Master thesis proposal: Erasing propositions in homotopy type theory

Gabe Dijkstra

April 9, 2013

- Introduce problem of "computationally useless" terms in DTP
- We need some sort of distinction
- One way is via the notion of propositions
- There is such a notion of proposition in HoTT: (-1)-types
- We investige whether this is a useful notion to use in optimisations
- And also some other stuff

1 Homotopy type theory

In *homotopy theory* we are interested in studying paths between points of some topological space. If we can continuously deform one path into the other, we call the paths *homotopic* to eachother. This concept can be taken higher: we can also consider continuous deformations between such homotopies, and so on.

Equality in Martin-Löf's type theory is often done by using *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 to the following correspondence:

type theory	**homotopy theory**
A is a type	A is 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.

Bruggetje hier?

This correspondence sheds new light on old problems from type theory (section 1.2.1), providing some sort of geometric intuition as to why we cannot write certain terms.

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)
- Data type definition in Agda syntax
- Give type of the elimination principle J

1.1.1 Function extensionality

- We cannot prove the following to be propositionally equal:
  - $f = \lambda n \to n + 0$ and $g = \lambda n \to 0 + n$
  - We can prove $(n : Nat) \to f\ n \equiv g\ n$
  - One way to show this is via canonicity and all.
  - But the they aren't observationally different

1.1.2 Uniqueness of identity proofs

- State UIP
- State how we can proof this in Agda

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

- Proving this with J instead of dependent pattern matching remained an open problem for a long time
- Streicher introduced another elimination principle K instead of J
- UIP does follow from K

1.2 Groupoids interpretation to construct countermodels

- If we cannot prove UIP/K, what can be prove?
- Consider *trans* to be composition
- We can prove the groupoid lawsHofmann and Streicher (1996)
- Note the higher equalities -¿ ∞-groupoids∞-groupoids¿ homotopy groupoids, hence the aforementioned correspondence of inhabitants of identity types corresponding to (higher) paths.
- Models of homotopy theory (formalised as model categories) are also models of Martin-Löf's type theory .

1.2.1 UIP and K revisited

- Motivate that we now have nice geometric intuitions behind the counterexamples.

1.3 Truncations

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

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

- (-2)-typeis *contractible*: $\top$
- (-1)-typesare *propositions*: $\top$ and $\bot$
- 0-types are *sets*: satisfy K and UIP.
- Strange numbering fault of homotopy theorists
- Every n-type is also an $(n+1)$-type.

1.4 Higher inductive types

- So far we have seen no way to break UIP
- Example from topology: S1
- How would one do this in type theory?
- *Higher inductive types*: add paths
- Show some pseudo-Agda, as it hasn't been implemented yet
- Although its semantics are pretty well understood, syntax is not quite there yet

1.5 Univalence

- In Martin-Löf's type theory , isomorphic types are observationally indistinguishable
- This is not reflected in propositional equality
- Something about coherence?
- The univalence axiom fixes this
- The axiom implies functon extensionality

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 type theory for the computer scientist who has some familiarity with type theory, but does not have the background in homotopy theory.

2.2 Optimisation à la collapsibility, based on (-1)-types

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 $\mathit{isProposition}$ looks a lot like *collapsibility* (Brady et al., 2004). Given some indexing type I, a family $D : I \rightarrow \mathit{Set}$ 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 $\mathit{isProposition}$, we notice that they are largely the same. $\mathit{isProposition}$ 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 $p : \mathit{Id}\ B\ x\ y$ 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/`

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.

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.