Inductive Data Types: Well-ordering Types Revisited

Healfdene Goguen Zhaohui Luo
LFCS, Dept. of Computer Science, Edinburgh Univ.

Abstract

We consider Martin-Löf's well-ordering type constructor in the context of an impredicative type theory. We show that the well-ordering types can represent various inductive types faithfully in the presence of the filling-up equality rules or η-rules. We also discuss various properties of the filling-up rules.

1 Introduction

Type theory is on the edge of two disciplines, constructive logic and computer science. Logicians see type theory as interesting because it offers a foundation for constructive mathematics and its formalization. For computer scientists, type theory promises to provide a uniform framework for programs, proofs, specifications, and their development. From each perspective, incorporating a general mechanism for inductively defined data types into type theory is an important next step.

Various type-theoretic approaches to inductive data types have been considered in the literature, both in Martin-Löf's predicative type theories (*e.g.*, [ML84, Acz86, Dyb88, Dyb91, Bac88, Men88]) and in impredicative type theories (*e.g.*, [BB85, Men87, CPM90, Ore90]). In this paper, we consider *well-ordering types* (W-types), introduced by Martin-Löf in his extensional type theory [ML84], in the context of an impredicative type theory where the notions of logical proposition and data type are distinguished.

Well-ordering types in Martin-Löf's extensional type theory can be used to represent various inductive data types (see, for example, [Dyb88]), due to the strong equality type $I(A, a, b)$ with the following elimination rule

$$\frac{\Gamma \vdash p : I(A, a, b)}{\Gamma \vdash a = b : A}$$

This type imposes a strong (or full) extensionality which allows us to assert a judgemental equality using the induction principle. However, there is a known problem in using W-types to represent inductive types in an intensional type theory (for example, Martin-Löf's intensional type theory [NPS90]): the W-representations of inductive types do not give the expected induction principle, because there is 'junk' in the representations (see chapter 15 of [NPS90] and explanations in section 2.2). In other words, W-types may not be used in an intensional type theory to represent inductive data types faithfully — some extensionality is required.

A basic observation in this paper is that, for W-types to represent inductive types faithfully, it is sufficient (and necessary) to have the *filling-up equality rules* (or η-rules) for the type constructors used in the W-type constructions. Although weaker than the strong equality which leads to strong extensionality, the filling-up rules provide sufficient uniqueness conditions (perhaps called weak extensionality) so that the representations of inductive types by W-types are faithful. Based on this observation, we consider incorporating W-types into the Extended Calculus of Constructions [Luo90a] and show that a large class of inductive data types can be faithfully represented by W-types with the help of filling-up rules.

Filling-up equality rules are not viewed as computational (or definitional). Their introduction may have certain side-effects: in particular, some meta-theoretic properties for intensional type theories may be invalid. We discuss several aspects of this and hope that such an attempt at reconsidering well-ordering types leads to a better understanding of inductive data types.

1.1 Motivations and related work

The general motivation for considering inductive data types lies in the idea of developing type theory as a uniform language for modular development of programs, proofs and specifications. The particular approach we have taken in developing type theories for this is based on the idea that, even in type theory, there should be a distinction between the notions of logical formula and data type. This idea has been reflected in the development of the Extended Calculus of Constructions (**ECC**) [Luo90a], where higher-order logical propositions reside in the impredicative universe (*c.f.*, the calculus of constructions [CH88]), while the data types (or sets) reside in the predicative universes. As in Martin-Löf's type theory, the predicative universes are supposed to be open in the sense that new types or type constructors (for example, inductive data types) may be added to the predicative world when they are needed. However, the impredicative world of logical propositions is viewed as relatively closed, in the sense that the ways in which logical propositions are formed are not supposed to be further extended. This view has some theoretical and pragmatic consequences. For example, since the logical universe is relatively closed, there is a proper notion of predicate (as propositional function) in the theory which is used in developing an approach to program specification and development [BM91, Luo91b] and a notion of mathematical theory in the application of abstract reasoning [Luo91a, Pol90]. On the other hand, since data types (sets) reside in predicative universes, the embedded logic is a conservative extension of higher-order predicate logic [Luo90b], and we are able to introduce type constructors like the strong sum (large Σ-types), which is important in abstract and modular development of programs and proofs.

It is obvious that, for a type theory like **ECC** to be useful in real applications such as program and proof development, we need to consider various concrete data types as types in the predicative universes (*c.f.*, similar research for Martin-Löf's type theory, for example, [ML84, Acz86, Dyb88, Dyb91]). Pollack's Lego proof development system allows users to introduce new inductive data types in a style similar to ALF [ACN90], which conforms with Martin-Löf's philosophical openness as explained above. However, it is necessary in applications to have a general mechanism in the theory which provides a large class of inductive data types.[1]

[1]We note that although the impredicative representation of data types (see [BB85]) is available in an impredicative type system like **ECC**, the representations are too weak.

Such studies of general mechanisms for inductive data types in the context of impredicative type theory have been considered recently in the literature. Coquand and Mohring [CPM90] have introduced a general schematic notion of inductive types and studied its model theory by considering an intermediate system which is essentially an extension of **ECC** by a fix-point type constructor. They have also used certain filling-up rules for the system. Ore [Ore90] has also considered extending **ECC** by a fix-point type constructor to incorporate inductive data types and given a realizability model for the resulting system. The proof-theoretic properties, such as strong normalization and confluence, of such systems are still under research.

In this line of research, the well-ordering type seems to be a natural alternative to consider as a general mechanism for inductive data types. W-types are simple, and using them has some advantages. The W-type is introduced by a single type constructor which does not need extra meta-level notions, such as that of strictly positive operator, for its introduction. It conforms to the general pattern of types in Martin-Löf type theory. Furthermore, its rules and the η-rules are readily understood, since their meaning is intuitively clear. Another point is that the introduction of W-types conforms with the intuitive idea of predicative types in type theory being developed in non-circular stages (*c.f.*, [ML84]) — we first have certain type constructors (*e.g.*, Π, Σ and W) and then use the reflection principle to introduce the names of these types into (predicative) universes. In contrast, the meta-theoretic notions of strictly positive operator is needed for the introduction of a least fix-point type constructor, and type universes are used in the works cited above to introduce fix-point types.

In section 2, we discuss the well-ordering type and the filling-up rules and their use in the representation of data types. We discuss in section 3 the expressiveness of W-types in representing inductive types by relating them to certain least fix-point types. In section 4, we consider well-ordering types with explicit least elements. We discuss some properties of the filling-up rules in the final section.

2 Well-ordering Types and Filling-up Rules

Appendix A presents an intensional type system ITT, which is essentially the system **ECC** extended by the empty type, the unit type and disjoint sums[2]. In this section and section 3, we consider introducing W-types and filling-up rules into this system. We adopt the conventions as described in Appendix A when presenting rules below and also use the following notational abbreviations.

Notations As usual, when $x \notin FV(B)$, we may write $A \to B$ for $\Pi x{:}A.B$ and $A \times B$ for $\Sigma x{:}A.B$. We also write $f a_1 ... a_n$ or $f(a_1, ..., a_n)$ for $\mathbf{app}(\mathbf{app}(...\mathbf{app}(f, a_1), ..., a_{n-1}), a_n)$, we omit the pairing operator, and we write $\lambda x{:}A.b$ to abbreviate $\lambda[\Pi x{:}A.B]x.b$ when making the domain of a function explicit.

We use $\circ$ to denote the following composition operator of functions: for f of type $\Pi x_1{:}X_1...\Pi x_n{:}X_n.Y$ with $x_{i_1}, ..., x_{i_k} \in FV(Y)$ and g of type $\Pi x_{i_1}{:}X_{i_1}...\Pi x_{i_k}{:}X_{i_k}\Pi y{:}Y.Z$,

$$g \circ f =_{\mathrm{df}} \lambda x_1 ... \lambda x_n.\ g(x_{i_1}, ..., x_{i_k})(f(x_1, ..., x_n))$$

which is of type $\Pi x_1{:}X_1...\Pi x_n{:}X_n.\ [f(x_1, ..., x_n)/y]Z$.

