Observational Equality, Now!

Thorsten Altenkirch Conor McBride Wouter Swierstra

University of Nottingham, UK

{txa,ctm,wss}@cs.nott.ac.uk

Abstract

This paper has something new and positive to say about propositional equality in programming and proof systems based on the Curry-Howard correspondence between propositions and types. We have found a way to present a propositional equality type

- which is *substitutive*, allowing us to reason by replacing equal for equal in propositions;
- which reflects the *observable* behaviour of values rather than their construction: in particular, we have *extensionality*—functions are equal if they take equal inputs to equal outputs;
- which retains *strong* normalisation, *decidable* typechecking and *canonicity*—the property that closed normal forms inhabiting datatypes have canonical constructors;
- which allows inductive data structures to be expressed in terms of a standard characterisation of *well-founded trees*;
- which is presented *syntactically*—you can implement it directly, and we are doing so—this approach stands at the core of Epigram 2;
- which you can play with now: we have simulated our system by a shallow embedding in Agda 2, shipping as part of the standard

examples package for that system [21].

Until now, it has always been necessary to sacrifice some of these aspects. The closest attempt in the literature is Altenkirch's construction of a setoid-model for a system with canonicity and extensionality on top of an intensional type theory with proof-irrelevant propositions [4]. Our new proposal simplifies Altenkirch's construction by adopting McBride's *heterogeneous* approach to equality [19].

Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Lambda calculus and related systems; D.3.1 [*Formal Definitions and Theory*]: Semantics

General Terms Languages,Theory

Keywords Type Theory, Equality

1. Introduction

Equations are ubiquitous in mathematical reasoning, and reasoning about programs is no exception. Moreover, notions of equality play a crucial internal rôle in the programs of any language where

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLPV'07, October 5, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-677-6/07/0010...$5.00

datatypes are indexed in some way. Case analysis relies on solving the equation between the type of the scrutinee and the possible types returned by each constructor, specialising the return type of each branch accordingly. For a standard example, consider length-indexed lists. We need to know that a length 4 list cannot be be made by the 'nil' constructor because 4 does not equal 0, but can be made by 'cons', given a head element and a tail of length 3.

Further, computation within types is now commonplace: appending two length 2 lists yields a length $2 + 2$ list, which should somehow be regarded also as a length 4 list. Further still, we sometimes need to exploit algebraic properties of type-level expressions beyond their symbolic evaluation: we may wish to use a length $x + y$ list where length $y + x$ is expected.

Of course, dependent type theories have always had this issue, but it is becoming increasingly prevalent in programming languages, whether they have 'full-blown' value-dependency, like Cayenne [5], Epigram [20, 11] and Agda [21], or a separate static language of indices, as in DML [29], Ωmega [25] and Haskell with Generalized Algebraic Datatypes [24].

This paper concerns the strength of equational reasoning available for both behind-the-scenes constraint solving and the explicit manipulation of type-level expressions in Martin-Löf type theories, hence in proof systems and programming languages based on the "propositions-as-types" principle. These tend to be divided into two camps:

extensional type theories (ETTs, as implemented in NuPRL [7]) identify the *definitional* equality (as used in typechecking) with the *propositional* equality (expressed as a type and used for reasoning), resulting in powerful systems with undecidable typechecking;

intensional type theories (ITTs, as implemented in Coq, Agda and Epigram) separate these notions, keeping the definitional equality (and hence typechecking) *decidable*, but at some cost to the power of the propositional equality—until now.

In this paper, we examine an approach which delivers a 'no compromise' candidate: Observational Type Theory (OTT), with all the good properties of both. OTT is an intensional type theory with all the key computational properties we expect of such systems, but a notion of propositional equality that identifies values *up to observation*, rather than by construction. It is just as powerful for reasoning as the extensional theories.

This paper also records an attempt to code the phenomena of dependently typed programming in a simple and closed core type theory. In particular, recursive data are introduced uniformly via the so-called 'W-types', inhabited by higher-order encodings of *well-founded* trees. This encoding has always been feasible in ETTs, but it has never worked in an ITT—until now. The OTT approach has allowed us to make the encoding, and to discover its price.

Our adventure has a strong practical component. We show how to *construct* an OTT by a shallow embedding in an existing intensional type theory, defined within the Agda 2 framework. By doing so, we expose the mechanics of the approach and gain evidence for key metatheoretical properties. In particular, we obtain at least a sketch proof for strong normalisation for expressions in general, and *canonicity* for closed expressions—the latter always compute to canonical values, as programmers might naturally hope. However, you can also download the OTT construction (now a standard Agda 2 example) and try out observational equality, now!

1.1 The Equality Dilemma

In traditional type theories, equational propositions are typically

represented as instances of a family of types:

$$\frac{\vdash A : \mathsf{Set} \quad a, b : A}{\vdash a =_A b : \mathsf{Prop}}$$

where Set is the sort of datatypes and Prop is the sort of propositional types—sometimes these coincide. Most people agree that equality should be reflexive, and provide the introduction rule:

$$\frac{\vdash a : A}{\vdash \mathsf{refl}_A\, a : a =_A a}$$

As these theories have nontrivial computations within types, they come with a notion of *definitional* equality, presented here as a typed equality judgment ($\equiv$). Types are identified up to definitional equality, as expressed by the *conversion* rule:

$$\frac{\vdash s : S \quad \vdash S \equiv T}{\vdash s : T}$$

The definitional equality is a congruence, so the full effect of the refl constructor is to embed $\equiv$ into $=$:

$$\text{if} \quad \vdash a \equiv b : A \quad \text{then} \quad \vdash a =_A a \;\equiv\; a =_A b : \mathsf{Prop}$$
$$\text{and therefore} \quad \vdash \mathsf{refl}_A\, a : a =_A b$$

The two camps diverge when it comes to the way that propositional equations are *used*. Extensional type theories adopt the *equality reflection* rule, allowing provable equations to impact directly and silently on typechecking:

$$\frac{\vdash q : a =_A b}{\vdash a \equiv b : A}$$

By contrast, intensional type theories require an explicit operation, coresponding to the 'Leibniz rule' for equality, transporting values between types which are merely provably—but perhaps not definitionally—equal

$$\frac{\vdash\ q : a =_A b \quad \vdash\ T : A \to \mathsf{Set} \quad \vdash\ t : T\ a}{\mathrm{subst}_{A;a;b}\ q\ T\ t : T\ b}$$

with the associated computational rule

$$\vdash\ \mathrm{subst}_{A;a;a}\ (\mathsf{refl}_A\ a)\ T\ t \equiv t : T\ a$$

making subst disappear when the proof $\mathsf{refl}_A\ a$ guarantees that initial and final types coincide definitionally. This approach preserves the desirable *canonicity* property of ITTs—*closed* expressions have canonical values. In the empty context, the only possible proof of an equation is by refl, so all top-level substs are guaranteed to vanish. Compilers for programming languages based on strongly-normalising ITTs can thus erase equation proofs and appeals to subst from run-time code.

Clearly, ITT results in terms which are more bureaucratic: there is no need for subst in ETT because the proof of $a =_A b$ gives us $t : T\ b$ by the conversion rule and equality reflection. However, a high price must be paid for this benefit.

Firstly, the equality reflection rule is clearly not *syntax-directed*: if we want a machine to recover the typing derivations for well-typed terms, it must be able to invent the proofs of equations demanded by appeals to equality reflection. In an ITT, the proofs are always explicit.

Secondly, ETTs identify all types in the presence of false hypotheses, making it unsafe to compute under binders. In an ITT, the machine can check *terms* and execute them safely if they are well-typed, regardless of the context. In an ETT, the checkable objects are the *derivations*, recording not only the appeals to typing rules but also every equational step. ETT terms may be smaller, but only because the evidence which justifies them is somewhere else.

Gonthier and Werner's proof of the Four-Colour Theorem is a relatively small ITT term, but its ETT derivation would be unfeasibly large because it relies very heavily on computation.

A big attraction of ETTs is that more desirable equations have proofs. In particular, the *extensionality* law holds for functions, just by congruence, equality reflection and the η-law:

$$\text{if } \forall x.\, f\, x = g\, x \text{ then } (\lambda x.\, f\, x) = (\lambda x.\, g\, x) \text{ so } f = g$$

This just cannot happen in ITTs following the refl and subst pattern. To see this, consider the functions $\lambda x.\ 0{+}x$ and $\lambda x.\ x{+}0$. One may certainly give a closed inductive proof

$$\text{plus0} : \forall x.\ 0{+}x = x{+}0$$

but the two functions are not definitionally equal, hence refl cannot show that $\lambda x.\ 0{+}x = \lambda x.\ x{+}0$ and correspondingly, there can be no closed proof of this mathematically intuitive fact. By the same token, it is difficult to reason equationally about higher-order functional programs in ITT-based systems, where laws like

$$\text{map id} = \text{id} \qquad \text{map}\,(f \circ g) = \text{map}\, f \circ \text{map}\, g$$

cannot be proven.

The challenge is to combine the computational power (and decidable typechecking) of intensional theories with the reasoning power of extensional theories. One cannot just add an axiom

$$\frac{q : \forall x.\, f\, x = g\, x}{\text{ext}\, q : f = g}$$

without losing canonicity:

$$\text{subst}\ (\text{ext plus0})\ (\lambda\_.\ \mathsf{Nat})\ 0 : \mathsf{Nat}$$

is a closed expression which cannot compute a number, because the subst has got stuck! The irony of this example is that $\lambda\_.\ \mathsf{Nat}$ makes no *use* of its argument. In fact, $\mathrm{subst}\ (\mathrm{ext}\ \mathrm{plus0})\ (\lambda\_.\ \mathsf{Nat})$ is a function from Nat to Nat which is provably equal (by ext) to id. If we only looked at the source and target, we could quite easily see how to perform the transportation! It is in this irony that we find some hope of resolution.

1.2 The Observational Approach

Our approach is designed to ensure the effectiveness of the maps induced by equations, transporting values between provably equal sets. We start by explicitly introducing a notion of set equality. This will not be defined inductively, but will rather *compute* what you need to know to transport canonical values from one set to another. Equations between sets induce *coercions*.

$$\frac{S, T : \mathsf{Set}}{S = T : \mathsf{Prop}} \qquad \frac{Q : S = T \quad s : S}{s\,[Q{:}S{=}T\rangle : T}$$

Of course, we still need to talk about equality of *values*. We allow the formation of *heterogeneous* equations, relating values in arbitrary sets. This is again a computed notion, telling you how to ensure that equal values support equal *observations*.

$$\frac{s : S \quad t : T}{(s : S) = (t : T) : \mathsf{Prop}}$$

Heterogeneous equality may seem a little disturbing: when would it make sense to say that $(0 : \mathsf{Nat}) = (\mathrm{id} : \mathsf{Nat} \to \mathsf{Nat})$? On the other hand, how would you state the fact that some function f preserves equality of arguments if its type is dependent—some

$\Pi a : A.\ B[a]$?[1] With heterogeneous equality, we can write

$$\forall x, y.\ (x : A) = (y : A) \rightarrow (f\ x : B[x]) = (f\ y : B[y])$$

but with the traditional homogeneous equality, it is rather clumsy to cope with the way the results have definitionally distinct types. Using homogeneous equality, with extra dependency and heavy use of subst, we would have written:

$$\forall x, y.\ \forall q : x =_A y.\ \mathrm{subst}\ q\ (\lambda a.\ B[a])\ (f\ x) =_{B[y]} f\ y$$

Our approach scales up much more conveniently. The crucial point is that heterogeneous equations are only *useful* if we know why their types are equal.

