UGP Report : Groupoid Interpretation of Martin-Lof Type Theory

Divyanshu Shende
Roll No.: 13264
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur.
email : divush@cse.iitk.ac.in

Mentor : Dr. Anil Seth,
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur.
email : seth@cse.iitk.ac.in

November 24, 2016

Abstract

Uniqueness of Identity Proofs (UIP) asks whether elements in identity sets, defined in intensional Martin-Löf Type Theory, are equal for all sets? In 1996, Martin Hofmann and Thomas Streicher[HS96] answered this in the negative by giving a Groupoid Interpretation of Martin-Löf Type Theory. We look at this result.

1 Martin-Löf Type Theory

To say that A is a type we write A *type*. While defining a type, we must say what an object of that type is and when two objects are equal. Let a be an object satisfying the conditions for type A, that a is an object of the type A is denoted by $a : A$. This is also called a *type judgement*. That A and B are two *definitionally equal* types is denoted by $A = B$. a and b are *definitionally equal* (i.e., have same normal form) objects of type A is denoted by $a = b : A$. In this type theory, we can also make judgements under a set of assumptions. We say that A *type* $[x : C]$ to denote that A is a type under the assumption that x is an object of type C. We can also say that A is a *family of types* over C. In case we have more than one assumptions, each holds in the context of the previous ones. For instance, if we have the assumption that A *type* $[x_1 : A_1, x_2 : A_2, ..., x_n : A_n]$, then $x_i : A_i$ $[x_1 : A_1, x_2 : A_2, ..., x_{i-1} : A_{i-1}]$. Suppose A *type* $[x : C]$, then $A[x \leftarrow c]$ is a type (provided $c : C$)which we obtain by substuting c for x. Also, we have that $a[x \leftarrow c] : A[x \leftarrow c]$.
We can also define a special type Set whose objects are *sets*. For detailed description and examples of this type, we refer the reader to [BNS90].

1.1 Dependent Function Space

Let A be a type and B be a family of types over A. The type of all (dependent) functions from A to B is denoted by $(x : A)B$. An object of this type is a function from A to B. Suppose $f : (x : A)B$, then $f(a) : B[x \leftarrow a]$ for all $a : A$. Identical objects in A give identical objects on function application. Two functions f, g of the type $(x : A)B$ are the identical iff for all $a : A$, $f(a) = g(a) : B[x \leftarrow a]$.

1.2 Propositional Equality

Two objects are *definitionally equal* iff they have the same normal form. For instance $5 = 5$ is a definitional equality because, in the definition of natural numbers, both have the same normal form, i.e., $S(S(S(S\,0)))$.
In contrast, take a look at commutativity property of addition addition. It is formalized as, $(a, b : Nat)\ a + b = b + a$. When we substitute values for a and b, then we can show (by computing) that the two sides have the same normal form. However, commutativity in general is a *theorem* and hence requires a *proof*. This is the idea behind *propositional equality* and is captured using *Identity Sets*.
For any set A and two objects $a, b : A$, we define a set $Id(A, a, b)$. The elements of this set are the proofs that allow us to deduce $a = b$. We can define a constructor id for the set Id as $id : (A : Set)(a : A)Id(A, a, a)$. The constructor id takes a set A, an object $a : A$ and gives a proof that $a = a$. We formalize this idea in the next section.

2 Identity Sets

Formalizing propositional equality led to the definition of Identity Sets. We discuss them in detail here. We define three constants as below.

1. Defining the constructor for Propositional Equality - $Id : (A : Set)\,(a, b : A)\,Set$
2. Defining the constructor for reflexivity - $refl : (A : Set)\,(a : A)Id(A, a, a)$
3. Defining the elimination operator -
   $J : (A : Set)\,(P : (a, b : A)(s : Id(A, a, b)Set)\ (d : (x : A)P(x, x, refl(A, x, x))$
   $(l, m : A)\,(s : Id(l, m, A))\ P(l, m, s)$

The Id constructor can be read as follows. Id is a (dependent) function that takes as argument a set A and two elements $a, b : A$. The output is a set that we call $Id(A, a, b)$. The definition of $refl$ can be read in the same way. Note $refl(A, a, a) : Id(A, a, a)$ so $refl$ actually gives a proof that $a = a$ in A.

