using  $\mathbf{transp}$  and  $\mathbf{refl}$ , but some of our computation rules need them, so it is easier and cleaner to add them as primitives. This is a benefit of strict propositions: since there is no computation in the irrelevant layer, we are free to add propositional constants without needing to endow them with computational content.

Contrarily to MLTT, where equality is a type constructor and cast computes on reflexivity, in  $\text{CC}^{\text{obs}}$  both the equality type and cast compute on the types. Yet, we still retain the following conversion, a generalisation of  $\beta$  reduction of  $\mathbf{cast}$  in MLTT – its special case when  $e$  is  $\mathbf{refl}_A$ . This should be seen as an extensionality rule, similar to  $\eta$ -rules.

$$\frac{\Gamma \vdash e : A \sim_{\mathcal{U}} A' \quad \Gamma \vdash t : A \quad \Gamma \vdash A \equiv A' : \mathcal{U}}{\Gamma \vdash \mathbf{cast}(A, A', e, t) \equiv t : A'}$$

<sup>2</sup> Thanks to proof irrelevance, our version of transport where the motive  $C$  depends on the proof of equality is inter-derivable with one without, so we include the stronger primitive for ease of use, while Pujet has the weaker primitive to simplify meta-theory.

**Quotient types.** Observational equality gives us a synthetic language to talk about *setoids* [17, 4], types equipped with an equivalence relation. We can exploit this to integrate quotient types  $A/R$ : observational equality at such a quotient type simply boils down to the equivalence relation  $R$ . Quotients come with a projection map  $\pi : A \rightarrow A/R$ , and their elimination captures the idea that a function out of a quotient must respect the relation, i.e. map related inputs to equal outputs.

## 2.2 Inductive types

We extend  $\text{CC}^{\text{obs}}$  with a form of first-class indexed inductive types, and explore their interaction with observational equality. Our approach is exploratory, and we do not pretend to have a fully fleshed-out design, especially since we do not carry any meta-theoretic study.

**First-class indexed inductive types.** Since our system does not distinguish between local and global context, our inductive types are *first-class* [11, 8]: we can bind them to variables, and generally treat them as any other value. They take the following form:

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \{\Gamma, F : A \rightarrow \mathcal{U} \vdash B_i : \mathcal{U}\}_i \quad \{\Gamma, F : A \rightarrow \mathcal{U}, x_i : B_i[F] \vdash a_i : A\}_i}{\Gamma \vdash \mu F : A \rightarrow \mathcal{U}. \overrightarrow{[C_i : (x_i : B_i) \rightarrow F a_i]} : A \rightarrow \mathcal{U}}$$

The variable  $F$  is bound by  $\mu$ .  $A$  represents the type of indices. We have a finite list of constructors, each with a name  $C_i$ , an argument of type  $B_i$ , and an index  $a_i$ , which might depend on the argument. The  $F$  at the end of each constructor is mere syntax indicating the type being constructed; it is not a free variable. In what follows, we use the following shortcut:  $\mu F \triangleq \mu F : A \rightarrow \mathcal{U}. \overrightarrow{[C_i : (x_i : B_i) \rightarrow F a_i]}$ , i.e.  $\mu F$  stands for a generic inductive.

We restrict inductive types to have exactly one index, and constructors to have exactly one argument. This loses no expressivity since we can pack arguments or indices together with  $\Sigma$  types, and simplifies the implementation. For further simplification, we also do not implement a positivity checker, so the typing rule allows non-strictly positive inductive types. As for universes, this is mainly orthogonal to our main concerns.

Since inductive types are first class, parameters can be handled by  $\lambda$ -abstraction, as illustrated by the standard example of vectors:

$$\text{Vec} \triangleq \lambda A : \mathcal{U}. \mu F : \mathbb{N} \rightarrow \mathcal{U}. [\text{Nil} : \mathbb{1} \rightarrow F 0 ; \text{Cons} : (x : \Sigma(n : \mathbb{N}). A \times F n) \rightarrow F(S(\text{fst } x))]$$

**Constructors.** Values of inductive types are created by the constructors. We use *fording* [22, p. 65]: we allow all constructors to build a value of type  $(\mu F) a$  for *any* index  $a$  if they provide a proof that  $a_i \sim_A a$ , where  $a_i$  is the index computed from the argument of the constructor. In the MLTT presentation of inductive types, this constraint is instead enforced definitionally.

$$\frac{\Gamma \vdash t : B_i[\mu F/F] \quad \Gamma \vdash e : a_i[\mu F/F, t/x_i] \sim_A a}{\Gamma \vdash C_i(t, e) : (\mu F : A \rightarrow \mathcal{U}. \overrightarrow{[C_i : (x_i : B_i) \rightarrow F a_i]}) a}$$

**Pattern-matching.** Elimination of inductive types is single-level, total pattern matching:

$$\frac{\Gamma \vdash t : (\mu F : A \rightarrow \mathcal{U}. \overrightarrow{[C_i : (x_i : B_i) \rightarrow F a_i]}) a \quad \Gamma, x : (\mu F) a \vdash C : \mathfrak{s} \quad \{\Gamma, x_i : B_i[\mu F/F], e_i : a_i[\mu F, x_i] \sim_A a \vdash t_i : C[(C_i(x_i, e_i))/x]\}_i}{\Gamma \vdash \text{match } t \text{ as } x \text{ return } C \text{ with } \{C_i(x_i, e_i) \rightarrow t_i\} : C[t/x]}$$

This rule might appear strange, as the motive  $C$  does *not* abstract over the index. Suppose  $Vec$  as above, and we match on  $v : Vec\ 0$ , so  $x : Vec\ 0 \vdash C : \mathfrak{s}$ . In the  $Cons$  branch, we substitute  $Cons$  into  $C$ . However, the index of  $Cons$  is  $S\ n$ , this looks ill-typed! But thanks to fording we *can* have  $Cons(x, e) : Vec\ 0$ , given a proof of  $S\ m \sim 0$ , which we do have in this branch. Moreover, since  $S\ m \sim 0 \equiv \perp$ , we can use this to witness that this branch is in fact unreachable. The  $\beta$  rule is straightforward, but for handling of the forced equality.

$$\overline{\Gamma \vdash \text{match } C_i(t, e) \text{ as } x \text{ return } C \text{ with } \{C_i(x_i, e_i) \rightarrow t_i\} \equiv t_i[t/x_i, e/e_i] : C[C_i(t, e)/x]}$$

**Observational equality for inductive types.** Observational equality between inductive types does *not* equate inductive types structurally: types with propositionally equal but definitionally different index and constructor types are not deemed equal.

$$\overline{\Gamma \vdash (\mu F)\ a \sim_{\mathcal{U}} (\mu F)\ a' \equiv a \sim_A a' : \Omega}$$

The goal is twofold. First, this simplifies the implementation, as it means we avoid having to compare telescopes of parameters with numerous casts. Second, a purely structural equality of inductive types would equate all “boolean” inductive types (those with two argumentless constructors), a severe case of boolean blindness [16].

When the two definitions are not convertible, observational equality is simply stuck,<sup>3</sup> i.e. it does not reduce further. This covers the case of definitely different inductive types (where we could be more eager and reduce to  $\perp$ ), but also of parameterized inductive types. Indeed, for two different variables  $A$  and  $A'$ ,  $Vec\ A\ n \sim_{\mathcal{U}} Vec\ A'\ n$  is stuck, and will compute further only if  $A$  and  $A'$  are substituted by convertible types. This equality therefore does not imply  $A \sim_{\mathcal{U}} A'$ . We thus do not implement a conversion rule like the following:

$$\text{cast}(Vec\ A\ 1, Vec\ A'\ 1, \dots, Cons(0, a, Nil(\dots))) \equiv Cons(0, \text{cast}(A, A', \dots, a), Nil(\dots))$$

Deriving these definitional equalities roughly amounts to deriving a general *map* operation for inductive types, a non-trivial enterprise which we did not attempt. Since the design of our prototype, Pujet and Tabareau [29] have explored the question further, and proposed a solution, which we did not try to reproduce.

Equality of general inductives behaves like that of natural numbers: when comparing equal constructors, the proposition steps to equality of the contents, and when they are different, it steps to  $\perp$ . These respectively reflect injectivity and no-confusion of constructors.

$$\overline{\Gamma \vdash C_i(t, e) \sim_{(\mu F)a} C_i(u, e') \equiv t \sim_{B_i[\mu F]} u : \Omega} \quad \overline{\Gamma \vdash C_i(t, e) \sim_{(\mu F)a} C_j(u, e') \equiv \perp : \Omega} \quad \frac{C_i \neq C_j}{\Gamma \vdash C_i(t, e) \sim_{(\mu F)a} C_j(u, e') \equiv \perp : \Omega}$$

For casts on constructors, as explained above we do not need to – and indeed cannot – propagate them deep in the structure. Instead, we merely need to handle indices, which amounts to composing equality proofs. Note the use of the **trans** primitive.

$$\overline{\Gamma \vdash \text{cast}((\mu F)\ a, (\mu F)\ a', e, C_i(t, e')) \equiv C_i(t, \text{trans}_{a_i[\mu F/F, t/x_i], a, a'}\ e'\ e) : (\mu F)\ a'}$$

<sup>3</sup> As noted by Pujet [26, 5.2.2], there is a lot of room in how much definitional equalities are imposed on observational equality, the sole constraint – coming from canonicity – being that casts between convertible closed types must compute.

**Mendler induction.** Pattern matching as eliminator for inductive types does not let us handle their recursive structure. That is, we have no notion of *induction*, as we can look only one level at a time. To correct this, we introduce **fix** in a style extending Mendler recursion [23] to dependent induction.

To express it, we first need an operation **lift** which applies a functor  $F$  to a type  $X$ .

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \{\Gamma, F : A \rightarrow \mathcal{U} \vdash B_i : \mathcal{U}\}_i \quad \{\Gamma, F : A \rightarrow \mathcal{U}, x_i : B_i[F] \vdash a_i : A\}_i}{\Gamma \vdash \mathbf{lift}[\mu F] : A \rightarrow \mathcal{U}. [\overline{C_i : (x_i : B_i) \rightarrow F a_i}] : (A \rightarrow \mathcal{U}) \rightarrow \mathcal{U}}$$

This operation is defined by the following equation:

$$\Gamma \vdash \mathbf{lift}[\mu F] B \equiv \mu F : A \rightarrow \mathcal{U}. [\overline{C_i : (x_i : B_i[B/F]) \rightarrow F a_i[B/F]}] : \mathcal{U}$$

In essence,  $\mathbf{lift}[\mu F] B$  is an inductive type *which is mute in the variable  $F$* , that is, non-recursive. It represents adding a single “layer” of  $F$ -structure over the family  $B$ .