Heterogeneity also allows us to formulate the *coherence* operator, another OTT primitive, whose definition requires us to ensure that all of our equality-induced coercions are in some sense just the identity:

$$\frac{Q : S = T \quad s : S}{\{s \parallel Q{:}S{=}T\} : (s : S) = (s\ [Q{:}S{=}T\rangle : T)}$$

Note that, by construction, the two sides of this equation have equal types. We shall need coherence in order to explain the action of coercion on types which involve binding.

The need to implement coercion between sets will drive the definition of equality on sets. We shall then be in a position to consider what equality on values must be.

1.3 A Brief History of Equality

There has been quite a lot of work addressing the problem that intensional type theories have weak reasoning properties, despite their good computational behaviour. The most significant contribution is that of Martin Hofmann, who showed that equality re-

flection, in the style of ETT, is conservative over ITT extended by axioms for 'extensional concepts' [14, 13]. The idea of extending ITT consistently with extensionality axioms goes back at least to Turner [27], but the problem of losing canonicity was also clear at that time.

Hofmann also pioneered the construction of *setoid* models, encoding a type theory where extensionality holds in an extensional theory. A setoid is a set equipped with its own notion of equivalence; functions between setoids form a setoid with functions equivalent when they take equivalent domain values to equivalent range value. In order to reason by substitution, we need to be sure that the predicates we can form over a setoid respect its equivalence. Hofmann's machinery ensures that this is always the case.

However, Hofmann's attempt to build a setoid model of a system with *large elimination*—computation of types from values—ran into technical difficulties. Altenkirch's setoid model [4] gets around these problems: his meta-logic has a more liberal definitional equality, identifying all proofs of the same proposition, allowing the extension of Hofmann's work.

Meanwhile, McBride's contribution of *heterogeneous* equality [19] reduced the bureaucracy of working with equations between values in dependent types, motivated by need to express the unification problems inherent to Coquand's dependent pattern

[1] We write Π when we range over a Set and $\forall$ when we range over a Prop.

matching [8]. The idea of using heterogeneous equality goes back at least to Huet and Saïbi's formalisation of category theory in type theory [16], itself an epic struggle. However, McBride's contribution was to provide a *homogeneous* substitution principle, equivalent to extending the usual intensional propositional equality with the so-called 'K-rule', expressing the uniqueness of identity proofs.

The latter is a trivial consequence of pattern matching, but was shown by Hofmann and Streicher to be unprovable from the usual induction principle for equality [15].

Heterogeneous equality was employed more recently by Oury to simplify Hofmann's proof, showing that an extensional variant of the Calculus of Constructions is conservative over the intensional version with extensional axioms [22]. Our present work is a heterogeneous variant of the theory for which Altenkirch constructed a setoid model. The consequent simplification allows us to give a direct syntactic presentation of OTT, calculating all the necessary machinery from the structure of the types involved.

In an exciting parallel development, the need to manage equational constraints for functional programming with Generalized Algebraic Datatypes in Haskell has resulted in an extension of System F with a notion of equality, inducing coercions which compute through the structure of values [26]. Our system has something of the same flavour, but type dependency involves more interaction between its layers: we have fewer, more general components, but we need a more delicate touch.

2. A Simple Core Type Theory

We shall start from a very basic type theory, then show how to equip it with a notion of observational equality. We should emphasize that we intend this theory to act as a very simple *core* language, with few syntactic conveniences and the minimal feature set required to investigate and illustrate the observational approach. In particular, our basic theory has no hierarchy of universes, although it still has type-level computation. Let us first establish the core syntax of sets and terms.

$$
\begin{array}{lll}
\text{set} & \mathbf{S} ::= & \mathbf{G} \mid \mathbf{B}\,\mathbf{X}{:}\mathbf{S}.\,\mathbf{S} \mid \mathsf{If}\ \mathbf{T}\ \mathsf{Then}\ \mathbf{S}\ \mathsf{Else}\ \mathbf{S} \\
\text{ground} & \mathbf{G} ::= & 0 \mid 1 \mid 2 \\
\text{binder} & \mathbf{B} ::= & \Pi \mid \Sigma \mid \mathsf{W}
\end{array}
$$

$$\text{term}\quad \mathbf{T} ::= \langle\rangle \mid \mathsf{tt} \mid \mathsf{ff} \mid \lambda\mathbf{X}{:}\mathbf{S}.\,\mathbf{T} \mid \langle \mathbf{T},\mathbf{T}\rangle_{\Sigma \mathbf{X}:\mathbf{S}.\,\mathbf{S}} \mid \mathbf{T} \lhd_{\mathsf{W}\mathbf{x}:\mathbf{S}.\,\mathbf{S}} \mathbf{T} \mid \mathbf{T}!\mathbf{S} \mid \mathsf{if}\ \mathbf{T}/\mathbf{X}.\mathbf{S}\ \mathsf{then}\ \mathbf{T}\ \mathsf{else}\ \mathbf{T} \mid \mathbf{T}\,\mathbf{T} \mid \mathsf{fst}\ \mathbf{T} \mid \mathsf{snd}\ \mathbf{T} \mid \mathsf{rec}\ \mathbf{T}/\mathbf{X}.\mathbf{S}\ \mathsf{with}\ \mathbf{T}$$

We have a fixed repertoire of set-formers: ground sets with 0 elements, 1 element $\langle\rangle$ and 2 elements $\{\mathsf{tt}, \mathsf{ff}\}$, and set binders Π for dependent functions (λ-terms), Σ for dependent pairs, and W for well-founded trees whose nodes $s \lhd_{\mathsf{W}x S.T} f$ comprise a choice s of node shape from S and a 'child-function' f from recursive positions $T[s]$ to subtrees. Informally, we shall drop type annotations from value constructors where they may readily be inferred.[2] We shall adopt the standard abbreviations $\to$ and $\times$ for non-dependent Π- and Σ-types, respectively. We also write $\lambda x\, y.\, t$ for $\lambda x.\, \lambda y.\, t$ and $\langle r,s,t\rangle$ for $\langle r,\langle s,t\rangle\rangle$.

Computation happens via the corresponding eliminators, with $-!T$ making a 'magic' element of T from a mythical element of 0, dependent if-then-else for 2, application for Π, projections for Σ, and dependent recursion for W. We have been careful to consider only functions between sets: dependent eliminators like if and rec abstract their target sets over a bound variable, leaving no need to internalise a type of set-valued functions. Informally, we shall drop this range annotation when eliminating a *variable*,

[2] More formally, we can introduce a bidirectional approach to typechecking which ensures that we can always drop these annotations, but that is a story for another time.

in which case the abstraction is readily determined from the type required. Despite the absence of set-valued functions, we shall still see computation within types via 'large' conditionals on elements of 2.

The operational semantics of computation is given by structural

closure of the following schemes:

$$
\begin{aligned}
(\lambda x.\, t)\, v &\rightsquigarrow t[v] \\
\mathsf{fst}\, \langle s,t \rangle &\rightsquigarrow s \\
\mathsf{snd}\, \langle s,t \rangle &\rightsquigarrow t \\
\mathsf{if}\ \mathsf{tt}/x.P\ \mathsf{then}\ t\ \mathsf{else}\ f &\rightsquigarrow t \\
\mathsf{if}\ \mathsf{ff}/x.P\ \mathsf{then}\ t\ \mathsf{else}\ f &\rightsquigarrow f \\
\mathsf{rec}\ s \triangleleft f/w.P\ \mathsf{with}\ p &\rightsquigarrow p\ s\ f\ (\lambda t.\ \mathsf{rec}\ f\ t/w.P\ \mathsf{with}\ p)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{If}\ \mathsf{tt}\ \mathsf{Then}\ T\ \mathsf{Else}\ F &\rightsquigarrow T \\
\mathsf{If}\ \mathsf{ff}\ \mathsf{Then}\ T\ \mathsf{Else}\ F &\rightsquigarrow F
\end{aligned}
$$

This system is just a fragment of many standard-issue type theories (Martin-Löf's Intensional Type Theory, the Calculus of Inductive Constructions, Luo's UTT, etc) all of which enjoy metatheoretical properties such as confluence and strong normalisation. We are therefore entitled to presume the existence of an operator $-^{\Downarrow}$, defined on well-typed terms, taking them to *values* (in this case, just the normal forms):

$$
\begin{array}{lll}
\text{set} & \bar{\mathbf{S}} ::= & \mathbf{G} \mid \mathbf{B}\,\mathbf{X}{:}\bar{\mathbf{S}}.\,\bar{\mathbf{S}} \mid \mathsf{If}\ \bar{\mathbf{N}}\ \mathsf{Then}\ \bar{\mathbf{S}}\ \mathsf{Else}\ \bar{\mathbf{S}} \\
\text{term} & \bar{\mathbf{T}} ::= & \bar{\mathbf{N}} \mid \langle\rangle \mid \mathsf{tt} \mid \mathsf{ff} \mid \lambda\mathbf{X}.\,\bar{\mathbf{T}} \mid \langle \bar{\mathbf{T}}, \bar{\mathbf{T}} \rangle \mid \bar{\mathbf{T}} \triangleleft \bar{\mathbf{T}} \\
\text{neutral} & \bar{\mathbf{N}} ::= & \mathbf{X} \mid \bar{\mathbf{N}}!\bar{\mathbf{S}} \mid \mathsf{if}\ \bar{\mathbf{N}}/\mathbf{X}.\bar{\mathbf{S}}\ \mathsf{then}\ \bar{\mathbf{T}}\ \mathsf{else}\ \bar{\mathbf{T}} \\
& & \mid\ \bar{\mathbf{N}}\,\bar{\mathbf{T}} \mid \mathsf{fst}\,\bar{\mathbf{N}} \mid \mathsf{snd}\,\bar{\mathbf{N}} \mid \mathsf{rec}\ \bar{\mathbf{N}}/\mathbf{X}.\bar{\mathbf{S}}\ \mathsf{with}\ \bar{\mathbf{T}}
\end{array}
$$

We say that normal terms are *canonical* if they are built by canonical constructors, i.e., $\langle\rangle$, tt, ff, $\lambda\_.\ \_$, $\langle \_,\_ \rangle$, $\_ \triangleleft \_$; other normal terms are *neutral*. Similarly, the *canonical sets* are the ground sets and those introduced by the three set-binders: the neutral sets are those given by large elimination on neutral elements of 2. As the base case for neutral terms is the *variable* case, we can readily see that there are no neutral terms in the empty context. Hence, in the

empty context, all normal forms are canonical, for both sets and values: we call this property *canonicity*.

