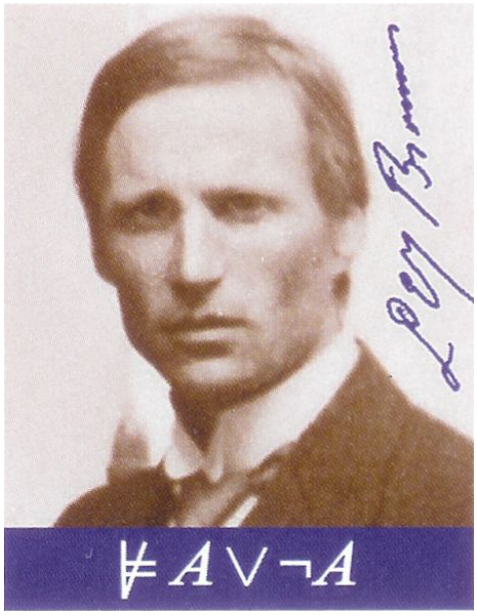


Final year project

Bertus: Implementing Observational Equality

Francesco Mazzoli <fm2209@ic.ac.uk>

Supervisor:
Dr. Steffen van Bakel

Second marker:
Dr. Philippa Gardner

July 1, 2013

Abstract

The marriage between programming and logic has been a fertile one. In particular, since the definition of the simply typed λ-calculus, a number of type systems have been devised with increasing expressive power.

Among this systems, Intuitionistic Type Theory (ITT) has been a popular framework for theorem provers and programming languages. However, reasoning about equality has always been a tricky business in ITT and related theories. In this thesis we shall explain why this is the case, and present Observational Type Theory (OTT), a solution to some of the problems with equality.

To bring OTT closer to the current practice of interactive theorem provers, we describe Bertus, a system featuring OTT in a setting more close to the one found in widely used provers such as Agda and Coq. Most notably, we feature user defined inductive and record types and a cumulative, implicit type hierarchy. Having implemented part of Bertus as a Haskell program, we describe some of the implementation issues faced.

Acknowledgements

I would like to thank Steffen van Bakel, my supervisor, who was brave enough to believe in my project and who provided support and invaluable advice.

I would also like to thank the Haskell and Agda community on `IRC`, which guided me through the strange world of types; and in particular Andrea Vezzosi and James Deikun, with whom I entertained countless insightful discussions over the past year. Andrea suggested Observational Type Theory as a topic of study: this thesis would not exist without him. Before them, Tony Field introduced me to Haskell, unknowingly filling most of my free time from that time on.

Finally, most of the work stems from the research of Conor McBride, who answered many of my doubts through these months. I also owe him the colours.

Contents

1 | Introduction

Functional programming is in good shape. In particular the 'well-typed' line of work originating from Milner's ML has been extremely fruitful, in various directions. Nowadays functional, well-typed programming languages like Haskell or OCaml are slowly being absorbed by the mainstream. An important related development—and in fact the original motivator for ML's existence—is the advancement of the practice of *interactive theorem provers*.

An interactive theorem prover, or proof assistant, is a tool that lets the user develop formal proofs with the confidence of the machine checking them for correctness. While the effort towards a full formalisation of mathematics has been ongoing for more than a century, theorem provers have been the first class of software whose implementation depends directly on these theories.

In a fortunate turn of events, it was discovered that well-typed functional programming and proving theorems in an *intuitionistic* logic are the same activity. Under this discipline, the types in our programming language can be interpreted as proposition in our logic; and the programs implementing the specification given by the types as their proofs. This fact stimulated an active transfer of techniques and knowledge between logic and programming language theory, in both directions.

Mathematics could provide programming with a wealth of abstractions and constructs developed over centuries. Moreover, identifying our types with a logic lets us focus on foundational questions regarding programming with a much more solid approach, given the years of rigorous study of logic. Programmers, on the other hand, had already developed a number of approaches to effectively collaborate with computers, through the study of programming languages.

In this space, we shall follow the discipline of Intuitionistic Type Theory, or Martin-Löf Type Theory, after its inventor. First formulated in the 70s and then adjusted through a series of revisions, it has endured as the core of many practical systems in wide use today, and it is the most prominent instance of the proposition-as-types and proofs-as-programs paradigm. One of the most debated subjects in this field has been regarding what notion of equality should be exposed to the user.

The tension when studying equality in type theory springs from the fact that there is a divide between what the user can prove equal *inside* the theory—what is *propositionally* equal—and what the theorem prover identifies as equal in its meta-theory—what is *definitionally* equal. If we want our system to be well behaved (mostly if we want to keep type checking decidable) we must keep the two notions separate, with definitional equality inducing propositional equality, but not the reverse. However in this scenario propositional equality is weaker than we would like: we can only prove terms equal based on their syntactical structure, and not based on their behaviour.

This thesis is concerned with exploring a new approach in this area, *observational* equality. Promising to provide a more adequate propositional equality while retaining well-behavedness, it still is a relatively unexplored notion. We set ourselves to change that by studying it in a setting more akin to the one found in currently available theorem provers.

1.1 Structure

Section 2 will give a brief overview of the λ-calculus, both typed and untyped. This will give us the chance to introduce most of the concepts mentioned above rigorously, and gain some intuition about them. An excellent introduction to types in general can be found in Pierce (2002), although not from the perspective of theorem proving.

Section 3 will describe a set of basic construct that form a 'baseline' Intuitionistic Type

Theory. The goal is to familiarise with the main concept of ITT before attacking the problem of equality. Given the wealth of material covered the exposition is quite dense. Good introductions can be found in Thompson (1991), Nordström *et al.* (1990), and Martin-Löf (1984) himself.

Section 4 will introduce propositional equality. The properties of propositional equality will be discussed along with its limitations. After reviewing some extensions, we will explain why identifying definitional equality with propositional equality causes problems.

Section 5 will introduce observational equality, following closely the original exposition by Altenkirch *et al.* (2007). The presentation is free-standing but glosses over the meta-theoretic properties of OTT, focusing on the mechanisms that make it work.

Section 6 is the central part of the thesis and will describe Bertus, a system we have developed incorporating OTT along constructs usually present in modern theorem provers. Along the way, we discuss these additional features and their trade-offs. Section 7 will describe an implementation implementing part of Bertus. A high level design of the software is given, along with a few specific implementation issues.

Finally, Section 8 will asses the decisions made in designing and implementing Bertusand the results achieved; and Section 9 will give a roadmap to bring Bertus on par and beyond the competition.

1.2 Contributions

The contribution of this thesis is threefold:

- Provide a description of observational equality 'in context', to make the subject more accessible. Considering the possibilities that OTT brings to the table, we think that introducing it to a wider audience can only be beneficial.
- Fill in the gaps needed to make OTT work with user-defined inductive types and a type hierarchy. We show how one notion of equality is enough, instead of separate notions of value- and type-equality as presented in the original paper. We are able to keep the type equalities 'small' while preserving subject reduction by exploiting the fact that we work within a cumulative theory. Incidentally, we also describe a generalised version of bidirectional type checking for user defined types.
- Provide an implementation to probe the possibilities of OTT in a more realistic setting. We have implemented an ITT with user defined types but due to the limited time constraints we were not able to complete the implementation of observational equality. Nonetheless, we describe some interesting implementation issues faced by the type theory implementor.

The system developed as part of this thesis, Bertus, incorporates OTT with features that are familiar to users of existing theorem provers adopting the proofs-as-programs mantra. The defining features of Bertus are:

Full dependent types In ITT, types are a very 'first class' notion and can be the result of computation—they can *depend* on values, thus the name *dependent types*. Bertus espouses this notion to its full consequences.

User defined data types and records Instead of forcing the user to choose from a restricted toolbox, we let her define types for greater flexibility. We have two kinds of user defined types: inductive data types, formed by various data constructors whose type signatures can contain recursive occurrences of the type being defined; and records, where we have just one data constructor, and projections to extract each each field in said constructor.

Consistency Our system is meant to be consistent with respect to the logic it embodies. For this reason, we restrict recursion to *structural* recursion on the defined inductive types, through the use of operators (destructors) computing on each type. Following the types-as-propositions interpretation, each destructor expresses an induction principle on the data type it operates on. To achieve the consistency of these operations we make sure that our recursive data types are *strictly positive*.

Bidirectional type checking We take advantage of a *bidirectional* type inference system in the style of Pierce & Turner (2000). This cuts down the type annotations by a considerable amount in an elegant way and at a very low cost. Bidirectional type checking is usually employed in core calculi, but in Bertus we extend the concept to user defined types.

Type hierarchy In set theory we have to take powerset-like objects with care, if we want to avoid paradoxes. However, the working mathematician is rarely concerned by this, and the consistency in this regard is implicitly assumed. In the tradition of Whitehead & Russell (1927), in Bertus we employ a *type hierarchy* to make sure that these size issues are taken care of; and we employ system so that the user will be free from thinking about the hierarchy, just like the mathematician is.

Observational equality The motivator of this thesis, Bertus incorporates a notion of observational equality, modifying the original presentation by Altenkirch *et al.* (2007) to fit our more expressive system. As mentioned, we reconcile OTT with user defined types and a type hierarchy.

Type holes When building up programs interactively, it is useful to leave parts unfinished while exploring the current context. This is what type holes are for.

1.3 Notation and syntax

Appendix A describes the notation and syntax used in this thesis.

2 | Simple and not-so-simple types

Well typed programs can't go wrong.

Robin Milner

2.1 The untyped λ-calculus

Along with Turing's machines, the earliest attempts to formalise computation lead to the definition of the λ-calculus (Church, 1936). This early programming language encodes computation with a minimal syntax and no 'data' in the traditional sense, but just functions. Here we give a brief overview of the language, which will give the chance to introduce concepts central to the analysis of all the following calculi. The exposition follows the one found in Chapter 5 of Queinnec (2003).

Definition (λ-terms)**.** *Syntax of the λ-calculus: variables, abstractions, and applications.*

syntax

$$\begin{aligned} term &::= x \mid \lambda x \mapsto term \mid (term\ term) \\ x &\in \text{Some enumerable set of symbols} \end{aligned}$$

Parenthesis will be omitted in the usual way, with application being left associative.

Abstractions roughly corresponds to functions, and their semantics is more formally explained by the β-reduction rule.

Definition (β-reduction)**.** *β-reduction and substitution for the λ-calculus.*

reduction: $term \rightsquigarrow term$

$$\begin{aligned} &(\lambda x \mapsto m)\ n \rightsquigarrow m[n/x] \textbf{ where} \\ &\quad y[n/x] \mid x = y \implies n \\ &\quad y[n/x] \implies y \\ &\quad (t\ m)[n/x] \implies (t[n/x]\ m[n/x]) \\ &\quad (\lambda x \mapsto m)[n/x] \implies \lambda x \mapsto m \\ &\quad (\lambda y \mapsto m)[n/x] \implies \lambda z \mapsto m[z/y][n/x] \\ &\qquad \textbf{with } x \neq y \text{ and } z \text{ not free in } m\ n \end{aligned}$$

The care required during substituting variables for terms is to avoid name capturing. We will use substitution in the future for other name-binding constructs assuming similar precautions.

These few elements have a remarkable expressiveness, and are in fact Turing complete. As a corollary, we must be able to devise a term that reduces forever ('loops' in imperative terms):

$$(\omega\ \omega) \rightsquigarrow (\omega\ \omega) \rightsquigarrow \cdots, \textbf{ where } \omega = \lambda x \mapsto x\ x$$

Definition (redex)**.** *A* redex *is a term that can be reduced.*

In the untyped λ-calculus this will be the case for an application in which the first term is an abstraction, but in general we call a term reducible if it appears to the left of a reduction rule.

Definition (normal form)**.** *A term that contains no redexes is said to be in* normal form*.*

syntax

$$
\begin{array}{rcl}
\mathit{term} & ::= & x \mid \lambda x{:}\mathit{type} \mapsto \mathit{term} \mid (\mathit{term}\ \mathit{term}) \\
\mathit{type} & ::= & \phi \mid \mathit{type} \to \mathit{type} \mid \\
x & \in & \text{Some enumerable set of symbols} \\
\phi & \in & \Phi
\end{array}
$$

typing: $\Gamma \vdash \mathit{term} : \mathit{type}$

$$
\frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma; x : A \vdash t : B}{\Gamma \vdash \lambda x{:}A \mapsto t : B} \qquad \frac{\Gamma \vdash m : A \to B \qquad \Gamma \vdash n : A}{\Gamma \vdash m\ n : B}
$$

Figure 1: Syntax and typing rules for the STLC. Reduction is unchanged from the untyped λ-calculus.

Definition (normalising terms and systems)**.** *Terms that reduce in a finite number of reduction steps to a normal form are* normalising*. A system in which all terms are normalising is said to have the* normalisation property*, or to be* normalising*.*

Given the reduction behaviour of $(\omega\,\omega)$, it is clear that the untyped λ-calculus does not have the normalisation property.

We have not presented reduction in an algorithmic way, but *evaluation strategies* can be employed to reduce term systematically. Common evaluation strategies include *call by value* (or *strict*), where arguments of abstractions are reduced before being applied to the abstraction; and conversely *call by name* (or *lazy*), where we reduce only when we need to do so to proceed—in other words when we have an application where the function is still not a λ. In both these strategies we never reduce under an abstraction. For this reason a weaker form of normalisation is used, where all abstractions are said to be in *weak head normal form* even if their body is not.

2.2 The simply typed λ-calculus

A convenient way to 'discipline' and reason about λ-terms is to assign *types* to them, and then check that the terms that we are forming make sense given our typing rules (Curry, 1934). The first most basic instance of this idea takes the name of *simply typed λ-calculus* (STLC).

Definition (Simply typed λ-calculus)**.** *The syntax and typing rules for the STLC are given in Figure 1.*

Our types contain a set of *type variables* Φ, which might correspond to some 'primitive' types; and $\to$, the type former for 'arrow' types, the types of functions. The language is explicitly typed: when we bring a variable into scope with an abstraction, we declare its type. Reduction is unchanged from the untyped λ-calculus.

In the typing rules, a context Γ is used to store the types of bound variables: ε is the empty context, and $\Gamma; x : A$ adds a variable to the context. $\Gamma(x)$ extracts the type of the rightmost occurrence of x.

This typing system takes the name of 'simply typed lambda calculus' (STLC), and enjoys a number of properties. Two of them are expected in most type systems (Pierce, 2002):

Definition (Progress)**.** *A well-typed term is not stuck—it is either a variable, or it does not appear on the left of the $\rightsquigarrow$ relation , or it can take a step according to the evaluation rules.*

Definition (Subject reduction). *If a well-typed term takes a step of evaluation, then the resulting term is also well-typed, and preserves the previous type.*

However, STLC buys us much more: every well-typed term is normalising (Tait, 1967). It is easy to see that we cannot fill the blanks if we want to give types to the non-normalising term shown before:

$$(\lambda x{:}\,?\mapsto x\,x)\,(\lambda x{:}\,?\mapsto x\,x)$$

This makes the STLC Turing incomplete. We can recover the ability to loop by adding a combinator that recurses:

Definition (Fixed-point combinator).

syntax

$$term ::= \cdots b \mid \underline{\text{fix}}\, x{:}type \mapsto term$$

typing: $\Gamma \vdash term : type$

$$\frac{\Gamma; x : A \vdash t : A}{\Gamma \vdash \underline{\text{fix}}\, x{:}A \mapsto t : A}$$

reduction: $\Gamma \vdash term : term$

$$\underline{\text{fix}}\, x{:}A \mapsto t \rightsquigarrow t[(\underline{\text{fix}}\, x{:}A \mapsto t)/x]$$

fix will deprive us of normalisation, which is a particularly bad thing if we want to use the STLC as described in the next section.

Another important property of the STLC is the Church-Rosser property:

Definition (Church-Rosser property). *A system is said to have the* Church-Rosser *property, or to be* confluent, *if given any two reductions m and n of a given term t, there is exist a term to which both m and n can be reduced.*

Given that the STLC has the normalisation property and the Church-Rosser property, each term has a *unique* normal form.

2.3 The Curry-Howard correspondence

As hinted in the introduction, it turns out that the STLC can be seen a natural deduction system for intuitionistic propositional logic. Terms correspond to proofs, and their types correspond to the propositions they prove. This remarkable fact is known as the Curry-Howard correspondence, or isomorphism.

The arrow ($\to$) type corresponds to implication. If we wish to prove that that $(A \to B) \to (B \to C) \to (A \to C)$, all we need to do is to devise a λ-term that has the correct type:

$$\lambda f{:}(A \to B) \mapsto \lambda g{:}(B \to C) \mapsto \lambda x{:}A \mapsto g\,(f\,x)$$

Which is known to functional programmers as function composition. Going beyond arrow types, we can extend our bare lambda calculus with useful types to represent other logical constructs.

Definition (The extended STLC). *Figure 2 shows syntax, reduction, and typing rules for the* extended simply typed λ-calculus.

Tagged unions (or sums, or coproducts—$+$ here, `Either` in Haskell) correspond to disjunctions, and dually tuples (or pairs, or products—$\times$ here, tuples in Haskell) correspond to conjunctions. This is apparent looking at the ways to construct and destruct the values

syntax

$$
\begin{aligned}
\mathit{term} ::= {} & \cdots \\
& \mid \langle\rangle \mid \mathbf{absurd}_{\mathit{type}}\, \mathit{term} \\
& \mid \mathsf{left}_{\mathit{type}}\, \mathit{term} \mid \mathsf{right}_{\mathit{type}}\, \mathit{term} \mid [\mathit{term}, \mathit{term}]\, \mathit{term} \\
& \mid \langle \mathit{term}, \mathit{term} \rangle \mid \mathbf{fst}\, \mathit{term} \mid \mathbf{snd}\, \mathit{term} \\
\mathit{type} ::= {} & \cdots \mid \mathsf{Unit} \mid \mathsf{Empty} \mid \mathit{term} + \mathit{term} \mid \mathit{type} \times \mathit{type}
\end{aligned}
$$

reduction: $\mathit{term} \rightsquigarrow \mathit{term}$

$$
\begin{aligned}
[m, n]\, (\mathsf{left}_A \;\; t) &\rightsquigarrow m\, t & \quad \mathbf{fst}\;\; \langle m, n \rangle &\rightsquigarrow m \\
[m, n]\, (\mathsf{right}_A\, t) &\rightsquigarrow n\, t & \quad \mathbf{snd}\, \langle m, n \rangle &\rightsquigarrow n
\end{aligned}
$$

typing: $\Gamma \vdash \mathit{term} : \mathit{type}$

$$
\frac{}{\Gamma \vdash \langle\rangle : \mathsf{Unit}} \qquad \frac{\Gamma \vdash t : \mathsf{Empty}}{\Gamma \vdash \mathbf{absurd}_A\, t : A}
$$

$$
\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{left}_B\, t : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{right}_A\, t : A + B}
$$

$$
\frac{\Gamma \vdash m : A \to B \qquad \Gamma \vdash n : A \to C \qquad \Gamma \vdash t : A + B}{\Gamma \vdash [m, n]\, t : C}
$$

$$
\frac{\Gamma \vdash m : A \qquad \Gamma \vdash n : B}{\Gamma \vdash \langle m, n \rangle : A \times B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathbf{fst}\, t : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathbf{snd}\, t : B}
$$

Figure 2: Rules for the extended STLC. Only the new features are shown, all the rules and syntax for the STLC apply here too.

inhabiting those types: for $+$ left and right correspond to $\vee$ introduction, and $[\_, \_]$ to $\vee$ elimination; for $\times$ $\langle \_, \_ \rangle$ corresponds to $\wedge$ introduction, **fst** and **snd** to $\wedge$ elimination.

The trivial type Unit corresponds to the logical $\top$ (true), and dually Empty corresponds to the logical $\bot$ (false). Unit has one introduction rule ($\langle\rangle$), and thus one inhabitant; and no eliminators—we cannot gain any information from a witness of the single member of Unit. Empty has no introduction rules, and thus no inhabitants; and one eliminator (**absurd**), corresponding to the logical *ex falso quodlibet*.

With these rules, our STLC now looks remarkably similar in power and use to the natural deduction we already know.

Definition (Negation). *$\neg A$ can be expressed as $A \to \mathsf{Empty}$.*

However, there is an important omission: there is no term of the type $A + \neg A$ (excluded middle), or equivalently $\neg\neg A \to A$ (double negation), or indeed any term with a type equivalent to those.

This has a considerable effect on our logic and it is no coincidence, since there is no obvious computational behaviour for laws like the excluded middle. Logics of this kind are called *intuitionistic*, or *constructive*, and all the systems analysed will have this characteristic since they build on the foundation of the STLC.[1]

As in logic, if we want to keep our system consistent, we must make sure that no closed terms (in other words terms not under a λ) inhabit Empty. The variant of STLC presented

[1] There is research to give computational behaviour to classical logic, but I will not touch those subjects.

here is indeed consistent, a result that follows from the fact that it is normalising. Going back to our $\underline{\text{fix}}$ combinator, it is easy to see how it ruins our desire for consistency. The following term works for every type A, including bottom:

$$(\underline{\text{fix}}\, x{:}A \mapsto x) : A$$

2.4 Inductive data

To make the STLC more useful as a programming language or reasoning tool it is common to include (or let the user define) inductive data types. These comprise of a type former, various constructors, and an eliminator (or destructor) that serves as primitive recursor.

Definition (Finite lists for the STLC)**.** *We add a* List *type constructor, along with an 'empty list' ([]) and 'cons cell' (::) constructor. The eliminator for lists will be the usual folding operation (***foldr***). Full rules in Figure 3.*

syntax

$$\begin{array}{rcl} \mathit{term} & ::= & \cdots \mid []_{\mathit{type}} \mid \mathit{term} :: \mathit{term} \mid \mathbf{foldr}\ \mathit{term}\ \mathit{term}\ \mathit{term} \\ \mathit{type} & ::= & \cdots \mid \mathsf{List}\ \mathit{type} \end{array}$$

reduction: $\mathit{term} \rightsquigarrow \mathit{term}$

$$\begin{array}{ll} \mathbf{foldr}\ f\ t\ []_A & \rightsquigarrow t \\ \mathbf{foldr}\ f\ t\ (m :: n) & \rightsquigarrow f\ m\ (\mathbf{foldr}\ f\ t\ n) \end{array}$$

typing: $\Gamma \vdash \mathit{term} : \mathit{type}$

$$\frac{}{\Gamma \vdash []_A : \mathsf{List}\ A} \qquad \frac{\Gamma \vdash m : A \qquad \Gamma \vdash n : \mathsf{List}\ A}{\Gamma \vdash m :: n : \mathsf{List}\ A}$$

$$\frac{\Gamma \vdash f : A \to B \to B \qquad \Gamma \vdash m : B \qquad \Gamma \vdash n : \mathsf{List}\ A}{\Gamma \vdash \mathbf{foldr}\ f\ m\ n : B}$$

Figure 3: Rules for lists in the STLC.

In Section 3.3.7 we will see how to give a general account of inductive data.

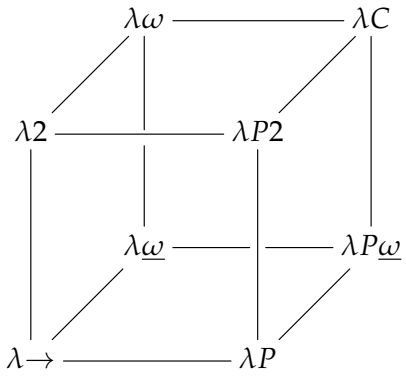
3 | Intuitionistic Type Theory

Martin-Löf's type theory is a well established and convenient arena in which computational Christians are regularly fed to logical lions.

Conor McBride

3.1 Extending the STLC

Barendregt (1991) succinctly expressed geometrically how we can add expressively to the STLC:



Here $\lambda\!\!\rightarrow$, in the bottom left, is the STLC. From there can move along 3 dimensions:

Terms depending on types (towards $\lambda 2$) We can quantify over types in our type signatures. For example, we can define a polymorphic identity function, where Type denotes the 'type of types':

$$(\lambda A{:}\mathsf{Type} \mapsto \lambda x{:}A \mapsto x) : (A{:}\mathsf{Type}) \to A \to A$$

The first and most famous instance of this idea has been System F. This form of polymorphism and has been wildly successful, also thanks to a well known inference algorithm for a restricted version of System F known as Hindley-Milner (Milner, 1978). Languages like Haskell and SML are based on this discipline. In Haskell the above example would be

```
id :: a -> a
id x = x
```

Where `a` implicitly quantifies over a type, and will be instantiated automatically when `id` is used thanks to the type inference.

Types depending on types (towards $\lambda\underline{\omega}$) We have type operators. For example we could define a function that given types R and A forms the type that represents a value of type A in continuation passing style:

$$(\lambda R\ A{:}\mathsf{Type} \mapsto (A \to R) \to R) : \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type}$$

In Haskell we can define type operator of sorts, although we must pair them with data constructors, to keep inference manageable:

```
newtype Cont r a = Cont ((a -> r) -> r)
```

Where the 'type' (kind in Haskell parlance) of `Cont` will be `* -> * -> *`, with `*` signifying the type of types.

Types depending on terms (towards λP) Also known as 'dependent types', give great expressive power. For example, we can have values of whose type depend on a boolean:

$$(\lambda x{:}\mathsf{Bool} \mapsto \textbf{if } x \textbf{ then } \mathbb{N} \textbf{ else } \mathbb{R}) : \mathsf{Bool} \to \mathsf{Type}$$

We cannot give an Haskell example that expresses this concept since Haskell does not support dependent types—it would be a very different language if it did.

All the systems placed on the cube preserve the properties that make the STLC well behaved. The one we are going to focus on, Intuitionistic Type Theory, has all of the above additions, and thus would sit where λC sits. It will serve as the logical 'core' of all the other extensions that we will present and ultimately our implementation of a similar logic.

3.2 A Bit of History

Logic frameworks and programming languages based on type theory have a long history. Per Martin-Löf described the first version of his theory in 1971, but then revised it since the original version was inconsistent due to its impredicativity.[2] For this reason he later gave a revised and consistent definition (Martin-Löf, 1984).

A related development is the polymorphic λ-calculus, and specifically the previously mentioned System F, which was developed independently by Girard and Reynolds. An overview can be found in (Reynolds, 1994). The surprising fact is that while System F is impredicative it is still consistent and strongly normalising. Coquand & Huet (1986) further extended this line of work with the Calculus of Constructions (CoC).

Most widely used interactive theorem provers are based on ITT. Popular ones include Agda (Norell, 2007), Coq (The Coq Team, 2013), Epigram (McBride & McKinna, 2004a; McBride, 2004), Isabelle (Paulson, 1990), and many others.

3.3 A simple type theory

The calculus I present follows the exposition in Thompson (1991), and is quite close to the original formulation of Martin-Löf (1984). Agda and Bertus renditions of the presented theory and all the examples (even the ones presented only as type signatures) are reproduced in Appendix B.1.

Definition (Intuitionistic Type Theory (ITT))**.** *The syntax and reduction rules are shown in Figure 4. The typing rules are presented piece by piece in the following sections.*

3.3.1 Types are terms, some terms are types

typing: $\Gamma \vdash term : term$

$$\frac{\Gamma \vdash t : A \qquad A \cong B}{\Gamma \vdash t : B} \qquad \frac{}{\Gamma \vdash \mathsf{Type}_l : \mathsf{Type}_{l+1}}$$

The first thing to notice is that the barrier between values and types that we had in the STLC is gone: values can appear in types, and the two are treated uniformly in the syntax.

While the usefulness of doing this will become clear soon, a consequence is that since types can be the result of computation, deciding type equality is not immediate as in the STLC.

