UNIVERSITY OF CAMBRIDGE

Candidate 2483F

A Normalisation by Evaluation Implementation of a Type Theory with Observational Equality

Part III Computer Science

This dissertation is submitted for Part III of the Computer Science Tripos in 2023

Total page count: 71
Main chapters: 48 (pp 6 – 53)
Main chapters word count: 11973
This word count was generated by the TEXcount script.

Abstract

In this project, we explore how *observational type theory*, in particular Pujet's $\mathrm{TT}^{\mathrm{obs}}$ system, can be implemented in Haskell. We use the *normalisation by evaluation* technique to produce an efficient normalisation function, which in turn is used to implement a bidirectional type checker. We then extend the core $\mathrm{TT}^{\mathrm{obs}}$ calculus with quotient types and inductive types, allowing for greater expressivity. To better the practicality of using the system also explore various proof-assistant features, notably pattern unification for inferring terms which can be deduced from context. For way of evaluation, we implement proofs within the system to exemplify the capabilities of the implementation.

Contents

Chapter 1

Introduction

Dependent type theories provide a framework for expressing programs with precise constraints, with proofs of correctness directly part of the program itself. Alternatively, they are a language for higher-order constructive logic associated with a mechanical procedure for checking correctness of proofs. This project details the implementation of a type checker for a dependent type theory with *observational equality*, called $\mathrm{TT}^{\mathrm{obs}}$ [1, 2]. By implementing this type system, we can automatically check correctness of proofs.

Central to dependent type theory is a relation for deciding when two terms are equal. We introduce a relation of *definitional equalities* which are simple enough to decide automatically. In general, semantic equality between arbitrary terms is undecidable, so we introduce an notion of *propositional* equality, living in the system itself.

Observational equality [3] is one formulation of propositional equality. It encodes *computation* of equality types: equalities reduce to their components, meaning proofs are broken down. We prove equality between terms by their observable behaviour, instead of how they were constructed. Observational equality behaves well with *proof-irrelevance*, where certain types contain at most one inhabitant with no computational meaning. When equality types are irrelevant, we recover canonicity of equalities by asserting that all proofs are equal. We also recover desirable properties which are impossible to prove in other settings, including function extensionality of proposition extensionality.

We explore how to apply *normalisation by evaluation* (NbE) [4] to observational type theory. NbE is a technique for efficiently computing normal forms of open terms by avoiding naïve substitution.

Of particular interest is dealing with the proof-irrelevant layer of $\mathrm{TT}^{\mathrm{obs}}$. As propositional proofs do not reduce, they require special treatment. We develop a novel technique for dealing with propositions in NbE in Section 3.2.3.

We first implement the core of $\mathrm{TT}^{\mathrm{obs}}$, which includes the only the essential components necessary. We design bidirectional typing rules and implement a type-checker (Section 3.3). Conversion checking – deciding convertibility of terms – compares *normal forms* of terms computed using NbE (Section 3.2). Next, we extend $\mathrm{TT}^{\mathrm{obs}}$ with a richer type system including quotient types (Section 3.4) and inductive types (Section 3.5). Quotients allow us to maximally exploit observa-

tional equality, by defining custom equivalence relations on types, and inductive types provide recursive data-structures with induction principles.

We then add *pattern unification*, a tool for automatically inferring terms which are uniquely determined from the surrounding context (Section 3.6). This does not extend the type theory itself, but improves the practicality of writing programs.

1.1 Related work

Normalisation by evaluation, both typed and untyped, is described in [4], also for dependent types. The implementation style for evaluation and bidirectional type-checking is based upon András Kovács' `elaboration-zoo` [5]. This is an extension of Coquand's algorithm for type-checking dependent types [6].

In [7], Fiore constructs a typed NbE algorithm from a categorical account. This is later translated into Agda code.

Abel et. al. [8] describe normalisation in the presence of proof-irrelevance. They introduce a special semantic value which canonically inhabits any proof-irrelevant type. This amounts to *proof erasure*: proof witnesses are erased during evaluation.

$\mathrm{TT}^{\mathrm{obs}}$ is given by Pujet and Tabareau in [1] and [2]. Metatheory and decidability are their primary focusses. They also gives the typing rules and theory behind quotient types and non-recursive inductive, which we build upon here.

Another earlier observational type theory was given by Altenkirch et. al. [9]. This paper gives the first example of an observational type theory, which reduces compromises from intensional and extensional type theories. $\mathrm{TT}^{\mathrm{obs}}$ is inspired by this work, and in particular adds definitional equality within proof-irrelevant types.

Chapter 2

Background

In this chapter, we give the necessary background on dependent type theory in Section 2.1 and normalisation by evaluation in Section 2.5 required for the following chapters.

2.1 Dependent type theory

Before introducing dependent type theory, consider the following grammar for the simply typed lambda calculus with syntax for types and terms respectively.

$$A, B ::= \top \mid \bot \mid \mathbb{N} \mid A \times B \mid A \to B$$
$$t, u ::= x \mid * \mid \mathbf{abort}\ t \mid 0 \mid S\ t \mid \mathbf{rec}(t, 0 \to t_0; S\ x \to t_S) \mid \langle t; u \rangle \mid \mathbf{fst}\ t \mid \mathbf{snd}\ t \mid \lambda x.\ t \mid t\ u$$

$\top$ is the unit type with one inhabitant, $*$, $\bot$ is the uninhabited type with elimination principle **abort**, and $\mathbb{N}$ is the type of Peano numbers generated by 0 and S, with a recursion principle **rec**. Variables are placeholders which are *substituted* for other terms. We write $t[u/x]$ for the term t with each free occurrence of x replaced by u.

We generally use the convention of uppercase Latin letters, A, B, C, etc. for types and lowercase Latin letters t, u, v, etc. for terms. Variables use letters x, y and z. We make a visual distinction between object-language syntax using bold text (e.g. **abort**) and meta-language syntax using sans-serif (e.g. Tm).

In a dependent type system, types depend on terms. A prominent example is dependent function types, which allow the codomain type to depend on the value of the argument. We also allow types and terms to depend on types and terms, giving the Calculus of Constructions [10]. Consider the following grammar from which we construct a dependent type system.

$$\begin{array}{rcll}
A, B, t, u & ::= & x & \text{Variable} \\
& \mid & * \mid \top & \text{Unit} \\
& \mid & \mathbf{abort}\ t \mid \bot & \text{Empty} \\
& \mid & 0 \mid S\ t \mid \mathbf{rec}[z.C](t, 0 \to t_0; (S\ x)\ y \to t_S) \mid \mathbb{N} & \text{Natural numbers} \\
& \mid & \langle t; u \rangle \mid \mathbf{fst}\ t \mid \mathbf{snd}\ t \mid \Sigma(x : A).\ B & \text{Dependent sums} \\
& \mid & \lambda x.\ t \mid t\ u \mid \Pi(x : A).\ B & \text{Dependent products} \\
& \mid & \mathcal{U} & \text{Universe}
\end{array}$$

Product and function types are replaced with Σ and Π types, their dependent analogues[1]. Both of these forms introduce a *variable* x into B, expressing a dependency. There is a new term, $\mathcal{U}$, representing the *universe of types*, also referred to as a *sort*. As types are now terms, they too must be typeable. Natural number recursion includes a *motive* $[z.\ C]$, which is an indexed return type, or an induction hypothese. This makes **rec** an *induction* principle.

Next, we introduce typing rules for these terms. We have three kinds of rules: *formation* rules, checking a type is valid, *introduction* forms, describing construction of a type, and *elimination* forms for using a value of a type. The context Γ binds free variables in *both* the term and the type in a judgement $\Gamma \vdash t : A$.

Consider the rules for Π-types.

$$\Pi\text{-Form}\quad \frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A \vdash B : \mathcal{U}}{\Gamma \vdash \Pi(x : A).\ B : \mathcal{U}}$$

$$\Pi\text{-Intro}\quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\ t : \Pi(x : A).\ B} \qquad \Pi\text{-Elim}\quad \frac{\Gamma \vdash t : \Pi(x : A).\ B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[u/x]}$$

The formation rule says $\Pi(x : A).\ B$ has type $\mathcal{U}$ (read: *is a type*) when A is a type, and B is a *family* of types indexed by A. The introduction rule says t must have type B in the extended context $\Gamma, x : A$. This implies B lives in the context $\Gamma, x : A$: it is allowed to depend on the argument to the function. Application substitutes the argument u for x in the return type, B.

Next we create typing rules for natural numbers.

$$\mathbb{N}\text{-Form}\quad \frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}} \qquad \mathbb{N}\text{-Intro-0}\quad \frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \mathbb{N}\text{-Intro-}S\quad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash S\ t : \mathbb{N}}$$

$$\mathbb{N}\text{-Elim}\quad \frac{\Gamma, z : \mathbb{N} \vdash C : \mathcal{U} \qquad \Gamma \vdash t : \mathbb{N} \qquad \Gamma \vdash t_0 : C[0/z] \qquad \Gamma, x : \mathbb{N}, y : C[x/z] \vdash t_s : C[S\ x/z]}{\Gamma \vdash \mathbf{rec}[z.C](t, 0 \to t_0; (S\ x)\ y \to t_S) : C[t/z]}$$

The formation and introduction rules are as expected. The induction principle is more involved. t_0 is the base case. Indeed, it is an element of the motive at 0. t_S is the inductive step. It acts generically in a natural number, x, and accesses the hypothesis through the variable y. It produces an element at index $S\ x$. Finally, the argument t picks out the inductively generated value at index t. This is *computed* by applying the inductive step to the base value t times.

As $\mathcal{U}$ is itself a term, it has a typing rule.

$$\mathcal{U}\text{-Form}\quad \frac{}{\Gamma \vdash \mathcal{U} : \mathcal{U}}$$

This in fact leads to inconsistency in the system through the Burali-Forti paradox [11]. We

[1]Confusingly, dependent *sums* (Σ-types) are the analogue of *products*.

ignore this problem here, noting it is resolved by introducing a countable hierarchy of universes of increasing size – each universe satisfies $\mathcal{U}_i : \mathcal{U}_{i+1}$, so no universe contains itself.

2.1.1 Definitional equality

Up to this point, we cannot *convert* types. For example, consider terms $f : \Pi(x : \top).\top$ and $u : (\lambda x.\top)\ *$. We want to judge $u : \top$, as $(\lambda x.\top)\ *$ *computes* to $\top$, making $f\ u : \top$ valid. In other words, $(\lambda x.\top)\ *$ should be *equal* to $\top$. To address this, we introduce *definitional equality*, $\Gamma \vdash t \equiv u : A$, which says t and u, both having type A, are equal. With this, we introduce another typing rule

$$\textsc{Conv}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}}{\Gamma \vdash t : B}$$

On terms, definitional equality is $\beta\eta$-equivalence.

$$\top\text{-}\eta\ \frac{}{\Gamma \vdash t \equiv u : \top}$$

$$\mathbb{N}\text{-}\beta_0\ \frac{}{\Gamma \vdash \mathbf{rec}[z.C](0, 0 \to t_0; (S\ x)\ y \to t_S) \equiv t_0 : C[0/z]}$$

$$\mathbb{N}\text{-}\beta_S\ \frac{\mathbf{rec}_t \triangleq \mathbf{rec}[z.C](t, 0 \to t_0; (S\ x)\ y \to t_S)}{\Gamma \vdash \mathbf{rec}[z.C](S\ t, 0 \to t_0; (S\ x)\ y \to t_S) \equiv t_S[t/x, \mathbf{rec}_t/y] : C[S\ t/z]}$$

$$\Sigma\text{-}\beta_1\ \frac{}{\Gamma \vdash \mathbf{fst}\ \langle t; u\rangle \equiv t : A} \qquad \Sigma\text{-}\beta_2\ \frac{}{\Gamma \vdash \mathbf{snd}\ \langle t; u\rangle \equiv u : B[t/x]} \qquad \Sigma\text{-}\eta\ \frac{}{\Gamma \vdash \langle \mathbf{fst}\ t; \mathbf{snd}\ t\rangle \equiv t : \Sigma(x : A).\ B}$$

$$\Pi\text{-}\beta\ \frac{}{\Gamma \vdash (\lambda x.\ t)\ u \equiv t[u/x] : B[u/x]} \qquad \Pi\text{-}\eta\ \frac{}{\Gamma \vdash \lambda x.\ t\ x \equiv t : \Pi(x : A).\ B}$$

β-rules correspond to reduction of terms, where an elimination form is adjacent to an introduction form. η-rules correspond to unicity principles: they exhibit the canonical form inhabiting a certain type (e.g. $\top$-η shows $*$ is the only inhabitant of $\top$). We do *not* have a η-rules for positive types: such laws enforce canonical forms, but are expensive when deciding this relation, or even undecidable. These rules define an equivalence relation on terms, however the β-rules are written suggestively indicating *reduction* when read left-to-right.

Alongside these rules, we need *congruence* rules allowing definitional equality to act recursively

on terms. For example, the following computes the term inside a projection when it is not a pair.

$$\textsc{Cong-}\mathbf{fst}\ \frac{\Gamma \vdash t \equiv u : \Sigma(x : A).\ B}{\Gamma \vdash \mathbf{fst}\ t \equiv \mathbf{fst}\ u : A}$$

Definitional equality on types computes by recursively comparing the subterms.

$$\top\textsc{-Conv}\ \frac{}{\Gamma \vdash \top \equiv \top : \mathcal{U}} \qquad \mathbb{N}\textsc{-Conv}\ \frac{}{\Gamma \vdash \mathbb{N} \equiv \mathbb{N} : \mathcal{U}} \qquad \mathcal{U}\textsc{-Conv}\ \frac{}{\Gamma \vdash \mathcal{U} \equiv \mathcal{U} : \mathcal{U}}$$

$$\Pi\textsc{-Conv}\ \frac{\Gamma \vdash A \equiv A' : \mathcal{U} \qquad \Gamma, z : A \vdash B[z/x] \equiv B'[z/x'] : \mathcal{U}}{\Gamma \vdash \Pi(x : A).\ B \equiv \Pi(x' : A').\ B' : \mathcal{U}}$$

Σ-type equality takes the same form as Π-type equality. For deciding definitional equality between Π-types, we check the domains are equal under Γ, and the codomains are equal under Γ extended with a fresh variable of type A. Note that B' is well-typed under $\Gamma, x : A'$, but z has type A, so this substitution is only well-typed after establishing $A \equiv A'$.

One final rule to tie this judgement together *variable equality*.

$$\textsc{Var-Conv}\ \frac{}{\Gamma \vdash x \equiv x : A}$$

This states that all variables are definitionally equal to themselves. In general different variables are not convertible.

2.2 Observational type theory

Observational type theory gives a notion of propositional equality. For an account of *inductive* equality, see Appendix A. The motivating idea is to equate terms using their *observable behaviour* instead of a uniform construction [9]. Using this, we add axioms like function and proposition extensionality to the system, without problems with normalisation. We enjoy canonicity due to *proof-irrelevance*.

2.2.1 Proof irrelevance

Before introducing observational equality, we require *proof-irrelevant propositions* to create types serving only as propositions. As mentioned in Section 2.1, (proof-relevant) types have sort $\mathcal{U}$. We introduce another sort, Ω, of proof-irrelevant types. To axiomatise irrelevance, we introduce the following definitional equality rule

$$\Omega\textsc{-Irrelevance}\ \frac{\Gamma \vdash A : \Omega \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \equiv u : A}$$

which says if two terms t and u both have type A, which is a *propositition* (it has sort Ω), then t and u are equal. Therefore, we cannot distinguish between proofs, so their content is irrelevant, implying proof-irrelevant types are subsingletons with at most one inhabitant.

2.2.2 Propositional logic

To define observational equality, we require additional machinery. We have a propositional universe containing proof-irrelevant types serving as propositions, however we have not discussed how to construct them.

First, we repurpose $\top$ and $\bot$ to act as true and false propositions (meaning they now live in Ω) since they are already subsingletons.

Dependent logical implication and universal quantification are given by Π-types. We extend Π-types with the ability to map between sorts.

$$\frac{\Gamma \vdash A : \mathfrak{s} \qquad \Gamma, x :_{\mathfrak{s}} A \vdash B : \mathfrak{s}'}{\Gamma \vdash \Pi(x :_{\mathfrak{s}} A).\ B : \mathfrak{s}'}\ \Pi\text{-Form}$$

The symbol $\mathfrak{s}$ is used for an arbitrary sort, $\mathcal{U}$ or Ω.

Note that the context and Π type include sort annotations, which are omitted when clear.

Dependent implication is given when both the sorts are Ω: it is a map from propositions to propositions. Universal quantification is given when the domain is *relevant*: the type represents an indexed family of propositions, and an inhabitant is a mapping from any element of the domain to a corresponding proof. Note that the sort of a Π-type is given by the codomain sort, so both implication and universal quantification are propositions.

We extend Σ-types in a similar manner. We reserve the notation of Σ-types for proof-relevant dependent pairs, and introduce $\exists$ for propositional dependent pairs.

$$\frac{\Gamma \vdash A : \mathfrak{s} \qquad \Gamma, x :_{\mathfrak{s}} A \vdash B : \Omega}{\Gamma \vdash \exists(x :_{\mathfrak{s}} A).\ B : \Omega}\ \exists\text{-Form}$$

When $\mathfrak{s}$ is Ω, we obtain dependent logical conjunction. When $\mathfrak{s}$ is $\mathcal{U}$, we instead have existential quantification.

2.2.3 Observational equality

Observational equality is introduced as a family of types.

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \sim_A u : \Omega}\ \text{Eq-Form} \qquad \frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathbf{refl}\ t : t \sim_A t}\ \text{Eq-Intro}$$

Eq-Elim

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A, p : t \sim_A x \vdash C : \Omega \qquad \Gamma \vdash u : C[t/x, \mathbf{refl}\ t/p] \qquad \Gamma \vdash t' : A \qquad \Gamma \vdash e : t \sim_A t'}{\Gamma \vdash \mathbf{transp}(t, x\ p.\ C, u, t', e) : C[t'/x, u/p]}$$

We have a *transport* operator, **transp** for manipulating irrelevant proofs. This transports a proof u, along a proof of equality, e. This is strong enough to prove that $\sim_A$ is a *congruence*: an equivalence relation supporting function application either side of the equality. To manipulate proof-relevant content via equality, we introduce a *casting* operator.

Cast

$$\frac{\Gamma \vdash A : \mathfrak{s} \qquad \Gamma \vdash B : \mathfrak{s} \qquad \Gamma \vdash e : A \sim_{\mathfrak{s}} B \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathbf{cast}(A, B, e, t) : B}$$

Casting works with both proof-relevant and propositional types, though the latter is subsumed by **transp**.

An important observation is that casts and transports do *not* compute on **refl** like J does with inductive. This is a feature of irrelevance: we care only that we have an equality proof, not that it is reflexivity.

Next, we discuss the definitional equalities associated with observational equality. The observational equality type $t \sim_A u$ behaves differently from other types we have seen. Equalities *reduce* by decomposition into smaller components.

We give some of the rules for how equalities reduce (recall that reduction is expressed through definitional equality). First, consider equalities concerning the natural numbers type.

Eq-$\mathbb{N}$

$$\frac{}{\Gamma \vdash \mathbb{N} \sim_{\mathcal{U}} \mathbb{N} \equiv \top : \Omega}$$

Eq-0

$$\frac{}{\Gamma \vdash 0 \sim_{\mathbb{N}} 0 \equiv \top : \Omega}$$

Eq-S

$$\frac{}{\Gamma \vdash S\ n \sim_{\mathbb{N}} S\ n' \equiv n \sim_{\mathbb{N}} n' : \Omega}$$

Eq-S-0

$$\frac{}{\Gamma \vdash S\ n \sim_{\mathbb{N}} 0 \equiv \bot : \Omega}$$

Eq-0-S

$$\frac{}{\Gamma \vdash 0 \sim_{\mathbb{N}} S\ n \equiv \bot : \Omega}$$

Viewing propositions logically, $\top$ is the type of a true proposition, and $\bot$ represents falsehood. Thus, saying $0 \sim_{\mathbb{N}} 0$ *reduces* to $\top$ is means it holds trivially. Similarly, $0 \sim_{\mathbb{N}} S\ n$ reducing to $\bot$ means $0 \sim_{\mathbb{N}} S\ n$ is uninhabited: there are no proofs of this equality.

Next we define observational equality rules for Π and Σ types.

Eq-Π

$$\frac{a \triangleq \mathbf{cast}(A', A, e, a')}{\Gamma \vdash \Pi(x :_{\mathfrak{s}} A).\ B \sim_{\mathcal{U}} \Pi(x' :_{\mathfrak{s}} A').\ B' \equiv \exists(e : A' \sim_{\mathfrak{s}} A).\ \Pi(a' : A').\ B[a/x] \sim_{\mathcal{U}} B'[a'/x'] : \Omega}$$

Eq-Fun

$$\frac{}{\Gamma \vdash f \sim_{\Pi(x:A).B} g \equiv \Pi(a : A).\ f\ a \sim_{B[a/x]} g\ a : \Omega}$$

Eq-Ω

$$\frac{}{\Gamma \vdash P \sim_\Omega Q \equiv (P \to Q) \wedge (Q \to P) : \Omega}$$

Eq-Σ

$$\frac{a' \triangleq \mathbf{cast}(A, A', e, a)}{\Gamma \vdash \Sigma(x :_\mathfrak{s} A).\ B \sim_\mathcal{U} \Sigma(x' :_\mathfrak{s} A').\ B' \equiv \exists(e : A \sim_\mathfrak{s} A').\ \Pi(a : A).\ B[a/x] \sim_\mathcal{U} B'[a'/x'] : \Omega}$$

Eq-Pair

$$\frac{\begin{array}{c}\mathbf{ap}\ B\ e \triangleq \mathbf{transp}(\mathbf{fst}\ t, z\ \_.\ B[\mathbf{fst}\ t/x] \sim_\mathcal{U} B[z/x], \mathbf{refl}\ B[\mathbf{fst}\ t/x], \mathbf{fst}\ u, e) \\ t_2 \triangleq \mathbf{cast}(B[\mathbf{fst}\ t/x], B[\mathbf{fst}\ u/x], \mathbf{ap}\ B\ e, \mathbf{snd}\ t)\end{array}}{\Gamma \vdash t \sim_{\Sigma(x:A).B} u \equiv \exists(e : \mathbf{fst}\ t \sim_A \mathbf{fst}\ u).\ t_2 \sim_{B[\mathbf{fst}\ u/x]} \mathbf{snd}\ u : \Omega}$$

The rule Eq-Π says equality of Π-types is equivalent to showing that their domains are equal, and for any argument $a' : A'$, their codomains are equal[2]. Eq-Σ computes similarly, but with a symmetric proof. The rule Eq-Fun says proving functions f and g are equal is equivalent to a proof of pointwise equality. We note that this is precisely function extensionality, posed as a definitional axiom: the original motivation for pursuing observational equality. We equate functions *observationally* by saying they are equal when they have the same observable behaviour. Eq-Ω axiomatises propositional extensionality: equality of propositions is equivalent to bi-implication – we denote non-dependent conjunction $\exists(\_ : A).\ B$ by $A \wedge B$. Eq-Pair says equality of pairs is a pair of proofs that the first and second components are equal. We use **ap** (implemented using **transp**) to apply the type family B to either side of the equality e.

Other rules for observational equality reduction are similar, with composite propositions equated with the appropriate combination of their components, trivial propositions equating to $\top$ and false propositions equating to $\bot$.

2.2.4 Casting rules

Casts between relevant types also reduce. As a simple example, we consider casting in the natural numbers.

Cast-0

$$\frac{}{\Gamma \vdash \mathbf{cast}(\mathbb{N}, \mathbb{N}, e, 0) \equiv 0 : \mathbb{N}}$$

Cast-S

$$\frac{}{\Gamma \vdash \mathbf{cast}(\mathbb{N}, \mathbb{N}, e, S\ n) \equiv S\ (\mathbf{cast}(\mathbb{N}, \mathbb{N}, e, n)) : \mathbb{N}}$$

These rules simply proceed by recursion on the structure of the given number. This anticipates more general positive types appearing later on.