The elimination operator is more involved. The argument P in it can be thought of as a propositional function on A. So, P takes two arguments $a, b : A$ and a proof $s : Id(A, a, b)$ and returns a set. Argument d in the above definition gives an element in the set $P(A, a, a)$. Essentially, J does the following : it takes a set A, a dependent set P on A, a method d to form elements in P for equal elements in A, two elements $l, m : A$ and a proof $s : Id(A, l, m)$ and gives an element of the type $P(l, m, s)$. We also have the following (definitional) equality imposed on J.

$$J(A,\ P,\ d,\ a,\ a,\ refl(A, a, a)) = d(a) : P(a,\ a,\ refl(A, a, a))$$

The elimination operator looks cumbersome but is powerful. It's main power lies in suitable defining P. The following constructs can be derived using J, their derivations are given [BNS90] so we shall not repeat them here :

1. $subst : (A : Set)\,(P : (a : A)Set)\,(s : Id(a, b, A))\ P(a) \to P(b)$
2. $sym : (A : Set)\,(a, b : A)\,Id(A, a, b) \to Id(A, b, a)$
3. $trans : (A : Set)\,(a, b, c : A)\,Id(A, b, c) \to Id(A, a, b) \to Id(A, a, c)$

subst gives an element of $P(b)$ given $P(a)$ and a proof that $a = b$. *sym* derives a proof of $b = a$ given a proof of $a = b$. Transitivity composes proofs of $b = c$ and $a = b$ to give a proof of $a = c$. The order of aplying proofs in transitivity is the application order.

2.1 Uniqueness of Identity Proofs (UIP)

In the language described above, UIP simply asks for all types, whether or not any two terms of $Id(A, x, y)$ are equal. In other words, does there exist a *unique* proof of the propositional equality $x = y$? Formally, this is stated as,

$$UIP \coloneqq (A : Set)\,(a, b : A)\,(s_1, s_2 : Id(A, a, b))\ Id((Id, A, b),\ s_1, s_2)$$

In their paper[HS96], Hofmann and Streicher gave an interpretation of type theory where UIP is not derivable. Therefore, the UIP, in general, is not derivable in Martin-Löf Type Theory.

3 Semantics

In giving the groupoid interpretation, our aim is to give a semantics for dependent type theory. However, due to dependent types, proving that our interpretation follows all the rules of the type theory could be cumbersome. For this reason, previous work in this area led to formalization of an abstract notion of semantics : category-theoretic semantics, which has proven to be useful. The correctness of such semantics has been established. We can use this abstract semantics and simply show that the mathematical structure of our interpretation forms a valid instance of the abstraction. This way we know that we have a valid interpretation. In our case, the abstract semantics framework used is *Category with Families (CwF)* first defined by [P96]. CwF captures the notion of type dependency. For details on abstract semantics and CwF, we refer the reader to [Hof97]. We shall define CwF in a later section.

4 The Groupoid Interpretation

The groupoid interpretation treats non-dependent types as groupoids. Closed terms of these types then become the *objects* of the groupoid. The *arrows*(or morphisms) are elements of the identity sets. Isomorphism accounts for symmetry and transitivity is accounted for by composition rule. The existence of identity morphism accounts for reflexivity. We now proceed to define other aspects of type theory. The interpretation requires some basic knowledge of category theory, namely categories, groupoids, functors and natural transformations. The reader can look these up from [Pie91].

4.1 Families of Groupoids

Let Γ be a groupoid. We define a family of groupoids over Γ as a functor $A : \Gamma \to GPD$ with the following properties[HS96] :

1. For every $\gamma \in \Gamma$, $A(\gamma)$ is a groupoid.
2. For every morphism $f : g \to g'$ in Γ, $A(f) : A(g) \to A(g')$ is a functor.

Using functorality of A and symmetry in Γ, we can show that all functors $A(f)$ are actually isomorphisms.

4.1.1 Notations

Given $f : g \to g'$ and a family of groupoids A, over Γ, we write $f._-$ for the functor $A(f)$. So, for $x \in A(g)$, $f.x = A(f)(x)$. $Ty(\Gamma)$ is used to denote the collection of families over Γ. Thus, $A \in Ty(\Gamma)$. Also, if $h : \Delta \to \Gamma$, then $A \circ h$ (denoted by $A\{h\}$) is clearly a functor from $\Delta \to GPD$ and is in $Ty(\Delta)$.