[2] In the early version there was only one universe Type and Type : Type; see Section 3.3.1 for an explanation on why this causes problems.

syntax

$$
\begin{aligned}
\mathit{term} ::= {} & x \mid \mathsf{Type}_{\mathit{level}} \mid \mathsf{Unit} \mid \langle\rangle \mid \mathsf{Empty} \mid \mathbf{absurd}_{\mathit{term}}\ \mathit{term} \\
\mid {} & \mathsf{Bool} \mid \mathsf{true} \mid \mathsf{false} \mid \mathbf{if}\ \mathit{term}/x.\mathit{term}\ \mathbf{then}\ \mathit{term}\ \mathbf{else}\ \mathit{term} \\
\mid {} & (x{:}\mathit{term}) \to \mathit{term} \mid \lambda x{:}\mathit{term} \mapsto \mathit{term} \mid (\mathit{term}\ \mathit{term}) \\
\mid {} & (x{:}\mathit{term}) \times \mathit{term} \mid \langle \mathit{term}, \mathit{term} \rangle_{x.\mathit{term}} \\
\mid {} & \mathbf{fst}\ \mathit{term} \mid \mathbf{snd}\ \mathit{term} \\
\mid {} & \mathsf{W}\ (x{:}\mathit{term})\ \mathit{term} \mid \mathit{term} \lhd_{x.\mathit{term}} \mathit{term} \\
\mid {} & \mathbf{rec}\ \mathit{term}/x.\mathit{term}\ \mathbf{with}\ \mathit{term} \\
\mathit{level} \in {} & \mathbb{N}
\end{aligned}
$$

reduction: $\mathit{term} \rightsquigarrow \mathit{term}$

$$
\begin{array}{lll}
\mathbf{if}\ \mathsf{true}\ /x.P\ \mathbf{then}\ m\ \mathbf{else}\ n \rightsquigarrow m & (\lambda x{:}A \mapsto m)\ n \rightsquigarrow m[n/x] & \mathbf{fst}\ \langle m, n \rangle \rightsquigarrow m \\
\mathbf{if}\ \mathsf{false}/x.P\ \mathbf{then}\ m\ \mathbf{else}\ n \rightsquigarrow n & & \mathbf{snd}\ \langle m, n \rangle \rightsquigarrow n
\end{array}
$$

$$
\mathbf{rec}\ (s \lhd_{x.T} f)/y.P\ \mathbf{with}\ p \rightsquigarrow p\ s\ f\ (\lambda t{:}T[s/x] \mapsto \mathbf{rec}\ f\ t/y.P\ \mathbf{with}\ t)
$$

Figure 4: Syntax and reduction rules for our type theory.

Definition (Definitional equality)**.** *We define* definitional equality, $\cong$, *as the congruence relation extending* $\rightsquigarrow$. *Moreover, when comparing terms syntactically we do it up to renaming of bound names (*α*-renaming).*

For example under this discipline we will find that

$$
\begin{aligned}
&\lambda x{:}A \mapsto x \cong \lambda y{:}A \mapsto y \\
&(\lambda f{:}A \to A \mapsto f)\ (\lambda y{:}A \mapsto y) \cong \lambda quux{:}A \mapsto quux
\end{aligned}
$$

Types that are definitionally equal can be used interchangeably. Here the 'conversion' rule is not syntax directed, but it is possible to employ $\rightsquigarrow$ to decide term equality in a systematic way, comparing terms by reducing them to their unique normal forms first; so that a separate conversion rule is not needed. Another thing to notice is that, considering the need to reduce terms to decide equality, for type checking to be decidable a dependently typed must be terminating and confluent; since every type needs to have a unique normal form for definitional equality to be decidable.

Moreover, we specify a *type hierarchy* to talk about 'large' types: Type_0 will be the type of types inhabited by data: Bool, $\mathbb{N}$, List, etc. Type_1 will be the type of Type_0, and so on—for example we have $\mathsf{true} : \mathsf{Bool} : \mathsf{Type}_0 : \mathsf{Type}_1 : \cdots$. Each type 'level' is often called a universe in the literature. While it is possible to simplify things by having only one universe Type with $\mathsf{Type} : \mathsf{Type}$, this plan is inconsistent for much the same reason that impredicative naïve set theory is (Hurkens, 1995). However various techniques can be employed to lift the burden of explicitly handling universes, as we will see in Section 6.4.

3.3.2 Contexts

context validity: $\Gamma \vdash \underline{\text{valid}}$

$$
\frac{}{\varepsilon \vdash \underline{\text{valid}}} \qquad \frac{\Gamma \vdash A : \mathsf{Type}_l}{\Gamma; x : A \vdash \underline{\text{valid}}}
$$

typing: $\Gamma \vdash \mathit{term} : \mathit{term}$

$$
\frac{\Gamma(x) = A}{\Gamma \vdash x : A}
$$

We need to refine the notion of context to make sure that every variable appearing is typed correctly, or that in other words each type appearing in the context is indeed a type and not a value. In every other rule, if no premises are present, we assume the context in the conclusion to be valid.

Then we can re-introduce the old rule to get the type of a variable for a context.

3.3.3 Unit, Empty

typing: $\Gamma \vdash term : term$

$$\frac{}{\Gamma \vdash \mathsf{Unit} : \mathsf{Type}_0} \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathsf{Unit}} \qquad \frac{\Gamma \vdash t : \mathsf{Empty} \qquad \Gamma \vdash A : \mathsf{Type}_l}{\Gamma \vdash \mathbf{absurd}_A\, t : A}$$

$$\Gamma \vdash \mathsf{Empty} : \mathsf{Type}_0$$

Nothing surprising here: Unit and Empty are unchanged from the STLC, with the added rules to type Unit and Empty themselves, and to make sure that we are invoking **absurd** over a type.

3.3.4 Bool, and dependent if

typing: $\Gamma \vdash term : term$

$$\frac{}{\Gamma \vdash \mathsf{Bool} : \mathsf{Type}_0} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash t : \mathsf{Bool} \qquad \Gamma : \mathsf{Bool} \vdash A : \mathsf{Type}_l \qquad \Gamma \vdash m : A[\mathsf{true}/x] \qquad \Gamma \vdash n : A[\mathsf{false}/x]}{\Gamma \vdash \mathbf{if}\, t/x.A\, \mathbf{then}\, m\, \mathbf{else}\, n : A[t/x]}$$

With booleans we get the first taste of the 'dependent' in 'dependent types'. While the two introduction rules for true and false are not surprising, the rule for **if** is. In most strongly typed languages we expect the branches of an **if** statements to be of the same type, to preserve subject reduction, since execution could take both paths. This is a pity, since the type system does not reflect the fact that in each branch we gain knowledge on the term we are branching on. Which means that programs along the lines of

```
if null xs then head xs else 0
```

are a necessary, well-typed, danger.

However, in a more expressive system, we can do better: the branches' type can depend on the value of the scrutinised boolean. This is what the typing rule expresses: the user provides a type A ranging over an x representing the boolean we are operating the **if** switch with, and each branch is type checked against A with the updated knowledge of the value of x.

3.3.5 $\to$, or dependent function

typing: $\Gamma \vdash term : term$

$$\frac{\Gamma \vdash A : \mathsf{Type}_{l_1} \qquad \Gamma; x : A \vdash B : \mathsf{Type}_{l_2}}{\Gamma \vdash (x{:}A) \to B : \mathsf{Type}_{l_1 \sqcup l_2}}$$

$$\frac{\Gamma; x : A \vdash t : B}{\Gamma \vdash \lambda x{:}A \mapsto t : (x{:}A) \to B} \qquad \frac{\Gamma \vdash m : (x{:}A) \to B \qquad \Gamma \vdash n : A}{\Gamma \vdash m\, n : B[n/x]}$$

Dependent functions are one of the two key features that characterise dependent types—the other being dependent products. With dependent functions, the result type can depend on the value of the argument. This feature, together with the fact that the result type might be a type itself, brings a lot of interesting possibilities. In the introduction rule, the return type is type checked in a context with an abstracted variable of domain's type; and in the elimination rule the actual argument is substituted in the return type. Keeping the correspondence with logic alive, dependent functions are much like universal quantifiers ($\forall$) in logic.

For example, assuming that we have lists and natural numbers in our language, using dependent functions we can write functions of types

$$\begin{aligned}
&\textbf{length} : (A{:}\mathsf{Type}_0) \to \mathsf{List}\ A \to \mathbb{N} \\
&_>_ : \mathbb{N} \to \mathbb{N} \to \mathsf{Type}_0 \\
&\textbf{head} : (A{:}\mathsf{Type}_0) \to (l{:}\mathsf{List}\ A) \to \textbf{length}\ A\ l > 0 \to A
\end{aligned}$$

length is the usual polymorphic length function. _>_ is a function that takes two naturals and returns a type: if the lhs is greater then the rhs, Unit is returned, Empty otherwise. This way, we can express a 'non-emptiness' condition in **head**, by including a proof that the length of the list argument is non-zero. This allows us to rule out the empty list case by invoking **absurd** in **length**, so that we can safely return the first element.

Finally, we need to make sure that the type hierarchy is respected, which is the reason why a type formed by $\to$ will live in the least upper bound of the levels of argument and return type.

3.3.6 $\times$, or dependent product

typing: $\Gamma \vdash term : term$

$$\frac{\Gamma \vdash A : \mathsf{Type}_{l_1} \qquad \Gamma; x : A \vdash B : \mathsf{Type}_{l_2}}{\Gamma \vdash (x{:}A) \times B : \mathsf{Type}_{l_1 \sqcup l_2}}$$

$$\frac{\Gamma \vdash m : A \qquad \Gamma \vdash n : B[m/x]}{\Gamma \vdash \langle m, n \rangle_{x.B} : (x{:}A) \times B} \qquad \frac{\Gamma \vdash t : (x{:}A) \times B}{\begin{array}{l}\Gamma \vdash \textbf{fst}\ t : A \\ \Gamma \vdash \textbf{snd}\ t : B[\textbf{fst}\ t/x]\end{array}}$$

If dependent functions are a generalisation of $\to$ in the STLC, dependent products are a generalisation of $\times$ in the STLC. The improvement is that the second element's type can depend on the value of the first element. The correspondence with logic is through the existential quantifier: $\exists x \in \mathbb{N}.even(x)$ can be expressed as $(x{:}\mathbb{N}) \times \textbf{even}\ x$. The first element will be a number, and the second evidence that the number is even. This highlights the fact that we are working in a constructive logic: if we have an existence proof, we can always ask for a witness. This means, for instance, that $\neg\forall\neg$ is not equivalent to $\exists$.

Additionally, we need to specify the type of the second element (ranging over the first element) explicitly when using $\langle \_, \_ \rangle$.

Another perhaps more 'dependent' application of products, paired with Bool, is to offer choice between different types. For example we can easily recover disjunctions:

$$\begin{aligned}
&_\vee_ : \mathsf{Type}_0 \to \mathsf{Type}_0 \to \mathsf{Type}_0 \\
&A \vee B \mapsto (x{:}\mathsf{Bool}) \times \mathbf{if}\ x\ \mathbf{then}\ A\ \mathbf{else}\ B
\end{aligned}$$

$$\begin{aligned}
&\mathbf{case} : (A\ B\ C{:}\mathsf{Type}_0) \to (A \to C) \to (B \to C) \to A \vee B \to C \\
&\mathbf{case}\ A\ B\ C\ f\ g\ x \mapsto \\
&\quad (\mathbf{if}\ \mathbf{fst}\ x\ /b.(\mathbf{if}\ b\ \mathbf{then}\ A\ \mathbf{else}\ B)\ \mathbf{then}\ f\ \mathbf{else}\ g)\ (\mathbf{snd}\ x)
\end{aligned}$$

3.3.7 W, or well-order

typing: $\Gamma \vdash term : term$

$$\frac{\Gamma \vdash A : \mathsf{Type}_{l_1} \qquad \Gamma; x : A \vdash B : \mathsf{Type}_{l_2}}{\Gamma \vdash \mathsf{W}\ (x{:}A)\ B : \mathsf{Type}_{l_1 \sqcup l_2}} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash f : B[t/x] \to \mathsf{W}\ (x{:}A)\ B}{\Gamma \vdash t \lhd_{x.B} f : \mathsf{W}\ (x{:}A)\ B}$$

$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{W}\ (x{:}S)\ T \qquad \Gamma; w : \mathsf{W}\ (x{:}S)\ T \vdash P : \mathsf{Type}_l \\ \Gamma \vdash p : (s{:}S) \to (f{:}T[s/x] \to \mathsf{W}\ (x{:}S)\ T) \to ((t{:}T[s/x]) \to P[f\ t/w]) \to P[f/w]\end{array}}{\Gamma \vdash \mathbf{rec}\ u/w.P\ \mathbf{with}\ p : P[u/w]}$$

Finally, the well-order type, or in short W-type, which will let us represent inductive data in a general way. We can form 'nodes' of the shape

$$t \lhd_{x.B} f : \mathsf{W}\ (x{:}A)\ B$$

where t is of type A and is the data present in the node, and f specifies a 'child' of the node for each member of $B[t/x]$. The **rec with** acts as an induction principle on W, given a predicate and a function dealing with the inductive case—we will gain more intuition about inductive data in Section 6.3.3.

For example, if we want to form natural numbers, we can take

$$\begin{aligned}
&\mathsf{Tr} : \mathsf{Bool} \to \mathsf{Type}_0 \\
&\mathsf{Tr}\ b \mapsto \mathbf{if}\ b\ \mathbf{then}\ \mathsf{Unit}\ \mathbf{else}\ \mathsf{Empty}
\end{aligned}$$

$$\begin{aligned}
&\mathbb{N} : \mathsf{Type}_0 \\
&\mathbb{N} \mapsto \mathsf{W}\ (b{:}\mathsf{Bool})\ (\mathsf{Tr}\ b)
\end{aligned}$$

Each node will contain a boolean. If true, the number is non-zero, and we will have one child representing its predecessor, given that Tr will return Unit. If false, the number is zero, and we will have no predecessors (children), given the Empty:

$$\begin{aligned}
&\mathsf{zero} : \mathbb{N} \\
&\mathsf{zero} \mapsto \mathsf{false} \lhd (\lambda x \mapsto \mathbf{absurd}_{\mathbb{N}}\ x)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{suc} : \mathbb{N} \to \mathbb{N} \\
&\mathsf{suc}\ x \mapsto \mathsf{true} \lhd (\lambda_ \mapsto x)
\end{aligned}$$

And with a bit of effort, we can recover addition:

```
plus : ℕ → ℕ → ℕ
plus x y ↦
  rec x / b.ℕ
  with λb ↦
    if b / b′.((Tr b′ → ℕ) → (Tr b′ → ℕ) → ℕ)
    then (λ_ f ↦ suc (f ⟨⟩)) else (λ_ _ ↦ y)
```

Note how we explicitly have to type the branches to make them match with the definition of ℕ. This gives a taste of the clumsiness of W-types but not the whole story. Well-orders are inadequate not only because they are verbose, but also because they face deeper problems due to the weakness of the notion of equality present in most type theories, which we will present in the next section (Dybjer, 1997). The 'better' equality we will present in Section 5 helps but does not fully resolve these issues.[3] For this reasons W-types have remained nothing more than a reasoning tool, and practical systems must implement more manageable ways to represent data.

[3] See http://www.e-pig.org/epilogue/?p=324, which concludes with 'W-types are a powerful conceptual tool, but they're no basis for an implementation of recursive data types in decidable type theories.'

4 | The struggle for equality

Half of my time spent doing research involves thinking up clever schemes to avoid needing functional extensionality.

@larrytheliquid

In the previous section we learnt how a type checker for ITT needs a notion of *definitional equality*. Beyond this meta-theoretic notion, in this section we will explore the ways of expressing equality *inside* the theory, as a reasoning tool available to the user. This area is the main concern of this thesis, and in general a very active research topic, since we do not have a fully satisfactory solution, yet. As in the previous section, everything presented is formalised in Agda in Appendix B.1.1.

4.1 Propositional equality

Definition (Propositional equality)**.** *The syntax, reduction, and typing rules for propositional equality and related constructs are defined as:*

syntax

$$\begin{aligned} \mathit{term} ::= {} & \cdots \\ \mid {} & = \mathit{term}\ \mathit{term}\ \mathit{term} \mid \mathsf{refl}\ \mathit{term} \\ \mid {} & =\textbf{-elim}\ \mathit{term}\ \mathit{term}\ \mathit{term} \end{aligned}$$

reduction: $\mathit{term} \rightsquigarrow \mathit{term}$

$$=\textbf{-elim}\ P\ (\mathsf{refl}\ m)\ n \rightsquigarrow n$$

typing: $\Gamma \vdash \mathit{term} : \mathit{term}$

$$\frac{\Gamma \vdash A : \mathsf{Type}_l \qquad \Gamma \vdash m : A \qquad \Gamma \vdash n : A}{\Gamma \vdash = A\ m\ n : \mathsf{Type}_l}$$

$$\frac{\Gamma \vdash m : A \qquad m \cong n}{\Gamma \vdash \mathsf{refl}\ m : = A\ m\ n} \qquad \frac{\Gamma \vdash P : (x\ y{:}A) \to (q{:=}\ A\ x\ y) \to \mathsf{Type}_l \qquad \Gamma \vdash q : = A\ m\ n \qquad \Gamma \vdash p : P\ m\ m\ (\mathsf{refl}\ m)}{\Gamma \vdash =\textbf{-elim}\ P\ q\ p : P\ m\ n\ q}$$

To express equality between two terms inside ITT, the obvious way to do so is to have equality to be a type. Here we present what has survived as the dominating form of equality in systems based on ITT up since Martin-Löf (1984) up to the present day.

Our type former is $=$, which given a type relates equal terms of that type. $=$ has one introduction rule, refl, which introduces an equality between definitionally equal terms—a proof by reflexivity.

Finally, we have one eliminator for $=$, $=$**-elim** (also known as '**J** axiom' in the literature). $=$**-elim** $P\ q\ p$ takes

- P, a predicate working with two terms of a certain type (say A) and a proof of their equality;
- q, a proof that two terms in A (say m and n) are equal;
- and p, an inhabitant of P applied to m twice, plus the trivial proof by reflexivity showing that m is equal to itself.

Given these ingredients, $=$**-elim** returns a member of P applied to m, n, and q. In other words $=$**-elim** takes a witness that P works with *definitionally equal* terms, and returns a

witness of P working with *propositionally equal* terms. Given its reduction rules, invocations of $=$**-elim** will vanish when the equality proofs will reduce to invocations to reflexivity, at which point the arguments must be definitionally equal, and thus the provided $P\ m\ m\ (\mathsf{refl}\ m)$ can be returned. This means that $=$**-elim** will not compute with hypothetical proofs, which makes sense given that they might be false.

While the $=$**-elim** rule is slightly convoluted, we can derive many more 'friendly' rules from it, for example a more obvious 'substitution' rule, that replaces equal for equal in predicates:

$$\begin{aligned}&\textbf{subst} : (A{:}\mathsf{Type}) \to (P{:}A \to \mathsf{Type}) \to (x\ y{:}A) \to = A\ x\ y \to P\ x \to P\ y\\ &\textbf{subst}\ A\ P\ x\ y\ q\ p \mapsto =\textbf{-elim}\ (\lambda x\ y\ q \mapsto P\ y)\ p\ q\end{aligned}$$

Once we have **subst**, we can easily prove more familiar laws regarding equality, such as symmetry, transitivity, congruence laws, etc.[4]

4.2 Common extensions

Our definitional and propositional equalities can be enhanced in various ways. Obviously if we extend the definitional equality we are also automatically extend propositional equality, given how refl works.

4.2.1 η-expansion

A simple extension to our definitional equality is achieved by η-expansion. Given an abstract variable $f : A \to B$ the aim is to have that $f \cong \lambda x{:}A \mapsto f\ x$. We can achieve this by 'expanding' terms depending on their types, a process known as *quotation*—a term borrowed from the practice of *normalisation by evaluation*, where we embed terms in some host language with an existing notion of computation, and then reify them back into terms, which will 'smooth out' differences like the one above (Abel *et al.* , 2007).

The same concept applies to $\times$, where we expand each inhabitant reconstructing it by getting its projections, so that $x \cong \langle \textbf{fst}\ x, \textbf{snd}\ x \rangle$. Similarly, all one inhabitants of Unit and all zero inhabitants of Empty can be considered equal. Quotation can be performed in a type-directed way, as we will witness in Section 6.5.6.

Definition (Congruence and η-laws)**.** *To justify quotation in our type system we add a congruence law for abstractions and a similar law for products, plus the fact that all elements of* Unit *or* Empty *are equal.*

definitional equality: $\Gamma \vdash m \cong n : term$

$$\frac{\Gamma; y : A \vdash f\ x \cong g\ x : B[y/x]}{\Gamma \vdash f \cong g : (x{:}A) \to B} \qquad \frac{\Gamma \vdash \langle \textbf{fst}\ m, \textbf{snd}\ m \rangle \cong \langle \textbf{fst}\ n, \textbf{snd}\ n \rangle : (x{:}A) \times B}{\Gamma \vdash m \cong n : (x{:}A) \times B}$$

$$\frac{\Gamma \vdash m : \mathsf{Unit} \qquad \Gamma \vdash n : \mathsf{Unit}}{\Gamma \vdash m \cong n : \mathsf{Unit}} \qquad \frac{\Gamma \vdash m : \mathsf{Empty} \qquad \Gamma \vdash n : \mathsf{Empty}}{\Gamma \vdash m \cong n : \mathsf{Empty}}$$

4.2.2 Uniqueness of identity proofs

Another common but controversial addition to propositional equality is the **K** axiom, which essentially states that all equality proofs are by reflexivity.

Definition (**K** axiom)**.**

[4] For definitions of these functions, refer to Appendix B.1.

typing: $\Gamma \vdash m \cong n : term$

$$\frac{\Gamma \vdash P : (x{:}A) \to = A\ x\ x \to \mathsf{Type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash p : P\ t\ (\mathsf{refl}\ t) \qquad \Gamma \vdash q : t{=}A t}{\Gamma \vdash \mathbf{K}\ P\ t\ p\ q : P\ t\ q}$$

Hofmann & Streicher (1994) showed that **K** is not derivable from **=-elim**, and McBride & McKinna (2004a) showed that it is needed to implement 'dependent pattern matching', as first proposed by Coquand (1992).[5] Thus, **K** is derivable in the systems that implement dependent pattern matching, such as Epigram and Agda; but for example not in Coq.

K is controversial mainly because it is at odds with equalities that include computational content, most notably Voevodsky's *Univalent Foundations*, which feature a *univalence* axiom that identifies isomorphisms between types with propositional equality. For example we would have two isomorphisms, and thus two equalities, between Bool and Bool, corresponding to the two permutations—one is the identity, and one swaps the elements. Given this, **K** and univalence are inconsistent, and thus a form of dependent pattern matching that does not imply **K** is subject of research.[6]

4.3 Limitations

Propositional equality as described is quite restricted when reasoning about equality beyond the term structure, which is what definitional equality gives us (extensions notwithstanding).

Definition (Extensional equality)**.** *Given two functions f and g of type $A \to B$, they are are said to be* extensionally equal *if*

$$(x{:}A) \to = B\ (f\ x)\ (g\ x)$$

The problem is best exemplified by *function extensionality*. In mathematics, we would expect to be able to treat functions that give equal output for equal input as equal. When reasoning in a mechanised framework we ought to be able to do the same: in the end, without considering the operational behaviour, all functions equal extensionally are going to be replaceable with one another.

However in ITT this is not the case, or in other words with the tools we have there is no closed term of type

$$\mathbf{ext} : (A\ B{:}\mathsf{Type}) \to (f\ g{:}A \to B) \to ((x{:}A) \to = B\ (f\ x)\ (g\ x)) \to = (A \to B)\ f\ g$$

To see why this is the case, consider the functions

$$\lambda x \mapsto 0 + x \text{ and } \lambda x \mapsto x + 0$$

where $+$ is defined by recursion on the first argument, gradually destructing it to build up successors of the second argument. The two functions are clearly extensionally equal, and we can in fact prove that

$$(x{:}\mathbb{N}) \to = \mathbb{N}\ (0 + x)\ (x + 0)$$

By induction on $\mathbb{N}$ applied to x. However, the two functions are not definitionally equal, and thus we will not be able to get rid of the quantification.

[5] See Section 9 for more on dependent pattern matching.

[6] More information about univalence can be found at `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations.html`.

For the reasons given above, theories that offer a propositional equality similar to what we presented are called *intensional*, as opposed to *extensional*. Most systems widely used today (such as Agda, Coq, and Epigram) are of the former kind.

This is quite an annoyance that often makes reasoning awkward or impossible to execute. For example, we might want to represent terms of some language in Agda and give their denotation by embedding them in Agda—if we had λ-terms, functions will be Agda functions, application will be Agda's function application, and so on. Then we would like to perform optimisation passes on the terms, and verify that they are sound by proving that the denotation of the optimised version is equal to the denotation of the starting term.

But if the embedding uses functions—and it probably will—we are stuck with an equality that identifies as equal only syntactically equal functions! Since the point of optimising is about preserving the denotational but changing the operational behaviour of terms, our equality falls short of our needs. Moreover, the problem extends to other fields beyond functions, such as bisimulation between processes specified by coinduction, or in general proving equivalences based on the behaviour of a term.

4.4 Equality reflection

One way to 'solve' this problem is by identifying propositional equality with definitional equality.

Definition (Equality reflection)**.**

typing: $\Gamma \vdash term : term$

$$\frac{\Gamma \vdash q : = A\ m\ n}{\Gamma \vdash m \cong n : A}$$

The *equality reflection* rule is a very different rule from the ones we saw up to now: it links a typing judgement internal to the type theory to a meta-theoretic judgement that the type checker uses to work with terms. It is easy to see the dangerous consequences that this causes:

- The rule is not syntax directed, and the type checker is presumably expected to come up with equality proofs when needed.
- More worryingly, type checking becomes undecidable also because computing under false assumptions becomes unsafe, since we derive any equality proof and then use equality reflection and the conversion rule to have terms of any type.

Given these facts theories employing equality reflection, like NuPRL (Constable & the PRL Group, 1986), carry the derivations that gave rise to each typing judgement to keep the systems manageable.

For all its faults, equality reflection does allow us to prove extensionality, using the extensions given in Section 4.2. Assuming that Γ contains

$$A, B : \mathsf{Type}; f, g : A \to B; q : (x{:}A) \to f\ x{=}g\ x$$

We can then derive

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma; x : A \vdash q : = A\ (f\ x)\ (g\ x)}{\Gamma; x : A \vdash f\ x \cong g\ x : B}\ \text{equality reflection}}{\Gamma \vdash (\lambda x \mapsto f\ x) \cong (\lambda x \mapsto g\ x) : A \to B}\ \text{congruence for } \lambda\text{s}}{\Gamma \vdash f \cong g : A \to B}\ \eta\text{-law for } \lambda}{\Gamma \vdash \mathsf{refl}\ f : = (A \to B)\ f\ g}\ \mathsf{refl}$$

For this reason, theories employing equality reflection are often grouped under the name of *Extensional Type Theory* (ETT). Now, the question is: do we need to give up well-behavedness of our theory to gain extensionality?

5 | The observational approach

A recent development by Altenkirch *et al.* (2007), *Observational Type Theory* (OTT), promises to keep the well behavedness of ITT while being able to gain many useful equality proofs,[7] including function extensionality. The main idea is have equalities to express structural properties of the equated terms, instead of blindly comparing the syntax structure. In the case of functions, this will correspond to extensionality, in the case of products it will correspond to having equal projections, and so on. Moreover, we are given a way to *coerce* values from A to B, if we can prove A equal to B, following similar principles to the ones described above. Here we give an exposition which follows closely the original paper.

5.1 A simpler theory, a propositional fragment

Definition (OTT's simple theory, with propositions)**.**

syntax

Type_l is replaced by Type.

$$\begin{aligned} \mathit{term} &::= \cdots \mid [\![\mathit{prop}]\!] \mid \mathbf{If}\ \mathit{term}\ \mathbf{Then}\ \mathit{term}\ \mathbf{Else}\ \mathit{term} \\ \mathit{prop} &::= \bot \mid \top \mid \mathit{prop} \wedge \mathit{prop} \mid \forall x{:}\mathit{term}.\ \mathit{prop} \end{aligned}$$

reduction: $\mathit{term} \rightsquigarrow \mathit{term}$

$$\begin{aligned} \mathbf{If}\ \mathsf{true}\ \mathbf{Then}\ A\ \mathbf{Else}\ B &\rightsquigarrow A \\ \mathbf{If}\ \mathsf{false}\ \mathbf{Then}\ A\ \mathbf{Else}\ B &\rightsquigarrow B \end{aligned}$$

typing: $\Gamma \vdash \mathit{term} : \mathit{term}$

$$\frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash [\![P]\!] : \mathsf{Type}} \qquad \frac{\Gamma \vdash t : \mathsf{Bool} \qquad \Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash \mathbf{If}\ t\ \mathbf{Then}\ A\ \mathbf{Else}\ B : \mathsf{Type}}$$

propositions: $\Gamma \vdash \mathit{prop} : \mathsf{Prop}$

$$\frac{}{\Gamma \vdash \top : \mathsf{Prop}} \qquad \frac{}{\Gamma \vdash \bot : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \wedge Q : \mathsf{Prop}} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma; x : A \vdash P : \mathsf{Prop}}{\Gamma \vdash \forall x{:}A.\ P : \mathsf{Prop}}$$

Our foundation will be a type theory like the one of Section 3, with only one level: Type_0. In this context we will drop the 0 and call Type_0 Type. Moreover, since the old **if_then_else** was able to return types thanks to the hierarchy (which is gone), we need to reintroduce an ad-hoc conditional for types, where the reduction rule is the obvious one.