[2]We could equivalently formulate this symmetrically, with a proof $e : A \sim A'$. The chosen direction is more convenient, as functions are contravariant in their domains.

There are casting rules for Π and Σ types.

CAST-Π

$$\frac{a \triangleq \mathbf{cast}(A', A, \mathbf{fst}\ e, a')}{\begin{array}{c}\Gamma \vdash \mathbf{cast}(\Pi(x : A).\ B, \Pi(x' : A').\ B', e, f) \equiv \\ \lambda a'.\ \mathbf{cast}(B[a/x], B'[a'/x], \mathbf{snd}\ e\ a', f\ a) : \Pi(x' : A').\ B'\end{array}}$$

CAST-Σ

$$\frac{a' \triangleq \mathbf{cast}(A, A', \mathbf{fst}\ e, \mathbf{fst}\ t)}{\Gamma \vdash \mathbf{cast}(\Sigma(x : A).\ B, \Sigma(x' : A').\ B', e, t) \equiv \langle a'; \mathbf{cast}(B[\mathbf{fst}\ t/x], B[a'/x], \mathbf{snd}\ e\ (\mathbf{fst}\ t), \mathbf{snd}\ t)\rangle}$$

Propositional equality types between Π and Σ types *reduce*, so the projections extract their subproofs. In each case, the second projection is a *family* of proofs, so we apply an argument to extract a particular proof.

The general form of **cast**s on positive types is to determine the head constructor of the term, and then use the same constructor to build a new term, proceeding to recursively cast the subterms. For negative types with a canonical introduction form (e.g. Π and Σ), the η laws let us preemptively determine the head constructor by η-expanding the term being cast.

When casting a *type* between equal universes, **cast** also reduces.

CAST-UNIV

$$\frac{}{\Gamma \vdash \mathbf{cast}(\mathfrak{s}, \mathfrak{s}, e, A) \equiv A : \mathfrak{s}}$$

It seems natural to add a rule to reduce casts whenever the two types are definitionally equal; after all, the term is already typeable under the target type. We might propose the rule

CAST-EQ

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

which subsumes many previous rules. However, this rule is problematic as it stands. If this rule is treated as a *reduction*, then there are multiple paths to reduce some cast expressions, which breaks determinism[3]. It also makes reduction and conversion checking mutually recursive.

Alternatively, this rule can exist within the conversion checking procedure, but this necessitates backtracking. We opt for this strategy here.

Another alternative is to add a propositional constant which witnesses this equality [1].

2.3 Quotient types

Quotient types allow us to exploit the *setoid* structure of types [1] by explicitly defining an equivalence relation over a pre-existing type. Types formed from a quotient must obey this artificial equivalence relation as if it were observational equality.

Quotient types are formed constructively from an underlying (proof-relevant) base type, a

[3]At this point we have no distinction between *reduction* and *definitional equality*. Mathematically, they play the same role, so this foreshadows future concerns.

binary relation on this type and a proof that this relation is an equivalence. Let A be the base type. A binary relation on A is a function $R : A \to A \to \Omega$ – each pair of elements maps to a proposition which is either inhabited or not, making R a predicate. We require an explicit proof that R is an equivalence relation.

The syntax for quotient types is as follows.

$$A/(x\ y.\ R, x\ R_r, x\ y\ xRy.\ R_s, x\ y\ z\ xRy\ yRz.\ R_t)$$

We use the shorthand notation A/R.

Quotient formation is typed by the following rule.

Quotient-Form

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A, y : A \vdash R : \Omega \qquad \Gamma, x : A \vdash R_r : R[x, x] \\ \Gamma, x : A, y : A, xRy : R[x, y] \vdash R_s : R[y, x] \\ \Gamma, x : A, y : A, z : A, xRy : R[x, y], yRz : R[y, z] \vdash R_t : R[x, z]}{\Gamma \vdash A/(x\ y.\ R, x.\ R_r, x\ y\ xRy.\ R_s, x\ y\ z\ xRy\ yRz.\ R_t) : \mathcal{U}}$$

We first check that A is a relevant type, and R is a binary relation over A. The proof term R_r is indexed over an arbitrary $x : A$, acting as a generic proof of reflexivity. The proofs R_s and R_t act similarly, and symmetrise and compose proofs respectively.

Terms of the quotient type are introduced by $\pi\ t$, which projects a term t of the base type into the quotient. The equivalence relation divides the underlying type into *equivalence classes* of related elements; projection sends an element to its equivalence class. The typing rule is as follows.

Quotient-Intro

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \pi\ t : A/R}$$

Finally, elimination is given by the term

$$\mathbf{Q\text{-}elim}[z.\ B](x.\ t_\pi, x\ y\ xRy.\ t_\sim, u)$$

Elimination is a map out of the underlying type which preserves the equivalence relation. Constructively, this means associating the map with a proof that related elements in the quotient map to observationally equal elements of the codomain.

Quotient-Elim

$$\frac{B[xRy] \triangleq \mathbf{transp}(\pi\ x, z\ \_.\ B[\pi\ x] \sim_{\mathfrak{s}} B[z], \mathbf{refl}\ B[\pi\ x], \pi\ y, xRy) \\ \Gamma, z : A/R \vdash B : \mathfrak{s} \qquad \Gamma, x : A \vdash t_\pi : B[\pi\ x] \\ \Gamma, x : A, y : A, xRy : R[x, y] \vdash t_\sim : \mathbf{cast}(B[\pi\ x], B[\pi\ y], B[xRy], t_\pi\ x) \sim_{B[\pi\ y]} (t_\pi\ y) \\ \Gamma \vdash u : A/R}{\Gamma \vdash \mathbf{Q\text{-}elim}[z.\ B](x.\ t_\pi, x\ y\ xRy.\ t_\sim, u) : B[\pi\ u]}$$

The term t_π is the dependent map out of the underlying type A. $t_\sim$ is the proof that the equivalence relation is respected as observational equality in the codomain. This way, we enforce the opacity of quotient types: distinct elements from A might both be projected into A/R, but if

they are related by R, they are no longer distinguishable.

Here, we treated $xRy : R[x, y]$ as a proof of $\pi\ x \sim_{A/R} \pi\ y$ to create a proof $B[\pi\ x] \sim_{\mathfrak{s}} B[\pi\ y]$. To justify this, we introduce observational equality rules for quotients in the following section.

2.3.1 Observational equality for quotient types

To integrate quotient types with observational type theory, we need to specify how observational equality and casting behaves on them.

Between two quotient types, observational equality reduces to a composite equality between the components: equality between the underlying types and the relations. The equivalence proofs are irrelevant, and therefore trivially equal. The rule encapsulating this is

Quotient-Eq

$$\frac{x' \triangleq \mathbf{cast}(A, A', e, x) \qquad y' \triangleq \mathbf{cast}(A, A', e, y)}{\Gamma \vdash A/R \sim_{\mathcal{U}} A'/R' \equiv \exists(e : A \sim_{\mathcal{U}} A').\ \Pi(x :_{\mathfrak{s}} A).\Pi(y :_{\mathfrak{s}} A).R[x, y] \sim_{\Omega} R'[x', y']}$$

which steps to a dependent proof-irrelevant conjunction of proofs that the base types are equal, and at any pair of elements, the relations are equal.

Equality on projected terms resolves to the equivalence relation R. This unifies the artificial equivalence relation with the observational-equality-induced setoid structure, justifying the Quotient-Elim rule.

Quotient-Proj-Eq

$$\frac{}{\Gamma \vdash \pi\ t \sim_{A/R} \pi\ u \equiv R[t, u]}$$

We also have a casting rule between projected elements.

Quotient-Proj-Cast

$$\frac{}{\Gamma \vdash \mathbf{cast}(A/R, A'/R', e, \pi\ t) \equiv \pi\ (\mathbf{cast}(A, A', \mathbf{fst}\ e, t))}$$

We reduce casts when the term being cast reaches a normal form, by pushing the cast under the projection. To do so, we project the first component of the equality proof e, where e is a pair of proofs between the base types and the relations.

2.4 Inductive types

Inductive types are recursively-defined positive types with named constructors. *Indexed* inductive types are *type families* parameterised by an index type A. Each constructor specifies the index of the value it constructs and recursive occurences may use different indices.

We start with some simple examples.

Example 2.4.1 (Length indexed vectors)**.** Consider the type $Vec\ n$ of length n vectors of natural numbers. We define the following inductive type.

$$
\begin{aligned}
&\mu\, Vec\ : \mathbb{N} \to \mathcal{U}.\\
&\qquad [\ Nil : Vec\ 0\\
&\qquad ;\ Cons : \Pi(n : \mathbb{N}).\ \mathbb{N} \to Vec\ n \to Vec\ (S\ n)\\
&\qquad]
\end{aligned}
$$

The *Nil* constructor creates a 0-length vector as it contains no data. The *Cons* constructor takes a number and a length n vector to create a length $n + 1$ vector. As we work in a constructive setting, we also require the witness n to be present in the constructor.

Our inductive types are *first-class* [12, 13] meaning we can bind them to variables, pass them into functions and generally treat them as any other value. Terms representing inductive type have the following form:

$$\mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]$$

The variable F is bound by μ. A represents the index type. We have a finite set of constructors, each with a name C_i, data of type B_i, and an index a_i, which depends on the data through x_i. The F at the end of each constructor is syntax indicating the type being constructed; it is not a free variable.

We restrict inductive types to exactly one parameter and one value in each constructor. This loses no expressivity (albeit is more difficult to use practically) but simplifies the theory and implementation.

The formation rule for inductive types is as follows.

Inductive-Form

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathcal{U} \qquad \{\Gamma, F : A \to \mathcal{U} \vdash B_i : \mathcal{U}\}_i\\ \{\Gamma, F : A \to \mathcal{U}, x_i : B_i[F] \vdash a_i : A\}_i\end{array}}{\Gamma \vdash \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}] : A \to \mathcal{U}}$$

This rule checks that A is a relevant type. Then, for each constructor, we bind F and check that B_i is a type. B_i recursively refers to the whole type via the variable F. Finally, we check that each index has type A.

Values of inductive types are created by the constructors. We use a trick called *fording* [3] to allow all constructors build a value of type $(\mu F)\ a$ for *any* index a, but require a proof that $a_i \sim_A a$, rather than enforcing this condition definitionally.

Inductive-Intro

$$\frac{\begin{array}{c}\mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]\\ \Gamma \vdash t : B_i[\mu F/F] \qquad \Gamma \vdash e : a_i[\mu F/F, t/x_i] \sim_A a\end{array}}{\Gamma \vdash C_i\ (t, e) : (\mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}])\ a}$$

We implicitly check that C_i is a constructor in the inductive type. Then, we check that t has type B_i, substituting μF into B_i to check subterms. Finally, we check e witnesses that a and a_i are equal.

Elimination of inductive types is single-level, total pattern matching. Totalilty is necessary for consistency; unmatched patterns could prove arbitrary propositions. The rule for pattern

matching follows.

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

First, we check the discriminant t has an inductive type. Then, we check the return type C is a type family indexed by the inductive type. Finally, for every constructor C_i, we check the corresponding branch t_i has type C with $C_i\ (x_i, e_i)$ substituted for x. Therefore, whichever constructor is matched on is substituted into C, justifying the return type of the whole expression, $C[t/x]$.

This rule appears strange, as the type C does *not* abstract over the index. Suppose we match on $Nil : Vec\ 0$ from Example 2.4.1, so $x : Vec\ 0 \vdash C : \mathfrak{s}$. In the *Cons* branch, we substitute *Cons* into C. However, the given index of *Cons* is $S\ n$, so this appears ill-typed! But due to fording, we *can* have $Cons : Vec\ 0$, given a proof $S\ m \sim 0$. We have access to this proof e in the *Cons* branch, but $S\ m \sim 0 \equiv \bot$, so we **abort** e into the correct return type. This makes sense, as the *Cons* branch is unreachable.

We conclude matching with the β-reduction rule.

$$\textsc{Inductive-}\beta\quad \frac{}{\Gamma \vdash \textbf{match}\ (C_i\ (t, e))\ \textbf{as}\ x\ \textbf{return}\ C\ \textbf{with}\ \{C_i\ (x_i, e_i) \to t_i\} \equiv t_i[t/x_i, e/e_i]}$$

This finds the matching branch and substitutes in the data from the constructor.

2.4.1 Observational equality for inductive types

Next we integrate inductive types with observational equality, so they interact with the rest of the $\mathrm{TT}^{\mathrm{obs}}$ system.

First, we introduce observational equality between inductive types. We might wish to reduce these to composite propositions proving the index and constructor types are equal. This is quite complex, so we instead opt for only reducing syntactically equal inductive types. Therefore, equality serves only to equate the indices. The reduction rule is

$$\textsc{Eq-Inductive}\quad \frac{\mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]}{\Gamma \vdash (\mu F)\ a \sim_{\mathcal{U}} (\mu F)\ a' \equiv a \sim_A a'}$$

Having a single index greatly simplifies this reduction – with an arbitrary number, we need to compare telescopes of parameters with numerous casts to ensure deeper equations are well-typed.

Inductive equality between constructors behaves like equality between natural numbers. When two constructors are equal, the proposition steps to equality of the contents. When the constructors are different, the proposition steps to $\bot$. This encodes injectivity of constructors – different

constructors never construct equal values.

$$\textsc{Eq-Cons}\quad \frac{\mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]}{\Gamma \vdash C_i\ (t, e) \sim_{(\mu F)a} C_i\ (u, e') \equiv t \sim_{B_i[\mu F]} u}$$

$$\textsc{Eq-Cons-}\neq\quad \frac{C_i \neq C_j \qquad \mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]}{\Gamma \vdash C_i\ (t, e) \sim_{(\mu F)a} C_j\ (u, e') \equiv \bot}$$

Casts on constructors reduce by composing equality proofs to correct the indices. Here we exploit the fording proof.

$$\textsc{Cast-Cons}\quad \frac{\mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]}{\Gamma \vdash \mathbf{cast}((\mu F)\ a, (\mu F)\ a', e, C_i\ (t, e')) \equiv C_i\ (t, e' \circ e)}$$

Here, $\circ$ represents proof concatenation, defined by

$$(p : x \sim y) \circ (q : y \sim z) \triangleq \mathbf{transp}(y, w\ \_.\ x \sim w, p, z, q)$$

The proofs have types $e : a \sim_A a'$ and $e' : a_i[\mu F, t] \sim_A a$, so their composite is $e' \circ e : a_i[\mu F, t] \sim_A a'$, making the constructor well typed at $(\mu F)\ a'$. Note that casting only modifies the proof at the top layer and never traverses the structure.

2.4.2 Mendler induction

Pattern matching as elimination for inductive types does not fully exploit inductive types. In particular, we have no notion of *induction* or *recursion* over data-structures – we can look only one level at a time with pattern matching. This motivates adding the induction principle, **fix**. The design of **fix** is guided by the categorical intuition found in Appendix B.

We present **fix**, in a style extending Mendler recursion [14] to dependent induction.

$$\textsc{Fix}\quad \frac{\begin{array}{c}\mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}] \\ F[X] \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i[X/F]) \to F\ a_i}] \\ \Gamma, G : A \to \mathcal{U}, p : A, x : G\ p \vdash C : \mathfrak{s} \\ \Gamma, G : A \to \mathcal{U}, f : \Pi(p : A).\ \Pi(x : G\ p).\ C[G, p, x], p : A, x : F[G]\ p \vdash t : C[F[G], p, x]\end{array}}{\Gamma \vdash \mathbf{fix}\ [\mu F\ \mathbf{as}\ G]\ f\ p\ x : C = t : \Pi(p : A).\ \Pi(x : (\mu F)\ p).\ C[\mu F, p, x]}$$

We make two auxilliary definitions. We abbreviate the inductive type to μF. $F[X]$ represents the functor defining the inductive type applied to the type family X. This is defined by constructing a new inductive type, where X is substituted for F into each B_i. Therefore, $F[X]$ is mute in the variable F. Intuitively, this represents adding a single layer of structure over the family X.

Next, we check that the fixed-point is well-formed. We require that the induction principle is *well-founded*: recursive calls (corresponding to the induction hypotheses) occur only on *strictly smaller* terms. Well-foundedness is essential for consistency of the system. Mendler induction

operates by introducing a generic type family, G, on which we make recursive calls. Then, we lift G to $F[G]$. This adds a single level on top of G, where each recursive position is replaced by the opaque type G.

C is the induction hypothesis. It depends on the opaque variable G, the index p and x, the argument of the fixed-point (one may note this is not very useful as G is opaque so nothing can be learned from x; this is addressed in Section 2.4.3). The body of the fixed-point additionally depends on the recursive function f, which acts only on G. Therefore, to invoke f recursively, we deconstruct $x : F[G]$ using pattern matching, so each recursive position is an element of G, and can be passed to f. This ensures recursive calls occur only on subtrees.

Computation with **fix** occurs when it is applied to an index and a constructor.

FIX-β

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

The fixed-point reduces by substituting *its own definition* for f in t, so recursive calls have access to the definition. The variable G is substituted with the inductive type μF, meaning the fixed-point always semantically acts on the "real" type: the variable G indeed only serves as a placeholder for statically ensuring termination. Finally, fixed-points only reduce when applied to constructors – without this, conversion is undecidable, as fixed-points can unroll indefinitely.

Together, fixed-points and pattern matching admit *catamorphisms*: generalised folds over tree-like structures. In the following section, we extension this to *paramorphisms*, providing primitive recursion.

2.4.3 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 about the inductive type.

This motivates extending catamorphisms to *paramorphisms*. We introduce an extra function $\iota : \Pi(p : A).\ G\ p \to (\mu F)\ p$ which *views* the opaque type G as the inductive type it represents. Semantically, this is the identity – after all, the variable G is substituted for the inductive type.

To type-check the body of the fixed-point, we need to lift $\iota : \Pi(p : A).\ G\ p \to (\mu F)\ p$ into $\iota' : \Pi(p : A).\ F[G]\ p \to (\mu F)\ p$. This requires the defining functor of μF to lift indexed functions. *Strict positivity* is a sufficient condition to construct this functor. However, we do not check strict positivity, so we instead require an explicit definition.

We extend inductive type definitions to include a proof of functoriality.

$$\mu F : A \to \mathcal{U}.\ \overrightarrow{[C_i : (x_i : B_i) \to F\ a_i]}\ \mathbf{functor}\ X\ Y\ f\ p\ x = t$$

The extended typing rule now checks the functor action on functions.

INDUCTIVE-FORM

$$\frac{\begin{array}{c} F[Z] \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i[Z/F]) \to F\ a_i}] \\ \Gamma \vdash A : \mathcal{U} \qquad \{\Gamma, F : A \to \mathcal{U} \vdash B_i : \mathcal{U}\}_i \qquad \{\Gamma, F : A \to \mathcal{U}, x_i : B_i[F] \vdash a_i : A\}_i \\ \Gamma, X : A \to \mathcal{U}, Y : A \to \mathcal{U}, f : \Pi(p : A).\ X\ p \to Y\ p, p : A, x : F[X]\ p \vdash t : F[Y]\ p \end{array}}{\Gamma \vdash \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]\ \textbf{functor}\ X\ Y\ f\ p\ x = t : A \to \mathcal{U}}$$

The definition of $F[Z]$ creates an inductive type *without* a functor instance. The new hypothesis in this rule checks that t defines the functor action on functions. For correctness, we need proofs that t satisfies the functor laws ($F[\mathrm{id}_X] = \mathrm{id}_{F[X]}$ and $F[g \circ f] = F[g] \circ F[f]$) which could be checked by definitional equality. For simplicity, we ignore these proofs.

With this definition, we introduce a term **fmap** for projecting the functorial action from an inductive type.

FMAP

$$\frac{\begin{array}{c} F[Z] \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i[Z/F]) \to F\ a_i}] \\ \mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]\ \textbf{functor}\ X\ Y\ f\ p\ x = t \end{array}}{\begin{array}{l} \Gamma \vdash \textbf{fmap}\ [\mu F] : \Pi(X : A \to \mathcal{U}).\ \Pi(Y : A \to \mathcal{U}).\ (\Pi(p : A).\ X\ p \to Y\ p) \\ \qquad \to \Pi(p : A).\ F[X]\ p \to F[Y]\ p \end{array}}$$

We also introduce a term witnessing the $\textbf{in} : F[\mu F] \to \mu F$ isomorphism.

INDUCTIVE-IN

$$\frac{\begin{array}{c} \mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}] \\ F[X] \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i[X/F]) \to F\ a_i}] \\ \Gamma \vdash t : (F[\mu F])\ a \end{array}}{\Gamma \vdash \textbf{in}\ t : (\mu F)\ a}$$

Semantically, **in** is the identity function on constructors: $\textbf{in}\ (C_i\ (t, e)) \equiv C_i\ (t, e)$

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

FIX

$$\frac{\begin{array}{c} \mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}] \\ F[X] \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i[X/F]) \to F\ a_i}] \\ \iota' \triangleq \lambda p.\ \lambda x.\ \textbf{in}\ (\textbf{fmap}\ [\mu F]\ G\ \mu F\ \iota\ p\ x) \qquad \mathrm{id} \triangleq \lambda p.\ \lambda x.\ x \\ \Gamma, G : A \to \mathcal{U}, \iota : \Pi(p : A).\ G\ p \to (\mu F)\ p, p : A, x : G\ p \vdash C : \mathfrak{s} \\ \Gamma, G : A \to \mathcal{U}, \iota : \Pi(p : A).\ G\ p \to (\mu F)\ p, f : \Pi(p : A).\ \Pi(x : G\ p).\ C[G, \iota, p, x], p : A, x : F[G]\ p \\ \qquad \vdash t : C[F[G], \iota', p, x] \end{array}}{\Gamma \vdash \textbf{fix}\ [\mu F\ \textbf{as}\ G\ \textbf{view}\ \iota]\ f\ p\ x : C = t : \Pi(p : A).\ \Pi(x : (\mu F)\ p).\ C[\mu F, \mathrm{id}, p, x]}$$

The variable ι is now accessible in both the induction hypothesis and the body of the fixed-point, facilitating primitive recursion. At the top level, ι is substituted by the identity function. By functoriality, this also means ι' is $\textbf{in} \circ \mathrm{id}$, which also behaves as the identity.

2.5 Normalisation by evaluation

We now turn away from the type theory, and towards how one might implement it. In particular, we need a decision procedure for definitional equality of terms. This is done by comparing β-*normal forms* structurally. β-normal forms are terms which cannot be further reduced. We do not formally define reduction, but note that it corresponds to definitional equalities read left-to-right, and excludes the η-laws.

Formally, a function nf, mapping terms to terms, *computes* β-normal forms if the conditions

$$(1)\ \ \Gamma \vdash t \equiv \mathsf{nf}(t) \qquad (2)\ \ \Gamma \vdash t \equiv u \Longrightarrow \Gamma \vdash \mathsf{nf}(t) \equiv_\eta \mathsf{nf}(u) \qquad (3)\ \ \mathsf{nf}(\mathsf{nf}(t)) = \mathsf{nf}(t)$$

each hold. Condition (1) says the normal form $\mathsf{nf}(t)$ is definitionally equal to t (normalisation does not alter the meaning of terms). (2) says definitionally equal terms have equal normal forms. Note an important caveat here that we only require nf to compute β-normal terms; it does not need to η-expand. The sub-relation $\equiv_\eta$ relates terms which differ only by η-laws, but are otherwise syntactically identical. We could alternatively define normalisation to compute η-long normal forms and compare them purely structurally, though this is harder practically. Condition (3) says normalisation is idempotent; iterating nf has no additional effect ($=$ represents syntactic equality).