[2] The type systems considered in this paper are subsystems of that in [Luo91c], where the second author discusses how subtyping between universes as in **ECC** may be introduced.

formation	$\dfrac{\Gamma \vdash A\ \mathbf{type} \quad \Gamma, x{:}A \vdash B\ \mathbf{type}}{\Gamma \vdash Wx{:}A.B\ \mathbf{type}}$	$\dfrac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash B = B'}{\Gamma \vdash Wx{:}A.B = Wx{:}A'.B'}$
introduction	$\dfrac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash b : [a/x]B \to Wx{:}A.B\end{array}}{\Gamma \vdash \mathbf{sup}[Wx{:}A.B](a,b) : Wx{:}A.B}$	$\dfrac{\begin{array}{c}\Gamma \vdash a = a' : A \\ \Gamma \vdash b = b' : [a/x]B \to Wx{:}A.B\end{array}}{\Gamma \vdash \mathbf{sup}(a,b) = \mathbf{sup}(a',b') : Wx{:}A.B}$
elimination	$\dfrac{\Gamma \vdash f : \Pi x{:}A \Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C}{\Gamma \vdash \mathbf{E}_W[\Pi z{:}(Wx{:}A.B).C](f) : \Pi z{:}(Wx{:}A.B).C}$	
	$\dfrac{\Gamma \vdash f = f' : \Pi x{:}A \Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C}{\Gamma \vdash \mathbf{E}_W(f) = \mathbf{E}_W(f') : \Pi z{:}(Wx{:}A.B).C}$	
computation	$\dfrac{\begin{array}{c}\Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B \to Wx{:}A.B \quad \Gamma, z{:}Wx{:}A.B \vdash C\ \mathbf{type} \\ \Gamma \vdash f : \Pi x{:}A \Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C\end{array}}{\Gamma \vdash \mathbf{E}_W(f)(\mathbf{sup}(a,b)) = f(a,b,\mathbf{E}_W(f) \circ b) : [\mathbf{sup}(a,b)/z]C}$	

Figure 1: Rules for W-types.

2.1 Martin-Löf's well-ordering types

Martin-Löf [ML84] introduces a type constructor W which is based on well-founded trees, whose rules are given in Figure 1. The formation rule for the W-type depends upon two types. The first of these types we think of as specifying the constructors for our new type. The second corresponds to a branching or selector type for each constructor. Given these intuitions, we can view the **sup** constructor as representing, for each constructor in the inductive type, the least upper bound of the elements indexed by a function from the branching type into the well-ordering. This could also be seen as creating a new node in a tree by giving a function which returns all the subnodes of this new node. The elimination rule states that if any property holding for all subnodes of a node implies that it holds for that node, then the property holds for all elements of the well-ordering.

The induction principle holds for the well-ordering types because in general some of the branching types are the empty type, giving a trivial set of predecessors. If our theory does not include the empty type, none of the W-types has an inhabitant, because there are no least elements with which to provide a basis.

2.2 A problem of faithful representation

First, let us explain the problem of faithful representation by considering the simple example of the natural numbers (*c.f.*, [NPS90]). Its W-type representation is

$$N =_{\mathrm{df}} Wx{:}A_N.B_N\ ,$$

where $A_N \equiv \mathbf{1} + \mathbf{1}$ and $B_N(x)$[3] is defined such that $B_N(\mathbf{i}(a)) = \emptyset$ and $B_N(\mathbf{j}(b)) = \mathbf{1}$[4]. The canonical natural numbers are defined as

$$0 =_{\mathrm{df}} \mathbf{sup}(\mathbf{i}(*), \mathbf{E}_\emptyset) \quad \text{and} \quad succ(n) =_{\mathrm{df}} \mathbf{sup}(\mathbf{j}(*), \mathbf{E}_\mathbf{1}(n))\ .$$

Unfortunately, in an intensional type theory, the canonical natural numbers are not the only objects of type N. For example, there is no way to show that

$$\mathbf{sup}(\mathbf{i}(*), \lambda x{:}\emptyset.0) = 0\ ,$$

[3] For readability, we shall write $B(a)$ for $[a/x]B$, where $x : A \vdash B\ \mathbf{type}$.

[4] Formally, $B_N(x) \equiv \mathbf{T}_0(\mathbf{E}_+(\mathbf{E}_\mathbf{1}(\emptyset), \mathbf{E}_\mathbf{1}(1))(x))$, where $\emptyset$ is the name of the empty type and 1 is the name of the unit type.

because $\lambda x{:}\emptyset.0$ and $\mathbf{E}_\emptyset$ are not computationally equal. The W-representation of the natural numbers is faithful only if we identify such extra objects with the canonical ones so that the intended elimination rule is derivable. In fact, if we consider rules (filling-up rules) like $(F_\emptyset)$: $f = \mathbf{E}_\emptyset$, where f is a function with domain $\emptyset$, and $(F_\mathbf{1})$: $f = \mathbf{E}_\mathbf{1}(f(*))$, where f is a function with domain $\mathbf{1}$, then we can derive the expected elimination rule, since the elimination operator can be defined as

$$rec_N(c, f) =_{\mathrm{df}} \mathbf{E}_W(f') \ ,$$

where $f' \equiv \mathbf{E}_+(\mathbf{E}_\mathbf{1}(\lambda y\lambda z.c), \mathbf{E}_\mathbf{1}(\lambda y\lambda z.f(y(*), z(*))))$. $rec_N(c, f)$ satisfies the following rule:

$$\frac{\Gamma, n{:}N \vdash C \ \mathbf{type} \quad \Gamma \vdash c : C(0) \quad \Gamma \vdash f : \Pi n{:}N.\ C(n) \to C(succ(n))}{\Gamma \vdash rec_N(c, f) : \Pi n{:}N.C(n)}$$

and the equalities $rec_N(c, f)(0) = c$ and $rec_N(c, f)(succ(n)) = f(n, rec_N(c, f)(n))$ hold. To see that $rec_N(c, f)$ has the correct type, we have to check that the function f' above has type $\Pi x{:}A_N.D(x)$, where

$$D(x) \equiv \Pi y{:}B_N(x) \to N.(\Pi v{:}B_N(x).C(yv) \to C(\mathbf{sup}(x, y)) \ .$$

This is the case since we have

1. $\mathbf{E}_\mathbf{1}(\lambda y\lambda z.c)$ is of type $\Pi x'{:}\mathbf{1}.\ D(\mathbf{i}(x'))$, since $\lambda y\lambda z.c$ is of type

$$\begin{aligned}
& \Pi y{:}\emptyset \to N.\ (\Pi v{:}\emptyset.C(yv)) \to C(0) \\
\equiv\ & \Pi y{:}\emptyset \to N.\ (\Pi v{:}\emptyset.C(yv)) \to C(\mathbf{sup}(\mathbf{i}(*), \mathbf{E}_\emptyset)) \\
\overset{(\mathrm{F}_\emptyset)}{=}\ & \Pi y{:}\emptyset \to N.\ (\Pi v{:}\emptyset.C(yv)) \to C(\mathbf{sup}(\mathbf{i}(*), y)) \\
=\ & \Pi y{:}B_N(\mathbf{i}(*)) \to N.\ (\Pi v{:}B_N(\mathbf{i}(*)).C(yv)) \to \ C(\mathbf{sup}(\mathbf{i}(*), y)) \\
\equiv\ & D(\mathbf{i}(*)) \ .
\end{aligned}$$

2. $\mathbf{E}_\mathbf{1}(\lambda y\lambda z.f(y(*), z(*)))$ is of type $\Pi y'{:}\mathbf{1}.\ D(\mathbf{j}(y'))$, since $\lambda y\lambda z.f(y(*), z(*))$ is of type

$$\begin{aligned}
& \Pi y{:}\mathbf{1} \to N.\ (\Pi v{:}\mathbf{1}.C(yv)) \to C(succ(y(*))) \\
\equiv\ & \Pi y{:}\mathbf{1} \to N.\ (\Pi v{:}\mathbf{1}.C(yv)) \to C(\mathbf{sup}(\mathbf{j}(*), \mathbf{E}_\mathbf{1}(y(*)))) \\
\overset{(\mathrm{F}_\mathbf{1})}{=}\ & \Pi y{:}\mathbf{1} \to N.\ (\Pi v{:}\mathbf{1}.C(yv)) \to C(\mathbf{sup}(\mathbf{j}(*), y)) \\
=\ & \Pi y{:}B_N(\mathbf{j}(*)) \to N.\ (\Pi v{:}B_N(\mathbf{j}(*)).C(yv)) \to C(\mathbf{sup}(\mathbf{j}(*), y)) \\
\equiv\ & D(\mathbf{j}(*)) \ .
\end{aligned}$$

2.3 Filling-up equality rules

From the above example, we can see that the key problem with faithful representation via W-types is the lack of the filling-up rules. The reason that W-types can be used to represent inductive data types faithfully in Martin-Löf's extensional type theory is that the filling-up rules are derivable using the elimination rule for the strong equality.

