Towards Observational Type Theory

Thorsten Altenkirch and Conor McBride
School of Computer Science and Information Technology
University of Nottingham
{txa,ctm}@cs.nott.ac.uk

Abstract

Observational Type Theory (OTT) combines beneficial aspects of Intensional and Extensional Type Theory (ITT/ETT). It separates definitional equality, decidable as in ITT, and a substitutive propositional equality, capturing extensional equality of functions, as in ETT. Moreover, canonicity holds: any closed term is definitionally reducible to a canonical value.

Building on previous work by each author, this article reports substantial progress in the form of a simplified theory with a straightforward syntactic presentation, which we have implemented.

As well as simplifying reasoning about functions, OTT offers potential foundational benefits, e.g. it gives rise to a closed type theory encoding inductive datatypes.

1. Introduction

We introduce **Observational Type Theory**, a type theory in the tradition of Per Martin-Löf [8, 11], which combines beneficial aspects of Intensional and Extensional Type Theory. Like in Intensional Type Theory reduction is always terminating, and definitional equality and typechecking are decidable. Like Extensional Type Theory, propositional equality is extensional, i.e. two functions are equal, if there are equal pointwise or to put it differently, *if all observations about them agree*. At the same time propositional equality is substitutive, i.e. we can explicitly coerce between types which contain observationally equivalent subterms. This is achieved without affecting the computational behaviour of Type Theory, i.e. canonicity holds: any closed term is definitionally reducible to a canonical value.

We have implemented Observational Type Theory as a part of the forthcoming new release of Epigram [10], a dependently typed programming language and proof system. Indeed, in some respects the implementation goes beyond what we can currently justify, hence the word *towards* in the title. We will discuss these extensions in section 5. Other extensions clearly suggest themselves, e.g. the addition of quotient types.

Intensional Type Theory which is the basis of most implementations of Type Theory, e.g. Coq, LEGO, ALF and the first release of Epigram, supports reasoning about **data**, like the natural numbers, which are defined by the way inhabitants are constructed, but it fails to take proper account of **codata**, like functions or coinductive types (e.g. streams), which can only be understood by the way their inhabitants can be used.

Observational reasoning can be simulated in Intensional Type Theory by the use of setoids, i.e. types with an explicit equivalence relation. However, using setoids complicates the formalisation of mathematical theories because equality is not automatically substitutive. Additional complications arise if a setoid depends on a setoid, this arises for example in the formalisation of category theory [7], where the setoid of homsets depends on the setoid of objects[1]. Potentially more important than the formalisation of mathematical theories is the development of correct software for communicatin systems, which typically exhibit infinite behaviour and hence demand observational reasoning.

Why not Extensional Type Theory? Extensional Type Theory avoids all the shortcomings of Intensional Type Theory discussed above and has been implemented in NuPRL [3]. In Extensional Type Theory definitional equality and propositional equality are identified and as a consequence a term does not contain enough information for typechecking. This can be adressed by storing a derivation instead of only the term. However, computation cannot be guaranteed to terminate in the context of possibly inconsistent assumptions and hence we cannot trust the system to execute computations automatically—this is an essential advantage of Intensional Type Theory.

Observational Type Theory combines the advantages of both approaches. We show as an example that, as in Ex-

[1] Current formalisations avoid this problem by viewing objects as a type not a setoid. However, this makes it difficult to faithfully represent constructions like arrow categories which turn morphisms into objects.

tensional Type Theory, we are able to code inductive types, like the natural numbers, using $\mathcal{W}$-types and we can derive the associated induction principle. Using the framework of containers [1] we can implement any strictly positive inductive or coinductive type in a closed Observational Type Theory with $\mathcal{W}$-types, this can also be extended to inductive families. The fact that we can, in principle, reduce any feature of our Type Theory to a closed core theory has important pragmatic advantages: we can keep the trusted code base small.

Related work

Martin Hofmann [5] showed that that Extensional Type Theory is a conservative extension of Intensional Type Theory augmented by two extensionality axioms, this has been extended and simplified in the context of Coq [12] using heterogeneous equality [9], which also plays an important role in the construction presented here. However, adding axioms destroys the computational properties of Intensional Type Theory, in particular we lose canonicity. The first author showed [2] that we can have a decidable Type Theory with extensional propositional equality and canonicity by a model construction using an Intensional Type Theory with a proof-irrelevant universe of propositions.

Results

- In section 2 we give a syntactic presentation of proof-relevant Observational Type Theory. Such a presentation is absent from [2] which introduces proof-irrelevant OTT only by a model construction.

- In section 3 we justify proof-relevant OTT by a syntactic translation. While this is similar to the model construction in [2], the presentation is dramatically simplified by using heterogeneous equality [9].

- The translation presented in section 3 translates into a conventional Intensional Type Theory, a feature strongly demanded by Per Martin-Löf when seeing [2], which translates into an intensional type theory with a proof-irrelevant universe of propositions.

- As an example of applying proof-relevant OTT we show that the encoding of natural numbers as $\mathcal{W}$-types is admissible in OTT in section 4. We also discuss some limitations of this encoding with respect to the definitional equality.

- In section 5 we present full OTT, combining proof-irrelevance with a definitional rule for eliminating reflexive coercions—the usual computational behaviour of equality in Intensional Type Theory. We have implemented this system but leave a detailed investigation of its metatheoretic properties to further work.

2. Proof-Relevant OTT

We start with a basic dependent type theory TT (see figure 1) with $\Pi, \Sigma, \mathsf{O}, 1, 2, \mathcal{W}$ but without equality. Note that the type of Booleans (2) comes with a large elimination constant, which expresses that $\mathtt{tt}$ is different from $\mathtt{ff}$. This type theory can be extended either to an Intensional Type Theory (ITT) or Extensional Type Theory (ETT) by adding different forms of propositional equality. In either case we introduce a type former and the canonical proof of reflexivity:

$$\frac{\Gamma \vdash s_0 : S \quad \Gamma \vdash s_1 : S}{\Gamma \vdash s_0 = s_1 \ \mathbf{set}} \qquad \frac{\Gamma \vdash s : S}{\Gamma \vdash \overline{s} : s = s}$$

In the case of ETT we simply add the *equality reflection* rule:

$$\frac{\Gamma \vdash s_{01} : s_0 = s_1}{\Gamma \vdash s_0 \equiv s_1}$$

This rule identifies definitional and propositional equality, in the process we are throwing away the proof s_{01} for the equality, hence terms of ETT are not sufficent as evidence that a type is inhabited.

Moreover, a type is no guarantee that ETT terms will compute safely in the presence of false hypotheses. Given some $Q : 1 = 1 \to 1$, we gain the ability to type

$$(\lambda u{:}1.\ u\ u)\ (\lambda u{:}1.\ u\ u)$$

and other such broken programs.

Alternatively, to extend TT to ITT we add an eliminator together with a definitional equality:

$$\frac{\begin{array}{c}\Gamma \vdash s : S \quad \Gamma \vdash s' : S \quad \Gamma \vdash q : s = s' \\ \Gamma; x{:}S; y{:}S; z{:}x = y \vdash P[x, y, z] \ \mathbf{set} \\ \Gamma \vdash p : \Pi x{:}S.\ P[x, x, \overline{x}]\end{array}}{\begin{array}{c}\Gamma \vdash q\ (\mathsf{eqElim}\ x, y, z.\ P[x, y, z] \mid p) : P[s, s', q] \\ \overline{s}\ (\mathsf{eqElim}\ x, y, z.\ P[x, y, z] \mid p) \equiv p\ s : P[s, s, \overline{s}]\end{array}}$$

This eliminator gets stuck unless the equation really holds definitionally, so computation remains safe in the presence of false hypotheses, but it also gets stuck in the presence of axioms. Moreover, it has been shown by Hofmann and Streicher [6] that this eliminator is insufficent to prove uniqueness of equality proofs, i.e. whether for $s, s' : s_0 = s_1$ we can construct $\mathsf{irr}\, s\, s' : s = s'$. This can be fixed by either adding a second eliminator, traditionally called K, or by formalising a heterogeneous equality as suggested in [9]. The latter has the practial advantage of avoiding the proliferation of equality coercions.