Using this primitive, we can express the type of Mendler-style induction. As above, we abbreviate the inductive type to  $\mu F$ . The type  $C$  is the one we want to inhabit by induction.

$$\frac{\Gamma, G : A \rightarrow \mathcal{U}, p : A, x : G p \vdash C : \mathfrak{s} \quad \Gamma, G : A \rightarrow \mathcal{U}, f : \Pi(p : A)(x : G p). C[G, p, x], p : A, x : \mathbf{lift}[\mu F] G p \vdash t : C[\mathbf{lift}[\mu F] G, p, x]}{\Gamma \vdash \mathbf{fix} [\mu F \text{ as } G] f p x : C = t : \Pi(p : A). \Pi(x : (\mu F) p). C[\mu F, p, x]}$$

To type the body of the fixed point, Mendler induction operates by introducing a generic type family,  $G$  and an inhabitant  $x$  of  $\mathbf{lift}[\mu F] G p$ , and demands that we construct an inhabitant of  $C$  at  $x$ , while having access to “recursive calls” only on inhabitants of  $G$  via the function  $f$ . Intuitively, we can pattern-match on  $x$ , recovering values of  $G$  corresponding to “subterms”, that can be used for recursive calls. Since  $G$  is abstract, this is the only way to call  $f$ , and so the fixed point we obtain is structurally decreasing.

The **fix** operation computes when applied to a constructor, as per the following rule. The argument *must* be a constructor in order to prevent infinite unfolding.

$$\frac{\mathbf{fix}_f \triangleq \mathbf{fix} [\mu F \text{ as } G] f p x : C = t}{\Gamma \vdash \mathbf{fix}_f u (C_i (v, e)) \equiv t[\mu F/G, \mathbf{fix}_f/f, u/p, (C_i (v, e))/x]}$$

**Views and paramorphisms.** Mendler induction ensures **fix** is well-founded by restricting recursive appeals to the induction hypothesis. However, in this process, we lose information: we only have access to recursive calls, but not to the current value of the inductive type. In categorical parlance, fixed-points and pattern matching implement *catamorphisms*: generalised folds over tree-like structures.

What we want instead are *paramorphisms*, which provide primitive recursion by giving access to the current value. We thus introduce an extra function  $\iota : \Pi(p : A). G p \rightarrow (\mu F) p$  to *view* the opaque  $G$  as the inductive type it represents. Once the knot of the fixed point is tied and the variable  $G$  is substituted for the inductive type, this becomes the identity.

To type the fixed point’s body we need to lift  $\iota : \Pi(p : A). G p \rightarrow (\mu F) p$  into  $\iota' : \Pi(p : A). \mathbf{lift}[\mu F] G p \rightarrow (\mu F) p$ . Hence, we need again the action of the functor  $\mu F$  on values, i.e. the associated *map*. As explained above, we do not provide infrastructure to derive this operation. We instead allow an inductive definition to come with a user-provided functor action.<sup>4</sup> Note that in the premise we use an inductive type *without* a functor.

<sup>4</sup> We do not check this indeed is the functor action, since generating the equalities a correct *map* operation satisfies is basically the same as deriving that operation itself.

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \{\Gamma, F : A \rightarrow \mathcal{U} \vdash B_i : \mathcal{U}\}_i \quad \{\Gamma, F : A \rightarrow \mathcal{U}, x_i : B_i[F] \vdash a_i : A\}_i}{\Gamma, X : A \rightarrow \mathcal{U}, Y : A \rightarrow \mathcal{U}, f : \Pi(p : A). X p \rightarrow Y p, p : A, x : \mathbf{lift}[\mu F] X p \vdash t : \mathbf{lift}[\mu F] Y p} \xrightarrow{\quad} \Gamma \vdash \mu F : A \rightarrow \mathcal{U}. [\overline{C_i : (x_i : B_i) \rightarrow F a_i}] \mathbf{functor} X Y f p x = t : A \rightarrow \mathcal{U}$$

To be able to use this definition, we introduce the term **fmap** for projecting the functorial action from an inductive type and **in**, witnessing the isomorphism between  $\mu F$  and  $\mathbf{lift}[\mu F](\mu F)$ , which is the identity function on constructors:  $\mathbf{in}(C_i(t, e)) \equiv C_i(t, e)$ .

$$\frac{\Gamma \vdash \mu F : \mathcal{U}}{\Gamma \vdash \mathbf{fmap}[\mu F] : \Pi(X : A \rightarrow \mathcal{U}). \Pi(Y : A \rightarrow \mathcal{U}). (\Pi(p : A). X p \rightarrow Y p) \rightarrow \Pi(p : A). \mathbf{lift}[\mu F] X p \rightarrow \mathbf{lift}[\mu F] Y p} \quad \frac{\Gamma \vdash t : \mathbf{lift}[\mu F] F a}{\Gamma \vdash \mathbf{in} t : (\mu F) a}$$

With this infrastructure, we extend the typing rule for fixed-points.

$$\frac{\begin{array}{l} \iota' \triangleq \lambda p. \lambda x. \mathbf{in}(\mathbf{fmap}[\mu F] G \mu F \iota p x) \quad \text{id} \triangleq \lambda p. \lambda x. x \\ \Gamma, G : A \rightarrow \mathcal{U}, \iota : \Pi(p : A). G p \rightarrow (\mu F) p, p : A, x : G p \vdash C : \mathfrak{s} \\ \Gamma, G : A \rightarrow \mathcal{U}, \iota : \Pi(p : A). G p \rightarrow (\mu F) p, f : \Pi(p : A). \Pi(x : G p). C[G, \iota, p, x], \\ \quad p : A, x : F[G] p \vdash t : C[F[G], \iota', p, x] \end{array}}{\Gamma \vdash \mathbf{fix}[\mu F \text{ as } G \text{ view } \iota] f p x : C = t : \Pi(p : A). \Pi(x : (\mu F) p). C[\mu F, \text{id}, p, x]}$$

The variable  $\iota$  is now accessible in both the motive  $C$  and the body  $t$  of the fixed-point, facilitating primitive recursion. At the top level,  $\iota$  is substituted by the identity function. By functoriality, this means  $\iota'$  is  $\mathbf{in} \circ \text{id}$ , which also behaves as the identity.

In summary, if the user provides a *map* operation, we leverage that to upgrade our Mendler-style catamorphisms to paramorphisms. Of course, in a full-fledged implementation we would derive that operation automatically, and simply give access to the paramorphism.

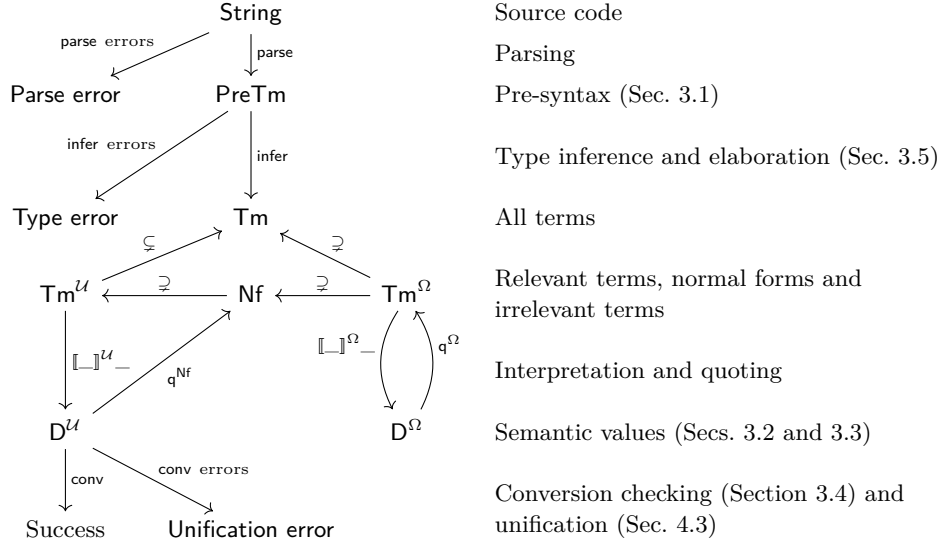
### 3 Core Implementation

Let us start with the core features of  $\text{CC}^{\text{obs}}$ : strict propositions and inductive equality. The implementation is written in Haskell, and makes extensive use of structural data types and declarative style to keep the code as close as possible to the theory. Figure 1 gives an overview of the implementation's structure. We use  $\rightarrow$  for the meta-level (i.e. Haskell's) function type, and a sans font for its types and functions.

#### 3.1 Syntax

**Syntax.** We present  $\text{CC}^{\text{obs}}$  in Russell style [21, 25], so we have a common grammar for both terms and types, given in Figure 2. This is roughly the one given in Pujet [26], with some additions: let bindings and type annotations, which are important with bidirectional typing, and, as explained in Section 2.1, constants for symmetry and transitivity.

We use a unit type  $\mathbb{1}$ , with the term-directed  $\eta$ -conversions  $t \equiv !$  and  $! \equiv t$ . These break transitivity of a term-directed algorithmic conversion: in the context  $x, y : \mathbb{1}$ , we have  $x \equiv ! \equiv y$  but  $x \not\equiv y$ . This is a trade-off: since we impose inductive types to have exactly one index and constructors exactly one argument, this unit type is handy to pad these positions with definitionally irrelevant content. In a more mature implementation, this would not be necessary, and we could drop our ill-behaved  $\mathbb{1}$ , or adopt the more complex – and satisfactory – solution proposed by Kovács [20].



■ **Figure 1** A high-level overview of the interaction of the components in the type-checker.

$\mathfrak{s}$	$::= \mathcal{U} \mid \Omega$	Universe sorts
$A, B, C, t, u, e$	$::= x$	Variable
	$\mid \mathfrak{s}$	Universe
	$\mid \lambda x_{\mathfrak{s}}. t \mid t @^{\mathfrak{s}} u \mid \Pi(x :_{\mathfrak{s}} A). B$	Dependent functions
	$\mid 0 \mid St \mid \mathbf{rec}[z.C](t, 0 \rightarrow t_0; (Sx) y \rightarrow t_S) \mid \mathbb{N}$	Natural numbers
	$\mid (t, u)_{\mathfrak{s}} \mid \mathbf{fst} t \mid \mathbf{snd} t \mid \Sigma x : A. B \mid \exists x : A. B$	Dependent pairs
	$\mid \mathbf{abort}_A t \mid \perp \mid * \mid \top \mid \mathbb{I} \mid !$	False, true and unit types
	$\mid \Box A \mid \Diamond t \mid \Box\text{-elim } t$	Box types
	$\mid t \sim_A u \mid \mathbf{refl} t \mid \mathbf{sym}_{t,u} e \mid \mathbf{trans}_{t,u,v} e e'$	Observational equality
	$\mid \mathbf{transp}(t, x y. C, u, t', e) \mid \mathbf{cast}(A, B, e, t)$	Transport and casting
	$\mid \mathbf{let} x :_{\mathfrak{s}} A = t \mathbf{in} u$	Let binding
	$\mid (t : A)$	Type annotation

■ **Figure 2** Basic syntax for  $\text{CC}^{\text{obs}}$ .

