Master thesis proposal: Erasing propositions in homotopy type theory

Gabe Dijkstra

April 11, 2013

When writing programs in a dependently typed language, we often find ourselves writing terms whose sole purpose is to convince the type checker that we are doing the right thing, instead of being relevant to the actual computation at hand. The concept of h-propositions from homotopy type theory helps us describes a class of such terms. We propose an optimisation based on h-propositions and compare this to other approaches, such as Coq's program extraction. Apart from this, we provide an introduction to homotopy theory and discuss several applications to dependently typed programming.

1 Homotopy type theory

In *homotopy theory* we are interested in studying paths between points of some topological space. All the paths of a space form again a space, called the path space. In such a path space, we have the following notion of (higher) paths: if we can continuously deform one path, i.e. a point in the path space, into another, we have a higher path in the path space. These higher paths are called *homotopies*. Since the path space is a space, we can repeat this construction indefinitely and talk about higher homotopies.

In Martin-Löf's type theory we have a notion of equality interal to the type theory, namely the *identity types* (section 1.1). These happen to behave in ways similar to the aforementioned homotopies. *Homotopy type theory* studies these similarities, which roughly boils down to the following correspondence:

type theory	**homotopy theory**
A is a type	A is a space
$x, y : A$	x and y are points in A
p, q: $\equiv y$	p and q are paths from x to y
$w : p \equiv q$	w is a homotopy between paths p and q

This table can be expanded indefinitely: there is no need to stop at homotopies between paths: we also have identity types analogous to homotopies between homotopies, ad infinitum.

In homotopy theory, one classifies spaces along the structure of their (higher) homotopies. Using the correspondence, this way of thinking can be translated to type theory, where we arrive at the notion of n-types (section 1.3). The (-1)-types (or h-propositions) are of particular interest in this thesis, as they provide a nice way to classify types of which its inhabitants only serve to convince the type checker as opposed to having any computational relevance. In section 2.2 we discuss how these can be used to optimise dependently typed programs.

The homotopic way of thinking has also inspired additions to the type theory, such as *higher inductive types* (section 1.4) and the *univalence axiom* (section 1.5). These additions have been successfully used to formalise proofs from topology, such as the calculation of the fundamental group of the circle (Licata and Shulman, 2013). Apart from their use in formal mathematics, these additions might also prove useful for programming purposes, which is the subject of section 2.3.

1.1 Identity types of Martin-Löf's type theory

Martin-Löf (1985) introduced a notion of equality in his type theory: *propositional* equality, defined using so called identity types. These types can be formulated in Agda syntax as follows:

$$\begin{aligned}&\textbf{data}\ \mathit{Id}\ (A : \mathit{Set}) : A \to A \to \mathit{Set}\ \textbf{where}\\ &\quad \mathit{refl} : (x : A) \to \mathit{Id}\ A\ x\ x\end{aligned}$$

In Martin-Löf's type theory we do not have pattern matching to work with inhabitants of an inductive type, but use the induction principles (or elimination operators) instead. The induction principle of the Id type is usually called J and has the following type:

$$\begin{aligned}J : \quad & (A : \mathit{Set})\\ & \to (P : (x\ y : A) \to (p : \mathit{Id}\ A\ x\ y) \to \mathit{Set})\\ & \to (c : (x : A) \to P\ x\ x\ (\mathit{refl}\ x))\\ & \to (x\ y : A) \to (p : \mathit{Id}\ A\ x\ y)\\ & \to P\ x\ y\ p\end{aligned}$$

