Completeness of Π

Siva Somayyajula

July 2017

Abstract

Π is a reversible programming language by Sabry et al. inspired by isomorphisms over finite types in type theory. We give a model for Π in a univalent universe of finite types: a chosen sub-universe in homotopy type theory that is a univalent fibration. Using properties of univalent fibrations, we give formal proofs in Agda that level 0 terms (types) and level one terms (isomorphisms) in Π are complete with respect to this model. We also discuss a formalization of completeness for level two terms (coherence between isomorphisms).

Contents

1 Introduction

1.1 Reversibility

Reversibility is prevalent in computing applications, giving rise to ad hoc implementations in both hardware and software alike. For example, the Fredkin and Toffoli gates are universal for reversible circuits, and version control systems like `darcs` are based on *patch theory*, an algebra for file changes. At the software level, this has motivated the development of general-purpose reversible programming languages—Janus, developed in 1982, is such a language with a formally verified interpreter.

While Janus is designed for imperative programming, there has not yet been such an effort for functional programming, whose emphasis on avoiding mutability, amongst other things, is amenable to reversibility. To elaborate, a natural type-theoretic notion of reversibility is given by type isomorphisms i.e. lossless transformations over structured data. Thus, a calculus for such isomorphisms would give rise to a reversible functional programming language. The Π programming language introduced by Sabry et al.is precisely that. However, to understand Π and its model, we give a brief introduction to type theory and its homotopy-theoretic interpretation.

1.2 Type Theory

A type theory is a formal system for *types*, *terms*, and their computational interactions. A helpful analogy to understand type theory at first is to conceptualize types as sets and terms as their elements. Like set theory, type theories have rules governing *type formation* as there are axioms about set construction e.g. the axiom of pairing, but this is where the analogy breaks down. Whereas set theory has an internal membership predicate (i.e. set membership is a proposition), type theories have a notion of membership external to the system via *typing judgments*: given a type A and a term a, one may derive the judgment $a : A$ (pronounced "a inhabits A") via *term introduction* and *elimination* rules. However, such judgments do not exist as statements within the language itself.

Perhaps the distinguishing feature of type theories are their explicit treatment of computation: computation rules dictate how terms evaluate to normal forms. To programming language theorists, type theories formally describe programming languages and computation rules are precisely the structured operational semantics. On the other hand, set theories have no such equivalent concept.

This emphasis on computation has several applications to computer science. First, the type systems of such programming languages as Haskell are based on certain type theories (specifically, System F). Aside from their utility in programming language design, sufficiently sophisticated type theories are suitable as alternative foundations of mathematics to set theory. In fact, Martin-Löf type theory (MLTT) is the basis of many programs aiming to formalize constructive mathematics. To understand how this is possible, recall that set theories consist of rules governing the behavior of sets as well as an underlying logic to express propositions and their truth. Thus, it remains to show that type theories, under the availiblity of certain type formers, may encode propositions as types and act as deductive systems in their own right.

Thus, we will first give a brief introduction to MLTT and then *homotopy type theory* (HoTT), which extends MLTT with new rules motivated by homotopy theory.

1.3 Martin-Löf Type Theory

MLTT is designed to be a foundations of constructive mathematics. First, we will describe the primitive types of this system, the various type formers, and

Definition 1.1 (Universes)**.** AU_1

Definition 1.2 (Finite types)**.** $\mathbb{0}$ is the type with no inhabitants, and $\mathbb{1}$ is the type with exactly one canonical inhabitant: $*$.

$ind_+ : \mathbb{0} \to$

Definition 1.3 (Dependent function)**.** The dependent function type is inhabited by functions whose codomains vary with their inputs: given $a : A$. `(a:A)→Pa`

A special case of the dependent function type is the function type $A \to B$ defined as $\prod_a$. That is, the codomain does not vary with inputs from the domain.

Definition 1.4 (Dependent pair)**.** Given a type A and a type family $P : A \to U$, the dependent pair type $\sum_{a:A} P(a)$ has inhabitants (a, b) where $a : A$ and $b : P(a)$. Furthermore, there are projection functions π_1 and π_2 such that $\pi_1(a, b) = a$ and $\pi_2(a, b) = b$.

Definition 1.5 (Cartesian product)**.** Given types A and B:

$$A \times B = \sum_{a:A} \lambda\_.\ B$$

That is, terms of this type are simply pairs (a, b) where $a : A$ and $b : B$ with the usual projection functions.

Definition 1.6 (Coproduct)**.** Given types A and B, their coproduct $A + B$ is characterized by the following rules:

- If $a : A$, then $\mathtt{inl}(a) : A + B$

- If $b : B$, then $\mathtt{inr}(b) : A + B$

The cartesian product and coproduct types should be familiar due to their obvious correspondence to set theory. However, the *propositions-as-types* principle gives a clear logical perspective on the dependent function and pair types.