Box types  $\Box A$  turn a type into a proposition – what is alternatively called squashing, truncation, or *inhabited* – see Pujet [26] for details.

Application  $t @^{\mathfrak{s}} u$  is tagged with the sort  $\mathfrak{s}$  of the argument  $u$ , and  $\lambda$ -terms and pairs with their domain sort. This is necessary for evaluation, but is always inferred during type-checking; users need not give these annotations in the source syntax.

This grammar inductively defines the type  $\text{PreTm}$  of *pre-terms*: syntactically well-formed, but untyped. We define another type  $\text{Tm}$  of well-typed terms. Every well-typed term has a (unique) *sort* –  $\mathcal{U}$  or  $\Omega$  –, the type of its type. We let  $\text{Tm}^{\mathcal{U}}$  and  $\text{Tm}^{\Omega}$  be the set of terms with sort  $\mathcal{U}$  and  $\Omega$  respectively. In the code we have separate types for  $\text{PreTm}$  and  $\text{Tm}$ , the former using named variables and the latter de Bruijn indices. We cannot enforce in Haskell that  $\text{Tm}$  contains only well-typed terms, but maintain this as an invariant.

In what follows, we do not always cover all cases, focusing on the most interesting or illustrative ones. The code can be consulted for the complete picture.



**Normal forms.** Normals and neutrals are defined mutually as predicates on  $\mathsf{Tm}^{\mathcal{U}}$ . In contrast to Pujet [27], they are deep, so subterms are also required to be normal.

$$\begin{aligned}
\mathsf{Nf} \ni v, w, V, W &::= n \mid \mathfrak{s} \mid \lambda x. v \mid \Pi(x : \mathfrak{s} V). W \mid 0 \mid S v \mid \mathbb{N} \mid (v, v)_{\mathcal{U}} \mid \Sigma(x : V). W \\
&\mid (v, v)_{\Omega} \mid \exists(x : V). W \mid \perp \mid \top \mid ! \mid \mathbb{1} \\
\mathsf{Ne} \ni n, N &::= x_i \mid n @^{\mathcal{U}} v \mid n @^{\Omega} t \mid \mathbf{rec}[z.V](n, 0 \rightarrow v; (S x) y \rightarrow w) \mid \mathbf{abort}_V t \\
&\mid v \sim_n w \mid n \sim_{\mathbb{N}} v \mid v \sim_{\mathbb{N}} n \mid n \sim_{\mathcal{U}} v \mid v \sim_{\mathcal{U}} n \\
&\mid \mathbf{cast}(\mathbb{N}, \mathbb{N}, e, n) \mid \mathbf{cast}(N, V, e, v) \mid \mathbf{cast}(V, N, e, v)
\end{aligned}$$

We only need to characterize normal forms in  $\mathsf{Tm}^{\mathcal{U}}$ , i.e. relevant terms. Indeed, since proof-irrelevant terms have no notion of evaluation, they *all* are “normal forms”. This means that normal forms are only unique up to  $\eta$ -equality and proof irrelevance. Luckily, thanks to typing constraints, irrelevant subterms of a relevant normal form can only appear as arguments to **abort**, to **cast**, or an application tagged with  $\Omega$ , which lets us decide which subterms (not) to compare purely syntactically, without typing information.

Normal forms, as standard, correspond to constructors – observational equality and cast are viewed as destructors, and so are not normal forms. Neutral forms are somewhat more involved. The observational equality type can be blocked in *three* places – the type, or either of the terms – although there are no neutral forms when the equality is at a  $\Pi$  or  $\Sigma$  type, as such equalities always reduce. Similarly, casts can also block in three positions: either of the types, or the term being cast. Again, casts between universes,  $\Pi$  or  $\Sigma$  types always reduce, and so cannot be blocked by the argument. To avoid quadratic blow-up in the presentation, some cases overlap, for example  $n \sim_{\mathbb{N}} v$  and  $v \sim_{\mathbb{N}} n$  both cover  $x \sim_{\mathbb{N}} y$ .

Pujet [26, 5.2.2] discusses the definitional equalities observational equality should satisfy, for instance whether  $\Pi(x : A). B \sim_{\mathcal{U}} \Sigma(x : A'). B'$  should evaluate to  $\perp$  or be stuck. He remarks we can replace the reduction of  $\Sigma(x : A). B \sim_{\mathcal{U}} \Sigma(x : A'). B'$  by two constants corresponding to the two projections out of the would-be reduct:

$$\mathbf{eqfst} : (\Sigma x : A. B) \sim_{\mathcal{U}} (\Sigma x : A'. B') \rightarrow A \sim_{\mathcal{U}} A' \quad \mathbf{eqsnd} : \dots$$

We chose to make conversion compute as much as possible as a minimal form of automation, although it is not entirely clear whether this makes using the proof assistant really easier.

### 3.2 Evaluation in the relevant layer

The overall picture for NbE is similar to the standard case [1]: we give a semantic interpretation into a domain of values, and a quoting function to reconstruct normal forms. The main difficulty is to account for complex reduction rules for propositions and casts, and proof-irrelevant propositions. We defer the latter to Section 3.3. Note that we implement untyped rather than type-directed NbE. Although the latter would have avoided issues around the unit type (see Sec. 3.1), we chose it to avoid an extra layer of complexity, and check that even a complex type theory like  $\mathsf{CC}^{\text{obs}}$  can still be implemented this way.

**Semantic domain.** First, we construct the various domains data-structures in Figure 3. The domains  $\mathsf{D}^{\mathcal{U}}$  and  $\mathsf{D}^{\text{ne}}$  represent respectively normal and neutral forms. The domain  $\mathsf{D}^{\Omega}$  represents propositional values and is explained in Section 3.3.  $\mathsf{D}$  is the union of  $\mathsf{D}^{\mathcal{U}}$  and  $\mathsf{D}^{\Omega}$ , and we often omit the injections  $\mathsf{V}$  and  $\mathsf{P}$ .

These are defined mutually with closures  $\mathsf{Clos}_k^{\mathfrak{s}}$  indexed by their return sort  $\mathfrak{s}$  and their arity  $k$ , as not all of them await a single argument. This is a landmark of defunctionalised NbE, which replaces semantic functions by such closures, making the various domains first-order strictly positive datatypes. Abel [1] needs a single kind of closure, consisting of an environment and a term, representing a meta-level function of type  $\mathsf{Tm} \rightarrow \mathsf{Tm}$ . Here, we need to defunctionalise various evaluation functions, and thus have multiple forms.

## 5:10 Implementing a Type Theory with Observational Equality, Using NbE

$D^{\mathcal{U}}$	$\ni$	$a, b, A, B$	$::=$	$U \text{ s} \mid \text{Lam } s \text{ } \mathcal{F}_1 \mid \text{Pi } s \text{ } A \text{ } \mathcal{B}_1 \mid Z \mid S \text{ } a \mid \text{Nat} \mid \uparrow e$ $\mid \text{Exists } A \text{ } \mathcal{B}_1 \mid \text{Empty} \mid \text{Unit}$
$D^{\text{ne}}$	$\ni$	$e$	$::=$	$\text{Var}_l \mid \text{App}_s \text{ } e \text{ } d \mid \text{Rec } \mathcal{A}_1 \text{ } e \text{ } a \text{ } \mathcal{F}_2 \mid \text{Eq } a \text{ } A \text{ } a' \mid \text{Cast } A \text{ } B \text{ } p \text{ } a$
$D^{\Omega}$	$\ni$	$p, q, P, Q$	$::=$	$\dots$ (see Sec. 3.3)
$D$	$\ni$	$d, f, g, D$	$::=$	$V \text{ } a \mid P \text{ } p$
$\text{Env}$	$\ni$	$\rho$	$::=$	$() \mid (\rho, d)$
$\text{Clos}_k^s$	$\ni$	$\mathcal{C}_k$	$::=$	$(\lambda t) \rho$ : $\text{Clos}_k^s$ $\mid \text{Lift } d$ : $\text{Clos}_k^s$ $\mid \text{EqFun } s \text{ } f \text{ } \mathcal{C}_1 \text{ } g$ : $\text{Clos}_1^s$ $\mid \text{EqPi } s \text{ } D \text{ } D' \text{ } \mathcal{C}_1 \text{ } \mathcal{C}'_1$ : $\text{Clos}_1^s$ $\mid \text{EqPi}' \text{ } s \text{ } D \text{ } D' \text{ } \mathcal{C}_1 \text{ } \mathcal{C}'_1 \text{ } p$ : $\text{Clos}_1^s$ $\mid \text{CastPi } s \text{ } D \text{ } D' \text{ } \mathcal{C}_1 \text{ } \mathcal{C}'_1 \text{ } p \text{ } f$ : $\text{Clos}_1^s$
$\text{Clos}_k^{\mathcal{U}}$	$\ni$	$\mathcal{A}_k, \mathcal{B}_k, \mathcal{F}_k$		
$\text{Clos}_k^{\Omega}$	$\ni$	$\mathcal{P}_k, \mathcal{Q}_k$		

■ **Figure 3** Semantic domains, environments and closures.

Neutral terms for observational equality and casting are simplified: we do not keep track of which of the three arguments to `Eq` and `Cast` is blocking. This is a trade-off between a precise semantic domain and a simpler interpretation function. As the goal here is to implement the system, not prove its correctness, we choose the simpler representation.

Environments contain values from *both* sorts. An environment interprets a context, where each variable has a sort. It is a program invariant that the domain sort in each position in the environment matches the typing context.