Normalisation sends each term to a representative of its $\equiv$-equivalence class. It is used to test equality between terms by checking equality of their normal forms. Representatives of each class are not unique, but are all equal up to η-equivalence, which is easy to check when equating terms [6].

In dependent type-checking, it is necessary to normalise *open* expressions for checking definitional equality. An implementation might perform normalisation by reduction rules using term substitution. This fits the declarative mathematical style of typing rules, however in practice is complex, error-prone and inefficient. The *normalisation by evaluation* (NbE) technique offers an efficient alternative to normalise open terms.

The idea lies in constructing a *semantic domain* of values, rather than reducing them directly. This domain is constructed to retain enough information to *quote* semantic values back to syntax. The key property is that the terms produced by quoting are β-normal forms, so a normalisation function is derived by first interpreting a term to a semantic value, and then quoting.

2.5.1 Semantic interpretation

To derive an NbE algorithm, we need an interpretation into a semantic domain. One way this is achieved is by choosing a category with a structure capable of modelling the semantics of the language. In this formulation, contexts and types are interpreted as objects, and well-typed terms are morphisms. However, designing a suitable structure categorically becomes increasingly difficult as the complexity of the language increases. Therefore, we use an *untyped* NbE algorithm, like in [4].

To motivate NbE in a small dependently typed language, first consider the following syntax

of terms, and their normal and neutral forms.

$$\begin{array}{llll} \mathsf{Tm} & \ni & A, B, t, u & ::= \quad x_i \mid * \mid \top \mid \lambda.\ t \mid t\ u \mid \Pi A.\ B \mid \mathcal{U} \\ \mathsf{Nf} & \ni & v, w, V, W & ::= \quad * \mid \top \mid \lambda.\ v \mid \Pi V.\ W \mid \mathcal{U} \mid u \\ \mathsf{Ne} & \ni & u, U & ::= \quad x_i \mid u\ v \end{array}$$

Variables are represented by *de Bruijn indices*, making binder names redundant. De Bruijn indices replace variable names by the number of binders between the variable and its binding-site[4]. In practice, names are retained at binding sites for quoting to human-readable terms.

All variables are neutral forms, regardless of their type – this means we allow non-η-unique normal forms, for example the variable f and the term $\lambda x.\ f\ x$ are equal by η-equivalence, and both normal forms.

We now construct a semantic domain D of values, given by the following mutually defined abstract datatype.

$$\begin{array}{llll} \mathsf{D} & \ni & a, b, A, B & ::= \quad \mathsf{Star} \mid \mathsf{Unit} \mid \mathsf{Lam}\ (\underline{\lambda} t)\rho \mid \mathsf{Pi}\ A\ (\underline{\lambda} t)\rho \mid \mathsf{U} \mid \uparrow e \\ \mathsf{D}^{\mathsf{ne}} & \ni & e & ::= \quad \mathsf{Var}_l \mid \mathsf{App}\ e\ a \\ \mathsf{Env} & \ni & \rho & ::= \quad () \mid (\rho, a) \end{array}$$

Environments are interpretations for contexts, represented by lists of values. In the Lam and Pi constructors we have a *closure* $(\underline{\lambda} t)\rho$ consisting of a term $t \in \mathsf{Tm}$ and an environment $\rho \in \mathsf{Env}$. Closures represent *continuations* – computations which need another value before proceeding.

In the semantic domain, we use de Bruijn *levels* for variables. Whereas de Bruijn indices count the binders between the variable and the binding site, levels count from the top level down to the binding site. Indices are problematic for semantic values because they change when moved under binders, which amounts to inefficient traversals of terms during substitutions. On the other hand, levels are constant for each variable, so substitutions avoid traversals.

An *interpretation* maps well-typed syntactic terms into the semantic domain. An interpretation in *typed* NbE is total by construction, however, in untyped NbE, we have no such guarantees, so we settle for a *partial* interpretation map

$$[\![\_]\!]\_ : \mathsf{Tm} \rightharpoonup \mathsf{Env} \rightharpoonup \mathsf{D}$$

We use $\rightharpoonup$ for partial meta-language functions.

This map is nonetheless constructed to be total on well-typed inputs. That is, if $\Gamma \vdash t : A$, and ρ interprets Γ, then $[\![t]\!]\rho$ is defined.

[4]With respect to the tree structure of the term, not a string representation.

With this, we give the semantics of terms in Tm

$$\begin{aligned}
[\![x_0]\!](\rho, a) &= a \\
[\![x_{i+1}]\!](\rho, a) &= [\![x_i]\!]\rho \\
[\![*]\!]\rho &= \mathsf{Star} \\
[\![\top]\!]\rho &= \mathsf{Unit} \\
[\![\lambda.t]\!]\rho &= \mathsf{Lam}\ (\underline{\lambda}t)\rho \\
[\![t\ u]\!]\rho &= [\![t]\!]\rho \cdot [\![u]\!]\rho \\
[\![\Pi A.\ B]\!]\rho &= \mathsf{Pi}\ [\![A]\!]\rho\ (\underline{\lambda}B)\rho \\
[\![\mathcal{U}]\!]\rho &= \mathsf{U}
\end{aligned}$$

We also need a mutually defined application operator, $\_ \cdot \_ : \mathsf{D} \to \mathsf{D} \to \mathsf{D}$. This is given by

$$\begin{aligned}
(\mathsf{Lam}\ (\underline{\lambda}t)\rho) \cdot a &= [\![t]\!](\rho, a) \\
(\uparrow e) \cdot a &= \uparrow(\mathsf{App}\ e\ a)
\end{aligned}$$

Here we see how closure continuations work. We store the environment when the λ-term is interpreted, which awaits one additional value: the argument to the function. When the Lam value is applied to an argument a, we proceed by evaluating the stored term t with the extended environment (ρ, a).

In the second case, a blocked neutral applied to a value becomes a blocked application; there is no way to reduce it.

2.5.2 Quoting

The final component of NbE is the quoting function. This maps the semantic domain back into syntactic terms, however it targets just those terms living in the Nf fragment. We define two mutually recursive functions $\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}} : \mathsf{D} \to \mathsf{Nf}$ and $\mathsf{q}^{\mathsf{Ne}}_{\mathsf{n}} : \mathsf{D}^{\mathsf{ne}} \to \mathsf{Ne}$. The subscript n represents the de Bruijn level we are quoting at. This is used to recover the de Bruijn index of variables.

$$\begin{aligned}
\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}}(\uparrow e) &= \mathsf{q}^{\mathsf{Ne}}_{\mathsf{n}}(e) \\
\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}}(\mathsf{Star}) &= * \\
\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}}(\mathsf{Unit}) &= \top \\
\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}}(\mathsf{Lam}\ (\underline{\lambda}t)\rho) &= \lambda.\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n+1}}([\![t]\!](\rho, \uparrow \mathsf{Var}_n)) \\
\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}}(\mathsf{Pi}\ A\ (\underline{\lambda}t)\rho) &= \Pi(\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}}(A)).\ (\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n+1}}([\![t]\!](\rho, \uparrow \mathsf{Var}_n))) \\
\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}}(\mathsf{U}) &= \mathcal{U} \\
\mathsf{q}^{\mathsf{Ne}}_{\mathsf{n}}(\mathsf{Var}_l) &= x_{n-l-1} \\
\mathsf{q}^{\mathsf{Ne}}_{\mathsf{n}}(\mathsf{App}\ e\ a) &= (\mathsf{q}^{\mathsf{Ne}}_{\mathsf{n}}(e))\ (\mathsf{q}^{\mathsf{Nf}}_{\mathsf{n}}(a))
\end{aligned}$$

The interesting cases in quoting are Lam and Pi. In both cases, we quote a closure by creating a fresh semantic variable and applying the closure to it. Note that semantic variables always quote into variables, regardless of type. This re-emphasises that we do not η-expand at function types.

Using de Bruijn levels for quoting ensures Var_n is always fresh.

Now we have constructed a semantic interpretation map, which maps terms into the semantic domain D, and a quoting function which maps semantic terms into normal forms. The final step is to compose these to get a function

$$\mathsf{nbe}_\Gamma = \mathsf{q}^{\mathsf{Nf}}_{|\Gamma|} \circ [\![\_]\!](\uparrow^\Gamma) : \mathsf{Tm} \to \mathsf{Nf}$$

where $|\Gamma|$ represents the length of the context, and

$$\uparrow^\diamond = ()$$
$$\uparrow^{\Gamma,x:A} = (\uparrow^\Gamma, \uparrow \mathsf{Var}_{|\Gamma|})$$

constructs an environment of variables interpreting Γ.

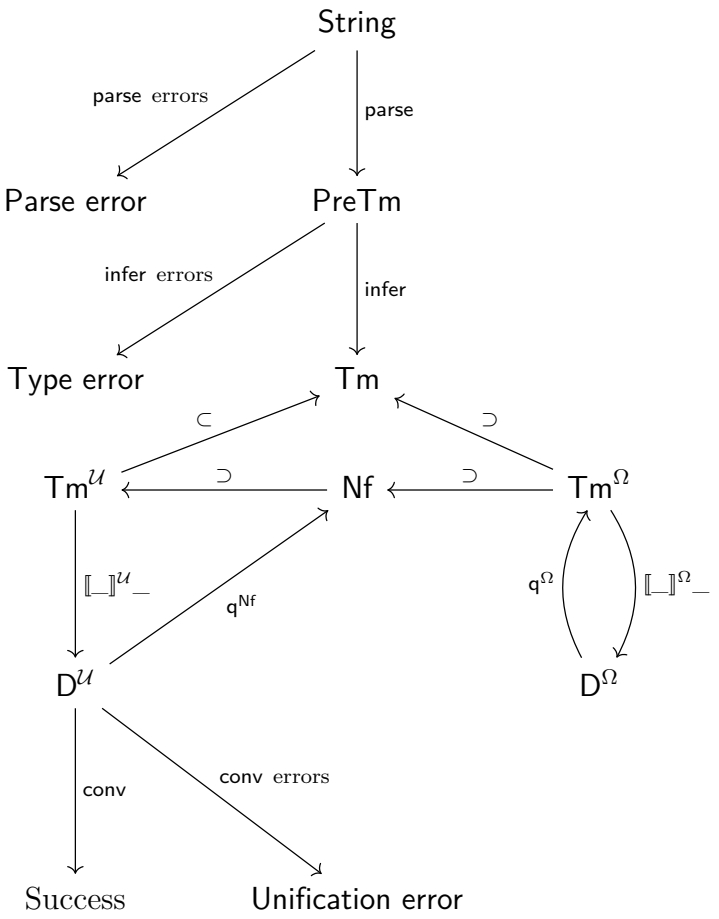
To summarise, we constructed a partial function $[\![\_]\!]\_$ which maps well-typed syntactic terms $\Gamma \vdash t : A$ and environments interpreting their free variables into a semantic domain D. Then we created a quoting function q^{Nf} to convert values back into normal forms. By composing these, we ended up with a normalisation function nbe_Γ.

Chapter 3

Implementation

In this chapter, we discuss the implementation of $\mathrm{TT}^{\mathrm{obs}}$. The implementation is written in Haskell, and makes extensive use of structural data types and declarative style to keep the implementation as close as possible to the theory.

In Section 3.1, we cover the core syntax of the language. Then, we look at the implementation of evaluation in Section 3.2. In Section 3.3 we implement the core typing rules of $\mathrm{TT}^{\mathrm{obs}}$, followed by extensions. Finally, we look at pattern unification in Section 3.6 which provides an essential tool for using the system in practice. Figure 3.1 gives a high-level overview of the implementation.



Source code

Parsing

Pre-syntax (Section 3.1)

Type inference and elaboration (Section 3.3.2)

All terms

Relevant terms, normal forms and irrelevant terms.

Interpretation and quoting

Semantic values (Section 3.2)

Conversion checking (Section 3.3.1) and unification (Section 3.6)

Figure 3.1: A high-level overview of the interaction of the components in the type-checker.

3.1 Syntax

Before discussing type-checking and normalisation by evaluation, we require an abstract syntax for $\mathrm{TT}^{\mathrm{obs}}$. In this section, we describe the core syntax; extensions are added later in this chapter.

$\mathrm{TT}^{\mathrm{obs}}$ is a dependent type system, so types depend on arbitrary terms and we create one combined grammar for everything, as described in Section 2.1. The grammar for terms is given in Figure 3.2, and is approximately the grammar given in [1] with the addition of let bindings and type annotations – these are important with bidirectional type-checking.

$\mathfrak{s}$	$::=$	$\mathcal{U} \mid \Omega$	Universe sorts
A, B, C, t, u, e	$::=$	x	Variable
	$\mid$	$\mathfrak{s}$	Universe
	$\mid$	$\lambda x_{\mathfrak{s}}.\ t \mid t\ u^{\mathfrak{s}} \mid \Pi(x :_{\mathfrak{s}} A).\ B$	Dependent products
	$\mid$	$0 \mid S\ t \mid \mathbf{rec}[z.C](t, 0 \to t_0; (S\ x)\ y \to t_S) \mid \mathbb{N}$	Natural numbers
	$\mid$	$(t, u) \mid \mathbf{fst}\ t \mid \mathbf{snd}\ t \mid \exists(x : A).\ B$	Proof-irrelevant sums
	$\mid$	$\mathbf{abort}_A\ t \mid \bot$	Empty type
	$\mid$	$* \mid \top$	Unit type
	$\mid$	$\mathbf{refl}\ t \mid \mathbf{transp}(t, x\ y.\ C, u, t', e) \mid t \sim_A u$	Observational equality
	$\mid$	$\mathbf{cast}(A, B, e, t)$	Casting
	$\mid$	$\mathbf{let}\ x :_{\mathfrak{s}} A = t\ \mathbf{in}\ u$	Let binding
	$\mid$	$(t : A)$	Type annotation

Figure 3.2: Basic syntax for $\mathrm{TT}^{\mathrm{obs}}$

Note that application is tagged with the sort of the argument, and λ-terms are annotated with their domain sort. This is necessary for evaluation, but is always inferred during type-checking; these annotations are not present in the source syntax.

This grammar inductively defines the type PreTm of *pre-terms*: syntactically well-formed, but otherwise untyped terms.

We define another type Tm of well-typed terms. Every well-typed term has a (unique) *sort*: $\mathcal{U}$ or Ω. We let $\mathsf{Tm}^{\mathcal{U}}$ and Tm^{Ω} be terms with sort $\mathcal{U}$ and Ω respectively. Certain functions have precondition restrictions on the terms they accept: notably, evaluation requires well-typed terms. Pre-terms use named variables, but typed terms use de Bruijn indices.

Normals and neutrals are defined mutually as a predicate on Tm.

Nf	$\ni$	v, w, V, W	$::=$	$n \mid \mathfrak{s} \mid \lambda x.\ v \mid \Pi(x :_{\mathfrak{s}} V).\ W \mid 0 \mid S\ v \mid \mathbb{N} \mid \exists(x : V).\ W \mid \bot \mid \top \mid e$
Ne	$\ni$	n, N	$::=$	$x_i \mid n\ v^{\mathfrak{s}} \mid \mathbf{rec}[z.V](n, 0 \to v; (S\ x)\ y \to w) \mid \mathbf{abort}_V\ t$
			$\mid$	$v \sim_n w \mid n \sim_{\mathbb{N}} v \mid v \sim_{\mathbb{N}} n \mid n \sim_{\mathcal{U}} v \mid v \sim_{\mathcal{U}} n$
			$\mid$	$\mathbf{cast}(\mathbb{N}, \mathbb{N}, e, n) \mid \mathbf{cast}(N, V, e, v) \mid \mathbf{cast}(V, N, e, v)$

The normal forms are as expected and correspond to constructors and types. In contrast to [1], these are not *weak-head* normal forms, so subterms are also required to be normal. The neutral forms are somewhat more involved.

First, the observational equality type can be blocked in *three* places: the type of the terms, or either of the terms themselves. There are no neutral forms when the type of the equality is at

a Π-type – this is because such equalities *always* reduce. Similarly, casts can also block in three positions: either of the types, or the term being cast. Casts between universes and Π-types also always reduce, and so cannot be blocked by the argument.

Second, we note proof-irrelevant terms appear indirectly. As they have no notion of reduction, we say *all* proof-irrelevant terms are normal forms, albeit they are treated slightly differently. The normal form e stands for any well-typed term with irrelevant sort.

Third, note that there are some overlapping cases in the definition of neutrals, for example $n \sim_{\mathbb{N}} v$ and $v \sim_{\mathbb{N}} n$, which overlap on $x \sim_{\mathbb{N}} y$.

3.2 Normalisation by evaluation

In Section 2.5, we exemplified untyped normalisation by evaluation. Here, we extend this to an NbE algorithm for $\mathrm{TT}^{\mathrm{obs}}$. The overall picture is the same: we give a semantic interpretation into an untyped domain of values, and a quoting function to reconstruct normal forms. In $\mathrm{TT}^{\mathrm{obs}}$ we must also account for proof-irrelevant propositions and complex reduction rules for propositions and casts.

3.2.1 Relevant layer

The reduction of the proof-relevant fragment of $\mathrm{TT}^{\mathrm{obs}}$ is implemented as an untyped NbE algorithm.

First, we construct the various domains data-structures in Figure 3.3. The domains $\mathsf{D}^{\mathcal{U}}$ and

$$
\begin{array}{llll}
\mathsf{D}^{\mathcal{U}} & \ni & a, b, A, B & ::= \mathsf{U}\ \mathfrak{s} \mid \mathsf{Lam}\ \mathfrak{s}\ \mathcal{F}_1 \mid \mathsf{Pi}\ \mathfrak{s}\ A\ \mathcal{B}_1 \\
 & & & \quad \mid \mathsf{Z} \mid \mathsf{S}\ a \mid \mathsf{Nat} \mid \mathsf{Exists}\ A\ \mathcal{B}_1 \mid \mathsf{Empty} \mid \mathsf{Unit} \\
 & & & \quad \mid \uparrow e \\
\mathsf{D}^{\mathsf{ne}} & \ni & e & ::= \mathsf{Var}_l \mid \mathsf{App}\ e\ d \mid \mathsf{Rec}\ \mathcal{A}_1\ e\ a\ \mathcal{F}_2 \\
 & & & \quad \mid \mathsf{Abort}\ A\ p \mid \mathsf{Eq}\ a\ A\ a' \mid \mathsf{Cast}\ A\ B\ p\ a \\
\mathsf{D}^{\Omega} & \ni & p, q, P, Q & ::= \cdots \\
\mathsf{D} & \ni & d, f, g, D & ::= \mathsf{V}\ a \mid \mathsf{P}\ p \\
\mathsf{Env} & \ni & \rho & ::= () \mid (\rho, d) \\
\mathsf{Closure}^{\mathfrak{s}}_k & \ni & \mathcal{C}_k & ::= (\underline{\lambda} t)\rho & : \mathsf{Closure}^{\mathfrak{s}}_k \\
 & & & \quad \mid \mathsf{Lift}\ d & : \mathsf{Closure}^{\mathfrak{s}}_k \\
 & & & \quad \mid \mathsf{EqFun}\ \mathfrak{s}\ f\ \mathcal{C}_1\ g & : \mathsf{Closure}^{\mathfrak{s}}_1 \\
 & & & \quad \mid \mathsf{EqPi}\ \mathfrak{s}\ D\ D'\ \mathcal{C}_1\ \mathcal{C}'_1 & : \mathsf{Closure}^{\mathfrak{s}}_1 \\
 & & & \quad \mid \mathsf{EqPi'}\ \mathfrak{s}\ D\ D'\ \mathcal{C}_1\ \mathcal{C}'_1\ p & : \mathsf{Closure}^{\mathfrak{s}}_1 \\
 & & & \quad \mid \mathsf{CastPi}\ \mathfrak{s}\ D\ D'\ \mathcal{C}_1\ \mathcal{C}'_1\ p\ f & : \mathsf{Closure}^{\mathfrak{s}}_1 \\
\mathsf{Closure}^{\mathcal{U}}_k & \ni & \mathcal{A}_k, \mathcal{B}_k, \mathcal{F}_k & & \\
\mathsf{Closure}^{\Omega}_k & \ni & \mathcal{P}_k, \mathcal{Q}_k & &
\end{array}
$$

Figure 3.3: Semantic domains, environments and closures.

D^{ne} represent proof-relevant values which reduce. The domain D^{Ω} represents propositional values, which are handled in Section 3.2.3. D is the union of $\mathsf{D}^{\mathcal{U}}$ and D^{Ω}, and we often omit the injections V and P.

The neutral terms for observational equality and casting given in Section 3.1 are simplified in semantic neutrals D^{ne} – this is a tradeoff 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.

In Section 2.5.1, closures had a uniform representation of an environment and a term. Here, there are multiple forms used to defunctionalise various reductions. We annotate closures with their arity, as not all closures await a single argument. We use $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{F}$ for closures with relevant codomain, and $\mathcal{P}$ and $\mathcal{Q}$ for irrelevant codomain.

The precondition for relevant interpretation is well-typed terms of sort $\mathcal{U}$; that is, the set $\mathsf{Tm}^{\mathcal{U}}$. Therefore, we construct the function

$$[\![\_]\!]^{\mathcal{U}}\_ : \mathsf{Tm}^{\mathcal{U}} \to \mathsf{Env} \to \mathsf{D}^{\mathcal{U}}$$

We discuss interpretation of propositions, $[\![\_]\!]^{\Omega}\_$, in Section 3.2.3. The interpretation is given in Figure 3.4. The underlined functions and $\cdot$ are auxilliary functions handling reduction steps for

$$\begin{aligned}
[\![x_0]\!]^{\mathcal{U}}(\rho, a) &= a \\
[\![x_{i+1}]\!]^{\mathcal{U}}(\rho, d) &= [\![x_i]\!]^{\mathcal{U}}\rho \\
[\![\mathfrak{s}]\!]^{\mathcal{U}}\rho &= \mathsf{U}\ \mathfrak{s} \\
[\![\lambda x_{\mathfrak{s}}.\ t]\!]^{\mathcal{U}}\rho &= \mathsf{Lam}\ \mathfrak{s}\ (\underline{\lambda} t)\rho \\
[\![t\ u^{\mathfrak{s}}]\!]^{\mathcal{U}}\rho &= ([\![t]\!]^{\mathcal{U}}\rho) \cdot ([\![u]\!]^{\mathfrak{s}}\rho) \\
[\![\Pi(x :_{\mathfrak{s}} A).\ B]\!]^{\mathcal{U}}\rho &= \mathsf{Pi}\ \mathfrak{s}\ ([\![A]\!]^{\mathcal{U}}\rho)\ (\underline{\lambda} B)\rho \\
[\![0]\!]^{\mathcal{U}}\rho &= \mathsf{Z} \\
[\![S\ t]\!]^{\mathcal{U}}\rho &= \mathsf{S}\ ([\![t]\!]^{\mathcal{U}}\rho) \\
[\![\mathbf{rec}[z.C](t, 0 \to t_0; (S\ x)\ y \to t_S)]\!]^{\mathcal{U}}\rho &= \underline{\mathsf{rec}}((\underline{\lambda} C)\rho, [\![t]\!]^{\mathcal{U}}\rho, [\![t_0]\!]^{\mathcal{U}}\rho, (\underline{\lambda} t_S)\rho) \\
[\![\mathbb{N}]\!]^{\mathcal{U}}\rho &= \mathsf{Nat} \\
[\![\exists(x : A).\ B]\!]^{\mathcal{U}}\rho &= \mathsf{Exists}\ ([\![A]\!]^{\mathcal{U}}\rho)\ (\underline{\lambda} B)\rho \\
[\![\mathbf{abort}_A\ t]\!]^{\mathcal{U}}\rho &= \uparrow(\mathsf{Abort}\ ([\![A]\!]^{\mathcal{U}}\rho)\ ([\![t]\!]^{\Omega}\rho)) \\
[\![\bot]\!]^{\mathcal{U}}\rho &= \mathsf{Empty} \\
[\![\top]\!]^{\mathcal{U}}\rho &= \mathsf{Unit} \\
[\![t \sim_A u]\!]^{\mathcal{U}}\rho &= \underline{\mathsf{eq}}([\![t]\!]^{\mathcal{U}}\rho, [\![A]\!]^{\mathcal{U}}\rho, [\![u]\!]^{\mathcal{U}}\rho) \\
[\![\mathbf{cast}(A, B, e, t)]\!]^{\mathcal{U}}\rho &= \underline{\mathsf{cast}}([\![A]\!]^{\mathcal{U}}\rho, [\![B]\!]^{\mathcal{U}}\rho, [\![e]\!]^{\Omega}\rho, [\![t]\!]^{\mathcal{U}}\rho) \\
[\![\mathbf{let}\ x :_{\mathfrak{s}} A = t\ \mathbf{in}\ u]\!]^{\mathcal{U}}\rho &= [\![u]\!]^{\mathcal{U}}(\rho, [\![t]\!]^{\mathfrak{s}}\rho) \\
[\![(t : A)]\!]^{\mathcal{U}}\rho &= [\![t]\!]^{\mathcal{U}}\rho
\end{aligned}$$

Figure 3.4: Semantic interpretation for relevant terms into $\mathsf{D}^{\mathcal{U}}$.

eliminators applied to introduction forms, defined mutually with evaluation.