We specify *definitional* equality ($\equiv$) on terms by giving an equivalence on their values. Until section 6, when we introduce an important modification, α-equivalence on values will suffice.

As usual, we take contexts to assign sets to variables:

$$\mathbf{Ctxt} ::= \mathcal{E} \mid \mathbf{Ctxt}; \mathbf{X} : \mathbf{S}$$

We can now give the rules of *context validity*, *set formation* and *type synthesis*.

$$\boxed{\mathbf{Ctxt} \vdash}$$

$$\frac{}{\mathcal{E} \vdash} \qquad \frac{\Gamma \vdash S\ \mathbf{set}}{\Gamma; x : S \vdash}\ x \notin \Gamma$$

$$\boxed{\mathbf{Ctxt} \vdash\ \mathbf{S\ set}}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash G\ \mathbf{set}}\ G \in \mathbf{G} \qquad \frac{\Gamma \vdash S\ \mathbf{set} \quad \Gamma; x : S \vdash T\ \mathbf{set}}{\Gamma \vdash Bx : S.\, T\ \mathbf{set}}\ B \in \mathbf{B}$$

$$\frac{\Gamma \vdash b : 2 \quad \Gamma \vdash T\ \mathbf{set} \quad \Gamma \vdash F\ \mathbf{set}}{\Gamma \vdash \mathsf{If}\ b\ \mathsf{Then}\ T\ \mathsf{Else}\ F\ \mathbf{set}}$$

$$\boxed{\mathbf{Ctxt} \vdash\ \mathbf{T} : \mathbf{S}}$$

$$\frac{\Gamma \vdash\ s : S \quad S \equiv T}{\Gamma \vdash\ s : T}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash\ \langle\rangle : \mathbf{1}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash\ \mathsf{tt} : 2} \qquad \frac{\Gamma \vdash}{\Gamma \vdash\ \mathsf{ff} : 2}$$

$$\frac{\Gamma; x{:}S \vdash\ t : T}{\Gamma \vdash\ \lambda x{:}S.\, t : \Pi x{:}S.\, T} \qquad \frac{\Gamma \vdash\ s : S \quad \Gamma \vdash\ t : T[s]}{\Gamma \vdash\ \langle s,t\rangle_{\Sigma x S.T} : \Sigma x{:}S.\, T}$$

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

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

$$\frac{\Gamma \vdash\ p : \Sigma x{:}S.\, T}{\Gamma \vdash\ \mathsf{fst}\, p : S} \qquad \frac{\Gamma \vdash\ p : \Sigma x{:}S.\, T}{\Gamma \vdash\ \mathsf{snd}\, p : T[\mathsf{fst}\, p]}$$

$$\frac{\begin{array}{l}\Gamma \vdash\ z : 0 \\ \Gamma \vdash\ P\ \mathbf{set}\end{array}}{\Gamma \vdash\ z!P : P} \qquad \frac{\begin{array}{ll}\Gamma \vdash\ b : 2 & \Gamma \vdash\ t : P[\mathsf{tt}] \\ \Gamma; x : 2 \vdash\ P\ \mathbf{set} & \Gamma \vdash\ f : P[\mathsf{ff}]\end{array}}{\Gamma \vdash\ \mathsf{if}\ b/x.P\ \mathsf{then}\ t\ \mathsf{else}\ f : P[b]}$$

$$\frac{\begin{array}{l}\Gamma \vdash\ u : \mathsf{W}x{:}S.\, T \quad \Gamma; w : \mathsf{W}x{:}S.\, T \vdash\ P\ \mathbf{set} \\ \Gamma \vdash\ p : \Pi s{:}S.\, \Pi f{:}T[s] \to \mathsf{W}x{:}S.\, T. \\ \qquad (\Pi t{:}T[s]\,.\, P[f\ t]) \to P[s \lhd f]\end{array}}{\Gamma \vdash\ \mathsf{rec}\ u/w.P\ \mathsf{with}\ p : P[u]}$$

2.1 Examples, and a Problem

Although this is rather a small theory, lots of familiar constructions can be expressed in terms of it. For example, the binary sum type, $S{+}T$ can be coded as the pair of a tag choice and a suitable element, whose type is computed from the tag by large elimination.

$$\begin{array}{rcl} S{+}T & \mapsto & \Sigma b{:}2.\ \mathsf{If}\ b\ \mathsf{Then}\ S\ \mathsf{Else}\ T \\ \mathsf{inl}\ s & \mapsto & \langle \mathsf{tt}, s\rangle \end{array}$$

$$\mathrm{inr}\, t \;\mapsto\; \langle \mathsf{ff}, t \rangle$$

In much the same way, the natural numbers can be defined as the *well-founded* structure with two shapes: the 'successor' shape has one recursive position and the 'zero' shape has none.

$$\begin{aligned} \mathrm{Tr}\, b &\mapsto \mathsf{If}\ b\ \mathsf{Then}\ 1\ \mathsf{Else}\ 0 \\ \mathrm{Nat} &\mapsto \mathsf{W} b{:}2.\ \mathrm{Tr}\, b \\ \mathrm{zero} &\mapsto \mathsf{ff} \triangleleft \lambda z.\ z!\mathrm{Nat} \\ \mathrm{suc}\, n &\mapsto \mathsf{tt} \triangleleft \lambda \_.\ n \end{aligned}$$

To construct elements, we either indicate the 'zero' shape and supply the trivial child-function, or we choose the 'successor' shape and make the child-function return the predecessor.

Operators such as addition can now be defined via the recursor:

$$\begin{aligned} \mathrm{plus} \;\mapsto\; & \lambda x\, y.\ \mathsf{rec}\ x\ \mathsf{with} \\ & \lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ \lambda f\, h.\ \mathrm{suc}\ (h\ \langle\rangle)\ \mathsf{else}\ \lambda f\, h.\ y \end{aligned}$$

Although it is reassuring that we can implement specific familiar operations, we run into trouble when we try to give a general induction scheme for this coding of Nat. Suppose we have some $n{:}\mathrm{Nat} \vdash P\ \mathbf{set}$. We should like to find

$$\begin{aligned} \mathrm{ind}_P : {} & P[\mathrm{zero}] \to (\Pi n{:}\mathrm{Nat}.\ P[n] \to P[\mathrm{suc}\, n]) \to \\ & \Pi n{:}\mathrm{Nat}.\ P[n] \end{aligned}$$

but the 'obvious' program, following the same pattern as plus,

$$\begin{aligned} \mathrm{ind}_P \;\mapsto\; & \lambda pz\, ps\, n.\ \mathsf{rec}\ n\ \mathsf{with} \\ & \lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ \lambda f\, h.\ \boxed{ps\ (f\ \langle\rangle)\ (h\ \langle\rangle)}\ \mathsf{else}\ \lambda f\, h.\ \boxed{pz} \end{aligned}$$

does not typecheck, and for the most infuriating of reasons.

In the 'zero' case, we must prove $P[\mathsf{ff} \triangleleft f]$, but the proof we offer is $pz : P[\mathsf{ff} \triangleleft \lambda z.\ z!\mathrm{Nat}]$. That is, we are obliged to show that P holds for any implementation of 'zero', but we only know that it holds for a specific implementation of 'zero'. Similarly, in

the 'successor' case, we need a proof of $P[\mathsf{tt} \lhd f]$, but we supply a proof of $P[\mathsf{tt} \lhd \lambda\_.\ f\ \langle\rangle]$, which is again a more specific thing.[3]

This is a real problem. There are infinitely many implementations of 'zero'. For example,

$$\mathrm{zero}' \ \mapsto\ \mathsf{ff} \lhd \lambda z.\ \mathrm{suc}\ (\mathrm{suc}\ \mathrm{zero})$$

defines 'zero' to be 'the number with no predecessors, all of which are two'! The frustrating thing is that these distinctions cannot be observed from within the theory. They are all *extensionally* equal. In each case of ind_P, we can prove that the child-functions given and required coincide on all inputs, so our program is typable in ETTs. With observational equality, we shall be able to bridge the typing gaps explicitly, with coercions.

2.2 An Inductive-Recursive Universe

We can define our core theory as an inductive-recursive universe in the intensional setting of Agda 2. Firstly, we must declare the type constructors we shall need: 0 is the datatype with no constructors,

```
data Empty : Set where
```

1 is the record type with no fields

```
record Unit : Set where
```

and 2 is the usual Boolean datatype

```
data Bool : Set where
  tt : 2
  ff : 2
```

Meanwhile, we can re-use the system's own Π-sets, written

$$(x : S) \to T\ x$$

we can define Σ-sets as records,

```
record Σ (S : Set)(T : S → Set) : Set where
  fst : S
  snd : T fst
```

and give W-sets as a datatype.

```
data W (S : Set)(T : S → Set) : Set where
  _◁_ : (x : S) → (T x → W S T) → W S T
```

Now we can build the collection of these sets, simultaneously defining a datatype to 'name' them and the function which decodes names as sets

```
mutual
  data 'set' : Set where
    '0', '1', '2' : 'set'
    'Π', 'Σ', 'W' : ('S' : 'set') → (⟦S⟧ → 'set') → 'set'

  ⟦_⟧ : 'set' → Set
  ⟦'0'⟧       = Empty
  ⟦'1'⟧       = Unit
  ⟦'2'⟧       = Bool
  ⟦'Π' S T⟧   = (x : ⟦S⟧) → ⟦T x⟧
  ⟦'Σ' S T⟧   = Σ ⟦S⟧ (λx ↦ ⟦T x⟧)
  ⟦'W' S T⟧ = W ⟦S⟧ (λx ↦ ⟦T x⟧)
```

With this encoding, we can readily define all our constructors and eliminators of our core theory as functions in Agda 2, either invoking the coresponding Agda constructors, or implementing the computational behaviour exactly as we have specified it. This gives us a translation $\hat{\_}$ taking core sets to elements of 'set' and core

[3] Adding η-laws for Π and 1 solves this particular problem, but does not help the general case.

elements of a core set S to elements of $[\![\hat{S}]\!]$, preserving both type and computational behaviour. This is the first step towards the inheritance of strong normalisation and canonicity from an existing intensional theory.