$\llbracket x_0 \rrbracket^{\mathcal{U}}(\rho, a) = a$	$\llbracket \Pi(x :_s A). B \rrbracket^{\mathcal{U}} \rho = \text{Pi } s \text{ } (\llbracket A \rrbracket^{\mathcal{U}} \rho) (\lambda B) \rho$
$\llbracket x_{i+1} \rrbracket^{\mathcal{U}}(\rho, d) = \llbracket x_i \rrbracket^{\mathcal{U}} \rho$	$\llbracket \lambda x_s. t \rrbracket^{\mathcal{U}} \rho = \text{Lam } s \text{ } (\lambda t) \rho$
$\llbracket s \rrbracket^{\mathcal{U}} \rho = U \text{ s}$	$\llbracket t \sim_A u \rrbracket^{\mathcal{U}} \rho = \text{eq}(\llbracket t \rrbracket^{\mathcal{U}} \rho, \llbracket A \rrbracket^{\mathcal{U}} \rho, \llbracket u \rrbracket^{\mathcal{U}} \rho)$
$\llbracket 0 \rrbracket^{\mathcal{U}} \rho = Z$	$\llbracket \text{cast}(A, B, e, t) \rrbracket^{\mathcal{U}} \rho = \underline{\text{cast}}(\llbracket A \rrbracket^{\mathcal{U}} \rho, \llbracket B \rrbracket^{\mathcal{U}} \rho, \llbracket e \rrbracket^{\Omega} \rho, \llbracket t \rrbracket^{\mathcal{U}} \rho)$
$\llbracket S \text{ } t \rrbracket^{\mathcal{U}} \rho = S \text{ } (\llbracket t \rrbracket^{\mathcal{U}} \rho)$	$\llbracket \text{let } x :_s A = t \text{ in } u \rrbracket^{\mathcal{U}} \rho = \llbracket u \rrbracket^{\mathcal{U}}(\rho, \llbracket t \rrbracket^{\mathcal{U}} \rho)$
$\llbracket N \rrbracket^{\mathcal{U}} \rho = \text{Nat}$	$\llbracket (t : A) \rrbracket^{\mathcal{U}} \rho = \llbracket t \rrbracket^{\mathcal{U}} \rho$
$\llbracket t \text{ } u^s \rrbracket^{\mathcal{U}} \rho = (\llbracket t \rrbracket^{\mathcal{U}} \rho) \cdot_s (\llbracket u \rrbracket^s \rho)$	
$\llbracket \text{rec}[z.C](t, 0 \rightarrow t_0; (S \text{ } x) \text{ } y \rightarrow t_s) \rrbracket^{\mathcal{U}} \rho = \underline{\text{rec}}((\lambda C) \rho, \llbracket t \rrbracket^{\mathcal{U}} \rho, \llbracket t_0 \rrbracket^{\mathcal{U}} \rho, (\lambda t_s) \rho)$	

■ **Figure 4** Semantic interpretation for relevant terms into the semantic domain  $D^{\mathcal{U}}$ .

**Interpretation.** Inputs to relevant interpretation are well-typed terms of sort  $\mathcal{U}$ ; that is, the set  $\text{Tm}^{\mathcal{U}}$ . Therefore, we construct the function

$$\llbracket \_ \rrbracket^{\mathcal{U}} : \text{Tm}^{\mathcal{U}} \rightarrow \text{Env} \rightarrow D^{\mathcal{U}}$$

The interpretation is given in Figure 4. The underlined functions and  $\cdot_s$  are semantic counterpart to eliminators, and are defined mutually with evaluation, see below.

We also need a function for applying a closure to  $k$  values of either sort:

$$\text{app}^{\mathcal{U}} : \text{Clos}_k^{\mathcal{U}} \rightarrow \underbrace{D \rightarrow \dots \rightarrow D}_k \rightarrow D^{\mathcal{U}}$$

Applying to argument of the correct sort is a code invariant but not statically type-checked.<sup>5</sup> We use the shorthand notation  $\mathcal{C}[d_1, \dots, d_k]$  for  $\text{app}^{\mathcal{U}} \mathcal{C} \text{ } d_1 \text{ } \dots \text{ } d_k$ .

<sup>5</sup> This could be enforced with closures indexed a type-level list of sorts.

As closures are used to defunctionalise specific operations, we define those alongside the corresponding case for `app`. To begin with, we have the primary cases of a continuation,  $(\lambda t)\rho$ , corresponding to substitution, and `Lift`, for a constant closure ignoring its argument.

$$\text{app}^{\mathcal{U}} (\lambda t)\rho \ d_1 \ \dots \ d_k = \llbracket t \rrbracket^{\mathcal{U}}(\rho, d_1, \dots, d_k) \quad \text{app}^{\mathcal{U}} (\text{Lift } a) \ d_1 \ \dots \ d_k = a$$

Semantic application  $\_ \cdot_s \_ : \mathcal{D}^{\mathcal{U}} \rightarrow \mathcal{D}^s \rightarrow \mathcal{D}^{\mathcal{U}}$  directly uses generic closure application.

$$(\text{Lam } s \ \mathcal{F}) \cdot_s d = \mathcal{F}[d] \quad (\uparrow e) \cdot_s d = \uparrow(\text{App}_s \ e \ d)$$

$$\begin{aligned} \underline{\text{eq}}(f, \text{Pi } s \ A \ \mathcal{B}, g) &= \text{Pi } A \ (\text{EqFun } s \ f \ \mathcal{B} \ g) \\ \underline{\text{eq}}(A, \text{U } \Omega, B) &= \text{Exists } (\text{Pi } A \ (\text{Lift } B)) \ (\text{Lift } (\text{Pi } B \ (\text{Lift } A))) \\ \underline{\text{eq}}(\text{Nat}, \text{U } \mathcal{U}, \text{Nat}) &= \underline{\text{eq}}(\text{U } s, \text{U } \mathcal{U}, \text{U } s) = \text{Unit} \\ \underline{\text{eq}}(A, \text{U } \mathcal{U}, B) &= \text{Empty} \quad \text{when } A \text{ and } B \text{ have different head constructors} \\ \underline{\text{eq}}(\text{Pi } s \ A \ \mathcal{B}, \text{U } \mathcal{U}, \text{Pi } s \ A' \ \mathcal{B}') &= \text{Exists } (\underline{\text{eq}}(A', \text{U } s, A)) \ (\text{EqPi } s \ A \ A' \ \mathcal{B} \ \mathcal{B}') \\ \underline{\text{eq}}(\text{Z}, \text{Nat}, \text{Z}) &= \text{Unit} \\ \underline{\text{eq}}(\text{S } a, \text{Nat}, \text{S } b) &= \underline{\text{eq}}(a, \text{Nat}, b) \\ \underline{\text{eq}}(\text{S } a, \text{Nat}, \text{Z}) &= \underline{\text{eq}}(\text{Z}, \text{Nat}, \text{S } a) = \text{Empty} \\ \underline{\text{eq}}(a, A, a') &= \uparrow(\text{Eq } a \ A \ a') \quad \text{otherwise (one of } A, a \text{ or } a' \text{ is neutral)} \\ \text{app}^{\mathcal{U}} (\text{EqFun } s \ f \ \mathcal{B} \ g) \ d &= \underline{\text{eq}}(f \cdot_s d, \mathcal{B}[d], g \cdot_s d) \\ \text{app}^{\mathcal{U}} (\text{EqPi } s \ A \ A' \ \mathcal{B} \ \mathcal{B}') \ p &= \text{Pi } s \ A' \ (\text{EqPi}' s \ A \ A' \ \mathcal{B} \ \mathcal{B}' \ p) \\ \text{app}^{\mathcal{U}} (\text{EqPi}' \ \mathcal{U} \ A \ A' \ \mathcal{B} \ \mathcal{B}' \ p) \ a' &= \underline{\text{eq}}(\mathcal{B}[a], \text{U } \mathcal{U}, \mathcal{B}'[a']) \quad \text{where } a \triangleq \underline{\text{cast}}(A', A, p, a') \\ \text{app}^{\mathcal{U}} (\text{EqPi}' \ \Omega \ A \ A' \ \mathcal{B} \ \mathcal{B}' \ p) \ q' &= \underline{\text{eq}}(\mathcal{B}[q], \text{U } \mathcal{U}, \mathcal{B}'[q']) \quad \text{where } q \triangleq \text{PCast } \phi(A') \ \phi(A) \ p \ q' \end{aligned}$$

■ **Figure 5** Semantic observational equality function, and the associated closures.

**Semantic observational equality.** To complete the semantic interpretation, we need to define `eq` and `cast`, the semantic counterpart to the observational equality type and to `cast`, which implement their reduction behaviour. The former is given in Figure 5.

Semantic equality straightforwardly implements the computational behaviour of observational equality. Note the use of the *two* defunctionalising closures `EqPi` and `EqPi'`: `EqPi` forwards the equality proof into a second closure `EqPi'` which performs the computation. This is necessary because we have two positions requiring arity-one closures, so they cannot be combined. When the  $\Pi$  types have an irrelevant domain, we need a propositional witness of type  $A$ , so we use `PCast` and the *freeze* function  $\phi : \mathcal{D} \hookrightarrow \mathcal{D}^{\Omega}$  for embedding relevant values into the irrelevant domain, both of which will be introduced in Section 3.3.

**Semantic cast.** The final component to complete evaluation is the semantic `cast` function, given in Figure 6, again straightforwardly implementing reduction rules for casts. Note the proof manipulation implemented by propositional application `PApp` and projections `PFst` and `PSnd`, and again the embedding  $\phi : \mathcal{D} \hookrightarrow \mathcal{D}^{\Omega}$ .

$$\begin{aligned}
 \text{cast}(\text{Nat}, \text{Nat}, p, Z) &= Z \\
 \text{cast}(\text{Nat}, \text{Nat}, p, S\ a) &= S\ (\text{cast}(\text{Nat}, \text{Nat}, p, a)) \\
 \text{cast}(\text{U}\ s, \text{U}\ s, p, A) &= A \\
 \text{cast}(\text{Pi}\ s\ A\ B, \text{Pi}\ s\ A'\ B', p, f) &= \text{Lam}\ s\ (\text{CastPi}\ s\ A\ A'\ B\ B'\ p\ f) \\
 \text{cast}(A, B, p, a) &= \uparrow(\text{Cast}\ A\ B\ p\ a) \\
 \text{app}^{\mathcal{U}}(\text{CastPi}\ \mathcal{U}\ A\ A'\ B\ B'\ p\ f)\ a' &= \text{cast}(B[a], B'[a'], \text{PApp}_{\Omega}(\text{PSnd}\ p)\ \phi(a'), f \cdot_{\mathcal{U}} a) \\
 &\quad \text{where } a \triangleq \text{cast}(A', A, \text{PFst}\ p, a') \\
 \text{app}^{\mathcal{U}}(\text{CastPi}\ \Omega\ A\ A'\ B\ B'\ p\ f)\ q' &= \text{cast}(B[q], B'[q'], \text{PApp}_{\Omega}(\text{PSnd}\ p)\ q', f \cdot_{\Omega} q) \\
 &\quad \text{where } q \triangleq \text{PCast}\ A'\ A\ (\text{PFst}\ p)\ q'
 \end{aligned}$$

■ **Figure 6** Semantic cast function, and the associated closure.

