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 itself 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

- Basic idea: Interpret an equality $p : x \equiv y$ as a path in a topological space
- Martin-Löf type theory can be interpreted in homotopy theory
- Recent field of study
- Last year: special year at Institute for Advance Study in Princeton
  - Book: *Homotopy type theory: univalent foundations of mathematics*
  - 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

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
- The composition satisfies certain properties
- They form a groupoid *up to homotopy* (∞-groupoid)

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

$$
\begin{array}{l}
\textbf{data}\ \mathit{Id}\ (A : \mathit{Type})\ (x : A) : (y : A) \rightarrow \mathit{Type}\ \textbf{where} \\
\quad \mathit{refl} : \mathit{Id}\ A\ x\ x
\end{array}
$$

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

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

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?

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

Uniqueness of identity proofs

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

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

- 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

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: $\Sigma\ (center : A)\ .\ ((x : A) \to Id\ A\ center\ x)$
  - Example: $\top$

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: $\Sigma\ (center : A)\ .\ ((x : A) \to Id\ A\ center\ x)$
  - Example: $\top$
- *h*-proposition: $(x\ y : A) \to isContractible\ (Id\ A\ x\ y)$
  - Examples: $\top$ and $\bot$
  - Satisfies *proof irrelevance*: $(x\ y : A) \to x \equiv y$

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: $\Sigma\,(center : A)\ .\ ((x : A) \to Id\ A\ center\ x)$
  - Example: $\top$
- h-proposition: $(x\ y : A) \to isContractible\ (Id\ A\ x\ y)$
  - Examples: $\top$ and $\bot$
  - Satisfies *proof irrelevance*: $(x\ y : A) \to x \equiv y$
- h-set: $(x\ y : A) \to is\text{-}hProp\ (Id\ A\ x\ y)$
  - 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

$$
\begin{array}{l}
\textbf{data}\ \mathit{Circle} : \mathit{Type}\ \textbf{where} \\
\quad \mathit{base} : \mathit{Circle} \\
\quad \mathit{loop} : \mathit{base} \equiv \mathit{base}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Circle\text{-}rec} : (B : \mathit{Set}) \\
\quad \to (b : B) \\
\quad \to (p : b \equiv b) \\
\quad \to \mathit{Circle} \to B
\end{array}
$$

Univalence

- Univalence: $(A\ B : \mathit{Type}) \to A\ \simeq\ B \to A \equiv B$
- *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

Applications of homotopy type theory

- Quotient types using higher inductive types
  - Example: implement sets using lists
  - Quotient lists by the following relation:
    - $x \sim y$ if x contains the same elements as y, disregarding order and multiplicity
- Views for abstract types

Applications of homotopy type theory

- Quotient types using higher inductive types
  - Example: implement sets using lists
  - Quotient lists by the following relation:
    - $x \sim y$ if x contains the same elements as y, disregarding order and multiplicity
- Views for abstract types
  - Views can be used to prove properties of abstract types
  - A view can be seen as a reference implementation of the abstract type
  - Univalence can be used to express the specification more succinctly
  - Approach only works for isomorphic views

Applications of homotopy type theory

- Quotient types using higher inductive types
  - Example: implement sets using lists
  - Quotient lists by the following relation:
    - $x \sim y$ if x contains the same elements as y, disregarding order and multiplicity
- Views for abstract types
  - Views can be used to prove properties of abstract types
  - A view can be seen as a reference implementation of the abstract type
  - Univalence can be used to express the specification more succinctly
  - Approach only works for isomorphic views
  - We have extended this to work with non-isomorphic views as well

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 $sort' : List\ \mathbb{N} \to List\ \mathbb{N}$

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*:

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

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

Erasing propositions in Agda

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

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

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

- 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

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

with elimination operator

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

and computation rules

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

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?