2.3 A Propositional Fragment

We shall shortly construct our propositional equality, but before we do so, let us consider what constitutes a *proposition* in this setting. Of course, we could just identify propositions with sets, but we prefer to be more precise: we want to know that *proofs* have no interesting computational content. Correspondingly, we shall identify a sublanguage of propositions, then explain how to interpret them as sets of proofs.

$$\mathbf{P} ::= \bot \mid \top \mid \mathbf{P} \wedge \mathbf{P} \mid \forall \mathbf{X}{:}\mathbf{S}.\, \mathbf{P}$$

Let us be clear that this is by no means the largest sublanguage of content-free sets. Rather, it is the smallest such language fit for our current purpose.

We introduce a judgment form to distinguish the well-formed propositions, and we show how the latter may be interpreted admissibly as sets of proofs,

$$\boxed{\mathbf{Ctxt} \vdash \mathbf{P}\ \mathbf{prop}} \qquad \frac{\Gamma \vdash \quad P\ \mathbf{prop}}{\Gamma \vdash \ \lceil P \rceil\ \mathbf{set}}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \ \bot\ \mathbf{prop}} \qquad \lceil \bot \rceil \mapsto 0$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \ \top\ \mathbf{prop}} \qquad \lceil \top \rceil \mapsto 1$$

$$\frac{\Gamma \vdash P\ \mathbf{prop} \quad \Gamma \vdash Q\ \mathbf{prop}}{\Gamma \vdash P \wedge Q\ \mathbf{prop}} \qquad \lceil P \wedge Q \rceil \mapsto \Sigma\_ : \lceil P \rceil .\ \lceil Q \rceil$$

$$\frac{\Gamma \vdash S\ \mathbf{set} \quad \Gamma; x : S \vdash P\ \mathbf{prop}}{\Gamma \vdash \forall x : S.\ P\ \mathbf{prop}} \qquad \lceil \forall x : S.\ P \rceil \mapsto \Pi x : S.\ \lceil P \rceil$$

Propositional implication, $P \Rightarrow Q$, is coded as $\forall \_ : \lceil P \rceil .\ Q$. We say that a term is a *proof* if it inhabits some $\lceil P \rceil$.

To some extent, this is just window-dressing. We could just have worked with sets which happen to be propositional. However, we have an ulterior motive. Consider which observations on proofs can interfere with the world of *non-propositional* sets: $\top$ has no observations; both projections from a proof of $P \wedge Q$ are proofs; applying a proof of $\forall x : S.\ P$ yields a proof; only from a contradiction can we make *data*!

This has at least two useful consequences. Firstly, it allows us to erase all proof objects from run-time code: if we never compute under λ, we shall never find a proof of $\bot$, so proofs are just dead code. Secondly, we may extend the language of proofs with whatever propositional laws we like, as long as they are *consistent*, and yet retain canonicity.

We can also express this propositional fragment as an inductive-recursive universe in Agda, this time decoding into our 'set' universe rather than the native Set, exactly following the rules above.

```
mutual
  data 'prop' : Set where
    '⊥', '⊤' : 'prop'
    '∧' : 'prop' → 'prop' → 'prop'
    '∀' : (S : 'set') → (⟦S⟧ → 'prop') → 'prop'
  ⌈_⌉ : 'prop' → 'set'
  ...
```

3. Equality, Coercion and Laziness

In this section, we shall try to implement the operation which transports values between equal sets. As we do so, we shall find out what the necessary consequences of set equality should be, and that will tell us how set equality should be defined! Let us introduce new operators for set equality and coercion, by recursion on sets,

$$\frac{\Gamma \vdash\ S\ \mathbf{set} \quad \Gamma \vdash\ T\ \mathbf{set}}{\Gamma \vdash\ S = T\ \mathbf{prop}} \qquad \frac{\Gamma \vdash\ Q : \lceil S = T \rceil \quad \Gamma \vdash\ s : S}{\Gamma \vdash\ s\,[Q{:}S{=}T\rangle : T}$$

together with their value-level counterparts, which we shall define in the next section.

$$\frac{\Gamma \vdash\ s : S \quad \Gamma \vdash\ t : T}{\Gamma \vdash\ (s : S) = (t : T)\ \mathbf{prop}}$$

$$\frac{\Gamma \vdash\ Q : \lceil S = T \rceil \quad \Gamma \vdash\ s : S}{\Gamma \vdash\ \{s \parallel Q{:}S{=}T\} : \lceil (s : S) = (s\,[Q{:}S{=}T\rangle : T) \rceil}$$

As you can see, coercion apparently introduces a new way for proofs to interfere with data. However, we shall define coercion recursively in terms of the existing operations on proofs, thus retaining both the erasability of propositions and, crucially, our freedom to add consistent laws.

As the sets inhabited by canonical values are themselves canonical, it is sufficient to explain how to compute a coercion between any two canonical sets, given a proof that they are equal. Let us deal with the easy cases first: at identical ground types, there is nothing to do!

$$z\,[Q{:}\,0{=}0\,\rangle \mapsto z$$
$$u\,[Q{:}\,1{=}1\,\rangle \mapsto u$$
$$b\,[Q{:}\,2{=}2\,\rangle \mapsto b$$

Correspondingly, we may take

$$0 = 0 \mapsto \top$$
$$1 = 1 \mapsto \top$$
$$2 = 2 \mapsto \top$$

The hard work will come when explaining what to do with types made with the same binder. Even harder are the off-diagonal cases, so we shall have to rule them out.

$$\begin{aligned}
f_0\ [Q:\ \Pi x_0 : S_0.\ T_0 = \Pi x_1 : S_1.\ T_1\ ] &\mapsto ? \\
p_0\ [Q:\ \Sigma x_0 : S_0.\ T_0 = \Sigma x_1 : S_1.\ T_1\ ] &\mapsto ? \\
(s_0 \lhd f_0)\ [Q:\ \mathsf{W} x_0 : S_0.\ T_0 = \mathsf{W} x_1 : S_1.\ T_1\ ] &\mapsto ? \\
x\ [Q:\ S = T\ ] &\mapsto Q!T \quad \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
(\Pi x_0 : S_0.\ T_0) = (\Pi x_1 : S_1.\ T_1) &\mapsto ? \\
(\Sigma x_0 : S_0.\ T_0) = (\Sigma x_1 : S_1.\ T_1) &\mapsto ? \\
(\mathsf{W} x_0 : S_0.\ T_0) = (\mathsf{W} x_1 : S_1.\ T_1) &\mapsto ? \\
S = T &\mapsto \bot \quad \text{for other canonical sets}
\end{aligned}$$

We must fill in those ?s. Let us start with Σ-types, as tuples are relatively mundane, compared to functions or well-founded trees. We must solve

$$\begin{aligned}
\ldots; Q : (\Sigma x_0 : S_0.\ T_0) = (\Sigma x_1 : S_1.\ T_1); \\
p_0 : \Sigma x_0 : S_0.\ T_0 \\
\vdash\ ? : \Sigma x_1 : S_1.\ T_1
\end{aligned}$$

Let us work by refinement, following the trail left by the types. Some moves are clear. Let us name the input's projections for convenient access, then construct the output componentwise. We use an informal **let** notation for 'definitions' to be substituted out in the final term. This notation is not part of our theory, but makes our terms a bit more legible.

$$\begin{aligned}
&\textbf{let} \\
&\quad \mathrm{s}_0 \mapsto \mathsf{fst}\ p_0 : S_0 \\
&\quad \mathrm{t}_0 \mapsto \mathsf{snd}\ p_0 : T_0[\mathrm{s}_0] \\
&\quad \mathrm{s}_1 \mapsto ? : S_1
\end{aligned}$$

$\mathrm{t}_1 \mapsto ? : T_1[\mathrm{s}_1]$
in $\langle \mathrm{s}_1, \mathrm{t}_1 \rangle$

Now, the only way we can make an output equal to the input is if the output components are equal to the input components. We must make s_1 by coercing s_0 and t_1 by coercing t_0. We get

let
$Fs_0 \mapsto \mathsf{fst}\, p_0 : S_0$
$Ft_0 \mapsto \mathsf{snd}\, p_0 : T_0[\mathrm{s}_0]$
$\mathrm{Q}_S \mapsto ? : \lceil S_0 = S_1 \rceil$
$\mathrm{s}_1 \mapsto \mathrm{s}_0\, [\mathrm{Q}_S : S_0 = S_1 \rangle : S_1$
$\mathrm{Q}_T \mapsto ? : \lceil T_0[\mathrm{s}_0] = T_1[\mathrm{s}_1] \rceil$
$\mathrm{t}_1 \mapsto \mathrm{t}_0\, [\mathrm{Q}_T : T_0[\mathrm{s}_0] = T_1[\mathrm{s}_1] \rangle : T_1[\mathrm{s}_1]$
in $\langle \mathrm{s}_1, \mathrm{t}_1 \rangle$

It remains to find proofs of the equations which justify the coercions. These concern the components of the Σ-types, and must surely be extracted from Q. We shall clearly need Q to tell us that $S_0 = S_1$. However, in the case of Q_T, we are obliged to show that the s_0 instance of T_0 equals the s_1 instance of T_1, where s_0 and s_1 are unknown when Q's type is being determined. It is too much to require that *arbitrary* instances of T_0 and T_1 are equal—that would force us to remove any meaningful dependency. We can, however require T_0 and T_1 to coincide whenever they are instantiated *equally*, for s_1 is a coercion of Fs_0, hence equal by *coherence*.

Let us therefore take

$$(\Sigma x_0 : S_0.\, T_0) = (\Sigma x_1 : S_1.\, T_1) \mapsto$$
$$S_0 = S_1 \wedge$$

$$\forall x_0 : S_0.\ \forall x_1 : S_1.\ (x_0 : S_0) = (x_1 : S_1) \Rightarrow T_0[x_0] = T_1[x_1]$$

so that we can have

$$\begin{array}{l} \mathrm{Q}_S \mapsto \mathsf{fst}\ Q \\ \mathrm{Q}_T \mapsto (\mathsf{snd}\ Q)\ \mathrm{s}_0\ \mathrm{s}_1\ \{\mathrm{s}_0 \parallel \mathrm{Q}_S : S_0 = S_1\} \end{array}$$

completing the case for Σ-types.