We also need a function

$$\mathsf{app}^{\mathcal{U}} : \mathsf{Closure}_k^{\mathcal{U}} \to \underbrace{\mathsf{D} \to \cdots \to \mathsf{D}}_{k} \to \mathsf{D}^{\mathcal{U}}$$

for applying a closure to k values of either sort. Applying arguments of the correct sort is a semantic invariant and not statically type safe. This could be formalised more precisely with closures annotated with a type-level list of sorts. We use the shorthand notation $\mathcal{C}[d_1, \ldots, d_k]$ for $\mathsf{app}^{\mathcal{U}}\ \mathcal{C}\ d_1\ \ldots\ d_k$. Closure application intuitively corresponds to substitution.

As many closures are used to defunctionalise specific operations, we define those alongside their associated usage. To begin with, we give the simple generic cases of a continuation, $(\underline{\lambda} t)\rho$, and Lift, which lifts a value into a constant closure which ignores each argument. These are the primary closures used throughout.

$$\begin{aligned}\mathsf{app}^{\mathcal{U}}\ (\underline{\lambda} t)\rho\ d_1\ \ldots\ d_k &= [\![t]\!]^{\mathcal{U}}(\rho, d_1, \cdots, d_k)\\ \mathsf{app}^{\mathcal{U}}\ (\mathsf{Lift}\ a)\ d_1\ \ldots\ d_k &= a\end{aligned}$$

Application $\_ \cdot \_ : \mathsf{D}^{\mathcal{U}} \to \mathsf{D} \to \mathsf{D}^{\mathcal{U}}$ is defined similarly to Section 2.5.1.

$$\begin{aligned}(\mathsf{Lam}\ \mathfrak{s}\ \mathcal{F}) \cdot d &= \mathcal{F}[d]\\ (\uparrow e) \cdot d &= \uparrow(\mathsf{App}\ e\ d)\end{aligned}$$

Next, we introduce the semantics of natural number induction. This computes iteratively on the structure of the number, and blocks when the principal argument is a neutral.

$$\begin{aligned}\underline{\mathsf{rec}}(\mathcal{A}, \mathsf{Z}, a_0, \mathcal{F}_S) &= a_0\\ \underline{\mathsf{rec}}(\mathcal{A}, \mathsf{S}\ n, a_0, \mathcal{F}_S) &= \mathcal{F}_S[n, \underline{\mathsf{rec}}(\mathcal{A}, n, a_0, \mathcal{F}_S)]\\ \underline{\mathsf{rec}}(\mathcal{A}, \uparrow e, a_0, \mathcal{F}_S) &= \uparrow(\mathsf{Rec}\ \mathcal{A}\ e\ a_0\ \mathcal{F}_S)\end{aligned}$$

3.2.2 Equality and casting

To complete the NbE semantic interpretation of $\mathrm{TT}^{\mathrm{obs}}$, we define the reduction semantics for equality types and casts.

In $\mathrm{TT}^{\mathrm{obs}}$, propositions are mechanically broken down so propositions can be proven by their parts. Similarly, existing proofs can be decomposed back into their components. As proofs themselves are irrelevant, the power of proof manipulation relies on the reduction of *propositions*. Reduction of equality types is implemented by the $\underline{\mathsf{eq}}$ function, which we give mutually with the relevant closure application cases in Figure 3.5.

Equality reduction implements an equality decision procedure for natural numbers. Equality at Π-types implements function extensionality. Note that the reductino computation lives inside the closure continuation. The case for irrelevant types implements propositional extensionality, by stepping to a pair of implications $A \leftrightarrow B$.

Equality between relevant types in $\mathcal{U}$ is handled by cases. For natural numbers and universes, this is a trivial conjunction of zero components, hence the unit type. For equality between Π-types, this is a dependent conjunction of a proof the domains are equal, and a proof that the codomains

$$
\begin{aligned}
\underline{\mathsf{eq}}(f, \mathsf{Pi}\ \mathfrak{s}\ A\ \mathcal{B}, g) &= \mathsf{Pi}\ A\ (\mathsf{EqFun}\ \mathfrak{s}\ f\ \mathcal{B}\ g) \\
\underline{\mathsf{eq}}(A, \mathsf{U}\ \Omega, B) &= \mathsf{Exists}\ (\mathsf{Pi}\ A\ (\mathsf{Lift}\ B))\ (\mathsf{Lift}\ (\mathsf{Pi}\ B\ (\mathsf{Lift} A))) \\
\underline{\mathsf{eq}}(\mathsf{Nat}, \mathsf{U}\ \mathcal{U}, \mathsf{Nat}) &= \mathsf{Unit} \\
\underline{\mathsf{eq}}(\mathsf{U}\ \mathfrak{s}, \mathsf{U}\ \mathcal{U}, \mathsf{U}\ \mathfrak{s}) &= \mathsf{Unit} \\
\underline{\mathsf{eq}}(A, \mathsf{U}\ \mathcal{U}, B) &= \mathsf{Empty} \quad \text{where } \mathrm{hd}(A) \neq \mathrm{hd}(B) \\
\underline{\mathsf{eq}}(\mathsf{Pi}\ \mathfrak{s}\ A\ \mathcal{B}, \mathsf{U}\ \mathcal{U}, \mathsf{Pi}\ \mathfrak{s}\ A'\ \mathcal{B}') &= \mathsf{Exists}\ (\underline{\mathsf{eq}}(A', \mathsf{U}\ \mathfrak{s}, A))\ (\mathsf{EqPi}\ \mathfrak{s}\ A\ A'\ \mathcal{B}\ \mathcal{B}') \\
\underline{\mathsf{eq}}(\mathsf{Z}, \mathsf{Nat}, \mathsf{Z}) &= \mathsf{Unit} \\
\underline{\mathsf{eq}}(\mathsf{S}\ a, \mathsf{Nat}, \mathsf{S}\ b) &= \underline{\mathsf{eq}}(a, \mathsf{Nat}, b) \\
\underline{\mathsf{eq}}(\mathsf{S}\ a, \mathsf{Nat}, \mathsf{Z}) &= \mathsf{Empty} \\
\underline{\mathsf{eq}}(\mathsf{Z}, \mathsf{Nat}, \mathsf{S}\ a) &= \mathsf{Empty} \\
\underline{\mathsf{eq}}(a, A, a') &= \uparrow(\mathsf{Eq}\ a\ A\ a')
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{app}^{\mathcal{U}}\ (\mathsf{EqFun}\ \mathfrak{s}\ f\ \mathcal{B}\ g)\ d &= \underline{\mathsf{eq}}(f \cdot d, \mathcal{B}[d], g \cdot d) \\
\mathsf{app}^{\mathcal{U}}\ (\mathsf{EqPi}\ \mathfrak{s}\ A\ A'\ \mathcal{B}\ \mathcal{B}')\ p &= \mathsf{Pi}\ \mathfrak{s}\ A'\ (\mathsf{EqPi}'\ \mathfrak{s}\ A\ A'\ \mathcal{B}\ \mathcal{B}'\ p) \\
\mathsf{app}^{\mathcal{U}}\ (\mathsf{EqPi}'\ \mathcal{U}\ A\ A'\ \mathcal{B}\ \mathcal{B}'\ p)\ a' &= \underline{\mathsf{eq}}(\mathcal{B}[a], \mathsf{U}\ \mathcal{U}, \mathcal{B}'[a']) \quad \text{where } a \triangleq \underline{\mathsf{cast}}(A', A, p, a') \\
\mathsf{app}^{\mathcal{U}}\ (\mathsf{EqPi}'\ \Omega\ A\ A'\ \mathcal{B}\ \mathcal{B}'\ p)\ q' &= \underline{\mathsf{eq}}(\mathcal{B}[q], \mathsf{U}\ \mathcal{U}, \mathcal{B}'[q']) \quad \text{where } q \triangleq \mathsf{PCast}\ \phi(A')\ \phi(A)\ p\ q'
\end{aligned}
$$

Figure 3.5: Equality proposition reduction function.

are equal. This time, we need *two* defunctionalising closures – 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 Π-types have an irrelevant domain, we need a propositional witness of type A, so we use PCast and the *freeze* function $\phi : \mathsf{D} \hookrightarrow \mathsf{D}^{\Omega}$ for embedding relevant values into the irrelevant domain, both of which will be introduced in Section 3.2.3.

The last case in $\underline{\mathsf{eq}}$ is a fall-through case applicable only when no other case matches. Typing invariants mean we only reach this when at least one position is blocked.

The final component of evaluation is the $\underline{\mathsf{cast}}$ function.

$$\begin{aligned}
\underline{\mathsf{cast}}(\mathsf{Nat}, \mathsf{Nat}, p, \mathsf{Z}) &= \mathsf{Z} \\
\underline{\mathsf{cast}}(\mathsf{Nat}, \mathsf{Nat}, p, \mathsf{S}\ a) &= \mathsf{S}\ (\underline{\mathsf{cast}}(\mathsf{Nat}, \mathsf{Nat}, p, a)) \\
\underline{\mathsf{cast}}(\mathsf{U}\ \mathfrak{s}, \mathsf{U}\ \mathfrak{s}, p, A) &= A \\
\underline{\mathsf{cast}}(\mathsf{Pi}\ \mathfrak{s}\ A\ \mathcal{B}, \mathsf{Pi}\ \mathfrak{s}\ A'\ \mathcal{B}', p, f) &= \mathsf{Lam}\ \mathfrak{s}\ (\mathsf{CastPi}\ \mathfrak{s}\ A\ A'\ \mathcal{B}\ \mathcal{B}'\ p\ f) \\
\underline{\mathsf{cast}}(A, B, p, a) &= \uparrow(\mathsf{Cast}\ A\ B\ p\ a)
\end{aligned}$$

$$\begin{aligned}
\mathsf{app}^{\mathcal{U}}\ (\mathsf{CastPi}\ \mathcal{U}\ A\ A'\ \mathcal{B}\ \mathcal{B}'\ p\ f)\ a' &= \underline{\mathsf{cast}}(\mathcal{B}[a], \mathcal{B}'[a'], \mathsf{PApp}\ (\mathsf{PSnd}\ p)\ \phi(a'), f \cdot a) \\
&\quad \text{where } a \triangleq \underline{\mathsf{cast}}(A', A, \mathsf{PFst}\ p, a') \\
\mathsf{app}^{\mathcal{U}}\ (\mathsf{CastPi}\ \Omega\ A\ A'\ \mathcal{B}\ \mathcal{B}'\ p\ f)\ q' &= \underline{\mathsf{cast}}(\mathcal{B}[q], \mathcal{B}'[q'], \mathsf{PApp}\ (\mathsf{PSnd}\ p)\ q', f \cdot q) \\
&\quad \text{where } q \triangleq \mathsf{PCast}\ A'\ A\ (\mathsf{PFst}\ p)\ q'
\end{aligned}$$

Reduction for natural numbers proceeds by iteration on the structure when both types resolve to $\mathbb{N}$, and the argument is a constructor. Casting a function between two Π-types produces a new λ-term which casts the argument from A' to A, calls the original function f and then casts the result back to $\mathcal{B}'[a']$. Note the proof manipulation implemented by propositional application and projection terms, and the embedding $\phi : \mathsf{D} \hookrightarrow \mathsf{D}^{\Omega}$; more details in Section 3.2.3. Like before, we give a fall-through case for when the cast is blocked.

3.2.2.1 Quoting

Quoting is defined as described in Section 2.5.2. We give two mutually recursive functions $\mathsf{q}_{\mathsf{n}}^{\mathsf{Nf}} : \mathsf{D}^{\mathcal{U}} \to \mathsf{Nf}$ and $\mathsf{q}_{\mathsf{n}}^{\mathsf{Ne}} : \mathsf{D}^{\mathsf{ne}} \to \mathsf{Ne}$ for quoting the domain back into normal and neutral forms at de Bruijn level n. Quoted terms have de Bruijn indices, so there are no explicit binder names included, however in practice these names are preserved for printing human-readable strings. Proposition quoting, q_{n}, is similar, so we omit it here.

First, we define a helper function $\mathsf{vs}_{\mathsf{n}}^{\mathcal{U}} : \mathsf{Closure}_k \to \mathsf{D}^{\mathcal{U}}$ to fully apply a closure to k fresh semantic variables.

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

With this, we define quoting in Figure 3.6.

This function 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 lands in Ne.

3.2.3 Propositional layer

Inhabitants of propositions living in universe Ω do not reduce. By removing the computational behaviour of propositional proofs, we treat propositions as *proof-irrelevant*, caring only whether an inhabitant exists, and not the data it contains. This is reflected in evaluation by never reducing irrelevant terms.

$$
\begin{aligned}
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{U}\ \mathfrak{s}) &= \mathfrak{s} \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{Lam}\ \mathfrak{s}\ \mathcal{F}) &= \lambda_\mathfrak{s}.\ \mathsf{q}_{\mathsf{n}+1}^\mathsf{Nf}(\mathsf{vs}_\mathsf{n}^\mathcal{U}(\mathcal{F})) \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{Pi}\ \mathfrak{s}\ A\ \mathcal{B}) &= \Pi\ \mathfrak{s}\ \left(\mathsf{q}_\mathsf{n}^\mathsf{Nf}(A)\right).\ \left(\mathsf{q}_{\mathsf{n}+1}^\mathsf{Nf}(\mathsf{vs}_\mathsf{n}^\mathcal{U}(\mathcal{B}))\right) \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{Z}) &= 0 \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{S}\ a) &= S\ (\mathsf{q}_\mathsf{n}^\mathsf{Nf}(a)) \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{Nat}) &= \mathbb{N} \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{Exists}\ A\ \mathcal{B}) &= \exists\ \left(\mathsf{q}_\mathsf{n}^\mathsf{Nf}(A)\right).\ \left(\mathsf{q}_{\mathsf{n}+1}^\mathsf{Nf}(\mathcal{B}[\uparrow \mathsf{PVar}_n])\right) \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{Empty}) &= \bot \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\mathsf{Unit}) &= \top \\
\mathsf{q}_\mathsf{n}^\mathsf{Nf}(\uparrow e) &= \mathsf{q}_\mathsf{n}^\mathsf{Ne}(e)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{q}_\mathsf{n}^\mathsf{Ne}(\mathsf{Var}_l) &= x_{n-l-1} \\
\mathsf{q}_\mathsf{n}^\mathsf{Ne}(\mathsf{App}\ e\ a) &= (\mathsf{q}_\mathsf{n}^\mathsf{Ne}(e))\ (\mathsf{q}_\mathsf{n}^\mathsf{Nf}(a)) \\
\mathsf{q}_\mathsf{n}^\mathsf{Ne}(\mathsf{Rec}\ \mathcal{A}\ e\ a_0\ \mathcal{F}_S) &= \mathbf{rec}[\mathsf{q}_{\mathsf{n}+1}^\mathsf{Nf}(\mathsf{vs}_\mathsf{n}^\mathcal{U}(\mathcal{A}))](\mathsf{q}_\mathsf{n}^\mathsf{Ne}(e), \mathsf{q}_\mathsf{n}^\mathsf{Nf}(a_0); \mathsf{q}_{\mathsf{n}+2}^\mathsf{Nf}(\mathsf{vs}_\mathsf{n}^\mathcal{U}(\mathcal{F}_S))) \\
\mathsf{q}_\mathsf{n}^\mathsf{Ne}(\mathsf{Abort}\ A\ p) &= \mathbf{abort}_{\mathsf{q}_\mathsf{n}^\mathsf{Nf}(A)}\ (q_n^\Omega(p)) \\
\mathsf{q}_\mathsf{n}^\mathsf{Ne}(\mathsf{Eq}\ a\ A\ a') &= \mathsf{q}_\mathsf{n}^\mathsf{Nf}(a) \sim_{\mathsf{q}_\mathsf{n}^\mathsf{Nf}(A)} \mathsf{q}_\mathsf{n}^\mathsf{Nf}(a') \\
\mathsf{q}_\mathsf{n}^\mathsf{Ne}(\mathsf{Cast}\ A\ B\ p\ a) &= \mathbf{cast}(\mathsf{q}_\mathsf{n}^\mathsf{Nf}(A), \mathsf{q}_\mathsf{n}^\mathsf{Nf}(B), q_n^\Omega(p), \mathsf{q}_\mathsf{n}^\mathsf{Nf}(a))
\end{aligned}
$$

Figure 3.6: Implementation of quoting for $\mathrm{TT}^{\mathrm{obs}}$

3.2.3.1 Proof erasure

Inhabitants of propositions live solely for the purpose of type-checking. Well-typed terms are a precondition for evaluation, so one strategy for handling proof-irrelevance is to *erase* irrelevant terms at evaluation, as done in [8]. Practically, this amounts to introducing a single semantic proposition. Therefore, we define $\mathsf{D}^\Omega ::= \mathsf{Witness}$ as a type with one constructor, so all semantic proofs contain no data. This simplifies the implementation, as many reduction rules in the system are made complex by the intricate proof manipulations to ensure the reduced term is well-typed.

Unfortunately, this technique poses problems when quoting: all irrelevant information is erased, so there is no hope of recoving arbitrary proofs. One solution is to indicate in the quoted term where proofs exist, but leaving the witness blank. This solution falls short, as the quoted term is no longer typeable, which is particularly important for pattern unification (Section 3.6).

3.2.3.2 Syntactic propositions

A natural alternative is to retain *syntactic* witnesses for proof terms during evaluation. The motivation is that in NbE, reduction only occurs in semantic values, so the syntactic witnesses are never reduced. For this, we define $\mathsf{D}^\Omega ::= \mathsf{Prop}\ t\ \rho$ – a witness t and an environment ρ interpreting the free variables in t.

This solution is still somewhat problematic, despite solving the problem with proof erasure.

While they do not reduce, propositions still admit *substitutions* of their free variables which must be accounted for. This is achieved using the closure of values, which are inserted in place of free variables during quoting.

More challenging is the proof manipulation which occurs in casting reduction rules. This requires inserting proof-relevant terms into the proof witness and shifting propositions to be well-typed in a different context. Therefore, evaluation depends mutually on quoting, and care must be taken to transform witnesses correctly. This solution is error-prone and unsatisfying, as we would prefer evaluation and quoting not to become mutually dependent on each other.

3.2.4 Semantic propositions

Taking inspiration from the NbE treatment of proof-relevant values, we introduce a novel idea for propositions designed to overcome the problems mentioned above. Instead of dealing with syntactic proof witnesses, we create another semantic domain similar to $\mathsf{D}^{\mathcal{U}}$. The idea is that values in this domain never reduce, for example when an argument is applied to a λ-expression. The only admitted reduction is *substitution*, which is handled by closures. Semantic propositions use de Bruijn levels rather than indices, so it is never necessary to shift or quote terms during evaluation. We note that semantic propositions also include embedded relevant terms, as relevant data might appear as subterms in propositions. However, these subterms still never reduce.

The domain of semantic propositions has a similar structure to terms, except we introduce explicit closures whenever there are bound variables, and use de Bruijn levels for variables.

$$
\begin{array}{lcll}
\mathsf{D}^{\Omega} & \ni & P, Q, p, q, e & ::= \quad \mathsf{PVar}_l \mid \mathsf{PU}\ \mathfrak{s} \mid \mathsf{PLam}\ \mathfrak{s}\ \mathcal{P}_1 \mid \mathsf{PApp}\ p\ q \mid \mathsf{PPi}\ \mathfrak{s}\ P\ \mathcal{Q}_1 \\
& & & \quad \mid \mathsf{PZ} \mid \mathsf{PS}\ p \mid \mathsf{PRec}\ \mathcal{Q}_1\ p\ q\ \mathcal{P}_2 \mid \mathsf{PNat} \\
& & & \quad \mid \mathsf{PPair}\ p\ q \mid \mathsf{PFst}\ p \mid \mathsf{PSnd}\ p \mid \mathsf{PExists}\ P\ \mathcal{Q}_1 \\
& & & \quad \mid \mathsf{PAbort}\ P\ p \mid \mathsf{PEmpty} \mid \mathsf{POne} \mid \mathsf{PUnit} \mid \mathsf{PRefl}\ p \\
& & & \quad \mid \mathsf{PTransp}\ p\ \mathcal{Q}_2\ q\ p'\ e \mid \mathsf{PEq}\ p\ P\ p' \\
& & & \quad \mid \mathsf{PCast}\ P\ Q\ e\ p \mid \mathsf{PLet}\ P\ p\ q \mid \mathsf{PAnn}\ p\ P
\end{array}
$$

We note even let bindings and type annotations have representation in semantic propositions. Calligraphic letters represent closures, which are the same as in Section 3.2.1, but have values from D^{Ω} in place of $\mathsf{D}^{\mathcal{U}}$.

Next, we introduce the inclusion $\phi : \mathsf{D}^{\mathcal{U}} \hookrightarrow \mathsf{D}^{\Omega}$, which we think of as *freezing* semantic values into propositions so they may be used in proof terms. ϕ is defined mutually with $\Phi_k : \mathsf{Closure}_k^{\mathcal{U}} \twoheadrightarrow \mathsf{Closure}_k^{\Omega}$ which embeds closures. The inclusion is natural, so we give only a few cases in Figure 3.7.