Here, we introduce OTT as yet another extension of TT. OTT is intensional in character, i.e. terms are evidence for inhabitation. OTT uses heterogeneous equality but more fundamentally it differs from both ITT and ETT in that reflexivity is not the canonical constructor for equality proofs but rather every constructor c comes equipped with a constructor of equality $c^{=}$ which witnesses that c preserves equality. In the case of λ the constant $\lambda^{=}$ implies the principle of extensionality. Using these constants we can reflect equality derivations in ETT as explicit proof objects in OTT.

2.1. Observational Equality

We now introduce sets representing proofs of equality, between sets and between inhabitants of sets. The idea is to introduce sets which reflect the definitional equality. We sometimes need to formulate equations between elements of types which are provably equal: in Extensional Type Theory, the equality reflection rule allows us to deduce that these elements have a common type; we avoid this by making the term-level equality *heterogeneous*.

$$\frac{\Gamma \vdash S_0 \ \mathbf{set} \quad \Gamma \vdash S_1 \ \mathbf{set}}{\Gamma \vdash S_0 = S_1 \ \mathbf{set}} \qquad \frac{\Gamma \vdash s_0 : S_0 \quad \Gamma \vdash s_1 : S_1}{\Gamma \vdash (s_0 : S_0) = (s_1 : S_1) \ \mathbf{set}}$$

Each of these is an equivalence in the appropriate sense.

$$\frac{\Gamma \vdash S \ \mathbf{set}}{\Gamma \vdash \overline{S} : S = S} \qquad \frac{\Gamma \vdash S_{01} : S_0 = S_1}{\Gamma \vdash S_{01}^{\smile} : S_1 = S_0}$$

$$\frac{\Gamma \vdash S_{01} : S_0 = S_1 \quad \Gamma \vdash S_{12} : S_1 = S_2}{\Gamma \vdash S_{01} \circ S_{12} : S_0 = S_2}$$

$$\frac{\Gamma \vdash s : S}{\Gamma \vdash \overline{s} : (s : S) = (s : S)} \qquad \frac{\Gamma \vdash s_{01} : (s_0 : S_0) = (s_1 : S_1)}{\Gamma \vdash s_{01}^{\smile} : (s_1 : S_1) = (s_0 : S_0)}$$

$$\frac{\Gamma \vdash s_{01} : (s_0 : S_0) = (s_1 : S_1) \quad \Gamma \vdash s_{12} : (s_1 : S_1) = (s_2 : S_2)}{\Gamma \vdash s_{01} \circ s_{12} : (s_0 : S_0) = (s_2 : S_2)}$$

You may notice that we often write equations which differ only in the subscripts of their schematic variables. This pattern is set to continue, hence we introduce the notation $\|\Phi[x, y]\|_i^j$ to stand for $\Phi[x_i, y_i] = \Phi[x_j, y_j]$ for any formula Φ. For example, the above transitivity rule becomes

$$\frac{\Gamma \vdash s_{01} : \|s : S\|_0^1 \quad \Gamma \vdash s_{12} : \|s : S\|_1^2}{\Gamma \vdash s_{01} \circ s_{12} : \|s : S\|_0^2}$$

In order to manipulate compound types, we shall need to be able to project out the equality proofs for their components. We introduce, for each $B \in \{\Pi, \Sigma, \mathcal{W}\}$

$$\frac{\Gamma \vdash Q : \|Bx : S.\, T[x]\|_0^1}{\Gamma \vdash Q\pi : \|S\|_0^1} \qquad \frac{\Gamma \vdash Q : \|Bx : S.\, T[x]\|_0^1 \quad \Gamma \vdash q : \|s : S\|_0^1}{\Gamma \vdash Q(q) : \|T[s]\|_0^1}$$

2.2. Type-directed Coercion

Just as the conversion rule allows terms to pass implicitly between *definitionally* equal types, so *provably* equal types have an *explicit* coercion between them. Moreover, this coercion is *coherent* in that its input is provably equal to its output.

$$\frac{\Gamma \vdash S_{01} : S_0 = S_1 \quad \Gamma \vdash s_0 : S_0}{\Gamma \vdash s_0[S_{01}\rangle_{S_0}^{S_1} : S_1}$$

$$\frac{\Gamma \vdash S_{01} : S_0 = S_1 \quad \Gamma \vdash s_0 : S_0}{\Gamma \vdash s_0[\![S_{01}]\!\rangle_{S_0}^{S_1} : (s_0 : S_0) = (s_0[S_{01}\rangle_{S_0}^{S_1} : S_1)}$$

We shall drop the annotations on coercions where they are clear. For convenience, we shall also abbreviate the symmetric images of these operators, carefully respecting the order of the type equation as we go:

$$\langle S_{01}]s_1 \mapsto s_1[S_{01}^{\smile}\rangle : S_0$$
$$\langle\!\langle S_{01}]\!]s_1 \mapsto (s_1[\![S_{01}^{\smile}]\!\rangle)^{\smile} : (\langle S_{01}]s_1 : S_0) = (s_1 : S_1)$$

The coercion operator computes to a mapping from one canonical type to another, provided they are *compatible*, ie., that they have the same type-former. For constant types, coercion is always the identity:

$$\frac{\Gamma \vdash s : C \quad \Gamma \vdash Q : C = C}{\Gamma \vdash s[Q\rangle \equiv s : C} \quad C \in \{\mathsf{O}, 1, 2\}$$

We coerce a function by translating its argument right-to-left and its result left-to-right:

$$\frac{\Gamma \vdash f_0 : \Pi x_0 : S_0.\, T_0[x_0] \quad \Gamma \vdash Q : \|\Pi x : S.\, T[x]\|_0^1}{\begin{array}{r}\Gamma \vdash f_0[Q\rangle \equiv \lambda x_1 : S_1.\, f_0\,(\langle Q\pi]x_1) \\ \qquad\qquad [Q(\,\langle\!\langle Q\pi]\!]x_1)\rangle \\ : \Pi x_1 : S_1.\, T_1[x_1]\end{array}}$$

Coercion for a pair just goes element by element:

$$\frac{\Gamma \vdash p_0 : \Sigma x_0 : S_0.\, T_0[x_0] \quad \Gamma \vdash Q : \|\Sigma x : S.\, T[x]\|_0^1}{\begin{array}{r}\Gamma \vdash p_0[Q\rangle \equiv \begin{pmatrix} p_0\,\pi_0[Q\pi\rangle\,, \\ p_0\,\pi_1[Q(p_0\,\pi_0[\![Q\pi]\!\rangle)\rangle\end{pmatrix} \\ : \Sigma x_1 : S_1.\, T_1[x_1]\end{array}}$$

For a tree, we obtain a recursive map from the $\mathcal{W}$-type elimination operator.

$$\frac{\Gamma \vdash w_0 : \mathcal{W}x_0 : S_0.\, T_0[x_0] \quad \Gamma \vdash Q : \|\mathcal{W}x : S.\, T[x]\|_0^1}{\begin{array}{l}\Gamma \vdash w_0[Q\rangle \equiv w_0 \left(\begin{array}{l}\mathsf{rec}\, w.\, \mathcal{W}x_1 : S_1.\, T_1[x_1] \ \mid \\ \lambda s_0 : S_0; \\ \quad f_0 : T_0[s_0] \to \mathcal{W}x_0 : S_0.\, T_0[x_0]; \\ \quad h : T_0[s_0] \to \mathcal{W}x_1 : S_1.\, T_1[x_1]. \\ \mathsf{node}\,(s_0[Q\pi\rangle)\,(\lambda t_1 : T_1[s_0[Q\pi\rangle].\, \\ \qquad\qquad h\,(\langle Q(s_0[\![Q\pi]\!\rangle)]t_1))\end{array}\right) \\ : \mathcal{W}x_1 : S_1.\, T_1[x_1]\end{array}}$$

contexts

$$\frac{}{\mathcal{E} \vdash \textbf{valid}} \qquad \frac{\Gamma \vdash S\ \textbf{set}}{\Gamma;\ x{:}S \vdash \textbf{valid}}$$

