Programming in homotopy type theory and erasing propositions

Gabe Dijkstra

Department of Information and Computing Sciences
Faculty of Science, Utrecht University

August 27, 2013

Martin-Löf type theory

- Can be seen as:
  - Logic to do formal mathematics in
  - Can be seen as a programming language
- Agda is an implementation of Martin-Löf type theory extended with pattern matching
- Martin-Löf type theory does not have pattern matching, but elimination principles

Introduction homotopy type theory

- Homotopy type theory studies propositional equality in (intensional) Martin-Löf type theory
- Propositional equality in type theory is a difficult concept:
  - Intensional Martin-Löf type theory
    - Cannot derive function extensionality
      $((f\ g : A \to B) \to ((x : A) \to f\ x \equiv g\ x) \to f \equiv g)$
    - Type checking is decidable
  - Extensional Martin-Löf type theory
    - Can derive function extensionality
    - Type checking is undecidable
  - Heterogeneous equality
  - Observational type theory

Introduction homotopy type theory

- Interpret an equality $p : x \equiv y$ as a path in a topological space
- Special year at IAS in Princeton: book
  - Focus on formalising mathematics
  - Aimed at mathematicians unfamiliar with type theory

Research question

What is homotopy type theory and why is interesting to do programming in?

Homotopy theory

- *Topology*: study of spaces and *continuous maps* between them
- *Homotopy*: study of *continuous deformations* in topological spaces
- Continuous deformation of point x into y is a continuous function: $p : [0, 1] \to A$ such that $p\,0 = x$ and $p\,1 = y$
- Continuous deformations have an interesting structure:
  - There is a constant deformation
  - Can be composed
  - Can be inverted
- They form a groupoid *up to homotopy*

Identity types

$$\textbf{data}\ \mathit{Id}\ (A : \mathit{Type})\ (x : A) : (y : A) \to \mathit{Type}\ \textbf{where}$$
$$\quad \mathit{refl} : \mathit{Id}\ A\ x\ x$$

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

Identity types

$$\textbf{data}\ \mathit{Id}\ (A : \mathit{Type})\ (x : A) : (y : A) \to \mathit{Type}\ \textbf{where}$$
$$\quad \mathit{refl} : \mathit{Id}\ A\ x\ x$$

$\mathit{Id}\ A\ x\ y$ forms an equivalence relation:

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

$\mathit{Id}\ A\ x\ y$ is also a groupoid *up to propositional equality*

Uniqueness of identity proofs

- *Id* has only one constructor: *refl*
- Shouldn't all terms of type $Id\ A\ x\ y$ be equal to eachother?

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

- Can we prove this using J?

J versus K

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

$$
\begin{aligned}
K \;:\; & (A : \mathit{Type})\ (x : A)\ (P : \mathit{Id}\ A\ x\ x \to \mathit{Type}) \\
& \to P\ \mathit{refl} \\
& \to (c : \mathit{Id}\ A\ x\ x) \\
& \to P\ c
\end{aligned}
$$

h-propositions and h-sets

- In homotopy theory we classify spaces along their homotopy groupoids
- In homotopy type theory we can classify types along their identity types
- Contractible type: $\Sigma\ (\mathit{center} : A)\ .\ ((x : A) \to \mathit{Id}\ A\ \mathit{center}\ x)$
  - Example: $\top$
- h-proposition: $(x\ y : A) \to \mathit{isContractible}\ (\mathit{Id}\ A\ x\ y)$
  - Examples: $\top$ and $\bot$
  - Satisfies *proof irrelevance*: $(x\ y : A) \to x \equiv y$
- h-set: $(x\ y : A) \to \mathit{is\text{-}hProp}\ (\mathit{Id}\ A\ x\ y)$
  - Example: *Bool*
  - Satisfies *uniqueness of identity proofs*

h-propositions and h-sets

- Are there types that are not h-sets, i.e. types that violate uniqueness of identity proofs?
- Higher inductive types

data $Circle : Type$ **where**
$base : Circle$
$loop : base \equiv base$

$Circle\text{-}rec : (B : Set)$
$\rightarrow (b : B)$
$\rightarrow (p : b \equiv b)$
$\rightarrow Circle \rightarrow B$

