What's the fuss about Homotopy Type Theory?

Jesper Cockx

KU Leuven

25 November 2013

Homotopy type theory is . . .

. . . a combination of distinct subjects

- Type theory
  - Mathematical logic
  - Theoretical computer science
  - Alternative to set theory
- Homotopy theory
  - Algebraic topology
  - Homological algebra
  - Category theory

Homotopy type theory is . . .

. . . a combination of distinct subjects

. . . a new foundation for mathematics

- Constructivity
- Proof-relevance
- Closer to informal reasoning
- More expressive than set theory

Homotopy type theory is . . .

. . . a combination of distinct subjects

. . . a new foundation for mathematics

. . . a field of current research

- Constructivity of univalence?
- How to define higher inductive types?
- New applications?

Why homotopy type theory?

We (for some value of "we") have long agreed that the language that the world is written in is higher category theory.

What is new now is that suddenly we realize that this higher category theory has an equivalent reformulation which, while equivalent, looks more fundamental, even, to some extent.

What's the fuss about Homotopy Type Theory?

1. The homotopy point of view
2. Higher inductive types
3. The univalence axiom
4. Application to generic programming

What's the fuss about Homotopy Type Theory?

1. The homotopy point of view
2. Higher inductive types
3. The univalence axiom
4. Application to generic programming

Syntax of type theory

$$a : T$$

"The term a has type T."

Type theory as a programming language

$$a : T$$

"The program a returns a value of type T."

Type theory as a proof assistant

$$a : T$$

"The proposition T is true,
as witnessed by the proof a"

Two interpretations of type theory

Programming language		Proof assistant
pairs	$A \times B$	‘and’
variants	$A + B$	‘or’
functions	$A \to B$	‘implies‘
unit type	$\top$	‘true’
empty type	$\bot$	‘false’

Dependent types

A dependent type $D\ x$ is a family of types indexed over terms $x : A$ of the base type A

Example: $\mathtt{Vec}\ A\ n$ is the type of vectors indexed by their length n

The Martin-Löf identity type

For each $x, y : A$ we have a type $Id_A\ x\ y$

This is called the (Martin-Löf) identity type

Terms of type $Id_A\ x\ y$ are identifications between x and y

Properties of the identity type

Reflexivity $\texttt{refl}_x : Id_A\ x\ x$

Symmetry If $p : Id_A\ x\ y$, then $p^{-1} : Id_A\ y\ x$

Transitivity If $p : Id_A\ x\ y$ and $q : Id_A\ y\ z$, then $p \blacksquare q : Id_A\ x\ z$

Congruence If $f : A \rightarrow B$ and $x, y : A$, then $ap_f : Id_A\ x\ y \rightarrow Id_B\ (f\ x)\ (f\ y)$

Substitution If P is a family over A and $p : Id_A\ x\ y$, then $p_* : P\ x \rightarrow P\ y$

Not a property of the identity type

Uniqueness of identity proofs

If $p, q : Id_A\ x\ y$, then $UIP_{p,q} : Id_{(Id_A\ x\ y)}\ p\ q$

UIP is *not* provable from the definition of Id

What else could there be hiding in $Id_A\ x\ y$?

We need a new way to think about these identifications . . .

Homotopy theory is about . . .

Topological spaces
(circle, sphere, torus, . . .)

Paths between points in these spaces

Homotopies between these paths

What can we do with paths?

We can take the constant path at a point

We invert a path

We can compose two paths

We can take the image of a path under a continuous function

We can transport points along a path from one fiber of a fibration to another

Homotopy interpretation of type theory

Types are spaces

Terms are points

Functions are continuous maps

Identifications are paths

Dependent types are fibrations

Path induction

Let $x_0 : A$ and let $P\ x\ p$ be a proposition about all $x : A$ with a $p : Id_A\ x_0\ x$.

The principle of path induction says:
In order to prove $P\ x\ p$ for all x and p, it is sufficient to prove $P\ x_0\ \mathtt{refl}_{x_0}$.

What's the fuss about Homotopy Type Theory?

1. The homotopy point of view
2. Higher inductive types
3. The univalence axiom
4. Application to generic programming

Inductive types

Inductive types are defined by a number of constructors

	Constructors
Bool	$\mathtt{true} : \mathit{Bool}$ $\mathtt{false} : \mathit{Bool}$
$\mathbb{N}$	$\mathtt{zero} : \mathbb{N}$ $\mathtt{suc} : \mathbb{N} \to \mathbb{N}$.

The idea of higher inductive types

> *To know that A is a type is to know how to form the canonical elements in the type and under what conditions two canonical elements are equal.*

So we allow constructors to also construct elements of $Id_D\ x\ y$.

Example: The circle

The circle S^1 is defined by two constructors:

- $base : S^1$
- $loop : Id_{S^1}\ base\ base$

To define a function $f : S^1 \to B$, it is sufficient to give $b : B$ and $p : Id_B\ base\ base$.

The circle refutes UIP

There is no homotopy between *loop* and *refl*.

So the circle S^1 is a counterexample to UIP!