---

binders

$$\frac{\Gamma;\ x{:}S \vdash T[x]\ \textbf{set}}{\Gamma \vdash Bx{:}S.\ T[x]\ \textbf{set}} \quad B \in \{\Pi, \Sigma, \mathcal{W}\}$$

constants

$$\frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash C\ \textbf{set}} \quad C \in \{\mathsf{O}, 1, 2\}$$

large elimination

$$\frac{\Gamma \vdash b : 2 \quad \Gamma \vdash T\ \textbf{set} \quad \Gamma \vdash F\ \textbf{set}}{\Gamma \vdash b\ (\mathsf{Case}\ T; F)\ \textbf{set}}$$

---

variables

$$\frac{\Gamma;\ x{:}S;\ \Delta \vdash \textbf{valid}}{\Gamma;\ x{:}S;\ \Delta \vdash x : S}$$

functions

$$\frac{\Gamma;\ x{:}S \vdash t[x] : T[x]}{\Gamma \vdash \lambda x{:}S.\ t[x] : \Pi x{:}S.\ T[x]}$$

$$\frac{\Gamma \vdash f : \Pi x{:}S.\ T[x] \quad \Gamma \vdash s : S}{\Gamma \vdash f\ s : T[s]}$$

tuples

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s]}{\Gamma \vdash (s, t) : \Sigma x{:}S.\ T[x]} \qquad \frac{\Gamma \vdash p : \Sigma x{:}S.\ T[x]}{\begin{array}{l}\Gamma \vdash p\ \pi_0 : S \\ \Gamma \vdash p\ \pi_1 : T[p\ \pi_0]\end{array}}$$

trees

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash f : T[s] \to \mathcal{W}x{:}S.\ T[x]}{\mathsf{node}\ s\ f : \mathcal{W}x{:}S.\ T[x]}$$

$$\frac{\begin{array}{l}\Gamma \vdash w : \mathcal{W}x{:}S.\ T[x] \\ \Gamma;\ w{:}\mathcal{W}x{:}S.\ T[x] \vdash P[w]\ \textbf{set} \\ \Gamma \vdash p : \Pi s{:}S;\ f{:}T[s] \to \mathcal{W}x{:}S.\ T[x];\ h{:}\Pi t{:}T[s].\ P[f\ t]. \\ \qquad P[\mathsf{node}\ s\ f]\end{array}}{\Gamma \vdash w\ (\mathsf{rec}\ w.\ P[w] \mid p) : P[w]}$$

constants

$$\frac{\Gamma \vdash \textbf{valid}}{\begin{array}{l}\Gamma \vdash () : 1 \\ \Gamma \vdash \mathsf{tt} : 2 \\ \Gamma \vdash \mathsf{ff} : 2\end{array}} \qquad \frac{\Gamma \vdash x : \mathsf{O} \quad \Gamma \vdash S\ \textbf{set}}{\Gamma \vdash x\ (\text{Œ}\ S) : S}$$

$$\frac{\Gamma \vdash b : 2 \qquad \Gamma;\ b{:}2 \vdash P[b]\ \textbf{set} \quad \Gamma \vdash t : P[\mathsf{tt}] \quad \Gamma \vdash f : P[\mathsf{ff}]}{\Gamma \vdash b\ (\mathsf{case}\ b.\ P[b] \mid t; f) : P[b]}$$

conversion

$$\frac{\Gamma \vdash s : S_0 \quad \Gamma \vdash S_0 \equiv S_1}{\Gamma \vdash s : S_1}$$

equivalence closure

$$\frac{\Gamma \vdash S\ \textbf{set}}{\Gamma \vdash S \equiv S} \quad \frac{\Gamma \vdash S_0 \equiv S_1}{\Gamma \vdash S_1 \equiv S_0} \quad \frac{\begin{array}{l}\Gamma \vdash S_0 \equiv S_1 \\ \Gamma \vdash S_1 \equiv S_2\end{array}}{\Gamma \vdash S_0 \equiv S_2}$$

structural closure

$$\frac{\Gamma \vdash S_0 \equiv S_1 \quad \Gamma;\ x{:}S_0 \vdash T_0[x] \equiv T_1[x]}{\Gamma \vdash Bx{:}S_0.\ T_0[x] \equiv Bx{:}S_1.\ T_1[x]} \quad B \in \{\Pi, \Sigma, \mathcal{W}\}$$

$$\frac{\Gamma \vdash b_0 \equiv b_1 : 2 \quad \Gamma \vdash T_0 \equiv T_1 \quad \Gamma \vdash F_0 \equiv F_1}{\Gamma \vdash b_0\ (\mathsf{Case}\ T_0; F_0) \equiv b_1\ (\mathsf{Case}\ T_1; F_1)}$$

large elimination $\quad (\mapsto \subseteq \equiv)$

$$\frac{\Gamma \vdash T\ \textbf{set} \quad \Gamma \vdash F\ \textbf{set}}{\begin{array}{l}\Gamma \vdash \mathsf{tt}\ (\mathsf{Case}\ T; F) \mapsto T \\ \Gamma \vdash \mathsf{ff}\ (\mathsf{Case}\ T; F) \mapsto F\end{array}}$$

---

equivalence closure

$$\frac{\Gamma \vdash s : S}{\Gamma \vdash s \equiv s : S} \quad \frac{\Gamma \vdash s_0 \equiv s_1 : S}{\Gamma \vdash s_1 \equiv s_0 : S} \quad \frac{\begin{array}{l}\Gamma \vdash s_0 \equiv s_1 : S \\ \Gamma \vdash s_1 \equiv s_2 : S\end{array}}{\Gamma \vdash s_0 \equiv s_2 : S}$$

η-rules

$$\frac{\Gamma;\ x{:}S \vdash f_0\ x \equiv f_1\ x : T[x]}{\Gamma \vdash f_0 \equiv f_1 : \Pi x{:}S.\ T[x]}$$

$$\frac{\Gamma \vdash p_0\ \pi_0 \equiv p_1\ \pi_0 : S \quad \Gamma \vdash p_0\ \pi_1 \equiv p_1\ \pi_1 : T[p_0\ \pi_0]}{\Gamma \vdash p_0 \equiv p_1 : \Sigma x{:}S.\ T[x]}$$

$$\frac{\Gamma \vdash s_0 : C \quad \Gamma \vdash s_1 : C}{\Gamma \vdash s_0 \equiv s_1 : C} \quad C \in \{\mathsf{O}, 1\}$$

structural closure

$$\frac{\Gamma \vdash s_0 \equiv s_1 : S \quad \Gamma \vdash f_0 \equiv f_1 : T[s_0] \to \mathcal{W}x{:}S.\ T[x]}{\Gamma \vdash \mathsf{node}\ s_0\ f_0 \equiv \mathsf{node}\ s_1\ f_1 : \mathcal{W}x{:}S.\ T[x]}$$

plus structural rules for elimination forms

β-rules $\quad (\mapsto \subseteq \equiv)$

$$\frac{\Gamma;\ x{:}S \vdash t[x] : T[x] \quad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x{:}S.\ t[x])\ s \mapsto t[s] : T[s]} \qquad \frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s]}{\begin{array}{l}\Gamma \vdash (s, t)\ \pi_0 \mapsto s : S \\ \Gamma \vdash (s, t)\ \pi_1 \mapsto t : T[s]\end{array}}$$