The filling-up rules for the type constructors in ITT are given in Figure 2. Note

$(F_\emptyset)$	$\dfrac{}{\Gamma \vdash f = \mathbf{E}_\emptyset : \Pi z{:}\emptyset.C}$
$(F_\mathbf{1})$	$\dfrac{}{\Gamma \vdash f = \mathbf{E}_\mathbf{1}(f(*)) : \Pi z{:}\mathbf{1}.C}$
(F_+)	$\dfrac{inl =_{\mathrm{df}} \lambda x.\mathbf{i}(x), \quad inr =_{\mathrm{df}} \lambda y.\mathbf{j}(y)}{\Gamma \vdash f = \mathbf{E}_+(f \circ inl, f \circ inr) : \Pi z{:}A+B.C}$
(F_Σ)	$\dfrac{pair =_{\mathrm{df}} \lambda x \lambda y.\mathbf{p}(x,y)}{\Gamma \vdash f = \mathbf{E}_\Sigma(f \circ pair) : \Pi z{:}(\Sigma x{:}A.B).C}$
(η)	$\dfrac{x \notin FV(f)}{\Gamma \vdash f = \lambda x.fx : \Pi x{:}A.B}$

Figure 2: The filling-up equality rules.

that there is a general form of the filling-up rules, except (η)[5]: informally, for a type constructor $\mathcal{K}$ with introduction operators $c_1, ..., c_n$ and elimination operator $\mathbf{E}_\mathcal{K}$, the filling-up rule for $\mathcal{K}$ is of the form

$$f = \mathbf{E}_\mathcal{K}(f \circ c_1, ..., f \circ c_n)$$

where f is a function whose domain is a $\mathcal{K}$-type.

The filling-up rules assert a uniqueness condition for the elimination operator of the corresponding type constructor. It is interesting to note that they are equivalent to the corresponding η-rules for types with one constructor (uniqueness of the object $*$ in the unit type, surjective pairing for Σ-types). For example, in any context with $v{:}\Sigma x{:}A.B(x)$,

$$v = \mathbf{p}(\pi_1 v, \pi_2 v) \ ,$$

where $\pi_1 =_{\mathrm{df}} \mathbf{E}_\Sigma(\lambda x \lambda y.x)$ and $\pi_2 =_{\mathrm{df}} \mathbf{E}_\Sigma(\lambda x \lambda y.y)$, follows because $v = (\lambda z.z)v \overset{(\mathrm{F}_\Sigma)}{=} \mathbf{E}_\Sigma(\lambda x \lambda y.\mathbf{p}(x,y))v = \mathbf{E}_\Sigma(\lambda x \lambda y.(\lambda z.\mathbf{p}(\pi_1 z, \pi_2 z))\mathbf{p}(x,y))v \overset{(\mathrm{F}_\Sigma)}{=} (\lambda z.\mathbf{p}(\pi_1 z, \pi_2 z))v = \mathbf{p}(\pi_1 v, \pi_2 v)$. For types with other than one constructor, the commuting conversions (see [GLT90]) follow from the filling-up rules.

We call the system described in Appendix A extended by W-types and the filling-up rules in figure 2 $ITT_\eta + W$ (a subsystem of that described in [Luo91c]). Note that we have tried to use the minimum number of filling-up rules necessary for the representation of inductive data types: we have not included such rules for propositions or the W-types. We remark that $ITT_\eta + W$ is model-theoretically consistent, as the realizability models as described in [CPM90, Ore90] show: in particular, the filling-up equality rules are valid in the ω–**Set** realizability model.

Another important remark is that, in the *intensional* system ITT without the filling-up rules, all of the filling-up equalities except $(F_\emptyset)$ for the empty type hold for closed terms. In fact, the corresponding logical propositions, of the form

$$\forall f{:}(\Pi z{:}\mathcal{K}.C).\forall k{:}\mathcal{K}.\ fk =_{C(k)} \mathbf{E}_\mathcal{K}(f \circ c_1, ..., f \circ c_n)k$$

[5]This is because, in our presentation, Π-types are also used as a meta-level mechanism playing a role similar to that of a logical framework. If we used a meta-language instead, there is a filling-up rule for Π of the above form.

where $=_{C(k)}$ is Leibniz equality, are provable using the logical induction principle as expressed by the elimination rules[6]. It follows by an equality reflection result (*c.f.*, [Luo90a]) that the filling-up equality holds for closed terms. The situation is similar for the weak equality in Martin-Löf's intensional type theory. However, this is *not* the case for the filling-up rule for the empty type, which we discuss further in section 4.

2.4 More examples of inductive data types

We consider several other examples to demonstrate the use of the well-ordering type and its interaction with the filling-up rules.

Examples

- The type of lists over the natural numbers:

  $$List(N) =_{\mathrm{df}} Wx{:}A_{List(N)}.B_{List(N)}\ ,$$

  where $A_{List(N)} \equiv \mathbf{1} + N$ and $B_{List(N)}(\mathbf{i}(u)) = \emptyset$ and $B_{List(N)}(\mathbf{j}(n)) = \mathbf{1}$. The canonical lists are defined as

  $$nil =_{\mathrm{df}} \mathbf{sup}(\mathbf{i}(*), \mathbf{E}_\emptyset) \quad \text{and} \quad cons(n,l) =_{\mathrm{df}} \mathbf{sup}(\mathbf{j}(n), \mathbf{E_1}(l))\ .$$

  and the elimination operator as, for $c : C(nil)$ and $f : \Pi n{:}N\Pi l{:}List(N).\ C(l) \to C(cons(n,l))$,

  $$rec_{List(N)}(c,f) =_{\mathrm{df}} \mathbf{E}_W(\mathbf{E}_+(\mathbf{E_1}(\lambda y\lambda z.c), \lambda n\lambda y\lambda z.f(n, y(*), z(*))))$$

  which is of type $\Pi l{:}List(N).C(l)$ (we need the filling-up rules for the empty type and the unit type to verify this) and satisfies the equalities $rec_{List(N)}(c,f)(nil) = c$ and $rec_{List(N)}(c,f)(cons(n,l)) = f(n,l,rec_{List(N)}(c,f)(l))$.

- We can similarly define binary trees as

  $$BT(N) =_{\mathrm{df}} Wx{:}A_{BT(N)}.B_{BT(N)}\ ,$$

  where $A_{BT(N)} \equiv \mathbf{1} + N$ and $B_{BT(N)}(x)$ is defined so that $B_{BT(N)}(\mathbf{i}(x)) = \emptyset$ and $B_{BT(N)}(\mathbf{j}(n)) = \mathbf{1} + \mathbf{1}$. We only remark that to have the expected elimination rule for binary trees , the filling-up rules for the empty type, the unit type, and disjoint sum are required.

- As a last example, we define the type of expressions in a simple language. Consider the following simple grammar:

  $$e ::= v \,|\, n \,|\, e\ +\ e \,|\, \mathbf{let}\, v = e\, \mathbf{in}\, e\, \mathbf{end}\ .$$

  This can be defined using the well-ordering type as, for any type Id (of identifiers),

  $$Expr(Id) =_{\mathrm{df}} Wx{:}A_{Expr(Id)}.B_{Expr(Id)}\ ,$$

  where $A_{Expr(Id)} \equiv (Id + N) + (\mathbf{1} + Id)$, $B_{Expr(Id)}(\mathbf{i}(a)) = \emptyset$ and $B_{Expr(Id)}(\mathbf{j}(b)) = \mathbf{1} + \mathbf{1}$. Once again, to check that the expected elimination rule is well-typed, we need the filling-up rules for the empty type, the unit type, and disjoint sum.

[6] Subject to the above footnote, if Π is formulated by a meta-language with a strong elimination rule, the logical proposition corresponding to η is also provable.