However, we have an addition: a universe of *propositions*, Prop.[8] Prop isolates a fragment of types at large, and indeed we can 'inject' any Prop back in Type with $[\![\_]\!]$.

Definition (Proposition decoding)**.**

proposition decoding: $[\![\mathit{term}]\!] \rightsquigarrow \mathit{term}$

$$\begin{aligned} [\![\bot]\!] &\rightsquigarrow \mathsf{Empty} & [\![\, P \wedge Q \,]\!] &\rightsquigarrow [\![P]\!] \times [\![Q]\!] \\ [\![\top]\!] &\rightsquigarrow \mathsf{Unit} & [\![\forall x{:}A.\ P]\!] &\rightsquigarrow (x{:}A) \to [\![P]\!] \end{aligned}$$

[7] It is suspected that OTT gains *all* the equality proofs of ETT, but no proof exists yet.

[8] Note that we do not need syntax for the type of props, Prop, since the user cannot abstract over them. In fact, we do not not need syntax for Type either, for the same reason.

Propositions are what we call the types of *proofs*, or types whose inhabitants contain no 'data', much like Unit. Types of these kind are called *irrelevant*. Irrelevance can be exploited in various ways—we can identify all equivalent proportions as definitionally equal equal, as we will see later; and erase all the top level propositions when compiling.

Why did we choose what we have in Prop? Given the above criteria, $\top$ obviously fits the bill, since it has one element. A pair of propositions $P \wedge Q$ still won't get us data, since if they both have one element the only possible pair is the one formed by said elements. Finally, if P is a proposition and we have $\forall x{:}A.\, P$, the decoding will be a constant function for propositional content. The only threat is $\bot$, by which we can fabricate anything we want: however if we are consistent there will be no closed term of type $\bot$ at, which is enough regarding proof erasure and term equality.

As an example of types that are *not* propositional, consider Booleans, which are the quintessential 'relevant' data, since they are often used to decide the execution path of a program through **if_then_else_** constructs.

5.2 Equality proofs

Definition (Equality proofs and related operations)**.**

syntax

$$\begin{array}{rcl} \mathit{term} & ::= & \cdots \mid \mathbf{coe}\ \mathit{term}\ \mathit{term}\ \mathit{term}\ \mathit{term} \mid \mathbf{coh}\ \mathit{term}\ \mathit{term}\ \mathit{term}\ \mathit{term} \\ \mathit{prop} & ::= & \cdots \mid \mathit{term} = \mathit{term} \mid (\mathit{term}{:}\mathit{term}) = (\mathit{term}{:}\mathit{term}) \end{array}$$

typing: $\Gamma \vdash \mathit{term} : \mathit{term}$

$$\frac{\Gamma \vdash P : [\![A = B]\!] \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathbf{coe}\ A\ B\ P\ t : B} \qquad \frac{\Gamma \vdash P : [\![A = B]\!] \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathbf{coh}\ A\ B\ P\ t : [\![(t{:}A) = (\mathbf{coe}\ A\ B\ P\ t{:}B)]\!]}$$

propositions: $\Gamma \vdash \mathit{prop} : \mathsf{Prop}$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A = B : \mathsf{Prop}} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash m : A \qquad \Gamma \vdash B : \mathsf{Type} \qquad \Gamma \vdash n : B}{\Gamma \vdash (m{:}A) = (n{:}B) : \mathsf{Prop}}$$

While isolating a propositional universe as presented can be a useful exercises on its own, what we are really after is a useful notion of equality. In OTT we want to maintain that things judged to be equal are still always replaceable for one another with no additional changes. Note that this is not the same as saying that they are definitionally equal, since as we saw extensionally equal functions, while satisfying the above requirement, are not.

Towards this goal we introduce two equality constructs in Prop—the fact that they are in Prop indicates that they indeed have no computational content. The first construct, $\_ = \_$, relates types, the second, $(\_{:}\_) = (\_{:}\_)$, relates values. The value-level equality is different from our old propositional equality: instead of ranging over only one type, we might form equalities between values of different types—the usefulness of this construct will be clear soon. In the literature this equality is known as 'heterogeneous' or 'John Major', since

> John Major's 'classless society' widened people's aspirations to equality, but also the gap between rich and poor. After all, aspiring to be equal to others than oneself is the politics of envy. In much the same way, $(\_{:}\_) = (\_{:}\_)$ forms equations between members of any type, but they cannot be treated as equals (ie substituted) unless they are of the same type. Just as before, each thing is only equal to itself. (McBride, 1999).

Correspondingly, at the term level, **coe** ('coerce') lets us transport values between equal types; and **coh** ('coherence') guarantees that **coe** respects the value-level equality, or in other words that it really has no computational component. If we transport $m : A$ to $n : B$, m and n will still be the same.

Before introducing the core machinery of OTT work, let us distinguish between *canonical* and *neutral* terms and types.

Definition (Canonical and neutral terms and types)**.** *In a type theory,* neutral *terms are those formed by an abstracted variable or by an eliminator (including function application). Everything else is* canonical*.*

*In the current system, data constructors ($\langle\rangle$, true, false, $\lambda x{:}A \mapsto t$, ...) will be canonical, the rest neutral. Correspondingly, canonical types are those arising from the ground types (*Empty, Unit, Bool*) and the three type formers ($\to$, $\times$, W). Neutral types are those formed by* **If_Then_Else_***.*

Definition (Canonicity)**.** *If in a system all canonical types are inhabited by canonical terms the system is said to have the* canonicity *property.*

The current system, and well-behaved systems in general, has the canonicity property. Another consequence of normalisation is that all closed terms will reduce to a canonical term.

5.2.1 Type equality, and coercions

The plan is to decompose type-level equalities between canonical types into decodable propositions containing equalities regarding the subtypes. So if are equating two product types, the equality will reduce to two subequalities regarding the first and second type. Then, we can **coe**rce to transport values between equal types. Following the subequalities, **coe** will proceed recursively on the subterms.

This interplay between the canonicity of equated types, type equalities, and **coe**, ensures that invocations of **coe** will vanish when we have evidence of the structural equality of the types we are transporting terms across. If the type is neutral, the equality will not reduce and thus **coe** will not reduce either. If we come across an equality between different canonical types, then we reduce the equality to bottom, thus making sure that no such proof can exist, and providing an 'escape hatch' in **coe**.

Definition (Type equalities reduction, and **coe**rcions)**.** *Figure 5 illustrates the rules to reduce equalities and to coerce terms. We use a* let *syntax for legibility.*

For ground types, the proof is the trivial element, and **coe** is the identity. For Unit, we can do better: we return its only member without matching on the term. For the three type binders the choices we make in the type equality are dictated by the desire of writing the **coe** in a natural way.

$\times$ is the easiest case: we decompose the proof into proofs that the first element's types are equal ($A_1 = A_2$), and a proof that given equal values in the first element, the types of the second elements are equal too ($\forall x_1{:}A_1.\, \forall x_2{:}A_2.\, (x_1{:}A_1) = (x_2{:}A_2) \Rightarrow B_1[x_1] = B_2[x_2]$).[9] This also explains the need for heterogeneous equality, since in the second proof we need to equate terms of possibly different types. In the respective **coe** case, since the types are canonical, we know at this point that the proof of equality is a pair of the shape described above. Thus, we can immediately coerce the first element of the pair using the first element of the proof, and then instantiate the second element of the proof with the two first elements and a proof by coherence of their equality, since we know that the types are equal.

[9] We are using $\Rightarrow$ to indicate a $\forall$ where we discard the quantified value. We write $B_1[x_1]$ to indicate that the x_1 in B_1 is re-bound to the x_1 quantified by the $\forall$, and similarly for x_2 and B_2.

equality reduction: $prop \rightsquigarrow prop$

$$
\begin{array}{rcll}
\mathsf{Empty} & = & \mathsf{Empty} & \rightsquigarrow \top \\
\mathsf{Unit} & = & \mathsf{Unit} & \rightsquigarrow \top \\
\mathsf{Bool} & = & \mathsf{Bool} & \rightsquigarrow \top \\
(x_1{:}A_1) \times B_1 & = & (x_2{:}A_2) \times A_2 & \rightsquigarrow \\
\multicolumn{4}{r}{A_1 = A_2 \wedge \forall x_1{:}A_1.\, \forall x_2{:}A_2.\, (x_1{:}A_1) = (x_2{:}A_2) \Rightarrow B_1[x_1] = B_2[x_2]} \\
(x_1{:}A_1) \to B_1 & = & (x_2{:}A_2) \to B_2 & \rightsquigarrow \cdots \\
\mathsf{W}\,(x_1{:}A_1)\,B_1 & = & \mathsf{W}\,(x_2{:}A_2)\,B_2 & \rightsquigarrow \cdots \\
A & = & B & \rightsquigarrow \bot \text{ if } A \text{ and } B \text{ are canonical.}
\end{array}
$$

reduction $term \rightsquigarrow term$

$$
\begin{array}{lllll}
\mathbf{coe}\ \mathsf{Empty} & \mathsf{Empty} & Q\ t & \rightsquigarrow t \\
\mathbf{coe}\ \mathsf{Unit} & \mathsf{Unit} & Q\ t & \rightsquigarrow \langle\rangle \\
\mathbf{coe}\ \mathsf{Bool} & \mathsf{Bool} & Q\ \mathsf{true} & \rightsquigarrow \mathsf{true} \\
\mathbf{coe}\ \mathsf{Bool} & \mathsf{Bool} & Q\ \mathsf{false} & \rightsquigarrow \mathsf{false} \\
\mathbf{coe}\ ((x_1{:}A_1) \times B_1) & ((x_2{:}A_2) \times B_2) & Q\ t_1 & \rightsquigarrow \\
\quad \underline{\mathsf{let}}\ m_1 \mapsto \mathbf{fst}\ t_1 : A_1 & & & \\
\qquad n_1 \mapsto \mathbf{snd}\ t_1 : B_1[m_1/x_1] & & & \\
\qquad Q_A \mapsto \mathbf{fst}\ Q : A_1 = A_2 & & & \\
\qquad m_2 \mapsto \mathbf{coe}\ A_1\ A_2\ Q_A\ m_1 : A_2 & & & \\
\qquad Q_B \mapsto (\mathbf{snd}\ Q)\ m_1\ m_2\ (\mathbf{coh}\ A_1\ A_2\ Q_A\ m_1) : [\![B_1[m_1/x_1] = B_2[m_2/x_2]]\!] & & & \\
\qquad n_2 \mapsto \mathbf{coe}\ B_1[m_1/x_1]\ B_2[m_2/x_2]\ Q_B\ n_1 : B_2[m_2/x_2] & & & \\
\quad \underline{\mathsf{in}}\ \langle m_2, n_2\rangle & & & \\
\mathbf{coe}\ ((x_1{:}A_1) \to B_1) & ((x_2{:}A_2) \to B_2) & Q\ t & \rightsquigarrow \cdots \\
\mathbf{coe}\ (\mathsf{W}\,(x_1{:}A_1)\,B_1) & (\mathsf{W}\,(x_2{:}A_2)\,B_2) & Q\ t & \rightsquigarrow \cdots \\
\mathbf{coe}\ A & B & Q\ t & \rightsquigarrow \mathbf{absurd}_B\ Q \text{ if } A \text{ and } B \text{ are canonical.}
\end{array}
$$

Figure 5: Reducing type equalities, and using them when **coe**rcing.

The cases for the other binders are omitted for brevity, but they follow the same principle with some twists to make **coe** work with the generated proofs; the reader can refer to the paper for details.

5.2.2 coe, laziness, and coherence

It is important to notice that the reduction rules for **coe** are never obstructed by the structure of the proofs. With the exception of comparisons between different canonical types we never 'pattern match' on the proof pairs, but always look at the projections. This means that, as long as we are consistent, and thus as long as we don't have $\bot$-inducing proofs, we can add propositional axioms for equality and **coe** will still compute. Thus, we can take **coh** as axiomatic, and we can add back familiar useful equality rules:

typing: $\Gamma \vdash term : term$

$$
\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{refl}\ t : [\![(t{:}A) = (t{:}A)]\!]}
$$

$$
\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma; x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \mathsf{R}\,(x{:}A)\,B : (y\,z{:}A) \to [\![(y{:}A) = (z{:}A) \Rightarrow B[y/x] = B[z/x]]\!]}
$$

refl is the equivalent of the reflexivity rule in propositional equality, and R asserts that if we have a we have a Type abstracting over a value we can substitute equal for equal—this lets us recover **subst**. Note that while we need to provide ad-hoc rules in the restricted, non-hierarchical theory that we have, if our theory supports abstraction over Types we can easily add these axioms as top-level abstracted variables.

5.2.3 Value-level equality

Definition (Value-level equality)**.**

equality reduction: $prop \rightsquigarrow prop$

$$
\begin{array}{l}
(\ t_1 : \mathsf{Empty}\ ) = (\ t_2 : \mathsf{Empty}\ ) \rightsquigarrow \top \\
(\ t_1 : \mathsf{Unit}\ ) = (\ t_2 : \mathsf{Unit}\ ) \rightsquigarrow \top \\
(\ \mathsf{true} : \mathsf{Bool}\ ) = (\ \mathsf{true} : \mathsf{Bool}\ ) \rightsquigarrow \top \\
(\ \mathsf{false} : \mathsf{Bool}\ ) = (\ \mathsf{false} : \mathsf{Bool}\ ) \rightsquigarrow \top \\
(\ \mathsf{true} : \mathsf{Bool}\ ) = (\ \mathsf{false} : \mathsf{Bool}\ ) \rightsquigarrow \bot \\
(\ \mathsf{false} : \mathsf{Bool}\ ) = (\ \mathsf{true} : \mathsf{Bool}\ ) \rightsquigarrow \bot \\
(\ t_1 : (A_1{:}x_1) \times B_1\ ) = (\ t_2 : (A_2{:}x_2) \times B_2\ ) \rightsquigarrow \\
\quad (\mathbf{fst}\, t_1{:}A_1) = (\mathbf{fst}\, t_2{:}A_2) \wedge (\mathbf{snd}\, t_1{:}B_1[\mathbf{fst}\, t_1/x_1]) = (\mathbf{snd}\, t_2{:}B_2[\mathbf{fst}\, t_2/x_2]) \\
(\ f_1 : (A_1{:}x_1) \to B_1) = (\ f_2 : (A_2{:}x_2) \to B_2) \rightsquigarrow \\
\quad \forall x_1{:}A_1.\, \forall x_2{:}A_2.\, (x_1{:}A_1) = (x_2{:}A_2) \Rightarrow (f_1\, x_1{:}B_1[x_1]) = (f_2\, x_2{:}B_2[x_2]) \\
(t_1 \lhd f_1 : \mathsf{W}\, (A_1{:}x_1)\, B_1\ ) = (t_1 \lhd f_1 : \mathsf{W}\, (A_2{:}x_2)\, B_2\ ) \rightsquigarrow \cdots \\
(\ t_1 : A_1\ ) = (\ t_2 : A_2\ ) \rightsquigarrow \bot \text{ if } A_1 \text{ and } A_2 \text{ are canonical.}
\end{array}
$$

As with type-level equality, we want value-level equality to reduce based on the structure of the compared terms. When matching propositional data, such as Empty and Unit, we automatically return the trivial type, since if a type has zero or one members, all members will be equal. When matching on data-bearing types, such as Bool, we check that such data matches, and return bottom otherwise. When matching on records and functions, we rebuild the records to achieve η-expansion, and relate functions if they are extensionally equal—exactly what we wanted. The case for W is omitted but unsurprising, checking that equal data in the nodes will bring equal children.

5.3 Proof irrelevance and stuck coercions

The last effort is required to make sure that proofs (members of Prop) are *irrelevant*. Since they are devoid of computational content, we would like to identify all equivalent propositions as the same, in a similar way as we identified all Empty and all Unit as the same in section 4.2.1.

Thus we will have a quotation that will not only perform η-expansion, but will also identify and mark proofs that could not be decoded (that is, equalities on neutral types). Then, when comparing terms, marked proofs will be considered equal without analysing their contents, thus gaining irrelevance.

Moreover we can safely advance 'stuck' **coe**rcions between non-canonical but definitionally equal types. Consider for example

$$\mathbf{coe}\ (\mathbf{If}\ b\ \mathbf{Then}\ \mathbb{N}\ \mathbf{Else}\ \mathsf{Bool})\ (\mathbf{If}\ b\ \mathbf{Then}\ \mathbb{N}\ \mathbf{Else}\ \mathsf{Bool})\ x$$

Where b and x are abstracted variables. This **coe** will not advance, since the types are not canonical. However they are definitionally equal, and thus we can safely remove the coerce and return x as it is.

6 | Bertus: the theory

The construction itself is an art, its application to the world an evil parasite.

Luitzen Egbertus Jan 'Bertus' Brouwer

Bertus is an interactive theorem prover developed as part of this thesis. The plan is to present a core language which would be capable of serving as the basis for a more featureful system, while still presenting interesting features and more importantly observational equality.

We will first present the features of the system, along with motivations and trade-offs for the design decisions made. Then we describe the implementation we have developed in Section 7. For an overview of the features of Bertus, see Section 1.2, here we present them one by one. The exception is type holes, which we do not describe holes rigorously, but provide more information about them in Section 8.1.

Note that in this section we will present Bertus terms in a fancy LaTeX dress to keep up with the presentation, but every term, reduced to its concrete syntax (which we will present in Section 7.1), is a valid Bertus term accepted by Bertus the software, and not only Bertus the theory. Appendix B.2 displays most of the terms in this section in their concrete syntax.

6.1 Bidirectional type checking

We start by describing bidirectional type checking since it calls for fairly different typing rules that what we have seen up to now. The idea is to have two kinds of terms: terms for which a type can always be inferred, and terms that need to be checked against a type. A nice observation is that this duality is in correspondence with the notion of canonical and neutral terms: neutral terms (abstracted or defined variables, function application, record projections, primitive recursors, etc.) *infer* types, canonical terms (abstractions, record/data types data constructors, etc.) need to be *checked*.

To introduce the concept and notation, we will revisit the STLC in a bidirectional style. The presentation follows Löh *et al.* (2010). The syntax for our bidirectional STLC is the same as the untyped λ-calculus, but with an extra construct to annotate terms explicitly—this will be necessary when dealing with top-level canonical terms. The types are the same as those found in the normal STLC.

Definition (Syntax for the annotated λ-calculus)**.**

syntax

$$term ::= x \mid \lambda x \mapsto term \mid (term\ term) \mid (term : type)$$

We will have two kinds of typing judgements: *inference* and *checking*. $\Gamma \vdash t \Uparrow A$ indicates that t infers the type A, while $\Gamma \vdash t \Downarrow A$ can be checked against type A. The arrows indicate the direction of the type checking—inference pushes types up, checking propagates types down.

The type of variables in context is inferred. The type of applications and annotated terms is inferred too, propagating types down the applied and annotated term, respectively. Abstractions are checked. Finally, we have a rule to check the type of an inferrable term.

Definition (Bidirectional type checking for the STLC)**.**

typing: $\Gamma \vdash term \Updownarrow term$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x \Uparrow A} \qquad \frac{\Gamma; x : A \vdash t : B}{\Gamma \vdash \lambda x \mapsto t \Downarrow (x{:}A) \to B}$$

$$\frac{\Gamma \vdash m \Uparrow A \to B \qquad \Gamma \vdash n \Downarrow A}{\Gamma \vdash m\, n : B} \qquad \frac{\Gamma \vdash t \Downarrow A}{\Gamma \vdash t : A \Uparrow A} \qquad \frac{\Gamma \vdash t \Uparrow A}{\Gamma \vdash t \Downarrow A}$$

For example, if we wanted to type function composition (in this case for naturals), we would have to annotate the term:

$$\textbf{comp} : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$$
$$\textbf{comp}\ f\ g\ x \mapsto f\ (g\ x)$$

But we would not have to annotate functions passed to it, since the type would be propagated to the arguments:

$$\textbf{comp}\ (\lambda x \mapsto x + 3)\ (\lambda x \mapsto x * 4)\ 42$$

6.2 Base terms and types

Let us begin by describing the primitives available without the user defining any data types, and without equality. The way we handle variables and substitution is left unspecified, and explained in section 7.2, along with other implementation issues. We are also going to give an account of the implicit type hierarchy separately in Section 6.4, so as not to clutter derivation rules too much, and just treat types as impredicative for the time being.

Definition (Syntax for base types in Bertus)**.**

syntax

$$\begin{aligned} term &::= name \mid \mathsf{Type} \\ &\quad \mid (x{:}term) \to term \mid \lambda x \mapsto term \mid (term\ term) \mid (term : term) \\ name &::= x \mid \mathbf{f} \end{aligned}$$

The syntax for our calculus includes just two basic constructs: abstractions and Types. Everything else will be user-defined. Since we let the user define values too, we will need a context capable of carrying the body of variables along with their type.

Definition (Context validity)**.** *Bound names and defined names are treated separately in the syntax, and while both can be associated to a type in the context, only defined names can be associated with a body.*

context validity: $\Gamma \vdash \underline{\text{valid}}$

$$\frac{}{\varepsilon \vdash \underline{\text{valid}}} \qquad \frac{\Gamma \vdash A \Downarrow \mathsf{Type} \qquad name \notin \Gamma}{\Gamma; name : A \vdash \underline{\text{valid}}} \qquad \frac{\Gamma \vdash t \Downarrow A \qquad \mathbf{f} \notin \Gamma}{\Gamma; \mathbf{f} \mapsto t : A \vdash \underline{\text{valid}}}$$

Now we can present the reduction rules, which are unsurprising. We have the usual function application (β-reduction), but also a rule to replace names with their bodies (δ-reduction), and one to discard type annotations. For this reason reduction is done in-context, as opposed to what we have seen in the past.

Definition (Reduction rules for base types in Bertus)**.**

reduction: $\Gamma \vdash term \rightsquigarrow term$

$$\frac{}{\Gamma \vdash (\lambda x \mapsto m)\, n \rightsquigarrow m[n/x]} \qquad \frac{\mathbf{f} \mapsto t : A \in \Gamma}{\Gamma \vdash \mathbf{f} \rightsquigarrow t} \qquad \frac{}{\Gamma \vdash m : A \rightsquigarrow m}$$

We can now give types to our terms. Although we include the usual conversion rule, we defer a detailed account of definitional equality to Section 6.5.6.

Definition (Bidirectional type checking for base types in Bertus)**.**

typing: $\Gamma \vdash term \Updownarrow term$

$$\frac{name : A \in \Gamma}{\Gamma \vdash name \Uparrow A} \qquad \frac{\mathbf{f} \mapsto t : A \in \Gamma}{\Gamma \vdash \mathbf{f} \Uparrow A} \qquad \frac{\Gamma \vdash t \Downarrow A}{\Gamma \vdash t : A \Uparrow A} \qquad \frac{\Gamma \vdash t \Uparrow A \qquad \Gamma \vdash A \cong B}{\Gamma \vdash t \Downarrow B}$$

$$\frac{}{\Gamma \vdash \mathsf{Type} \Uparrow \mathsf{Type}} \qquad \frac{\Gamma \vdash A \Downarrow \mathsf{Type} \qquad \Gamma; x : A \vdash B \Downarrow \mathsf{Type}}{\Gamma \vdash (x{:}A) \to B \Uparrow \mathsf{Type}}$$

$$\frac{\Gamma \vdash m \Uparrow (x{:}A) \to B \qquad \Gamma \vdash n \Downarrow A}{\Gamma \vdash m\, n \Uparrow B[n/x]} \qquad \frac{\Gamma; x : A \vdash t \Downarrow B}{\Gamma \vdash \lambda x \mapsto t \Downarrow (x{:}B) \to B}$$

6.3 Elaboration

As we mentioned, Bertus allows the user to define not only values but also custom data types and records. *Elaboration* consists of turning these declarations into workable syntax, types, and reduction rules. The treatment of custom types in Bertus is heavily inspired by McBride's and McKinna's early work on Epigram (McBride & McKinna, 2004a), although with some differences.

6.3.1 Term vectors, telescopes, and assorted notation

Definition (Term vector)**.** *A* term vector *is a series of terms. The empty vector is represented by* ε*, and a new element is added with* $\_;\_$*, similarly to contexts—*$\vec{t}; m$*.*

We denote term vectors with the usual arrow notation, e.g. $\vec{t}$, $\vec{t}; m$, etc. We often use term vectors to refer to a series of term applied to another. For example $\mathsf{D}\ \vec{A}$ is a shorthand for $\mathsf{D}\ A_1 \cdots A_n$, for some n. n is consistently used to refer to the length of such vectors, and i to refer to an index such that $1 \leq i \leq n$.

Definition (Telescope)**.** *A* telescope *is a series of typed bindings. The empty telescope is represented by* ε*, and a binding is added via* $\_;\_$*.*

To present the elaboration and operations on user defined data types, we frequently make use what de Bruijn (1991) called *telescopes*, a construct that will prove useful when dealing with the types of type and data constructors. We refer to telescopes with Δ, Δ', Δ_i, etc. If Δ refers to a telescope, δ refers to the term vector made up of all the variables bound by Δ. $\Delta \to A$ refers to the type made by turning the telescope into a series of $\to$. For example we have that

$$(x{:}\mathbb{N}); (p : \mathbf{even}\ x) \to \mathbb{N} = (x{:}\mathbb{N}) \to (p : \mathbf{even}\ x) \to \mathbb{N}$$

We make use of various operations to manipulate telescopes:

- $\textsc{head}(\Delta)$ refers to the first type appearing in Δ: $\textsc{head}((x{:}\mathbb{N}); (p : \mathbf{even}\ x)) = \mathbb{N}$. Similarly, $\textsc{ix}_i(\Delta)$ refers to the i^{th} type in a telescope (1-indexed).

- $\textsc{take}_i(\Delta)$ refers to the telescope created by taking the first i elements of Δ: $\textsc{take}_1((x{:}\mathbb{N}); (p : \textbf{even}\ x)) = (x{:}\mathbb{N})$.
- $\Delta\vec{A}$ refers to the telescope made by 'applying' the terms in $\vec{A}$ on Δ: $((x{:}\mathbb{N}); (p : \textbf{even}\ x))42 = (p : \textbf{even}\ 42)$.

Additionally, when presenting syntax elaboration, We use $term^n$ to indicate a term vector composed of n elements. When clear from the context, we use term vectors to signify their length, e.g. $term^{\Delta}$, or $1 \leq i \leq \Delta$.

6.3.2 Declarations syntax

Definition (Syntax of declarations in Bertus)**.**

syntax

$$
\begin{aligned}
\mathit{decl} &::= x : \mathit{term} \mapsto \mathit{term} \\
&\mid \underline{\text{abstract}}\ x : \mathit{term} \\
&\mid \underline{\text{data}}\ \mathsf{D}\ \mathit{telescope}\ \underline{\text{where}}\ \{\mathsf{c} : \mathit{telescope} \mid \cdots\} \\
&\mid \underline{\text{record}}\ \mathsf{D}\ \mathit{telescope}\ \underline{\text{where}}\ \{\mathbf{f} : \mathit{term}, \cdots\} \\
\mathit{telescope} &::= \varepsilon \mid \mathit{telescope}; (x{:}\mathit{term}) \\
\mathit{name} &::= \cdots \mid \mathsf{D} \mid \mathsf{D.c} \mid \mathsf{D.f}
\end{aligned}
$$

In Bertus we have four kind of declarations:

Defined value A variable, together with a type and a body.

Abstract variable An abstract variable, with a type but no body.

Inductive data A *data type*, with a *type constructor* (denoted in blue, capitalised, sans serif: D) various *data constructors* (denoted in red, lowercase, sans serif: c), quite similar to what we find in Haskell. A primitive *eliminator* (or *destructor*, or *recursor*; denoted by green, lowercase, roman: **elim**) will be used to compute with each data type.