Along with this type, we have the following computation rule:

$$J\ A\ P\ c\ x\ x\ (\mathit{refl}\ x) \triangleq c\ x$$

We will make use of a slightly different, but equivalent formulation of these types, due to Paulin-Mohring:

$$\begin{array}{l}\textbf{data}\ Id'\ (A : Set)\ (x : A) : A \to Set\ \textbf{where}\\ \quad refl : Id'\ A\ x\ x\end{array}$$

with induction principle:

$$\begin{array}{ll}J' : & (A : Set)\\ & \to (P : (x\ y : A) \to (p : Id'\ A\ x\ y) \to Set)\\ & \to (c : P\ x\ x\ refl)\\ & \to (x\ y : A) \to (p : Id'\ A\ x\ y)\\ & \to P\ x\ y\ p\end{array}$$

and computation rule:

$$J'\ A\ P\ c\ x\ refl \stackrel{\Delta}{=} c$$

To make things look more like the equations we are used to, we will for the most part use infix notation, leaving the type parameter implicit: $Id\ A\ x\ y$ becomes $x \equiv y$. In some cases we will fall back to the $Id\ A\ x\ y$ notation, when it is a bit harder to infer the type parameter.

Using the identity types and their induction principles, we can show that it is an equivalence relation, i.e. given $A : Set$ and $x\ y\ z : A$, we can find inhabitants of the following types:

- $refl : Id\ A\ x\ x$
- $symm : Id\ A\ x\ y \to Id\ A\ y\ x$
- $trans : Id\ A\ x\ y \to Id\ A\ y\ z \to Id\ A\ x\ z$

1.1.1 Function extensionality

To prove properties about functions, it is often useful to have the principle of function extensionality:

$$\begin{array}{lll}functionExtensionality & : & (A\ B : Set) \to (f\ g : A \to B)\\ & & \to ((x : A) \to f\ x \equiv g\ x)\\ & & \to f \equiv g\end{array}$$

However, in Martin-Löf's type theory there is no term of that type. Since this theory has the canonicity property, having a propositional equality in the empty context, i.e. $\vdash p : x \equiv y$, we know that p must be canonical: it is definitionally equal to $refl$. In order for it to type check, we then know that x and y must be definitionally equal. Now consider the functions $f = \lambda n \to n + 0$ and $g = \lambda n \to 0 + n$, with the usual definition of $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, we can prove that $(n : \mathbb{N}) \to f\ n \equiv g\ n$, but not that $f \equiv g$, since that would imply they are definitionally equal, which they are not.

1.1.2 Uniqueness of identity proofs

The canonicity property implies that if, in the empty context, we have two identity proofs $p\ q : Id\ A\ x\ y$, these proofs are both $refl$, hence they are equal to eachother. Using dependent pattern matching, we can prove this property in Agda, called *uniqueness of identity proofs*:

$$UIP : (A : Set)\ (x\ y : A)\ (p\ q : Id\ A\ x\ y) \to Id\ (Id\ A\ x\ y)\ p\ q$$
$$UIP\ A\ x\ .x\ refl\ refl = refl$$

Proving this using J instead of dependent pattern matching has remained an open problem for a long time and has eventually been shown to be impossible (Hofmann and Streicher, 1996). As an alternative for J, Streicher introduced the induction principle K:

$$\begin{aligned} K\ :\ & (A : Set)\ (x : A)\ (P : Id\ A\ x\ x \to Set) \\ & \to P\ refl \\ & \to (p : Id\ A\ x\ x) \\ & \to P\ c \end{aligned}$$

Using K we can prove UIP, and the other way around. It has been shown that in type theory along with axiom K, we can rewrite definitions written with dependent pattern matching to ones that use the induction principles and axiom K (Goguen et al., 2006).

1.2 Groupoid interpretation of types

One way to show that we cannot prove UIP and K from J, is to construct models of type theory with identity types in which there are types that do not exhibit the UIP property. In Hofmann and Streicher (1996), the authors note that types have a groupoid structure. This means that if we can find a groupoid that violates UIP, translated to the language of groupoids, then UIP is not provable within Martin-Löf's type theory with identity types.