For Π-types, we can expect a little contravariant twist. We need to coerce the argument s_1 right-to-left, so that we can apply the function f_0, then coerce the result left-to-right. Working in the same style, we have

$$\begin{array}{l} \ldots;\ Q : (\Pi x_0 : S_0.\ T_0) = (\Pi x_1 : S_1.\ T_1); \\ \quad f_0 : \Pi x_0 : S_0.\ T_0 \\ \vdash\ \lambda s_1.\ \mathsf{let} \\ \qquad\qquad F\mathrm{Q}_S \mapsto ? : \lceil S_1 = S_0 \rceil \\ \qquad\qquad \mathrm{s}_0 \mapsto s_1\ [\mathrm{Q}_S : S_1 = S_0\rangle : S_0 \\ \qquad\qquad \mathrm{t}_0 \mapsto f_0\ \mathrm{s}_0 : T_0[\mathrm{s}_0] \\ \qquad\qquad \mathrm{Q}_T \mapsto ? : \lceil T_0[\mathrm{s}_0] = T_1[s_1]\ \rceil \\ \qquad\qquad \mathrm{t}_1 \mapsto \mathrm{t}_0\ [\mathrm{Q}_T : T_0[\mathrm{s}_0] = T_1[s_1]\rangle : T_1[s_1] \\ \qquad\ \mathsf{in}\ \mathrm{t}_1 \\ \quad : \Pi x_1 : S_1.\ T_1 \end{array}$$

Correspondingly, we should take

$$\begin{array}{l} (\Pi x_0 : S_0.\ T_0) = (\Pi x_1 : S_1.\ T_1)\ \mapsto \\ \quad S_1 = S_0\ \wedge \\ \quad \forall x_1 : S_1.\ \forall x_0 : S_0.\ (x_1 : S_1) = (x_0 : S_0) \Rightarrow T_0[x_0] = T_1[x_1] \end{array}$$

$$\begin{array}{l} \mathrm{Q}_S \mapsto \mathsf{fst}\ Q \\ \mathrm{Q}_T \mapsto (\mathsf{snd}\ Q)\ s_1\ \mathrm{s}_0\ \{s_1 \parallel \mathrm{Q}_S : S_1 = S_0\} \end{array}$$

In the case of W types, we shall need a recursive coercion, shifting shapes and child-functions left-to-right: for the latter, we

shall need to shift child-positions right-to-left, and the resulting child-trees left-to-right recursively. Perhaps you can already guess that we shall take

$$
\begin{array}{l}
(\mathsf{W}x_0 : S_0.\ T_0) = (\mathsf{W}x_1 : S_1.\ T_1) \ \mapsto \\
\quad S_0 = S_1 \wedge \\
\quad \forall x_0 : S_0.\ \forall x_1 : S_1.\ (x_0 : S_0) = (x_1 : S_1) \Rightarrow T_1[x_1] = T_0[x_0]
\end{array}
$$

The coercion can then be defined recursively as follows:

$$
\begin{array}{l}
(s_0 \lhd f_0)\ [Q{:}\mathsf{W}x_0 : S_0.\ T_0 = \mathsf{W}x_1 : S_1.\ T_1\rangle \ \mapsto \\
\quad \textbf{let} \\
\qquad \mathrm{Q}_S \mapsto \mathsf{fst}\ Q : \lceil S_0 = S_1 \rceil \\
\qquad \mathrm{s}_1 \mapsto s_0\ [\mathrm{Q}_S{:}S_0 = S_1\rangle : S_1 \\
\qquad \mathrm{Q}_T \mapsto (\mathsf{snd}\ Q)\ \mathrm{s}_0\ \mathrm{s}_1\ \{s_0 \parallel \mathrm{Q}_S{:}S_0 = S_1\} : \lceil T_1[\mathrm{s}_1] = T_0[s_0] \rceil \\
\quad \textbf{in}\ \mathrm{s}_1 \lhd \lambda t_1.\ \textbf{let}\ \mathrm{t}_0 \mapsto t_1\ [\mathrm{Q}_T{:}T_1[\mathrm{s}_1] = T_0[s_0]\rangle : T_0[s_0] \\
\qquad\qquad \textbf{in}\ f_0\ \mathrm{t}_0\ [Q{:}\mathsf{W}x_0 : S_0.\ T_0 = \mathsf{W}x_1 : S_1.\ T_1\rangle
\end{array}
$$

We have completed the first stage of our journey. What we have produced may look complex and detailed, but there was relatively little choice at any point. The structure of the types determined the structure of the coercions between them. This delivered in turn the requirements which determined what set equality should be: impossible off the diagonal; trivial at ground types; for binders, equal domains and equal range instances for equal domain values, twisted according to variance.

3.1 Value Equality

We must now explain when values are equal. The idea is that values should be equal when they support equal observations. For those sets equipped with inductive eliminators (0, 2, W-sets), this amounts to equality of construction. For those equipped only with projective eliminators (1, Σ, Π), we are free to require only that projections coincide.

$$
\begin{aligned}
(z_0 : 0) = (z_1 : 0) &\mapsto \top \\
(u_0 : 1) = (u_1 : 1) &\mapsto \top \\
(\mathsf{tt} : 2) = (\mathsf{tt} : 2) &\mapsto \top \\
(\mathsf{tt} : 2) = (\mathsf{ff} : 2) &\mapsto \bot \\
(\mathsf{ff} : 2) = (\mathsf{tt} : 2) &\mapsto \bot \\
(\mathsf{ff} : 2) = (\mathsf{ff} : 2) &\mapsto \top
\end{aligned}
$$

$$
\begin{aligned}
(f_0 : \Pi x_0 : S_0 . T_0) = (f_1 : \Pi x_1 : S_1 . T_1) &\mapsto \\
\forall x_0 : S_0 . \forall x_1 : S_1 . (x_0 : S_0) = (x_1 : S_1) &\Rightarrow \\
(f_0\ x_0 : T_0[x_0]) = (f_1\ x_1 : T_1[x_1]) &
\end{aligned}
$$

$$
\begin{aligned}
(p_0 : \Sigma x_0 : S_0 . T_0) = (p_1 : \Sigma x_1 : S_1 . T_1) &\mapsto \\
(\mathsf{fst}\ p_0 : S_0) = (\mathsf{fst}\ p_1 : S_1) \wedge & \\
(\mathsf{snd}\ p_0 : T_0[\mathsf{fst}\ p_0]) = (\mathsf{snd}\ p_1 : T_1[\mathsf{fst}\ p_1]) &
\end{aligned}
$$

$$
\begin{aligned}
(s_0 \lhd f_0 : \mathsf{W} x_0 : S_0 . T_0) = (s_1 \lhd f_1 : \mathsf{W} x_1 : S_1 . T_1) &\mapsto \\
(s_0 : S_0) = (s_1 : S_1) \wedge & \\
\forall y_0 : T_0[s_0] . \forall y_1 : T_1[s_1] . (y_0 : T_0[s_0]) = (y_1 : T_1[s_0]) &\Rightarrow \\
(f_0\ y_0 : \mathsf{W} x_0 : S_0 . T_0) = (f_1\ y_1 : \mathsf{W} x_1 : S_1 . T_1) &
\end{aligned}
$$

$$
\begin{aligned}
(\_ : T_0) = (\_ : T_1) &\mapsto \bot \\
T_0, T_1 \text{ other canonical sets} &
\end{aligned}
$$

As you can see, two functions are equal if they take equal inputs to equal outputs; two pairs are equal if they have equal projections. Meanwhile, as the eliminators for 2 and W allow us to observe the construction of their contents, equality for elements of these types is equality of construction. Note that this definition of value equality preserves set equality: if we know that the values on the left inhabit equal sets, then we know that every equation on the right relates values in equal sets.

What can impede the computation of $=$ for values? Only the presence of a neutral set, or of a neutral value in 2 or a W-set.

We have shown how to construct equations between sets and how these equations, if provable, yield coercions. We have also

shown how to construct observational justifications for the equality of terms. This does *not* complete our presentation of OTT. We shall need to add more introduction rules for equations which are not directly observational. However, we have now given all of OTT's *computation* schemes. Whatever we add will not change the computational behaviour of the system, only enlarge the collection of

provable equations. Correspondingly, it is a good time to consider the metatheoretical properties of these computation schemes.

3.2 Modelling Equality and Coercion yields Normalisation

The definition of equality and coercion, given above, is entirely representable in our Agda model. We show how our universe can readily be equipped with equality and coercion by *definitional extension*. Moreover, we ensure that all the computation schemes are faithfully simulated. By this means, we shall be able to demonstrate that OTT is strongly normalising. We define

```
mutual
  Eq : 'set' → 'set' → 'prop'
  ...
  eq : (S : 'set') → ⟦S⟧ → (T : 'set') → ⟦T⟧ → 'prop'
  ...
```

with operational behaviour delivering exactly the above computation schemes. These functions cover all the canonical sets and values from our core theory: they introduce no 'new' propositions.

Of course, we must be sure that these programs terminate. The contravariant twists we used to keep coercion simple mean that our recursion is not directly structural: although we appeal to recursion

on smaller elements of 'set', we swap the argument positions in which they appear. We shall address this issue shortly.

Once we know what equality is, we may introduce coercion and coherence as a mutual definition:

mutual

$$\begin{aligned}
&\mathrm{coe} : (S : \text{'set'})(T : \text{'set'}) \to [\![\lceil \mathrm{Eq}\ S\ T \rceil]\!] \to [\![S]\!] \to [\![T]\!] \\
&\mathrm{coe}\ \text{'0'}\ \text{'0'}\ Q\ z\ =\ z \\
&\mathrm{coe}\ \text{'0'}\ \text{'1'}\ Q\ z\ =\ Q!\mathsf{Unit} \\
&\vdots \\
&\mathrm{coe}\ (\text{'W'}\ S_0\ T_0)\ (\text{'W'}\ S_1\ T_1)\ Q\ (s_0 \lhd f_0)\ =\ \cdots \\
&\mathrm{coh} : (S : \text{'set'})(T : \text{'set'})(Q : [\![\lceil \mathrm{Eq}\ S\ T \rceil]\!])(s : [\![S]\!]) \to \\
&\qquad \mathrm{eq}\ S\ s\ T\ (\mathrm{coe}\ S\ T\ Q\ s) \\
&\mathrm{coh}\ S\ T\ Q\ s\ =\ ?
\end{aligned}$$

The definition of coercion follows the same twisted recursion scheme on the names of types, combined lexicographically with structural recursion on trees in W-types, when required. Correspondingly, the computation schemes for coercion are faithfully simulated by the embedding. We shall keep you in suspense about the definition of coherence a little while longer.

LEMMA 1 (Strong Normalisation). *OTT is strongly normalising.*

PROOF SKETCH Our approach to strong normalisation is to *simulate* the reduction behaviour of OTT within a strongly normalising intensional type theory. We proceed in two steps.

Firstly, we must account for the non-structual twisting in contravariant positions. There are advanced ways to do this [2], but the basic, clumsy way is to define the equality in the model without twists and close the simulating set modelling each OTT equation under symmetry. If we do this, we must extend the model of co-

ercion to give us both directions at once, and switch direction in contravariant positions. By doing so, we acquire a model in the system of inductive-recursive definitions proposed by Dybjer and Setzer [9]. Although this system has a set-theoretic model and is widely suspected to be normalising, we cannot consider the job done.