Parameterized inductive data types can also be defined by means of universes. For example, for any universe U, we can define

$$list_U =_{\mathrm{df}} \lambda a{:}U.\ wx{:}(1 \oplus a).\mathbf{E}_+(\mathbf{E_1}(\emptyset), \mathbf{E_1}(1))(x)$$

Then, we have $\mathbf{T}_U(list_U(a)) = List(\mathbf{T}_U(a))$, for $a : U$.

3 The Expressiveness of Well-ordering Types

Dybjer [Dyb88] has shown that W-types in Martin-Löf's extensional type theory can be used to represent a large class of inductive data types. In this section, we prove a similar result to show that the well-ordering types with the filling-up rules are strong enough to represent all types expressible as a least fix-point type of the form $\mu X.\Phi(X)$, where $\Phi(X)$ is a strictly positive operator built from constant types and X by $\times$, $+$ and $\to$.

The proof of the result is essentially based on the lemma 3.2 below, which shows that certain types are isomorphic to each other in the following sense.

Definition 3.1 *Let A and B be types. A is* isomorphic *to B, notation $A \cong B$, if there exist terms $f{:}A \to B$ and $g{:}B \to A$ such that $g \circ f = id_A$ and $f \circ g = id_B$, where id_A and id_B are the identity functions over A and B.*

Lemma 3.2 *The following isomorphisms hold:*

1. $\mathbf{1} \cong \emptyset \to A$
2. $A \cong \mathbf{1} \to A$
3. $A \cong A \times \mathbf{1}$
4. $A \cong \mathbf{1} \times A$
5. $\Pi x{:}A.\Sigma y{:}B(x).C(x,y) \cong \Sigma f{:}(\Pi x{:}A.B(x)).\Pi x{:}A.C(x, fx)$
6. $\Pi x{:}A.\Pi y{:}B(x).C(x,y) \cong \Pi z{:}(\Sigma x{:}A.B(x)).C(\pi_1 z, \pi_2 z)$
7. $\Sigma x_1{:}A_1.B_1(x_1) + \Sigma x_2{:}A_2.B_2(x_2) \cong \Sigma x{:}A_1 + A_2.B(x)$ *where* $B(\mathbf{i}(x_1)) = B_1(x_1)$ *and* $B(\mathbf{j}(x_2)) = B_2(x_2)$
8. $\Sigma x_1{:}A_1.B_1(x_1) \times \Sigma x_2{:}A_2.B_2(x_2) \cong \Sigma x{:}A_1 \times A_2.B(x)$ *where* $B(x_1, x_2) = B_1(x_1) + B_2(x_2)$
9. $A \to C \cong B \to C$, *if* $A \cong B$

Proof We show cases (1) and (5). The others are similar. For case (1), let $f =_{\mathrm{df}} \mathbf{E_1}(\mathbf{E}_\emptyset)$ and $g =_{\mathrm{df}} \lambda v{:}\emptyset \to A.*$. Then we have

$$\begin{array}{rcl} g \circ f & \equiv & \lambda v.(\lambda v.*)(fv) = \lambda x.* = \mathbf{E_1}(id_\mathbf{1} *) \overset{(\mathrm{F}_\mathbf{1})}{=} id_\mathbf{1} \\ f \circ g & = & \lambda v.(\mathbf{E_1}(\mathbf{E}_\emptyset))* = \lambda v.\mathbf{E}_\emptyset \overset{(\mathrm{F}_\emptyset)}{=} id_{\emptyset \to A} \end{array}$$

For case (5), let

$$
\begin{aligned}
f &=_{\mathrm{df}} \lambda v : (\Pi x{:}A.\Sigma y{:}B(x).C(x,y)).\mathbf{p}(\pi_1 \circ v, \pi_2 \circ v) \\
g &=_{\mathrm{df}} \lambda v : (\Sigma f{:}(\Pi x{:}A.B(x)).\Pi x{:}A.C(x,fx))\lambda x{:}A.\mathbf{p}((\pi_1 v)x, (\pi_2 v)x)
\end{aligned}
$$

Then

$$
\begin{aligned}
g \circ f &= \lambda v \lambda x.\mathbf{p}(\pi_1(vx), \pi_2(vx)) = \lambda v \lambda x.vx \overset{\eta}{=} id_{\Pi x:A.\Sigma y:B(x).C(x,y)} \\
f \circ g &= \lambda v.\mathbf{p}(\pi_1 v, \pi_2 v) = id_{\Sigma f:(\Pi x:A.B(x))\Pi x:A.C(x,fx))}
\end{aligned}
$$

We now show how to represent the least fix-point of a strictly positive operator in $ITT_\eta + W$. Thus, we assume a definition for strictly positive operators $\Phi(X)$ constructed with constant types and X by $\to$, $\times$, $+$, as well as the functorial extension of such operators. For each such $\Phi(X)$ we shall define the type $\mu X.\Phi(X)$ as a W-type such that we can define the introduction and elimination operators satisfying the following rules:

$$\frac{\Gamma \vdash x : \Phi(\mu X.\Phi(X))}{\Gamma \vdash intro_\Phi(x) : \mu X.\Phi(X)}$$

$$\frac{\Gamma, z : \mu X.\Phi(X) \vdash C\,\mathbf{type} \quad \Gamma \vdash u : \Pi x{:}\Phi(\Sigma v{:}(\mu X.\Phi(X)).C(v)).C(intro_\Phi((\Phi(\pi_1))x))}{\Gamma \vdash rec_\Phi(u) : \Pi x{:}(\mu X.\Phi(X)).C(x)}$$

$$\frac{\Gamma, z{:}\mu X.\Phi(X) \vdash C\,\mathbf{type} \quad \Gamma \vdash x{:}\mu X.\Phi(X) \quad \Gamma \vdash u : \Pi x{:}\Phi(\Sigma v{:}(\mu X.\Phi(X)).C(v)).C(intro_\Phi((\Phi(\pi_1))x))}{rec_\Phi(u)(intro_\Phi(x)) = u((\Phi(\lambda z.\mathbf{p}(z, rec_\Phi(u)(z))))x) : C(intro_\Phi(x))}$$

We refer the reader to [CPM90, Ore90] for further details on μ-types.

Our method for defining these least fix-points will be to exhibit an A_Φ and B_Φ such that there is a natural isomorphism $\Phi(X) \cong \Sigma x{:}A_\Phi.(B_\Phi(x) \to X)$. We shall then be able to use this isomorphism to define $\mu X.\Phi(X)$ as $Wx{:}A_\Phi.B_\Phi$.

Theorem 3.3 *For any strictly positive operator $\Phi(X)$, there exist A_Φ and B_Φ such that, letting $F_\Phi(X) =_{\mathrm{df}} \Sigma x{:}A_\Phi.(B_\Phi(x) \to X)$, there is a natural isomorphism $\phi_X : \Phi(X) \cong F_\Phi(X)$. Thus, the following diagram commutes for any $f : X \to Y$:*

$$
\begin{tikzcd}
\Phi(X) \arrow[r, "\phi_X"] \arrow[d, "\Phi(f)"'] & F_\Phi(X) \arrow[d, "F_\Phi(f)"] \\
\Phi(Y) \arrow[r, "\phi_Y"'] & F_\Phi(Y)
\end{tikzcd}
$$

Proof We can define A_Φ and B_Φ by induction on the structure of $\Phi(X)$ as follows:

- $\Phi(X) = K$: $A_\Phi =_{\mathrm{df}} K$ and $B_\Phi(x) =_{\mathrm{df}} \emptyset$.
- $\Phi(X) = X$: $A_\Phi =_{\mathrm{df}} \mathbf{1}$ and $B_\Phi(x) =_{\mathrm{df}} \mathbf{1}$.
- $\Phi(X) = \Phi_1(X) + \Phi_2(X)$: $A_\Phi =_{\mathrm{df}} A_{\Phi_1} + A_{\Phi_2}$ and define B_Φ such that $B_\Phi(\mathbf{i}(x_1)) = B_{\Phi_1}(x_1)$ and $B_\Phi(\mathbf{j}(x_2)) = B_{\Phi_2}(x_2)$.
- $\Phi(X) = \Phi_1(X) \times \Phi_2(X)$: $A_\Phi =_{\mathrm{df}} A_{\Phi_1} \times A_{\Phi_2}$ and $B_\Phi(x_1, x_2) =_{\mathrm{df}} B_{\Phi_1}(x_1) + B_{\Phi_2}(x_2)$.