$$\begin{aligned}
 q_n^{\text{Nf}}(\uparrow e) &= q_n^{\text{Ne}}(e) \\
 q_n^{\text{Nf}}(\text{U}\ s) &= s \\
 q_n^{\text{Nf}}(Z) &= 0 \\
 q_n^{\text{Nf}}(S\ a) &= S\ (q_n^{\text{Nf}}(a)) \\
 q_n^{\text{Nf}}(\text{Nat}) &= \mathbb{N} \\
 q_n^{\text{Nf}}(\text{Lam}\ s\ \mathcal{F}) &= \lambda_s. q_{n+1}^{\text{Nf}}(\text{vs}_n^{\mathcal{U}}(\mathcal{F})) \\
 q_n^{\text{Nf}}(\text{Pi}\ s\ A\ B) &= \Pi\ s\ (q_n^{\text{Nf}}(A)) \cdot (q_{n+1}^{\text{Nf}}(\text{vs}_n^{\mathcal{U}}(B))) \\
 q_n^{\text{Ne}}(\text{Var}_l) &= x_{n-l-1} \\
 q_n^{\text{Ne}}(\text{App}_s\ e\ a) &= (q_n^{\text{Ne}}(e))\ (q_n^{\text{Nf}}(a))^s \\
 q_n^{\text{Ne}}(\text{Rec}\ \mathcal{A}\ e\ a_0\ \mathcal{F}_S) &= \text{rec}[q_{n+1}^{\text{Nf}}(\text{vs}_n^{\mathcal{U}}(\mathcal{A}))] \\
 &\quad (q_n^{\text{Ne}}(e), q_n^{\text{Nf}}(a_0); q_{n+2}^{\text{Nf}}(\text{vs}_n^{\mathcal{U}}(\mathcal{F}_S))) \\
 q_n^{\text{Ne}}(\text{Eq}\ a\ A\ a') &= q_n^{\text{Nf}}(a) \sim_{q_n^{\text{Nf}}(A)} q_n^{\text{Nf}}(a') \\
 q_n^{\text{Ne}}(\text{Cast}\ A\ B\ p\ a) &= \text{cast}(q_n^{\text{Nf}}(A), q_n^{\text{Nf}}(B), q_n^{\Omega}(p), q_n^{\text{Nf}}(a))
 \end{aligned}$$

■ **Figure 7** Implementation of quoting for  $\text{CC}^{\text{obs}}$ :  $q_n^{\text{Nf}} : D^{\mathcal{U}} \rightarrow \text{Nf}$  and  $q_n^{\text{Ne}} : D^{\text{ne}} \rightarrow \text{Ne}$ .

**Quoting.** The mutually recursive functions  $q_n^{\text{Nf}} : D^{\mathcal{U}} \rightarrow \text{Nf}$  and  $q_n^{\text{Ne}} : D^{\text{ne}} \rightarrow \text{Ne}$  let us quote domain values back into normal and neutral forms, at de Bruijn level  $n$ . We rely on a helper function  $\text{vs}_n^{\mathcal{U}} : \text{Clos}_k \rightarrow D^{\mathcal{U}}$  to fully apply a closure to fresh variables.

$$\text{vs}_n^{\mathcal{U}}(\mathcal{A}) = \mathcal{A}[\uparrow \text{Var}_n, \dots, \uparrow \text{Var}_{n+k-1}]$$

Quoting is given in Figure 7, and follows the expected pattern. The cases for equality and casting do not ensure statically that they produce neutral forms. However, it is a program invariant that one position will be a blocking neutral, so the term is a neutral form in  $\text{Ne}$ .

### 3.3 Semantic propositions

Proofs, i.e. inhabitants of propositions, are all equal. We should thus never evaluate such irrelevant terms, and this should be reflected in the design of the domain of semantic propositions  $D^{\Omega}$ . We explored three different approaches, the first two of which (proof erasure and syntactic propositions) we found unsatisfactory, until we reached the final design we are happy with: implementing  $D^{\Omega}$  as a semantic domain.

**Proof erasure.** The simplest strategy to handle proof-irrelevance is to rather brutally *erase* irrelevant terms at evaluation, as suggested in Abel et al. [2]. This amounts to having a single semantic proposition, i.e. defining  $D^{\Omega} ::= \text{Witness}$ . This vastly simplifies the implementation, by removing intricate proof manipulations.

Although this approach is correct in terms of deciding the equational theory, it has a major drawback: normal forms cannot be quoted back to terms. Indeed, since proofs are erased, there is no information left to quote! This has nasty consequences. First, we must deal with this missing information when printing terms back to users, for instance in error messages. Quoting is also using during unification, to provide term solutions for unification variables – see Sec. 4.3.

As a mitigation, Abel et al. propose to extend the language by a constant  $O$  which proves every provable proposition. In practice, this would amount to indicate in quoted terms where proofs exist, but without giving a precise witness. However, one must be careful to exclude  $O$  from the source language, as it makes type-checking wildly undecidable by making it depend on inhabitation. Decidability could be recovered by instead having  $O$  prove *all* propositions, at the cost of making the type theory inconsistent, also far from ideal. Thus, in this approach we have to maintain a careful and clumsy distinction between “user-issued” terms and “kernel-generated” ones, with annoying consequences in the interaction with users: for instance, it means they would not be able to copy code from messages. We thus chose to reject this solution and look for another one.

**Syntactic propositions.** A natural alternative is to retain *syntactic* witnesses for proof terms during evaluation: since we never need to evaluate propositions, there seems to be no point in translating them to a domain such as  $D^U$ , which is designed for efficient evaluation. This amounts to setting  $D^\Omega ::= \text{Prop } t \ \rho$  – a proof witness  $t$  and an environment  $\rho$  interpreting its free variables. Indeed, while they do not reduce, propositions still admit *substitutions* of their variables, which happens when evaluating relevant terms, and must be accounted for. This is the point of the stored environment, used to interpret free variables during quoting.

More challenging, though, is the proof manipulation which occurs in casting reduction rules. This requires inserting a proof-relevant value into the proof witness and shifting propositions to be well-typed in a different context. In this approach, evaluation and quoting become mutually defined, and great care must be taken to transform witnesses correctly. This entanglement of evaluation and quoting is both unsatisfying and error-prone.

**Semantic propositions.** We therefore introduce a third design: using a semantic domain similar to  $D^U$ . The idea is that values in this domain never reduce, they only admit *substitution*, which is handled by closures, but they use de Bruijn levels, making it unnecessary to shift or quote terms during evaluation. As relevant data might appear as subterms of propositions,  $D^\Omega$  also embeds relevant terms. However, these do not reduce: they are “frozen”.

The domain of semantic propositions has a structure similar to terms, except for the use of closures for bound variables, and de Bruijn levels. Even let bindings and type annotations are represented, unevaluated. Calligraphic letters represent closures, which are the same as in Section 3.2, but have values from  $D^\Omega$  in place of  $D^U$ .

$$\begin{aligned} D^\Omega \ni P, Q, p, q ::= & \text{PVar}_l \mid \text{PU } s \mid \text{PLet } P \ p \ q \mid \text{PAnn } p \ P \mid \text{PAbsort } P \ p \mid \text{PEmpty} \\ & \mid \text{PLam } s \ \mathcal{P}_1 \mid \text{PApp}_s \ p \ q \mid \text{PPi } s \ P \ \mathcal{Q}_1 \mid \text{PZ} \mid \text{PS } p \mid \text{PRec } \mathcal{Q}_1 \ p \ q \ \mathcal{P}_2 \mid \text{PNat} \\ & \mid \text{POne} \mid \text{PUnit} \mid \text{PPair } p \ q \mid \text{PFst } p \mid \text{PSnd } p \mid \text{PExists } P \ \mathcal{Q} \\ & \mid \text{PTransp } p \ \mathcal{Q}_2 \ q \ p' \ e \mid \text{PEq } p \ P \ p' \mid \text{PCast } P \ \mathcal{Q} \ p \ q \mid \text{PRef } p \end{aligned}$$

Next, we introduce *freezing*, the mapping  $\phi : D^U \rightarrow D^\Omega$  of semantic relevant values into semantic propositions, so that they may be used in proof terms. It is defined mutually with  $\Phi_k : \text{Clos}_k^U \rightarrow \text{Clos}_k^\Omega$  which embeds closures. Representative cases are given as follows:

$$\begin{array}{llll}
\phi(\uparrow e) & = & \phi^{\text{Ne}}(e) & \phi^{\text{Ne}}(\text{Var}_l) & = & \text{PVar}_l \\
\phi(\text{U } s) & = & \text{PU } s & \phi^{\text{Ne}}(\text{App}_s e a) & = & \text{PApp}_s (\phi^{\text{Ne}}(e)) (\phi(a)) \\
\phi(\text{Lam } s \mathcal{F}) & = & \text{PLam } s (\Phi_1(\mathcal{F})) & \phi^{\text{Ne}}(\text{App}_s e p) & = & \text{PApp}_s (\phi^{\text{Ne}}(e)) p \\
\phi(\text{Prop } p) & = & p & & & \dots
\end{array}$$

Freezing has to traverse the whole term, which is rather inefficient, especially if we repeatedly freeze the same term. In retrospect, it might be possible to replace it by a mere constructor which freely embeds  $\mathcal{D}^{\mathcal{U}}$  into  $\mathcal{D}^{\Omega}$ .

Semantic interpretation for propositions,  $\llbracket \_ \rrbracket^{\Omega} : \text{Tm} \rightarrow \text{Env} \rightarrow \mathcal{D}^{\Omega}$ , is particularly easy as there are no reductions. We give only a few cases, others are entirely similar.

$$\begin{array}{llll}
\llbracket x_0 \rrbracket^{\Omega}(\rho, \text{P } p) & = & p & \llbracket \lambda x_s. t \rrbracket^{\Omega} \rho & = & \text{PLam } s (\lambda t) \rho \\
\llbracket x_0 \rrbracket^{\Omega}(\rho, \text{V } a) & = & \phi(a) & \llbracket t \text{ u}^s \rrbracket^{\Omega} \rho & = & \text{PApp}_s (\llbracket t \rrbracket^{\Omega} \rho) (\llbracket u \rrbracket^{\Omega} \rho) \\
\llbracket x_{i+1} \rrbracket^{\Omega}(\rho, d) & = & \llbracket x_i \rrbracket^{\Omega} \rho & & & \dots
\end{array}$$

Projections from the environment are like in relevant interpretation, although when the entry is relevant, we freeze it with  $\phi : \mathcal{D}^{\mathcal{U}} \rightarrow \mathcal{D}^{\Omega}$ . This handles substitution – syntactic variables  $x_i$  are substituted by values from the environment.  $\lambda$ -expressions introduce a closure which can be entered, but this never happens due to application, only during quoting. Interpretation of application always produces a **PApp**, even when the interpretation of  $t$  is **PLam**.