Correspondingly, the second step is to code the induction-recursion within a standard type theory such as Luo's UTT [17] or the Calculus of Inductive Constructions [28]. The usual coding trick, discovered independently by Capretta, McBride and doubtless others, is to turn the decoding function into an index. Where Agda allows us

$$\textbf{data}\ \text{`set'} : \mathsf{Set}\ \ldots\ [\![\_]\!] : \text{`set'} \to \mathsf{Set}$$

we may instead resort to an indexed-family, given a larger universe

$$\mathsf{HasCode} : \mathsf{Set} \to \mathsf{Type}$$

and rearrange our constructions accordingly,

$$\text{`set'} = \Sigma S{:}\ \mathsf{Set}.\ \mathsf{HasCode}\ S \qquad [\![\_]\!] = \mathrm{fst}$$

We may define the operations, all as a package, using just the standard eliminator for structural recursion on HasCode. □

4. Canonicity from Consistency

We are now in a position to investigate the issue of canonicity for OTT. By adding coercion, we introduced a new way for proofs of propositions to interact with calculations on data, so we must check that coercion never 'gums up the works'. Our careful separation of propositions from sets allows us to proceed in two stages: first, we check that *logical contradiction* is the only possible source of non-canonical *data*; then, we check that our *proof*-language is free of such contradictions.

LEMMA 2 (Canonicity from Consistency). *Suppose OTT is consistent, i.e. that there is no s such that $\mathcal{E} \vdash s : 0$. Then, for all normal S and s,*

- *if $\mathcal{E} \vdash S$ **set** then S is canonical;*
- *if $\mathcal{E} \vdash s : S$ then either s is canonical, or s is a proof.*

PROOF We proceed by mutual induction on the normal forms S and s. Clearly we need only consider how to rule out neutral sets and values. We have three forms of neutral set:

- $\mathcal{E} \vdash \mathsf{If}\ b\ \mathsf{Then}\ T\ \mathsf{Else}\ F$ **set** This must follow from some $\mathcal{E} \vdash b : 2$; however, 2 is not a proposition, so inductively, b must be tt or ff, hence our set computes further, contradicting the hypothesis that it is in normal form.
- $\mathcal{E} \vdash \lceil S_0 = S_1 \rceil$ **set** This must follow from $\mathcal{E} \vdash S_0$ **set** and $\mathcal{E} \vdash S_1$ **set**. Inductively, S_0 and S_1 are canonical, so the equation computes further.
- $\mathcal{E} \vdash \lceil (s_0 : S_0) = (s_1 : S_1) \rceil$ **set** Inductively, the two sets must be canonical. Moreover, if S_i is 2 or a W-set, then s_i is also canonical. Hence the equation computes further.

Meanwhile, for the values, the result holds for

- canonical term-formers, directly;
- variables, vacuously, because the context is empty;
- $z!S$, by our assumption of consistency;
- $\mathsf{if}\ b/x.P\ \mathsf{then}\ t\ \mathsf{else}\ f$, because $b : 2$ must be canonical, inductively, hence the conditional computes;
- $\mathsf{rec}\ w/x.P\ \mathsf{with}\ p$, because $w : \mathsf{W}x : S.\ T$ must be canonical, inductively, hence the recursor computes;
- $f\ a$, because, inductively, f must be either
  - $\lambda x.\ t$, contradicting normality of $f\ a$, or

  - a proof, in which case so is $f\ a$;
- fst p and snd p, by the same argument—these must be proofs because, inductively, p must be a proof;
- $\{s \parallel Q{:}S{=}T\}$, because this is a proof;
- $s\,[Q{:}S{=}T\rangle$, because inductively, S and T are canonical which is enough to make the coercion compute, except in the case where they are both W-sets, in which case we also need the inductive hypothesis that s is canonical. □

The impact of this lemma is significant: it tells us that we can add introduction rules for any *propositions* we like, provided that they are consistent, without threatening either normalisation (because they do not compute) or canonicity (because this is ensured by consistency). Our hard work in designing this coercion mechanism to ensure the transportation of canonical values between equal types has bought us the freedom to design the propositional fragment of OTT for our convenience, within reason.

4.1 Laws for Equality

Which equational laws shall we have? We should certainly like to recover refl and subst, so let us have

$$\frac{\Gamma \vdash\ s : S}{\Gamma \vdash\ \overline{s{:}S} : \lceil (s : S) = (s : S) \rceil}$$

$$\frac{\Gamma \vdash\ S\ \mathbf{set} \quad \Gamma; x{:}S \vdash\ T\ \mathbf{set}}{\Gamma \vdash\ \mathrm{R}x{:}S.\,T : \lceil \forall y{:}S.\,\forall z{:}S. \\ (y : S) = (z : S) \Rightarrow T[y] = T[z] \rceil}$$

The former is just reflexivity for values. The latter asserts that any set abstracting a value must preserve the value equality—any such abstracted set thus induces a substitution operator for values.

$$\begin{aligned}\mathrm{subst}[x\!:\!S.\,T] \mapsto\ & \lambda y\ z\ q\ t.\ t\ [(\mathrm{R}x\!:\!S.\,T)\ y\ z\ q{:}T[y]{=}T[z]\,\rangle \\ & :\Pi y\!:\!S.\,\Pi z\!:\!S. \\ & \qquad \lceil (y:S) = (z:S) \rceil \to T[y] \to T[z]\end{aligned}$$

Note that we can now form

$$\mathrm{subst}[x\!:\!S.\,T]\ s\ s\ (\overline{s\!:\!S}) : T[s] \to T[s]$$

but that the computational behaviour of this is not uniformly the identity: it depends on the structure of T, and the value being transported. In particular,

$$\mathrm{subst}[x\!:\!2.\ \mathsf{If}\ x\ \mathsf{Then}\ T\ \mathsf{Else}\ F]\ b\ b\ (\overline{b\!:\!2})\ t$$

is bound to get stuck if b is neutral, because we can only coerce between canonical types. Altenkirch's extensional model construction [4] also had this issue, losing some computational properties of ITTs to gain some reasoning properties from ETTs. This is a big issue: the vanishing of trivial substs is what gives elaborated Epigram programs their expected computational behaviour [20, 12]. Fortunately, as you shall see later, we can recover this intensional behaviour without further adjusting our notion of evaluation.

Of course, *coherence* is enough to tell us that trivial substs are *provably* equal to the identity. We have not yet defined coherence, but as it is a propositional law, we do not need to. Let us leave our coherence operator without computational behaviour!

We should like our equalities to be equivalences. We have reflexivity for values; for sets, we can take

$$\overline{S} \mapsto (\mathrm{R}x{:}1.\ S)\ \langle\rangle\ \langle\rangle\ \langle\rangle : S = S$$

We could derive symmetry and transitivity directly if our theory was rich enough to support abstraction over sets, but even with the restricted theory of this paper, it is enough to add that equality respects equality, i.e.

$$\frac{\Gamma \vdash\ P : \lceil A = C \rceil \quad \Gamma \vdash\ Q : \lceil B = D \rceil}{\Gamma \vdash\ P \parallel Q : \lceil \lceil A = B \rceil = \lceil C = D \rceil \rceil}$$

and similarly for values, although we must be careful to consider only equations on values between equal sets

$$\frac{\begin{array}{ll}\Gamma \vdash\ P : \lceil A = C \rceil & \Gamma \vdash\ Q : \lceil B = D \rceil \\ \Gamma \vdash\ p : \lceil (a{:}A) = (c{:}C) \rceil & \Gamma \vdash\ q : \lceil (b{:}B) = (d{:}D) \rceil\end{array}}{\Gamma \vdash\ P, p \parallel Q, q : \lceil \lceil (a{:}A) = (b{:}B) \rceil = \lceil (c{:}C) = (d{:}D) \rceil \rceil}$$

Now, if $Q : X = Y$, then we can derive symmetry

$$Q^{\smile} \mapsto \overline{X}\ [Q \parallel \overline{X}{:}\lceil X = X \rceil = \lceil Y = X \rceil \rangle : \lceil Y = X \rceil$$

and if, further, $Q' : Y = Z$, then we can derive transitivity

$$Q \circ Q' \mapsto Q\ [\overline{X} \parallel Q'{:}\lceil X = Y \rceil = \lceil X = Z \rceil \rangle : \lceil X = Z \rceil$$

with similar constructions serving for equal values in equal sets.

Further, we should like every syntactic construct to respect the observational equality, making equal objects from equal subobjects. These laws already hold within OTT for each syntactic construct of our base theory, and we have just added that equality respects equality. We can easily use coherence to prove that coercion respects equality. This leaves us only with the need to prove that all of our propositional laws respect equality. We can get this in one go by adding a *proof-irrelevance* law: "equal propositions have equal

proofs".

$$\frac{\Gamma \vdash P_o \ \mathbf{prop} \quad \Gamma \vdash P_1 \ \mathbf{prop} \quad \Gamma \vdash Q : \lceil P_0 \rceil = \lceil P_1 \rceil}{\mathrm{Irr}\, Q : \lceil \forall p_0 : \lceil P_0 \rceil .\, \forall p_1 : \lceil P_1 \rceil .\, (p_0 : P_0) = (p_1 : P_1) \rceil}$$

Now, it is the case that many of our laws can be implemented by 'generic programming' in our Agda embedding, where we have access to recursion over 'set'. Certainly, Irr holds by induction on the structure of propositions. However, it is also clear that not all such laws can be provable in Agda: we represent OTT functions by Agda functions, but extensionality holds in OTT. To see this, suppose $FG : \lceil \forall x : S.\, (f\ x : T[x]) = (g\ x : T[x]) \rceil$, and observe

$$\begin{array}{l} \lambda x\ y\ xy.\ FG\ x \circ \mathsf{snd}\ (\overline{g : \Pi x : S.\, T[x]})\ x\ y\ xy \\ :\ \lceil (f : \Pi x : S.\, T[x]) = (g : \Pi x : S.\, T[x]) \rceil \end{array}$$

Extensionality does not hold in Agda, although it is consistent to assume it. Fortunately for us, our Agda embedding delivers all the computational behaviour of OTT without any need to derive all these laws. Coercion never looks at the proof, so why should we trouble to compute it?

4.2 Consistency from the Extensional Theory

There is nothing to stop us embedding OTT into an extensional type theory. The obvious way to achieve this is to embed our existing intensional translation into the corresponding extensional type theory. The latter may have poor computational properties, but it is at least *consistent*, so we can prove consistency of OTT just by finishing our homework: we must derive our propositional laws in the extensional setting.

Let us write $\equiv_A$ for the built-in propositional equality of the extensional theory, which coincides with the definitional equality by the conversion rule in one direction, and the equality reflection rule in the other. The key fact is this:

LEMMA 3 (Equality Interchange). *For all* $S_0, S_1 : \text{'set'}$,

$$[\![\lceil S_0 = S_1 \rceil]\!] \Leftrightarrow S_0 \equiv_{\text{'set'}} S_1$$

Moreover, if $S_0 \equiv_{\text{'set'}} S_1$, *then for all* $s_0 : S_0$ *and* $s_1 : S_1$,

$$[\![\lceil (s_0 : S_0) = (s_1 : S_1) \rceil]\!] \Leftrightarrow s_0 \equiv_{[\![S_0]\!]} s_1$$

PROOF Note that we are using equality reflection even to *state* this fact—the value level equation is only homogeneous because we have a proof that the sets coincide, hence their interpretations are definitionally equal.