Record A *record*, which like data types consists of a type constructor but only one data constructor. The user can also define various *fields*, with no recursive occurrences of the type. The functions extracting the fields' values from an instance of a record are called *projections* (denoted in the same way as destructors).

Elaborating defined variables consists of type checking the body against the given type, and updating the context to contain the new binding. Elaborating abstract variables and abstract variables consists of type checking the type, and updating the context with a new typed variable.

Definition (Elaboration of defined and abstract variables)**.**

context elaboration: $\Gamma \vdash \mathit{decl} \rhd \Gamma$

$$
\frac{\Gamma \vdash t \Downarrow A \qquad \mathbf{f} \notin \Gamma}{\Gamma \vdash \mathbf{f} : A \mapsto t \ \rhd\ \Gamma; \mathbf{f} \mapsto t : A}
\qquad
\frac{\Gamma \vdash A \Downarrow \mathsf{Type} \qquad \mathbf{f} \notin \Gamma}{\Gamma \vdash \underline{\text{abstract}}\ \mathbf{f} : A \ \rhd\ \Gamma; \mathbf{f} : A}
$$

6.3.3 User defined types

Elaborating user defined types is the real effort. First, we will explain what we can define, with some examples.

Natural numbers To define natural numbers, we create a data type with two constructors: one with zero arguments (zero) and one with one recursive argument (suc):

$$\underline{\text{data}}\ \mathbb{N}\ \underline{\text{where}}\ \{\mathsf{zero} \mid \mathsf{suc}\ \mathbb{N}\}$$

This is very similar to what we would write in Haskell:

```
data Nat = Zero | Suc Nat
```

Once the data type is defined, Bertus will generate syntactic constructs for the type and data constructors, so that we will have

$$\frac{}{\Gamma \vdash \mathbb{N} \Uparrow \mathsf{Type}} \qquad \frac{}{\Gamma \vdash \mathbb{N}.\mathsf{zero} \Uparrow \mathbb{N}} \qquad \frac{\Gamma \vdash t \Downarrow \mathbb{N}}{\Gamma \vdash \mathbb{N}.\mathsf{suc}\ t \Uparrow \mathbb{N}}$$

While in Haskell (or indeed in Agda or Coq) data constructors are treated the same way as functions, in Bertus they are syntax, so for example using $\mathbb{N}.\mathsf{suc}$ on its own will give a syntax error. This is necessary so that we can easily infer the type of polymorphic data constructors, as we will see later.

Moreover, each data constructor is prefixed by the type constructor name, since we need to retrieve the type constructor of a data constructor when type checking. This measure aids in the presentation of the theory but it is not needed in the implementation, where we can have a dictionary to look up the type constructor corresponding to each data constructor. When using data constructors in examples I will omit the type constructor prefix for brevity, in this case writing zero instead of $\mathbb{N}.\mathsf{zero}$ and suc instead of $\mathbb{N}.\mathsf{suc}$.

Along with user defined constructors, Bertus automatically generates an *eliminator*, or *destructor*, to compute with natural numbers: If we have $t : \mathbb{N}$, we can destruct t using the generated eliminator '$\mathbb{N}.\mathbf{elim}$':

$$\frac{\Gamma \vdash t \Downarrow \mathbb{N}}{\Gamma \vdash \mathbb{N}.\mathbf{elim}\ t \Uparrow \begin{array}{l}(P{:}\mathbb{N} \to \mathsf{Type}) \to \\ P\ \mathsf{zero} \to ((x{:}\mathbb{N}) \to P\ x \to P\ (\mathsf{suc}\ x)) \to \\ P\ t\end{array}}$$

$\mathbb{N}.\mathbf{elim}$ corresponds to the induction principle for natural numbers: if we have a predicate on numbers (P), and we know that predicate holds for the base case ($P\ \mathsf{zero}$) and for each inductive step ($(x{:}\mathbb{N}) \to P\ x \to P\ (\mathsf{suc}\ x)$), then P holds for any number. As with the data constructors, we require the eliminator to be applied to the 'destructed' element.

While the induction principle is usually seen as a mean to prove properties about numbers, in the intuitionistic setting it is also a mean to compute. In this specific case $\mathbb{N}.\mathbf{elim}$ returns the base case if the provided number is zero, and recursively applies the inductive step if the number is a successor:

$$\begin{array}{l}\mathbb{N}.\mathbf{elim}\ \mathsf{zero}\quad P\ pz\ ps \rightsquigarrow pz \\ \mathbb{N}.\mathbf{elim}\ (\mathsf{suc}\ t)\ P\ pz\ ps \rightsquigarrow ps\ t\ (\mathbb{N}.\mathbf{elim}\ t\ P\ pz\ ps)\end{array}$$

The Haskell equivalent would be

```
elim :: Nat -> a -> (Nat -> a -> a) -> a
elim Zero     pz ps = pz
elim (Suc n) pz ps = ps n (elim n pz ps)
```

Which buys us the computational behaviour, but not the reasoning power, since we cannot express the notion of a predicate depending on $\mathbb{N}$—the type system is far too weak.

Binary trees Now for a polymorphic data type: binary trees, since lists are too similar to natural numbers to be interesting.

$$\underline{\text{data}}\ \mathsf{Tree}\ (A{:}\mathsf{Type})\ \underline{\text{where}}\ \{\mathsf{leaf} \mid \mathsf{node}\ (\mathsf{Tree}\ A)\ A\ (\mathsf{Tree}\ A)\}$$

Now the purpose of 'constructors as syntax' can be explained: what would the type of leaf be? If we were to treat it as a 'normal' term, we would have to specify the type parameter of the tree each time the constructor is applied:

$$\begin{aligned}\mathsf{leaf} &: (A{:}\mathsf{Type}) \to \mathsf{Tree}\ A\\ \mathsf{node} &: (A{:}\mathsf{Type}) \to \mathsf{Tree}\ A \to A \to \mathsf{Tree}\ A \to \mathsf{Tree}\ A\end{aligned}$$

The problem with this approach is that creating terms is incredibly verbose and dull, since we would need to specify the type parameter of Tree each time. For example if we wished to create a $\mathsf{Tree}\ \mathbb{N}$ with two nodes and three leaves, we would write

$$\mathsf{node}\ \mathbb{N}\ (\mathsf{node}\ \mathbb{N}\ (\mathsf{leaf}\ \mathbb{N})\ (\mathsf{suc}\ \mathsf{zero})\ (\mathsf{leaf}\ \mathbb{N}))\ \mathsf{zero}\ (\mathsf{leaf}\ \mathbb{N})$$

The redundancy of $\mathbb{N}$s is quite irritating. Instead, if we treat constructors as syntactic elements, we can 'extract' the type of the parameter from the type that the term gets checked against, much like what we do to type abstractions:

$$\frac{\Gamma \vdash A \Downarrow \mathsf{Type}}{\Gamma \vdash \mathsf{leaf} \Downarrow \mathsf{Tree}\ A} \qquad \frac{\Gamma \vdash m \Downarrow \mathsf{Tree}\ A \qquad \Gamma \vdash t \Downarrow A \qquad \Gamma \vdash m \Downarrow \mathsf{Tree}\ A}{\Gamma \vdash \mathsf{node}\ m\ t\ n \Downarrow \mathsf{Tree}\ A}$$

Which enables us to write, much more concisely

$$\mathsf{node}\ (\mathsf{node}\ \mathsf{leaf}\ (\mathsf{suc}\ \mathsf{zero})\ \mathsf{leaf})\ \mathsf{zero}\ \mathsf{leaf} : \mathsf{Tree}\ \mathbb{N}$$

We gain an annotation, but we lose the myriad of types applied to the constructors. Conversely, with the eliminator for Tree, we can infer the type of the arguments given the type of the destructed:

$$\frac{\Gamma \vdash t \Uparrow \mathsf{Tree}\ A}{\Gamma \vdash \mathsf{Tree.}\mathbf{elim}\ t \Uparrow \begin{array}{l}(P{:}\mathsf{Tree}\ A \to \mathsf{Type}) \to\\ P\ \mathsf{leaf} \to\\ ((l{:}\mathsf{Tree}\ A)(x{:}A)(r{:}\mathsf{Tree}\ A) \to P\ l \to P\ r \to P\ (\mathsf{node}\ l\ x\ r)) \to\\ P\ t\end{array}}$$

As expected, the eliminator embodies structural induction on trees. We have a base case for $P\ \mathsf{leaf}$, and an inductive step that given two subtrees and the predicate applied to them needs to return the predicate applied to the tree formed by a node with the two subtrees as children.

Empty type We have presented types that have at least one constructors, but nothing prevents us from defining types with *no* constructors:

$$\underline{\text{data}}\ \mathsf{Empty}\ \underline{\text{where}}\ \{\}$$

What shall the 'induction principle' on Empty be? Does it even make sense to talk about induction on Empty? Bertus does not care, and generates an eliminator with no 'cases':

$$\frac{\Gamma \vdash t \Uparrow \mathsf{Empty}}{\Gamma \vdash \mathsf{Empty.elim}\ t \Uparrow (P{:}t \to \mathsf{Type}) \to P\ t}$$

which lets us write the **absurd** that we know and love:

```
absurd : (A:Type) → Empty → A
absurd A x ↦ Empty.elim x (λ_ ↦ A)
```

Ordered lists Up to this point, the examples shown are nothing new to the {Haskell, SML, OCaml, functional} programmer. However dependent types let us express much more than that. A useful example is the type of ordered lists. There are many ways to define such a thing, but we will define ours to store the bounds of the list, making sure that consing respects that.

First, using Unit and Empty, we define a type expressing the ordering on natural numbers, **le**—'less or equal'. **le** $m\ n$ will be inhabited only if $m \leq n$:

```
le : ℕ → ℕ → Type
le n ↦
  ℕ.elim
    n
    (λ_ ↦ ℕ → Type)
    (λ_ ↦ Unit)
    (λn f m ↦ ℕ.elim m (λ_ ↦ Type) Empty (λm' _ ↦ f m'))
```

We return Unit if the scrutinised is zero (every number in less or equal than zero), Empty if the first number is a successor and the second a zero, and we recurse if they are both successors. Since we want the list to have possibly 'open' bounds, for example for empty lists, we create a type for 'lifted' naturals with a bottom ($\leq$ everything but itself) and top ($\geq$ everything but itself) elements, along with an associated comparison function:

```
data Lift where {bot | lift ℕ | top}
le' : Lift → Lift → Type
le' l₁ ↦
  Lift.elim
    l₁
    (λ_ ↦ Lift → Type)
    (λ_ ↦ Unit)
    (λn₁ l₂ ↦ Lift.elim l₂ (λ_ ↦ Type) Empty (λn₂ ↦ le n₁ n₂) Unit)
    (λl₂ ↦ Lift.elim l₂ (λ_ ↦ Type) Empty (λ_ ↦ Empty) Unit)
```

Finally, we can define a type of ordered lists. The type is parametrised over two *values* representing the lower and upper bounds of the elements, as opposed to the *type* parameters that we are used to in Haskell or similar languages. An empty list will have to have evidence that the bounds are ordered, and each time we add an element we require the list to have a matching lower bound:

```
data OList (low upp:Lift) where
  {nil (le' low upp) | cons (n:ℕ) (OList (lift n) upp) (le' low (lift n))}
```

Note that in the cons constructor we quantify over the first argument, which will determine the type of the following arguments—again something we cannot do in systems like Haskell. If we want we can then employ this structure to write and prove correct various sorting algorithms.[10]

[10] See this presentation by Conor McBride: https://personal.cis.strath.ac.uk/conor.mcbride/Pivotal.pdf, and this blog post by the author: http://mazzo.li/posts/AgdaSort.html.

Dependent products Apart from data, Bertus offers us another way to define types: record. A record is a data type with one constructor and 'projections' to extract specific fields of the said constructor.

For example, we can recover dependent products:

$$\underline{\text{record}}\ \mathsf{Prod}\ (A{:}\mathsf{Type})\ (B{:}A \to \mathsf{Type})\ \underline{\text{where}}\ \{\mathbf{fst} : A, \mathbf{snd} : B\ fst\}$$

Here **fst** and **snd** are the projections, with their respective types. Note that each field can refer to the preceding fields—in this case we have the type of **snd** depending on the value of **fst**. A constructor will be automatically generated, under the name of Prod.constr. Dually to data types, we will omit the type constructor prefix for record projections.

Following the bidirectionality of the system, we have that projections (the destructors of the record) infer the type, while the constructor gets checked:

$$\frac{\Gamma \vdash m \Downarrow A \qquad \Gamma \vdash n \Downarrow B\ m}{\Gamma \vdash \mathsf{Prod.constr}\ m\ n \Downarrow \mathsf{Prod}\ A\ B} \qquad \frac{\Gamma \vdash t \Uparrow \mathsf{Prod}\ A\ B}{\begin{array}{c}\Gamma \vdash \mathbf{fst}\ t \Uparrow A \\ \Gamma \vdash \mathbf{snd}\ t \Uparrow B\ (\mathbf{fst}\ t)\end{array}}$$

What we have defined here is equivalent to ITT's dependent products.

Definition (Elaboration for user defined types)**.** *Following the intuition given by the examples, the full elaboration machinery is presented Figure 6.*

Our elaboration is essentially a modification of Figure 9 of McBride & McKinna (2004a). However, our data types are not inductive families,[11] we do bidirectional type checking by treating constructors/destructors as syntax, and we have records.

Definition (Strict positivity)**.** *A inductive type declaration is* strictly positive *if recursive occurrences of the type we are defining do not appear embedded anywhere in the domain part of any function in the types for the data constructors.*

In data type declarations we allow recursive occurrences as long as they are strictly positive, which ensures the consistency of the theory. To achieve that we employing a syntactic check to make sure that this is the case—in fact the check is stricter than necessary for simplicity, given that we allow recursive occurrences only at the top level of data constructor arguments. For example a definition of the W type is accepted in Agda but rejected in Bertus. This is to make the eliminator generation simpler, and in practice it is seldom an impediment.

Without these precautions, we can easily derive any type with no recursion:

```
data Fix a = Fix (Fix a -> a) -- Negative occurrence of 'Fix a'
-- Term inhabiting any type 'a'
boom :: a
boom = (\f -> f (Fix f)) (\x -> (\(Fix f) -> f) x x)
```

See Dybjer (1991) for a more formal treatment of inductive definitions in ITT.

For what concerns records, recursive occurrences are disallowed. The reason for this choice is answered by the reason for the choice of having records at all: we need records to give the user types with η-laws for equality, as we saw in Section 4.2.1 and in the treatment of OTT in Section 5. If we tried to η-expand recursive data types, we would expand forever.

[11] See Section 9 for a brief description of inductive families.

syntax

$$\mathit{name} ::= \cdots \mid \mathsf{D} \mid \mathsf{D.c} \mid \mathsf{D.f}$$

syntax elaboration: $\mathit{decl} \triangleright \mathit{term} ::= \cdots$

$$\begin{array}{l} \underline{\mathsf{data}}\ \mathsf{D}\Delta\ \underline{\mathsf{where}}\ \{\cdots \mid \mathsf{c}_n : \Delta_n\} \\ \triangleright\ \ \mathit{term} ::= \cdots \mid \mathsf{D}\ \mathit{term}^{\Delta} \mid \cdots \mid \mathsf{D.c}_n\ \mathit{term}^{\Delta_n} \mid \mathsf{D.elim}\ \mathit{term} \end{array}$$

context elaboration: $\Gamma \vdash \mathit{decl} \triangleright \Gamma$

$$\frac{\begin{array}{c} \Gamma \vdash \Delta \to \mathsf{Type} \Uparrow \mathsf{Type} \qquad \mathsf{D} \notin \Gamma \\ \Gamma;\ \mathsf{D} : \Delta \to \mathsf{Type} \vdash \Delta; \Delta_i \to \mathsf{D}\ \delta \Uparrow \mathsf{Type} \quad (1 \le i \le n) \\ \text{For each } (x{:}A) \text{ in each } \Delta_i, \text{ if } \mathsf{D} \in A, \text{ then } A = \mathsf{D}\ \vec{t}. \end{array}}{\begin{array}{l} \Gamma \vdash \underline{\mathsf{data}}\ \mathsf{D}\Delta\ \underline{\mathsf{where}}\ \{\cdots \mid \mathsf{c}_n : \Delta_n\} \\ \quad \triangleright\ \Gamma;\ \mathsf{D} : \Delta \to \mathsf{Type};\ \cdots;\ \mathsf{D.c}_n : \Delta; \Delta_n \to \mathsf{D}\ \delta; \\ \qquad \mathsf{D.elim} : \Delta \to (x{:}\mathsf{D}\ \delta) \to \qquad\qquad \textbf{target} \\ \qquad\qquad (P{:}\mathsf{D}\ \delta \to \mathsf{Type}) \to \qquad\qquad \textbf{motive} \\ \qquad\qquad\quad \vdots \qquad\qquad\qquad\qquad \Big\} \ \textbf{methods} \\ \qquad\qquad (\Delta_n; \textsc{hyps}(P, \Delta_n) \to P\ (\mathsf{D.c}_n\ \delta_n)) \to \\ \qquad\qquad P\ x \end{array}}$$

$$\begin{array}{lrcl} \textbf{where} & \textsc{hyps}(P, \varepsilon) & \Longrightarrow & \varepsilon \\ & \textsc{hyps}(P, (r{:}\mathsf{D}\ \vec{t}); \Delta) & \Longrightarrow & (r'{:}P\ r); \textsc{hyps}(P, \Delta) \\ & \textsc{hyps}(P, (x{:}A); \Delta) & \Longrightarrow & \textsc{hyps}(P, \Delta) \end{array}$$

reduction elaboration: $\mathit{decl} \triangleright \Gamma \vdash \mathit{term} \rightsquigarrow \mathit{term}$

$$\underline{\mathsf{data}}\ \mathsf{D}\Delta\ \underline{\mathsf{where}}\ \{\cdots \mid \mathsf{c}_n : \Delta_n\}\ \triangleright\ \frac{\mathsf{D} : \Delta \to \mathsf{Type} \in \Gamma \qquad \mathsf{D.c}_i : \Delta; \Delta_i \to \mathsf{D}\ \delta \in \Gamma}{\Gamma \vdash \mathsf{D.elim}\ (\mathsf{D.c}_i\ \vec{t})\ P\ \vec{m} \rightsquigarrow m_i\ \vec{t}\ \textsc{recs}(P, \vec{m}, \Delta_i)}$$

$$\begin{array}{lrcl} \textbf{where} & \textsc{recs}(P, \vec{m}, \varepsilon) & \Longrightarrow & \varepsilon \\ & \textsc{recs}(P, \vec{m}, (r{:}\mathsf{D}\ \vec{A}); \Delta) & \Longrightarrow & (\mathsf{D.elim}\ r\ P\ \vec{m}); \textsc{recs}(P, \vec{m}, \Delta) \\ & \textsc{recs}(P, \vec{m}, (x{:}A); \Delta) & \Longrightarrow & \textsc{recs}(P, \vec{m}, \Delta) \end{array}$$

syntax elaboration: $\Gamma \vdash \mathit{decl} \triangleright \mathit{term} ::= \cdots$

$$\begin{array}{l} \Gamma \vdash \underline{\mathsf{record}}\ \mathsf{D}\Delta\ \underline{\mathsf{where}}\ \{\cdots, \mathbf{f}_n : F_n\} \\ \quad \triangleright\ \ \mathit{term} ::= \cdots \mid \mathsf{D}\ \mathit{term}^{\Delta} \mid \mathsf{D.constr}\ \mathit{term}^{n} \mid \cdots \mid \mathsf{D.}\mathbf{f}_n\ \mathit{term} \end{array}$$

context elaboration: $\Gamma \vdash \mathit{decl} \triangleright \Gamma$

$$\frac{\begin{array}{c} \Gamma \vdash \Delta \to \mathsf{Type} \Uparrow \mathsf{Type} \qquad \mathsf{D} \notin \Gamma \\ \Gamma; \Delta; (f_j : F_j)_{j=1}^{i-1} \vdash F_i \Uparrow \mathsf{Type} \quad (1 \le i \le n) \end{array}}{\begin{array}{l} \Gamma \vdash \underline{\mathsf{record}}\ \mathsf{D}\Delta\ \underline{\mathsf{where}}\ \{\cdots, \mathbf{f}_n : F_n\} \\ \quad \triangleright\ \Gamma;\ \mathsf{D} : \Delta \to \mathsf{Type};\ \cdots;\ \mathsf{D.}\mathbf{f}_n : \Delta \to (x{:}\mathsf{D}\ \delta) \to F_n[\mathbf{f}_i\ x / f_i]_{i=1}^{n-1}; \\ \qquad \mathsf{D.constr} : \Delta \to F_1 \to \cdots \to F_n \to \mathsf{D}\ \delta; \end{array}}$$

reduction elaboration: $\mathit{decl} \triangleright \Gamma \vdash \mathit{term} \rightsquigarrow \mathit{term}$

$$\underline{\mathsf{record}}\ \mathsf{D}\Delta\ \underline{\mathsf{where}}\ \{\cdots, \mathbf{f}_n : F_n\}\ \triangleright\ \frac{\mathsf{D} \in \Gamma}{\Gamma \vdash \mathsf{D.}\mathbf{f}_i\ (\mathsf{D.constr}\ \vec{t}) \rightsquigarrow t_i}$$

Figure 6: Elaboration for data types and records.

Definition (Bidirectional type checking for elaborated types)**.** *To implement bidirectional type checking for constructors and destructors, we store their types in full in the context, and then instantiate when due.*

typing: $\Gamma \vdash term \Updownarrow term$

$$\frac{\mathsf{D} : \Delta \to \mathsf{Type} \in \Gamma \qquad \mathsf{D.c} : \Delta; \Delta' \to \mathsf{D}\,\delta \in \Gamma \qquad \Delta'' = (\Delta; \Delta')\vec{A} \qquad \Gamma; \textsc{take}_{i-1}(\Delta'') \vdash t_i \Downarrow \textsc{ix}_i(\Delta'') \quad (1 \leq i \leq \Delta'')}{\Gamma \vdash \mathsf{D.c}\,\vec{t} \Downarrow \mathsf{D}\,\vec{A}}$$

$$\frac{\mathsf{D} : \Delta \to \mathsf{Type} \in \Gamma \qquad \mathsf{D.f} : \Delta; (x{:}\mathsf{D}\,\delta) \to F \qquad \Gamma \vdash t : \mathsf{D}\,\vec{A}}{\Gamma \vdash \mathsf{D.f}\,t \Uparrow (\Delta; (x{:}\mathsf{D}\,\delta) \to F)(\vec{A}; t)}$$

Note that for 0-ary type constructors, like $\mathbb{N}$, we do not need to check canonical terms: we can automatically infer that zero and suc n are of type $\mathbb{N}$. Bertus implements this measure, even if it is not shown in the typing rule for simplicity.

6.3.4 Why user defined types? Why eliminators?

The hardest design choice in developing Bertus was to decide whether user defined types should be included, and how to handle them. As we saw, while we can devise general structures like W, they are unsuitable both for for direct usage and 'mechanical' usage. Thus most theorem provers in the wild provide some means for the user to define structures tailored to specific uses.

Even if we take user defined types for granted, while there is not much debate on how to handle records, there are two broad schools of thought regarding the handling of data types:

Fixed points and pattern matching The road chosen by Agda and Coq. Functions are written like in Haskell—matching on the input and with explicit recursion. An external check on the recursive arguments ensures that they are decreasing, and thus that all functions terminate. This approach is the best in terms of user usability, but it is tricky to implement correctly.

Elaboration into eliminators The road chose by Bertus, and pioneered by the Epigram line of work. The advantage is that we can reduce every data type to simple definitions which guarantee termination and are simple to reduce and type. It is however more cumbersome to use than pattern matching, although McBride & McKinna (2004a) has shown how to implement an expressive pattern matching interface on top of a larger set of combinators of those provided by Bertus.

We can go ever further down this road and elaborate the declarations for data types themselves to a small set of primitives, so that our 'core' language will be very small and manageable (Dagand & McBride, 2012; Chapman *et al.* , 2010).

We chose the safer and easier to implement path, given the time constraints and the higher confidence of correctness. See also Section 9 for a brief overview of ways to extend or treat user defined types.

6.4 Cumulative hierarchy and typical ambiguity

Having a well founded type hierarchy is crucial if we want to retain consistency, otherwise we can break our type systems by proving bottom, as shown in Appendix B.3.

However, hierarchy as presented in section 3 is a considerable burden on the user, on various levels. Consider for example how we recovered disjunctions in Section 3.3.6: we

have a function that takes two Type_0 and forms a new Type_0. What if we wanted to form a disjunction containing something a Type_1, or Type_{42}? Our definition would fail us, since $\mathsf{Type}_1 : \mathsf{Type}_2$.

One way to solve this issue is a *cumulative* hierarchy, where $\mathsf{Type}_{l_1} : \mathsf{Type}_{l_2}$ iff $l_1 < l_2$. This way we retain consistency, while allowing for 'large' definitions that work on small types too.

Definition (Cumulativity for Bertus' base types)**.** *Figure 7 gives a formal definition of* cumulativity *for the base types. Similar measures can be taken for user defined types, withe the type living in the least upper bound of the levels where the types contained data live.*

For example we might define our disjunction to be

$$\_\vee\_ : \mathsf{Type}_{100} \to \mathsf{Type}_{100} \to \mathsf{Type}_{100}$$

And hope that Type_{100} will be large enough to fit all the types that we want to use with our disjunction. However, there are two problems with this. First, clumsiness of having to manually specify the size of types is still there. More importantly, if we want to use $\vee$ itself as an argument to other type-formers, we need to make sure that those allow for types at least as large as Type_{100}.

A better option is to employ a mechanised version of what Russell called *typical ambiguity*: we let the user live under the illusion that $\mathsf{Type} : \mathsf{Type}$, but check that the statements about types are consistent under the hood. Bertus implements this following the plan given by Huet (1988). See also Harper & Pollack (1991) for a published reference, although describing a more complex system allowing for both explicit and explicit hierarchy at the same time.

We define a partial ordering on the levels, with both weak ($\leq$) and strong ($<$) constraints, the laws governing them being the same as the ones governing $<$ and $\leq$ for the natural numbers. Each occurrence of Type is decorated with a unique reference. We keep a set of constraints regarding the ordering of each occurrence of Type, each represented by its unique reference. We add new constraints as we type check, generating new references when needed.

For example, when type checking the type $\mathsf{Type}\, r_1$, where r_1 denotes the unique reference assigned to that term, we will generate a new fresh reference and return the type $\mathsf{Type}\, r_2$, adding the constraint $r_1 < r_2$ to the set. When type checking $\Gamma \vdash (x{:}A) \to B$, if $\Gamma \vdash A : \mathsf{Type}\, r_1$ and $\Gamma; x : B \vdash B : \mathsf{Type}\, r_2$; we will generate new reference r and add $r_1 \leq r$ and $r_2 \leq r$ to the set.

If at any point the constraint set becomes inconsistent, type checking fails. Moreover, when comparing two Type terms—during the process of deciding definitional equality for two terms—we equate their respective references with two $\leq$ constraints. Implementation details are given in Section 7.3.

cumulativity: $\Gamma \vdash term \preceq term$

$$\frac{\Gamma \vdash A \cong B}{\Gamma \vdash A \preceq B} \qquad \frac{}{\Gamma \vdash \mathsf{Type}_l \preceq \mathsf{Type}_{l+1}} \qquad \frac{\Gamma \vdash A \preceq B \qquad \Gamma \vdash B \preceq C}{\Gamma \vdash A \preceq C}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash A_1 \cong A_2 \qquad \Gamma; x : A_1 \vdash B_1 \preceq B_2}{\Gamma (x{:}A_1) \to B_1 \preceq (x{:}A_2) \to B_2}$$

Figure 7: Cumulativity rules for base types in Bertus, plus a 'conversion' rule for cumulative types.

Another more flexible but also more verbose alternative is the one chosen by Agda, where levels can be quantified so that the relationship between arguments and result in type formers can be explicitly expressed:

$$\_\vee\_ : (l_1\ l_2 : \mathsf{Level}) \to \mathsf{Type}_{l_1} \to \mathsf{Type}_{l_2} \to \mathsf{Type}_{l_1 \sqcup l_2}$$

Inference algorithms to automatically derive this kind of relationship are currently subject of research. We choose a less flexible but more concise way, since it is easier to implement and better understood.

6.5 Observational equality, Bertus style

There are two correlated differences between Bertus and the theory used to present OTT. The first is that in Bertus we have a type hierarchy, which lets us, for example, abstract over types. The second is that we let the user define inductive types and records.