$$\frac{\begin{array}{l}\Gamma \vdash s : S \quad \Gamma \vdash f : T[s] \to \mathcal{W}x{:}S.\ T[x] \\ \Gamma;\ w{:}\mathcal{W}x{:}S.\ T[x] \vdash P[w]\ \textbf{set} \\ \Gamma \vdash p : \Pi s{:}S;\ f{:}T[s] \to \mathcal{W}x{:}S.\ T[x];\ h{:}\Pi t{:}T[s].\ P[f\ t]. \\ \qquad P[\mathsf{node}\ s\ f]\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{node}\ s\ f\ (\mathsf{rec}\ w.\ P[w] \mid p) \mapsto \\ \qquad p\ s\ f\ (\lambda t{:}T[s].\ f\ t\ (\mathsf{rec}\ w.\ P[w] \mid p)) : P[\mathsf{node}\ s\ f]\end{array}}$$

$$\frac{\Gamma;\ b{:}2 \vdash P[b]\ \textbf{set} \quad \Gamma \vdash t : P[\mathsf{tt}] \quad \Gamma \vdash f : P[\mathsf{ff}]}{\begin{array}{l}\Gamma \vdash \mathsf{tt}\ (\mathsf{case}\ b.\ P[b] \mid t; f) \mapsto t : P[\mathsf{tt}] \\ \Gamma \vdash \mathsf{ff}\ (\mathsf{case}\ b.\ P[b] \mid t; f) \mapsto f : P[\mathsf{ff}]\end{array}}$$

conversion

$$\frac{\Gamma \vdash s_0 \equiv s_1 : S_0 \quad \Gamma \vdash S_0 \equiv S_1}{\Gamma \vdash s_0 \equiv s_1 : S_1}$$

Figure 1. TT, a type theory without propositional equality

The domain coercion tells us how to translate the source shape $s_0 : S_0$ to the target shape; the codomain coercion translates the target position of each subtree to the source position of the subtree from which it is recursively obtained, via the inductive hypothesis h. It is just as if we had defined the coercion by this guarded recursive program:

$$\begin{array}{l}\mathsf{node}\ s_0\ f_0\ [Q\rangle \mapsto \mathsf{node}\ (\mathsf{from}\ s_0)\ (\lambda t_1.\ h\ (\mathsf{to}\ t_1)\ [Q\rangle) \\ \quad \text{where}\ \ \mathsf{from}\ s_0 \mapsto \quad s_0[Q\pi\rangle \\ \qquad\qquad \mathsf{to}\ t_1 \quad \mapsto \langle Q(s_0[\![Q\pi\rangle\!])]t_1\end{array}$$

The point here is that coercions of canonical values between *compatible always* compute, regardless of the particular equational justification for that coercion. We can guarantee compatibility for each type equation provable in the empty context, and hence we shall have the *canonicity* property we need.

Meanwhile, in the presence of $Q : 1 = 1 \to 1$, we cannot type $\lambda u{:}1.\ u\ u$, but we do have

$$\begin{array}{l}(\lambda u{:}1.\ u[Q\rangle\ u)\ (\langle Q](\lambda u{:}1.\ u[Q\rangle\ u)) : 1 \\ \quad \mapsto (\langle Q](\lambda u{:}1.\ u[Q\rangle\ u))[Q\rangle\ (\langle Q](\lambda u{:}1.\ u[Q\rangle\ u))\end{array}$$

which then stops because Q equates incompatible types.

2.3. Structural Equality Proofs

We now reflect the η- and structural equality rules by introducing corresponding term-formers for equality proofs. Wherever the equality rules bind parameters, our equality proofs abstract over pairs of provably equal values. It will prove convenient to write $\{x{:}S\}_0^1$ as an abbreviation for the sequence of bindings $x_0 : S_0;\ x_1 : S_1;\ x_{01} : x_0 = x_1$. We may write the corresponding argument sequence $s_0\ s_1\ s_{01}$ as $\vec{s_{01}}$.

We establish equality between compatible types by providing the means to construct the coercion between them. For $B \in \{\Pi, \Sigma, \mathcal{W}\}$, we have

$$\frac{\Gamma \vdash S_{01} : \|S\|_0^1 \quad \Gamma \vdash T_{01} : \Pi\{x{:}S\}_0^1.\ \|T[x]\|_0^1}{\Gamma \vdash B^{=}\ S_{01}\ T_{01} : \|Bx{:}S.\ T[x]\|_0^1}$$

We also add constant proofs for constants:

$$\frac{\Gamma \vdash \mathbf{valid}}{\Gamma \vdash C^{=} : C = C} \quad C \in \{\mathsf{O}, 1, 2\}$$

Of course, these already follow by reflexivity, but we shall later show how to compute reflexivity proofs, so we shall need some values for the constants.

For the large elimination, we have

$$\frac{\Gamma \vdash b_{01} : \|b\|_0^1 \quad \Gamma \vdash T_{01} : \|T\|_0^1 \quad \Gamma \vdash F_{01} : \|F\|_0^1}{\Gamma \vdash b_{01}\ (\mathsf{Case}^{=}\ T_{01}; F_{01}) : \|b\ (\mathsf{Case}\ T; F)\|_0^1}$$

The pattern continues at the level of values. Again, we have constants $()^{=}, \mathsf{tt}^{=}, \mathsf{ff}^{=}$. For functions, we have this:

$$\frac{\Gamma \vdash S_{01} : \|S\|_0^1 \quad \Gamma \vdash t_{01} : \Pi\{x{:}S\}_0^1.\ \|t[x]\|_0^1}{\Gamma \vdash \lambda^{=}\ S_{01}\ t_{01} : \|\lambda x{:}S.\ t[x]\|_0^1}$$

$$\frac{\Gamma \vdash f_{01} : \|f{:}\Pi x{:}S.\ T[x]\|_0^1 \quad \Gamma \vdash s_{01} : \|s{:}S\|_0^1}{\Gamma \vdash f_{01}\ {}^{=}\ s_{01} : \|f\ s{:}T[s]\|_0^1}$$

Note that the $\lambda^{=}$ constructor expresses the *extensional* equality between functions, given that the definitional equality supports η-expansion.

For pairs, we supply

$$\frac{\Gamma \vdash s_{01} : \|s{:}S\|_0^1 \quad \Gamma \vdash t_{01} : \|t{:}T[s]\|_0^1}{\Gamma \vdash (s_{01}, t_{01}) : \|(s,t){:}\Sigma x{:}S.\ T[x]\|_0^1}$$

$$\frac{\Gamma \vdash p_{01} : \|p{:}\Sigma x{:}S.\ T[x]\|_0^1}{\begin{array}{l}\Gamma \vdash p_{01}\ \pi_0^{=} : \|p\ \pi_0{:}S\|_0^1 \\ \Gamma \vdash p_{01}\ \pi_1^{=} : \|p\ \pi_1{:}T[p\ \pi_0]\|_0^1\end{array}}$$

Reflexivity provides sufficient introduction behaviour for the constant types, but let us have structural rules for the eliminators[2]:

$$\frac{\Gamma \vdash x_{01} : \|x{:}\mathsf{O}\|_0^1 \quad \Gamma \vdash S_{01} : \|S\|_0^1}{\Gamma \vdash x_{01}\ (\text{Œ}^{=}\ S_{01}) : \|x\ (\text{Œ}\ S){:}S\|_0^1}$$

$$\frac{\begin{array}{l}\Gamma \vdash b_{01} : \|b{:}2\|_0^1 \quad \Gamma \vdash P_{01} : \Pi\{x{:}2\}_0^1.\ \|P[x]\|_0^1 \\ \Gamma \vdash t_{01} : \|t{:}P[\mathsf{tt}]\|_0^1 \quad \Gamma \vdash f_{01} : \|t{:}P[\mathsf{ff}]\|_0^1\end{array}}{\begin{array}{l}\Gamma \vdash b_{01}\ (\mathsf{case}^{=}\ P_{01} \mid t_{01}; f_{01}) \\ \qquad : \|b\ (\mathsf{case}\ x.\ P[x] \mid t; f){:}P[b]\|_0^1\end{array}}$$