4.1.2 Dependent Objects

A dependent object M of A, (where $A \in Ty(\Gamma)$), consists of the following :

1. For every $\gamma \in \Gamma$, an object $M(\gamma) \in A(\gamma)$.

2. For every morphism $f : g \to g'$, a morphism $M(f) : f.(M(g)) \to M(g')$. Note that $M(f)$ is a morphism in $A(g')$.

The reason $M(f)$ is defined in such a manner is because it is not necessary that $f.(M(g)) = M(g')$. Note that $f.(M(g)) = A(f)(M(g))$, which is the image of $M(g)$ in $A(g')$ under $A(f)$. It is interesting to note that the identity and composition laws for M hold apart from a small adjustment required in the composition part. We use the notation $Tm(A)$ to represent the collection of dependent objects of A.

4.2 Category with Families

As mentioned earlier, our aim is to fit our theory into the framework of an abstract semantics, namely category with types. Before moving further, let us look at what CwF consists of[HS96] :

- A category of contexts and substitutions C with [] as the terminal object representing empty context.
- A functor $Ty : C^{op} \to Set$, which assigns, to every type Γ, a collection of types that depend on it.
- For all $\Gamma \in C$, a collection of terms $Tm(\Gamma, A)$, where $A \in Ty(\Gamma)$ along with a function $Tm(h, A) : Tm(\Gamma, A) \to Tm(\Delta, A\{h\})$, where $h : \Delta \to \Gamma$.
- A *context extension*, $\Gamma.A$, which has the property that homset $C(\Delta, \Gamma.A)$ is isomorphic to $\{ (h, M) \mid h : \Delta \to \Gamma \text{ and } M \in Tm(\Delta, A\{h\}) \}$

In our interpretation, we have already described the category of contexts as GPD and defined Ty, Tm. Now, we need to define *context extension*, which we do below.

4.3 Context Extension

Let $A \in Ty(\Gamma)$. We define $\Gamma.A$ as a category as follows. The objects of $\Gamma.A$ are pairs (γ, a), where $\gamma \in \Gamma$ and $a \in A$. Given two objects (γ, a) and (γ', a'), we define a morphism between them to be a pair (p, q) such that $p : \gamma \to \gamma'$ and $q : p.a \to a'$. Let $(p, q) : (\gamma, a) \to (\gamma', a')$ and $(p', q') : (\gamma', a') \to (\gamma'', a'')$. Then their composition of $(p', q') \circ (p, q)$, where is given by $(p' \circ p, q' \circ (p'.q))$. Inverse of (p, q) is $(p^{-1}, p^{-1}.q^{-1})$.
We note that even though A is a functor, the context extension $\Gamma.A$ is a groupoid and this is why, we can define families over this new context. We also define a projection function $\mathtt{p}$ as follows. $\mathtt{p}(\gamma, a) = \gamma$ and $\mathtt{p}(p, q) = p$. Thus $\mathtt{p}$ is a functor from $\Gamma.A$ to Γ.
We can also show that the two sets are isomorphic as required by the CwF abstraction.[HS96] With that, we have shown that our interpretation thus far forms an instance of CwF. Now we interpret the remaining syntax to complete the interpretation.

4.4 Dependent Function Space

We define $\Pi(A, B)$ to be the type of (dependent) functions from A to B. In this section, we also aim to interpret abstraction and application. Suppose $A \in Ty(\Gamma)$ and $B \in Ty(\Gamma.A)$, then we have $\Pi(A, B) \in Ty(\Gamma)$. Before coming to abstraction and application, we define a few things. For dependent object $M \in Tm(A)$, we associate a functor $\overline{M} : \Gamma \to \Gamma.A$. The object part is given by $\overline{M}(\gamma) = (\gamma, M(\gamma)$ and the morphism part is given by $\overline{M}(p) = (p, M(p))$. Given f, such that $\mathtt{p} \circ f = id_\Gamma$, we must have $f(\gamma) = (\gamma, M(\gamma))$ and $f(p) = (p, M(p))$ for fixed M. This allows us to view $Tm(A)$ as a groupoid. A morphism h between $\overline{M}$ and $\overline{N}$ is given by a collection of morphisms $h_\gamma : M(\gamma) \to N(\gamma)$, such that for all $p : \gamma \to \gamma'$, we have $h(\gamma') \circ \overline{M}(p) = h(\gamma) \circ \overline{N}(p)$. We use this interpretation as groupoid below.