Reconciling propositions for OTT and a hierarchy had already been investigated by Conor McBride,[12] and we follow some of his suggestions, with some innovation. Most of the dirty work, as an extension of elaboration, is to handle reduction rules and coercions for data types—both type constructors and data constructors.

6.5.1 The Bertus prelude, and Propositions

Before defining Prop, we define some basic types inside Bertus, as the target for the Prop decoder.

Definition (Bertus' propositional prelude)**.**

$$\begin{array}{l}
\underline{\mathsf{data}}\ \mathsf{Empty}\ \underline{\mathsf{where}}\ \{\} \\
\mathbf{absurd} : (A{:}\mathsf{Type}) \to \mathsf{Empty} \to A \mapsto \\
\quad \lambda A\ bot \mapsto \mathsf{Empty}.\mathbf{elim}\ bot\ (\lambda_ \mapsto A) \\
\\
\underline{\mathsf{record}}\ \mathsf{Unit}\ \underline{\mathsf{where}}\ \{\} \\
\\
\underline{\mathsf{record}}\ \mathsf{Prod}\ (A\ B{:}\mathsf{Type})\ \underline{\mathsf{where}}\ \{\mathbf{fst} : A, \mathbf{snd} : B\}
\end{array}$$

Definition (Propositions and decoding)**.**

syntax

$$\begin{array}{rcl}
term & ::= & \cdots \mid [\![prop]\!] \\
prop & ::= & \bot \mid \top \mid prop \wedge prop \mid \forall x{:}term.\ prop
\end{array}$$

proposition decoding: $[\![term]\!] \rightsquigarrow term$

$$\begin{array}{ll}
[\![\bot]\!] \rightsquigarrow \mathsf{Empty} & [\![\ P \wedge Q\ ]\!] \rightsquigarrow \mathsf{Prod}\ [\![P]\!]\ [\![Q]\!] \\
[\![\top]\!] \rightsquigarrow \mathsf{Unit} & [\![\forall x{:}A.\ P]\!] \rightsquigarrow (x{:}A) \to [\![P]\!]
\end{array}$$

[12] See http://www.e-pig.org/epilogue/index.html?p=1098.html.

We will overload the $\wedge$ symbol to define 'nested' products, and π_n to project elements from them, so that

$$
\begin{aligned}
&A \wedge B = A \wedge (B \wedge \top) \\
&A \wedge B \wedge C = A \wedge (B \wedge (C \wedge \top)) \\
&\quad \vdots \\
&\pi_1 : [\![A \wedge B]\!] \to [\![A]\!] \\
&\pi_2 : [\![A \wedge B \wedge C]\!] \to [\![B]\!] \\
&\quad \vdots
\end{aligned}
$$

And so on, so that π_n will work with all products with at least than n elements. Logically a 0-ary $\wedge$ will correspond to $\top$.

6.5.2 Some OTT examples

Before presenting the direction that Bertus takes, let us consider two examples of use-defined data types, and the result we would expect given what we already know about OTT, assuming the same propositional equalities.

Product types Let us consider first the already mentioned dependent product, using the alternate name Σ[13] to avoid confusion with the Prod in the prelude:

$$\underline{\text{record}}\ \Sigma\ (A{:}\mathsf{Type})\ (B{:}A \to \mathsf{Type})\ \underline{\text{where}}\ \{\mathbf{fst} : A, \mathbf{snd} : B\ fst\}$$

First type-level equality. The result we want is

$$
\begin{aligned}
&\Sigma\ A_1\ B_1 = \Sigma\ A_2\ B_2 \rightsquigarrow \\
&\quad A_1 = A_2 \wedge \forall x_1{:}A_1.\, \forall x_2{:}A_2.\, (x_1{:}A_1) = (x_2{:}A_2) \Rightarrow B_1\ x_1 = B_2\ x_2
\end{aligned}
$$

The difference here is that in the original presentation of OTT the type binders are explicit, while here B_1 and B_2 are functions returning types. We can do this thanks to the type hierarchy, and this hints at the fact that heterogeneous equality will have to allow Type 'to the right of the colon'. Indeed, heterogeneous equalities involving abstractions over types will provide the solution to simplify the equality above.

If we take, just like we saw previously in OTT

$$
\begin{aligned}
&(f_1{:}(A_1{:}x_1) \to B_1) = (f_2{:}(A_2{:}x_2) \to B_2) \rightsquigarrow \\
&\quad \forall x_1{:}A_1.\, \forall x_2{:}A_2.\, (x_1{:}A_1) = (x_2{:}A_2) \Rightarrow (f_1\ x_1{:}B_1[x_1]) = (f_2\ x_2{:}B_2[x_2])
\end{aligned}
$$

Then we can simply have

$$
\begin{aligned}
&\Sigma\ A_1\ B_1 = \Sigma\ A_2\ B_2 \rightsquigarrow \\
&\quad A_1 = A_2 \wedge (B_1{:}A_1 \to \mathsf{Type}) = (B_2{:}A_2 \to \mathsf{Type})
\end{aligned}
$$

Which will reduce to precisely what we desire, but with an heterogeneous equalities relating types instead of values:

$$
\begin{aligned}
&A_1 = A_2 \wedge (B_1{:}A_1 \to \mathsf{Type}) = (B_2{:}A_2 \to \mathsf{Type}) \rightsquigarrow \\
&A_1 = A_2 \wedge \forall x_1{:}A_1.\, \forall x_2{:}A_2.\, (x_1{:}A_1) = (x_2{:}A_2) \Rightarrow (B_1\ x_1{:}\mathsf{Type}) = (B_2\ x_2{:}\mathsf{Type})
\end{aligned}
$$

[13] For extra confusion, 'dependent products' are often called 'dependent sums' in the literature, referring to the interpretation that identifies the first element as a 'tag' deciding the type of the second element, which lets us recover sum types (disjuctions), as we saw in Section 3.3.5. Thus, Σ.

If we pretend for the moment that those heterogeneous equalities were type equalities, things run smoothly. For what concerns coercions and quotation, things stay the same (apart from the fact that we apply to the second argument instead of substituting). We can recognise records such as Σ as such and employ projections in value equality and coercions; as to not impede progress if not necessary.

Lists Now for finite lists, which will give us a taste for data constructors:

$$\underline{\mathsf{data}}\ \mathsf{List}\ (A{:}\mathsf{Type})\ \underline{\mathsf{where}}\ \{\mathsf{nil} \mid \mathsf{cons}\ A\ (\mathsf{List}\ A)\}$$

Type equality is simple—we only need to compare the parameter:

$$\mathsf{List}\ A_1 = \mathsf{List}\ A_2 \rightsquigarrow A_1 = A_2$$

For coercions, we transport based on the constructor, recycling the proof for the inductive occurrence:

$$\begin{aligned}
&\mathbf{coe}\ (\mathsf{List}\ A_1)\ (\mathsf{List}\ A_2)\ Q\ \mathsf{nil} &&\rightsquigarrow \mathsf{nil}\\
&\mathbf{coe}\ (\mathsf{List}\ A_1)\ (\mathsf{List}\ A_2)\ Q\ (\mathsf{cons}\ m\ n) &&\rightsquigarrow\\
&\quad \mathsf{cons}\ (\mathbf{coe}\ A_1\ A_2\ Q\ m)\ (\mathbf{coe}\ (\mathsf{List}\ A_1)\ (\mathsf{List}\ A_2)\ Q\ n)
\end{aligned}$$

Value equality is unsurprising—we match the constructors, and return bottom for mismatches.

$$\begin{aligned}
&(\quad \mathsf{nil} \quad : \mathsf{List}\ A_1) = (\quad \mathsf{nil} \quad : \mathsf{List}\ A_2) \rightsquigarrow \top\\
&(\mathsf{cons}\ m_1\ n_1 : \mathsf{List}\ A_1) = (\mathsf{cons}\ m_2\ n_2 : \mathsf{List}\ A_2) \rightsquigarrow\\
&\quad (m_1{:}A_1) = (m_2{:}A_2) \wedge (n_1{:}\mathsf{List}\ A_1) = (n_2{:}\mathsf{List}\ A_2)\\
&(\quad \mathsf{nil} \quad : \mathsf{List}\ A_1) = (\mathsf{cons}\ m_2\ n_2 : \mathsf{List}\ A_2) \rightsquigarrow \bot\\
&(\mathsf{cons}\ m_1\ n_1 : \mathsf{List}\ A_1) = (\quad \mathsf{nil} \quad : \mathsf{List}\ A_2) \rightsquigarrow \bot
\end{aligned}$$

6.5.3 Only one equality

Given the examples above, a more 'flexible' heterogeneous equality must emerge, since of the fact that in Bertus we re-gain the possibility of abstracting and in general handling types in a way that was not possible in the original OTT presentation. Moreover, we found that the rules for value equality work well if used with user defined type abstractions—for example in the case of dependent products we recover the original definition with explicit binders, in a natural manner.

Definition (Propositions, coercions, coherence, equalities and equality reduction for Bertus)**.** *See Figure 8.*[14]

Definition (Type equality in Bertus)**.** *We define* $A = B$ *as an abbreviation for* $(A{:}\mathsf{Type}) = (B{:}\mathsf{Type})$.

In fact, we can drop a separate notion of type-equality, which will simply be served by $(A{:}\mathsf{Type}) = (B{:}\mathsf{Type})$. We shall still distinguish equalities relating types for hierarchical purposes. We exploit record to perform η-expansion. Moreover, given the nested $\wedge$s, values of data types with zero constructors (such as Empty) and records with zero destructors (such as Unit) will be automatically always identified as equal. As in the original OTT, and for the same reasons, we can take **coh** as axiomatic.

[14] We discovered a problem with the proposed model, see Appendix C for details.

syntax

$$\begin{array}{rl} \mathit{term} ::= & \cdots \mid \mathbf{coe}\ \mathit{term}\ \mathit{term}\ \mathit{term}\ \mathit{term} \mid \mathbf{coh}\ \mathit{term}\ \mathit{term}\ \mathit{term}\ \mathit{term} \\ \mathit{prop} ::= & \cdots \mid (\mathit{term}{:}\mathit{term}) = (\mathit{term}{:}\mathit{term}) \end{array}$$

typing: $\Gamma \vdash \mathit{term} \Leftrightarrow \mathit{term}$

$$\frac{\Gamma \vdash P \Downarrow [\![A = B]\!] \qquad \Gamma \vdash t \Downarrow A}{\Gamma \vdash \mathbf{coe}\ A\ B\ P\ t \Uparrow B} \qquad \frac{\Gamma \vdash P \Downarrow [\![A = B]\!] \qquad \Gamma \vdash t \Downarrow A}{\Gamma \vdash \mathbf{coh}\ A\ B\ P\ t \Uparrow [\![(t{:}A) = (\mathbf{coe}\ A\ B\ P\ t{:}B)]\!]}$$

propositions: $\Gamma \vdash \mathit{prop} : \mathsf{Prop}$

$$\frac{}{\Gamma \vdash \top : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \wedge Q : \mathsf{Prop}}$$

$$\Gamma \vdash \bot : \mathsf{Prop}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma; x : A \vdash P : \mathsf{Prop}}{\Gamma \vdash \forall x{:}A.\, P : \mathsf{Prop}} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash m : A \qquad \Gamma \vdash B : \mathsf{Type} \qquad \Gamma \vdash n : B}{\Gamma \vdash (m{:}A) = (n{:}B) : \mathsf{Prop}}$$

equality reduction: $\Gamma \vdash \mathit{prop} \rightsquigarrow \mathit{prop}$

$$\frac{}{\Gamma \vdash (\mathsf{Type}{:}\mathsf{Type}) = (\mathsf{Type}{:}\mathsf{Type}) \rightsquigarrow \top}$$

$$\frac{}{\Gamma \vdash ((x_1{:}A_1) \to B_1{:}\mathsf{Type}) = ((x_2{:}A_2) \to B_2{:}\mathsf{Type}) \rightsquigarrow A_2 = A_1 \wedge \forall x_2{:}A_2.\, \forall x_1{:}A_1.\, (x_2{:}A_2) = (x_1{:}A_1) \Rightarrow B_1[x_1] = B_2[x_2]}$$

$$\frac{}{\Gamma \vdash (f_1{:}(x_1{:}A_1) \to B_1) = (f_2{:}(x_2{:}A_2) \to B_2) \rightsquigarrow \forall x_1{:}A_1.\, \forall x_2{:}A_2.\, (x_1{:}A_1) = (x_2{:}A_2) \Rightarrow (f_1\ x_1{:}B_1[x_1]) = (f_2\ x_2{:}B_2[x_2])}$$

$$\frac{\mathsf{D} : \Delta \to \mathsf{Type} \in \Gamma}{\Gamma \vdash (\mathsf{D}\ \vec{A}{:}\mathsf{Type}) = (\mathsf{D}\ \vec{B}{:}\mathsf{Type}) \rightsquigarrow \bigwedge_{i=1}^{n} ((A_n{:}\textsc{head}(\Delta(A_1 \cdots A_{i-1}))) = (B_i{:}\textsc{head}(\Delta(B_1 \cdots B_{i-1}))))}$$

$$\frac{\textsc{datatype}(\mathsf{D}, \Gamma) \qquad \mathsf{D.c} : \Delta; \Delta' \to \mathsf{D}\ \delta \in \Gamma \qquad \Delta_A = (\Delta; \Delta')\vec{A} \qquad \Delta_B = (\Delta; \Delta')\vec{B}}{\Gamma \vdash (\mathsf{D.c}\ \vec{l}{:}\mathsf{D}\ \vec{A}) = (\mathsf{D.c}\ \vec{r}{:}\mathsf{D}\ \vec{B}) \rightsquigarrow \bigwedge_{i=1}^{n} ((m_i{:}\textsc{head}(\Delta_A(A_i \cdots A_{i-1}))) = (n_i{:}\textsc{head}(\Delta_B(B_i \cdots B_{i-1}))))}$$

$$\frac{\textsc{datatype}(\mathsf{D}, \Gamma)}{\Gamma \vdash (\mathsf{D.c}\ \vec{l}{:}\mathsf{D}\ \vec{A}) = (\mathsf{D.c'}\ \vec{r}{:}\mathsf{D}\ \vec{B}) \rightsquigarrow \bot}$$

$$\frac{\textsc{record}(\mathsf{D}, \Gamma) \qquad \mathsf{D.f}_i : \Delta; (x{:}\mathsf{D}\ \delta) \to F_i \in \Gamma}{\Gamma \vdash (l{:}\mathsf{D}\ \vec{A}) = (r{:}\mathsf{D}\ \vec{B}) \rightsquigarrow \bigwedge_{i=1}^{n} ((\mathsf{D.f}_1\ l{:}(\Delta; (x{:}\mathsf{D}\ \delta) \to F_i)(\vec{A}; l)) = (\mathsf{D.f}_i\ r{:}(\Delta; (x{:}\mathsf{D}\ \delta) \to F_i)(\vec{B}; r)))}$$

$$\frac{}{(m{:}A) = (n{:}B) \rightsquigarrow \bot \text{ if } A \text{ and } B \text{ are canonical types.}}$$

Figure 8: Propositions and equality reduction in Bertus. We assume the presence of datatype and record as operations on the context to recognise whether a user defined type is a data type or a record.

6.5.4 Coercions

For coercions the algorithm is messier and not reproduced here for lack of a decent notation—the details are hairy but uninteresting. To give an idea of the possible complications, let us conceive a type that showcases trouble not arising in the previous examples.

$$\begin{array}{l}\underline{\mathsf{data}}\ \mathsf{Max}\ (A{:}\mathbb{N} \to \mathsf{Type})\ (B{:}(x{:}\mathbb{N}) \to A\ x \to \mathsf{Type})\ (k{:}\mathbb{N})\ \underline{\mathsf{where}} \\ \quad \{\mathsf{max}\ (A\ k)\ (x{:}\mathbb{N})\ (a{:}A\ x)\ (B\ x\ a)\}\end{array}$$

For type equalities we will have

$$(\mathsf{Max}\ A_1\ B_1\ k_1{:}\mathsf{Type}) = (\mathsf{Max}\ A_2\ B_2\ k_2{:}\mathsf{Type}) \rightsquigarrow$$

$$\begin{array}{l}(A_1{:}\mathbb{N} \to \mathsf{Type}) = (A_2{:}\mathbb{N} \to \mathsf{Type}) \wedge \\ (B_1{:}(x{:}\mathbb{N}) \to A_1\ x \to \mathsf{Type}) = (B_2{:}(x{:}\mathbb{N}) \to A_2\ x \to \mathsf{Type}) \\ (k_1{:}\mathbb{N}) = (k_2{:}\mathbb{N})\end{array} \rightsquigarrow$$

$$\begin{array}{l}(\mathbb{N} = \mathbb{N} \wedge (\forall x_1\ x_2{:}\mathbb{N}.\ (x_1{:}\mathbb{N}) = (x_2{:}\mathbb{N}) \Rightarrow A_1\ x_1 = A_2\ x_2)) \wedge \\ (\mathbb{N} = \mathbb{N} \wedge \begin{pmatrix}\forall x_1\ x_2{:}\mathbb{N}.\ (x_1{:}\mathbb{N}) = (x_2{:}\mathbb{N}) \Rightarrow \\ (B_1\ x_1{:}A_1\ x_1 \to \mathsf{Type}) = (B_2\ x_2{:}A_2\ x_2 \to \mathsf{Type})\end{pmatrix}) \wedge \\ (k_1{:}\mathbb{N}) = (k_2{:}\mathbb{N})\end{array} \rightsquigarrow$$

$$\begin{array}{l}(\top \wedge (\forall x_1\ x_2{:}\mathbb{N}.\ (x_1{:}\mathbb{N}) = (x_2{:}\mathbb{N}) \Rightarrow A_1\ x_1 = A_2\ x_2)) \wedge \\ (\top \wedge \begin{pmatrix}\forall x_1\ x_2{:}\mathbb{N}.\ (x_1{:}\mathbb{N}) = (x_2{:}\mathbb{N}) \Rightarrow \\ \forall y_1{:}A_1\ x_1.\ \forall y_2{:}A_2\ x_2.\ (y_1{:}A_1\ x_1) = (y_2{:}A_2\ x_2) \Rightarrow \\ B_1\ x_1\ y_1 = B_2\ x_2\ y_2\end{pmatrix}) \wedge \\ (k_1{:}\mathbb{N}) = (k_2{:}\mathbb{N})\end{array}$$

The result, while looking complicated, is actually saying something simple—given equal inputs, the parameters for Max will return equal types. Moreover, we have evidence that the two k parameters are equal. When coercing, we need to mechanically generate one proof of equality for each argument, and then coerce:

$$\begin{array}{l}\mathbf{coe}\ (\mathsf{Max}\ A_1\ B_1\ k_1)\ (\mathsf{Max}\ A_2\ B_2\ k_2)\ Q\ (\mathsf{max}\ ak_1\ n_1\ a_1\ b_1) \rightsquigarrow \\ \quad \underline{\mathsf{let}}\ Q_{Ak} \mapsto \boxed{?} : [\![A_1\ k_1 = A_2\ k_2]\!] \\ \qquad ak_2 \mapsto \mathbf{coe}\ (A_1\ k_1)\ (A_2\ k_2)\ Q_{Ak}\ ak_1 : A_1\ k_2 \\ \qquad Q_{\mathbb{N}} \mapsto \boxed{?} : [\![\mathbb{N} = \mathbb{N}]\!] \\ \qquad n_2 \mapsto \mathbf{coe}\ \mathbb{N}\ \mathbb{N}\ Q_{\mathbb{N}}\ n_1 : \mathbb{N} \\ \qquad Q_A \mapsto \boxed{?} : [\![A_1\ n_1 = A_2\ n_2]\!] \\ \qquad a_2 \mapsto \mathbf{coe}\ (A_1\ n_1)\ (A_2\ n_2)\ Q_A\ : A_2\ n_2 \\ \qquad Q_B \mapsto \boxed{?} : [\![B_1\ n_1\ a_1 = B_1\ n_2\ a_2]\!] \\ \qquad b_2 \mapsto \mathbf{coe}\ (B_1\ n_1\ a_1)\ (B_2\ n_2\ a_2)\ Q_B\ : B_2\ n_2\ a_2 \\ \quad \underline{\mathsf{in}}\ \mathsf{max}\ ak_2\ n_2\ a_2\ b_2\end{array}$$

For equalities regarding types that are external to the data type we can derive a proof by reflexivity by invoking refl as defined in Section 5.2.2, and the instantiate arguments if we need too. In this case, for $\mathbb{N}$, we do not have any arguments. For equalities concerning arguments of the type constructor or already coerced arguments of the type constructor we have to refer to the right proof and use **coh**erence when due, which is where the technical

annoyance lies:

$$
\begin{array}{l}
\textbf{coe}\ (\mathsf{Max}\ A_1\ B_1\ k_1)\ (\mathsf{Max}\ A_2\ B_2\ k_2)\ Q\ (\mathsf{max}\ ak_1\ n_1\ a_1\ b_1) \rightsquigarrow \\
\quad \underline{\text{let}}\ Q_{Ak} \mapsto (\pi_2\ (\pi_1\ Q))\ k_1\ k_2\ (\pi_3\ Q) : [\![A_1\ k_1 = A_2\ k_2]\!] \\
\qquad ak_2 \mapsto \textbf{coe}\ (A_1\ k_1)\ (A_2\ k_2)\ Q_{Ak}\ ak_1 : A_1\ k_2 \\
\qquad Q_{\mathbb{N}} \mapsto \mathsf{refl}\ \mathbb{N} : [\![\mathbb{N} = \mathbb{N}]\!] \\
\qquad n_2 \mapsto \textbf{coe}\ \mathbb{N}\ \mathbb{N}\ Q_{\mathbb{N}}\ n_1 : \mathbb{N} \\
\qquad Q_A \mapsto (\pi_2\ (\pi_1\ Q))\ n_1\ n_2\ (\textbf{coh}\ \mathbb{N}\ \mathbb{N}\ Q_{\mathbb{N}}\ n_1) : [\![A_1\ n_1 = A_2\ n_2]\!] \\
\qquad a_2 \mapsto \textbf{coe}\ (A_1\ n_1)\ (A_2\ n_2)\ Q_A\ : A_2\ n_2 \\
\qquad Q_B \mapsto (\pi_2\ (\pi_2\ Q))\ n_1\ n_2\ Q_{\mathbb{N}}\ a_1\ a_2\ (\textbf{coh}\ (A_1\ n_1)\ (A_2\ n_2)\ Q_A\ a_1) : [\![B_1\ n_1\ a_1 = B_1\ n_2\ a_2]\!] \\
\qquad b_2 \mapsto \textbf{coe}\ (B_1\ n_1\ a_1)\ (B_2\ n_2\ a_2)\ Q_B\ : B_2\ n_2\ a_2 \\
\quad \underline{\text{in}}\ \mathsf{max}\ ak_2\ n_2\ a_2\ b_2
\end{array}
$$

6.5.5 Prop and the hierarchy

We shall have, at each universe level, not only a Type_l but also a Prop_l. Where will propositions placed in the type hierarchy? The main indicator is the decoding operator, since it converts into things that already live in the hierarchy. For example, if we have

$$[\![\mathbb{N} \to \mathsf{Bool} = \mathbb{N} \to \mathsf{Bool}]\!] \rightsquigarrow \top \wedge ((x\ y : \mathbb{N}) \to \top \to \top)$$

we will better make sure that the 'to be decoded' is at level compatible (read: larger) with its reduction. In the example above, we will have that proposition to be at least as large as the type of $\mathbb{N}$, since the reduced proof will abstract over it. Pretending that we had explicit, non cumulative levels, it would be tempting to have

$$\frac{\Gamma \vdash Q : \mathsf{Prop}_l}{\Gamma \vdash [\![Q]\!] : \mathsf{Type}_l} \qquad \frac{\Gamma \vdash A : \mathsf{Type}_l \qquad \Gamma \vdash B : \mathsf{Type}_l}{\Gamma \vdash (A{:}\mathsf{Type}_l) = (B{:}\mathsf{Type}_l) : \mathsf{Prop}_l}$$

$\bot$ and $\top$ living at any level, $\wedge$ and $\forall$ following rules similar to the ones for $\times$ and $\to$ in Section 3. However, we need to be careful with value equality since for example we have that

$$[\![(f_1{:}(x_1{:}A_1) \to B_1) = (f_2{:}(x_2{:}A_2) \to B_2)]\!] \rightsquigarrow (x_1{:}A_1) \to (x_2{:}A_2) \to \cdots$$

where the proposition decodes into something of at least type Type_l, where $A_l : \mathsf{Type}_l$ and $B_l : \mathsf{Type}_l$. We can resolve this tension by making all equalities larger:

$$\frac{\Gamma \vdash m : A \qquad \Gamma \vdash A : \mathsf{Type}_l \qquad \Gamma \vdash n : B \qquad \Gamma \vdash B : \mathsf{Type}_l}{\Gamma \vdash (m{:}A) = (m{:}A) : \mathsf{Prop}_l}$$

This is disappointing, since type equalities will be needlessly large: $[\![(A{:}\mathsf{Type}_l) = (B{:}\mathsf{Type}_l)]\!] : \mathsf{Type}_{l+1}$.

However, considering that our theory is cumulative, we can do better. Assuming rules for Prop cumulativity similar to the ones for Type, we will have (with the conversion rule reproduced as a reminder):

$$\frac{\Gamma \vdash A \preceq B \qquad \Gamma \vdash t : A}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash A : \mathsf{Type}_l \qquad \Gamma \vdash B : \mathsf{Type}_l}{\Gamma \vdash (A{:}\mathsf{Type}_l) = (B{:}\mathsf{Type}_l) : \mathsf{Prop}_l}$$

$$\frac{\Gamma \vdash m : A \qquad \Gamma \vdash A : \mathsf{Type}_l \qquad \Gamma \vdash n : B \qquad \Gamma \vdash B : \mathsf{Type}_l \qquad A \text{ and } B \text{ are not } \mathsf{Type}_{l'}}{\Gamma \vdash (m{:}A) = (m{:}A) : \mathsf{Prop}_l}$$

That is, we are small when we can (type equalities) and large otherwise. This would not work in a non-cumulative theory because subject reduction would not hold. Consider for instance

$$(\mathbb{N}\text{:}\mathbf{If}\,\mathsf{true}\,\mathbf{Then}\,\mathsf{Type}_0\,\mathbf{Else}\,\mathsf{Type}_0) = (\mathsf{Bool}\text{:}\mathbf{If}\,\mathsf{true}\,\mathbf{Then}\,\mathsf{Type}_0\,\mathbf{Else}\,\mathsf{Type}_0) : \mathsf{Prop}_1$$

which reduces to

$$(\mathbb{N}\text{:}\mathsf{Type}_0) = (\mathsf{Bool}\text{:}\mathsf{Type}_0) : \mathsf{Prop}_0$$

We need members of Prop_0 to be members of Prop_1 too, which will be the case with cumulativity. This buys us a cheap type level equality without having to replicate functionality with a dedicated construct.

6.5.6 Quotation and definitional equality

Now we can give an account of definitional equality, by explaining how to perform quotation (as defined in Section 4.2.1) towards the goal described in Section 5.3.

We want to:

- Perform η-expansion on functions and records.
- As a consequence of the previous point, identify all records with no projections as equal, since they will have only one element.
- Identify all members of types with no constructors (and thus no elements) as equal.
- Identify all equivalent proofs as equal—with 'equivalent proof' we mean those proving the same propositions.
- Advance coercions working across definitionally equal types.

Towards these goals and following the intuition between bidirectional type checking we define two mutually recursive functions, one quoting canonical terms against their types (since we need the type to type check canonical terms), one quoting neutral terms while recovering their types.

Definition (Quotation for Bertus)**.** *The full procedure for quotation is shown in Figure 9.*

We $\boxed{\text{box}}$ the neutral proofs and neutral members of empty types, following the notation in Altenkirch *et al.* (2007), and we make use of $\cong_{\square}$ which compares terms syntactically up to α-renaming, but also up to equivalent proofs: we consider all boxed content as equal.

Our quotation will work on normalised terms, so that all defined values will have been replaced. Moreover, we match on data type eliminators and all their arguments, so that $\mathbb{N}.\mathbf{elim}\ m\ P\ \vec{n}$ will stand for $\mathbb{N}.\mathbf{elim}$ applied to the scrutinised $\mathbb{N}$, the predicate, and the two cases. This measure can be easily implemented by checking the head of applications and 'consuming' the needed terms. Thus, we gain proof irrelevance, and not only for a more useful definitional equality, but also for example to eliminate all propositional content when compiling.