Finally, for trees, perhaps you will allow us to omit some of the bureaucratic details:

$$\frac{\Gamma \vdash s_{01} : \|s{:}S\|_0^1 \quad \Gamma \vdash f_{01} : \|f{:}\mathcal{W}x{:}S.\ T[x]\|_0^1}{\Gamma \vdash \mathsf{node}^{=}\ s_{01}\ f_{01} : \|\mathsf{node}\ s\ f : \mathcal{W}x{:}S.\ T[x]\|_0^1}$$

$$\frac{\begin{array}{c}\Gamma \vdash w_{01} : \|w{:}\mathcal{W}x{:}S.\ T[x]\|_0^1 \\ \Gamma \vdash P_{01} : \Pi\{y{:}\mathcal{W}x{:}S.\ T[x]\}_0^1.\ \|P[y]\|_0^1 \quad \Gamma \vdash p_{01} : \|p\|_0^1\end{array}}{\Gamma \vdash w_{01}\ (\mathsf{rec}^{=}\ P_{01} \mid p_{01}) : \|w\ (\mathsf{rec}\ y.\ P[y] \mid p){:}P[w]\|_0^1}$$

We have given structural proof rules for the type theory without equality, but when are equality types and equality proofs equal? Intuitively, equations are equal if their respective sides are equal, hence we add:

$$\frac{\Gamma \vdash S_{01} : \|S\|_0^1 \quad \Gamma \vdash T_{01} : \|T\|_0^1}{\Gamma \vdash S_{01} =^{=} T_{01} : \|S = T\|_0^1}$$

$$\frac{\Gamma \vdash s_{01} : \|s{:}S\|_0^1 \quad \Gamma \vdash t_{01} : \|t{:}T\|_0^1}{\Gamma \vdash s_{01} =^{=} t_{01} : \|(s{:}S) = (t{:}T)\|_0^1}$$

[2]The rule for (Œ) is clearly derivable, but we include it anyway to maintain our systematic approach.

Now, we have one equality proof former for each of our original term formers, but these proof formers are now term formers too! Are we in a vicious circle? Fortunately not: to figure out which equality proofs should be provably equal, consider the *observations* which are performed on them. We only use equality proofs to construct other equality proofs and to coerce values between equal types; coercion does not inspect the proof of the equation; hence all proofs of an equation may be considered equal. We therefore add

$$\frac{\Gamma \vdash Q_0 : S = S \quad \Gamma \vdash Q_1 : S = S}{\Gamma \vdash \mathsf{Irr}(S)\ Q_0\ Q_1 : \|Q\|_0^1}$$

$$\frac{\Gamma \vdash q_0 : (s{:}S) = (s{:}S) \quad \Gamma \vdash q_1 : (s{:}S) = (s{:}S)}{\Gamma \vdash \mathsf{irr}(s{:}S)\ q_0\ q_1 : \|q\|_0^1}$$

It is sufficient to consider proofs of reflexive equations here, because any proof of any equation may be coerced to an equal proof of a reflexive equation: if $Q : S = T$, then $\overline{S} =^{=} Q^{\smile} : (S = T) = (S = S)$.

What we have done here is to reflect equality in the opposite direction to ETT. We have provided a checkable term language for those equality derivations which require more than just computation rules. Every equality derivation in ETT becomes an equality proof in OTT: structural rules map to structural proofs, comptational rules map to reflexive proofs, equality reflection just pastes in the proof which it previously concealed.

3. Justifying Proof-Relevant OTT

In this section we give a syntactic translation from Proof-Relevant OTT into the pure theory TT. Formally, this translation can be understood as a model construction where contexts Γ are translated into extended contexts Γ_0^1 as prescribed below, thus the respect operator is definable in the target theory. Since reductions and definitional equalities in OTT are simulated by their counterparts in the translated theory, we can conclude that OTT shares TT's metatheoretic properties, i.e. normalisation and decidability of definitional equality. Consequently, we do not have to implement OTT via this translation but can instead reduce terms and decide equality for OTT directly.

We may construct our observational equality for canonical types and values by recursion on canonical types (and then, in the case of $\mathcal{W}$-types, on the values they contain). We exactly pack up the proofs required by the corresponding structural rules. For each $B \in \{\Pi, \Sigma, \mathcal{W}\}$, we take

$$\|Bx{:}S.\ T[x]\|_0^1 \mapsto \|S\|_0^1 \times \Pi\{x{:}S\}_0^1.\ \|T[x]\|_0^1$$

Equality for equal constant types $C \in \{\mathsf{O}, 1, 2\}$ is trivial

$$C = C \mapsto 1$$

Equality for incompatible types yields O. We construct equality for values as follows:

$$\begin{array}{ll}
\|f{:}\Pi x{:}S.\ T[x]\|_0^1 & \mapsto \|S\|_0^1 \times \Pi\{x{:}S\}_0^1.\ \|f\ x{:}T[x]\|_0^1 \\
\|p{:}\Sigma x{:}S.\ T[x]\|_0^1 & \mapsto \|p\ \pi_0{:}S\|_0^1 \times \|p\ \pi_1{:}T[p\ \pi_0]\|_0^1 \\
\|z{:}\mathsf{O}\|_0^1 & \mapsto 1 \\
\|u{:}1\|_0^1 & \mapsto 1 \\
\|\mathsf{tt}{:}2\|_0^1 & \mapsto 1 \\
\|\mathsf{ff}{:}2\|_0^1 & \mapsto 1 \\
(\mathsf{tt}{:}2) = (\mathsf{ff}{:}2) & \mapsto \mathsf{O} \\
(\mathsf{ff}{:}2) = (\mathsf{tt}{:}2) & \mapsto \mathsf{O} \\
\|\mathsf{node}\ s\ f : \mathcal{W}x{:}S.\ T[x]\|_0^1 & \mapsto \|s{:}S\|_0^1 \times \\
\quad (\|T[s]\|_0^1 \times \Pi\{t{:}T[s]\}_0^1.\ \|f\ t{:}\mathcal{W}x{:}S.\ T[x]\|_0^1)
\end{array}$$

Observe that, as we might hope, our definition yields

$$\|\mathsf{node}\ s\ f\|_0^1 \equiv \|s\|_0^1 \times \|f\|_0^1$$

but we give the expanded form to show that the recursive definition is suitably *guarded*.

Now that equations *compute* in terms of other types, we do not need any extra work to explain when equations are equal. Notice also that all our equations are given ultimately by types constructed from Π, Σ, O and 1. It is correspondingly direct, albeit laborious, to show that all proofs of equations are provably equal and thus implement the Irr and irr operations. We now implement the remaining operations.

3.1. Structural Proof Rules

By construction, we may implement the structural equality rules for the canonical type- and term-formers. We have also ensured that the components we collect for each equality proof on canonical objects are exactly those we need to implement the non-canonical operations. On types, we have our projections for the domains and ranges of equal compound types:

$$Q\pi \mapsto Q\ \pi_0 \qquad Q(q) \mapsto Q\ \pi_1\ \vec{q}$$

Our type-directed coercions will now reduce accordingly.