We also define a function  $\text{app}^{\Omega} : \text{Clos}_k^{\Omega} \rightarrow \mathcal{D} \rightarrow \dots \rightarrow \mathcal{D} \rightarrow \mathcal{D}^{\Omega}$  for applying closures. This works similarly to  $\text{app}^{\mathcal{U}}$ , only no reduction occurs: compared to Figure 5, we replace each semantic operator by a constructor. For example,

$$\text{app}^{\Omega} (\text{EqFun } s \text{ p } \mathcal{Q} \text{ p}') \text{ q} = \text{PEq} (\text{PApp}_s \text{ p } \text{q}) \mathcal{Q}[q] (\text{PApp}_s \text{ p}' \text{ q})$$

Finally, we also have quoting for semantic propositions  $\mathbf{q}_n^{\Omega}$ , which computes in the same way as quoting for  $\mathcal{D}^{\mathcal{U}}$ , fully applying closures to fresh variables.

### 3.4 Conversion checking

Conversion checking is used to decide the conversion relation  $\equiv$ , by operating on *semantic values*. This relation is non-trivial because normalisation, due to being untyped, does not handle extensionality rules. Yet, we still want to check equality up to  $\eta$  and irrelevance, so this is implemented when comparing semantic values. Conversion checking is *untyped* and *term-directed*, so  $\eta$ -expansion is triggered when one side of the conversion equation is a constructor, and the other is neutral.<sup>6</sup> For  $\Pi$  types, for instance, we get

$$\frac{\Gamma, x \vdash \mathcal{F}[\text{Var}_{[\Gamma]}^s] \equiv t' \cdot_s \text{Var}_{[\Gamma]}^s}{\Gamma \vdash \text{Lam } s \mathcal{F} \equiv t'}$$

In practice,  $\Gamma$  is replaced by an integer representing its length.

Conversion checking between irrelevant terms always succeeds. In fact, we only define conversion checking between values in  $\mathcal{D}^{\mathcal{U}}$ , and proof irrelevance is implemented by *not recursively comparing irrelevant terms*. Consider for instance applications:

$$\frac{\Gamma \vdash \uparrow e \equiv \uparrow e' \quad \Gamma \vdash a \equiv a'}{\Gamma \vdash \uparrow(\text{App}_{\mathcal{U}} e a) \equiv \uparrow(\text{App}_{\mathcal{U}} e' a')} \quad \frac{\Gamma \vdash \uparrow e \equiv \uparrow e'}{\Gamma \vdash \uparrow(\text{App}_{\Omega} e p) \equiv \uparrow(\text{App}_{\Omega} e' p')}$$

<sup>6</sup> This strategy is due to Coquand [10], and implemented in Coq and in Agda for functions.

When arguments are irrelevant,  $p \equiv p'$  is automatic and thus not checked. This explains the annotation of applications with the sort of their argument, to decide which rule to choose.

On top of  $\eta$ -conversion, we also implement the extra **cast**( $A, A, e, t$ )  $\equiv t$  equation. Treating this rule as a reduction breaks determinism, as there are multiple paths to evaluate some cast expressions. More worrying, it makes reduction and conversion checking mutually recursive. Thus, it is easier to implement it during conversion checking, although this necessitates backtracking. The implementation uses the following rules.

$$\begin{array}{c}
\text{CAST-EQ-LEFT} \\
\frac{\Gamma \vdash A \equiv B \quad \Gamma \vdash a \equiv a'}{\Gamma \vdash \text{Cast } A \ B \ e \ a \equiv a'} \\
\\
\text{CAST-EQ-RIGHT} \\
\frac{\Gamma \vdash A' \equiv B' \quad \Gamma \vdash a \equiv a'}{\Gamma \vdash a \equiv \text{Cast } A' \ B' \ e' \ a'} \\
\\
\text{CAST-CONV} \\
\frac{\Gamma \vdash A \equiv A' \quad \Gamma \vdash B \equiv B' \quad \Gamma \vdash a \equiv a'}{\Gamma \vdash \text{Cast } A \ B \ e \ a \equiv \text{Cast } A' \ B' \ e' \ a'}
\end{array}$$

When the left-hand side of the equality is a cast between  $A$  and  $B$ , we first try CAST-EQ-LEFT. If this fails, we proceed with CAST-EQ-RIGHT, and if that fails too, CAST-CONV. Backtracking is necessary as it is impossible to know a priori which strategy succeeds. Fortunately, eagerly applying CAST-EQ-LEFT and CAST-EQ-RIGHT is complete, i.e. it does not lose solutions. Indeed, if CAST-EQ-LEFT and CAST-CONV both apply to derive  $\text{Cast } A \ B \ e \ a \equiv \text{Cast } A' \ B' \ e' \ a'$ , then all four types  $A, B, A'$  and  $B'$  are convertible, and so CAST-CONV can be replaced by CAST-EQ-LEFT and CAST-EQ-RIGHT.

### 3.5 Bidirectional type checking

Our approach to type-checking, based on bidirectional typing, is close to e.g. Gratzner et al. [15], or what Agda implements [24]. We mutually implement the *two* judgements

$$\Gamma; \rho \vdash t \Rightarrow A \text{ (infer)} \qquad \Gamma; \rho \vdash t \Leftarrow A \text{ (check)}$$

Besides the context  $\Gamma$ , we have an environment  $\rho$  interpreting it, which is necessary to handle let-binders, as explained below. Typing uses *semantic* types,  $A \in \mathcal{D}^u$ . Although we do not include it on paper, type-checking actually *elaborates* terms: while checking a pre-term in  $\text{PreTm}$ , we generate a term in  $\text{Tm}$ , for instance annotating applications with a sort.

In this approach, constructors are checked, and destructors are inferred, and thus only normal forms are typable. To allow for more flexibility, we add two mechanisms. Type annotations let us locally record a type in a term to get back to type-checking when necessary, as per the following rule:

$$\frac{\Gamma; \rho \vdash A \Rightarrow U \ \mathfrak{s} \quad \Gamma; \rho \vdash t \Leftarrow \llbracket A \rrbracket^u \rho}{\Gamma; \rho \vdash (t : A) \Rightarrow \llbracket A \rrbracket^u \rho}$$

Let-bindings, on the other hand, can be checked or inferred, and the current mode is forwarded to the body  $u$ , while the bound term  $t$  is always checked against the (evaluation of the) type  $A$ .

$$\frac{\Gamma; \rho \vdash A \Leftarrow U \ \mathfrak{s} \quad \Gamma; \rho \vdash u \Leftarrow \llbracket A \rrbracket^u \rho \quad \Gamma; \rho \vdash A \Leftarrow U \ \mathfrak{s} \quad \Gamma; \rho \vdash u \Leftarrow \llbracket A \rrbracket^u \rho}{\Gamma, x :_{\mathfrak{s}} \llbracket A \rrbracket^u \rho; (\rho, \llbracket u \rrbracket^{\mathfrak{s}} \rho) \vdash t \Rightarrow B \quad \Gamma, x :_{\mathfrak{s}} \llbracket A \rrbracket^u \rho; (\rho, \llbracket u \rrbracket^{\mathfrak{s}} \rho) \vdash t \Leftarrow B}}{\Gamma; \rho \vdash \text{let } x :_{\mathfrak{s}} A = u \text{ in } t \Rightarrow B \quad \Gamma; \rho \vdash \text{let } x :_{\mathfrak{s}} A = u \text{ in } t \Leftarrow B}$$

The environment is extended with the *value* of the variable  $x$ , so that it can be used transparently while typing  $u$ . We also get sharing, because a let-bound  $t$  is only evaluated once at its binding site. Compare with a  $\beta$ -redex, where the environment is extended by a variable:

$$\frac{\Gamma; \rho \vdash A \Rightarrow \mathbf{U} \mathfrak{s} \quad \Gamma, x :_{\mathfrak{s}} A; (\rho, \text{Var}_{[\Gamma]}^{\mathfrak{s}}) \vdash B \Rightarrow \mathbf{U} \mathfrak{s} \quad \Gamma, x :_{\mathfrak{s}} A; (\rho, \text{Var}_{[\Gamma]}^{\mathfrak{s}}) \vdash t \Leftarrow \mathcal{B}[\text{Var}_{[\Gamma]}^{\mathfrak{s}}] \quad \Gamma; \rho \vdash u \Leftarrow A}{\Gamma; \rho \vdash (\lambda x. t : \Pi x : A. B) u \Rightarrow \mathcal{B}[u]}$$

Finally, while in vanilla MLTT **refl** and the equality type are constructors, this is not true in  $\text{CC}^{\text{obs}}$ . Since equality types reduce, it is unfeasible, given a semantic type, to guess which equality  $a \sim_A a'$  it represents. For instance, many equalities reduce to  $\perp$ . The reasonable solution is to have reflexivity infer, in line with viewing it as a destructor on the universe:

$$\frac{\Gamma; \rho \vdash t \Rightarrow A}{\Gamma; \rho \vdash \mathbf{refl} \, t \Rightarrow \mathbf{eq}(\llbracket t \rrbracket^{\mathcal{U}} \rho, A, \llbracket t \rrbracket^{\mathcal{U}} \rho)}$$

Note the use of the semantic equality type  $\mathbf{eq}$  as the inferred type. So, for example, **refl** 0 will infer  $\top$ , as we immediately reduce the type  $0 \sim_{\mathbb{N}} 0$ . Also note that this issue could have been avoided by having an equality type which does not compute, which is in hindsight a rather compelling argument for this approach.

## 4 Extensions

### 4.1 Quotient types

The ability to easily handle *quotient types* is one of the landmarks of observational type theory, it is thus natural to consider them as our first extension. The adaptation of NbE is actually relatively straightforward, and follows the same patterns as the core implementation. We first extend normal and neutral forms:

$$\begin{aligned} \text{Nf} \ni v, w, V, W &::= \dots \mid V/(x \, y. W, x. R_r, x \, y \, xRy. R_s, x \, y \, z \, xRy \, yRz. R_t) \mid \pi \, v \\ \text{Ne} \ni n, N &::= \dots \mid \mathbf{Q}\text{-elim}[z. V](x. v_{\pi}, x \, y \, xRy. t_{\sim}, n) \end{aligned}$$

Note that the proofs that the relation is an equivalence – in quotient formation – and that the quotient is respected – in the eliminator – are arbitrary, as they are propositions.