Definition 1.7 (Propositions-as-types)**.** Propositions can be encoded as types. If A is such a type, and $a : A$, then a is a proof. The exact correspondence is given below.

type	proposition
$\mathbb{0}$	$\prod_{a:A} P(a)$
$\forall_{a:A} P(a)$	
$\sum_{a:A} P(a)$	$\exists_{a:A} P(a)$

This principle gives us a deductive system for constructive logic, in the sense that proofs compute the truth of a proposition. For example, a proof of a universally quantified formula is a computable function that sends an input to evidence that a particular predicate holds on that input. Similarly, a proof of an existentially quantified formula is exactly one term paired with evidence that a predicate holds on that term, and so on.

1.4 Homotopy Type Theory

Homotopy type theory

Definition 1.8 (PathFrom)**.**

$$PathFrom(x) \triangleq \sum_{y:A} x = y$$

Definition 1.9 (Paulin-Mohring's J)**.** Given a type family $P : PathFrom(x) \to U$, $J : P(x, \mathtt{refl}_x) \to \prod_{c:PathFrom(x)} P(c)$ with the following computation rule:

$$J(r, (x, \mathtt{refl}_x)) \mapsto r$$

Without using other induction principles for the identity type (such as Axiom K), it is impossible to prove *or* disprove within HoTT that inhabitants of the identity type are propositionally equal to reflexivity, a principle which is called *uniqueness of identity proofs* (UIP). In fact, one can only prove that inhabitants of

$PathFrom(x)$ are propositionally equal to $(x, \mathtt{refl}_x)$. As a result, this allows us to add so-called nontrivial inhabitants to the identity type via separate inference rules without rendering the system inconsistent.

This peculiar result motivates, for example, the axiom of function extensionality. Given functions $f, g : A \to B$, if one has evidence $\alpha : \prod_{x:A} f(x) = g(x)$, then one has $\mathtt{funext}(\alpha) : f = g$. However, the crux of HoTT lies in Voevodsky's univalence axiom, which is an extensionality axiom for *types*. Before we introduce it, we must first define what it means for two types to be equivalent.

Definition 1.10 (Quasi-inverse)**.** A *quasi-inverse* of a function $f : A \to B$ is the following data:

- $g : B \to A$
- $\alpha : \prod_{x:A} g(f(x)) = x$
- $\beta : \prod_{x:B} f(g(x)) = x$

For the purposes of this paper, we will refer to functions that have quasi-inverses as equivalences, although there are other equivalent notions in type theory.

Definition 1.11 (Type equivalence)**.** Given types X and Y, $X \simeq Y$ if there exists a function $f : X \to Y$ that is an equivalence.

An immediate result is that an equality between types can be converted to an equivalence.

Lemma 1.1 (`idtoeqv`)**.** Given types A and B, there is a term $\mathtt{idtoeqv} : A = B \to A \simeq B$ defined as:

$$\lambda p.\ J((\lambda(B, \_).\ A \simeq B), ide(A), (B, p))$$

Axiom 1.1 (Univalence)**.** `idtoeqv` is an equivalence.

By declaring that `idtoeqv` has a quasi-inverse, this axiom gives us the following data:

- $\mathtt{ua} : A \simeq B \to A = B$, a function that converts equivalences to paths
- $\mathtt{ua} - \beta : \pi_{f:A\simeq B} \mathtt{idtoeqv}(\mathtt{ua}(f)) = f$

- $\texttt{ua} - \eta : \pi_{f:A\simeq B}\texttt{ua}(\texttt{idtoeqv}(p)) = p$

The last two data are called *propositional computation rules*, as they dictate how ua reduces propositionally, outside of the computation rules built into type theory. However, this raises the question: how do terms compute to a normal form in the presence of univalence? This is actually still an open question—for now, homotopy type theory lacks *canonicity*, the guarantee that every well-typed term has a normal (canonical) form.

Before moving onto Π and its model, we must establish one last concept and rethink our previous conception of propositions-as-types.

Definition 1.12 (Mere proposition)**.** A type is a *mere proposition* if all of its inhabitants are propositionally equal. That is, the following type is inhabited:

$$is - prop(A) = \prod_{x,y:A} x == y$$

Definition 1.13 (Propositional truncation)**.** For a type A, its propositional truncation A is described by the following

- If $a : A$, then $a : A$
- $\texttt{identify} : \Pi_{x,y:A} x == y$

By identify, the propositional truncation of any type is a proposition, hence the name. Truncation “forgets” all the information of terms in a type other than inhabitance,

1.5 Univalent Fibrations