6.5.7 Why Prop?

It is worth to ask if Prop is needed at all. It is perfectly possible to have the type checker identify propositional types automatically, and in fact in some sense we already do during equality reduction and quotation. However, this has the considerable disadvantage that we

canonical quotation: $\textsc{quote}{\Downarrow}(\Gamma, term : term) \Longrightarrow term$

$$
\begin{array}{ll}
\textsc{quote}{\Downarrow}(\Gamma,\ t : \mathsf{D}\ \vec{A} &) \mid \textsc{empty}(\Gamma, \mathsf{D}) \Longrightarrow \boxed{t} \\
\textsc{quote}{\Downarrow}(\Gamma,\ t : \mathsf{D}\ \vec{A} &) \mid \textsc{record}(\Gamma, \mathsf{D}) \Longrightarrow \mathsf{D.constr} \cdots \textsc{quote}{\Downarrow}(\Gamma, \mathsf{D.f}_n : (\Gamma(\mathsf{D.f}_n))(\vec{A}; t)) \\
\textsc{quote}{\Downarrow}(\Gamma,\ \mathsf{D.c}\ \vec{t} : \mathsf{D}\ \vec{A} &) \Longrightarrow \cdots \\
\textsc{quote}{\Downarrow}(\Gamma,\ f : (x{:}A) \to B) & \Longrightarrow \lambda x \mapsto \textsc{quote}{\Downarrow}(\Gamma; x : A, f\ x : B) \\
\textsc{quote}{\Downarrow}(\Gamma,\ p : [\![P]\!] &) \Longrightarrow \boxed{p} \\
\textsc{quote}{\Downarrow}(\Gamma,\ t : A &) \Longrightarrow t' \textbf{ where } t' : \_ = \textsc{quote}{\Uparrow}(\Gamma, t)
\end{array}
$$

neutral quotation: $\textsc{quote}{\Uparrow}(\Gamma, term) \Longrightarrow term : term$

$$
\begin{array}{ll}
\textsc{quote}{\Uparrow}(\Gamma,\ x &) \Longrightarrow x : \Gamma(x) \\
\textsc{quote}{\Uparrow}(\Gamma,\ \mathsf{Type} &) \Longrightarrow \mathsf{Type} : \mathsf{Type} \\
\textsc{quote}{\Uparrow}(\Gamma,\ (x{:}A) \to B &) \Longrightarrow (x{:}\textsc{quote}{\Uparrow}(\Gamma, A)) \to \textsc{quote}{\Uparrow}(\Gamma; x : A, B) : \mathsf{Type} \\
\textsc{quote}{\Uparrow}(\Gamma,\ \mathsf{D}\ \vec{A} &) \Longrightarrow \mathsf{D} \cdots \textsc{quote}{\Downarrow}(\Gamma, \textsc{head}((\Gamma(\mathsf{D}))(A_1 \cdots A_{n-1}))) : \mathsf{Type} \\
\textsc{quote}{\Uparrow}(\Gamma,\ [\![(m{:}A) = (n{:}B)]\!] &) \Longrightarrow \\
\quad [\![(\textsc{quote}{\Downarrow}(\Gamma, m : A){:}A') = (\textsc{quote}{\Downarrow}(\Gamma, n : B){:}B')]\!] & : \mathsf{Type} \\
\quad \textbf{where } A' : \_ = \textsc{quote}{\Uparrow}(\Gamma, A) & \\
\qquad\quad B' : \_ = \textsc{quote}{\Uparrow}(\Gamma, B) & \\
\textsc{quote}{\Uparrow}(\Gamma,\ \mathsf{D.f}\ t &) \mid \textsc{record}(\Gamma, \mathsf{D}) \Longrightarrow \mathsf{D.f}\ t' : (\Gamma(\mathsf{D.f}))(\vec{A}; t) \\
\quad \textbf{where } t' : \mathsf{D}\ \vec{A} = \textsc{quote}{\Uparrow}(\Gamma, t) & \\
\textsc{quote}{\Uparrow}(\Gamma,\ \mathsf{D.elim}\ t\ P &) \mid \textsc{empty}(\Gamma, \mathsf{D}) \Longrightarrow \mathsf{D.elim}\ \boxed{t}\ \textsc{quote}{\Uparrow}(\Gamma, P) : P\ t \\
\textsc{quote}{\Uparrow}(\Gamma,\ \mathsf{D.elim}\ m\ P\ \vec{n} &) \Longrightarrow \mathsf{D.elim}\ m'\ \textsc{quote}{\Uparrow}(\Gamma, P) \cdots : P\ m \\
\quad \textbf{where } m' : \mathsf{D}\ \vec{A} = \textsc{quote}{\Uparrow}(\Gamma, m) & \\
\textsc{quote}{\Uparrow}(\Gamma,\ f\ t &) \Longrightarrow f'\ \textsc{quote}{\Downarrow}(\Gamma, t : A) : B[t/x] \\
\quad \textbf{where } f' : (x{:}A) \to B = \textsc{quote}{\Uparrow}(\Gamma, f) & \\
\textsc{quote}{\Uparrow}(\Gamma,\ \textbf{coe}\ A\ B\ Q\ t &) \mid \textsc{quote}{\Uparrow}(\Gamma, A) \cong_{\square} \textsc{quote}{\Uparrow}(\Gamma, B) \Longrightarrow \textsc{quote}{\Uparrow}(\Gamma, t) \\
\textsc{quote}{\Uparrow}(\Gamma,\ \textbf{coe}\ A\ B\ Q\ t &) \Longrightarrow \textbf{coe}\ \textsc{quote}{\Uparrow}(\Gamma, A)\ \textsc{quote}{\Uparrow}(\Gamma, B)\ \boxed{Q}\ \textsc{quote}{\Uparrow}(\Gamma, t)
\end{array}
$$

Figure 9: Quotation in Bertus. Along the already used record meta-operation on the context we make use of empty, which checks if a certain type constructor has zero data constructor. The 'data constructor' cases for non-record, non-empty, data types are omitted for brevity.

can never identify abstracted variables[15] of type Type as Prop, thus forbidding the user to talk about Prop explicitly.

This is a considerable impediment, for example when implementing *quotient types*. With quotients, we let the user specify an equivalence class over a certain type, and then exploit this in various way—crucially, we need to be sure that the equivalence given is propositional, a fact which prevented the use of quotients in dependent type theories (Jacobs, 1994).

[15] And in general neutral terms, although we currently do not have neutral propositions apart from equalities on neutral terms.

7 | Bertus: the practice

It's alive!

Henry Frankenstein

The codebase consists of around 2500 lines of Haskell,[16] as reported by the `cloc` utility.

We implement the type theory as described in Section 6. The author learnt the hard way the implementation challenges for such a project, and ran out of time while implementing observational equality. While the constructs and typing rules are present, the machinery to make it happen (equality reduction, coercions, quotation, etc.) is not present yet.

This considered, everything else presented in Section 6 is implemented and working well—and in fact all the examples presented in this thesis, apart from the ones that are equality related, have been encoded in Bertus in the Appendix. Moreover, given the detailed plan in the previous section, finishing off should not prove too much work.

The interaction with the user takes place in a loop living in and updating a context of Bertus declarations, which presents itself as in Figure 10. Files with lists of declarations can also be loaded. The REPL is a available both as a command-line application and in a web interface, which is available at `bertus.mazzo.li`.

A REPL cycle starts with the user inputting a Bertus declaration or another REPL command, which then goes through various stages that can end up in a context update, or in failures of various kind. The process is described diagrammatically in figure 11.

Parse In this phase the text input gets converted to a sugared version of the core language. For example, we accept multiple arguments in arrow types and abstractions, and we represent variables with names, while as we will see in Section 7.2 the final term types uses a *nameless* representation.

Desugar The sugared declaration is converted to a core term. Most notably we go from names to nameless.

[16] The full source code is available under the GPL3 license at `https://github.com/bitonic/kant`. 'Kant' was a previous incarnation of the software, and the name remained.

```
B E R T U S
Version 0.0, made in London, year 2013.
>>> :h
<decl>     Declare value/data type/record
:t <term>  Typecheck
:e <term>  Normalise
:p <term>  Pretty print
:l <file>  Load file
:r <file>  Reload file (erases previous environment)
:i <name>  Info about an identifier
:q         Quit
>>> :l data/samples/good/common.ka
OK
>>> :e plus three two
suc (suc (suc (suc (suc zero))))
>>> :t plus three two
Type: Nat
```

Figure 10: A sample run of the Bertus prompt.

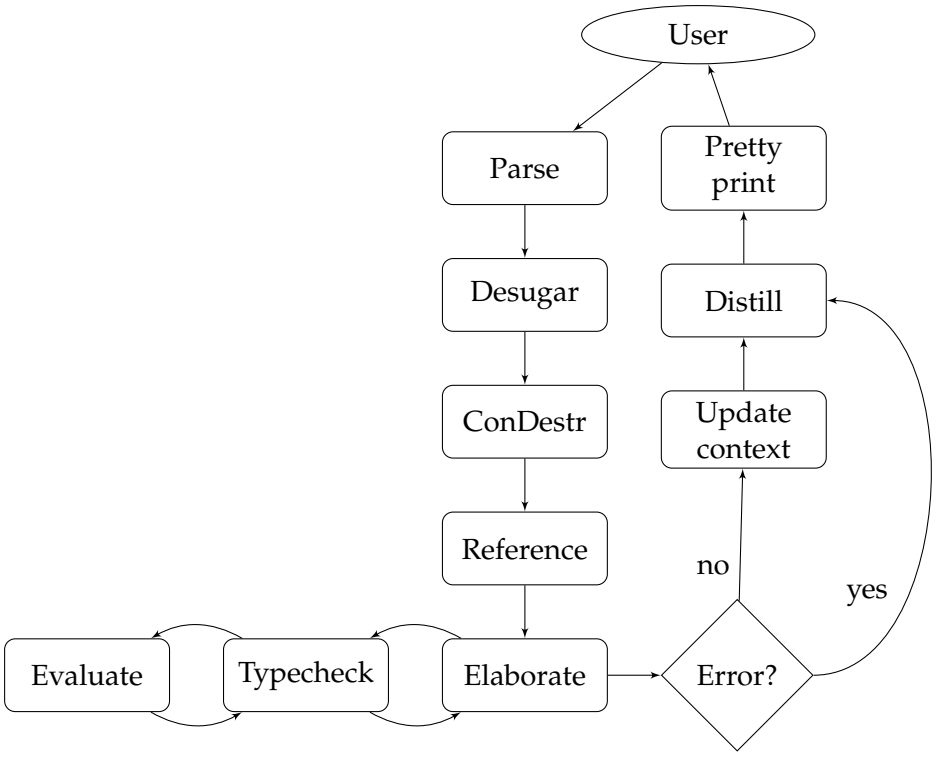


Figure 11: High level overview of the life of a BERTUS prompt cycle.

ConDestr Short for 'Constructors/Destructors', converts applications of data destructors and constructors to a special form, to perform bidirectional type checking.

Reference Occurrences of Type get decorated by a unique reference, which is necessary to implement the type hierarchy check.

Elaborate/Typecheck/Evaluate **Elaboration** converts the declaration to some context items, which might be a value declaration (type and body) or a data type declaration (constructors and destructors). This phase works in tandem with **Type checking**, which in turns needs to **Evaluate** terms.

Distill and report the result. 'Distilling' refers to the process of converting a core term back to a sugared version that we can show to the user. This can be necessary both to display errors including terms or to display result of evaluations or type checking that the user has requested. Among the other things in this stage we go from nameless back to names by recycling the names that the user used originally, as to fabricate a term which is as close as possible to what it originated from.

Pretty print Format the terms in a nice way, and display them to the user.

Here we will review only a sampling of the more interesting implementation challenges present when implementing an interactive theorem prover.

7.1 Syntax

The syntax of BERTUS is presented in Figure 12. Examples showing the usage of most of the constructs—excluding the OTT-related ones—are present in Appendices B.1.2, B.2, and B.3; plus a tutorial in Section 8.1. The syntax has grown organically with the needs of the language, and thus is not very sophisticated. The grammar is specified in and processed

by the `happy` parser generator for Haskell.[17] Tokenisation is performed by a simple hand written lexer.

7.2 Term representation

7.2.1 Naming and substituting

Perhaps surprisingly, one of the most difficult challenges in implementing a theory of the kind presented is choosing a good data type for terms, and specifically handling substitutions in a sane way.

There are two broad schools of thought when it comes to naming variables, and thus substituting:

Nameful Bound variables are represented by some enumerable data type, just as we have described up to now, starting from Section 2.1. The problem is that avoiding name capturing is a nightmare, both in the sense that it is not performant—given that we need to rename rename substitute each time we 'enter' a binder—but most importantly given the fact that in even slightly more complicated systems it is very hard to get right, even for experts.

One of the sore spots of explicit names is comparing terms up to α-renaming, which again generates a huge amounts of substitutions and requires special care.

Nameless We can capture the relationship between variables and their binders, by getting rid of names altogether, and representing bound variables with an index referring to the 'binding' structure, a notion introduced by de Bruijn (1972). Usually 0 represents the variable bound by the innermost binding structure, 1 the second-innermost, and so on. For instance with simple abstractions we might have

$$\lambda\,(\lambda\,0\,(\lambda\,0))\,(\lambda\,1\,0) : ((A \to A) \to B) \to B, \text{ which corresponds to}$$
$$\lambda f \mapsto (\lambda g \mapsto g\,(\lambda x \mapsto x))\,(\lambda x \mapsto f\,x) : ((A \to A) \to B) \to B$$

While this technique is obviously terrible in terms of human usability,[18] it is much more convenient as an internal representation to deal with terms mechanically—at least in simple cases. α-renaming ceases to be an issue, and term comparison is purely syntactical.

Nonetheless, more complex constructs such as pattern matching put some strain on the indices and many systems end up using explicit names anyway.

In the past decade or so advancements in the Haskell's type system and in general the spread new programming practices have made the nameless option much more amenable. Bertus thus takes the nameless path through the use of Edward Kmett's excellent `bound` library.[19] We describe the advantages of `bound`'s approach, but also its pitfalls in the previously relatively unknown territory of dependent types—`bound` being created mostly to handle more simply typed systems.

`bound` builds on the work of Bird & Paterson (1999), who suggested to parametrising the term type over the type of the variables, and 'nest' the type each time we enter a scope. If we wanted to define a term for the untyped λ-calculus, we might have

```
-- A type with no members.
data Empty
```

[17] Available at `http://www.haskell.org/happy`.
[18] With some people going as far as defining it akin to an inverse Turing test.
[19] Available at `http://hackage.haskell.org/package/bound`.

A name, in regexp notation.

⟨*name*⟩ ::= `[a-zA-Z] [a-zA-Z0-9'_-]*`

A binder might or might not (_) bind a name.

⟨*binder*⟩ ::= `_` | ⟨*name*⟩

A series of typed bindings.

⟨*telescope*⟩ ::= (`[` ⟨*binder*⟩ `:` ⟨*term*⟩ `]`)∗

Terms, including propositions.

⟨*term*⟩	::= ⟨*name*⟩	A variable.
	| `*`	Type.
	| `{|` ⟨*term*⟩∗ `|}`	Type holes.
	| `Prop`	Prop.
	| `Top` | `Bot`	⊤ and ⊥.
	| ⟨*term*⟩ `/\` ⟨*term*⟩	Conjuctions.
	| `[|` ⟨*term*⟩ `|]`	Proposition decoding.
	| `coe` ⟨*term*⟩ ⟨*term*⟩ ⟨*term*⟩ ⟨*term*⟩	Coercion.
	| `coh` ⟨*term*⟩ ⟨*term*⟩ ⟨*term*⟩ ⟨*term*⟩	Coherence.
	| `(` ⟨*term*⟩ `:` ⟨*term*⟩ `) = (` ⟨*term*⟩ `:` ⟨*term*⟩ `)`	Heterogeneous equality.
	| `(` ⟨*compound*⟩ `)`	Parenthesised term.
⟨*compound*⟩	::= `\` ⟨*binder*⟩∗ `=>` ⟨*term*⟩	Untyped abstraction.
	| `\` ⟨*telescope*⟩ `:` ⟨*term*⟩ `=>` ⟨*term*⟩	Typed abstraction.
	| `forall` ⟨*telescope*⟩ ⟨*term*⟩	Universal quantification.
	| ⟨*arr*⟩	
⟨*arr*⟩	::= ⟨*telescope*⟩ `->` ⟨*arr*⟩	Dependent function.
	| ⟨*term*⟩ `->` ⟨*arr*⟩	Non-dependent function.
	| ⟨*term*⟩+	Application.

Typed names.

⟨*typed*⟩ ::= ⟨*name*⟩ `:` ⟨*term*⟩

Declarations.

⟨*decl*⟩ ::= ⟨*value*⟩ | ⟨*abstract*⟩ | ⟨*data*⟩ | ⟨*record*⟩

Defined values. The telescope specifies named arguments.

⟨*value*⟩ ::= ⟨*name*⟩ ⟨*telescope*⟩ `:` ⟨*term*⟩ `=>` ⟨*term*⟩

Abstracted variables.

⟨*abstract*⟩ ::= `postulate` ⟨*typed*⟩

Data types, and their constructors.

⟨*data*⟩ ::= `data` ⟨*name*⟩ `:` ⟨*telescope*⟩ `->` `*` `=>` `{` ⟨*constrs*⟩ `}`

⟨*constrs*⟩ ::= ⟨*typed*⟩
| ⟨*typed*⟩ `|` ⟨*constrs*⟩

Records, and their projections. The ⟨*name*⟩ before the projections is the constructor name.

⟨*record*⟩ ::= `record` ⟨*name*⟩ `:` ⟨*telescope*⟩ `->` `*` `=>` ⟨*name*⟩ `{` ⟨*projs*⟩ `}`

⟨*projs*⟩ ::= ⟨*typed*⟩
| ⟨*typed*⟩ `,` ⟨*projs*⟩

Figure 12: Bertus' syntax. The non-terminals are marked with ⟨angle brackets⟩ for greater clarity. The syntax in the implementation is actually more liberal, for example giving the possibility of using arrow types directly in constructor/projection declarations.
Additionally, we give the user the possibility of using Unicode characters instead of their ASCII counterparts, e.g. → in place of `->`, λ in place of `\`, etc.

```
data Var v = Bound | Free v

data Tm v
    = V v                -- Bound variable
    | App (Tm v) (Tm v) -- Term application
    | Lam (Tm (Var v))  -- Abstraction
```

Closed terms would be of type `Tm Empty`, so that there would be no occurrences of `V`. However, inside an abstraction, we can have `V Bound`, representing the bound variable, and inside a second abstraction we can have `V Bound` or `V (Free Bound)`. Thus the term

$$\lambda x \mapsto \lambda y \mapsto x$$

can be represented as

```
-- The top level term is of type ‘Tm Empty’.
-- The inner term ‘Lam (Free Bound)’ is of type ‘Tm (Var Empty)’.
-- The second inner term ‘V (Free Bound)’ is of type ‘Tm (Var (Var
-- Empty))’.
Lam (Lam (V (Free Bound)))
```

This allows us to reflect the ‘nestedness’ of a type at the type level, and since we usually work with functions polymorphic on the parameter `v` it’s very hard to make mistakes by putting terms of the wrong nestedness where they do not belong.

Even more interestingly, the substitution operation is perfectly captured by the `>>=` (bind) operator of the `Monad` type class:

```
class Monad m where
  return :: m a
  (>>=)  :: m a -> (a -> m b) -> m b

instance Monad Tm where
  -- ‘return’ing turns a variable into a ‘Tm’
  return = V

  -- ‘t >>= f’ takes a term ‘t’ and a mapping from variables to terms
  -- ‘f’ and applies ‘f’ to all the variables in ‘t’, replacing them
  -- with the mapped terms.
  V v     >>= f = f v
  App m n >>= f = App (m >>= f) (n >>= f)

  -- ‘Lam’ is the tricky case: we modify the function to work with bound
  -- variables, so that if it encounters ‘Bound’ it leaves it untouched
  -- (since the mapping refers to the outer scope); if it encounters a
  -- free variable it asks ‘f’ for the term and then updates all the
  -- variables to make them refer to the outer scope they were meant to
  -- be in.
  Lam s   >>= f = Lam (s >>= bump)
    where bump Bound    = return Bound
          bump (Free v) = f v >>= V . Free
```

With this in mind, we can define functions which will not only work on `Tm`, but on any `Monad`!

```
-- Replaces free variable ‘v’ with ‘m’ in ‘n’.
subst :: (Eq v, Monad m) => v -> m v -> m v -> m v
subst v m n = n >>= \v' -> if v == v' then m else return v'

-- Replace the variable bound by ‘s’ with term ‘t’.
inst :: Monad m => m v -> m (Var v) -> m v
inst t s = s >>= \v -> case v of
                           Bound   -> t
                           Free v' -> return v'
```

The beauty of this technique is that with a few simple functions we have defined all the core operations in a general and 'obviously correct' way, with the extra confidence of having the type checker looking our back. For what concerns term equality, we can just ask the H Haskell compiler to derive the instance for the `Eq` type class and since we are nameless that will be enough (modulo fancy quotation).

Moreover, if we take the top level term type to be `Tm String`, we get a representation of terms with top-level definitions; where closed terms contain only `String` references to said definitions—see also McBride & McKinna (2004b).

What are then the pitfalls of this seemingly invincible technique? The most obvious impediment is the need for polymorphic recursion. Functions traversing terms parameterized by the variable type will have types such as

```
-- Infer the type of a term, or return an error.
infer :: Tm v -> Either Error (Tm v)
```

When traversing under a `Scope` the parameter changes from `v` to `Var v`, and thus if we do not specify the type for our function explicitly inference will fail—type inference for polymorphic recursion being undecidable (Henglein, 1993). This causes some annoyance, especially in the presence of many local definitions that we would like to leave untyped.

But the real issue is the fact that giving a type parameterized over a variable—say `m v`—a `Monad` instance means being able to only substitute variables for values of type `m v`. This is a considerable inconvenience. Consider for instance the case of telescopes, which are a central tool to deal with contexts and other constructs. In Haskell we can give them a faithful representation with a data type along the lines of

```
data Tele m v = Empty (m v) | Bind (m v) (Tele m (Var v))
type TeleTm = Tele Tm
```

The problem here is that what we want to substitute for variables in `Tele m v` is `m v` (probably `Tm v`), not `Tele m v` itself! What we need is

```
bindTele  :: Monad m => Tele m a -> (a -> m b) -> Tele m b
substTele :: (Eq v, Monad m) => v -> m v -> Tele m v -> Tele m v
instTele  :: Monad m => m v -> Tele m (Var v) -> Tele m v
```

Not what `Monad` gives us. Solving this issue in an elegant way has been a major sink of time and source of headaches for the author, who analysed some of the alternatives—most notably the work by Weirich *et al.* (2011)—but found it impossible to give up the simplicity of the model above.

That said, our term type is still reasonably brief, as shown in full in Appendix B.4. The fact that propositions cannot be factored out in another data type is an instance of the problem described above. However the real pain is during elaboration, where we are forced to treat everything as a type while we would much rather have telescopes. Future work would include writing a library that marries more flexibility with a nice interface similar to the one of `bound`.

We also make use of a 'forgetful' data type (as provided by `bound`) to store user-provided variables names along with the 'nameless' representation, so that the names will not be considered when compared terms, but will be available when distilling so that we can recover variable names that are as close as possible to what the user originally used.

7.2.2 Evaluation

Another source of contention related to term representation is dealing with evaluation. Here Bertus does not make bold moves, and simply employs substitution. When type checking we match types by reducing them to their weak head normal form, as to avoid unnecessary evaluation.

We treat data types eliminators and record projections in an uniform way, by elaborating declarations in a series of *rewriting rules*:

```
type Rewr =
    forall v.
    Tm v   ->    -- Term to which the destructor is applied
    [Tm v] ->    -- List of other arguments
    -- The result of the rewriting, if the eliminator reduces.
    Maybe [Tm v]
```

A rewriting rule is polymorphic in the variable type, guaranteeing that it just pattern matches on terms structure and rearranges them in some way, and making it possible to apply it at any level in the term. When reducing a series of applications we match the first term and check if it is a destructor, and if that's the case we apply the reduction rule and reduce further if it yields a new list of terms.

This has the advantage of simplicity, at the expense of being quite poor in terms of performance and that we need to do quotation manually. An alternative that solves both of these is the already mentioned *normalisation by evaluation*, where we would compute by turning terms into Haskell values, and then reify back to terms to compare them—a useful tutorial on this technique is given by Löh *et al.* (2010).

However, quotation has its disadvantages. The most obvious one is that it is less simple: we need to set up some infrastructure to handle the quotation and reification, while with substitution we have a uniform representation through the process of type checking. The second is that performance advantages can be rendered less effective by the continuous quoting and reifying, although this can probably be mitigated with some heuristics.

7.2.3 Parameterize everything!

Through the life of a REPL cycle we need to execute two broad 'effectful' actions:

- Retrieve, add, and modify elements to an environment. The environment will contain not only types, but also the rewriting rules presented in the previous section, and a counter to generate fresh references for the type hierarchy.
- Throw various kinds of errors when something goes wrong: parsing, type checking, input/output error when reading files, and more.

Haskell taught us the value of monads in programming languages, and in Bertus we keep this lesson in mind. All of the plumbing required to do the two actions above is provided by a very general *monad transformer* that we use through the codebase, `KMonadT`:

```
newtype KMonad f v m a = KMonad (StateT (f v) (ErrorT KError m) a)

data KError
```

```
  = OutOfBounds Id
  | DuplicateName Id
  | IOError IOError
  | ...
```

Without delving into the details of what a monad transformer is,[20] this is what `KMonadT` works with and provides:

- The `v` parameter represents the parameterized variable for the term type that we spoke about at the beginning of this section. More on this later.
- The `f` parameter indicates what kind of environment we are holding. Sometimes we want to traverse terms without carrying the entire environment, for various reasons—`KMonatT` lets us do that. Note that `f` is itself parameterized over `v`. The inner `StateT` monad transformer lets us retrieve and modify this environment at any time.
- The `m` is the 'inner' monad that we can 'plug in' to be able to perform more effectful actions in `KMonatT`. For example if we plug the `IO` monad in, we will be able to do input/output.
- The inner `ErrorT` lets us throw errors at any time. The error type is `KError`, which describes all the possible errors that a Bertus process can throw.
- Finally, the `a` parameter represents the return type of the computation we are executing.

The clever trick in `KMonadT` is to have it to be parametrised over the same type as the term type. This way, we can easily carry the environment while traversing under binders. For example, if we only needed to carry types of bound variables in the environment, we can quickly set up the following infrastructure:

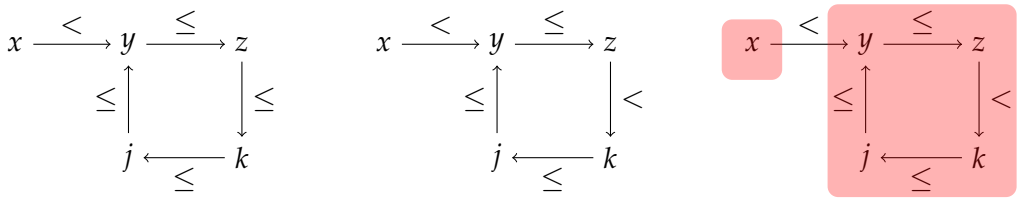
```
data Tm v = ...

-- A context is a mapping from variables to types.
newtype Ctx v = Ctx (v -> Tm v)

-- A context monad holds a context.
type CtxMonad v m = KMonadT Ctx v m

-- Enter into a scope binding a type to the variable, execute a
-- computation there, and return exit the scope returning to the 'current'
-- context.
nestM :: Monad m => Tm v -> CtxMonad (Var v) m a -> CtxMonad v m a
nestM = ...
```

Again, the types guard our back guaranteeing that we add a type when we enter a scope, and we discharge it when we get out. The author originally started with a more traditional representation and often forgot to add the right variable at the right moment. Using this practices it is very difficult to do so—we achieve correctness through types.

In the actual Bertus codebase, we have also abstracted the concept of 'context' further, so that we can easily embed contexts into other structures and write generic operations on all context-like structures.[21]

[20] See https://en.wikibooks.org/wiki/Haskell/Monad_transformers.