- $\Phi(X) = K \to \Phi_1(X)$: $A_\Phi =_{\mathrm{df}} K \to A_{\Phi_1}$ and $B_\Phi(f) =_{\mathrm{df}} \Sigma y{:}K.B_{\Phi_1}(fy)$.

Then, we can show the isomorphism and naturality by the same induction principle, using lemma 3.2.

An immediate consequence of the naturality is that

$$\Phi(f) = \phi_Y^{-1} \circ (F_\Phi(f)) \circ \phi_X \tag{1}$$

where ϕ_Y^{-1} is the inverse of ϕ_Y. Also, by definition of the functorial extension of $F_\Phi(X)$,

$$F_\Phi(f) = \lambda p.\mathbf{p}(\pi_1 p, \lambda v.f((\pi_2 p)v)) : F_\Phi(X) \to F_\Phi(Y) \tag{2}$$

We can now define $\mu X.\Phi(X)$ to be $Wx{:}A_\Phi.B_\Phi$, for each strictly positive operator $\Phi(X)$. We define the introduction and elimination operators for $\mu X.\Phi(X)$ as

$$\begin{array}{rcl} intro_\Phi(x) & =_{\mathrm{df}} & \mathbf{sup}(\pi_1(\phi(x)), \pi_2(\phi(x))) \\ rec_\Phi(u) & =_{\mathrm{df}} & \mathbf{E}_W(\lambda a \lambda b \lambda p.u(\phi^{-1}(\mathbf{p}(a, \lambda v.\mathbf{p}(bv, pv))))) \end{array}$$

The definition of $intro_\Phi$ follows from the terms verifying isomorphisms (6) and (9) in lemma 3.2. The definition of rec_Φ follows from the term verifying the isomorphism

$$\begin{array}{rl} & \Pi y{:}(\Sigma x{:}A_\Phi.(B_\Phi(x) \to (\Sigma z{:}(Wx{:}A_\Phi.B_\Phi).C(z)))).C(\mathbf{sup}(\pi_1((F_\Phi(\pi_1))y), \pi_2((F_\Phi(\pi_1))y))) \\ \cong & \Pi x{:}A_\Phi.\Pi y{:}(B_\Phi(x) \to Wx{:}A_\Phi.B_\Phi).(\Pi v{:}B_\Phi(x).C(yv)) \to C(\mathbf{sup}(x, y)) \end{array}$$

applied to $u \circ \phi^{-1}$, which is well-typed by the definition of $intro_\Phi$ and naturality.

Hence,

$$\begin{array}{rl} & rec_\Phi(u)(intro_\Phi(x)) \\ \equiv & \mathbf{E}_W(\lambda a \lambda b \lambda p.u(\phi^{-1}(\mathbf{p}(a, \lambda v.\mathbf{p}(bv, pv)))))(\mathbf{sup}(\pi_1(\phi(x)), \pi_2(\phi(x)))) \\ = & u(\phi^{-1}(\mathbf{p}(\pi_1(\phi(x)), \lambda v.\mathbf{p}(((\pi_2(\phi(x)))v), ((rec_\Phi(u)) \circ (\pi_2(\phi(x))))v)))) \\ = & u(\phi^{-1}(\mathbf{p}(\pi_1(\phi(x)), \lambda v.(\lambda z.\mathbf{p}(z, rec_\Phi(u)(z)))((\pi_2(\phi(x)))v)))) \\ = & u(\phi^{-1}(F_\Phi(\lambda z.\mathbf{p}(z, rec_\Phi(u)(z))))(\phi(x))) \\ = & u((\Phi(\lambda z.\mathbf{p}(z, rec_\Phi(u)(z))))x) \end{array}$$

where the last two lines use the equalities (2) and (1). This justifies our definition of μ-types by W-types.

We note that W-types can also be modelled easily using the least fix-point operator by $Wx{:}A.B =_{\mathrm{df}} \mu X.(\Sigma x{:}A.(B(x) \to X))$. The introduction and elimination rules can be defined in this direction simply by using the isomorphisms of lemma 3.2.

4 Well-ordering Types with Bottom Objects

4.1 A problem with the empty type

We have pointed out that the filling-up rule for the empty type does not correspond to an inductively provable logical proposition, that is, $\forall f{:}(\Pi x{:}\emptyset.C(x)).f =_{\Pi x:\emptyset.C(x)} \mathbf{E}_\emptyset$ is not provable in the intensional system ITT. In fact, with this rule, there is a typable closed term which is not strongly normalizable under the usual β-reduction. For example, in the context $z{:}\emptyset$, any two objects a and b of the same type are equal, since

$$a = (\lambda z{:}\emptyset.a)(z) = \mathbf{E}_\emptyset(z) = (\lambda z{:}\emptyset.b)(z) = b\ .$$

Therefore, for an arbitrary type A, the names of types A and $A \to A$ are equal in the context $z{:}\emptyset$, so we can show $z{:}\emptyset \vdash A = A \to A$, from which it is easy to see that $\vdash \lambda z{:}\emptyset.(\lambda x.xx)(\lambda x.xx) : \emptyset \to A$ is derivable[7].

We do not know whether type-checking and conversion for the system with filling-up rules are decidable. However, the above problem with the empty type and the fact that its filling-up rule does not hold for closed terms in ITT suggest considering a formulation which does not include the filling-up rule for the empty type.

4.2 W-types with bottom objects

Instead of using the empty type to introduce bottom objects for a well-ordering type, we can introduce them explicitly. The rules for the W-types with a bottom object are similar to the original formulation, except that we have introduction rules

$$\frac{}{\Gamma \vdash \bot[Wx{:}A.B] : Wx{:}A.B} \qquad\qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B \to Wx{:}A.B}{\Gamma \vdash \mathbf{sup}[Wx{:}A.B](a,b) : Wx{:}A.B}$$

elimination rule

$$\frac{\Gamma \vdash c : [\bot/z]C \quad \Gamma \vdash f : \Pi x{:}A\Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C}{\Gamma \vdash \mathbf{E}_W[\Pi z{:}(Wx{:}A.B).C](c,f) : \Pi z{:}(Wx{:}A.B).C}$$

and computation rules

$$\frac{\begin{array}{c}\Gamma, z{:}Wx{:}A.B \vdash C\ \mathbf{type} \quad \Gamma \vdash c : [\bot/z]C \\ \Gamma \vdash f : \Pi x{:}A\Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C\end{array}}{\Gamma \vdash \mathbf{E}_W(c,f)(\bot) = c : [\bot/z]C}$$

$$\frac{\begin{array}{c}\Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B \to Wx{:}A.B \quad \Gamma, z{:}Wx{:}A.B \vdash C\ \mathbf{type} \quad \Gamma \vdash c : [\bot/z]C \\ \Gamma \vdash f : \Pi x{:}A\Pi y{:}(B \to Wx{:}A.B).\ (\Pi v{:}B.[yv/z]C) \to [\mathbf{sup}(x,y)/z]C\end{array}}{\Gamma \vdash \mathbf{E}_W(c,f)(\mathbf{sup}(a,b)) = f(a,b,\mathbf{E}_W(c,f)\circ b) : [\mathbf{sup}(a,b)/z]C}$$

We shall call the system obtained from ITT with W-types as formulated above and without the empty type $ITT_\eta + W_\bot$. We now show some examples of inductive data types in this system.

Examples We simply give the definitions and leave the reader to check that the corresponding elimination and computation rules are derivable.

- The type of natural numbers:

  $$N =_{\mathrm{df}} Wx{:}\mathbf{1}.\mathbf{1}$$

  with $0 =_{\mathrm{df}} \bot$, $succ(n) =_{\mathrm{df}} \mathbf{sup}(*, \mathbf{E_1}(n))$, and for $c : C(0)$ and $f : \Pi n{:}N.C(n) \to C(succ(n))$, $rec_N(c,f) =_{\mathrm{df}} \mathbf{E}_W(c, \mathbf{E_1}(\lambda y\lambda z.f(y*, z*)))$. To check that $rec_N(c,f)$ has the expected type, the filling-up rule $(F_\mathbf{1})$ is used.