To complete the story on the type level, we must give the structural proof for the large elimination. We use a double small elimination on the elements themselves to compute which of the two proofs we need to deploy. In the off-diagonal cases, the proof that the two scrutinees coincide inhabits O. We summarize the proof term in the form of a functional program which is readily constructable:

For $b_{01} : \|b\|_0^1, T_{01} : \|T\|_0^1, F_{01} : \|F\|_0^1$,
$b_{01}\ (\mathsf{Case}^{=}\ T_{01}; F_{01}) \mapsto \mathsf{resp}_{\mathsf{Case}}\ b_0\ b_1\ b_{01}$ where
$\mathsf{resp}_{\mathsf{Case}} : \Pi\{b{:}2\}_0^1.\ \|b\ (\mathsf{Case}\ T; F)\|_0^1$
$\mathsf{resp}_{\mathsf{Case}}\ \mathsf{tt}\ \mathsf{tt}\ () \mapsto T_{01}$
$\mathsf{resp}_{\mathsf{Case}}\ \mathsf{tt}\ \mathsf{ff}\ z \mapsto z\ (\text{Œ}\ T_0 = F_1)$
$\mathsf{resp}_{\mathsf{Case}}\ \mathsf{ff}\ \mathsf{tt}\ z \mapsto z\ (\text{Œ}\ T_1 = F_0)$
$\mathsf{resp}_{\mathsf{Case}}\ \mathsf{ff}\ \mathsf{ff}\ () \mapsto F_{01}$

Meanwhile, at the value level, application becomes application and projection becomes projection

$$(S_{01}, f_{01})^{=}\ s_{01} \mapsto f_{01}\ \vec{s_{01}} \qquad \begin{array}{l} p_{01}\ \overline{\pi_0} \mapsto p_{01}\ \pi_0 \\ p_{01}\ \overline{\pi_1} \mapsto p_{01}\ \pi_1 \end{array}$$

The proof for the small elimination on 2 goes like the proof for the large elimination, so we omit it, and if you are lucky enough to possess *two* elements of the O type, you may use either to prove any structural rule you like!

For the recursor on $\mathcal{W}$-types, we must prove that equal inductions on equal trees yield equal results. Again, we abbreviate a cumbersome double elimination by a functional program whose recursion is clearly guarded.

$$\begin{array}{l} \text{For } w_{01} : \|w{:}\mathcal{W}x{:}S_0.\ T_0[x]\|^1_0,\ p_{01} : \|p\|^1_0, \\ w_{01}\ (\mathsf{rec}^{=}\ P_{01} \mid p_{01}) \mapsto \mathsf{resp}_{\mathsf{rec}}\ w_0\ w_1\ w_{01} \quad \text{where} \\ \quad \mathsf{resp}_{\mathsf{rec}} : \Pi\{w{:}\mathcal{W}x{:}S.\ T[x]\}^1_0. \\ \qquad\qquad \|w\ (\mathsf{rec}\ y.\ P[y] \mid p){:}P[w]\|^1_0 \\ \quad \mathsf{resp}_{\mathsf{rec}}\ (\mathsf{node}\ s_0\ f_0)\ (\mathsf{node}\ s_1\ f_1)\ (s_{01}, f_{01}\ \text{as}\ (T^s_{01}, \_)) \\ \qquad \mapsto p_{01}{}^{=}\ s_{01}{}^{=}\ f_{01}{}^{=} \\ \qquad\quad (\lambda^{=}\ T^s_{01}\ (\lambda\{t{:}T[s]\}^1_0. \\ \qquad\qquad \mathsf{resp}_{\mathsf{rec}}\ (f_0\ t_0)\ (f_1\ t_1)\ (f_{01}{}^{=}\ t_{01}))) \end{array}$$

3.2. Reflexivity via Respect

Given the structural operations, we can show how to construct reflexivity proofs by mapping each term former to its equality proof former. We shall need to generalise this operation in order to push it under binders: each bound variable in the source term becomes a pair of provably equal bound variables in the proof. In particular, we must show that terms remain invariant with respect to equations on *context extensions*.

If $\Gamma; \Delta \vdash \textbf{valid}$, we say Δ is a Γ-extension. Given such a Δ, we may construct (given fresh names), the Γ-extension Δ^1_0 and term sequences Δ_i as follows

$$\begin{array}{rl} \mathcal{E}^1_0 & \mapsto \mathcal{E} \\ (\Delta; x{:}S[\Delta])^1_0 & \mapsto \Delta^1_0; x_0{:}S[\Delta_0]; x_1{:}S[\Delta_1]; x_{01}{:}\|x\|^1_0 \\ \mathcal{E}_i & \mapsto \varepsilon \\ (\Delta; x{:}S[\Delta])_i & \mapsto \Delta_i; x{:}S[\Delta_i] \end{array}$$

The idea is that as we go under binders Δ in a term, we go under binders Δ^1_0 in the corresponding structural proof. We may now define the *respect* operator by mutual recursion on the syntax of types and terms

$$\frac{\Gamma; \Delta \vdash T[\Delta]\ \textbf{set}}{\Gamma; \Delta^1_0 \vdash T[\![\Delta]\!] : T[\Delta_0] = T[\Delta_1]}$$

$$\frac{\Gamma; \Delta \vdash t[\Delta] : T[\Delta]}{\Gamma; \Delta^1_0 \vdash t[\![\Delta]\!] : (t[\Delta_0]{:}T[\Delta_0]) = (t[\Delta_1]{:}T[\Delta_1])}$$

For types, with $B \in \{\Pi, \Sigma, \mathcal{W}\}$ and $C \in \{\mathsf{O}, 1, 2\}$

$$\begin{array}{rl} (Bx{:}S.\ T[x])[\![\Delta]\!] & \mapsto B^{=}\ S[\![\Delta]\!]\ (\lambda\{x{:}S\}^1_0.\ T[\![\Delta; x{:}S]\!]) \\ C & \mapsto C^{=} \\ (b\ (\mathsf{Case}\ T; F))[\![\Delta]\!] & \mapsto b[\![\Delta]\!]\ (\mathsf{Case}^{=}\ T[\![\Delta]\!]; F[\![\Delta]\!]) \end{array}$$

For terms, we give the basic picture,

$$\begin{array}{rl} x[\![\Delta]\!] & \mapsto x_{01} \ \text{ if } x \in \Delta \\ (\lambda x{:}S.\ t[x])[\![\Delta]\!] & \mapsto \lambda^{=}\ S[\![\Delta]\!]\ (\lambda\{x{:}S\}^1_0.\ t[\![\Delta; x{:}S]\!]) \\ (f\ s)[\![\Delta]\!] & \mapsto f[\![\Delta]\!]^{=}\ s[\![\Delta]\!] \\ & \vdots \end{array}$$

We may now give the constructions for reflexivity just as

$$\overline{T} \mapsto T[\![\mathcal{E}]\!] \qquad \bar{t} \mapsto t[\![\mathcal{E}]\!]$$

The respect operator computes structurally through the syntax of terms, but it gets stuck at free variables—those from Γ rather than Δ—so we must add the respect operator to the syntax of the theory in which we are performing the construction. Of course, the computation continues whenever the free variables are instantiated. For this to make sense, the proofs computed by the respect operator must be invariant with the definitional equality. It suffices to ensure that the β-rules commute with respect—we have

$$\begin{array}{l} (\lambda^{=}\ S[\![\Delta]\!]\ (\lambda\{x{:}S\}^1_0.\ t[\![\Delta; x : S]\!]))^{=}\ s[\![\Delta]\!] \\ \quad \equiv t[\![\Delta; x{:}S]\!][s[\Delta_0], s[\Delta_1], s[\![\Delta]\!]] \\ \quad \equiv t[s][\![\Delta]\!] \end{array}$$

and similarly for the others.

3.3. Symmetry, Transitivity and Coherence

Symmetry is defined by recursion on type and proceeds componentwise. Only the compound types require any thought, but symmetry of value equations over the domains allows us to exploit the proof that the ranges coincide at equal values:

$$(S, T)^{\smile} \mapsto \left(S^{\smile}, \lambda\{x{:}S\}^0_1.\ \left(T\ \overrightarrow{x^{\smile}_{10}}\right)^{\smile}\right)$$

The same construction works for equations on functions:

$$(S, f)^{\smile} \mapsto \left(S^{\smile}, \lambda\{x{:}S\}^0_1.\ \left(f\ \overrightarrow{x^{\smile}_{10}}\right)^{\smile}\right)$$

We must now establish the coherence of these coercions, by mutual recursion on types.

$$\begin{array}{rcl} f_0[\![(S,T)]\!]^{\Pi x_1{:}S_1.\ T_1[x_1]}_{\Pi x_0{:}S_0.\ T_0[x_0]} & & \\ \quad : & f_0 = \lambda x_1{:}S_1.\ f_0\ (\langle S]x_1)\ \left[T\ \overrightarrow{(\!|S]\!]x_1}\right\rangle & \\ \quad \mapsto & \lambda^{=}\ S\ (\lambda\{x{:}S\}^1_0.\ \overline{f_0}^{=}\ (x_{01} \circ (([\!|S]\!]x_1)^{\smile}) \circ & \\ & \qquad f_0\ (\langle S]x_1)\ \left[\!\left[T\ \overrightarrow{(\!|S]\!]x_1}\right]\!\right\rangle) & \end{array}$$

Here, $x_{01} : x_0 = x_1$ and $(\langle\!|S|\!\rangle x_1)^\smile : x_1 = \langle S]x_1$, hence we establish that $f_0\, x_0 = f_0\, (\langle S]x_1)$. The coherence of the codomain coercion does the rest.