Semantic interpretation for propositions, $[\![\_]\!]^{\Omega}\_ : \mathsf{Tm} \twoheadrightarrow \mathsf{Env} \twoheadrightarrow \mathsf{D}^{\Omega}$, is particularly easy as there are no reductions, apart from substitutions. We give only a few cases in the hope the rest

$$
\begin{aligned}
\phi(\uparrow e) &= \phi^{\mathsf{Ne}}(e) \\
\phi(\mathsf{U}\ \mathfrak{s}) &= \mathsf{PU}\ \mathfrak{s} \\
\phi(\mathsf{Lam}\ \mathfrak{s}\ \mathcal{F}) &= \mathsf{PLam}\ \mathfrak{s}\ (\Phi_1(\mathcal{F})) \\
\phi(\mathsf{Exists}\ \mathcal{B}) &= \mathsf{PExists}\ (\Phi_1(\mathcal{B})) \\
\phi(\mathsf{Prop}\ p) &= p
\end{aligned}
$$

$$
\begin{aligned}
\phi^{\mathsf{Ne}}(\mathsf{Var}_l) &= \mathsf{PVar}_l \\
\phi^{\mathsf{Ne}}(\mathsf{App}\ e\ a) &= \mathsf{PApp}\ (\phi^{\mathsf{ne}}(e))\ (\phi(a)) \\
\phi^{\mathsf{Ne}}(\mathsf{App}\ e\ p) &= \mathsf{PApp}\ (\phi^{\mathsf{ne}}(e))\ p \\
\phi^{\mathsf{Ne}}(\mathsf{Rec}\ \mathcal{A}\ e\ a_0\ \mathcal{F}_S) &= \mathsf{PRec}\ (\Phi_1(\mathcal{A}))\ (\phi^{\mathsf{ne}}(e))\ (\phi(a_0))\ (\Phi_2(\mathcal{F}_S))
\end{aligned}
$$

$$
\begin{aligned}
\Phi_k((\underline{\lambda}t)\rho) &= (\underline{\lambda}t)\rho \\
\Phi_k(\mathsf{Lift}\ a) &= \mathsf{Lift}\ (\phi(a)) \\
\Phi_1(\mathsf{EqFun}\ \mathfrak{s}\ f\ \mathcal{B}\ g) &= \mathsf{EqFun}\ \mathfrak{s}\ (\phi(f))\ (\Phi(\mathcal{B}_1))\ (\phi(g))
\end{aligned}
$$

Figure 3.7: Partial implementation of freeze embedding $\phi : \mathsf{D}^{\mathcal{U}} \hookrightarrow \mathsf{D}^{\Omega}$

are obvious.

$$
\begin{aligned}
[\![x_0]\!]^{\Omega}(\rho, \mathsf{P}\ p) &= p \\
[\![x_0]\!]^{\Omega}(\rho, \mathsf{V}\ a) &= \phi(a) \\
[\![x_{i+1}]\!]^{\Omega}(\rho, d) &= [\![x_i]\!]^{\Omega}\rho \\
[\![\lambda x_{\mathfrak{s}}.\ t]\!]^{\Omega}\rho &= \mathsf{PLam}\ \mathfrak{s}\ (\underline{\lambda}t)\rho \\
[\![t\ u]\!]^{\Omega}\rho &= \mathsf{PApp}\ ([\![t]\!]^{\Omega}\rho)\ ([\![u]\!]^{\Omega}\rho)
\end{aligned}
$$

Projections from the environment are like in relevant interpretation, only when the entry is relevant, we freeze it with $\phi : \mathsf{D}^{\mathcal{U}} \twoheadrightarrow \mathsf{D}^{\Omega}$. This handles substitution – syntactic variables x_i are substituted by values from the environment. λ-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 $\mathsf{app}^{\Omega} : \mathsf{Closure}_k^{\Omega} \twoheadrightarrow \mathsf{D} \twoheadrightarrow \cdots \twoheadrightarrow \mathsf{D} \twoheadrightarrow \mathsf{D}^{\Omega}$ for applying closures. This works similarly to $\mathsf{app}^{\mathcal{U}}$, only no reduction occurs. For example, consider the case for EqFun

$$\mathsf{app}\ \ (\mathsf{EqFun}\ \mathfrak{s}\ p\ \mathcal{Q}\ p')\ q = \mathsf{PEq}\ (\mathsf{PApp}\ p\ q)\ \mathcal{Q}[q]\ (\mathsf{PApp}\ p'\ q)$$

where we replace each reduction operator by a constructor, compared to the relevant interpretation (Figure 3.5).

Finally, we also have quoting for semantic propositions. This computes in the same way as quoting for $\mathsf{D}^{\mathcal{U}}$, by fully applying closures to fresh variables. We omit the details of this, and point instead to the code.

3.3 Type system

Central to the implementation of $\mathrm{TT}^{\mathrm{obs}}$ is the type-checker. We implement a *bidirectional type-checker* [15] which either infers a type for a term or checks a term against a type. The choice between these strategies is driven by the syntax of the term. Bidirectional type checking also naturally determines when to invoke conversion checking.

3.3.1 Conversion checking

Conversion checking is used to decide the conversion relation $\equiv$, operating on semantic values.

We pointed out in Section 2.5 that our NbE does not perform η-expansion. We still want to check η-equality (for negative types), so this is implemented as part of conversion checking. Conversion checking is term-directed, so η-expansion is triggered when one side of the conversion equation has the canonical introduction form for a given type, and the other is neutral. We give the following example for Π types

$$\textsc{Π-η-Conv}\quad \frac{\Gamma, x \vdash \mathcal{F}[\mathsf{Var}^{\mathfrak{s}}_{|\Gamma|}] \equiv t' \cdot \mathsf{Var}^{\mathfrak{s}}_{|\Gamma|}}{\Gamma \vdash \mathsf{Lam}\ \mathfrak{s}\ \mathcal{F} \equiv t'}$$

Note that conversion checking is *untyped*, and Γ is in practice a de Bruijn level representing the context length. We apply the closure $\mathcal{F}$ to the fresh variable. Then, we use the $\_ \cdot \_$ operator to apply the same variable to the right hand side, and compare the results.

Conversion checking between irrelevant terms always succeeds. In fact, algorithmically we only define conversion checking between values in $\mathsf{D}^{\mathcal{U}}$. The only rule invoking conversion compares two types, which always have sort $\mathcal{U}$. Therefore, proof irrelevance is implemented by *not comparing irrelevant terms*. Consider the following two conversion rules for applications.

$$\textsc{App-$\mathcal{U}$-Conv}\quad \frac{\Gamma \vdash \uparrow e \equiv \uparrow e' \qquad \Gamma \vdash a \equiv a'}{\Gamma \vdash \uparrow(\mathsf{App}\ e\ a) \equiv \uparrow(\mathsf{App}\ e'\ a')} \qquad \textsc{App-Ω-Conv}\quad \frac{\Gamma \vdash \uparrow e \equiv \uparrow e'}{\Gamma \vdash \uparrow(\mathsf{App}\ e\ p) \equiv \uparrow(\mathsf{App}\ e'\ p')}$$

When the argument is irrelevant, there is no check that $p \equiv p'$, as it holds automatically.

We mentioned in Section 2.2.4 that we implement the definitional casting rule

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

in conversion checking.

The implementation uses the following rules.

$$\textsc{Cast-Eq-Left}\quad \frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash a \equiv a'}{\Gamma \vdash \mathsf{Cast}\ A\ B\ e\ a \equiv a'} \qquad \textsc{Cast-Eq-Right}\quad \frac{\Gamma \vdash A' \equiv B' \qquad \Gamma \vdash a \equiv a'}{\Gamma \vdash a \equiv \mathsf{Cast}\ A'\ B'\ e'\ a'}$$

$$\textsc{Cast-Conv}\quad \frac{\Gamma \vdash A \equiv A' \qquad \Gamma \vdash B \equiv B' \qquad \Gamma \vdash a \equiv a'}{\Gamma \vdash \mathsf{Cast}\ A\ B\ e\ a \equiv \mathsf{Cast}\ A'\ B'\ e'\ a'}$$

These rules require backtracking. When the left hand side of the equality is a cast between A and B, we first check if $A \equiv B$. If this check fails, we proceed with Cast-Eq-Right, and if that fails, Cast-Conv. This backtracking is necessary when comparing two casts – it is impossible to know a priori which strategy to attempt first. With these rules, we allow this definitional equation without breaking the determinism of evaluation.

3.3.2 Bidirectional type-checker

A bidirectional type-checker is an algorithmic method used to implement typing rules. In the declarative type system, we used the judgement $\Gamma \vdash t : A$, but here we instead use *two* judgements

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

Besides the context Γ, we have an environment ρ interpreting Γ. This is because for some rules we need to evaluate terms during type-checking. Typing always uses *semantic* types, $A \in \mathsf{D}^{\mathcal{U}}$. As well as type-checking, we *elaborate* terms. This maps well-typed pre-terms PreTm to Tm by replacing variables with de Bruijn indices.

Intuitively, certain terms, for example abstractions $\lambda x.\ t$ are easier to *check* so we know how to extend the context to check the body t. Inference is far harder, as the type of x in t is not known a priori. On the other hand, certain terms such as applications $t\ u$ admit *inference*. Checking is difficult, as the return type has insufficient information to check t and u. Alternatively, if we infer that t has type $\Pi(x : A).\ B$, we can check $u : A$, and infer $t\ u : B[u/x]$.

In general, *constructors* are checked, and *destructors* are inferred. From this, we derive algorithmic bidirectional rules from the declarative rules of the type theory. As examples, the rules for abstractions and applications are

$$\frac{\Gamma, x :_{\mathfrak{s}} A; (\rho, \mathsf{Var}^{\mathfrak{s}}_{|\Gamma|}) \vdash t \Leftarrow \mathcal{B}[\mathsf{Var}^{\mathfrak{s}}_{|\Gamma|}]}{\Gamma; \rho \vdash \lambda x.\ t \Leftarrow \mathsf{Pi}\ \mathfrak{s}\ A\ \mathcal{B}}\ \Pi\text{-I-Check} \qquad \frac{\Gamma; \rho \vdash t \Rightarrow \mathsf{Pi}\ \mathfrak{s}\ A\ \mathcal{B} \qquad \Gamma; \rho \vdash u \Leftarrow A}{\Gamma; \rho \vdash t\ u \Rightarrow \mathcal{B}[[\![u]\!]^{\mathfrak{s}}\rho]}\ \Pi\text{-E-Infer}$$

When we extend the context with the variable x in Π-I-Check, we also extend the environment with a fresh variable (we let $\mathsf{Var}^{\mathfrak{s}}$ mean either Var or PVar depending on the sort). We apply the closure $\mathcal{B}$ to this variable, giving a type to check against. When inferring the application, we evaluate u in ρ, and apply the closure $\mathcal{B}$.

Bidirectional rules also determine precisely when to invoke conversion checking. In particular, all inferred terms can also be checked. We construct the rule

$$\frac{\Gamma; \rho \vdash t \Rightarrow A' \qquad \Gamma \vdash A \equiv A'}{\Gamma; \rho \vdash t \Leftarrow A}\ \text{Conv-Check}$$

This rule says to check such a term, we first infer a type for it then check the two are convertible.

The appeal of bidirectional rules is that they are simple to implement. In a sense, declarative rules have an implicit existential quantification over types required for the derivation, however a bidirectional system shows explicitly the source of all data required for each judgement.

The reader may notice that a term like $(\lambda x.\ x)\ t$ is untypeable in this system. It is an application, so it is inferred, but inference of an application requires inferring the type of $\lambda x.\ x$, which must be checked. In particular, we can *only type normal forms*. This is standard in many systems, such as Agda [16]. We have a rule for transferring from checking to inference, Conv-Check (where we have an excess of data), but switch from inference to checking (where we have a lack of data) we need an explicit annotation.

To allow more general terms, we use both typed let-bindings and type annotations. Having an explicit annotation allows us to invoke the checking procedure, even if the whole term is being inferred. Consider the following rules

Let-Infer
$$\frac{\Gamma; \rho \vdash A \Leftarrow \mathsf{U}\ \mathfrak{s} \qquad \Gamma; \rho \vdash t \Leftarrow [\![A]\!]^{\mathcal{U}}\rho \qquad \Gamma, x :_{\mathfrak{s}} [\![A]\!]^{\mathcal{U}}\rho; (\rho, [\![t]\!]^{\mathfrak{s}}\rho) \vdash u \Rightarrow B}{\Gamma; \rho \vdash \mathbf{let}\ x :_{\mathfrak{s}} A = t\ \mathbf{in}\ u \Rightarrow B}$$

Let-Check
$$\frac{\Gamma; \rho \vdash A \Leftarrow \mathsf{U}\ \mathfrak{s} \qquad \Gamma; \rho \vdash t \Leftarrow [\![A]\!]^{\mathcal{U}}\rho \qquad \Gamma, x :_{\mathfrak{s}} [\![A]\!]^{\mathcal{U}}\rho; (\rho, [\![t]\!]^{\mathfrak{s}}\rho) \vdash u \Leftarrow B}{\Gamma; \rho \vdash \mathbf{let}\ x :_{\mathfrak{s}} A = t\ \mathbf{in}\ u \Leftarrow B}$$

Annotation-Infer
$$\frac{\Gamma; \rho \vdash A \Rightarrow \mathsf{U}\ \mathfrak{s} \qquad \Gamma; \rho \vdash t \Leftarrow [\![A]\!]^{\mathcal{U}}\rho}{\Gamma; \rho \vdash (t : A) \Rightarrow [\![A]\!]^{\mathcal{U}}\rho}$$

The first two rules indicate that let-bindings can be checked or inferred, and the current process is then forwarded to the subterm u after the **in** keyword. The term t is *always checked*, allowing us to invoke checking from inference. We check against the type A, which is first evaluated. In both cases, we extend the environment with the *definition* of the variable x, rather than just a generic variable like with λ-terms before. This way, the concrete definition can be used during typing. Annotations behave similarly.

An interesting exception to the rule that constructors are checked, is the term **refl** which constructs a reflexivity proof for equality. This term is hard to check due to the reduction of equality types – equalities frequently reduce to other type constructors. Therefore, reflexivity is inferred.

Refl-Infer
$$\frac{\Gamma; \rho \vdash t \Rightarrow A}{\Gamma; \rho \vdash \mathbf{refl}\ t \Rightarrow \underline{\mathsf{eq}}([\![t]\!]^{\mathcal{U}}\rho, A, [\![t]\!]^{\mathcal{U}}\rho)}$$

We trigger the equality reduction procedure in the inferred type. So, for example, **refl** 0 will infer the type $\top$, as we immediately reduce the type $0 \sim_{\mathbb{N}} 0$.

Under this framework, the bidirectional formulation of rules for $\mathrm{TT}^{\mathrm{obs}}$ is implemented naturally in code.

3.4 Quotient types

Our first extension to the core of $\mathrm{TT}^{\mathrm{obs}}$ is *quotient types*. These types allow us to exploit the setoid structure of types by explicitly defining an equivalence relation over a pre-existing type. The theory of quotient types in $\mathrm{TT}^{\mathrm{obs}}$ is given in Section 2.3

We first extend the syntax with quotient types.

$$\begin{array}{rcl} A, R, t, u & ::= & \cdots \\ & \mid & A/(x\ y.\ R, x\ R_r, x\ y\ xRy.\ R_s, x\ y\ z\ xRy\ yRz.\ R_t) \\ & \mid & \pi\ t \mid \mathbf{Q\text{-}elim}[z.\ B](x.\ t_\pi, x\ y\ xRy.\ t_\sim, u) \end{array}$$

Three new syntactic elements are introduced: formation from a base type and equivalence relation, introduction projecting from the base type into the quotient, and elimination, giving an equality-preserving map out of the quotient type.

Normal and neutral forms are given as follows.

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

3.4.1 NbE for quotient types

Next, we turn to the NbE implementation, which closely follows the same patterns as the core implementation given in Section 3.2.

First, we extend the domains with new semantic values. We also need three new closures to defunctionalise the equality type reduction between quotient types.

$$\begin{array}{rcccll} \mathsf{D}^{\mathcal{U}} & \ni & A, B, a, b & ::= & \cdots & \\ & & & \mid & \mathsf{Quotient}\ A\ \mathcal{B}_2\ \mathcal{P}_1^r\ \mathcal{P}_3^s\ \mathcal{P}_5^t & \\ & & & \mid & \mathsf{Qproj}\ a & \\ \mathsf{D}^{\mathsf{ne}} & \ni & e & ::= & \cdots & \\ & & & \mid & \mathsf{Qelim}\ \mathcal{B}_1\ \mathcal{F}_1\ \mathcal{Q}_3\ e & \\ \mathsf{D}^{\Omega} & \ni & P, Q, p, q & ::= & \cdots & \\ & & & \mid & \mathsf{PQuotient}\ P\ \mathcal{Q}_2\ \mathcal{P}_1^r\ \mathcal{P}_3^s\ \mathcal{P}_5^t & \\ & & & \mid & \mathsf{PQproj}\ p & \\ & & & \mid & \mathsf{PQelim}\ \mathcal{P}_1\ \mathcal{P}_1'\ \mathcal{Q}_3\ p & \\ \mathsf{Closure}_k^{\mathfrak{s}} & \ni & \mathcal{C}_k & ::= & \cdots & \\ & & & \mid & \mathsf{EqQuotientY}\ p\ a\ D\ D'\ \mathcal{C}_2\ \mathcal{C}_2 & : \mathsf{Closure}_1^{\mathfrak{s}} \\ & & & \mid & \mathsf{EqQuotientX}\ p\ D\ D'\ \mathcal{C}_2\ \mathcal{C}_2 & : \mathsf{Closure}_1^{\mathfrak{s}} \\ & & & \mid & \mathsf{EqQuotient}\ D\ D'\ \mathcal{C}_2\ \mathcal{C}_2 & : \mathsf{Closure}_1^{\mathfrak{s}} \end{array}$$

The three closures are needed to defunctionalise the rule Quotient-Eq (Section 2.3.1), as it introduces three binders.

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

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

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

$$\underline{\mathsf{Qelim}}(\mathcal{B}, \mathcal{F}_\pi, \mathcal{P}_\sim, \mathsf{Qproj}\ b) = \mathcal{F}_\pi[b]$$
$$\underline{\mathsf{Qelim}}(\mathcal{B}, \mathcal{F}_\pi, \mathcal{P}_\sim, \uparrow e) = \uparrow(\mathsf{Qelim}\ \mathcal{B}\ \mathcal{F}_\pi\ \mathcal{P}_\sim\ e)$$

Elimination reduces when the argument is of the normal form $\mathsf{Qproj}\ b$ by calling the function $\mathcal{F}_\pi$ with the argument b.

Irrelevant interpretation, $[\![\_]\!]^{\Omega}\_$, is trivial. Once again, there is no reduction when an eliminator is applied to a projection.

Quoting also takes the same form as before. The rule for quoting quotient types is very cumbersome, as we fully apply each closure to fresh variables, so we omit the full rule here. The full details are seen in the code.

3.5 Inductive types and Mendler induction

Our next extension is to add inductive types to $\mathrm{TT}^{\mathrm{obs}}$. Inductive types allow for sum-of-product data types with named constructors. Furthermore, inductive types recursively refer to themselves in constructors, and may be indexed. A detailed account of the theory is given in Section 2.4.

First, we extend the syntax with inductive types.

$$\begin{aligned}
A, B, C, M, t, u \quad ::= \quad & \cdots \\
\mid \quad & \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}] \\
\mid \quad & \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}]\ \mathbf{functor}\ X\ Y\ f\ p\ x = t \\
\mid \quad & C_i\ (t, e) \mid \mathbf{match}\ t\ \mathbf{as}\ x\ \mathbf{return}\ C\ \mathbf{with}\ \{C_i\ (x_i, e_i) \to t_i\}_i \\
\mid \quad & \mathbf{fix}\ [M\ \mathbf{as}\ G]\ f\ p\ x : C = t \mid \mathbf{fix}\ [M\ \mathbf{as}\ G\ \mathbf{view}\ \iota]\ f\ p\ x : C = t \\
\mid \quad & \mathbf{in}\ t \mid \mathbf{lift}[M]\ A \mid \mathbf{fmap}[M](A, B, f, u, t)
\end{aligned}$$

We note that we have parallel versions of inductive type and fixed-points definitions. This is due to practicality considerations, and means it is not necessary to always provide a functor instance when defining a type. This is in fact particularly useful *when* defining functor instances, as we can access the inductive type within the functor definition. Morally, every inductive type must define an underlying functor, however we do not enforce it syntactically. A future extension might include a strict positivity check, from which a functor instance could be automatically derived.

Fixed-points first specify the inductive type they act on. This is given by the term M, which must resolve to an inductive type definition statically. In Mendler induction, we control termination by giving the inductive type an opaque name in recursive calls – this name is made explicit by the variable G. We then have three binders: f, the recursively-bound name of the

function, p the index parameter and x, the value. C represents the induction hypothesis, or return type, and t is the body of the definition.

The second fixed-point construction admits a view parameter ι. When the view is used, the inductive type M must include a functor instance. The view parameter extends the fixed-point to a paramorphism supporting primitive recursion.

Next, we have a term **in** witnessing the defining isomorphism of inductive types. This is useful within fixed-points using views, as lifting $\iota : \Pi(p : A).\ G\ p \to (\mu F)\ p$ to $\iota' : \Pi(p : A).\ F[G]\ p \to (\mu F)\ p$ is defined as the composition of the functorial action on ι and **in**. We also give a term $\mathbf{lift}[M]\ A$, which again requires M be an inductive type. This term is the functor action on objects – it lifts type family A to $M[A]$. Similarly, **fmap** is the action on functions. These three terms, **in**, **lift** and **fmap**, are admissible – each can be avoided by repeating the definition they expand into.

Constructors and pattern matching are as defined in Section 2.4.

We also need to specify the normal and neutral forms. We omit the normal forms for inductive types without functor instances and fixed-points without views; they follow the same shape as those presented.

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

Note that since inductive types form type *families*, they can be applied to an argument. Such applications never reduce; hence also are in normal form.

We described fixed-points as an *elimination* principle for inductive types. However, fixed-points also introduce Π-types. Therefore, a fixed-point in isolation is a normal form. Similarly, when applied to a single index parameter, we also have a normal form. When the final parameter is applied, we *block* if the term is a neutral form. This is to avoid infinite unfolding of syntactic fixed-points.

The **lift** and **fmap** terms block when their inductive type is unknown – we cannot lift type families and functions without the definition at hand. **in** blocks when applied to a neutral.

3.5.1 NbE for inductive types

As usual, the first step for NbE 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 practice, we have optional value for the functor definition).

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

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 application of the inductive type to an index; as noted before, this never reduces. There is a similar structure for fixed-points. Constructors hold both a value and a fording term. D^{Ω} and Closure are also extended, but we omit them here.

The neutral form FixApp represents the case when a neutral argument is applied to a fixed-point. As mentioned before, this does *not* reduce.

We do not explicitly define the semantic propositions here, but as usual we have a semantic proposition for every term which maintain the term structure, but include closures for each binder. We also omit the defunctionalising closures.

The extension of semantic interpretation and evaluation now includes more intricate reduction rules, especially in application reduction. First, we give the semantics of the newly added terms.

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

The implementations of $\underline{\mathsf{match}}$, $\underline{\mathsf{lift}}$, $\underline{\mathsf{fmap}}$ and $\underline{\mathsf{in}}$ are omitted here, but found in the code.

Reduction of fixed points is a special case of *application*, as fixed-points are functions. We extend $\_ \cdot \_$ from Section 3.2.1 as follows.

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

When either a semantic inductive type, or semantic fixed-point have not been applied to their index parameter (indicated by their final component being Nothing), we shift this parameter into the value. When a fixed-point with an index is applied to a constructor, we invoking the

closure $\mathcal{F}$. First, the inductive type is passed in. Then, we pass the identity function, defined as $\underline{\mathrm{id}} \triangleq \underline{\lambda}\_\ a \to a$ in for the view parameter. Next, we pass the whole fixed-point to give $\mathcal{F}$ recursive access to the definition. Finally, we pass the index and the constructor value.

The propositional interpretation $[\![\_]\!]^{\Omega}\_$ and quoting are also extended to support inductive types. Both of these functions are entirely mechanical, so we choose to omit their explicit construction.

3.6 Pattern unification

With the extended type system in place, we turn to a very different part of the implementation. *Pattern unification* [17, 18] does not extend the type theory, but instead makes writing proofs more practical. Often when writing dependently typed programs, many terms are kept purely for bookkeeping purposes, and are uniquely determined from the surrounding context. Pattern unification deduces these unique terms and substitutes them into the code to reduce unnecessary handwritten code.

An example of this phenomenon is the *map* function

$$map : \Pi(A : \mathcal{U}).\ \Pi(B : \mathcal{U}).\ (A \to B) \to List\ A \to List\ B$$

When called, we write

$$map\ A\ B\ f\ ls$$

In particular, we write the types A and B, even if we know the type of f is $A \to B$.

When type-checking this term, we work from left-to-right. First, we infer the type of *map*. We then check each argument in turn. When we reach f, we check against the type $A \to B$. As f is a variable, we infer the type $A \to B$, then check these types are convertible.