We extend the semantic domains accordingly, introducing three new closures to defunctionalise the equality type reduction between quotient types and its three binders.

$$\begin{aligned} \mathcal{D}^{\mathcal{U}} &::= \dots \mid \text{Quotient } A \, \mathcal{B}_2 \, \mathcal{P}_1^r \, \mathcal{P}_3^s \, \mathcal{P}_5^t \mid \mathbf{Qproj} \, a \\ \mathcal{D}^{\text{ne}} &::= \dots \mid \mathbf{Qelim} \, \mathcal{B}_1 \, \mathcal{F}_1 \, \mathcal{Q}_3 \, e \\ \mathcal{D}^{\Omega} &::= \dots \mid \mathbf{PQuotient} \, P \, \mathcal{Q}_2 \, \mathcal{P}_1^r \, \mathcal{P}_3^s \, \mathcal{P}_5^t \mid \mathbf{PQproj} \, p \mid \mathbf{PQelim} \, \mathcal{P}_1 \, \mathcal{P}_1' \, \mathcal{Q}_3 \, p \\ \text{Clos}_1^{\mathfrak{s}} &::= \dots \mid \mathbf{EqQuotientY} \, p \, a \, D \, D' \, \mathcal{C}_2 \, \mathcal{C}_2 \mid \mathbf{EqQuotientX} \, p \, D \, D' \, \mathcal{C}_2 \, \mathcal{C}_2 \mid \mathbf{EqQuotient} \, D \, D' \, \mathcal{C}_2 \, \mathcal{C}_2 \end{aligned}$$

Relevant interpretation for quotients follows the same pattern as the core NbE algorithm. Quotient types are interpreted into the **Quotient** constructor, with the appropriate closures. Projections are interpreted into the **Qproj** constructor. Elimination is defined by

$$\llbracket \mathbf{Q}\text{-elim}[z.B](t_{\pi}, t_{\sim}, u) \rrbracket^{\mathcal{U}}(\rho) = \mathbf{Qelim}((\lambda B)\rho, (\lambda t_{\pi})\rho, (\lambda t_{\sim})\rho, \llbracket u \rrbracket^{\mathcal{U}} \rho)$$

where  $\mathbf{Qelim}$  reduces quotient eliminators applied to projections.

$$\begin{aligned} \mathbf{Qelim}(\mathcal{B}, \mathcal{F}_{\pi}, \mathcal{P}_{\sim}, \mathbf{Qproj} \, b) &= \mathcal{F}_{\pi}[b] \\ \mathbf{Qelim}(\mathcal{B}, \mathcal{F}_{\pi}, \mathcal{P}_{\sim}, \uparrow e) &= \uparrow(\mathbf{Qelim} \, \mathcal{B} \, \mathcal{F}_{\pi} \, \mathcal{P}_{\sim} \, e) \end{aligned}$$

Irrelevant interpretation,  $\llbracket \_ \rrbracket^{\Omega} \_$ , is trivially extended: once again, there is no reduction when an eliminator is applied to a projection. Quoting also takes the same form as before.



## 4.2 Inductive types and Mendler-style induction

Our next extension is to add inductive types, following the theory laid down in Section 2.2. The normal forms are as follows – we omit the cases of inductive types without functor instances and fixed-points without views; they follow the same shape as those presented.

$$\begin{aligned}
\text{Nf} &::= \dots \mid \overrightarrow{\mu F : V \rightarrow \mathcal{U}. [C_i : (x_i : W_i) \rightarrow F v_i]} \text{ functor } X \ Y \ f \ p \ x = v \\
&\mid (\mu F : V \rightarrow \mathcal{U}. [C_i : (x_i : W_i) \rightarrow F v_i]) \text{ functor } X \ Y \ f \ p \ x = v) \ w \\
&\mid C_i (v, e) \mid \text{fix } [V \text{ as } G \text{ view } \iota] \ f \ p \ x : W = v \\
&\mid (\text{fix } [V \text{ as } G \text{ view } \iota] \ f \ p \ x : W = v) \ w \\
\text{Ne} &::= \dots \mid \text{match } n \text{ as } x \text{ return } V \text{ with } \{C_i (x_i, e_i) \rightarrow v_i\}_i \\
&\mid (\text{fix } [V \text{ as } G] \ f \ p \ x : W = v) \ w \ n \\
&\mid \text{in } n \mid \text{lift}[N] \ V \mid \text{fmap}[N]
\end{aligned}$$

Both unapplied inductive type (families) and fixed points are new normal forms. Inductive types applied to their indices are normal forms at the universe, and similarly for fixed-points. Finally, a fixed point applied to an index is neutral when its second argument, on which the fixed point computes, is itself neutral. The **lift** and **fmap** terms block when their inductive type is unknown – we cannot lift type families and functions without the definition at hand –, while **in** blocks when applied to a neutral.

**Semantic domains.** As usual, the first step is to extend the domains, driven by the structure of the normal and neutral forms. Like before, we omit values for inductive types without functor definitions (in the implementation, we have optional value for the functor definition).

$$\begin{aligned}
D &::= \dots \mid \text{Mu } A \ (\text{List } (\mathcal{B}_1, \mathcal{A}_2)) \ \mathcal{F}_5 \ (\text{Maybe } a) \mid \text{Cons Name } a \ p \mid \text{Fix } A \ \mathcal{B}_4 \ \mathcal{F}_5 \ (\text{Maybe } a) \\
D^{\text{ne}} &::= \dots \mid \text{Match } e \ \mathcal{B}_1 \ (\text{List } \mathcal{A}_2) \mid \text{FixApp } A \ \mathcal{B}_4 \ \mathcal{F}_5 \ a \ e \mid \text{In } e \mid \text{Lift } E \ A \mid \text{Fmap } E \ A \ B \ a \ b \ c
\end{aligned}$$

The semantic value of inductive types, **Mu**, contains the index type, followed by a list of pairs representing constructor types and indices. We have an optional value representing the presence or absence of an index. Fixed-points have a similar structure. The neutral form **FixApp** represents a fixed-point blocked by a neutral argument. Defunctionalizing closures and semantic propositions are entirely unsurprising, and we omit them here.

**Interpretation.** First, we give the semantics of the newly added terms.

$$\begin{aligned}
\llbracket \mu F : A \rightarrow \mathcal{U}. [C_i : (x_i : B_i) \rightarrow F a_i] \text{ functor } X \ Y \ f \ p \ x = t \rrbracket^{\mathcal{U}} \rho &= \\
&\text{Mu } (\llbracket A \rrbracket^{\mathcal{U}} \rho) \ [(\lambda B_i) \rho, (\lambda a_i) \rho]_i \ (\lambda t) \rho \ \text{Nothing} \\
\llbracket C_i (t, e) \rrbracket^{\mathcal{U}} \rho &= \text{Cons } C_i \ (\llbracket t \rrbracket^{\mathcal{U}} \rho) \ (\llbracket e \rrbracket^{\Omega} \rho) \\
\llbracket \text{match } t \text{ as } x \text{ return } C \text{ with } \{t_i\}_i \rrbracket^{\mathcal{U}} \rho &= \text{match}(\llbracket t \rrbracket^{\mathcal{U}} \rho, (\lambda C) \rho, [(\lambda a_i) \rho]_i) \\
\llbracket \text{fix } [M \text{ as } G \text{ view } \iota] \ f \ p \ x : C = t \rrbracket^{\mathcal{U}} \rho &= \text{Fix } (\llbracket M \rrbracket^{\mathcal{U}} \rho) \ (\lambda C) \rho \ (\lambda t) \rho \ \text{Nothing} \\
\llbracket \text{in } t \rrbracket^{\mathcal{U}} \rho &= \text{in}(\llbracket t \rrbracket^{\mathcal{U}} \rho) \\
\llbracket \text{lift}[M] \ A \rrbracket^{\mathcal{U}} \rho &= \text{lift}(\llbracket M \rrbracket^{\mathcal{U}} \rho, \llbracket A \rrbracket^{\mathcal{U}} \rho) \\
\llbracket \text{fmap}[M] (A, B, f, u, t) \rrbracket^{\mathcal{U}} \rho &= \text{fmap}(\llbracket M \rrbracket^{\mathcal{U}} \rho, \llbracket A \rrbracket^{\mathcal{U}} \rho, \llbracket B \rrbracket^{\mathcal{U}} \rho, \llbracket f \rrbracket^{\mathcal{U}} \rho, \llbracket u \rrbracket^{\mathcal{U}} \rho, \llbracket t \rrbracket^{\mathcal{U}} \rho)
\end{aligned}$$

The implementations of **match**, **lift**, **fmap** and **in** all branch on their argument being either a constructor, in which case the relevant branch is taken, or a neutral, which leads to a neutral. They are found in the code.

Reduction of fixed points is handled by application, and we thus extend  $\_ \cdot_s \_$  as follows:

$$\begin{aligned} (\text{Mu } A \text{ cs } \mathcal{F} \text{ Nothing}) \cdot_{\mathcal{U}} a &= \text{Mu } A \text{ cs } \mathcal{F} (\text{Just } a) \\ (\text{Fix } A \text{ C } \mathcal{F} \text{ Nothing}) \cdot_{\mathcal{U}} a &= \text{Fix } A \text{ C } \mathcal{F} (\text{Just } a) \\ (\text{Fix } A \text{ C } \mathcal{F} (\text{Just } a)) \cdot_{\mathcal{U}} (\text{Cons } C_i \text{ b } p) &= \mathcal{F}[A, \text{id}, \text{Fix } A \text{ C } \mathcal{F} \text{ Nothing}, a, \text{Cons } C_i \text{ b } p] \\ (\text{Fix } A \text{ C } \mathcal{F} (\text{Just } a)) \cdot_{\mathcal{U}} (\uparrow e) &= \uparrow (\text{FixApp } A \text{ C } \mathcal{F} a e) \end{aligned}$$

When a semantic inductive type or fixed-point have not been applied to their index – indicated by their final component being **Nothing** –, we move this index into the value. When a fixed-point with an index is applied to a constructor, we invoke the closure  $\mathcal{F}$ , i.e. the body of the fixed point. Note how we pass the semantic identity function  $\text{id} \triangleq \llbracket \lambda p \ x.x \rrbracket^{\mathcal{U}}$  for the view.

### 4.3 Pattern unification

*Pattern unification* [3, 18] does not extend the theory, but makes writing terms more practical. Our implementation combines the contextual metavariables of Abel and Pientka [1] and NbE. Another interesting aspect, as we already mentioned, is that unification partly motivates the structure we chose for semantic propositions in Section 3.3, as we will shortly explain. Contrarily to other work, we do not insist on computing only most general solution, which would require further modifications to the structure of neutral forms.