For pairs, coherence is componentwise:

$$\begin{array}{rcl}(s_0, t_0)[\![(S,T)]\!\rangle^{\Sigma x_1 S_1.\, T_1[x_1]}_{\Sigma x_0 S_0.\, T_0[x_0]} & & \\ \quad : \quad (s_0, t_0) = \left(s_0[S\rangle\, , t_0 \left[T\ \overrightarrow{s_0[\![S]\!\rangle}\right\rangle\right) & & \\ \quad \mapsto \quad \left(s_0[\![S]\!\rangle\, ;^{=} t_0 \left[\!\left[T\ \overrightarrow{s_0[\![S]\!\rangle}\right|\right\rangle\right) & & \end{array}$$

For trees, we must supply an inductive proof. Here we write the corresponding program:

$$\begin{array}{l} w_0[\![(S,T)]\!\rangle^{\mathcal{W} x_1 S_1.\, T_1[x_1]}_{\mathcal{W} x_0 S_0.\, T_0[x_0]} \mapsto \mathsf{coh}\ w_0 \text{ where} \\ \quad \mathsf{coh} : \Pi w_0 : \mathcal{W} x_0 : S_0.\, T_0[x_0].\ \ w_0 = w_0[(S,T)\rangle \\ \quad \mathsf{coh}\ (\mathsf{node}\ s_0\ f_0) \mapsto \\ \qquad \mathsf{node}^{=}\ s_{01}\ (\lambda^{=}(T_{01}\ \vec{s_{01}})\ (\lambda\{t : T[s]\}^1_0. \\ \qquad \overline{f_0}^{\,=}\ (t_{01} \circ (\langle\!|T\ \vec{s_{01}}[\![t_1)^\smile) \circ \mathsf{coh}\ (f_0\ (\langle T\ \vec{s_{01}}]t_1)))) \\ \qquad \text{where}\ \ s_1 \mapsto s_0[S\rangle : S_1 \\ \qquad\qquad\ \ s_{01} \mapsto s_0[\![S]\!\rangle : \|s : S\|^1_0 \end{array}$$

The coherence proof for each canonical type may invoke transitivity of equality for its components. Transitivity for type and value equalities is mutually defined with coherence by recursion on type.

For compound types, we have

$$(S_{01}, T_{01}) \circ (S_{12}, T_{12}) \mapsto \begin{pmatrix} S_{01} \circ S_{12}, \\ \lambda\{x : S\}^2_0.\ T_{01}\ \overrightarrow{x_0[\![S_{01}]\!\rangle} \circ \\ \quad T_{12}\ \overrightarrow{(x_0[\![S_{01}]\!\rangle)^\smile} \circ x_{02} \end{pmatrix}$$

Note the way we get from $T_0[x_0]$ to $T_2[x_2]$ by building a stepping stone, $x_0[S_{01}\rangle : S_1$, equal to x_0 and hence x_2. We could equally have chosen $\langle S_{12}]x_2$. The same thing happens in the case of equality for functions—it is to enable this construction that an equation between functions carries the equation between their domains.

$$(S_{01}, f_{01}) \circ (S_{12}, f_{12}) \mapsto \begin{pmatrix} S_{01} \circ S_{12}, \\ \lambda\{x : S\}^2_0.\ f_{01}\ \overrightarrow{x_0[\![S_{01}]\!\rangle} \circ \\ \quad f_{12}\ \overrightarrow{(x_0[\![S_{01}]\!\rangle)^\smile} \circ x_{02} \end{pmatrix}$$

In all other cases, transitivity proceeds componentwise.

4. Encoding datatypes

In Extensional Type Theory with $\mathcal{W}$-types we can encode a vast variety of datatypes, inductive types like natural numbers or lists, inductive families like vectors but also coinductive types or families [1]. We can implement datatypes with the associated constructors and eliminators in OTT - however, in the theory as presented so far this encoding is not completely faithful: not all expected definitional equalities hold.

The natural numbers provide an illustrative example. We may define

$$\begin{array}{l} \mathsf{Nat} \mapsto \mathcal{W} b : 2.\ b\ (\mathsf{Case}\ \mathsf{O}; 1) \\ \mathsf{zero} \mapsto \mathsf{node}\ \mathsf{tt}\ (\lambda z : \mathsf{O}.\ z\ (\text{Œ}\ \mathsf{Nat})) \\ \mathsf{suc} \ \mapsto \lambda n : \mathsf{Nat}.\ \mathsf{node}\ \mathsf{ff}\ (\lambda u : 1.\ n) \end{array}$$

Now let us show how to derive instances of the induction schema. For each $n : \mathsf{Nat} \vdash P[n]$ **set**, we need to define

$$\frac{\begin{array}{cc} & z : P[\mathsf{zero}] \\ m : \mathsf{Nat} & s : \Pi n : \mathsf{Nat}.\ P[n] \to P[\mathsf{suc}\ n] \end{array}}{m\ (\mathsf{NatE}\ n.\ P[n] \mid z; s) : P[m]}$$

via primitive recursion on $\mathcal{W}$-types. We give the informal recursive definition:

$$\begin{array}{l} \mathsf{node}\ \mathsf{tt}\ f\ (\mathsf{NatE}\ n.\ P[n] \mid z; s) \mapsto \\ \quad z[P[\![n]\!][\mathsf{zero}, \mathsf{node}\ \mathsf{tt}\ f, prf]\rangle \quad \text{where} \\ \qquad prf \mapsto \lambda^{=}\{z : \mathsf{O}\}^1_0.\ z_0\ (\text{Œ}\ z_0\ (\text{Œ}\ \mathsf{Nat}) = f\ z_1) \\ \mathsf{node}\ \mathsf{ff}\ f\ (\mathsf{NatE}\ n.\ P[n] \mid z; s) \mapsto \\ \quad s\ (f\ ())\ (f\ ()\ (\mathsf{NatE}\ n.\ P[n] \mid z; s)) \end{array}$$

Observe that $z : P[\mathsf{node}\,\mathsf{tt}\,(\lambda z : \mathsf{O}.\, z\,(\text{Œ}\,\mathsf{Nat}))]$ where we require a proof of $P[\mathsf{node}\ \mathsf{tt}\ f]$: there is more than one representation of zero, but now we can at least *prove* that they are all equal: the coercion crucially exploits the extensional equality of functions on the empty domain.

The same technique lifts to tree-like datatypes in general: we use functions with a finite domain to code tuples of subtrees: every such function is provably equal to a case analysis. In ITT, we cannot establish this identity and the induction principles for $\mathcal{W}$-type encodings are not derivable.

However, we have lost something. Only in the case that the domain is 1 is the definitional equality strong enough to identify the arbitrary function in the general node with the particular implementation chosen by the defined constructor: $\lambda u : 1.\ f\ () \equiv f$ by the η-rules for Π and 1.

In general, we can expect a coercion to fix up the type in each case. The computation rules we usually give for datatypes in ITT do not hold definitionally here, although they are provable by coherence of coercion. This is just because the $\mathcal{W}$-type encoding replaces tuples whose definitional equality is componentwise, by functions whose definitional equality is only schematic.