Consider instead if the values A and B were omitted, and replaced by unknown term placeholders $?m_1$ and $?m_2$ called *metavariables*. These stand for terms which are currently unknown, but should be solved for. During checking we now attempt the conversion $?m_1 \to ?m_2 \equiv A \to B$. This is satisfied by defining $?m_1 := A$ and $?m_2 := B$.

We rewrite the term as

$$map\ \_\ \_\ f\ ls$$

where _ represents a "hole" to be inferred. A natural extension of this is truly *implicit* parameters which are omitted altogether; we leave this as future work.

3.6.1 The metacontext

Syntactically, we leave *holes* in the program. Each hole represents a *metavariable*.

Metavariables require a new context which records solutions. This context contains both solved and unsolved metavariables. Metacontexts are defined by the following structure:

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

We have entries stating variables exist, and entries mapping variables to their definitions. The definitions are syntactic terms.

As noted, metavariables are solved during conversion checking. Solving metavariables is a *stateful* action. We only ever commit to definitionally unique solutions, so we reuse the first solution we find. The type and conversion checking judgements are now of the form

$$\Sigma; \Gamma; \rho \vdash t \rightsquigarrow t' \Leftarrow A;\ \Sigma' \qquad \Sigma; \Gamma; \rho \vdash t \rightsquigarrow t' \Rightarrow A;\ \Sigma' \qquad \Sigma; \Gamma \vdash t \equiv u;\ \Sigma'$$

where Σ is the initial metacontext, updated to Σ' with new solutions. We also make the elaborated term t' explicit in the judgement. The metacontext is threaded through judgements monadically.

We add an inference rule for holes.

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

To infer a hole, we create two new metavariables, one representing the term, and one representing its type. We add both to the metacontext.

3.6.2 Solving metavariables

To motivate metavariable solving, we first inspect a syntactic reduction system. We relate this to NbE in the next section.

The syntactic form of metavariables is extended to include a *parallel substitution*. A parallel substitution, $\Gamma \vdash \sigma : \Delta$, is a map from context Γ to Δ. Here, σ is a list of terms typed in Γ, one for each variable in Δ. We can act contravariantly on a term $\Delta \vdash t : A$ by replacing each free variable with its definition in σ, giving a term $\Gamma \vdash t[\sigma] : A[\sigma]$ typed in Γ.

We give some cases of substitution on terms.

$$\begin{aligned} x_0[\sigma, t] &= t \\ x_{i+1}[\sigma, t] &= x_i[\sigma] \\ (t\ u)[\sigma] &= (t[\sigma])\ (u[\sigma]) \\ (\lambda x.\ t)[\sigma] &= \lambda x.\ (t[\uparrow \sigma, x_0]) \\ (\Pi(x : A).\ B)[\sigma] &= \Pi(x : A[\sigma]).\ B[\uparrow \sigma, x_0] \end{aligned}$$

Here, $\uparrow$ shifts the substitution pointwise: each term shifts its free de Bruijn indices by one. In the λ and Π cases, we extend the substitution by a variable x_0. This behaves as the identity on nested variables, as σ should not update them.

An example of when substitution is triggered is in reducing an application. we reduce $\Gamma \vdash (\lambda x.\ t)\ u \Rightarrow t[\uparrow \mathrm{id}_\Gamma, u]$.

The motivation behind adding substitutions is that metavariables stand for an unknown term, which might depend on its free variables. So, we maintain a substitution to apply when the term is known. We write $?m_i[\sigma]$ for the metavariable with a captured substitution σ.

At the point the metavariable is created, it is typed under a context Γ. We initialise the

captured substitution to id_Γ: the identity substitution. This is updated when another substitution is applied. We add the following definition to the substitution procedure.

$$m_i[\sigma][\sigma'] =?m_i[\sigma \circ \sigma']$$

Composition of substitutions $\sigma \circ \sigma'$ is defined by replacing free variables in σ by their definitions in σ'.

The substitution σ on a metavariable $?m_i[\sigma]$ has type $\Delta \vdash \sigma : \Gamma$, where Γ is context $?m_i$ was created in, and Δ is the context typing $?m_i[\sigma]$. Therefore, σ is a list of Δ-typed definitions for each free variable in Γ.

Now we consider the case of conversion checking $?m_i[\sigma]$ against t in context Δ.

$$\Sigma; \Delta \vdash ?m_i[\sigma] \equiv t; \Sigma'$$

The equation $\Delta \vdash ?m_i[\sigma] \equiv t$ gives a constraint on the definition of $?m_i$. If we can *solve* this equation for $?m_i$, such that the solution is unique up to definitional equality, then we have achieved our goal.

To find a solution, we construct a *partial right-inverse substitution* σ^{-1} for $\Delta \vdash \sigma : \Gamma$. This σ^{-1} is a section of σ, meaning $\sigma \circ \sigma^{-1} = \mathrm{id}_\Gamma$.

With σ^{-1}, we solve the equation as follows.

$$\begin{aligned} \Gamma \vdash ?m_i[\sigma][\sigma^{-1}] &\equiv t[\sigma^{-1}] \\ \Gamma \vdash ?m_i &\equiv t[\sigma^{-1}] \end{aligned}$$

yielding a solution $?m_i := t[\sigma^{-1}]$ in context Γ.

A unique σ^{-1} solving the equation $\Delta \vdash ?m_i[\sigma] \equiv t$ is constructible when the following *pattern conditions* are met.

1. The substitution σ contains only variables;
2. No variable in σ appears more than once;
3. Every free variable in t appears in σ;
4. The metavariable $?m_i$ does not appear in t.

We call the first two conditions *linearity*. The third condition checks there are no *escaping variables*: variables used in the solution which are not in scope at Γ. The final condition is the *occurs check* which prevents self-referential solutions.

There is one subtlety to the first condition when a variable is *defined* by a let-binding. We say t is defined in a substitution when it arises from a reduction

$$\Gamma \vdash \mathbf{let}\ x : A = t\ \mathbf{in}\ u \Rightarrow u[\uparrow \mathrm{id}_\Gamma, t]$$

Definitions violate linearity, as they are arbitrary terms, not variables. However, since definitions can be unfolded, we ignore them during the linearity check.

As σ contains a unique variable x_i from Δ for each variable y_j in Γ, there is an injective map $s : \Gamma \rightarrowtail \Delta$, sending $y_j \mapsto x_i$. This is precisely a partial inverse σ^{-1}: at each variable x_i in Δ, if

there is a unique y_j such that $s(y_j) = x_i$, then $\sigma_i^{-1} = y_j$. Otherwise, σ_i^{-1} is undefined. Since every free variable in t appears in σ, $t[\sigma^{-1}]$ never accesses an undefined position in σ^{-1}.

3.6.3 Pattern unification and NbE

In the NbE world, substitutions become *environments*. Instead of lists of terms, environments are lists of values interpreting the free variables of a term.

We begin by extending the domains of values and propositions. We also add a data-structure for partial renamings.

$$
\begin{array}{llllll}
\mathsf{D}^{\mathsf{ne}} & \ni & e & ::= & \cdots \\
 & & & \mid & \mathsf{Meta}\ ?m_i\ \rho \\
\mathsf{D}^{\Omega} & \ni & p & ::= & \cdots \\
 & & & \mid & \mathsf{PMeta}\ ?m_i\ \rho \\
\mathsf{Renaming} & \ni & \theta & ::= & () \mid (\theta, \mathsf{Var}_l) \mid (\theta, \Uparrow)
\end{array}
$$

In both domains, the semantic metavariable is the variable with an environment. Metavariables are also *neutral* – they block further computation. Note that unlike regular variables, metavariables can *unblock* when solved.

Partial renamings are lists of variables (represent as de Bruijn levels) and undefined values, $\Uparrow$. We equivalently view renamings as partial functions on variables.

Semantic evaluation now requires ambient access to the metacontext Σ to substitute in solved metavariables. In practise, this is achieved using a reader monad providing implicit access to the metacontext.

$$
\begin{array}{ll}
[\![?m_i]\!]^{\mathcal{U}}\rho = [\![t]\!]^{\mathcal{U}}\rho & \text{when } \Sigma = \Sigma', ?m_i := t, \Sigma'' \\
[\![?m_i]\!]^{\mathcal{U}}\rho = \mathsf{Meta}\ ?m_i\ \rho & \text{when } \Sigma = \Sigma', ?m_i, \Sigma'' \\
[\![?m_i]\!]^{\Omega}\rho = [\![t]\!]^{\Omega}\rho & \text{when } \Sigma = \Sigma', ?m_i := t, \Sigma'' \\
[\![?m_i]\!]^{\Omega}\rho = \mathsf{PMeta}\ ?m_i\ \rho & \text{when } \Sigma = \Sigma', ?m_i, \Sigma''
\end{array}
$$

When the solved term exists in the metacontext, we evaluate it in the current environment. Otherwise, we store the environment and metavariable in a semantic meta value.

Metavariables are solved during conversion checking. For example, we might have an equation

$$\Delta \vdash \mathsf{Meta}\ ?m_i\ \rho \equiv a$$

In the previous section, we described how partial inverse substitutions are created. We create a partial function $\mathsf{invert} : \mathsf{Env} \to\!\!\!\!\triangleright\ \mathsf{Renaming}$ which succeeds when the environment satisfies the linearity conditions.

In conversion checking, we always compare *values*. However, metavariable solutions are terms. Therefore, when applying the renaming ρ^{-1} to the value a, we both update free variables and quote the semantic value into a term. We define a function $\mathsf{rename}^{n}_{?m_i} : \mathsf{D}^{\mathcal{U}} \to\!\!\!\!\triangleright\ \mathsf{Renaming} \to\!\!\!\!\triangleright\ \mathsf{Tm}^{\mathcal{U}}$. 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.

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

The first rule ensures the variable Var_k is not escaping; by checking it is defined in θ. The second rule implements the occurs check by ensuring the metavariable $?m_i$ does not appear in its own solution. This way, we isolate the concerns of linearity, which depends only on the environment ρ, from the other checks. When going under a binder, we apply the closure to a fresh variable and extend the renaming in the natural way.

The final step is to piece these parts together to solve metavariables in conversion checking, then update the metacontext.

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

Both invert and rename might (validly) fail, so we take $\doteq$ to mean both that the result is defined, and is assigned to the variable on the left. When both of the hypotheses are defined, we find a valid solution for the metavariable and replace it in the metacontext.

3.6.4 Extended unification for negative types

We can extend pattern unification to apply to more general situations. Currently we cannot solve equations when the metavariable is inside an eliminator, even when there is a unique solution. Consider the follow example equations

$$\Gamma \vdash \mathbf{fst}\ (?m[\sigma]) \equiv t \qquad\qquad \Gamma \vdash (?m'[\sigma])\ x \equiv t'$$

In both cases, the metavariable has a negative type with a single constructor. Therefore, we can expand the metavariables using the η law. We therefore define $?m := \langle ?m_1; ?m_2 \rangle$ and $?m' := \lambda.\ ?m'_1$ for freshly created metavariables $?m_1$, $?m_2$ and $?m'$.

Now the eliminators can compute, so we resolve the equations to

$$\Gamma \vdash ?m_1[\sigma] \equiv t \qquad\qquad \Gamma \vdash ?m'_1[\uparrow \sigma, x] \equiv t'$$

both of which now have the form from the previous section, and can be solved provided the pattern conditions hold.

Chapter 4

Evaluation

In this chapter, we evaluate our $\mathrm{TT}^{\mathrm{obs}}$ implementation. The goal of this project was to create a suitable, practical implementation of $\mathrm{TT}^{\mathrm{obs}}$ [2, 1].

We give a quantitative evaluation of the features of the system. Expressivity of the language is important, but also tools such as helpful error messages allow the user to efficiently write correct code.

4.1 Error messages

Bidirectional type-checking identifies locations for suitable errors, for example when rules have hypotheses with a specific structure, we throw an error when the shape of the term is incorrect. We start with lexer and parser errors which detect an ill-formed term. We then have precise errors for type-checking, inference, conversion and pattern unification.

One of the most prominent errors is the *conversion* error, thrown when conversion between two types fails. The error message shows the macroscopic view of the two types when checking began, and the specific point at which conversion failed.

While error messages are extremely useful, they are problematic for more complex terms. Quoted terms are normal forms, which tend to be extremely large because let-definitions are always unfolded. A useful future improvement would add controlled unfolding of let-bindings to avoid unnecessary expansions which harm readability.

4.2 Proof assistant tools

In Section 3.6, we introduced *pattern unification*. This tool assists users by reconstructing terms from the surrounding context, helping reduce verbosity of programs. Furthermore, we introduce *syntactic sugar* which creates terms with holes in them.

A useful example of this is in proof concatenation. We include a term $\mathbf{trans}(A, B, C, e, e')$[1] which concatenates proofs $e : A \sim B$ and $e' : B \sim C$. In general, we need the endpoints A, B

[1] **trans** (transitivity) is admissible as it is subsumed by **transp** (transport).

and C for type-checking. But often the endpoints are inferrable from e and e'. Therefore, we introduce the following shorthand.

$$e \circ e' \triangleq \textbf{trans}(\_, \_, \_, e, e')$$

Another tool we include is *proof goals*. Goals are inserted into the source code to throw an error at a specific location. 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

$$?\{t_1, t_2, \ldots, t_n\}$$

For example, consider the following example program, in which we insert a proof goal.

```
let f : ℕ → ℕ =
  λx. S x
in
let x : ℕ =
  S (S (S 0))
in
f ?{f, x}
```

Running the checker on this input, we get the following message.

```
[error]: Found proof goal.
      ╭─▶ <test-file>@8:7-8:14
      │
    8 │     f ?{f, x}
      •       ┬──────
      •       ╰╴ Expected type [ℕ] at goal.
      •
      •          List of relevant terms and their types:
      •          f : ℕ → ℕ
      •          x : ℕ
      ╵
```

4.3 Quotient types example

We demonstrate implementation of the Booleans via a quotient on the natural numbers. This works by creating an equivalence relation which artificially equates all numbers $n \geq 1$, leaving exactly two elements: zero and "everything else". The full code for this example is given in Appendix C.

The implementation has two primary components: the relation R and proof it is an equivalence, and the dependent **if** expression to eliminate the booleans. Since our underlying type is $\mathbb{N}$, the equivalence proofs are by induction. The proofs themselves are somewhat complex, but manageable. Elimination of quotients requires an underlying function on the base type, and a proof that the equivalence relation is preserved. The preservation proof is relatively complex, and again proven by natural number induction.

It is very hard to write such a proof without the assistance of the type-checker. We make use of pattern unification when writing equality proofs, and utilise the syntactic sugar $t \sim u$, which infers the type at which the equality is formed. We also use use $e \circ e'$ for proof concatenation.

We use goals to help write proof terms. In fact, goals also help us simplify code. Consider the following subterm of the proof in Appendix C, lines $31 - 33$.

```
rec(x'. R x' (S l) → R (S l) (S k) → R x' (S k),
    . λ_. w,
    _ _. λ_. λ_. *, x)
```

If we place a goal in the code, we quickly learn the expected type is

```
rec(_. Ω, ⊥, _ _. ⊤, x) → ⊤ → rec(_. Ω, ⊥, _ _. ⊤, x)
```

which is inhabited by `λw. λ_. w`. This helps us determine this nonobvious simplification.

4.4 Simply typed lambda calculus example

Our final example exhibits Fiore's implementation of NbE for the simply typed lambda calculus [7]. This proof demonstrates the expressivity of the system, and makes extensive use of inductive types. Despite the proof being almost 500 lines long, the type-checker completes in under three seconds The code listing is found in Appendix D.

In this proof, we use *mix-fix* operators. These let us construct custom mathematical syntax, for example $[\![\tau]\!]$ for type denotations. We also redefine various type constructors to more readable names, for example we alias the `'Function` constructor.

```
let _⇒_ : Type ! → Type ! → Type ! =
  λdom. λcod. 'Function ((dom; cod), *)
in
...
```

A major difficulty encountered was making mutually inductive definitions for normal and neutral forms. We have no direct mutual definitions, so we instead use a work-around. We construct a datatype representing the union of normals and neutrals, and use the index as a predicate to divide the definition into two types. A similar trick is used in defining the quote and unquote functions, which are mutually recursive.

While this demonstrates a difficult barrier in the system, it also shows the expressive capabilities, and that even in a relatively primitive stage, it is possible to construct complex proofs.

Chapter 5

Conclusions

In this project, we used untyped normalisation by evaluation to create the first implementation of $\mathrm{TT}^{\mathrm{obs}}$. We designed suitable bidirectional typing rules to implement the typing relation. Then, we constructed an NbE algorithm to produce normal forms of terms for conversion checking. We introduced a novel technique for handling semantic propositions using a second untyped domain and another interpretation function performing no reduction besides substitution.

On top of the core calculus in [1] and [2], we added extensions of quotient types and inductive types. Quotient types exhibit the power of observational equality by providing control over the equivalence relation in types – a feature hard to recreate in other popular proof assistants. Inductive types are essential to use the language in a practical setting; datatypes are a core feature of any language. We designed and implemented a well-founded induction principle for inductive types corresponding to primitive recursion. Facilitating induction over arbitrary structures is another key feature of a useful proof assistant.

We added pattern unification to improve the practicality of using the system – with it, we can omit terms which are automatically determined from the context. Alongside this, we introduced proof goals to give type information while writing programs.

Overall, the product created throughout this project was a success and proved capable of implementing challenging proofs. While implementing these examples, the tool responded with useful messages to guide the user in filling out the details of the proof.

5.1 Future work

Designing a fully featured proof assistant is a huge task. Therefore, there are a large number of extensions this work would benefit from. We mentioned various extensions throughout the exposition, but here highlight some interesting ideas.

- Quotient inductive types [19]. A useful extension would be inductive definitions specifying quotients directly. The user would specify the points of a type, and equalities between them. The equivalence relation would be the reflexive transitive closure of the defined paths.
- Irrelevant proof term solver. Pattern unification only commits to definitionally unique

solutions. By definition, irrelevant terms are unique, so we might create a more aggressive solver which searches for proofs of a given proposition.

- Explicit mutual induction. Currently, we implement mutually defined datatypes and functions using a proxy. This is impractical and becomes very verbose. A first-class notion of mutual induction would make this process much easier.
- Generalised pattern matching. Nested pattern matching would allow for easier destructuring of datatypes. Matching on built-in types like natural numbers and Σ-types would offer greater flexibility, especially in nested patterns.

References

[1] Loïc Pujet and Nicolas Tabareau. Observational equality: Now for good. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–27, January 2022.

[2] Loïc Pujet. *Computing with Extensionality Principles in Dependent Type Theory*. PhD thesis, Nantes Université, December 2022.

[3] Thorsten Altenkirch and Conor McBride. Towards Observational Type Theory. 2006.

[4] Andreas Abel. *Normalization by Evaluation Dependent Types and Impredicativity*. PhD thesis, Institut für Informatik Ludwig-Maximilians-Universität München, München, May 2013.

[5] András Kovács. Elaboration-zoo, May 2023.

[6] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, May 1996.

[7] Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. *Mathematical Structures in Computer Science*, 32(8):1028–1065, September 2022.

[8] Andreas Abel, Thierry Coquand, and Miguel Pagano. A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance.

[9] 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.

[10] T Coquand and Gérard Huet. The calculus of constructions.

[11] Barkley Rosser. The Burali-Forti paradox. *Journal of Symbolic Logic*, 7(1):1–17, March 1942.

[12] 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.

[13] 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.

[14] 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.

[15] Jana Dunfield and Neel Krishnaswami. Bidirectional Typing. *ACM Computing Surveys*, 54(5):1–38, June 2022.

[16] Ulf Norell. Towards a practical programming language based on dependent type theory.

[17] 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.

[18] András Kovács. Elaboration with first-class implicit function types. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, August 2020.

[19] Marcelo P. Fiore, Andrew M. Pitts, and S. C. Steenkamp. Quotients, inductive types, and quotient inductive types, June 2022.

[20] Per Martin-Löf. Intuitionistic type theory. In *Studies in Proof Theory*, 1984.

[21] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[22] Martin Hofmann. *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh, 1995.

Appendix A

Inductive propositional equality

Here we give background on Martin-Löf-style inductive equality. This is not central to the $\mathrm{TT}^{\mathrm{obs}}$ system, as we introduce the alternative notion of *observational* equality. Also, we do not revisit these rules, so they are only serve as a reference.

Definitional equality decides many equalities between terms. However, there exist many semantically equivalent terms which are *not* definitionally equal; for example some equalities must be proven by induction. *Propositional* equality allows us to construct an explicit proof of equality between terms, and use it to judge them equal. With equality types, we introduce a form of Martin-Löf Type Theory [20].

We add the following terms to the grammar given in Section 2.1.

$$A, B, t, u \quad ::= \quad \cdots \mid \mathbf{refl}\ t \mid J[x\ z.C](t, u, v) \mid t =_A u \quad \text{Propositional equality}$$

$t =_A u$ is the type of *proofs* that t equals u at type A. In a constructive setting, *proofs themselves* are objects. The introduction form $\mathbf{refl}\ t$ constructs a reflexivity proof for any term at any type. The eliminator J is a based path-induction principle on equality proofs [21]. This means we create a motive C which depends on a variable x, and a proof that $t =_A x$. We start with a value u inhabiting C when x is defined as t – therefore, in this context, they x is *definitionally* equal to t. We then pull the endpoint x from t to t' along their proof of equality, v. This path induction is *based* because the first endpoint stays fixed throughout. This is provably equivalent to full path induction, where both endpoints are transported **??**.

Typing rules for inductive equality are as follows.

Id-Form
$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t =_A u : \mathcal{U}}$$

Id-Intro
$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathbf{refl}\ t : t =_A t}$$

Id-Elim
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : A \qquad \Gamma, x : A, z : t =_A x \vdash C : \mathcal{U} \qquad \Gamma \vdash u : t =_A t' \qquad \Gamma \vdash v : C[t/x, \mathbf{refl}\ t/z]}{\Gamma \vdash J[x\ z.C](t, t', u, v) : C[t'/x, u/z]}$$

The formation and **refl** rules are self-explanatory. The induction principle is more involved. The motive is a family indexed by a term $x : A$ and an equality proof $z : t =_A x$. The term $u : t =_A t'$

is a proof that t is equal to t'. We then give a term v which inhabits $C[t, \mathbf{refl}\ t]$, so the endpoints of the equality proof are *definitionally* equal. This lets us locally treat propositional equality as if it were definitional. The result of the J eliminator is the term v is transported along the proof of equality $u : t =_A t'$: the motive C now mentions t' in place of t.

Example A.0.1 (Casting)**.** Given a propositional proof $p : A =_{\mathcal{U}} B$ of equality, we construct a *casting* function.

$$J[B'\ \_.\ A \to B'](A, B, p, \lambda x.x) : A \to B$$

where $A \to B$ is shorthand for the non-dependent Π type $\Pi(\_ : A).B$. The motive says to construct a function type $A \to B$, it suffices to construct a function of type $A \to A$. J then transports the codomain to B using the proof $A =_{\mathcal{U}} B$. Of course, $A \equiv A$ (*definitional* equality), so the identity function inhabits $A \to A$.

As with the other types, we have definitional equality rules.

$$\textsc{Id-}\beta\ \frac{}{\Gamma \vdash J[x\ z.C](t, t, \mathbf{refl}\ t, v) \equiv v : C[t/x, \mathbf{refl}\ t/z]} \qquad \textsc{Id-Eq}\ \frac{\Gamma \vdash A \equiv A' : \mathcal{U} \qquad \Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash u \equiv u' : A}{\Gamma \vdash (t =_A u) \equiv (t' =_{A'} u') : \mathcal{U}}$$

The β rule says J reduces when its argument is a reflexivity proof, at which point v has the correct type. This indicates Example A.0.1 ultimately dissolves to the identity function, which makes sense: casting from A to B should amount to doing nothing.