[21] See the `Kant.Cursor` module for details.



(a) Before $z < k$. (b) After $z < k$. (c) SCCs.

Figure 13: Strong constraints overrule weak constraints.

7.3 Turning a hierarchy into some graphs

In this section we will explain how to implement the typical ambiguity we have spoken about in 6.4 efficiently, a subject which is often dismissed in the literature. As mentioned, we have to verify a the consistency of a set of constraints each time we add a new one. The constraints range over some set of variables whose members we will denote with $x, y, z, \ldots$ and are of two kinds:

$$x \leq y \quad x < y$$

Predictably, $\leq$ expresses a reflexive order, and $<$ expresses an irreflexive order, both working with the same notion of equality, where $x < y$ implies $x \leq y$—they behave like $\leq$ and $<$ do for natural numbers (or in our case, levels in a type hierarchy). We also need an equality constraint ($x = y$), which can be reduced to two constraints $x \leq y$ and $y \leq x$.

Given this specification, we have implemented a standalone Haskell module—that we plan to release as a library—to efficiently store and check the consistency of constraints. The problem predictably reduces to a graph algorithm, and for this reason we also implement a library for labelled graphs, since the existing Haskell graph libraries fell short in different areas.[22] The interfaces for these modules are shown in Appendix B.5. The graph library is implemented as a modification of the code described by King & Launchbury (1995).

We represent the set by building a graph where vertices are variables, and edges are constraints between them, labelled with the appropriate constraint: $x < y$ gives rise to a $<$-labelled edge from x to y, and $x \leq y$ to a $\leq$-labelled edge from x to y. As we add constraints, $\leq$ constraints are replaced by $<$ constraints, so that if we started with an empty set and added

$$x < y,\ y \leq z,\ z \leq k,\ k < j,\ j \leq y$$

it would generate the graph shown in Figure 13a, but adding $z < k$ would strengthen the edge from z to k, as shown in 13b.

Definition (Strongly connected component)**.** *A* strongly connected component *in a graph with vertices V is a subset of V, say V', such that for each $(v_1, v_2) \in V' \times V'$ there is a path from v_1 to v_2.*

The SCCs in the graph for the constraints above is shown in Figure 13c. If we have a strongly connected component with a $<$ edge—say $x < y$—in it, we have an inconsistency, since there must also be a path from y to x, and by transitivity it must either be the case that $y \leq x$ or $y < x$, which are both at odds with $x < y$.

Moreover, if we have a SCC with no $<$ edges, it means that all members of said SCC are equal, since for every $x \leq y$ we have a path from y to x, which again by transitivity means that $y \leq x$. Thus, we can *condense* the SCC to a single vertex, by choosing a variable among

[22] We tried the `Data.Graph` module in `http://hackage.haskell.org/package/containers`, and the much more featureful `fgl` library `http://hackage.haskell.org/package/fgl`.

the SCC as a representative for all the others. This can be done efficiently with disjoint set data structure, and is crucial to keep the graph compact, given the very large number of constraints generated when type checking.

7.4 (Web) REPL

Finally, we take a break from the types by giving a brief account of the design of our REPL, being a good example of modular design using various constructs dear to the Haskell programmer.

Keeping in mind the `KMonadT` monad described in Section 7.2.3, the REPL is represented as a function in `KMonadT` consuming input and hopefully producing output. Then, front ends can very easily written by marshalling data in and out of the REPL:

```
data Input
    = ITyCheck String           -- Type check a term
    | IEval String              -- Evaluate a term
    | IDecl String              -- Declare something
    | ...

data Output
    = OTyCheck TmRefId [HoleCtx] -- Type checked term, with holes
    | OPretty TmRefId            -- Term to pretty print, after evaluation
      -- Just holes, classically after loading a file
    | OHoles [HoleCtx]
    | ...

-- KMonadT is parametrised over the type of the variables, which depends
-- on how deep in the term structure we are.  For the REPL, we only deal
-- with top-level terms, and thus only 'Id' variables---top level names.
type REPL m = KMonadT Id m

repl :: ReadFile m => Input -> REPL m Output
repl = ...
```

The `ReadFile` monad embodies the only 'extra' action that we need to have access too when running the REPL: reading files. We could simply use the `IO` monad, but this will not serve us well when implementing front end facing untrusted parties accessing the application running on our servers. In our case we expose the REPL as a web application, and we want the user to be able to load only from a pre-defined directory, not from the entire file system.

For this reason we specify `ReadFile` to have just one function:

```
class Monad m => ReadFile m where
    readFile' :: FilePath -> m (Either IOError String)
```

While in the command-line application we will use the `IO` monad and have `readFile'` to work in the 'obvious' way—by reading the file corresponding to the given file path—in the web prompt we will have it to accept only a file name, not a path, and read it from a pre-defined directory:

```
-- The monad that will run the web REPL.  The 'ReaderT' holds the
-- filepath to the directory where the files loadable by the user live.
-- The underlying 'IO' monad will be used to actually read the files.
newtype DirRead a = DirRead (ReaderT FilePath IO a)
```

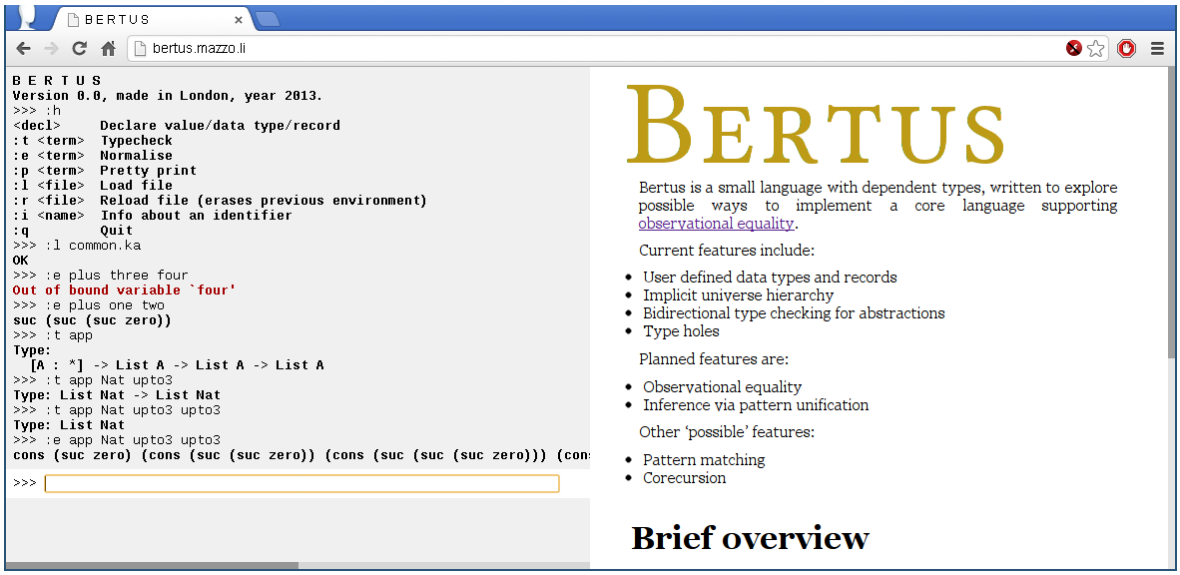


Figure 14: A sample run of the web prompt.

```
instance ReadFile DirRead where
    readFile' fp =
        do -- We get the base directory in the 'ReaderT' with 'ask'
           dir <- DirRead ask
           -- Is the filepath provided an unqualified file name?
           if snd (splitFileName fp) == fp
              -- If yes, go ahead and read the file, by lifting
              -- 'readFile'' into the IO monad
              then DirRead (lift (readFile' (dir </> fp)))
              -- If not, return an error
              else return (Left (strMsg ("Invalid file name '" ++ fp ++ "'")))
```

Once this light-weight infrastructure is in place, adding a web interface was an easy exercise. We use Jasper Van der Jeugt's `websockets` library[23] to create a proxy that receives `JSON`[24] messages with the user input, turns them into `Input` messages for the REPL, and then sends back a `JSON` message with the response. Moreover, each client is handled in a separate threads, so crashes of the REPL for a certain client will not bring the whole application down.

On the front end side, we had to write some JavaScript to accept input from a form, and to make the responses appear on the screen. The web prompt is publicly available at `http://bertus.mazzo.li`, a sample session is shown Figure 14.

[23] Available at `http://hackage.haskell.org/package/websockets`.

[24] `JSON` is a popular data interchange format, see `http://json.org` for more info.

8 | Evaluation

Going back to our goals in Section 1.2, we feel that this thesis fills a gap in the description of observational type theory. In the design of Bertus we willingly patterned the core features against the ones present in Agda, with the hope that future implementors will be able to refer to this document without embarking on the same adventure themselves. We gave an original account of heterogeneous equality by showing that in a cumulative hierarchy we can keep equalities as small as we would be able too with a separate notion of type equality. As a side effect of developing Bertus, we also gave an original account of bidirectional type checking for user defined types, which get rid of many types while keeping the language very simple.

Through the design of the theory of Bertus we have followed an approach where study and implementation were continuously interleaved, as a 'reality check' for the ideas that we wished to implement. Given the great effort necessary to build a theorem prover capable of 'real-world' proofs we have not attempted to compare Bertus's capabilities to those of Agda and Coq, the theorem provers that the author is most familiar with and in general two of the main players in the field. However we have ported a lot of simpler examples to check that the key features are working, some of which have been used in the previous sections and are reproduced in the appendices[25]. A full example of interaction with Bertus is given in Section 8.1.

The main culprits for the delays in the implementation are two issues that revealed themselves to be far less obvious than what the author predicted. The first, as we have already remarked in Section 7.2, is to have an adequate term representation that lets us express the right constructs in a safe way. There is still no widely accepted solution to this problem, which is approached in many different ways both in the literature and in the programming practice. The second aspect is the treatment of user defined data types. Again, the best techniques to implement them in a dependently typed setting still have not crystallised and implementors reinvent many wheels each time a new system is built. The author is still conflicted on whether having user defined types at all it is the right decision: while they are essential, the recent discovery of a paper by Dagand & McBride (2012) describing a way to efficiently encode user-defined data types to a set of core primitives—an option that seems very attractive.

In general, implementing dependently typed languages is still a poorly understood practice, and almost every stage requires experimentation on behalf of the author. Another example is the treatment of the implicit hierarchy, where no resources are present describing the problem from an implementation perspective (we described our approach in Section 7.3). Hopefully this state of things will change in the near future, and recent publications are promising in this direction, for example an unpublished paper by Brady (2013) describing his implementation of the Idris programming language. Our ultimate goal is to be a part of this collective effort.

8.1 A type holes tutorial

As a taster and showcase for the capabilities of Bertus, we present an interactive session with the Bertus REPL. While doing so, we present a feature that we still have not covered: type holes.

Type holes are, in the author's opinion, one of the 'killer' features of interactive theorem provers, and one that is begging to be exported to mainstream programming—although it is much more effective in a well-typed, functional setting. The idea is that when we are

[25]The full list is available in the repository: https://github.com/bitonic/kant/tree/master/data/samples/good.

developing a proof or a program we can insert a hole to have the software tell us the type expected at that point. Furthermore, we can ask for the type of variables in context, to better understand our surroundings.

In Bertus we use type holes by putting them where a term should go. We need to specify a name for the hole and then we can put as many terms as we like in it. Bertus will tell us which type it is expecting for the term where the hole is, and the type for each term that we have included. For example if we had:

```
plus [m n : Nat] : Nat ⇒ (
    {| h1 m n |}
)
```

And we loaded the file in Bertus, we would get:

```
>>> :l plus.ka
Holes:
  h1 : Nat
    m : Nat
    n : Nat
```

Suppose we wanted to define the 'less or equal' ordering on natural numbers as described in Section 6.3.3. We will incrementally build our functions in a file called `le.ka`. First we define the necessary types, all of which we know well by now:

```
data Nat : ⋆ ⇒ { zero : Nat | suc : Nat → Nat }

data Empty : ⋆ ⇒ { }
absurd [A : ⋆] [p : Empty] : A ⇒ (
    Empty-Elim p (λ _ ⇒ A)
)

record Unit : ⋆ ⇒ tt { }
```

Then fire up Bertus, and load the file:

```
% ./bertus
B E R T U S
Version 0.0, made in London, year 2013.
>>> :l le.ka
OK
```

So far so good. Our definition will be defined by recursion on a natural number `n`, which will return a function that given another number `m` will return the trivial type `Unit` if $\texttt{n} \leq \texttt{m}$, and the `Empty` type otherwise. However we are still not sure on how to define it, so we invoke `Nat-Elim`, the eliminator for natural numbers, and place holes instead of arguments. In the file we will write:

```
le [n : Nat] : Nat → ⋆ ⇒ (
  Nat-Elim n (λ _ ⇒ Nat → ⋆)
    {|h1|}
    {|h2|}
)
```

And then we reload in Bertus:

```
>>> :r le.ka
Holes:
  h1 : Nat → ⋆
  h2 : Nat → (Nat → ⋆) → Nat → ⋆
```

Which tells us what types we need to satisfy in each hole. However, it is not that clear what does what in each hole, and thus it is useful to have a definition vacuous in its arguments just to clear things up. We will use `Le` aid in reading the goal, with `Le m n` as a reminder that we to return the type corresponding to $\texttt{m} \leq \texttt{n}$:

```
Le [m n : Nat] : ⋆ ⇒ ⋆

le [n : Nat] : [m : Nat] → Le n m ⇒ (
  Nat-Elim n (λ n ⇒ [m : Nat] → Le n m)
    {|h1|}
    {|h2|}
)
```

```
>>> :r le.ka
Holes:
  h1 : [m : Nat] → Le zero m
  h2 : [x : Nat] → ([m : Nat] → Le x m) → [m : Nat] → Le (suc x) m
```

This is much better! Bertus, when printing terms, does not substitute top-level names for their bodies, since usually the resulting term is much clearer. As a nice side-effect, we can use tricks like this to find guidance.

In this case in the first case we need to return, given any number $\texttt{m}$, $0 \leq \texttt{m}$. The trivial type will do, since every number is less or equal than zero:

```
le [n : Nat] : [m : Nat] → Le n m ⇒ (
  Nat-Elim n (λ n ⇒ [m : Nat] → Le n m)
    (λ _ ⇒ Unit)
    {|h2|}
)
```

```
>>> :r le.ka
Holes:
  h2 : [x : Nat] → ([m : Nat] → Le x m) → [m : Nat] → Le (suc x) m
```

Now for the important case. We are given our comparison function for a number, and we need to produce the function for the successor. Thus, we need to re-apply the induction principle on the other number, `m`:

```
le [n : Nat] : [m : Nat] → Le n m ⇒ (
  Nat-Elim n (λ n ⇒ [m : Nat] → Le n m)
    (λ _ ⇒ Unit)
    (λ n' f m ⇒ Nat-Elim m (λ m' ⇒ Le (suc n') m') {|h2|} {|h3|})
)
```

```
>>> :r le.ka
Holes:
  h2 : ⋆
  h3 : [x : Nat] → Le (suc n') x → Le (suc n') (suc x)
```

In the first hole we know that the second number is zero, and thus we can return empty. In the second case, we can use the recursive argument `f` on the two numbers:

```
le [n : Nat] : [m : Nat] → Le n m ⇒ (
  Nat-Elim n (λ n ⇒ [m : Nat] → Le n m)
    (λ _ ⇒ Unit)
    (λ n' f m ⇒
       Nat-Elim m (λ m' ⇒ Le (suc n') m') Empty (λ f _ ⇒ f m'))
)
```

We can verify that our function works as expected:

```
>>> :e le zero zero
Unit
>>> :e le zero (suc zero)
Unit
>>> :e le (suc (suc zero)) (suc zero)
Empty
```

The other functionality of type holes is examining types of things in context. Going back to the examples in Section 3.3.1, we can implement the safe `head` function with our newly defined `le`:

```
data List : [A : ⋆] → ⋆ ⇒
  { nil : List A | cons : A → List A → List A }

length [A : ⋆] [l : List A] : Nat ⇒ (
  List-Elim l (λ _ ⇒ Nat) zero (λ _ _ n ⇒ suc n)
)

gt [n m : Nat] : ⋆ ⇒ (le (suc m) n)

head [A : ⋆] [l : List A] : gt (length A l) zero → A ⇒ (
  List-Elim l (λ l ⇒ gt (length A l) zero → A)
    (λ p ⇒ {|h1 p|})
    {|h2|}
)
```

We define `Lists`, a polymorphic `length` function, and express $<$ (`gt`) in terms of $\leq$. Then, we set up the type for our `head` function. Given a list and a proof that its length is greater than zero, we return the first element. If we load this in BERTUS, we get:

```
>>> :r le.ka
Holes:
  h1 : A
    p : Empty
  h2 : [x : A] [x1 : List A] →
       (gt (length A x1) zero → A) →
       gt (length A (cons x x1)) zero → A
```

In the first case (the one for `nil`), we have a proof of `Empty`—surely we can use `absurd` to get rid of that case. In the second case we simply return the element in the `cons`:

```
head [A : ⋆] [l : List A] : gt (length A l) zero → A ⇒ (
  List-Elim l (λ l ⇒ gt (length A l) zero → A)
```

```
        (λ p ⇒ absurd A p)
        (λ x _ _ _ ⇒ x)
    )
```

Now, if we tried to get the head of an empty list, we face a problem:

```
>>> :t head Nat nil
Type: Empty → Nat
```

We would have to provide something of type `Empty`, which hopefully should be impossible. For non-empty lists, on the other hand, things run smoothly:

```
>>> :t head Nat (cons zero nil)
Type: Unit → Nat
>>> :e head Nat (cons zero nil) tt
zero
```

This should give a vague idea of why type holes are so useful and in more in general about the development process in Bertus. Most interactive theorem provers offer some kind of facility to... interactively develop proofs, usually much more powerful than the fairly bare tools present in Bertus. Agda in particular offers a celebrated interactive mode for the `Emacs` text editor.

9 | Future work

The first move that the author plans to make is to work towards a simple but powerful term representation. A good plan seems to be to associate each type (terms, telescopes, etc.) with what we can substitute variables with, so that the term type will be associated with itself, while telescopes and propositions will be associated to terms. This can probably be accomplished elegantly with Haskell's *type families* (Chakravarty *et al.* , 2005). After achieving a more solid machinery for terms, implementing observational equality fully should prove relatively easy.

Beyond this steps, we can go in many directions to improve the system that we described—here we review the main ones.

Pattern matching and recursion Eliminators are very clumsy, and using them can be especially frustrating if we are used to writing functions via explicit recursion. Giménez (1995) showed how to reduce well-founded recursive definitions to primitive recursors. Intuitively, defining a function through an eliminators corresponds to pattern matching and recursively calling the function on the recursive occurrences of the type we matched against.

Nested pattern matching can be justified by identifying a notion of 'structurally smaller', and allowing recursive calls on all smaller arguments. Epigram goes all the way and actually implements recursion exclusively by providing a convenient interface to the two constructs above (McBride, 2004; McBride & McKinna, 2004a).

However as we extend the flexibility in our recursion elaborating definitions to eliminators becomes more and more laborious. For example we might want mutually recursive definitions and definitions that terminate relying on the structure of two arguments instead of just one. For this reason both Agda and Coq (Agda putting more effort) let the user write recursive definitions freely, and then employ an external syntactic one the recursive calls to ensure that the definitions are terminating.

Moreover, if we want to use dependently typed languages for programming purposes, we will probably want to sidestep the termination checker and write a possibly non-terminating function; maybe because proving termination is particularly difficult. With explicit recursion this amounts to turning off a check, if we have only eliminators it is impossible.

More powerful data types A popular improvement on basic data types are inductive families (Dybjer, 1991), where the parameters for the type constructors can change based on the data constructors, which lets us express naturally types such as $\mathsf{Vec} : \mathbb{N} \to \mathsf{Type}$, which given a number returns the type of lists of that length, or $\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$, which given a number n gives the type of numbers less than n. This apparent omission was motivated by the fact that inductive families can be represented by adding equalities concerning the parameters of the type constructors as arguments to the data constructor, in much the same way that Generalised Abstract Data Types (The GHC Team, 2012) are handled in Haskell. Interestingly the modified version of System F that lies at the core of recent versions of GHC features coercions reminiscent of those found in OTT, motivated precisely by the need to implement GADTs in an elegant way (Sulzmann *et al.* , 2007).

Another concept introduced by Dybjer (2000) is induction-recursion, where we define a data type in tandem with a function on that type. This technique has proven extremely useful to define embeddings of other calculi in an host language, by defining the representation of the embedded language as a data type and at the same time a

function decoding from the representation to a type in the host language. The decoding function is then used to define the data type for the embedding itself, for example by reusing the host's language functions to describe functions in the embedded language, with decoded types as arguments.

It is also worth mentioning that in recent times there has been work (Dagand & McBride, 2012; Chapman *et al.* , 2010) to show how to define a set of primitives that data types can be elaborated into. The big advantage of the approach proposed is enabling a very powerful notion of generic programming, by writing functions working on the 'primitive' types as to be workable by all the other 'compatible' elaborated user defined types. This has been a considerable problem in the dependently type world, where we often define types which are more 'strongly typed' version of similar structures,[26] and then find ourselves forced to redefine identical operations on both types.

Pattern matching and inductive families The notion of inductive family also yields a more interesting notion of pattern matching, since matching on an argument influences the value of the parameters of the type of said argument. This means that pattern matching influences the context, which can be exploited to constraint the possible data constructors for *other* arguments (McBride & McKinna, 2004a).

Type inference While bidirectional type checking helps at a very low cost of implementation and complexity, a much more powerful weapon is found in *pattern unification*, which allows Hindley-Milner style inference for dependently typed languages. Unification for higher order terms is undecidable and unification problems do not always have a most general unifier (Huet, 1973). However Miller (1992) identified a decidable fragment of higher order unification commonly known as pattern unification, which is employed in most theorem provers to drastically reduce the number of type annotations. Gundry & McBride (2013) provide a tutorial on this practice.

Coinductive data types When we specify inductive data types, we do it by specifying its *constructors*—functions with the type we are defining as codomain. Then, we are offered way of compute by recursively *destructing* or *eliminating* a member of the defined data type.

Coinductive data types are the dual of this approach. We specify ways to destruct data, and we are given a way to generate the defined type by repeatedly 'unfolding' starting from some seed. For example, we could defined infinite streams by specifying a **head** and **tail** destructors—here using a syntax reminiscent of Bertus records:

$$\begin{array}{l}\underline{\text{codata}}\ \mathsf{Stream}\ (A{:}\mathsf{Type})\ \underline{\text{where}}\\ \quad \{\mathbf{head} : A, \mathbf{tail} : \mathsf{Stream}\ A\}\end{array}$$

which will hopefully give us something like

$$\begin{array}{l}\mathbf{head} : (A{:}\mathsf{Type}) \to \mathsf{Stream}\ A \to A\\ \mathbf{tail} : (A{:}\mathsf{Type}) \to \mathsf{Stream}\ A \to \mathsf{Stream}\ A\\ \mathsf{Stream.unfold} : (A\ B{:}\mathsf{Type}) \to (A \to B \times A) \to A \to \mathsf{Stream}\ B\end{array}$$

Where, in unfold, $B \times A$ represents the fields of Stream but with the recursive occurrence replaced by the 'seed' type A.

Beyond simple infinite types like Stream, coinduction is particularly useful to write non-terminating programs like servers or software interacting with a user, while

[26] For example the OList presented in Section 6.3.3 being a 'more typed' version of an ordinary list.

guaranteeing their liveliness. Moreover it lets us model possibly non-terminating computations in an elegant way (Capretta, 2005), enabling for example the study of operational semantics for non-terminating languages (Danielsson, 2012).

Cockett & Fukushima (1992) pioneered this approach in their programming language Charity, and coinduction has since been adopted in systems such as Coq (Giménez, 1996) and Agda. However these implementations are unsatisfactory, since Coq's break subject reduction; and Agda, to avoid this problem, does not allow types to depend on the unfolding of codata. McBride (2009) has shown how observational equality can help to resolve these issues, since we can reason about the unfoldings in a better way, like we reason about functions' extensional behaviour.

The author looks forward to the study and possibly the implementation of these ideas in the years to come.

A | Notation and syntax

Syntax, derivation rules, and reduction rules, are enclosed in frames describing the type of relation being established and the syntactic elements appearing, for example

typing: $\Gamma \vdash \mathit{term} : \mathit{type}$

Typing derivations here.

In the languages presented and Agda code samples we also highlight the syntax, following a uniform colour, capitalisation, and font style convention:

Sans	Type constructors.
sans	Data constructors.
roman	Keywords of the language.
roman	Defined values and destructors.
math	Bound variables.

When presenting grammars, we use a word in *math* font (e.g. *term* or *type*) to indicate indicate nonterminals. Additionally, we use quite flexibly a *math* font to indicate a syntactic element in derivations or meta-operations. More specifically, terms are usually indicated by lowercase letters (often t, m, or n); and types by an uppercase letter (often A, B, or C).

When presenting type derivations, we often abbreviate and present multiple conclusions, each on a separate line:

$$\frac{\Gamma \vdash t : A \times B}{\begin{array}{l}\Gamma \vdash \mathbf{fst}\, t : A \\ \Gamma \vdash \mathbf{snd}\, t : B\end{array}}$$

We often present 'definitions' in the described calculi and in Bertus itself, like so:

name : *type*
name $\mathit{arg}_1\ \mathit{arg}_2 \cdots \mapsto \mathit{term}$

To define operators, we use a mixfix notation similar to Agda, where _s denote arguments:

$\_ \wedge \_ : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$
$b_1 \wedge b_2 \mapsto \cdots$

In explicitly typed systems, we omit type annotations when they are obvious, e.g. by not annotating the type of parameters of abstractions or of dependent pairs.
We introduce multiple arguments in one go in arrow types:

$$(x\, y{:}A) \to \cdots = (x{:}A) \to (y{:}A) \to \cdots$$

and in abstractions:

$$\lambda x\, y \mapsto \cdots = \lambda x \mapsto \lambda y \mapsto \cdots$$

We also omit arrows to abbreviate types:

$$(x{:}A)(y{:}B) \to \cdots = (x{:}A) \to (y{:}B) \to \cdots$$

Meta operations names are displayed in smallcaps and written in a pattern matching style, also making use of boolean guards. For example, a meta operation operating on a context and terms might look like this:

$\textsc{quux}(\Gamma, x) \mid x \in \Gamma \Longrightarrow \Gamma(x)$
$\textsc{quux}(\Gamma, x) \Longrightarrow \textsc{outofbounds}$
$\vdots$

From time to time we give examples in the Haskell programming language as defined by Marlow (2010), which we typeset in `teletype` font. I assume that the reader is already familiar with Haskell, plenty of good introductions are available (Lipovača, 2009; Hutton, 2007).