We have a notion of composition of proofs of propositional equality: the term $trans : Id\ A\ x\ y \to Id\ A\ y\ z \to Id\ A\ x\ z$, as such we will use the notation $\_ \circ \_$ instead of $trans$. The same goes for $symm : Id\ A\ x\ y \to Id\ A\ y\ x$, which we will denote as $\_^{-1}$. We can prove that this gives us a groupoid, i.e. we can prove the following laws hold:

Given $a, b, c, d : A$ and $p : a \equiv b$, $p : b \equiv c$ and $q : c \equiv d$ we have:

- Associativity: $p \circ (q \circ r) \equiv (p \circ q) \circ r$
- Left inverses: $p\ ^{-1} \circ p \equiv refl$
- Right inverses: $p \circ p\ ^{-1} \equiv refl$
- Left identity: $refl \circ p \equiv p$
- Right identity: $p \circ refl \equiv p$

Groupoids can be seen as categories in which every arrow is invertible. Using this correspondence, $p{:}x \equiv y$ can be seen as an arrow $p{:}x \to y$. In this language, UIP means that if we have two arrows $p, q : x \to y$, then p and q are the same. The category consisting of two objects x and y with distinct arrows $p, q : x \to y$ along with their inverses is then a counterexample of UIP.

One thing we glossed over is what kind of equalities we were talking about: associativity, etc. all hold up to propositional equality one level higher. The identity type $Id\ A\ x\ y$ is of course a type and therefore has a groupoid structure of its own. Every type gives rise to a tower of groupoids that can interact with eachother. These structures, called ∞-groupoids , also show up in homotopy theory, hence we have the correspondence between types and spaces as mentioned earlier.

1.3 Truncations

The tower of (nested) identity types of a type can tell us all sorts of things. For example, for a lot of types, the identity types become less complicated the more deeply they are nested, until they reach a the point where they are isomorphic to the unit type, $\top$. In homotopy theory, spaces homotopic to the space consisting of one point, are called contractible spaces. One way to formulate this in type theory is with the following definition:

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

If the structure of the identity types peters out after a nesting depth of n, we call such a type $(n-2)$-truncated, or a $(n-2)$-type[1]:

$$
\begin{aligned}
&n\text{-truncated} : \mathbb{N} \to Set \to Set \\
&n\text{-truncated}\ (-2)\ \ A = isContractible\ A \\
&n\text{-truncated}\ (S\ n)\ A = (x : A) \to (y : A) \to n\text{-truncated}\ n\ (Id\ A\ x\ y)
\end{aligned}
$$

These truncation levels have the property that every n-type is also an $(n+1)$-type, i.e. n-truncated defines a filtration on the universe of types.

The *contractible* types are the types that are isomorphic to $\top$ in the sense that every inhabitant of a contractible type is unique up to propositional equality. In section 1.4 we will see examples of contractible types that have more than one canonical element.

Types of truncation level -1 are called *h-propositions*. (-1)-types are either empty ($\bot$) or, if they are inhabited, contractible, hence isomorphic to $\top$. h-propositions satisfy the principle of proof irrelevance:

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

[1]The somewhat strange numbering, starting at -2 comes from homotopy theory, where they first considered groupoids without any higher structure to be 0-truncated and then generalised backwards.

This fits the classical view of propositions and their proofs: we only care about whether or not we have a proof of a proposition and do not distinguish between two proofs of the same proposition.

h-Sets are the 0-types. Every type that satisfies K and UIP is an h-set. This is the highest truncation level we can get to in Agda, without adding extra axioms.