Univalence

- Univalence: $(A\ B : \mathit{Type}) \to A$ isomorphism $B \to A \equiv B$
- *Type* does not satisfy uniqueness of identity proofs:
  - $\mathit{refl} : \mathit{Bool} \equiv \mathit{Bool}$
  - $\mathit{univalence}\ \mathit{Bool}\ \mathit{Bool}\ \mathit{notIso} : \mathit{Bool} \equiv \mathit{Bool}$
- Univalence implies function extensionality

Applications of homotopy type theory

- Quotient types using higher inductive types
- Views for abstract types

Implementation efforts

- Status quo: use Agda/Coq and postulate the extra equalities
- Is sufficient if all you want to do is type checking
- Computations get stuck
- Computational content of univalence is an open problem
- Licata/Harper: canonicity for a restricted version of homotopy type theory
  - No decidability result for type checking

Conclusions and future work

What is homotopy type theory and why is interesting to do programming in?

- Giving up pattern matching is a (big) step backwards
- Higher inductive types and univalence can become two steps forwards

Erasing propositions

- When we write certified programs we can distinguish between:
  - *proof* (of correctness) parts
  - *program* parts
- The proof parts are only needed during type checking
- At run-time we do not want to carry the proof parts around:
  - We want to *erase* those parts after type checking

Erasing propositions

$$sort : (xs : List\ \mathbb{N}) \to \Sigma\ (ys : List\ \mathbb{N})\ .\ (isSorted\ xs\ ys)$$

- *isSorted xs ys* is only interesting during type checking
- We only care that we have a proof, not what kind of proof it is
  - Recall proof irrelevance: $(x\ y : A) \to x \equiv y$
  - *h*-propositions
- At run-time we want a function $sort' : List\ \mathbb{N} \to List\ \mathbb{N}$

Can we provide an optimisation based on the concept of *h*-propositions?

Erasing propositions

- Can't we separate concerns?
  - $sort' : List\ \mathbb{N} \to List\ \mathbb{N}$
  - $sortCorrect : (xs : List\ \mathbb{N}) \to isSorted\ xs\ (sort'\ xs)$
- This does not always work:
  - $elem : (A : Type)\ (xs : List\ A)\ (i : \mathbb{N}) \to i < length\ xs \to A$
  - We need $i < length\ xs$ during type checking

Erasing propositions in Agda

- In Agda we can mark things as *irrelevant*:

$$\begin{array}{l}\textbf{record}\ \Sigma\text{-}irr\ (A : Type)\ (B : A \to Type) : Type\ \textbf{where}\\ \quad \textbf{constructor}\ \_,\_\\ \quad field\\ \quad\quad fst\ \ \ : A\\ \quad\quad .snd : B\ fst\end{array}$$

$$elem : (A : Type)\ (xs : List\ A)\ (i : \mathbb{N}) \to\ .(i < length\ xs) \to A$$

- We may not pattern match on irrelevant arguments
- Irrelevant arguments may only be passed on to irrelevant contexts
  - This prevents us from writing $.A \to A$

Collapsibility

$$\begin{aligned}&\textbf{data}\ \_ < \_ : \mathbb{N} \to \mathbb{N} \to \mathit{Type}\ \textbf{where}\\ &\quad \mathit{ltZ} : (y : \mathbb{N}) \to Z < S\ y\\ &\quad \mathit{ltS} : (x\ y : \mathbb{N}) \to x < y \to S\ x < S\ y\end{aligned}$$

with elimination operator

$$\begin{aligned}<\text{-}\mathit{elim}\ :\ &(P : (x\ y : \mathbb{N}) \to x < y \to \mathit{Type})\\ &(m_Z : (y : \mathbb{N}) \to P\ 0\ (S\ y)\ (\mathit{ltZ}\ y))\\ &(m_S : (x\ y : \mathbb{N}) \to (\mathit{pf} : x < y) \to P\ x\ y\ \mathit{pf}\\ &\quad \to P\ (S\ x)\ (S\ y)\ (\mathit{ltS}\ x\ y\ \mathit{pf}))\\ &(x\ y : \mathbb{N})\\ &(\mathit{pf} : x < y)\\ \to\ &P\ x\ y\ \mathit{pf}\end{aligned}$$

and computation rules

$$\begin{aligned}&<\text{-}\mathit{elim}\ P\ m_Z\ m_S\ 0\ \ \ (S\ y)\ (\mathit{ltZ}\ y) = m_Z\ y\\ &<\text{-}\mathit{elim}\ P\ m_Z\ m_S\ (S\ x)\ (S\ y)\ (\mathit{ltS}\ x\ y\ \mathit{pf}) = m_S\ x\ y\ \mathit{pf}\ (<\text{-}\mathit{elim}\ P\ m_Z\end{aligned}$$