One approach to solving this problem is to consider extending the definitional equality for functions, comparing them at a finite *covering* of their domain, not just an arbitrary x: functions from O would be equal automatically, functions from 2 would be equal if they coincided at tt and at ff, and so on. Another approach is to seek an alternative set of type-forming primitives which retain the first-order character of first-order data.

5. Full OTT

Although the observational type theory we have presented here is sufficient to capture the *provable* equations of Extensional Type Theory, its *definitional* equality gives us less than we might hope. In this section, we consider the problems and propose extensions to our basic system which might improve the situation.

One clearly desirable extension is *definitional proof irrelevance*, internalising the equality of equality proofs. We would add

$$\frac{\Gamma \vdash Q_0 : \|S\|_0^1 \quad Q_1 : \|S\|_0^1}{\Gamma \vdash Q_0 \equiv Q_1 : \|S\|_0^1}$$

and the corresponding rule for terms, dropping the Irr and irr operators. Operationally this is harmless, as equality is decided in a type-directed manner and coercion does not inspect equality proofs—they actually *are* irrelevant! Indeed, the fact that we were able to construct an observational type theory without definitional proof irrelevance came as quite a surprise to us: the first author's model construction in [2] relies on proof irrelevance, in the absence of a heterogeneous equality.

A more serious concern is that our system has actually lost some of the definitional equations of ITT. In the latter setting, where equality is effectively a datatype with reflexivity as its sole constructor, we have

$$\overline{b}\,(\mathsf{eqElim}\; x, y, z.\; x\;(\mathsf{Case}\; T; F) \mid p) \equiv p\; b$$

for an arbitrary non-canonical b. Our coercion operator computes only for compatible canonical types. Here,

$$p[b]\left[\overline{b\;(\mathsf{Case}\; T; F)}\right\rangle \not\equiv p[b]$$

although the proposition

$$p[b]\left[\overline{b\;(\mathsf{Case}\; T; F)}\right\rangle = p[b]$$

holds by coherence of coercion. The second author's construction of dependent pattern matching [9] relies heavily on the manipulation of equality proofs on pattern variables, inevitably non-canonical: the pattern matching equations hold definitionally, but only because the equational machinery computes away when the proofs are by reflexivity. One tempting approach is just to add

$$\frac{\Gamma \vdash x : X}{\Gamma \vdash x\left[\overline{X}\right\rangle \equiv x : X} \qquad \frac{\Gamma \vdash x : X}{\Gamma \vdash x\left[\!\left[\overline{X}\right]\!\right\rangle \equiv \overline{x} : \|x : X\|_0^1}$$

Care must be taken to avoid critical pairs in the presence of the type-directed coercion rules. Our η-rules avoid this problem for Π- and Σ-types, but for $\mathcal{W}$-types, we risk expanding an operation which is definitionally the identity to one which is only provably the identity. If we replace our current reduction of coercion for $\mathcal{W}$ to the rec operator with its equivalent defined by direct recursion, we escape this problem: a recursive call to a reflexive coercion is also a reflexive coercion.

If we seek to combine these two extensions, we need to take into account the fact that a direct appeal to $\overline{X}$ is not the only syntactic form which a proof of $X = X$ can take. We would need a rule like

$$\frac{\Gamma \vdash S \equiv T \quad \Gamma \vdash Q : S = T \quad \Gamma \vdash s : S}{\Gamma \vdash s[Q\rangle_S^T \equiv s : T}$$

The above coherence rule would then become derivable by proof irrelevance. This proposed coercion rule has significant operational implications: to decide whether it 'fires' or not, the machine must decide the definitional equality of S_0 and S_1. That is, evaluation and equality become *mutually recursive*.

Of course, it is not hard to write a program which appears to implement such a mutual recursion—indeed we have done so—but it is not easy to explain why it gives rise to a complete decision procedure, or indeed why evaluation terminates at all. It certainly falls outside the scope of the existing proof methods, hence we make no bold claims on the subject.

Finally, in a programming language like Epigram [10] we should like to be able to introduce datatypes directly, whether or not they have suitable encodings via $\mathcal{W}$-types in principle. Functional codings of first-order data are hard to compile efficiently: in order to preserve sharing, one must memoize the functions, effectively recovering their first-order representation by trickery.

Epigram supports the generous notion of *inductive family* proposed by Dybjer in [4], where constructors may target *specific* indices. A standard example is the family of finite sets:

$$\frac{n : \mathsf{Nat}}{\mathsf{Fin}\; n\; \mathbf{set}} \qquad \frac{n : \mathsf{Nat}}{\mathsf{fz}\; n : \mathsf{Fin}\;(\mathsf{suc}\; n)} \qquad \frac{n : \mathsf{Nat} \quad i : \mathsf{Fin}\; n}{\mathsf{fs}\; n\; i : \mathsf{Fin}\;(\mathsf{suc}\; n)}$$

This definition fits badly with observational equality: given a proof $q : (\mathsf{suc}\; n) = m$, we should expect

$$\mathsf{fz}\; n\; [\mathsf{Fin}^{=}\; q\rangle$$

to reduce to a canonically constructed element of $\mathsf{Fin}\, m$, but there is no such thing for a general m.

One solution is to enforce the standard trick for introducing such definitions in systems which require the constructors of inductive types to target all indices uniformly: replace constraint-by-instantiation with constraint-by-equation, in the style of Henry Ford—'any k you like as

long as it's $\mathsf{suc}\, n$'. The definition becomes

$$\frac{k : \mathsf{Nat}}{\mathsf{Fin}\, k\ \mathbf{set}} \qquad \frac{n : \mathsf{Nat} \quad q : \mathsf{suc}\, n = k}{\mathsf{fz}\, n\, q : \mathsf{Fin}\, k}$$

$$\frac{n : \mathsf{Nat} \quad i : \mathsf{Fin}\, n \quad q : \mathsf{suc}\, n = k}{\mathsf{fs}\, n\, i : \mathsf{Fin}\, k}$$

Of course, the old constructors are still definable, giving q by reflexivity, but coercion is now able to compute under the constructors by appeal to transitivity:

$$\mathsf{fz}\, n\, q\, [\mathsf{Fin}^{=}\, q' \rangle \equiv \mathsf{fz}\, n\, (q \circ q')$$
$$\mathsf{fs}\, n\, i\, q\, [\mathsf{Fin}^{=}\, q' \rangle \equiv \mathsf{fs}\, n\, i\, (q \circ q')$$

6. Conclusions and further work

We have presented and justified proof-relevant OTT, a theory whose propositional equality can simulate equality in Extensional Type Theory, and which at the same time has the desirable property of canonicity. We go further and add both definitional proof-irrelevance for equality types and reductions for reflexivity coercions leading to OTT. We have implemented OTT as part of Epigram and are hopeful that we can justify this theory, and in particular establish the important metatheoretic properties of normalisation and decidability.

Moreover, we are confident that we can eliminate proofs at runtime and hence generate efficent code from OTT definitions. We can also implement inductive families as primitive in OTT and plan to add coinductive families such that the corresponding coinduction principles are provable. Another important extension is the addition of quotient types which give us a notion of abstract datatypes relevant not only for the faithful formalisation of mathematical theories but also in software engineering. We are looking forward to many exciting applications of OTT ranging from a faithful presentation of category theory to the implementation of well-behaved concurrent programs within Type Theory.

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005.

[2] T. Altenkirch. Extensional equality in intensional type theory. In *LICS 99*, 1999.

[3] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[4] P. Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[5] M. Hofmann. Conservativity of equality reflection over intensional type theory. In *TYPES 95*, pages 153–164, 1995.

[6] M. Hofmann and T. Streicher. A groupoid model refutes uniqueness of identity proofs. In *LICS 94*, pages 208–212, 1994.

[7] G. Huet and A. Sai. *Constructive Category Theory*. MIT Press, 1998.

[8] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[9] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[10] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[11] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[12] N. Oury. Extensionality in the calculus of constructions. In *TPHOL 05*, pages 278–293, 2005.