We proceed by the usual variance-twisted induction on S_0 and S_1. The off-diagonal cases are trivial, as are the cases for equal ground sets.

For 'Π', we must show

$$[\![\lceil S_1 = S_0 \wedge \forall x_1 : S_1.\, \forall x_0 : S_0.\, (x_1 : S_1) = (x_0 : S_0) \Rightarrow T_0\, x_0 = T_1\, x_1 \rceil]\!] \Leftrightarrow \text{'}\Pi\text{'}\ S_0\ T_0 \equiv_{\text{'set'}} \text{'}\Pi\text{'}\ S_1\ T_1$$

Going right-to-left, injectivity of constructors allows us to identify $S_0 \equiv_{\text{'set'}} S_1$ and hence $T_0 \equiv_{[\![S_0]\!] \to \text{'set'}} T_1$. Using our inductive hypothesis for the domain sets, we may deduce $[\![\lceil S_1 = S_0 \rceil]\!]$; for the range, we can take $x_1 \equiv_{[\![S_0]\!]} x_0$ by the inductive hypothesis for domain values (noting that the domains have been identified), hence we have $T_0\ x_0 \equiv_{\text{'set'}} T_1\ x_1$, so, by the range inductive hypothesis, we have $[\![\lceil T_0\, x_0 = T_1\, x_1 \rceil]\!]$.

Left-to-right, our inductive hypothesis for domain sets immediately identifies them. To deduce that $T_0 \equiv_{[\![S_0]\!] \to \text{'set'}} T_1$, we may appeal to extensionality and show

$$(x : S_0) \to T_0\ x \equiv_{\text{'set'}} T_1\ x$$

our domain value hypothesis tells us that $[\![\lceil (x : S_1) = (x : S_0) \rceil]\!]$, hence we may deduce $[\![\lceil T_0\ x\ =\ T_1\ x \rceil]\!]$, ready for the range hypothesis.

On the value level, with domain and range identified, we must show

$$\begin{array}{l} [\![\lceil \forall x_0 : S.\ \forall x_1 : S.\ (x_0 : S) = (x_1 : S) \Rightarrow \\ \qquad\qquad (f_0\ x_0 : T\ x_0) = (f_1\ x_1 : T\ x_1) \rceil]\!] \\ \quad \Leftrightarrow \\ f_0 \equiv_{(x:S) \to T} f_1 \end{array}$$

Right-to-left, introducing the hypotheses allows us to identify $x_0 \equiv_S x_1$, yielding $f_0\ x_0 \equiv_{T\ x_0} f_1\ x_1$, ready for the range value hypothesis to complete the derivation. Left to right, we again appeal to extensionality, just as we did with the range sets.

For 'Σ' and 'W' the proofs on the set-level go just as they do for 'Π'. On the value-level, pairs are treated componentwise and trees require a further induction. The crux remains that whenever we have a hypothetical equation on two values, we always know their sets are equal, so our inductive hypotheses allow us to identify them. □

THEOREM 4 (Consistency). *There is no s such that* $\mathcal{E} \vdash s : 0$.

PROOF Inspecting our propositional laws, we have maintained the property that values are equated only when they inhabit provably

equal sets. We may therefore appeal to the equality interchange lemma to derive these laws in our extensional model from their counterparts with $=$ replaced by $\equiv$. All these laws hold trivially in extensional type theory. Correspondingly, every closed term in OTT induces a closed term in ETT inhabiting the corresponding set. As ETT is consistent, no closed OTT term can inhabit 0. □

COROLLARY 5 (Canonicity). *If* $\mathcal{E} \vdash S$ **set** *then* S *is canonical. If* $\mathcal{E} \vdash s : S$ *then* s *is either canonical or a proof.*

A more arduous but arguably more satisfying proof would be to complete the missing derivations of the propositional laws in an intensional theory, consistently extended with axioms for "extensional concepts", as proposed by Hofmann [13]. Such extensions give the strength of the extensional theory, whilst retaining the checkability of constructions. W. Swierstra has done most of the work required, for a minor variation of the theory presented here: his construction leaves only reflexivity unfinished, and that is where the appeal to extensionality is required.

5. Induction for Natural Numbers

Let us now put observational equality to work, implementing the induction principle for natural numbers defined via W-sets:

$$\begin{aligned}\mathrm{ind}_P : P[\mathrm{zero}] \to (\Pi n{:}\mathrm{Nat}.\ P[n] \to P[\mathrm{suc}\ n]) \to \\ \Pi n{:}\mathrm{Nat}.\ P[n]\end{aligned}$$

Recall that the problem was to show that the proof of P given for the standard implementation of a number is good for any implementation of that number. We are now ready to solve this problem.

$$\begin{aligned}&\mathrm{ind}_P \mapsto \\ &\quad \lambda pz\, ps\, n.\ \mathsf{rec}\ n\ \mathsf{with} \\ &\qquad \lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ \lambda f\, h.\ ps\ (f\ \langle\rangle)\ (h\ \langle\rangle)\end{aligned}$$

$$[?{:}P[\mathrm{suc}\ (f\ \langle\rangle)] = P[\mathsf{tt} \lhd f]\ \rangle$$
$$\mathsf{else}\ \lambda f\ h.\ pz\ [?{:}P[\mathrm{zero}] = P[\mathsf{ff} \lhd f]\ \rangle$$

To fill the holes, we can appeal to $\mathrm{R}x : P[x]$ and then provide the requisite proofs that the numbers coincide: it is trivial to show that they have the same shape, but the child-functions are equal only up to observation:

$$\begin{array}{l}\langle\langle\rangle, \lambda x\ y\ q.\ (\overline{f : 1 \to \mathrm{Nat}})\ \langle\rangle\ y\ \langle\rangle\rangle \\ \quad : (\mathrm{suc}\ (f\ \langle\rangle) : \mathrm{Nat}) = (\mathsf{tt} \lhd f : \mathrm{Nat})\end{array}$$

$$\begin{array}{l}\langle\langle\rangle, \lambda x\ y\ q.\ x!\lceil(x!\mathrm{Nat} : \mathrm{Nat}) = (f\ y : \mathrm{Nat})\rceil\rangle \\ \quad : (\mathrm{zero} : \mathrm{Nat}) = (\mathsf{ff} \lhd f : \mathrm{Nat})\end{array}$$

So, coercion via proofs based on extensional reasoning repairs the derivation of the induction principle, without loss of canonicity: $2{+}2$ is certainly a successor. We can play the same game with other inductive sets: related work on *containers* [1] shows how to encode a wide variety of structures as W-sets in an extensional setting. We may now import all those constructions, wholesale, acquiring at least the inductive families accepted by Luo's schema, as used in Epigram [10, 17].

This is progress, but, of course we want more! We want the usual *computational* behaviour as well, on *open* terms. Unfortunately

$$\begin{array}{l}\mathrm{ind}_P\ pz\ ps\ \mathrm{zero} \mapsto pz\ [\cdots{:}P[\mathrm{zero}] = P[\mathrm{zero}]\ \rangle \\ \mathrm{ind}_P\ pz\ ps\ (\mathrm{suc}\ n) \mapsto \\ \qquad ps\ n\ (\mathrm{ind}_P\ pz\ ps\ n)\ [\cdots{:}P[\mathrm{suc}\ n] = P[\mathrm{suc}\ n]\ \rangle\end{array}$$

If we are lucky, P will have a form which allows these coercions to compute, but in general, they get stuck, even though the source and target are *definitionally* equal sets.

Notice, of course, that this is not an inherent problem with the observational approach, just with the computational behaviour of higher-order encodings of data. If we add first-order presentations of datatypes to our theory, they behave as they always did.

Even with the higher-order encodings, the fact that we lose the computation rules because of coercions which a computer can tell are unnecessary is somewhat ironic. Let us see how to use computers to solve this problem!

6. Type-Directed Quotation

So far, our definitional equality has been the most basic available—α-equivalence of β-normal forms. We can do better, by post-processing our β-normal values in a type-directed way. In the literature of *normalisation by evaluation*, this process is usually called 'quotation', as it is usually the means by which semantically-represented values are reified syntactically.

We define mutually recursive operations, $S \Uparrow^{\Gamma} v$ quoting values in a known set as values, $\Downarrow^{\Gamma} n$ quoting neutral terms as neutral terms and reconstructing the set which they inhabit, $\Uparrow^{\Gamma} S$ quoting normal sets, $\Downarrow^{\Gamma} N$ quoting neutral sets. These operations perform η-expansion, much in style of Abel et al. [3]. Moreover, they detect the presence of propositional types and mark the corresponding

proofs with a box. We shall also make $\Downarrow^{\Gamma}$ eliminate stuck coercions between equal sets.

Definitional equality, $\equiv$, for sets and for values in a set now becomes $\alpha\Box$-equivalence of quoted normal forms, written $\equiv_{\alpha\Box}$. That is, we identify quotations up to renaming of variables and the equivalence of arbitrary boxed proofs. We now have *proof irrelevant* propositions.