We now define the family $\Pi(A, B)(\gamma)$ as a collection of terms in B_γ viewed as a groupoid, where B_γ is a family of groupoids over $A(\gamma)$ given by,[HS96]

- $B_\gamma(a) = B(\gamma, a)$
- $B_\gamma(p)(\alpha) = (id_\gamma, p).\alpha$

We define abstraction of term $M \in Tm(B)$ as a term $\lambda_{A,B}(M) \in Tm(\Pi(A, B))$ and define it as follows [HS96] :

- $\lambda_{A,B}(M)(\gamma)(a) = M(\gamma, a)$
- $\lambda_{A,B}(M)(id_\gamma)(l) = M(id_\gamma, l)$

Also, we define $\lambda^{-1}_{A,B} \in Tm(B)$ to interpret application as follows[HS96] :

- $\lambda^{-1}_{A,B} M(\gamma, a) = M(\gamma)(a)$
- $\lambda^{-1}_{A,B} M(p, q) = M(\gamma')(q) \circ (id_{gamma'}, q).M(p)_{p.a}$

It is noteworthy interpretation of $Tm(B_\gamma)$ as a groupoid allowed us to define $\lambda^{-1}_{A,B}$

4.5 Identity Sets

We interpret identity sets as a family of groupoids over a groupoid whose elements are (A, a_1, a_2). The morphisms in this category are given by (p, q_1, q_2) so that $p : A \to A'$ is an isomorphism of groupoids and $q_i : p(a_i) \to a'_i$. Our identity sets are defined as follows[HS96]:

- $Id(A, a_1, a_2) = \Delta((A(a_1, a_2))$, where Δ represents the discrete category with only identities as morphisms.
- $Id(p, q_1, q_2)(s) = q_2 \circ s \circ q_1^{-1}$, where $s \in A(a_1, a_2)$.

4.5.1 Reflexivity

Now, we need to define $refl$ which is a dependent term of type Id. We now define the constructor $refl$ as a term in $\Delta(Id(A, a, a))$ as follows[HS96]:

- $refl(A, a) = id_a \in A(a, a)$
- For morphism, we observe that $q \circ p(refl(A, a)) \circ q^{-1} = refl(A', a')$, where $(p, q, q) : (A, a, a) \to (A', a')$.

4.5.2 Elimination

All that remains now, is to define elimination operator J. After currying, we can show that we need to define a term over $J \in Tm(C(a_1, a_2, s)[\Gamma, a_1 : A, a_2 : A, s : Id(A, a_1, a_2)])$, where $\Gamma = [A : Set, C : (a_1, a_2 : A, s : Id(A, a_1, a_2))Set, d : (a : A)C(a, a, refl(A, a, a))]$. Note that J is a term of C in the context. The context here is Γ given by the appropriate context extension. Let x be an term of the context given by (γ, a_1, a_2, s). We define $f(x)$ as follows:
$f(x) \coloneqq (id_\gamma, id_{a_1}, s, \star)$
Here $\star$, is a morphism of the discrete category. $f(x)$ essentially is a morphism in the category of our context Γ. Also, let $x' = (\gamma', a_1', a_2', s')$ and let $h : x \to x'$, so that $h = (p_1, q_1, q_2, \star)$. We define the object J as follows[HS96]:

- $J(x) \coloneqq f(x).d(\gamma, a_1)$. (Note that $d(\gamma, a_1)$ is an object of $C(\gamma, a_1, a_1, refl(A, a_1))$).
- The morphism is defined by $J(h) = f(x').d(p, q_1)$.

In the above, we have $p : \gamma \to \gamma'$. It is a simple exercise to verify that the definitions are indeed valid. See [HS96] for the proofs.

5 Verdict on UIP

We end the report with the following theorem on UIP.

Theorem 1. *[HS96] There does not exist any term of the type UIP.*

Proof. (by contradiction)
Suppose a term u existed. Consider the one object groupoid $\mathbb{Z}_2$. From category theory, we know that it has only one element x and two morphisms *distinct* id_x and p such that $p \circ p = id_x$. By definition of UIP type, $u(A, x, x, p, id_x)$ (remember u is a UIP term), is a proof that $p = id_x$. However, we know that they must be distinct. Therefore, such a u cannot exist. □

The groupoid interpretation is sound, and therefore we can extend the above claim to say that we *cannot* define a closed term for UIP.

Bibliography

[BNS90] K. Petersson B. Nordström and J. M. Smith. "Martin-Löfś type theory". In: (1990).

[Pie91] Benjamin Pierce. "Basic Category Theory for Computer Scientists". In: (1991).

[HS96] Martin Hofmann and Thomas Streicher. "The groupoid interpretation of type theory". In: *Twenty-five years of constructive type theory (Venice, 1995)* (1996).

[P96] Dybjer P. "Internal Type Theory". In: (1996).

[Hof97] Martin Hofmann. "Syntax and Symantics of Dependent Types". In: (1997).