[7]This is Martin-Löf's example showing that his extensional type theory is not strongly normalizing. In fact, as we show here, the key reason is that the filling-up rule for the empty type is derivable from the strong equality type.

- The type of binary trees:

$$BT(N) =_{\mathrm{df}} Wx{:}N.\mathbf{1} + \mathbf{1}\ ,$$

with $empty =_{\mathrm{df}} \perp$, $node(n,t,t') =_{\mathrm{df}} \mathbf{sup}(n, \mathbf{E}_+(\mathbf{E_1}(t), \mathbf{E_1}(t')))$, and if $e : C(empty)$ and $f : \Pi n{:}N\Pi t,t'{:}BT(N).(C(t)) \to (C(t')) \to C(node(n,t,t'))$, define $rec_{BT(N)}(e,f)$ as $\mathbf{E}_W(e, \lambda x\lambda y\lambda p.f(x, y(\mathbf{i}(*)), y(\mathbf{j}(*)), p(\mathbf{i}(*)), p(\mathbf{j}(*))))$. Showing $rec_{BT(N)}$ to be well-typed requires the filling-up rules for the unit type and disjoint sum.

We may also introduce more than one bottom object, for example, indexed by an arbitrary type, to get a more general formulation of the W-types with bottom objects. This allows us to represent types of the form $\mu X.\Phi(X)$, where $\Phi(X) \cong K + \Phi'(X)$ and there is an occurrence of X in each disjoint subterm of $\Phi'(X)$. This is less of a restriction than that for least fix-points in [Dyb88]. With this definition of W-types, our example of expressions can be represented in a similar manner to that above.

4.3 Data types vs. logical propositions

Note that the problem of the empty type is caused by its filling-up rule. In $ITT_\eta + W_\perp$, we have removed the empty type from the theory. We feel this is justified because in our type theory the notions of logical formula and data type (set) are distinguished. Therefore, unlike in Martin-Löf's type theory where the empty type plays the roles of the empty set and the logical constant *false*, in our theory it is viewed only as a set: the logical constant *false* is defined as $\forall P{:}Prop.P$. Without its filling-up rule, the empty type as a set can be approximated by the types $\Pi X{:}Type_i.\mathbf{T}_i(X)$. Thus, the roles that the empty type plays in Martin-Löf's type theory are shared by the types $\Pi X{:}Type_i.\mathbf{T}_i(X)$ and the proposition $\forall P{:}Prop.P$.

The relationship between the strong equality in Martin-Löf's extensional type theory and filling-up rules is further shown through Streicher's observation [Str91] concerning the filling-up rule for the weak equality $\mathbf{Id}$ in Martin-Löf's intensional type theory [NPS90]. The filling-up rule for $\mathbf{Id}_A$ is, for f of type $\Pi x,y{:}A\Pi z{:}\mathbf{Id}_A(x,y).C(x,y,z)$,

$$f = J_A(\lambda x.f(x,x,r_A(x)))$$

where r_A and J_A are the introduction and elimination operators for $\mathbf{Id}_A$, respectively. Streicher's observation is that, with the above rule, the weak equality becomes strongly extensional, in the sense that the following rule is derivable:

$$\frac{\Gamma \vdash q : \mathbf{Id}_A(a,b)}{\Gamma \vdash a = b : A}$$

This is because, letting $f_1(x,y,z) = x$, $f_2(x,y,z) = y$, and given $\Gamma \vdash q : \mathbf{Id}_A(a,b)$, we have, $a = f_1(a,b,q) = J_A(\lambda x.f_1(x,x,r(x)))(a,b,q) = J_A(\lambda x.f_2(x,x,r(x)))(a,b,q) = f_2(a,b,q) = b$.

We have a similar situation for the Leibniz equality over propositions definable in our theory (in fact, in the pure calculus of constructions). In this case, the filling-up rule for Leibniz equality implies that if two proofs are Leibniz equal in an arbitrary context, then they are convertible.

5 Further Discussion of the Filling-up Rules

In this paper, we consider the filling-up rules as a technical tool to allow the well-ordering types to represent inductive types faithfully. One of the main concerns for us is the decidability of systems with these rules: in particular for $ITT_\eta + W_\perp$, which does not have the empty type. The filling-up rules give rise to several meta-theoretic difficulties.

The study of properties such as Church-Rosser and strong normalization is made more difficult. First, it is not clear which direction of reduction for the filling-up rules is preferable. If we take the 'usual' orientation, the system is *not* Church-Rosser (even for well-typed terms), as the example concerning the unit type in [LS86] shows. In fact, the problem is not limited to the unit type. The reduction rule $\mathbf{E}_+(f \circ inl, f \circ inr) \quad f$ (with inl and inr defined as in figure 2) for disjoint union together with the usual β-rule is not Church-Rosser for well-typed terms: consider the term $\mathbf{E}_+(f \circ inl, f \circ inr)$ with $f \equiv \lambda x.c$, where c is a normal form.

There are two possible directions to explore. One is that we take the 'unusual' (but more natural) orientation to define the reduction relation, with certain constraints so that there is a notion of long normal form, for example, [Jay90]. Another possible approach is to consider other notions of reduction, for example based on work in term rewriting, *e.g.*, [CC91]. These are left as future research topics.

We may also consider a filling-up rule for the W-types, which has the form $f = \mathbf{E}_W(f \circ sup)$ (or $f = \mathbf{E}_W(f(\perp), f \circ sup)$ for W-types with a bottom object), with $sup =_{\mathrm{df}} \lambda a \lambda b \lambda p.\mathbf{sup}(a, b)$. It is unclear to us how such rules would change the theory and its meta-properties. With either of the filling-up rules for W-types, the inductive types defined as W-types would also have their corresponding filling-up rules derivable. For example, for the type of natural numbers, the following would hold:

$$f = rec_N(f(0), f \circ succ') : \Pi z{:}N.C(z)$$

where $succ' \equiv \lambda x \lambda y.succ(x)$. This seems to relate to the Böhm and Berarducci's notion of program equivalence in their consideration of representation of data types in the second-order λ-calculus [BB85].

Acknowledgements We would like to thank Thierry Coquand and Per Martin-Löf for their comments on this work and Thorsten Altenkirch, Rod Burstall, Peter Dybjer, Martin Hofmann, James McKinna, Randy Pollack and Thomas Streicher for interesting discussions. We would also like to thank two anonymous referees who read the paper carefully and suggested improvements.

References

[ACN90] L. Augustsson, Th. Coquand, and B. Nordström. A short description of another logical framework. In G. Huet and G. Plotkin, editors, *Preliminary Proc of Logical Frameworks*, 1990.

[Acz86] P. Aczel. The type theoretic interpretation of constructive set theory: inductive definitions. In *Logic, Methodology and Philosophy of Science VII*, 1986.

[Bac88] R. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In A. Avron et al, editor, *Workshop on General Logic*. LFCS

Report Series, ECS-LFCS-88-52, Dept. of Computer Science, University of Edinburgh, 1988.

[BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39, 1985.

[BM91] R. Burstall and J. McKinna. Deliverables: an approach to program development in the calculus of constructions. LFCS report ECS-LFCS-91-133, Dept of Computer Science, 1991.

[CC91] P.-L. Curien and R. Di Cosmo. A confluent reduction for the λ-calculus with surjective pairing and terminal object. In *Proc. ICALP'91*, 1991.

[CH88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[CPM90] Th. Coquand and Ch. Paulin-Mohring. Inductively defined types. *Lecture Notes in Computer Science*, 417, 1990.

[Dyb88] P. Dybjer. Inductively defined sets in Martin-Löf's set theory. In A. Avron et al, editor, *Workshop on General Logic*. LFCS Report Series, ECS-LFCS-88-52, Dept. of Computer Science, University of Edinburgh, 1988.

[Dyb91] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.

[GLT90] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1990.

[Jay90] B. Jay. personal communication, 1990.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*, volume VII of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[Luo90a] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. Also as Report CST-65-90/ECS-LFCS-90-118, Department of Computer Science, University of Edinburgh.

[Luo90b] Z. Luo. A problem of adequacy: conservativity of calculus of constructions over higher-order logic. Technical report, LFCS report series ECS-LFCS-90-121, Department of Computer Science, University of Edinburgh, 1990.

[Luo91a] Z. Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90(1):107–137, 1991.