1.4 Higher inductive types

In order to do some homotopy theory in type theory, we need to be able to construct interesting spaces in our type theory. One way to define spaces inductively, is by giving the constructors of the points in the space along with (higher) paths between them. For example, the interval can be seen as two points and a path connecting these two points, which in pseudo-Agda would look like:

$$\begin{aligned}&\textbf{data}\ \mathit{Interval} : \mathit{Set}\ \textbf{where}\\ &\quad \mathit{zero} : \mathit{Interval}\\ &\quad \mathit{one}\ : \mathit{Interval}\\ &\quad \mathit{segment} : \mathit{zero} \equiv \mathit{one}\end{aligned}$$

Inductive types with added equalities/paths are called *higher inductive types*. As of yet, these have not been implemented in Agda, but there are ways to simulate them.[2]

Apart from defining the constructors and its paths, we also need an induction principle. Intuitively, to write a function from a higher inductive type into something else, we need to specify what to do with the constructors, just as with normal inductive types, but we need to do this in such a way that we preserve the added equalities.

As one would expect from homotopy theory, we can prove that the type $\mathit{Interval}$ is contractible, even though it has more than one canonical element. It is indeed isomorphic to the unit type, since isomorphism is defined with propositional equality and we have a proof of $\mathit{zero} \equiv \mathit{one}$.

1.5 Univalence

A property satisfied by a popular model of homotopy theory, the category of simplicial sets, is the *univalence* property. In type theory terms this roughly means we have the following axiom:

$$\mathit{univalence} : (A\ B : \mathit{Set}) \to \mathit{Iso}\ A\ B \to \mathit{Id}\ \mathit{Set}\ A\ B$$

where $\mathit{Iso}\ A\ B$ is a record containing functions between A and B and proofs that these functions are eachother's inverses.

[2] `http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/`

Unfortunately, the full axiom is a bit more involved, since we have additional coherence properties that need to hold: the proofs in the $Iso\ A\ B$ record need to interact in a certain way. However, the axiom as stated above does hold for all h-sets, where there higher structure is simple enough to not have to worry about coherence.

2 Contributions

2.1 Introduction to homotopy type theory

There are several introductions to homotopy type theory (e.g. Awodey (2012), Pelayo and Warren (2012) and Rijke (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 type theory for the computer scientist who has some familiarity with type theory, but does not have the background in homotopy theory.

2.2 Optimisations based on h-propositions

In homotopy type theory we have a class of types called h-propositions or (-1)-types, satisfying the proof irrelevance principle:

$$proofIrrelevance : Set \to Set$$
$$proofIrrelevance\ A = (x\ y : A) \to Id\ A\ x\ y$$

i.e., we can find a term of type $(A{:}Set) \to n$-truncated $(-1)\ A \to proofIrrelevance\ A$. The other way around also holds: if a type exhibits proof irrelevance, it is an h-proposition. Examples of h-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 of h-propositions looks a lot like *collapsibility* (Brady et al., 2004). Given some indexing type I, a family $D : I \to Set$ 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 induction principles 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 that of h-propositions, we notice that they are largely the same. The latter 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 h-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 assume extra equalities to hold in order 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 an h-proposition, then the only functions we $f : A \to B$ 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 h-propositions 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. h-Propositions also have this property in a certain sense.

> **Contribution:** We provide a comparison between Coq's $Prop$ universe to the h-propositions 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:

$$
\begin{aligned}
&\textbf{data}\ Quotient\ (A : Set)\ (R : A \to A \to Proposition) : Set\ \textbf{where}\\
&\quad project : A \to Quotient\ A\ R\\
&\quad relate\ \ : (x\ y : A) \to R\ x\ y \to Id\ (Quotient\ A\ R)\ (project\ x)\ (project\ y)\\
&\quad contr\ \ : (x\ y : Quotient\ A\ R) \to (p\ q : Id\ (Quotient\ A\ R)\ x\ y)\\
&\qquad\qquad\qquad\qquad \to Id\ (Id\ (Quotient\ A\ R)\ x\ y)\ p\ q
\end{aligned}
$$

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.[3]

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 transform any such pair into an isomorphism.

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

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

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.

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.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, pages 521–540. Springer, 2006.

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

Daniel R Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. *arXiv preprint arXiv:1301.3443*, 2013.

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.

Egbert M Rijke. Homotopy type theory. Master's thesis, Universiteit Utrecht, 2012.