Quotation for normal values performs η-expansion where possible, pushes through constructors, marks inhabitants of 0 and proofs of neutral propositions (i.e., unexpanded equations). The remaining terms are necessarily neutral inhabitants of datatypes 2, W-sets or

some $\mathsf{If}\ b\ \mathsf{Then}\ T\ \mathsf{Else}\ F$, so we change direction.

$$
\begin{array}{rl}
0 \Uparrow^\Gamma z & \mapsto \boxed{z} \\
1 \Uparrow^\Gamma z & \mapsto \langle\rangle \\
2 \Uparrow^\Gamma \mathsf{tt} & \mapsto \mathsf{tt} \\
2 \Uparrow^\Gamma \mathsf{ff} & \mapsto \mathsf{ff} \\
\Pi x{:}S.\,T \Uparrow^\Gamma f & \mapsto \lambda x.\,(T \Uparrow^{\Gamma;x:S} f\ x) \\
\Sigma x{:}S.\,T \Uparrow^\Gamma p & \mapsto \langle S \Uparrow^\Gamma \mathsf{fst}\ p, T[\mathsf{fst}\ p] \Uparrow^\Gamma \mathsf{snd}\ p\rangle \\
\mathsf{W}x{:}S.\,T \Uparrow^\Gamma s \triangleleft f & \mapsto (S \Uparrow^\Gamma s) \triangleleft (T[s] \to \mathsf{W}x{:}S.\,T \Uparrow^\Gamma f) \\
\lceil N \rceil \Uparrow^\Gamma p & \mapsto \boxed{p} \\
T \Uparrow^\Gamma n & \mapsto n' \quad \text{if } \Downarrow^\Gamma n \mapsto n' : \_, \text{ otherwise}
\end{array}
$$

For neutral terms, we follow the typing rules

$$
\begin{array}{rl}
\Downarrow^{\Gamma;x:S;\Gamma'} x \mapsto x : S & \\
\Downarrow^\Gamma z!S \mapsto \boxed{z}!\Uparrow^\Gamma S : S & \\
\Downarrow^\Gamma \mathsf{if}\ b/x.P\ \mathsf{then}\ t\ \mathsf{else}\ f \mapsto & \\
\quad \mathsf{if}\ 2 \Uparrow^\Gamma b/x.\ \Uparrow^{\Gamma;x:2} P\ \mathsf{then}\ P[\mathsf{tt}] \Uparrow^\Gamma t\ \mathsf{else}\ P[\mathsf{ff}] \Uparrow^\Gamma f : P[b] & \\
\Downarrow^\Gamma f\ s \mapsto f'\ (S \Uparrow^\Gamma s) : T[s] & \\
\quad \text{if } \Downarrow^\Gamma f \mapsto f' : \Pi x{:}S.\,T & \\
\Downarrow^\Gamma \mathsf{fst}\ p \mapsto \mathsf{fst}\ p' : S & \\
\quad \text{if } \Downarrow^\Gamma p \mapsto p' : \Sigma x{:}S.\,T & \\
\Downarrow^\Gamma \mathsf{snd}\ p \mapsto \mathsf{snd}\ p' : T[\mathsf{fst}\ p] & \\
\quad \text{if } \Downarrow^\Gamma p \mapsto p' : \Sigma x{:}S.\,T & \\
\Downarrow^\Gamma \mathsf{rec}\ u/w.P\ \mathsf{with}\ p \mapsto \mathsf{rec}\ u'/w.\ \Uparrow^{\Gamma;u:\mathsf{W}x:S.\,T} P & \\
\quad \mathsf{with}\ \Pi \cdots \Uparrow^\Gamma p & \\
\quad \text{if } \Downarrow^\Gamma u \mapsto u' : \mathsf{W}x{:}S.\,T &
\end{array}
$$

where you can recover the missing type of the recursive method from the typing rule for rec with . However, the real excitement is in the quotation of neutral coercions.

$$\Downarrow^\Gamma s\,[Q{:}S{=}T\rangle \mapsto T \Uparrow^\Gamma s : T \quad \text{if } \Uparrow^\Gamma S \equiv_{\alpha\square} \Uparrow^\Gamma T$$
$$\mapsto (S \Uparrow^\Gamma s)\,[\boxed{Q}{:}(\Uparrow^\Gamma S){=}(\Uparrow^\Gamma T)\rangle : T \quad \text{otherwise}$$

This procedure eliminates stuck coercions between equal sets, solving our problem, but does it make sense? The crucial point is that if our coercion is stuck and the sets are equal, s must be neutral: if s were canonical, then S would be canonical and T would be equal to it, hence the coercion would *reduce*.

Quotation for sets is straightforward:

$$\Uparrow^\Gamma G \mapsto G \qquad \text{ground sets}$$
$$\Uparrow^\Gamma Bx{:}S.\,T \mapsto Bx{:}(\Uparrow^\Gamma S).\,(\Uparrow^{\Gamma;x:S} T) \qquad \text{binder sets}$$
$$\Uparrow^\Gamma N \mapsto \Downarrow^\Gamma N$$

$$\Downarrow^\Gamma \mathsf{If}\ b\ \mathsf{Then}\ T\ \mathsf{Else}\ F \mapsto \mathsf{If}\ (2 \Uparrow^\Gamma b)\ \mathsf{Then}\ (\Uparrow^\Gamma T)\ \mathsf{Else}\ (\Uparrow^\Gamma F)$$
$$\Downarrow^\Gamma \lceil S = T \rceil \mapsto \lceil (\Uparrow^\Gamma S) = (\Uparrow^\Gamma T) \rceil$$
$$\Downarrow^\Gamma \lceil (s{:}S) = (t{:}T) \rceil \mapsto \lceil (S \Uparrow^\Gamma s :\Uparrow^\Gamma S) = (T \Uparrow^\Gamma t :\Uparrow^\Gamma T) \rceil$$

What is remarkable about this is that we did not have to change *evaluation*, just our equivalence on values. In particular, our deletion of coercions happens only within the equational theory of neutral terms. Of course, we had better check that

LEMMA 6 (Coercion Elimination).

$$S \equiv T \textit{ implies } s\,[Q{:}S{=}T\rangle \equiv s : T$$

even when the coercion is not stuck.

PROOF By induction on the quoted normal forms of S and T. For ground sets, this is exactly the computational behaviour of coercion. For neutral sets, quotation erases the coercion. The interest lies in binder sets.

If $Bx{:}S_0.\,T_0 \equiv_{\alpha\Box} Bx{:}S_1.\,T_1$ then $S_0 \equiv_{\alpha\Box} S_1$ and $T_0 \equiv_{\alpha\Box} T_1$. Recall that in each case, the coercion operates by coercing a domain component $s\ [\mathsf{fst}\ Q{:}S_i = S_j\rangle$ with i, j being $0, 1$ or $1, 0$, according to variance. As $SS_0 \equiv_{\alpha\Box} S_1$, we have inductively that $s\ [\mathsf{fst}\ Q{:}S_i = S_j\rangle \equiv s$, and hence that $T_i[s\ [\mathsf{fst}\ Q{:}S_i{=}S_j\rangle] \equiv T_j[s]$ for either twisting. The latter guarantees inductively that the coercion of the range component disappears. Hence the action of coercion between equal sets is at worst η-expansion, which certainly falls within our new equivalence. □

The effect of this extension of definitional equality is thus to recover the lost computational behavour of intensional type theories. Substitution by reflexivity is equal to the identity, and the computation rules for Nat's induction principle hold definitionally.

7. Conclusions and Further Work

Where do we stand now, with Observational Type Theory? The basic system given in this paper has been coded in Agda 2, which gives some evidence for its computational properties and consistency. We have yet to execute our plan to code the construction in a less convenient but more standard system.

We hope to follow Oury's proof method to show that ETT is a conservative extension of OTT, and we can certainly validate the key axioms underpinning his translation of extensional derivations to intensional terms [22]. His work gives us good reason to conjecture that OTT has the full reasoning power of the corresponding extensional type theory, and hence that we really have no need to suffer the negative computational consequences of the equality reflection rule in order to obtain its logical benefits.

Moreover, as we have shown, we do not need to compromise any of the computational equalities of intensional type theory to achieve this result. Substituting by a reflexive equation is still the identity and induction principles compute as they should. Indeed, we have shown that we can add proof irrelevance to an intensional

type theory without serious modification to its evaluation process. In particular, we can now elaborate Epigram's pattern matching to OTT, retaining all the same computational behaviour, even on open terms.

For a realistic implementation, it is crucial to erase equality proofs and coercions from *run-time* objects. It is perfectly safe to do so, provided we never compute under a binder. This is nothing new: ETTs, where coercions are invisible, have always supported *weak* normalisation, and *program extraction* from ITTs has always supported the erasure of proofs [23].

Three key pieces of the jigsaw are still missing:

Hierarchical universes. We have illustrated the observational approach with a minimal type theory, excluding abstraction over sets. Clearly, we need to introduce a type hierarchy [17] which allows us to scale up. This certainly does not preclude the extension of the observational approach.

Coinductive data. As with functions, the useful notion of equality for coinductive data is observational in character: equal codata have equal one-step unfoldings. Of course, it is consistent to add the propositional law that a *bisimulation* induces equality on codata: as we have seen, this has no impact on the computational properties of the system.

Quotients. We should very much like to internalise setoids—sets of values up to a programmer-supplied equivalence—as quotient sets. We plan to represent quotients as abstract datatypes, allowing you access to the element of the underlying set only if you can prove that you respect the equivalence. Correspondingly, observational equality on quotient sets should just reduce to the given equivalence.

The potential applications of observational equality are considerable. There are many constructions and developments in the literature which have struggled to cope with the rigidity of intensional equality for functions. McBride's correctness proof for unification [18] is burdened throughout by the need to show explicitly that predicates respect the observational equality for substitutions represented functionally.

Moreover, we expect formalisations of categorical structure to be greatly simplified by the ability to use sets and observational equality rather than setoids with a hand-cranked equivalence. We hope that Buisse and Dybjer will be able to take advantage of OTT to streamline their recent study of categories-with-families, allowing us to model the mathematical structures underlying dependent type systems [6].

If we are to integrate real programming with proof, we need to be able to reason effectively about effects, higher-order objects, processes, abstract data types and the like. That means we need the monad laws to hold; that means we need to reason by observational congruence for processes; that means we need to exploit the equivalences preserved by encapsulation. This paper shows that Observational Type Theory can provide a computational foundation for dependently typed programming, integrated with a logic which steps up to that challenge.

Acknowledgments

The authors wish to express their sincere gratitude to Peter Hancock, for Agda driving lessons and wise insights. Nicolas Oury has made many helpful contributions to our work and is now the prime force behind the implementation of OTT in Epigram 2. We should also like to thank Thierry Coquand and Peter Dybjer for useful con-

versations, the anonymous referees for their helpful remarks, and Ulf Norell for his spectacular new version of Agda which made this story so much easier to tell.

References

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

[2] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[3] Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In *Logic in Computer Science*, 2007.

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

[5] Lennart Augustsson. Cayenne – a language with dependent types. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[6] Alexandre Buisse and Peter Dybjer. Formalizing categorical models of type theory in type theory. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2007.

[7] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[8] Thierry Coquand. Pattern matching with dependent types. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pages 71–84. Dept. of Computing Science,

Chalmers Univ. of Technology and Göteborg Univ., 1992.

[9] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Typed Lambda Calculi and Applications*, 1581:129–146, 1999.

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

[11] Conor McBride et al. Epigram, 2004. http://www.e-pig.org.

[12] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2006.

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

[14] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1995. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/95/ECS-LFCS-95-327/`.

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

[16] G. Huet and A. Saïbi. *Constructive Category Theory*. MIT Press, 1998.

[17] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[18] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

[19] Conor McBride. Elimination with a Motive. In Paul Callaghan,

Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277. Springer-Verlag, 2002.

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

[21] Ulf Norell. Agda 2. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.

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

[23] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Prog ramming Languages*, Austin, January 1989. ACM.

[24] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 2006.

[25] Tim Sheard. Putting Curry-Howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, 2005.

[26] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM Press.

[27] David A. Turner. A new formulation of constructive type theory. In Peter Dybjer et al., editor, *Proceedings of the Workshop on Programming Logic*. Programming Methodology Group, University of Gothenburg and Chalmers University of Technology, 1989.

[28] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[29] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.