[Luo91b] Z. Luo. Program specification and data refinement in type theory. *Proc. of the Fourth Inter. Joint Conf. on the Theory and Practice of Software Development (TAPSOFT)*, 1991. Also as LFCS report ECS-LFCS-91-131, Dept. of Computer Science, Edinburgh University.

[Luo91c] Z. Luo. A unifying theory of dependent types I. Technical report, LFCS report series ECS-LFCS-91-154, 1991.

[Men87] N. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of POPL*, 1987.

[Men88] N. Mendler. *Recursive Definition in Type Theory.* PhD thesis, Cornell University, 1988.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an introduction.* Oxford University Press, 1990.

[Ore90] C.-E. Ore. The extended calculus of constructions (**ECC**) with inductive types. To appear in Information and Computation, 1990.

[Pol90] R. Pollack. The Tarski fixpoint theorem. communication on TYPES e-mail network, 1990.

[Str91] T. Streicher. private communication, 1991.

A Inference rules of an intensional type theory

In this appendix, we present an intensional type theory, named ITT, which is the core of the type systems considered in this paper. There are five forms of judgements:

$$\Gamma \ \mathbf{valid} \quad \Gamma \vdash A \ \mathbf{type} \quad \Gamma \vdash A = B \quad \Gamma \vdash a : A \quad \Gamma \vdash a = b : A$$

Derivation and derivability of judgements are defined as usual, but in order to simplify the presentation, we adopt the following conventions.

1. To derive $\Gamma \vdash A$ **type**, Γ **valid** must have been derived.
2. To derive $\Gamma \vdash A = B$, $\Gamma \vdash A$ **type** and $\Gamma \vdash B$ **type** must have been derived.
3. To derive $\Gamma \vdash a : A$, $\Gamma \vdash A$ **type** must have been derived.
4. To derive $\Gamma \vdash a = b : A$, $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$ must have been derived.

We shall also often omit the type information in terms, *e.g.*, to write $\lambda x.b$ for $\lambda[\forall x{:}A.B]x.b$. In equality rules, the premises concerning the omitted type information are also omitted. For example, the introduction equality rule in A.3.1 abbreviates the following one:

$$\frac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash B = B' \quad \Gamma, x{:}A \vdash b = b' : B}{\Gamma \vdash \lambda[\Pi x{:}A.B]x.b = \lambda[\Pi x{:}A'.B']x.b' : \Pi x{:}A.B}$$

A.1 General rules

A.1.1 contexts and assumptions

$$\frac{}{\langle\rangle \ \mathbf{valid}} \qquad \frac{\Gamma \vdash A \ \mathbf{type} \quad x \notin FV(\Gamma)}{\Gamma, x{:}A \ \mathbf{valid}} \qquad \frac{}{\Gamma, x{:}A, \Gamma' \vdash x : A}$$

Note that, according to our conventions, to use the last rule to derive $\Gamma, x{:}A, \Gamma' \vdash x : A$, we must have derived $\Gamma, x{:}A, \Gamma'$ **valid** and $\Gamma, x{:}A, \Gamma' \vdash A$ **type**. We do not comment on such conventions any more below.

A.1.2 general equality rules

$$\frac{}{\Gamma \vdash A = A} \qquad \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A} \qquad \frac{\Gamma \vdash A = B \quad \Gamma \vdash B = C}{\Gamma \vdash A = C}$$

$$\frac{}{\Gamma \vdash a = a : A} \qquad \frac{\Gamma \vdash a = b : A}{\Gamma \vdash b = a : A} \qquad \frac{\Gamma \vdash a = b : A \quad \Gamma \vdash b = c : A}{\Gamma \vdash a = c : A}$$

A.1.3 equality typing

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A = B}{\Gamma \vdash a : B} \qquad \frac{\Gamma \vdash a = a' : A \quad \Gamma \vdash A = B}{\Gamma \vdash a = a' : B}$$

A.2 The impredicative universe

$$\frac{}{\Gamma \vdash Prop\ \mathbf{type}} \qquad \frac{\Gamma \vdash A\ \mathbf{type} \quad \Gamma, x{:}A \vdash P : Prop}{\Gamma \vdash \forall x{:}A.P : Prop} \qquad \frac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash P = P' : Prop}{\Gamma \vdash \forall x{:}A.P = \forall x{:}A'.P' : Prop}$$

formation	$\frac{\Gamma \vdash P : Prop}{\Gamma \vdash \mathbf{Prf}(P)\ \mathbf{type}}$	$\frac{\Gamma \vdash P = P' : Prop}{\Gamma \vdash \mathbf{Prf}(P) = \mathbf{Prf}(P')}$
introduction	$\frac{\Gamma, x{:}A \vdash p : \mathbf{Prf}(P)}{\Gamma \vdash \Lambda[\forall x{:}A.P]x.p : \mathbf{Prf}(\forall x{:}A.P)}$	$\frac{\Gamma, x{:}A \vdash p = p' : \mathbf{Prf}(P)}{\Gamma \vdash \Lambda x.p = \Lambda x.p' : \mathbf{Prf}(\forall x{:}A.P)}$
elimination	$\frac{\Gamma \vdash p : \mathbf{Prf}(\forall x{:}A.P) \quad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{App}[\forall x{:}A.P](p,a) : \mathbf{Prf}([a/x]P)}$	$\frac{\Gamma \vdash p = p' : \mathbf{Prf}(\forall x{:}A.P) \quad \Gamma \vdash a = a' : A}{\Gamma \vdash \mathbf{App}(p,a) = \mathbf{App}(p',a') : \mathbf{Prf}([a/x]P)}$
computation	$\frac{\Gamma, x{:}A \vdash p : \mathbf{Prf}(P) \quad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{App}(\Lambda x.p, a) = [a/x]p : \mathbf{Prf}([a/x]P)}$	

A.3 Type constructors

A.3.1 product types

formation	$\frac{\Gamma \vdash A\ \mathbf{type} \quad \Gamma, x{:}A \vdash B\ \mathbf{type}}{\Gamma \vdash \Pi x{:}A.B\ \mathbf{type}}$	$\frac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash B = B'}{\Gamma \vdash \Pi x{:}A.B = \Pi x{:}A'.B'}$
introduction	$\frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda[\Pi x{:}A.B]x.b : \Pi x{:}A.B}$	$\frac{\Gamma, x{:}A \vdash b = b' : B}{\Gamma \vdash \lambda x.b = \lambda x.b' : \Pi x{:}A.B}$
elimination	$\frac{\Gamma \vdash f : \Pi x{:}A.B \quad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{app}[\Pi x{:}A.B](f,a) : [a/x]B}$	$\frac{\Gamma \vdash f = f' : \Pi x{:}A.B \quad \Gamma \vdash a = a' : A}{\Gamma \vdash \mathbf{app}(f,a) = \mathbf{app}(f',a') : [a/x]B}$
computation	$\frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{app}(\lambda x.b, a) = [a/x]b : [a/x]B}$	

A.3.2 strong sum

formation	$\frac{\Gamma \vdash A\ \mathbf{type} \quad \Gamma, x{:}A \vdash B\ \mathbf{type}}{\Gamma \vdash \Sigma x{:}A.B\ \mathbf{type}}$	$\frac{\Gamma \vdash A = A' \quad \Gamma, x{:}A \vdash B = B'}{\Gamma \vdash \Sigma x{:}A.B = \Sigma x{:}A'.B'}$
introduction	$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B}{\Gamma \vdash \mathbf{p}[\Sigma x{:}A.B](a,b) : \Sigma x{:}A.B}$	$\frac{\Gamma \vdash a = a' : A \quad \Gamma \vdash b = b' : [a/x]B}{\Gamma \vdash \mathbf{p}(a,b) = \mathbf{p}(a',b') : \Sigma x{:}A.B}$
elimination	$\frac{\Gamma \vdash f : \Pi x{:}A \Pi y{:}B.[\mathbf{p}(x,y)/z]C}{\Gamma \vdash \mathbf{E}_\Sigma[\Pi z{:}(\Sigma x{:}A.B).C](f) : \Pi z{:}(\Sigma x{:}A.B).C}$	$\frac{\Gamma \vdash f = f' : \Pi x{:}A \Pi y{:}B.[\mathbf{p}(x,y)/z]C}{\Gamma \vdash \mathbf{E}_\Sigma(f) = \mathbf{E}_\Sigma(f') : \Pi z{:}(\Sigma x{:}A.B).C}$
computation	$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : [a/x]B \quad \Gamma, z{:}\Sigma x{:}A.B \vdash C\ \mathbf{type} \quad \Gamma \vdash f : \Pi x{:}A \Pi y{:}B.[\mathbf{p}(x,y)/z]C}{\Gamma \vdash \mathbf{E}_\Sigma(f)(\mathbf{p}(a,b)) = f(a,b) : [\mathbf{p}(a,b)/z]C}$	

