Master thesis proposal: Erasing propositions in homotopy type theory

Gabe Dijkstra

April 5, 2013

1 Introduction

1.1 TODO Introduce homotopy type theory as some recent development

Relating results from homotopy theory to type theory and vice versa.

1.2 TODO The type theory we are interested in is Martin-Löf's

Intensional type theory Martin-Löf (1985).

1.3 TODO Introduce identity types and elimination principle J in Agda syntax.

1.4 TODO Footnote about how and why this is called propositional equality.

```
data Id (A : Set) (x : A) : A → Set where
  refl : Id A x x
```

1.5 TODO Introduce $\equiv$ notation as well

1.6 TODO "Implementing" equality like this has nice properties

such as: type checking is decidable, we have canonicity

1.7 TODO Using J we can easily prove: *refl*, *trans*, *symm*

1.8 TODO It has its cons as well

We cannot prove function extensionality even though it holds in all models.

When we look at the definition of *Id*, it seems that there is only one way to construct a proof of propositional equality, namely by applying the *refl* constructor. In Agda, using dependent pattern matching, we can prove this fact, usually called *uniqueness of identity proofs*:

$$\mathit{UIP} : (A : \mathit{Set})\ (x\ y : A)\ (p\ q : \mathit{Id}\ A\ x\ y) \rightarrow \mathit{Id}\ (\mathit{Id}\ A\ x\ y)\ p\ q$$
$$\mathit{UIP}\ A\ x \circ x\ \mathit{refl}\ \mathit{refl} = \mathit{refl}$$

1.9 TODO Fix dot pattern lhs2tex

However, this fact cannot be proven using only the elimination principle J.

1.10 TODO Hofmann and Streicher (1996) showed that this impossible

They noticed that if we can interpret identity types as groupoids: we can interpret *trans* as composition, *refl* as the identity operation and *sym* as the inverse operation.

1.11 TODO Introduce new notation

- Associativity: Id (Id A x y) ((a * b) * c) (a * (b * c))
- Reft and right inverses: Id (Id A x y) (a^{-1} * a) e and Id (Id A x y) (a * a^{-1}) e
- Left identity: Id (Id A x y) (e * a) a
- Right identity: Id (Id A x y) (a * e) a

1.12 TODO Do we really want to elaborate on all of that here, right now?

I can always move it elsewhere I suppose.

Notice that in the type theoretic formulation of the laws, the equalities are equalities between identity types, not definitional equality, but equalities *one level higher*. This nesting can be made arbitrarily deep, leading to a construction homotopy theorists call weak ∞-groupoids.

The homotopic intuition behind weak ∞-groupoids leads to the following correspondence between homotopy and type theory:

type theory	homotopy theory
type A	space
$x, y : A$	points in A
$p : IdAxy$	path from x to y
$w : Id(IdAxy)pq$	homotopies

1.13 TODO Fix lhs2tex here

Using this correspondence, uniqueness of identity proofs can be interpreted as having every path in our space to be homotopic to the constant path. In homotopy theory it is easy to see that this is not the case in general. The simplest counter example being the circle: we can go around the circle n-times (in either direction), and none of those loops are homotopic to the constant loop.

1.14 Truncations

Looking at a type its (higher) identity types leads to an interesting way to classify types. The simplest possible structure we can have for an identity type of some type A is if A has only one element, up to propositional equality. A type satisfying this property is called *contractible*:

$$\begin{aligned}&\mathit{isContractible} : \mathit{Set} \to \mathit{Set}\\ &\mathit{isContractible}\ A = \Sigma\ A\ (\lambda \mathit{center} \to (x : A) \to (\mathit{Id}\ A\ \mathit{center}\ x))\end{aligned}$$

Translating this statement back to the language of homotopy theory, it indeed corresponds to the homotopic notion of contractibility. An example of a contractible type is the unit type $\top$. In fact, all contractible types are isomorphic to $\top$.

Contractible types indeed have the simplest possible structure of identity types: all identity proofs are propositionally equal to *refl*, hence the same holds for the higher equalities.

1.14.1 TODO Note that one should be careful with reading this type

There's some hidden continuity that makes this definition different from simply connectedness.

1.14.2 TODO Note that there are more interesting ways to build contractible types

Such as the disc.