**Storing solved metavariables.** Pattern unification relies on metavariables, written  $?m$  and recorded in a metavariable context  $\Sigma$ , which contains metavariables already solved through unification, and yet unsolved ones, as follows.

$$\Sigma ::= \cdot \mid \Sigma, ?m_i \mid \Sigma, ?m_i := t \mid \Sigma, ?s_i \mid \Sigma, ?s_i := s$$

Note that we have metavariables  $?s$  for sorts, too. But what should  $t$  be? That is, should metavariables stand for terms or for semantic values? To answer this, we must understand what operations we want to perform on them.

Metavariables are introduced during typing. As explained in Section 3.5, typing really is an elaboration procedure, taking in a preterm and constructing a term. In the presence of metavariables, it becomes *stateful*, as the metavariable context can be updated. The interesting rule is that which handles a hole in the surface syntax, creating two new metavariables representing the term and its type, and adding both to the metavariable context.

$$\frac{?m_i, ?m_j \text{ fresh in } \Sigma}{\Sigma; \Gamma \vdash \_ \rightsquigarrow ?m_i \Rightarrow ?m_j; \Sigma, ?m_j, ?m_i}$$

As any other term, we need to be able to evaluate metavariables into the semantic domains. If a metavariable is unsolved, it behaves like a variable and blocks the term, creating a new form of neutrals. We extend the semantic domain accordingly:

$$\text{D}^{\text{ne}} \ni e ::= \dots \mid \text{Meta } ?m_i \rho$$

If a metavariable is defined, though, evaluation should reflect it. We achieve this as follows:

$$\begin{aligned} \llbracket ?m_i \rrbracket^{\mathcal{U}} \rho &= \llbracket t \rrbracket^{\mathcal{U}} \rho && \text{when } (?m_i := t) \in \Sigma \\ \llbracket ?m_i \rrbracket^{\mathcal{U}} \rho &= \text{Meta } ?m_i \rho && \text{when } ?m_i \in \Sigma \end{aligned}$$

Indeed, we want solved metavariables to admit a substitution operation, so that we can use them in a context different from the one they were created in. Evaluation of terms is

tailored to handle an environment representing a substitution, leading to the definition above. Semantic values, on the contrary, do not easily admit substitution. This leads to the choice of taking  $t \in \mathbf{Tm}$  above, and to use evaluation to handle the environment.

**Solving metavariables.** Metavariables are solved during conversion checking, from equations of the form  $\Delta \vdash \text{Meta } ?m_i \rho \equiv a$ . To solve such an equation, we create a partial function  $\text{invert} : \text{Env} \rightarrow \text{Renaming}$  which succeeds when the environment  $\rho$  satisfies the linearity conditions necessary to invert it. Partial renamings are lists of variables (with de Bruijn levels) and undefined values,  $\uparrow$ , i.e. they are given by the following datastructure

$$\text{Renaming} \ni \theta ::= () \mid (\theta, \text{Var}_l) \mid (\theta, \uparrow)$$

We equivalently view them as partial functions on variables when this view is more useful.

In conversion, we always compare *values*. However, metavariable solutions are terms. Therefore, when applying the renaming  $\rho^{-1}$  to the value  $a$ , we simultaneously update free variables and quote the semantic value into a term. This gives us an operation  $\text{rename}_{?m_i}^n : \mathbf{D}^{\mathcal{U}} \rightarrow \text{Renaming} \rightarrow \mathbf{Tm}^{\mathcal{U}}$ . Like with quoting, the level  $n$  representing the depth we rename at. We also have access to the metavariable  $?m_i$  for the occurs check.

$$\begin{aligned} \text{rename}_{?m_i}^n(\text{Var}_k) \theta &= x_{n-l+1} && \text{when } \theta(\text{Var}_k) = \text{Var}_l \\ \text{rename}_{?m_i}^n(\text{Meta } ?m_j) \theta &= ?m_j && \text{when } ?m_i \neq ?m_j \\ \text{rename}_{?m_i}^n(\uparrow (\text{App } n \ a)) \theta &= (\text{rename}_{?m_i}^n(\uparrow n) \theta) (\text{rename}_{?m_i}^n(a) \theta) \\ \text{rename}_{?m_i}^n(\uparrow (\text{Lam } s \ \mathcal{F})) \theta &= \lambda_s. (\text{rename}_{?m_i}^{n+1}(\mathcal{F}[\text{Var}_n])(\theta, \text{Var}_{n+1})) \end{aligned}$$

The first rule ensures the variable  $\text{Var}_k$  is not escaping, by checking it is defined in  $\theta$ . The second rule implements the occurs check by ensuring the metavariable  $?m_i$  does not appear in its own solution. This way, we isolate it from linearity, which depends only on the environment  $\rho$ . When going under a binder, we apply the closure to a fresh variable and extend the renaming in the natural way. A similar renaming operation is defined for semantic propositions in  $\mathbf{D}^{\Omega}$ . Since we want to obtain proper terms after renaming, we need  $\mathbf{D}^{\Omega}$  to contain enough information, explaining why the proof erasure semantics is not well-suited.

The final step is to piece these parts together to solve metavariables in conversion checking, then update the metavariable context. Both  $\text{invert}$  and  $\text{rename}$  might (validly) fail, so we take  $\doteq$  to mean that the result is defined, and assigned to the variable on the left.

$$\frac{\rho^{-1} \doteq \text{invert}(\rho) \quad t \doteq \text{rename}_{?m_i}^{|\Delta|}(a) \rho^{-1}}{(\Sigma, ?m_i, \Sigma'); \Delta \vdash ?m_i[\rho] \equiv a; (\Sigma, ?m_i := t, \Sigma')}$$

#### 4.4 A lightweight mechanism for goals

To be able to write complex terms, it is handy to build them incrementally. For this, we need a form of interactivity, where the proof assistant informs us as to what we need to provide to complete the proof: *proof goals*. In full-fledged proof assistants, goals are handled by the IDE and displayed using a secondary window. Implementing this is heavy, but proof goals are very necessary to be able to design even moderately-sized examples.

Instead, we implement a lightweight mechanism, where goals are inserted into the source code to throw an error at the specific location we want to investigate. The error message indicates the expected type at that point, if known. Additionally, goals optionally contain lists of terms whose types are reported to the user. We write  $\{t_1, t_2, \dots, t_n\}$  for a goal term and a list of terms  $t_i$ . For example, consider the following code, in which we insert a proof goal. Running the checker on the code on the left, we get the message on the right.

```

let f :  $\mathbb{N} \rightarrow \mathbb{N}$  =
   $\lambda x. S\ x$ 
in
let x :  $\mathbb{N}$  =
  S (S (S 0))
in
f {f, x}

```

## 5 Conclusion

isation by evaluation, and dependent type theory in general were reliable and very useful in the design of our system. Although still very much a prototype, critically missing a proper universe system and a positivity checker, our language is usable: we have been able to implement NbE for the simply typed  $\lambda$ -calculus in the style of Fiore [12] in it.

## — References

- 1 Andreas Abel. *Normalization by Evaluation Dependent Types and Impredicativity*. PhD thesis, Institut für Informatik Ludwig-Maximilians-Universität München, München, May 2013. URL: <https://www.cse.chalmers.se/~abela/habil.pdf>.
- 2 Andreas Abel, Thierry Coquand, and Miguel Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, pages 5–19, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-02273-9\_3.
- 3 Andreas Abel and Brigitte Pientka. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In Luke Ong, editor, *Typed Lambda Calculi and Applications*, volume 6690, pages 10–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-21691-6\_5.

- 4 Thorsten Altenkirch. Extensional equality in intensional type theory. In *Proceedings - Symposium on Logic in Computer Science*, pages 412–420, February 1999. doi:10.1109/LICS.1999.782636.
- 5 Thorsten Altenkirch and Conor McBride. Towards Observational Type Theory, 2006.
- 6 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, pages 57–68, Freiburg Germany, October 2007. ACM. doi:10.1145/1292597.1292608.
- 7 Bob Atkey. Simplified observational type theory, 2017. URL: <https://github.com/bobatkey/sott>.
- 8 Henning Basold and Herman Geuvers. Type Theory based on Dependent Inductive and Coinductive Types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 327–336, July 2016. doi:10.1145/2933575.2934514.
- 9 Rafael Bocquet. obstt, 2023. URL: <https://gitlab.com/RafaelBocquet/obstt>.
- 10 Thierry Coquand. An algorithm for testing conversion in type theory. *Logical frameworks*, 1:255–279, 1991.
- 11 Thierry Coquand and Christine Paulin. Inductively defined types. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, Per Martin-Löf, and Grigori Mints, editors, *COLOG-88*, volume 417, pages 50–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990. doi:10.1007/3-540-52335-9\_47.
- 12 Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. *Mathematical Structures in Computer Science*, 32(8):1028–1065, September 2022. doi:10.1017/S0960129522000263.
- 13 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without k. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, January 2019. doi:10.1145/329031610.1145/3290316.
- 14 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur*. PhD thesis, Université Paris VII, 1972.
- 15 Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341711.
- 16 Robert Harper. Boolean Blindness, 2011. URL: <https://existentialtype.wordpress.com/2011/03/15/boolean-blindness/>.
- 17 Martin Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995. URL: <https://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-327/>.
- 18 András Kovács. Elaboration with first-class implicit function types. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, August 2020. doi:10.1145/3408983.
- 19 András Kovács. Elaboration-zoo, May 2023. URL: <https://github.com/AndrasKovacs/elaboration-zoo>.
- 20 András Kovács. Efficient evaluation with controlled definition unfolding. In *3rd Workshop on the Implementation of Type Systems*, 2025.
- 21 Per Martin-Löf. Intuitionistic type theory. In *Studies in Proof Theory*, 1984.
- 22 Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 1999.
- 23 Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–172, March 1991. doi:10.1016/0168-0072(91)90069-X.
- 24 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.
- 25 Erik Palmgren. *On universes in type theory*, pages 191—204. Oxford University Press, 1998. doi:10.1093/oso/9780198501275.003.0012.

- 26   Loïc Pujet. *Computing with Extensionality Principles in Dependent Type Theory*. PhD thesis, Nantes Université, December 2022.
- 27   Loïc Pujet and Nicolas Tabareau. Observational equality: Now for good. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–27, January 2022. doi:10.1145/3498693.
- 28   Loïc Pujet and Nicolas Tabareau. Impredicative observational equality. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571739.
- 29   Loïc Pujet and Nicolas Tabareau. Observational equality meets cic. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 275–301. Springer Nature Switzerland, 2024. doi:10.1007/978-3-031-57262-3\_12.
- 30   Matthew Sirman. A normalisation by evaluation implementation of a type theory with observational equality. Bachelor’s thesis, University of Cambridge, 2023.
- 31   Matthew Sirman. observational-type-theory, May 2023. URL: <https://github.com/matthew-sirman/observational-type-theory>.
- 32   The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.