Despite only having a single constructor, **refl**, propositional equality types express many properties. Nonetheless, there are still useful properties which *cannot* be proven. A notable example is function extensionality, where proving pointwise equality of two functions is insufficient to prove the functions themselves are equal. At first, it seems a reasonable extension to add the rule

$$\textsc{FunExt}\ \frac{\Gamma \vdash p : \Pi(z : A).\ f\ z =_{B[z/x]} g\ z}{\Gamma \vdash \mathbf{ext}(p) : f =_{\Pi(x:A).\ B} g}$$

however this breaks *canonicity*, meaning there are closed identity proofs which never reduce to **refl**, and J blocks on these proofs. A resolution is to add *equality reflection*, postulating that propositional equalities hold definitionally, but this breaks decidability of type checking [22].

This propositional equality is *proof relevant*: the proof itself is a term with computational content. It is, in particular, possible to construct definitionally distinct proofs of equality between two terms. For this reason it makes sense to reason with *iterated* proofs of equality (i.e. equalities between equalities, and so on). This gives rise to an ∞-*groupoid* structure on types [21], where equalities are witnessed up to higher equivalences, which themselves have higher witnesses and so on. In $\mathrm{TT}^{\mathrm{obs}}$, we instead have a *setoid* structure. Setoids are sets equipped with a truncated equivalence relation. That is to say, there are no *higher* equivalences: all proofs of equality are definitionally unique.

Appendix B

Categorical interpretation of Mendler induction

To motivate the induction principle, we look at inductive types from a categorical perspective. We view inductive type definitions as defining an *endofunctor* $\hat{F} : \mathcal{U}^A \to \mathcal{U}^A$ on the functor category $\mathcal{U}^A$ representing A-indexed types. This functor is a sum-of-products *signature* functor defined in the form

$$\hat{F}(F) = B_1 + B_2 + \cdots + B_n$$

where each B_i is some functor $A \to \mathcal{U}$, possibly depending on F. In particular, given a functor $X : A \to \mathcal{U}$, we obtain a new type family $\hat{F}(X) : A \to \mathcal{U}$. Here, X is thought of as an interpretation for the free variable F in each type B_i. Now suppose we start with the empty type family $\mathcal{E}$ – that is, at each index $p : A$, $\mathcal{E}(p)$ is the empty type. Consider the type $\hat{F}(\mathcal{E})$. Every constructor which depends on the free variable F will be empty, as it is a product with the empty family. Therefore, the only inhabitants are those which do not include recursive data; in other words the trees of depth at most one. Iterating the endofunctor again introduces another level to these trees, and so on. Ultimately, we want to reach a *fixed point* with this process – that is, when adding one more layer does not add more elements to the type. This type must satisfy $\hat{F}(\mu\hat{F}) \cong \mu\hat{F}$, meaning substituting the family $\mu\hat{F}$ into $\hat{F}$ gives the same type $\mu\hat{F}$ (up to isomorphism). In this sense, $\mu\hat{F}$ is a fixed point of the functor $\hat{F}$.

For elimination of inductive types, we need an *induction principle*. Categorically, this is given by a univeral mapping-out property from the object $\mu\hat{F}$. In particular, for every other functor $X : A \to \mathcal{U}$ with a morphism $\hat{F}(X) \xrightarrow{\phi} X$, we want a (unique) map $\mu\hat{F} \xrightarrow{\mathbf{fix}_X} X$ such that the diagram

$$\begin{array}{ccc} \hat{F}(\mu\hat{F}) & \xrightarrow{\cong} & \mu\hat{F} \\ {\scriptstyle \hat{F}(\mathbf{fix}_X)}\downarrow & & \vdots\,{\scriptstyle \mathbf{fix}_X} \\ \hat{F}(X) & \xrightarrow{\phi} & X \end{array}$$

commutes. Here, $\mathbf{fix}_X$ is a natural transformation, so given an index $p : A$, we have a function from the inductive type at index p, $(\mu\hat{F})_p$, to X_p. This is the diagram for an initial $\hat{F}$-algebra,

which is indeed a suitable construction for inductive types. By Lambek's lemma, this additionally gives us the desired isomorphism between $\hat{F}(\mu\hat{F})$ and $\mu\hat{F}$ witnessed by

$$\hat{F}(\mu\hat{F}) \xrightarrow{\textbf{in}} \mu\hat{F}$$
$$\mu\hat{F} \xrightarrow{\textbf{out}} \hat{F}(\mu\hat{F})$$

We generalise the induction principle $\mathbf{fix}_X$ to allow the type X_p to depend on the value of type $(\mu\hat{F})_p$ for the fully dependent induction principle.

The type-theoretic induction principle given in Section 2.4.2 does in fact correspond to the initial algebra $\mu\hat{F}$. Recall the following typing rule.

Fix

$$\frac{\begin{array}{c}\mu F \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i) \to F\ a_i}] \\ F[X] \triangleq \mu F : A \to \mathcal{U}.\ [\overrightarrow{C_i : (x_i : B_i[X/F]) \to F\ a_i}] \\ \Gamma, G : A \to \mathcal{U}, p : A, x : G\ p \vdash C : \mathfrak{s} \\ \Gamma, G : A \to \mathcal{U}, f : \Pi(p : A).\ \Pi(x : G\ p).\ C[G, p, x], p : A, x : F[G]\ p \vdash t : C[F[G], p, x]\end{array}}{\Gamma \vdash \mathbf{fix}\ [\mu F\ \mathbf{as}\ G]\ f\ p\ x : C = t : \Pi(p : A).\ \Pi(x : (\mu F)\ p).\ C[\mu F, p, x]}$$

In fact, f is really a natural transformation between G and C, and therefore a morphism in the functor category $\mathcal{U}^A$. We can lift it using $\hat{F}$ to give $\hat{F}(f) : \hat{F}(G) \to \hat{F}(C)$. We consider x as an element of $\hat{F}(G)$, so $\hat{F}(f)(x)$ is an element of $\hat{F}(C)$ as an immediate consequence from the data of G, f and x. Since t is a map from this data into C, we can equivalently view t categorically as a morphism $\hat{F}(C) \to C$, making (C, t) a candidate $\hat{F}$-algebra; in this sense Mendler recursion does indeed correspond with the unique induction morphism $\mathbf{fix}_C : \mu\hat{F} \to C$ induced by initiality.

Section 2.4.3 details how we extend the induction principle for inductive types to allow primitive recursion. This operates by introducing an extra parameter $\iota : \Pi(p : A).\ G\ p \to (\mu F)\ p$, which allows us to "view" the opaque type G as the real type μF.

Categorically, ι is a natural transformation $G \to \mu\hat{F}$. Therefore, we can lift it using $\hat{F}$ to get a natural transformation $\hat{F}(\iota) : \hat{F}(G) \to \hat{F}(\mu\hat{F})$. But, $\hat{F}(\mu\hat{F})$ is isomorphic to $\mu\hat{F}$, so we define

$$\iota' \triangleq \mathbf{in} \circ \hat{F}(\iota) : \hat{F}(G) \to \mu\hat{F}$$

At the top level of the rule, we substitute the identity function in for ι. In typing the body, t, we create $\iota' = \mathbf{in} \circ \hat{F}(\iota)$. This becomes $\mathbf{in} \circ \hat{F}(\mathrm{id})$. By functoriality, $\hat{F}(\mathrm{id}) = \mathrm{id}$. So this resolves to $\mathbf{in}$, which is an isomorphism, and behaves like an identity.

Appendix C

Code for quotient types example

Here we give a formalisation implementing Booleans as a quotient on the natural numbers.

```
let cast_compose : (A :U U) → (B :U U) → (C :U U)
                 → (AB :Ω A ~ B) → (BC :Ω B ~ C)
                 → (x :U A)
                 → cast(A, C, AB ∘ BC, x) ~ cast(B, C, BC, cast(A, B, AB, x)) =
  λA. λB. λC. λAB. λBC. λx.
        transp(B,
               B' BB'.
                 cast(A, B', AB ∘ BB', x) ~ cast(B, B', BB', cast(A, B, AB, x)),
               refl (cast(A, B, AB, x)), C, BC)
in
let R : ℕ → ℕ → Ω =
  λx. λy. rec(_. Ω, rec(_. Ω, ⊤, _ _. ⊥, y), _ _. rec(_. Ω, ⊥, _ _. ⊤, y), x)
in
let Rr : (x :U ℕ) → R x x =
  λx. rec(z. R z z, *, _ _. *, x)
in
let Rs : (x :U ℕ) → (y :U ℕ) → R x y → R y x =
  λx. λy. rec(y'. R x y' → R y' x,
              rec(x'. R x' 0 → R 0 x', λw. w, _ _. λw. w, x),
              k _. rec(x'. R x' (S k) → R (S k) x', λw. w, _ _. λw. w, x),
              y)
in
let Rt : (x :U ℕ) → (y :U ℕ) → (z :U ℕ) → R x y → R y z → R x z =
  λx. λy. λz. rec(z'. R x y → R y z' → R x z',
                  rec(y'. R x y' → R y' 0 → R x 0,
                      λx0. λ_. x0,
                      k _. λ_. λw. abort(R x 0, w),
                      y),
                  k _. rec(y'. R x y' → R y' (S k) → R x (S k),
                           λ_. λw. abort(R x (S k), w),
                           l _. rec(x'. R x' (S l) → R (S l) (S k) → R x' (S k),
                                    λw. λ_. w,
                                    _ _. λ_. λ_. *, x), y), z)
in
```

```
35 let Bool : U =
36   ℕ / (x y. R x y,
37         x. Rr x, x y xRy. Rs x y xRy, x y z xRy yRz. Rt x y z xRy yRz)
38 in
39 let true : Bool = π 0 in
40 let false : Bool = π (S 0) in
41 let if : (B :U Bool → U) → (c :U Bool) → B true → B false → B c =
42   λB. λc. λt. λf.
43     let congB : (x :U ℕ) → (y :U ℕ) → R x y → B (π x) ~ B (π y) =
44       λx. λy. λxRy. ap(U, x. B x, (π x : Bool), π y, xRy)
45     in
46     let choose : (x :U ℕ) → B (π x) =
47       λx. rec(x'. B (π x'), t, k _. cast(B false, B (π (S k)),
48                                         congB (S 0) (S k) *,
49                                         f), x)
50     in
51     let presTRhs : (x :U ℕ) → (y :U ℕ) → R x y → Ω =
52       λx. λy. λxRy.
53         (choose x) ~ cast(B (π y), B (π x), congB y x (Rs x y xRy), choose y)
54     in
55     let presT : (x :U ℕ) → (y :U ℕ) → Ω =
56       λx. λy. (xRy :Ω R x y) → presTRhs x y xRy
57     in
58     let pres : (x :U ℕ) → (y :U ℕ) → presT x y =
59       λx. λy. rec(x'. presT x' y,
60                   rec(y'. presT 0 y',
61                       λ_. refl t,
62                       l _. λw. abort(presTRhs 0 (S l) w, w),
63                       y),
64                   k _.
65                     rec(y'. presT (S k) y',
66                         λw. abort(presTRhs (S k) 0 w, w),
67                         l _. λ_. cast_compose (B false) (B (π (S l))) (B (π (S
   ↪  k)))
68                                               (congB (S 0) (S l) *)
69                                               (congB (S l) (S k) *)
70                                               f, y), x)
71     in
72     Q-elim(z. B z, x. choose x, x y e. pres x y e, c)
73 in
74 if (λb. if (λ_. U) b ℕ (ℕ × ℕ)) true (S 0) (0; S (S 0))
```

Lines $11-12$ implement the equivalence relation R on the naturals. We relate $R\ 0\ 0$ and $R\ (S\ n)\ (S\ m)$ for all $n, m \in \mathbb{N}$. This gives two equivalence classes, $\{0\}$ and $\{1, 2, 3, \dots\}$.

Lines $14-33$ prove that R is an equivalence relation.

Lines $35-40$ construct a new type of Booleans by quotienting the naturals with the relation R, and values for true and false.

Lines $41-72$ implement the dependent eliminator for the Booleans: **if**. This includes an underlying map **choose** (lines $46-49$), which sends 0 to t and all other numbers to f. We then

prove that this preserves the relation R on lines $51 - 70$. This is enough information to construct a map out of the Booleans.

Line 74 shows a simple use-case of the dependent eliminator. We first specify the return type: in the true branch, we return a number, and in the false branch, a pair of numbers. Then we provide the data for the two branches.