For some types, if we look at the higher identity types, the structure eventually peters out and we end up with a contractible type. If the interesting identity type structure vanishes after n steps, the type is called a $n-2$-type or $n-2$-truncated:

$$
\begin{aligned}
&n - \mathit{truncated} : \mathit{Nat} \to \mathit{Set} \to \mathit{Set} \\
&n - \mathit{truncated}\ Z\ A = \mathit{isContractible}\ A \\
&n - \mathit{truncated}\ (S\ n)\ A = (x : A) \to (y : A) \to n - \mathit{truncated}\ n\ (\mathit{Id}\ A\ x\ y)
\end{aligned}
$$

1.14.3 TODO Fix notation here

Such that the counting in the code starts at -2 as well

1.14.4 TODO Note that the weird numbering comes from homotopy theory

1.14.5 TODO Note that every n-type is also an $n+1$-type

1.14.6 Sets

The most familiar types for the programmer are 0-types, also called *sets*: these are types like Nat that lack non-trivial equalities: every element is only equal to itself via *refl*. In other words: these are the types for which axioms K and UIP hold.

- **TODO** Elaborate, maybe?

1.14.7 Propositions

(-1)-types are also of interest: these types are either empty or isomorphic to $\top$, hence they are also called *propositions*. This might be confusing if one looks at type theory through the propositions as types lens. The stance we take here is that we do not regard every type as a proposition, but only ones that adhere certain properties.

Usually, when you are doing mathematics, we do not really care what proof a proposition has: we only care whether a proposition has a proof. Any two proofs of a proposition are regarded as equal. If we translate this into type theory, we get the following:

$$
\begin{aligned}
&\mathit{isProposition} : \mathit{Set} \to \mathit{Set} \\
&\mathit{isProposition}\ A = (x\ y : A) \to \mathit{Id}\ A\ x\ y
\end{aligned}
$$

This definition is equivalent to the definition of (-1)-types, in the following sense: if we can prove $\mathit{isProposition}\ A$ then we can also prove $n - \mathit{truncated}\ (-1)\ A$ and vice versa.

Note that we have two distinct (-1)-types: it is either empty ($\bot$) or, if it is inhabited, isomorphic to $\top$. Hence the universe of (-1)-types is similar to the complete Boolean algebra $\{t, f\}$ we know from classical logic.

1.15 Higher inductive types

1.15.1 TODO So far, we haven't seen anything *new*, only new ways to look at old problems.

1.15.2 TODO Also, we haven't seen any strict 1-types.

1.16 Univalence

1.16.1 TODO We cannot distinguish isomorphic types, so we want them to be equal.

2 Contributions

2.1 Introduction to homotopy type theory

There are several introductions to homotopy type theory (e.g. Awodey (2012) and Pelayo and Warren (2012)), but these are geared towards mathematicians who know about homotopy theory, but do not know about type theory. For the computer scientist who knows some type theory, but has never seen any homotopy theory, there is virtually no material.

> **Contribution:** We provide an introduction to homotopy theory for the computer scientist who has some familiarity with type theory, but does not have the background in homotopy theory.

2.2 Erasing propositions

One way to explain type theory is using the propositions-as-types analogy, also called the Curry-Howard correspondence. In practice, this identification is not precise enough. If we have two proofs of a proposition, we tend to regard those proofs as equal. We care more about the fact that we have a proof rather than how the proof is constructed exactly. However, there are types where we definitely do not want to regard all inhabitants to be equal, for example the natural numbers.

Instead of viewing all types as propositions, we identify propositions only with those types whose inhabitants are all (propositionally) equal to eachother:

$$\begin{array}{l} \mathit{isProposition} : \mathit{Set} \to \mathit{Set} \\ \mathit{isProposition}\ A = (x\ y : A) \to \mathit{Id}\ A\ x\ y \end{array}$$

In homotopy type theory a type that has this property is usually called *(-1)-truncated* or a *(-1)-type*.[1]

[1] The somewhat strange numbering comes from homotopy theory, where 0-truncated intuitively means that we have a set, (-1)-truncated a set of at most one element and (-2)-truncated an empty set

Examples of propositions are the empty type $\bot$ and the unit type $\top$. In fact, we can prove that if a type is inhabitated and it is a proposition, it is isomorphic to the unit type.