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 : *Type*, $D : I \to$ *Type* is *collapsible* if for every $x, y : D\ i$:

$$\vdash x, y : D\ i \text{ implies} \vdash x \triangleq y$$

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 $x, y : D\ i$:

$$\vdash x, y : D\ i \text{ implies} \vdash x \triangleq y$$

- This looks familiar

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

Internalising collapsibility

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

$$
\begin{aligned}
&\mathit{isInternallyCollapsible} : (I : \mathit{Type})\ (A : I \rightarrow \mathit{Type}) \rightarrow \mathit{Type} \\
&\mathit{isInternallyCollapsible}\ I\ A = (i : I) \rightarrow (x\ y : A\ i) \rightarrow x \equiv y
\end{aligned}
$$

- Do the two concepts coincide?

Internalising collapsibility

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

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

- Do the two concepts coincide?
  - Internal collapsibility implies collapsibility
    if we have $\vdash p : x \equiv y$, then $p \triangleq \mathit{refl}$ and $x \triangleq y$

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 $p \triangleq \mathit{refl}$ and $x \triangleq y$
  - 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:

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

- Why internalise it in the first place?

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

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

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

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?

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?

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
  - We want *definitional* equality

Internal collapsibility and homotopy type theory

data I : Set **where**
$zero : Interval$
$one\ \ : Interval$

$segment : zero \equiv one$

with elimination principle

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

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

Internal collapsibility and homotopy type theory

$$
\begin{aligned}
&\textit{I-id} : I \to I \\
&\textit{I-id} = \textit{I-elim}\ I\ \textit{zero}\ \textit{one}\ \textit{segment}
\end{aligned}
$$

$$
\begin{aligned}
&\textit{I-const-zero} : I \to I \\
&\textit{I-const-zero} = \textit{I-elim}\ I\ \textit{zero}\ \textit{zero}\ \textit{refl}
\end{aligned}
$$

- $\textit{I-id} \equiv \textit{I-const-zero}$, but they do differ definitionally
  - $\textit{I-id}\ \textit{one} \triangleq \textit{one}$
  - $\textit{I-const-zero}\ \textit{one} \triangleq \textit{zero}$
- 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

Internal collapsibility and homotopy type theory

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

- Sometimes it does work out
- Consider functions $f : I \to \textit{Bool}$
- *Bool* only has *refl* paths
- We either have for every $i : I$ that $f\ i \triangleq \textit{True}$ or we have for every $i : I$ that $f\ i \triangleq \textit{False}$
- 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?

$$
\begin{array}{l}
\textbf{data } \mathbb{N}\textit{-truncated} : \textit{Type} \textbf{ where} \\
\quad 0 : \mathbb{N}\textit{-truncated} \\
\quad S : (n : \mathbb{N}\textit{-truncated}) \to \mathbb{N}\textit{-truncated} \\
\quad \textit{equalTo0} : (n : \mathbb{N}\textit{-truncated}) \to 0 \equiv n
\end{array}
$$

with elimination principle

$$
\begin{array}{l}
\mathbb{N}\textit{-truncated-elim} : (B : \textit{Type}) \\
\quad \to (b_0 : B) \\
\quad \to (b_S : B \to B) \\
\quad \to (p : (b : B) \to b_0 \equiv b) \\
\quad \to \mathbb{N}\textit{-truncated} \to B
\end{array}
$$

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

Internal collapsibility and homotopy type theory

- Can we always find such a condition?

$$\begin{array}{l} \textbf{data}\ \mathbb{N}\text{-}\mathit{truncated} : \mathit{Type}\ \textbf{where} \\ \quad 0 : \mathbb{N}\text{-}\mathit{truncated} \\ \quad S : (n : \mathbb{N}\text{-}\mathit{truncated}) \to \mathbb{N}\text{-}\mathit{truncated} \\ \quad \mathit{equalTo0} : (n : \mathbb{N}\text{-}\mathit{truncated}) \to 0 \equiv n \end{array}$$

with elimination principle

$$\begin{array}{l} \mathbb{N}\text{-}\mathit{truncated}\text{-}\mathit{elim} : (B : \mathit{Type}) \\ \quad \to (b_0 : B) \\ \quad \to (b_S : B \to B) \\ \quad \to (p : (b : B) \to b_0 \equiv b) \\ \quad \to \mathbb{N}\text{-}\mathit{truncated} \to B \end{array}$$

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

Conclusions

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

- *Is homotopy type theory and why is interesting to do programming in?*

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?*

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