Collapsibility

- A canonical value $p : x < y$ is determined completely by its indices x and y
- Only way to inspect p is via $<$-*elim*
- $<$-*elim* does not need to inspect p
- p can be erased
- When can we do this?
  - Collapsible family: given $I : \mathit{Type}$, $D : I \to \mathit{Type}$ is *collapsible* if for every $x, y : D\ i$:

$$\vdash x, y : D\ i \text{ implies} \vdash x \stackrel{\Delta}{=} y$$

- This looks familiar

$$\mathit{is\text{-}hProp} : (A : \mathit{Type}) \to \mathit{Type}$$
$$\mathit{is\text{-}hProp}\ A = (x\ y : A) \to x \equiv y$$

Internalising collapsibility

- Collapsibility looks like an indexed version of h-propositions
- Can we *internalise* the collapsibility concept?

$$\mathit{isInternallyCollapsible} : (I : \mathit{Type})\ (A : I \to \mathit{Type}) \to \mathit{Type}$$
$$\mathit{isInternallyCollapsible}\ I\ A = (i : I) \to (x\ y : A\ i) \to x \equiv y$$

- Do the two concepts coincide?
  - Internal collapsibility implies collapsibility
    if we have $\vdash p : x \equiv y$, then $p \stackrel{\Delta}{=} \mathit{refl}$ and $x \stackrel{\Delta}{=} y$
  - The other way around does not hold
    $\mathit{Id}\ A$ is a collapsible family for every A, but not internally collapsible: we cannot prove uniqueness of identity proofs

Internalising the collapsibility optimisation

- We can internalise the collapsiblity concept: *isInternallyCollapsible*
- Can we do the same with the optimisation, i.e. can we implement the following:

$$
\begin{aligned}
&\mathit{optimiseFunction} : \\
&\quad (I : \mathit{Type})\ (D : I \to \mathit{Type})\ (B : \mathit{Type}) \\
&\quad (\mathit{isInternallyCollapsible}\ I\ D) \\
&\quad (f : (i : I) \to D\ i \to B) \\
&\quad \to ((i : I) \to\ .(D\ i) \to B)
\end{aligned}
$$

- Why internalise it in the first place?
  - Collapsibility can only be established by the compiler
  - It is undecidable
  - Internalising it means the user can provide a proof if the compiler fails to do so

Internalising the collapsibility optimisation

$$
\begin{aligned}
&\mathit{optimiseFunction} : \\
&\quad (I : \mathit{Type})\ (D : I \to \mathit{Type})\ (B : \mathit{Type}) \\
&\quad (\mathit{isInternallyCollapsible}\ I\ D) \\
&\quad (f : (i : I) \to D\ i \to B) \\
&\quad \to ((i : I) \to\ .(D\ i) \to B)
\end{aligned}
$$

- Every $A\ i$ is either empty or isomorphic to $\top$
- We cannot "pattern match" on this fact: type inhabitation is undecidable

$$
\begin{aligned}
&\mathit{isInternallyCollapsibleDecidable} : (I : \mathit{Type})\ (D : I \to \mathit{Type}) \to \mathit{Type} \\
&\mathit{isInternallyCollapsibleDecidable}\ I\ D = (i : I) \\
&\quad \to (((x\ y : D\ i) \to x \equiv y)\ \times\ (D\ i\ +\ (D\ i \to \bot)))
\end{aligned}
$$

Internalising the collapsibility optimisation

- If we use *isInternallyCollapsibleDecidable* instead of *isInternallyCollapsible*, we can implement *optimiseFunction*
- We can also prove its correctness:

$$
\begin{aligned}
&\mathit{optimiseFunctionCorrect} : \\
&\quad (I : \mathit{Type})\ (D : I \to \mathit{Type})\ (B : \mathit{Type}) \\
&\quad (\mathit{pf} : \mathit{isInternallyCollapsibleDecidable}\ I\ D) \\
&\quad (f : (i : I) \to D\ i \to B) \\
&\quad (i : I)\ (x : D\ i) \\
&\quad \to \mathit{optimiseFunction}\ I\ D\ B\ \mathit{pf}\ f\ i\ x \equiv f\ i\ x
\end{aligned}
$$