2.2.1 Comparison with collapsibility

The definition $isProposition$ looks a lot like *collapsibility* (Brady et al., 2004). Given some indexing type I, a family $D : I \rightarrow Set$ is called *collapsible* if for all indices i and inhabitants $x, y : D\ i$, x and y are definitionally equal. In other words: every $D\ i$ is either empty or has one element (up to definitional equality).

If we know that a family is collapsible, we can optimise its constructors and elimination operators by erasing certain parts, since we know that the relevant parts (if there are any) can be recovered from the indices.

Comparing the definition of collapsible families to $isProposition$, we notice that they are largely the same. $isProposition$ is a can be seen as an internalised version of collapsibility: we have replaced definitional equality with propositional equality.

A question one might ask is whether the optimisations based on collapsibility also hold for propositions. If our type theory satisfies canonicity, propositional equality implies definitional equality in the empty context. This means, that we can indeed use the concept of propositions for the same optimisations as those for collapsible families.

In homotopy type theory, one usually adds axioms such as the univalence axiom, or axioms to implement higher inductive types. This means we lose canonicity hence we no longer have that propositional equality always implies definitional equality.

Not all is lost: let B be a type for which all proofs $p : Id\ B\ x\ y$ are definitionally equal to *refl* and let A be a (-1)-type, then the only functions we $f : A \rightarrow B$ we can write are (definitionally) constant functions, hence we can at run-time ignore what value of A we get exactly.

> **Contribution:** We identify cases where (-1)-types can be safely erased: we provide an optimisation in the spirit of Brady et al. (2004)

2.2.2 Comparison with Prop in Coq

In Coq we can make the distinction between informative or computational parts of our program (everything that lives in *Set*) and logical parts (everything that lives in *Prop*). This distinction is also used when extracting a Coq development to another language: we can safely erase terms of sort $Prop$.

Another property of the *Prop* universe is that it is impredicative: propositions can quantify over propositions. (-1)-types also have this property in a certain sense.

> **Contribution:** We provide a comparison between Coq's *Prop* universe to the (-1)-types of homotopy type theory and our run-time optimisation.

2.3 Applications of homotopy type theory to programming

2.3.1 Quotients

Higher inductive types provide for a natural construction of quotients. In pseudo-Agda this would look as follows:

```
data Quotient (A : Set) (R : A → A → Proposition) : Set where
  project : A → Quotient A R
  relate  : (x y : A) → R x y → Id (Quotient A R) (project x) (project y)
  contr   : (x y : Quotient A R) → (p q : Id (Quotient A R) x y)
                                  → Id (Id (Quotient A R) x y) p q
```

Quotienting out by the given relation means that we need to regard two terms x and y related to eachother with R as propositionally equal, which is witnessed by the *relate* constructor. In order for the result to be a set (in the sense of being a discrete groupoid), we also need to ensure that it satisfies UIP, which in turn is witnessed by the *contr* constructor.

> **Contribution:** We compare this approach to quotients to other approaches, such as definable quotients (Altenkirch et al.).

2.3.2 Views on abstract types

The univalence axiom should make it more easy to work with views in a dependently typed setting.[2]

There are cases where it makes sense to have an implementation that has more structure than we want to expose to the user using the view. Instead of having an isomorphism, we then have a section/retraction pair. Since we have quotients to our disposal, we can make this into an isomorphism.

> **Contribution:** Identify examples of non-isomorphic views and determine whether quotients are easy to work with for this use case.

References

Thorsten Altenkirch, Thomas Anberrée, and Nuo Li. Definable quotients in type theory.

Steve Awodey. Type theory and homotopy. In *Epistemology versus Ontology*, pages 183–201. Springer, 2012.

[2][[http://homotopytypetheory.org/2012/11/12/abstract-types-with-isomorphic-types/][http://homotopytypetheory.org/2012/11/12/abstract-types-with-isomorphic-types/]]

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, pages 115–129. Springer, 2004.

Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, 1996.

Per Martin-Löf. Constructive mathematics and computer programming. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 167–184. Prentice-Hall, Inc., 1985.

Álvaro Pelayo and Michael A Warren. Homotopy type theory and Voevodsky's univalent foundations. *arXiv preprint arXiv:1210.5658*, 2012.