Examples of Bertus code will be typeset nicely with LaTeXin Section 6, to adjust with the rest of the presentation; and in `teletype` font in the rest of the document, including Section 7 and in the appendices. All the Bertus code shown is meant to be working and ready to be inputted in a Bertus prompt or loaded from a file. Snippets of sessions in the Bertus prompt will be displayed with a left border, to distinguish them from snippets of code:

```
>>> :t ⋆
Type: ⋆
```

B | Code

B.1 ITT renditions

B.1.1 Agda

Note that in what follows rules for 'base' types are universe-polymorphic, to reflect the exposition. Derived definitions, on the other hand, mostly work with Set, reflecting the fact that in the theory presented we don't have universe polymorphism.

```
module ITT where
  open import Level

  data Empty : Set where

  absurd : ∀ {a} {A : Set a} → Empty → A
  absurd ()

  ¬_ : ∀ {a} → (A : Set a) → Set a
  ¬ A = A → Empty

  record Unit : Set where
    constructor tt

  record _×_ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
    constructor _,_
    field
      fst : A
      snd : B fst
  open _×_ public

  data Bool : Set where
    true false : Bool

  if_/_then_else_ : ∀ {a} (x : Bool) (P : Bool → Set a) → P true → P false → P x
  if true / _ then x else _ = x
  if false / _ then _ else x = x

  if_then_else_ : ∀ {a} (x : Bool) {P : Bool → Set a} → P true → P false → P x
  if_then_else_ x {P} = if_/_then_else_ x P

  data W {s p} (S : Set s) (P : S → Set p) : Set (s ⊔ p) where
    _◁_ : (s : S) → (P s → W S P) → W S P

  rec : ∀ {a b} {S : Set a} {P : S → Set b}
    (C : W S P → Set) →          - some conclusion we hope holds
    ((s : S) →                   - given a shape...
      (f : P s → W S P) →        - ...and a bunch of kids...
      ((p : P s) → C (f p)) →    - ...and C for each kid in the bunch...
      C (s ◁ f)) →               - ...does C hold for the node?
    (x : W S P) →                - If so, ...
    C x                          - ...C always holds.
```

```
  rec C c (s ◁ f) = c s f (λ p → rec C c (f p))

module Examples-→ where
  open ITT

  data ℕ : Set where
    zero : ℕ
    suc : ℕ → ℕ

  - These pragmas are needed so we can use number literals.
  {-# BUILTIN NATURAL ℕ #-}
  {-# BUILTIN ZERO zero #-}
  {-# BUILTIN SUC suc #-}

  data List (A : Set) : Set where
    [] : List A
    _::_ : A → List A → List A

  length : ∀ {A} → List A → ℕ
  length [] = zero
  length (_ :: l) = suc (length l)

  _>_ : ℕ → ℕ → Set
  zero > _ = Empty
  suc _ > zero = Unit
  suc x > suc y = x > y

  head : ∀ {A} → (l : List A) → length l > 0 → A
  head [] p = absurd p
  head (x :: _) _ = x

module Examples-× where
  open ITT
  open Examples-→

  even : ℕ → Set
  even zero = Unit
  even (suc zero) = Empty
  even (suc (suc n)) = even n

  6-even : even 6
  6-even = tt

  5-not-even : ¬ (even 5)
  5-not-even = absurd

  there-is-an-even-number : ℕ × even
  there-is-an-even-number = 6 , 6-even

  _∨_ : (A B : Set) → Set
  A ∨ B = Bool × (λ b → if b then A else B)
```

```
  left : ∀ {A B} → A → A ∨ B
  left x = true , x

  right : ∀ {A B} → B → A ∨ B
  right x = false , x

  [_,_] : {A B C : Set} → (A → C) → (B → C) → A ∨ B → C
  [ f , g ] x =
    (if (fst x) / (λ b → if b then _ else _ → _) then f else g) (snd x)

module Examples-W where
  open ITT
  open Examples-×

  Tr : Bool → Set
  Tr b = if b then Unit else Empty

  ℕ : Set
  ℕ = W Bool Tr

  zero : ℕ
  zero = false ◁ absurd

  suc : ℕ → ℕ
  suc n = true ◁ (λ _ → n)

  plus : ℕ → ℕ → ℕ
  plus x y = rec
    (λ _ → ℕ)
    (λ b →
      if b / (λ b → (Tr b → ℕ) → (Tr b → ℕ) → ℕ)
      then (λ _ f → (suc (f tt))) else (λ _ _ → y))
    x

module Equality where
  open ITT

  data _≡_ {a} {A : Set a} : A → A → Set a where
    refl : ∀ x → x ≡ x

  ≡-elim : ∀ {a b} {A : Set a}
    (P : (x y : A) → x ≡ y → Set b) →
    ∀ {x y} → P x x (refl x) → (x≡y : x ≡ y) → P x y x≡y
  ≡-elim P p (refl x) = p

  subst : ∀ {A : Set} (P : A → Set) → ∀ {x y} → (x≡y : x ≡ y) → P x → P y
  subst P x≡y p = ≡-elim (λ _ y _ → P y) p x≡y

  sym : ∀ {A : Set} (x y : A) → x ≡ y → y ≡ x
  sym x y p = subst (λ y′ → y′ ≡ x) p (refl x)
```

```
trans : ∀ {A : Set} (x y z : A) → x ≡ y → y ≡ z → x ≡ z
trans x y z p q = subst (λ z' → x ≡ z') q p

cong : ∀ {A B : Set} (x y : A) → x ≡ y → (f : A → B) → f x ≡ f y
cong x y p f = subst (λ z → f x ≡ f z) p (refl (f x))
```

B.1.2 Bertus

The following things are missing: W-types, since our positivity check is overly strict, and equality, since we haven't implemented that yet.

```
------------------------------------------------------------
-- Core ITT (minus W)

data Empty : ⋆ ⇒ { }

absurd [A : ⋆] [x : Empty] : A ⇒ (
  Empty-Elim x (λ _ ⇒ A)
)

neg [A : ⋆] : ⋆ ⇒ (A → Empty)

record Unit : ⋆ ⇒ tt { }

record Prod : [A : ⋆] [B : A → ⋆] → ⋆ ⇒
  prod {fst : A, snd : B fst}

data Bool : ⋆ ⇒ { true : Bool | false : Bool }

-- The if_then_else_ is provided by Bool-Elim

-- A large eliminator, for convenience
ITE [b : Bool] [A B : ⋆] : ⋆ ⇒ (
  Bool-Elim b (λ _ ⇒ ⋆) A B
)

------------------------------------------------------------
-- Examples →

data Nat : ⋆ ⇒ { zero : Nat | suc : Nat → Nat }

gt [n : Nat] : Nat → ⋆ ⇒ (
  Nat-Elim
    n
    (λ _ ⇒ Nat → ⋆)
    (λ _ ⇒ Empty)
    (λ n f m ⇒ Nat-Elim m (λ _ ⇒ ⋆) Unit (λ m' _ ⇒ f m'))
)

data List : [A : ⋆] → ⋆ ⇒
  { nil : List A | cons : A → List A → List A }

length [A : ⋆] [xs : List A] : Nat ⇒ (
    List-Elim xs (λ _ ⇒ Nat) zero (λ _ _ n ⇒ suc n)
)
```

```
head [A : ⋆] [xs : List A] : gt (length A xs) zero → A ⇒ (
    List-Elim
      xs
      (λ xs ⇒ gt (length A xs) zero → A)
      (λ p ⇒ absurd A p)
      (λ x _ _ _ ⇒ x)
)

------------------------------------------------------------
-- Examples ×

data Parity : ⋆ ⇒ { even : Parity | odd : Parity }

flip [p : Parity] : Parity ⇒ (
  Parity-Elim p (λ _ ⇒ Parity) odd even
)

parity [n : Nat] : Parity ⇒ (
  Nat-Elim n (λ _ ⇒ Parity) even (λ _ ⇒ flip)
)

even [n : Nat] : ⋆ ⇒ (Parity-Elim (parity n) (λ _ ⇒ ⋆) Unit Empty)

one   : Nat ⇒ (suc zero)
two   : Nat ⇒ (suc one)
three : Nat ⇒ (suc two)
four  : Nat ⇒ (suc three)
five  : Nat ⇒ (suc four)
six   : Nat ⇒ (suc five)

even-6 : even six ⇒ tt

even-5-neg : neg (even five) ⇒ (λ z ⇒ z)

there-is-an-even-number : Prod Nat even ⇒ (prod six even-6)

Or [A B : ⋆] : ⋆ ⇒ (Prod Bool (λ b ⇒ ITE b A B))

left  [A B : ⋆] [x : A] : Or A B ⇒ (prod true  x)
right [A B : ⋆] [x : B] : Or A B ⇒ (prod false x)

case [A B C : ⋆] [f : A → C] [g : B → C] [x : Or A B] : C ⇒ (
  (Bool-Elim (fst x) (λ b ⇒ ITE b A B → C) f g) (snd x)
)
```

B.2 BERTUS examples

```
------------------------------------------------------------
-- Naturals

data Nat : ⋆ ⇒ { zero : Nat | suc : Nat → Nat }

one   : Nat ⇒ (suc zero)
two   : Nat ⇒ (suc one)
three : Nat ⇒ (suc two)

------------------------------------------------------------
```

```
-- Binary trees

data Tree : [A : ⋆] → ⋆ ⇒
  { leaf : Tree A | node : Tree A → A → Tree A → Tree A }

------------------------------------------------------------
-- Empty types

data Empty : ⋆ ⇒ { }

------------------------------------------------------------
-- Ordered lists

record Unit : ⋆ ⇒ tt { }

le [n : Nat] : Nat → ⋆ ⇒ (
  Nat-Elim
    n
    (λ _ ⇒ Nat → ⋆)
    (λ _ ⇒ Unit)
    (λ n f m ⇒ Nat-Elim m (λ _ ⇒ ⋆) Empty (λ m' _ ⇒ f m'))
)

data Lift : ⋆ ⇒
  { bot : Lift | lift : Nat → Lift | top : Lift }

le' [l1 : Lift] : Lift → ⋆ ⇒ (
  Lift-Elim
    l1
    (λ _ ⇒ Lift → ⋆)
    (λ _     ⇒ Unit)
    (λ n1 l2 ⇒ Lift-Elim l2 (λ _ ⇒ ⋆) Empty (λ n2 ⇒ le n1 n2) Unit)
    (λ l2    ⇒ Lift-Elim l2 (λ _ ⇒ ⋆) Empty (λ _  ⇒ Empty)    Unit)
)

data OList : [low upp : Lift] → ⋆ ⇒
  { onil  : le' low upp → OList low upp
  | ocons : [n : Nat] → OList (lift n) upp → le' low (lift n) → OList low upp
  }

data List : [A : ⋆] → ⋆ ⇒
  { nil : List A | cons : A → List A → List A }

------------------------------------------------------------
-- Dependent products

record Prod : [A : ⋆] [B : A → ⋆] → ⋆ ⇒
  prod {fst : A, snd : B fst}
```

B.3 Bertus' hierachy

This rendition of the Hurken's paradox does not type check with the hierachy enabled, type checks and loops without it. Adapted from an Agda version, available at http://code.haskell.org/Agda/test/succeed/Hurkens.agda.

```
bot : ⋆ ⇒ ([A : ⋆] → A)

not [A : ⋆] : ⋆ ⇒ (A → bot)
```

```
P [A : ⋆] : ⋆ ⇒ (A → ⋆)

U : ⋆ ⇒ ([X : ⋆] → (P (P X) → X) → P (P X))

tau [t : P (P U)] : U ⇒ (
  λ X f p ⇒ t (λ x ⇒ p (f (x X f)))
)

sigma [s : U] : P (P U) ⇒ (s U tau)

Delta : P U ⇒ (
  λ y ⇒ not ([p : P U] → sigma y p → p (tau (sigma y)))
)

Omega : U ⇒ (
  tau (λ p ⇒ [x : U] → sigma x p → p x)
)

D : ⋆ ⇒ (
  [p : P U] → sigma Omega p → p (tau (sigma Omega))
)

lem1 [p : P U] [H1 : [x : U] → sigma x p → p x] : p Omega ⇒ (
  H1 Omega (λ x ⇒ H1 (tau (sigma x)))
)

lem2 : not D ⇒ (
  lem1 Delta (λ x H2 H3 ⇒ H3 Delta H2 (λ p ⇒ H3 (λ y ⇒ p (tau (sigma y)))))
)

lem3 : D ⇒ (
  λ p ⇒ lem1 (λ y ⇒ p (tau (sigma y)))
)

loop  : bot ⇒ (lem2 lem3)
```

B.4 Term representation

Data type for terms in Bertus.

```
-- A top-level name.
type Id    = String
-- A data/type constructor name.
type ConId = String

-- A term, parametrised over the variable (‘v’) and over the reference
-- type used in the type hierarchy (‘r’).
data Tm r v
    = V v                       -- Variable.
    | Ty r                      -- Type, with a hierarchy reference.
    | Lam (TmScope r v)         -- Abstraction.
    | Arr (Tm r v) (TmScope r v) -- Dependent function.
    | App (Tm r v) (Tm r v)     -- Application.
    | Ann (Tm r v) (Tm r v)     -- Annotated term.
      -- Data constructor, the first ConId is the type constructor and
      -- the second is the data constructor.
    | Con ADTRec ConId ConId [Tm r v]
```

```
      -- Data destrutor, again first ConId being the type constructor
      -- and the second the name of the eliminator.
    | Destr ADTRec ConId Id (Tm r v)
      -- A type hole.
    | Hole HoleId [Tm r v]
      -- Decoding of propositions.
    | Dec (Tm r v)

      -- Propositions.
    | Prop r -- The type of proofs, with hierarchy reference.
    | Top
    | Bot
    | And (Tm r v) (Tm r v)
    | Forall (Tm r v) (TmScope r v)
      -- Heterogeneous equality.
    | Eq (Tm r v) (Tm r v) (Tm r v) (Tm r v)

-- Either a data type, or a record.
data ADTRec = ADT | Rc

-- Either a coercion, or coherence.
data Coeh = Coe | Coh
```

B.5 Graph and constraints modules

The modules are respectively named `Data.LGraph` (short for 'labelled graph'), and `Data.Constraint`. The type class constraints on the type parameters are not shown for clarity, unless they are meaningful to the function. In practice we use the `Hashable` type class on the vertex to implement the graph efficiently with hash maps.

B.5.1 Data.LGraph

```
module Data.LGraph where

-- | A ‘representative’ for a vertex.  Each vertex in the graph is
--   initially its own representative, and when condensing cycles a new
--   representative for all the vertices is choosen.
data Rep v

-- | A graph with vertices of type ‘v’ and labels of type ‘l’.
data Graph v l

-- | An ‘Edge’ is two vertices and a label.  Used by the user.
type Edge v l = (v, l, v)
-- | An edge between representatives.  Internally, this is what we will
--   have.
type RepEdge v l = (Rep v, l, Rep v)

-- | Empty graph.
empty :: Graph v l

-- | Adds an ‘Edge’ and returns the new ‘Graph’.  Inserts the vertices
--   if missing.  If the edge exists already but the label is ‘greater’
--   than the existing one, the label is replaced by the new one.  Thus
--   the ‘Ord’ constraint on ‘l’.
addEdge :: Ord l => Edge v l -> Graph v l -> Graph v l

-- | All the vertices in the graph.
```

```
vertices :: Graph v l -> [Rep v]

-- | All the edges in the graph.
edges :: Graph v l -> [RepEdge v l]

-- | Gets all the edges between the provided vertices.
inEdges :: [Rep v] -> Graph v l -> [RepEdge v l]

-- | Gets the transpose of the graph.
transpose :: Graph v l -> Graph v l

-- | A 'Tree' has a root node and a 'Forest' of successors.
data Tree v l   = Node (Rep v) (Forest v l)
type Forest v l = [Tree v l]

-- | A depth first search on the graph, starting from the given
--   vertices.  If there are cycles, an infinite 'Forest' will be
--   generated---which is fine, since the result is produced lazily.
dfs :: Graph v l -> [Rep v] -> Forest v l

-- | 'dff gr = dfs gr (vertices gr)'
dff :: Graph v l -> Forest v l

-- | The vertices of the graph in post order.
postOrd :: Graph v l -> [Rep v]

-- | A strongly connected component (SCC) is either a single acyclic
--   vertex, or a list of vertices V where for each v1, v2 in V there is a
--   path from v1 to v2.
data SCC v l = Acyclic (Rep v) | Cyclic [Rep v]

-- | All the SCCs of the graph.
scc :: Graph v l -> [SCC v l]

-- | Condense a given SCC, choosing one representative among the nodes
--   and updating the edges accordingly.
condense :: SCC v l -> Graph v l -> Graph v l
```

B.5.2 Data.Constraint

```
module Data.Constraint where

-- | Data type holding the set of constraints, parametrised over the
--   type of the variables.
data Constrs a

-- | A representation of the constraints that we can add.
data Constr a = a :<=: a | a :<: a | a :==: a

-- | An empty set of constraints.
empty :: Constrs a

-- | Adds one constraint to the set, returns the new set of constraints
--   if consistent.
addConstr :: Constr a -> Constrs a -> Maybe (Constrs a)
```

C | Addendum

After the submission of this report the author discovered a problem in the treatement of OTT for what concerns user defined types. For future reference we keep the report intact as submitted, and in this section we illustrate the problem and sketch a solution.

In Section 6.5.3, we decompose equalities between user defined types be equating all the parameters of the type constructors. This is certainly enough to coerce values of those types. In fact, always equating all the type parameters is more than we need, and this gives rise to problems with the hierarchy.

Pretending again that we had explicit levels, consider the type

$$\underline{\text{record}}\ \mathsf{Foo}\ (A{:}\mathsf{Type}_1)\ \underline{\text{where}}\ \{\}$$

Now, the type of the type constructor Foo might very well be

$$\mathsf{Foo} : (A{:}\mathsf{Type}_1) \to \mathsf{Type}_0$$

We can have Type_0 since we do not use the type parameter in any data constructor, and as remarked in Section 6.4 the overall level must be at least as large to contain the types of all the data in the data constructors.

Now, the problem is that when we equate two types formed by Foo, the equality will be smaller than necessary:

$$[\![(\mathsf{Foo}\ A_1{:}\mathsf{Type}_0) = (\mathsf{Foo}\ A_2{:}\mathsf{Type}_0)]\!] : \mathsf{Type}_0 \rightsquigarrow$$
$$[\![(A_1{:}\mathsf{Type}_1) = (A_2{:}\mathsf{Type}_1)]\!] : \mathsf{Type}_1$$

This breaks subject reduction.

The easiest 'solution' is to make the type constructors return larger types, keeping in consideration parameters even if they are not used. However, as already noted, keeping things small when possible is convenient, and here we ought to be able to do so since when coercing we will not need equalities such as the one generated above.

In fact, it is worth asking if it is the case that, for example, $\mathsf{Foo}\,\mathbb{N} = \mathsf{Foo}\,\mathsf{Bool}$. If we were to mock Foo in a core theory, we could have

$$\mathsf{Foo'} : \mathsf{Type}_1 \to \mathsf{Type}_0$$
$$\mathsf{Foo'}\ A \mapsto \top$$

Foo' is isomorphic to Foo, and it is certainly the case that $\mathsf{Foo'}\,\mathbb{N} = \mathsf{Foo'}\,\mathsf{Bool}$. In general it makes sense to identify defined types that 'discard' parameters as equal, so that we would have

$$\mathsf{Foo}\ A_1 = \mathsf{Foo}\ A_2 \rightsquigarrow \top$$

In fact, the current Agda implementation has such a ceck, although with other aims.[27]

However this is not enough for our purposes. Consider the other example

$$\textbf{some-type} : \mathsf{Type}_1$$
$$\textbf{some-type} \mapsto \cdots$$

$$\underline{\text{data}}\ \mathsf{Bar}\ (F{:}\mathsf{Type}_1 \to \mathsf{Type}_0)\ \underline{\text{where}}\ \{\mathsf{bar} : F\ \textbf{some-type}\}$$

Here F is definitely relevant to Bar, but requiring the parameter itself to be equal is again more than we need and infringes subject reduction, since to equate two Fs we will need to

[27] See the notes about polarity at http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Version-2-3-2.

abstract over things of type Type_1, which are larger than types formed by Bar itself. What we really want here is

$$(\mathsf{Bar}\ F_1{:}\mathsf{Type}_0) = (\mathsf{Bar}\ F_2{:}\mathsf{Type}_0) \rightsquigarrow (F_1\ \textbf{some-type}{:}\mathsf{Type}_0) = (F_2\ \textbf{some-type}{:}\mathsf{Type}_0)$$

A more convincing solution to treat cases like the one above is to look directly at the data constructors to find out what equalities to require, seeing the equalities as a mean to coerce, as the original OTT paper does (Altenkirch *et al.* , 2007). From this perspective we can equate the type of each element in the data constructors, abstracting over variables of the previous elements' types.

The problematic part in the plan above is dealing with recursive occurrences of type in the data constructors. Intuitively we never need additional equalities for recursive occurrences. When coercing, irrelevant parameters (parameters to the type constructors which are not used in the data constructors) do not matter, since when constructing new data we will be able to decide whatever parameter we want. With relevant parameters, we know that we have the equalities we need somewhere, since we equate all the types of the data in the data constructors.

For example, if **incr** is the function that increments a number by one, and $\circ$ is function composition, we might have

$$\underline{\mathsf{data}}\ \mathsf{Bar}\ (F{:}\mathbb{N} \to \mathsf{Type})\ \underline{\mathsf{where}}\ \{\mathsf{foo} : (n{:}\mathbb{N})\ (F\ n)\ (\mathsf{Foo}\ (F \circ \textbf{incr}))\}$$

Given

$$\mathsf{Bar}\ F_1 = \mathsf{Bar}\ F_2$$

We would demand evidence that

$$\forall x_1{:}\mathbb{N}.\ \forall x_2{:}\mathbb{N}.\ (x_1{:}\mathbb{N}) = (x_2{:}\mathbb{N}) \Rightarrow (F_1\ x_1{:}\mathsf{Type}) = (F_2\ x_2{:}\mathsf{Type})$$

Which lets us prove, using refl and a congruence rule, that

$$(\mathsf{Bar}\ (F_1 \circ \textbf{incr}){:}\mathsf{Type}) = (\mathsf{Bar}\ (F_2 \circ \textbf{incr}){:}\mathsf{Type})$$

The details of the algorithm to generate equalities and perform coercions remain to be worked out, and raise interesting points regarding when two inductively defined types are equal.

Bibliography

Abel, Andreas, Coquand, Thierry, & Dybjer, Peter. 2007. Normalization by evaluation for Martin-Lof type theory with typed equality judgements. *Pages 3–12 of: Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*. IEEE.

Altenkirch, Thorsten, McBride, Conor, & Swierstra, Wouter. 2007. Observational equality, now! *Proceedings of the 2007 workshop on Programming languages meets program verification - PLPV '07*, 57.

Barendregt, Henk. 1991. Introduction to generalized type systems. *Journal of functional programming*.

Bird, Richard S., & Paterson, Ross. 1999. De Bruijn notation as a nested datatype. *J. Functional Programming*, **9**(1), 77–91.

Brady, Edwin. 2013. Implementing General Purpose Dependently Typed Programming Languages. *Unpublished draft*.

Capretta, Venanzio. 2005. General recursion via coinductive types. *Logical Methods in Computer Science*, **1**(2).

Chakravarty, Manuel MT, Keller, Gabriele, Jones, Simon Peyton, & Marlow, Simon. 2005. Associated types with class. *Pages 1–13 of: ACM SIGPLAN Notices*, vol. 40. ACM.

Chapman, James, Dagand, Pierre-Évariste, McBride, Conor, & Morris, Peter. 2010. The gentle art of levitation. *Pages 3–14 of: ACM Sigplan Notices*, vol. 45. ACM.

Church, Alonzo. 1936. An unsolvable problem of elementary number theory. *American journal of mathematics*, **58**(2), 345–363.

Cockett, Robin, & Fukushima, Tom. 1992. *About charity*. Tech. rept.

Constable, Robert L., & the PRL Group. 1986. *Implementing Mathematics with The NuPRL Proof Development System*.

Coquand, Thierry. 1992. Pattern matching with dependent types. *Pages 66–79 of: Informal proceedings of Logical Frameworks*, vol. 92.

Coquand, Thierry, & Huet, Gerard. 1986. The calculus of constructions.

Curry, Haskell B. 1934. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, **511**(1930), 584–590.

Dagand, Pierre-Evariste, & McBride, Conor. 2012. Elaborating inductive definitions. *arXiv preprint arXiv:1210.6390*.

Danielsson, Nils Anders. 2012. Operational semantics using the partiality monad. *SIGPLAN Not.*, **47**(9), 127–138.

de Bruijn, Nicolaas Govert. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Pages 381–392 of: Indagationes Mathematicae (Proceedings)*, vol. 75. Elsevier.

de Bruijn, Nicolaas Govert. 1991. Telescopic mappings in typed lambda calculus. *Information and Computation*, **91**(2), 189–204.

Dybjer, Peter. 1991. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. *Logical Frameworks*.

Dybjer, Peter. 1997. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, **176**(1), 329–335.

Dybjer, Peter. 2000. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 525–549.

Giménez, Eduardo. 1995. Codifying guarded definitions with recursive schemes. *Pages 39–59 of:* Dybjer, Peter, Nordström, Bengt, & Smith, Jan (eds), *Types for Proofs and Programs*. Lecture Notes in Computer Science, vol. 996. Springer Berlin Heidelberg.

Giménez, Eduardo. 1996. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. Ph.D. thesis, Ecole Normale Supérieure de Lyon.

Gundry, Adam, & McBride, Conor. 2013. *A tutorial implementation of dynamic pattern unification*. Unpublished draft.

Harper, Robert, & Pollack, Robert. 1991. Type checking with universes. *Theoretical computer science*, **89**(1), 107–136.

Henglein, Fritz. 1993. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **15**(2), 253–289.

Hofmann, Martin, & Streicher, Thomas. 1994. The groupoid model refutes uniqueness of identity proofs. *Pages 208–212 of: Logic in Computer Science, 1994. LICS'94. Proceedings., Symposium on*. IEEE.

Huet, Gerard P. 1973. The undecidability of unification in third order logic. *Information and Control*, **22**(3), 257–267.

Huet, Gerard P. 1988. *Extending The Calculus of Constructions with Type:Type*. Unpublished draft.

Hurkens, Antonius J.C. 1995. A simplification of Girard's paradox. *Typed Lambda Calculi and Applications*.

Hutton, Graham. 2007. *Programming in Haskell*. Cambridge University Press.

Jacobs, Bart. 1994. Quotients in Simple Type Theory. *Manuscript, Math. Inst*.

King, David J, & Launchbury, John. 1995. Structuring depth-first search algorithms in Haskell. *Pages 344–354 of: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM.

Lipovača, Miran. 2009. *Learn You a Haskell for Great Good!* http://learnyouahaskell.com/.

Löh, Andres, McBride, Conor, & Swierstra, Wouter. 2010. A Tutorial Implementation of a Dependently Typed Lambda Calculus. *Fundam. Inform.*, **102**(2), 177–207.

Marlow, Simon. 2010. *Haskell 2010, Language Report*. http://www.haskell.org/onlinereport/haskell2010/.

Martin-Löf, Per. 1984. *Intuitionistic type theory*. Bibliopolis.

McBride, C., & McKinna, J. 2004a. The view from the left. *Journal of Functional Programming*, **14**(1), 69–111.

McBride, Conor. 1999. *Dependently typed functional programs and their proofs*. Ph.D. thesis, University of Edinburgh.

McBride, Conor. 2004. *Epigram: Practical Programming with Dependent Types*. http://strictlypositive.org/epigram-notes.ps.gz.

McBride, Conor. 2009. Let's see how things unfold: Reconciling the infinite with the intensional. *Pages 113–126 of: Algebra and Coalgebra in Computer Science*. Springer.

McBride, Conor, & McKinna, James. 2004b. I am not a number: I am a free variable. *In: Proceedings of the ACM SIGPLAN Haskell Workshop*.

Miller, Dale. 1992. Unification under a mixed prefix. *Journal of symbolic computation*, **14**(4), 321–358.

Milner, Robin. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences*, **17**(3), 348–375.

Nordström, Bengt, Petersson, Kent, & Smith, Jan M. 1990. *Programming in Martin-Löf Type Theory: An Introduction*. Oxford University Press.

Norell, Ulf. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology and Göteborg University.

Paulson, Lawrence C. 1990. Isabelle: The Next 700 Theorem Provers. *Pages 361–386 of:* Odifreddi, P. (ed), *Logic and Computer Science*. Academic Press.

Pierce, Benjamin C. 2002. *Types and Programming Languages*. The MIT Press.

Pierce, Benjamin C., & Turner, David N. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems*, **22**(1), 1–44.

Queinnec, Christian. 2003. *Lisp in Small Pieces*. Cambridge University Press.

Reynolds, John C. 1994. An introduction to the polymorphic lambda calculus. *Logical Foundations of Functional Programming*.

Sulzmann, Martin, Chakravarty, Manuel M. T., Jones, Simon Peyton, & Donnelly, Kevin. 2007. System F with type equality coercions. *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation - TLDI '07*, 53.

Tait, William W. 1967. Intensional Interpretations of Functionals of Finite Type. *The Journal of Symbolic Logic*, **32**(2), 198–212.

The Coq Team. 2013. *The Coq Proof Assistant*. http://coq.inria.fr.

The GHC Team. 2012. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.3*. http://www.haskell.org/ghc/docs/7.6.3/html/users_guide/.

Thompson, Simon. 1991. *Type Theory and Functional Programming*. Addison-Wesley.

Weirich, Stephanie, Yorgey, Brent A, & Sheard, Tim. 2011. Binders unbound. *Pages 333–345 of: ACM SIGPLAN Notices*, vol. 46. ACM.

Whitehead, Alfred North, & Russell, Bertrand Arthur William. 1927. *Principia mathematica; 2nd ed.* Cambridge: Cambridge Univ. Press.