- Is it actually an optimisation?
- *pf* provides us with a function $(i : I) \to D\ i$ that we use to recover the erased value
- *pf* is written by the user: no guarantees on its time complexity
- We can write terms in an *EDSL* that keeps track of time complexity

Internal collapsibility and homotopy type theory

- In "plain" Martin-Löf type theory run-time can be seen as evaluation in the empty context
- In homotopy type theory we have axioms for the added equalities
- Does the optimisation still work?
- *optimiseFunctionCorrect* still type checks
- but it only establishes *propositional* equality

Internal collapsibility and homotopy type theory

data $I : Set$ **where**

$zero : Interval$

$one : Interval$

$segment : zero \equiv one$

with elimination principle

$I\text{-}elim : (B : Type)$
$\rightarrow (b_0 : B)$
$\rightarrow (b_1 : B)$
$\rightarrow (p : b_0 \equiv b_1)$
$\rightarrow I \rightarrow B$

- I is an h-proposition
- Every function $I \rightarrow B$ is a "constant" function (up to propositional equality)

Internal collapsibility and homotopy type theory

$I\text{-}id : I \to I$
$I\text{-}id = I\text{-}elim\ I\ zero\ one\ segment$

$I\text{-}const\text{-}zero : I \to I$
$I\text{-}const\text{-}zero = I\text{-}elim\ I\ zero\ zero\ refl$

- $I\text{-}id \equiv I\text{-}const\text{-}zero$, but they do differ definitionally
  - $I\text{-}id\ one \stackrel{\Delta}{=} one$
  - $I\text{-}const\text{-}zero\ one \stackrel{\Delta}{=} zero$
- We cannot transform any $f : I \to B$ into $\widetilde{f} : .I \to B$ by presupposing the argument to be *zero*

Internal collapsibility and homotopy type theory

$$
\begin{aligned}
&\mathit{I\text{-}elim} : (B : \mathit{Type}) \\
&\quad \to (b_0 : B) \\
&\quad \to (b_1 : B) \\
&\quad \to (p : b_0 \equiv b_1) \\
&\quad \to I \to B
\end{aligned}
$$

- Sometimes it does work out
- Consider functions $f : I \to \mathit{Bool}$
- *Bool* only has *refl* paths
- We either have for every $i : I$ that $f\ i \stackrel{\Delta}{=} \mathit{True}$
  or we have for every $i : I$ that $f\ i \stackrel{\Delta}{=} \mathit{False}$
- If the p argument to *I-elim* is *refl*, it is safe to presuppose the I argument to be *zero*

Internal collapsibility and homotopy type theory

- Can we always find such a condition?

  data $\mathbb{N}$-*truncated* : *Type* **where**
  $0 : \mathbb{N}$-*truncated*
  $S : (n : \mathbb{N}\text{-}truncated) \to \mathbb{N}\text{-}truncated$
  $equalTo0 : (n : \mathbb{N}\text{-}truncated) \to 0 \equiv n$

  with elimination principle

  $\mathbb{N}$-*truncated-elim* : $(B : Type)$
  $\to (b_0 : B)$
  $\to (b_S : B \to B)$
  $\to (p : (b : B) \to b_0 \equiv b)$
  $\to \mathbb{N}\text{-}truncated \to B$

- $\mathbb{N}$-*truncated* is an *h*-proposition
- We have to check that for every $b : B$ we have $p\ b \stackrel{\Delta}{=} refl$
  - This is undecidable

Conclusions

- *Can we provide an optimisation based on the concept of h-propositions?*
  - In plain Martin-Löf type theory (with Agda's irrelevance mechanism): yes, if we restrict ourselves to decidable *h*-propositions, but time complexity is an issue
  - In homotopy type theory: generally not
- *Is homotopy type theory and why is interesting to do programming in?*
  - Yes: we get function extensionality, quotient types, better manipulation of isomorphic types via univalence
  - Not yet: computational content is lacking / we lose pattern matching