A.3.3 the empty type

formation	elimination
$\overline{\Gamma \vdash \emptyset \ \mathbf{type}}$	$\overline{\Gamma \vdash \mathbf{E}_{\emptyset}[\Pi z{:}\emptyset.C] : \Pi z{:}\emptyset.C}$

A.3.4 the unit type

formation	$\overline{\Gamma \vdash \mathbf{1}\ \mathbf{type}}$	
introduction	$\overline{\Gamma \vdash * : \mathbf{1}}$	
elimination	$\dfrac{\Gamma \vdash c : [*/z]C}{\Gamma \vdash \mathbf{E_1}[\Pi z{:}\mathbf{1}.C](c) : \Pi z{:}\mathbf{1}.C}$	$\dfrac{\Gamma \vdash c = c' : [*/z]C}{\Gamma \vdash \mathbf{E_1}(c) = \mathbf{E_1}(c') : \Pi z{:}\mathbf{1}.C}$
computation	$\dfrac{\Gamma, z{:}\mathbf{1} \vdash C\ \mathbf{type} \quad \Gamma \vdash c : [*/z]C}{\mathbf{E_1}(c)(*) = c : [*/z]C}$	

A.3.5 disjoint sum

formation	$\dfrac{\Gamma \vdash A\ \mathbf{type} \quad \Gamma \vdash B\ \mathbf{type}}{\Gamma \vdash A + B\ \mathbf{type}}$	$\dfrac{\Gamma \vdash A = A' \quad \Gamma \vdash B = B'}{\Gamma \vdash A + B = A' + B'}$
introduction	$\dfrac{\Gamma \vdash a : A}{\Gamma \vdash \mathbf{i}[A+B](a) : A + B}$	$\dfrac{\Gamma \vdash a = a' : A}{\Gamma \vdash \mathbf{i}(a) = \mathbf{i}(a') : A + B}$
	$\dfrac{\Gamma \vdash b : B}{\Gamma \vdash \mathbf{j}[A+B](b) : A + B}$	$\dfrac{\Gamma \vdash b = b' : B}{\Gamma \vdash \mathbf{j}(b) = \mathbf{j}(b') : A + B}$
elimination	$\dfrac{\Gamma \vdash f : \Pi x{:}A.[\mathbf{i}(x)/z]C \quad \Gamma \vdash g : \Pi y{:}B.[\mathbf{j}(y)/z]C}{\Gamma \vdash \mathbf{E}_{+}[\Pi z{:}A+B.C](f,g) : \Pi z{:}A+B.C}$	$\dfrac{\Gamma \vdash f = f' : \Pi x{:}A.[\mathbf{i}(x)/z]C \quad \Gamma \vdash g = g' : \Pi y{:}B.[\mathbf{j}(y)/z]C}{\Gamma \vdash \mathbf{E}_{+}(f,g) = \mathbf{E}_{+}(f',g') : \Pi z{:}A+B.C}$
computation	$\dfrac{\Gamma \vdash a : A \quad \Gamma, z{:}A+B \vdash C\ \mathbf{type} \quad \Gamma \vdash f : \Pi x{:}A.[\mathbf{i}(x)/z]C \quad \Gamma \vdash g : \Pi y{:}B.[\mathbf{j}(y)/z]C}{\Gamma \vdash \mathbf{E}_{+}(f,g)(\mathbf{i}(a)) = f(a) : [\mathbf{i}(a)/z]C}$	
	$\dfrac{\Gamma \vdash b : B \quad \Gamma, z{:}A+B \vdash C\ \mathbf{type} \quad \Gamma \vdash f : \Pi x{:}A.[\mathbf{i}(x)/z]C \quad \Gamma \vdash g : \Pi y{:}B.[\mathbf{j}(y)/z]C}{\Gamma \vdash \mathbf{E}_{+}(f,g)(\mathbf{j}(b)) = g(b) : [\mathbf{j}(b)/z]C}$	

A.4 Predicative universes

We introduce predicative universes $Type_i$ ($i \in \omega$), whose formation rules are:

$$\frac{}{\Gamma \vdash Type_i \ \mathbf{type}} \qquad \frac{\Gamma \vdash a : Type_i}{\Gamma \vdash \mathbf{T}_i(a) \ \mathbf{type}} \qquad \frac{\Gamma \vdash a = a' : Type_i}{\Gamma \vdash \mathbf{T}_i(a) = \mathbf{T}_i(a')}$$

The following are the introduction and reflection rules, where Φ/ϕ_i stand for Π/π_i and Σ/σ_i (and W/w_i, when the system is extended with W-types), respectively.

$\emptyset$	$\frac{}{\Gamma \vdash \emptyset : Type_0}$	$\frac{}{\Gamma \vdash \mathbf{T}_0(\emptyset) = \emptyset}$
$\mathbf{1}$	$\frac{}{\Gamma \vdash 1 : Type_0}$	$\frac{}{\Gamma \vdash \mathbf{T}_0(1) = \mathbf{1}}$
$+$	$\frac{\Gamma \vdash a : Type_i \quad \Gamma \vdash b : Type_i}{\Gamma \vdash a \oplus_i b : Type_i}$	$\frac{\Gamma \vdash a : Type_i \quad \Gamma \vdash b : Type_i}{\Gamma \vdash \mathbf{T}_i(a \oplus_i b) = \mathbf{T}_i(a) + \mathbf{T}_i(b)}$
Φ	$\frac{\Gamma \vdash a : Type_i \quad \Gamma, x{:}\mathbf{T}_i(a) \vdash b : Type_i}{\Gamma \vdash \phi_i x{:}a.b : Type_i}$	$\frac{\Gamma \vdash a : Type_i \quad \Gamma, x{:}\mathbf{T}_i(a) \vdash b : Type_i}{\Gamma \vdash \mathbf{T}_i(\phi_i x{:}a.b) = \Phi x{:}\mathbf{T}_i(a).\mathbf{T}_i(b)}$
$Type_i$	$\frac{}{\Gamma \vdash type_i : Type_{i+1}}$	$\frac{}{\Gamma \vdash \mathbf{T}_{i+1}(type_i) = Type_i}$
$\mathbf{T}_i$	$\frac{\Gamma \vdash a : Type_i}{\Gamma \vdash \mathbf{t}_{i+1}(a) : Type_{i+1}}$	$\frac{\Gamma \vdash a : Type_i}{\Gamma \vdash \mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) = \mathbf{T}_i(a)}$
$Prop$	$\frac{}{\Gamma \vdash prop : Type_0}$	$\frac{}{\Gamma \vdash \mathbf{T}_0(prop) = Prop}$
$\mathbf{Prf}$	$\frac{\Gamma \vdash P : Prop}{\Gamma \vdash \mathbf{t}_0(P) : Type_0}$	$\frac{\Gamma \vdash P : Prop}{\Gamma \vdash \mathbf{T}_0(\mathbf{t}_0(P)) = \mathbf{Prf}(P)}$

$$\frac{\Gamma \vdash a = a' : Type_i \quad \Gamma \vdash b = b' : Type_i}{\Gamma \vdash a \oplus_i b = a' \oplus_i b' : Type_i} \qquad \frac{\Gamma \vdash a = a' : Type_i \quad \Gamma, x{:}\mathbf{T}_i(a) \vdash b = b' : Type_i}{\Gamma \vdash \phi_i x{:}a.b = \phi_i x{:}a'.b' : Type_i}$$

$$\frac{\Gamma \vdash a = a' : Type_i}{\Gamma \vdash \mathbf{t}_{i+1}(a) = \mathbf{t}_{i+1}(a') : Type_{i+1}} \qquad \frac{\Gamma \vdash P = P' : Prop}{\Gamma \vdash \mathbf{t}_0(P) = \mathbf{t}_0(P') : Type_0}$$