Appendix D

Code for simply typed lambda calculus example

Here we give the code listing for the NbE implementation of the simply typed lambda calculus.

```
let Type : 1 → U =
  μTy : 1 → U.
    [ 'Unit : 1 → Ty !
    ; 'Product : (Ty ! × Ty !) → Ty !
    ; 'Function : (Ty ! × Ty !) → Ty !
    ]
  functor A B f _ x =
    match x as _ return (lift [Ty] B) ! with
    | 'Unit (_, _) → 'Unit (!, *)
    | 'Product (τ₁-τ₂, _) → 'Product ((f ! (fst τ₁-τ₂); f ! (snd τ₁-τ₂)), *)
    | 'Function (τ₁-τ₂, _) → 'Function ((f ! (fst τ₁-τ₂); f ! (snd τ₁-τ₂)), *)
in
let 𝟙 : Type ! = 'Unit (!, *) in
let _*_ : Type ! → Type ! → Type ! =
  λt. λu. 'Product ((t; u), *)
in
let _⇒_ : Type ! → Type ! → Type ! =
  λdom. λcod. 'Function ((dom; cod), *)
in
let 𝔽↓T =
  μCtx : 1 → U.
    [ 'Empty : 1 → Ctx !
    ; 'Extend : (Ctx ! × Type !) → Ctx !
    ]
  functor A B f _ x =
    match x as _ return (lift [Ctx] B) ! with
    | 'Empty (_, _) → 'Empty (!, *)
    | 'Extend (Γ-τ, _) → 'Extend ((f ! (fst Γ-τ); snd Γ-τ), *)
in
let · : 𝔽↓T ! = 'Empty (!, *) in
let _::_ : 𝔽↓T ! → Type ! → 𝔽↓T ! =
  λΓ. λτ. 'Extend ((Γ; τ), *)
in
let Ix =
  μIx : (Type ! × 𝔽↓T !) → U.
    [ 'Ix0 : (τ-Γ : Type ! × 𝔽↓T !) → Ix (fst τ-Γ; (snd τ-Γ) :: (fst τ-Γ))
    ; 'IxS : (τ-Γ-τ' : Σ(τ : Type !). (Σ(Γ : 𝔽↓T !). Type ! × Ix (τ; Γ)))
      → Ix (fst τ-Γ-τ'; (fst (snd τ-Γ-τ')) :: (fst (snd (snd τ-Γ-τ'))))
    ]
in
```

```
let 𝔽↓t̃ : (𝔽↓T ! × 𝔽↓T !) → U =
  λCs.
    let Δ : 𝔽↓T ! = fst Cs in
    let Γ : 𝔽↓T ! = snd Cs in
    (τ :U Type !) → Ix (τ; Δ) → Ix (τ; Γ)
in
let Term =
  μTm : (Type ! × 𝔽↓T !) → U.
    [ 'Var : (τ-Γ : Σ(τ : Type !). Σ(Γ : 𝔽↓T !). Ix (τ; Γ))
      → Tm (fst τ-Γ; fst (snd τ-Γ))
    ; 'One : (Γ : 𝔽↓T !) → Tm (𝟙; Γ)
    ; 'Pair : (τ₁-τ₂-Γ : Σ(τ₁ : Type !). Σ(τ₂ : Type !).
                         Σ(Γ : 𝔽↓T !). (Tm (τ₁; Γ) × Tm (τ₂; Γ)))
      → Tm ((fst τ₁-τ₂-Γ) ∗ (fst (snd τ₁-τ₂-Γ)); fst (snd (snd τ₁-τ₂-Γ)))
    ; 'Fst : (τ₁-Γ : Σ(τ₁ : Type !). Σ(Γ : 𝔽↓T !).
                     Σ(τ₂ : Type !). Tm ((τ₁ ∗ τ₂); Γ))
      → Tm (fst τ₁-Γ; fst (snd τ₁-Γ))
    ; 'Snd : (τ₂-Γ : Σ(τ₂ : Type !). Σ(Γ : 𝔽↓T !).
                     Σ(τ₁ : Type !). Tm ((τ₁ ∗ τ₂); Γ))
      → Tm (fst τ₂-Γ; fst (snd τ₂-Γ))
    ; 'Lambda : (τ₁-τ₂-Γ : Σ(τ₁ : Type !). Σ(τ₂ : Type !).
                           Σ(Γ : 𝔽↓T !). Tm (τ₂; (Γ :: τ₁)))
      → Tm ((fst τ₁-τ₂-Γ) ⇒ (fst (snd τ₁-τ₂-Γ)); fst (snd (snd τ₁-τ₂-Γ)))
    ; 'App : (τ₂-Γ : Σ(τ₂ : Type !). Σ(Γ : 𝔽↓T !). Σ(τ₁ : Type !).
                     Tm ((τ₁ ⇒ τ₂); Γ) × Tm (τ₁; Γ))
      → Tm (fst τ₂-Γ; fst (snd τ₂-Γ))
    ]
in
let Form =
  μForm : 1 → U. ['Ne : 1 → Form !; 'Nf : 1 → Form !]
in
let Ne : Form ! = 'Ne (!, *) in
let Nf : Form ! = 'Nf (!, *) in
let Normal =
  μNormal : (Form ! × (Type ! × 𝔽↓T !)) → U.
    [ 'VVar : (τ-Γ : Σ(τ : Type !). Σ(Γ : 𝔽↓T !). Ix (τ; Γ))
      → Normal (Ne; (fst τ-Γ; fst (snd τ-Γ)))
    ; 'VOne : (Γ : 𝔽↓T !) → Normal (Nf; (𝟙;Γ))
    ; 'VPair : (τ₁-τ₂-Γ : Σ(τ₁ : Type !). Σ(τ₂ : Type !). Σ(Γ : 𝔽↓T !).
                          Normal (Nf; (τ₁; Γ)) × Normal (Nf; (τ₂; Γ)))
      → Normal (Nf; ((fst τ₁-τ₂-Γ) ∗ (fst (snd τ₁-τ₂-Γ)); fst (snd (snd τ₁-τ₂-Γ))))
    ; 'VFst : (τ₁-Γ : Σ(τ₁ : Type !). Σ(Γ : 𝔽↓T !). Σ(τ₂ : Type !).
                      Normal (Ne; (τ₁ ∗ τ₂; Γ)))
      → Normal (Ne; (fst τ₁-Γ; fst (snd τ₁-Γ)))
    ; 'VSnd : (τ₂-Γ : Σ(τ₂ : Type !). Σ(Γ : 𝔽↓T !). Σ(τ₁ : Type !).
                      Normal (Ne; (τ₁ ∗ τ₂; Γ)))
      → Normal (Ne; (fst τ₂-Γ; fst (snd τ₂-Γ)))
    ; 'VLambda : (τ₁-τ₂-Γ : Σ(τ₁ : Type !). Σ(τ₂ : Type !). Σ(Γ : 𝔽↓T !).
                            Normal (Nf; (τ₂; (Γ :: τ₁))))
      → Normal (Nf; ((fst τ₁-τ₂-Γ) ⇒ (fst (snd τ₁-τ₂-Γ)); fst (snd (snd τ₁-τ₂-Γ))))
    ; 'VApp : (τ₂-Γ : Σ(τ₂ : Type !). Σ(Γ : 𝔽↓T !). Σ(τ₁ : Type !).
                      Normal (Ne; (τ₁ ⇒ τ₂; Γ)) × Normal (Nf; (τ₁; Γ)))
      → Normal (Ne; (fst τ₂-Γ; fst (snd τ₂-Γ)))
    ]
in
let 𝓜 : Type ! → 𝔽↓T ! → U = λτ. λΓ. Normal (Ne; (τ; Γ)) in
let 𝓝 : Type ! → 𝔽↓T ! → U = λτ. λΓ. Normal (Nf; (τ; Γ)) in
let pshf : (τ :U Type !) → (Δ :U 𝔽↓T !) → 𝓜 τ Δ
         → (Γ :U 𝔽↓T !) → 𝔽↓t̃ (Δ; Γ) → 𝓜 τ Γ =
  λτ. λΔ.
    (fix [Normal as N] pshf f-τ'-Δ' v :
      let f = fst f-τ'-Δ' in
      let τ' = fst (snd f-τ'-Δ') in
```

```
    let Δ' = snd (snd f-τ'-Δ') in
    (Γ :U 𝔽↓T !) → 𝔽↓t̃ (Δ'; Γ) → Normal (f; (τ'; Γ)) =
  let f = fst f-τ'-Δ' in
  let τ' = fst (snd f-τ'-Δ') in
  let Δ' = snd (snd f-τ'-Δ') in
  λΓ. λρ.
    match v as _ return Normal (f; (τ'; Γ)) with
    | 'VVar (τ'-Δ''-ix, pf) →
      let τ' = fst τ'-Δ''-ix in
      let Δ'' = fst (snd τ'-Δ''-ix) in
      let ix = snd (snd τ'-Δ''-ix) in
      let ρ' =
        cast(𝔽↓t̃ (Δ'; Γ), 𝔽↓t̃ (Δ''; Γ),
             ap(U, Ξ. 𝔽↓t̃ (Ξ; Γ), _, _, sym (snd (snd pf))), ρ)
      in
      'VVar ((τ'; (Γ; ρ' τ' ix)), <fst pf, <fst (snd pf), refl Γ>>)
    | 'VOne (_, pf) → 'VOne (Γ, <fst pf, <fst (snd pf), refl Γ>>)
    | 'VPair (τ₁-τ₂-Δ''-t-u, pf) →
      let τ₁ = fst τ₁-τ₂-Δ''-t-u in
      let τ₂ = fst (snd τ₁-τ₂-Δ''-t-u) in
      let Δ'' = fst (snd (snd τ₁-τ₂-Δ''-t-u)) in
      let t = fst (snd (snd (snd τ₁-τ₂-Δ''-t-u))) in
      let u = snd (snd (snd (snd τ₁-τ₂-Δ''-t-u))) in
      let ρ' =
        cast(𝔽↓t̃ (Δ'; Γ), 𝔽↓t̃ (Δ''; Γ),
             ap(U, Ξ. 𝔽↓t̃ (Ξ; Γ), _, _, sym (snd (snd pf))), ρ)
      in
      'VPair ((τ₁; (τ₂; (Γ; (pshf (Nf; (τ₁; Δ'')) t Γ ρ';
                              pshf (Nf; (τ₂; Δ'')) u Γ ρ')))),
              <fst pf, <fst (snd pf), refl Γ>>)
    | 'VFst (τ₁-Δ''-τ₂-t, pf) →
      let τ₁ = fst τ₁-Δ''-τ₂-t in
      let Δ'' = fst (snd τ₁-Δ''-τ₂-t) in
      let τ₂ = fst (snd (snd τ₁-Δ''-τ₂-t)) in
      let t = snd (snd (snd τ₁-Δ''-τ₂-t)) in
      let ρ' =
        cast(𝔽↓t̃ (Δ'; Γ), 𝔽↓t̃ (Δ''; Γ),
             ap(U, Ξ. 𝔽↓t̃ (Ξ; Γ), _, _, sym (snd (snd pf))), ρ)
      in
      'VFst ((τ₁; (Γ; (τ₂; pshf (Ne; (τ₁ ∗ τ₂; Δ'')) t Γ ρ'))),
             <fst pf, <fst (snd pf), refl Γ>>)
    | 'VSnd (τ₂-Δ''-τ₁-t, pf) →
      let τ₂ = fst τ₂-Δ''-τ₁-t in
      let Δ'' = fst (snd τ₂-Δ''-τ₁-t) in
      let τ₁ = fst (snd (snd τ₂-Δ''-τ₁-t)) in
      let t = snd (snd (snd τ₂-Δ''-τ₁-t)) in
      let ρ' =
        cast(𝔽↓t̃ (Δ'; Γ), 𝔽↓t̃ (Δ''; Γ),
             ap(U, Ξ. 𝔽↓t̃ (Ξ; Γ), _, _, sym (snd (snd pf))), ρ)
      in
      'VSnd ((τ₂; (Γ; (τ₁; pshf (Ne; (τ₁ ∗ τ₂; Δ'')) t Γ ρ'))),
             <fst pf, <fst (snd pf), refl Γ>>)
    | 'VLambda (τ₁-τ₂-Δ''-t, pf) →
      let τ₁ = fst τ₁-τ₂-Δ''-t in
      let τ₂ = fst (snd τ₁-τ₂-Δ''-t) in
      let Δ'' = fst (snd (snd τ₁-τ₂-Δ''-t)) in
      let t = snd (snd (snd τ₁-τ₂-Δ''-t)) in
      let ρ' : 𝔽↓t̃ (Δ'' :: τ₁; Γ :: τ₁) =
        λτ. λix.
          match ix as _ return Ix (τ; Γ :: τ₁) with
          | 'Ix0 (τ''-Ξ, pf') → 'Ix0 ((fst τ''-Ξ; Γ),
                                      <fst pf', <refl Γ, snd (snd pf')>>)
          | 'IxS (τ''-Ξ-τ'-ix, pf') →
```

```
              let τ'' = fst τ''-Ξ-τ'-ix in
              let Ξ = fst (snd τ''-Ξ-τ'-ix) in
              let τ' = fst (snd (snd τ''-Ξ-τ'-ix)) in
              let ix =
                cast(Ix (τ''; Ξ), Ix (τ; Δ'),
                     <fst pf', fst (snd pf') ∘ snd (snd pf)>,
                     snd (snd (snd τ''-Ξ-τ'-ix)))
              in
              'IxS ((τ; (Γ; (τ'; ρ τ ix))), <refl τ, <refl Γ, snd (snd pf')>>)
        in
        'VLambda ((τ₁; (τ₂; (Γ; pshf (Nf; (τ₂; Δ'' :: τ₁)) t (Γ :: τ₁) ρ'))),
                  <fst pf, <fst (snd pf), refl Γ>>)
      | 'VApp (τ₂-Δ'-τ₁-t-u, pf) →
        let τ₂ = fst τ₂-Δ'-τ₁-t-u in
        let Δ'' = fst (snd τ₂-Δ'-τ₁-t-u) in
        let τ₁ = fst (snd (snd τ₂-Δ'-τ₁-t-u)) in
        let t = fst (snd (snd (snd τ₂-Δ'-τ₁-t-u))) in
        let u = snd (snd (snd (snd τ₂-Δ'-τ₁-t-u))) in
        let ρ' =
          cast(𝔽↓T̃ (Δ'; Γ), 𝔽↓T̃ (Δ''; Γ),
               ap(U, Ξ. 𝔽↓T̃ (Ξ; Γ), _, _, sym (snd (snd pf))), ρ)
        in
        'VApp ((τ₂; (Γ; (τ₁; (pshf (Ne; (τ₁ ⇒ τ₂; Δ'')) t Γ ρ';
                              pshf (Nf; (τ₁; Δ'')) u Γ ρ')))),
               <fst pf, <fst (snd pf), refl Γ>>)
    ) (Ne; (τ; Δ))
in
let ⟦_⟧_ : Type ! → 𝔽↓T ! → U =
  (fix [Type as Ty] SemTy _ ty : 𝔽↓T ! → U = λΓ.
    match ty as _ return U with
    | 'Unit (_, _) → 1
    | 'Product (p, _) →
      let τ₁ = fst p in
      let τ₂ = snd p in
      SemTy ! τ₁ Γ × SemTy ! τ₂ Γ
    | 'Function (f, _) →
      let τ₁ = fst f in
      let τ₂ = snd f in
      (Δ :U 𝔽↓T !) → 𝔽↓T̃ (Γ; Δ) → SemTy ! τ₁ Δ → SemTy ! τ₂ Δ) !
in
let Π : 𝔽↓T ! → 𝔽↓T ! → U =
  (fix [𝔽↓T as Ctx] Env _ Γ : 𝔽↓T ! → U = λΔ.
    match Γ as _ return U with
    | 'Empty (_, _) → 1
    | 'Extend (Γ-τ, _) →
      let Γ = fst Γ-τ in
      let τ = snd Γ-τ in
      Env ! Γ Δ × ⟦ τ ⟧ Δ) !
in
let rn : (Γ :U 𝔽↓T !) → (Δ :U 𝔽↓T !) → 𝔽↓T̃ (Δ; Γ) → (τ :U Type !)
      → ⟦ τ ⟧ Δ → ⟦ τ ⟧ Γ =
  λΓ. λΔ. λρ.
    (fix [Type as Ty view ι] rn _ τ : ⟦ (ι ! τ) ⟧ Δ → ⟦ (ι ! τ) ⟧ Γ =
      match τ as τ' return
        let τ' : Type ! = in (fmap[Type](Ty, Type, ι, !, τ')) in
        ⟦ τ' ⟧ Δ → ⟦ τ' ⟧ Γ
      with
      | 'Unit (_, _) → λ_. !
      | 'Product (τ₁-τ₂, _) →
        let τ₁ = fst τ₁-τ₂ in
        let τ₂ = snd τ₁-τ₂ in
        λpair.
          let t = fst pair in
```

```
          let u = snd pair in
          (rn ! τ₁ (fst pair); rn ! τ₂ (snd pair))
      | 'Function (τ₁-τ₂, _) →
        let τ₁ = fst τ₁-τ₂ in
        let τ₂ = snd τ₁-τ₂ in
        λf. λΔ'. λρ'. f Δ' (λχ. λix. ρ' χ (ρ χ ix))) !
in
let lookup : (τ :U Type !) → (Γ :U 𝔽↓T !) → Ix (τ; Γ)
           → (Δ :U 𝔽↓T !) → Π Γ Δ → ⟦ τ ⟧ Δ =
  λτ. λΓ.
  (fix [Ix as I] lookup τ-Γ ix : (Δ :U 𝔽↓T !) → Π (snd τ-Γ) Δ → ⟦ (fst τ-Γ) ⟧ Δ =
    let τ = fst τ-Γ in
    let Γ = snd τ-Γ in
    λΔ. λenv.
      match ix as _ return ⟦ τ ⟧ Δ with
      | 'Ix0 (τ'-Γ', pf) →
        let Γ' = snd τ'-Γ' in
        let env-cast =
          cast(Π Γ Δ, Π (Γ' :: τ) Δ, ap(U, Ξ. Π Ξ Δ, _, Γ' :: τ, sym (snd pf)), env)
        in
        snd env-cast
      | 'IxS (τ'-Γ'-τ''-ix, pf) →
        let τ' = fst τ'-Γ'-τ''-ix in
        let Γ' = fst (snd τ'-Γ'-τ''-ix) in
        let τ'' = fst (snd (snd τ'-Γ'-τ''-ix)) in
        let ix = snd (snd (snd τ'-Γ'-τ''-ix)) in
        let ix' =
          cast(I (τ'; Γ'), I (τ; Γ'), ap(U, χ. I (χ; Γ'), _, _, fst pf), ix)
        in
        let env-cast =
          cast(Π Γ Δ, Π (Γ' :: τ') Δ, ap(U, Ξ. Π Ξ Δ, _, Γ' :: τ', sym (snd pf)), env)
        in
        lookup (τ; Γ') ix' Δ (fst env-cast)) (τ; Γ)
in
let __⟦_⟧__ : (Γ :U 𝔽↓T !) → (τ :U Type !) → Term (τ; Γ)
            → (Δ :U 𝔽↓T !) → Π Γ Δ → ⟦ τ ⟧ Δ =
  λΓ. λτ.
  (fix [Term as Tm ] eval τ-Γ tm : (Δ :U 𝔽↓T !) → Π (snd τ-Γ) Δ → ⟦ (fst τ-Γ) ⟧ Δ =
    let τ = fst τ-Γ in
    let Γ = snd τ-Γ in
    λΔ. λenv.
      match tm as _ return ⟦ τ ⟧ Δ with
      | 'Var (τ'-Γ'-ix, pf) →
        let τ' = fst τ'-Γ'-ix in
        let Γ' = fst (snd τ'-Γ'-ix) in
        let ix = snd (snd τ'-Γ'-ix) in
        let env' =
          cast(Π Γ Δ, Π Γ' Δ, ap(U, Ξ. Π Ξ Δ, _, _, sym (snd pf)), env)
        in
        cast(⟦ τ' ⟧ Δ, ⟦ τ ⟧ Δ,
             ap(U, τ''. ⟦ τ'' ⟧ Δ, _, _, fst pf), lookup τ' Γ' ix Δ env')
      | 'One (_, pf) → cast(𝟙, ⟦ τ ⟧ Δ, ap(U, τ'. ⟦ τ' ⟧ Δ, 𝟙, τ, fst pf), !)
      | 'Pair (τ₁-τ₂-Γ'-t-u, pf) →
        let τ₁ = fst τ₁-τ₂-Γ'-t-u in
        let τ₂ = fst (snd τ₁-τ₂-Γ'-t-u) in
        let Γ' = fst (snd (snd τ₁-τ₂-Γ'-t-u)) in
        let t = fst (snd (snd (snd τ₁-τ₂-Γ'-t-u))) in
        let u = snd (snd (snd (snd τ₁-τ₂-Γ'-t-u))) in
        let env' =
          cast(Π Γ Δ, Π Γ' Δ, ap(U, Ξ. Π Ξ Δ, _, _, sym (snd pf)), env)
        in
        let vt : ⟦ τ₁ ⟧ Δ =
          eval (τ₁; Γ') t Δ env'
```

```
      in
      let vu : ⟦ τ₂ ⟧ Δ =
        eval (τ₂; Γ') u Δ env'
      in
      cast(⟦ τ₁ ⟧ Δ × ⟦ τ₂ ⟧ Δ, ⟦ τ ⟧ Δ,
           ap(U, τ'. ⟦ τ' ⟧ Δ, τ₁ ∗ τ₂, τ, fst pf), (vt; vu))
    | 'Fst (τ₁-Γ'-τ₂-t, pf) →
      let τ₁ = fst τ₁-Γ'-τ₂-t in
      let Γ' = fst (snd τ₁-Γ'-τ₂-t) in
      let τ₂ = fst (snd (snd τ₁-Γ'-τ₂-t)) in
      let t = snd (snd (snd τ₁-Γ'-τ₂-t)) in
      let env' =
        cast(Π Γ Δ, Π Γ' Δ, ap(U, Ξ. Π Ξ Δ, _, _, sym (snd pf)), env)
      in
      let vt : ⟦ τ₁ ⟧ Δ × ⟦ τ₂ ⟧ Δ =
        eval (τ₁ ∗ τ₂; Γ') t Δ env'
      in
      cast(⟦ τ₁ ⟧ Δ, ⟦ τ ⟧ Δ, ap(U, τ'. ⟦ τ' ⟧ Δ, _, _, fst pf), fst vt)
    | 'Snd (τ₂-Γ'-τ₁-t, pf) →
      let τ₂ = fst τ₂-Γ'-τ₁-t in
      let Γ' = fst (snd τ₂-Γ'-τ₁-t) in
      let τ₁ = fst (snd (snd τ₂-Γ'-τ₁-t)) in
      let t = snd (snd (snd τ₂-Γ'-τ₁-t)) in
      let env' =
        cast(Π Γ Δ, Π Γ' Δ, ap(U, Ξ. Π Ξ Δ, _, _, sym (snd pf)), env)
      in
      let vt : ⟦ τ₁ ⟧ Δ × ⟦ τ₂ ⟧ Δ =
        eval (τ₁ ∗ τ₂; Γ') t Δ env'
      in
      cast(⟦ τ₂ ⟧ Δ, ⟦ τ ⟧ Δ, ap(U, τ'. ⟦ τ' ⟧ Δ, _, _, fst pf), snd vt)
    | 'Lambda (τ₁-τ₂-Γ'-t, pf) →
      let τ₁ = fst τ₁-τ₂-Γ'-t in
      let τ₂ = fst (snd τ₁-τ₂-Γ'-t) in
      let Γ' = fst (snd (snd τ₁-τ₂-Γ'-t)) in
      let t = snd (snd (snd τ₁-τ₂-Γ'-t)) in
      let env' =
        cast(Π Γ Δ, Π Γ' Δ, ap(U, Ξ. Π Ξ Δ, _, _, sym (snd pf)), env)
      in
      let Λt : (Δ' :U 𝔽↓T !) → 𝔽↓t̃ (Δ; Δ') → ⟦ τ₁ ⟧ Δ' → ⟦ τ₂ ⟧ Δ' =
        λΔ'. λf. λχ.
          let rn-env : (Ξ :U 𝔽↓T !) → Π Ξ Δ → 𝔽↓t̃ (Δ; Δ') → Π Ξ Δ' =
            (fix [𝔽↓T as Ctx view ι] rn-env _ Ξ : Π (ι ! Ξ) Δ → 𝔽↓t̃ (Δ; Δ')
                                                → Π (ι ! Ξ) Δ' =
              match Ξ as Ξ' return
                let Ξ'' : 𝔽↓T ! = in (fmap[𝔽↓T](Ctx, 𝔽↓T, ι, !, Ξ')) in
                Π Ξ'' Δ → 𝔽↓t̃ (Δ; Δ') → Π Ξ'' Δ'
              with
              | 'Empty (_, _) → λ_. λ_. !
              | 'Extend (Ξ'-τ', _) →
                let Ξ' = fst Ξ'-τ' in
                let τ' = snd Ξ'-τ' in
                λε-χ. λρ. (rn-env ! Ξ' (fst ε-χ) ρ; rn Δ' Δ ρ τ' (snd ε-χ))) !
          in
          eval (τ₂; Γ' ∷ τ₁) t Δ' (rn-env Γ' env' f; χ)
      in
      cast ((Δ' :U 𝔽↓T !) → 𝔽↓t̃ (Δ; Δ') → ⟦ τ₁ ⟧ Δ' → ⟦ τ₂ ⟧ Δ', ⟦ τ ⟧ Δ,
            ap(U, τ'. ⟦ τ' ⟧ Δ, τ₁ ⇒ τ₂, _, fst pf), Λt)
    | 'App (τ₂-Γ'-τ₁-t-u, pf) →
      let τ₂ = fst τ₂-Γ'-τ₁-t-u in
      let Γ' = fst (snd τ₂-Γ'-τ₁-t-u) in
      let τ₁ = fst (snd (snd τ₂-Γ'-τ₁-t-u)) in
      let t = fst (snd (snd (snd τ₂-Γ'-τ₁-t-u))) in
      let u = snd (snd (snd (snd τ₂-Γ'-τ₁-t-u))) in
```

```
        let env' =
          cast(Π Γ Δ, Π Γ' Δ, ap(U, Ξ. Π Ξ Δ, _, _, sym (snd pf)), env)
        in
        let val : ⟦ τ₂ ⟧ Δ =
          (eval (τ₁ ⇒ τ₂; Γ') t Δ env') Δ (λ_. λx. x) (eval (τ₁; Γ') u Δ env')
        in
        cast(⟦ τ₂ ⟧ Δ, ⟦ τ ⟧ Δ, ap(U, τ'. ⟦ τ' ⟧ Δ, _, _, fst pf), val)) (τ; Γ)
in
let q-u : (τ :U Type !) →
      (f :U Form !) → (Γ :U 𝔽↓T !) →
      (match f as _ return U with
      | 'Ne (_, _) → 𝒩 τ Γ → ⟦ τ ⟧ Γ
      | 'Nf (_, _) → ⟦ τ ⟧ Γ → 𝒩 τ Γ) =
  λτ. (fix [Type as Ty view ι] q-u _ τ :
      (f :U Form !) → (Γ :U 𝔽↓T !) →
      (match f as _ return U with
      | 'Ne (_, _) → 𝒩 (ι ! τ) Γ → ⟦ (ι ! τ) ⟧ Γ
      | 'Nf (_, _) → ⟦ (ι ! τ) ⟧ Γ → 𝒩 (ι ! τ) Γ) =
    let q : (τ' :U Ty !) → (Γ' :U 𝔽↓T !) → ⟦ (ι ! τ') ⟧ Γ' → 𝒩 (ι ! τ') Γ' =
      λτ'. q-u ! τ' Nf
    in
    let u : (τ' :U Ty !) → (Γ' :U 𝔽↓T !) → 𝒩 (ι ! τ') Γ' → ⟦ (ι ! τ') ⟧ Γ' =
      λτ'. q-u ! τ' Ne
    in
    λf. λΓ.
      match f as f return
        let τ' : Type ! = in (fmap[Type](Ty, Type, ι, !, τ)) in
        match f as _ return U with
        | 'Ne (_, _) → 𝒩 τ' Γ → ⟦ τ' ⟧ Γ
        | 'Nf (_, _) → ⟦ τ' ⟧ Γ → 𝒩 τ' Γ
      with
      -- Unquote
      | 'Ne (_, _) →
        (match τ as τ' return
          let τ' : Type ! = in (fmap[Type](Ty, Type, ι, !, τ')) in
          𝒩 τ' Γ → ⟦ τ' ⟧ Γ
        with
        | 'Unit (_, _) → λ_. !
        | 'Product (τ₁-τ₂, _) →
          let τ₁ = fst τ₁-τ₂ in
          let τ₂ = snd τ₁-τ₂ in
          λm. (u τ₁ Γ ('VFst ((ι ! τ₁; (Γ; (ι ! τ₂; m))),
                              refl ((Ne; (ι ! τ₁; Γ)) : Form ! × (Type ! × 𝔽↓T !))));
               u τ₂ Γ ('VSnd ((ι ! τ₂; (Γ; (ι ! τ₁; m))),
                              refl ((Ne; (ι ! τ₂; Γ)) : Form ! × (Type ! × 𝔽↓T !)))))
        | 'Function (τ₁-τ₂, _) →
          let τ₁ = fst τ₁-τ₂ in
          let τ₂ = snd τ₁-τ₂ in
          let τ₁⇒τ₂ : Type ! = (ι ! τ₁) ⇒ (ι ! τ₂) in
          λm. λΔ. λρ. λχ.
            u τ₂ Δ ('VApp ((ι ! τ₂; (Δ; (ι ! τ₁; (pshf τ₁⇒τ₂ Γ m Δ ρ; q τ₁ Δ χ)))),
                           refl ((Ne; (ι ! τ₂; Δ)) : Form ! × (Type ! × 𝔽↓T !))))
        )
      -- Quote
      | 'Nf (_, _) →
        (match τ as τ return
          let τ' : Type ! = in (fmap[Type](Ty, Type, ι, !, τ)) in
          ⟦ τ' ⟧ Γ → 𝒩 τ' Γ
        with
        | 'Unit (_, _) → λ_. 'VOne (Γ, <*, <*, refl Γ>>)
        | 'Product (τ₁-τ₂, _) →
          let τ₁ = fst τ₁-τ₂ in
          let τ₂ = snd τ₁-τ₂ in
```

```
419           λp.
420             let t = fst p in
421             let u = snd p in
422             'VPair ((ι ! τ₁; (ι ! τ₂; (Γ; (q τ₁ Γ t; q τ₂ Γ u)))),
423                     <*, <<refl (ι ! τ₁), refl (ι ! τ₂)>, refl Γ>>)
424           | 'Function (τ₁-τ₂, _) →
425             let τ₁ = fst τ₁-τ₂ in
426             let τ₁' = ι ! τ₁ in
427             let τ₂ = snd τ₁-τ₂ in
428             let τ₂' = ι ! τ₂ in
429             λf.
430               let χ : ⟦ τ₁' ⟧ (Γ :: τ₁') =
431                 u τ₁ (Γ :: τ₁') ('VVar ((τ₁'; (Γ :: τ₁'; 'Ix0 ((τ₁'; Γ),
432                                                                 <refl τ₁', <refl Γ, refl τ₁'>>))),
433                                         <*, <refl τ₁', <refl Γ, refl τ₁'>>>))
434               in
435               let ↑ : 𝔽↓t̃ (Γ; Γ :: τ₁') =
436                 λτ'. λixΓ. 'IxS ((τ'; (Γ; (τ₁'; ixΓ))), <refl τ', <refl Γ, refl τ₁'>>)
437               in
438               'VLambda ((τ₁'; (τ₂'; (Γ; q τ₂ (Γ :: τ₁') (f (Γ :: τ₁') ↑ χ)))),
439                         <*, <<refl τ₁', refl τ₂'>, refl Γ>>))) ! τ
440   in
441   let q : (τ :U Type !) → (Γ :U 𝔽↓T !) → ⟦ τ ⟧ Γ → 𝒩 τ Γ =
442     λτ. q-u τ Nf
443   in
444   let u : (τ :U Type !) → (Γ :U 𝔽↓T !) → ℳ τ Γ → ⟦ τ ⟧ Γ =
445     λτ. q-u τ Ne
446   in
447   let nbe : (τ :U Type !) → (Γ :U 𝔽↓T !) → Term (τ; Γ) → 𝒩 τ Γ =
448     λτ. λΓ. λt.
449       let xs : Π Γ Γ =
450         (fix [𝔽↓T as Ctx view ι] xs _ Γ : Π (ι ! Γ) (ι ! Γ) =
451           match Γ as Γ return
452             let Γ' : 𝔽↓T ! = in (fmap[𝔽↓T](Ctx, 𝔽↓T, ι, !, Γ)) in
453             Π Γ' Γ'
454           with
455           | 'Empty (_, _) → !
456           | 'Extend (Γ'-τ, _) →
457             let Γ' = fst Γ'-τ in
458             let Γ'' = ι ! Γ' in
459             let τ = snd Γ'-τ in
460             let χ : ⟦ τ ⟧ (Γ'' :: τ) =
461               u τ (Γ'' :: τ) ('VVar ((τ; (Γ'' :: τ; 'Ix0 ((τ; Γ''),
462                                                           <refl τ, <refl Γ'', refl τ>>))),
463                                     <*, <refl τ, <refl Γ'', refl τ>>>))
464             in
465             let shift : (Δ :U 𝔽↓T !) → Π Δ Γ'' → Π Δ (Γ'' :: τ) =
466               (fix [𝔽↓T as Ctx view ι] shift _ Δ : Π (ι ! Δ) Γ'' → Π (ι ! Δ) (Γ'' :: τ) =
467                 match Δ as Δ return
468                   let Δ' : 𝔽↓T ! = in (fmap[𝔽↓T](Ctx, 𝔽↓T, ι, !, Δ)) in
469                   Π Δ' Γ'' → Π Δ' (Γ'' :: τ)
470                 with
471                 | 'Empty (_, _) → λ_. !
472                 | 'Extend (Δ'-τ', _) →
473                   let Δ' = fst Δ'-τ' in
474                   let τ' = snd Δ'-τ' in
475                   let ↑ : 𝔽↓t̃ (Γ''; Γ'' :: τ) =
476                     λτ''. λixΓ''.
477                       'IxS ((τ''; (Γ''; (τ; ixΓ''))), <refl τ'', <refl Γ'', refl τ>>)
478                   in
479                   λenv. (shift ! Δ' (fst env); rn (Γ'' :: τ) Γ'' ↑ τ' (snd env))
480               ) !
481             in
```

```
482           (shift (ι ! Γ') (xs ! Γ'); χ)
483         ) ! Γ
484     in
485     q τ Γ (Γ τ ⟦ t ⟧ Γ xs)
486 in
487 nbe 𝟙 · ('App ((𝟙; (·; (𝟙;
488                 ('Lambda ((𝟙; (𝟙; (·;
489                           'Var ((𝟙; (· ∷ 𝟙; 'Ix0 ((𝟙; ·), <*, <*, *>>))),
490                                 <*, <*, *>>))),
491                           <<*, *>, *>);
492                 'One (·, <*, *>))))),
493          <*, *>))
```