Example: The interval

The interval I is defined by:

- $0 : I$
- $1 : I$
- $seg : Id_I\ 0\ 1$

To define a function $f : I \to B$, it is sufficient to give $b_0, b_1 : B$ and $p : Id_B\ b_0\ b_1$

The interval implies functional extensionality

For $f, g : A \to B$, we have a type of homotopies:

$$f \sim g = (x : A) \to Id_{B\ x}\ (f\ x)\ (g\ x)$$

Functional extensionality states:

$$funext : f \sim g \to Id_{A \to B}\ f\ g$$

This is implied by the existence of the interval I!

Example: Set quotients

Given a type A and a family $R\ x\ y$ over $x, y : A$, we define the set quotient A/R by:

- $q : A \to A/R$
- $e : (x\ y : A) \to R\ x\ y$
  $\to Id_{A/R}\ (q\ x)\ (q\ y)$
- $t : (x\ y : A/R)(p\ q : Id_{A/R}\ x\ y)$
  $\to Id_{(Id_{A/R}\ x\ y)}\ p\ q$

Example of a quotient type: Bags are lists modulo permutation

1. Define a type *List* of lists
2. Define a type *Perm* of permutations
3. Define a type *Bag* with two constructors

   $$makeBag : List \to Bag$$
   $$permute : (b : Bag)(\sigma : Perm) \to Id_{Bag}\ (\sigma\ b)\ b$$

Other examples

- n-dimensional spheres
- Suspensions
- CW-complexes
- Pushouts
- Truncations

Computation on higher paths

Open question:

How to define an induction principle for general higher inductive types?

What's the fuss about Homotopy Type Theory?

The Universe

A universe is a type whose terms are types.

We fix a universe U of small types

$\quad$ $Bool$, $\mathbb{N}$, $(n : \mathbb{N}) \to Vec\ n$, S^1, $\ldots$

$\quad$ But not U itself!

What is a term of type $Id_U\ A\ B$?
i.e.: When can two types be identified?

Equivalence of types: $A \simeq B$

A left inverse of $f : A \to B$ is a function $g : B \to A$ such that $g \circ f \sim id_B$ (similar for a right inverse).

A function $f : A \to B$ is an equivalence if it has both a left inverse and a right inverse.

In this case, A and B are called equivalent.

The univalence axiom

The univalence axiom specifies that equivalent types can be identified.

Define `id-to-equiv` : $Id_U\ A\ B \to A \simeq B$ by path induction.

The univalence axiom states that `id-to-equiv` has a left and right inverse

$$ua : A \simeq B \to Id_U\ A\ B$$

Application of univalence: Code reuse

One representation of a dictionary:

$$Dict = [(Key, Value)]$$

Another representation:

$$Dict' = BBT\ Key\ Value$$

By univalence, an isomorphism $Dict \simeq Dict'$ transports operations on $Dict$ to $Dict'$.

Can we give a constructive meaning to univalence?

There are no evaluation rules for `ua` x:
such terms are stuck.

Open question:
Can we give computational meaning to `ua`?

What's the fuss about Homotopy Type Theory?

1. The homotopy point of view
2. Higher inductive types
3. The univalence axiom
4. Application to generic programming

Abstract types in type theory

Let *Seq A* be an abstract type with interface:

$$
\begin{array}{lcl}
\texttt{empty} & : & Seq\ A \\
\texttt{single} & : & A \to Seq\ A \\
\texttt{append} & : & Seq\ A \to Seq\ A \to Seq\ A \\
\texttt{map} & : & (A \to B) \to Seq\ A \to Seq\ B \\
\texttt{reduce} & : & (A \to B \to B) \to B \to Seq\ A \to B
\end{array}
$$

How do we know that e.g.

$$\texttt{map}\ f\ (\texttt{single}\ x) = \texttt{single}\ (f\ x)?$$

We don't!

A solution: view types

Add this to the interface:

$$
\begin{aligned}
&\texttt{toList} &&: \; \mathit{Seq}\ A \to [A] \\
&\texttt{fromList} &&: \; [A] \to \mathit{Seq}\ A \\
&\texttt{empty-spec} &&: \; \mathit{Id}_{[A]}\ (\texttt{toList empty})\ [] \\
&\texttt{single-spec} &&: \; \mathit{Id}_{[A]}\ (\texttt{toList}\ (\texttt{single}\ x))\ [x] \\
&\texttt{append-spec} &&: \; \ldots \\
&\ldots
\end{aligned}
$$

But this is boring and hard to work with . . .

Specifying view types in HoTT

The interface only needs one field:

$$spec : Seq\ A \simeq [A]$$

By univalence, this allows us to replace $Seq\ A$ by $[A]$ in all our proofs!

In particular, we can extract the old interface.

Conclusion

- HoTT is being pushed as a new foundation for mathematics
- It already has some interesting applications for programming
- More importantly, it gives us a new way of thinking about identity in type theory
- It shows a lot of promise for the future

References

- The HoTT book (`homotopytypetheory.org/book/`).
- The HoTT blog (`homotopytypetheory.org/blog/`).
- The n-Category Café (`golem.ph.utexas.edu/category/`).
- Martin Hofmann and Thomas Streicher, *The groupoid model refutes uniqueness of identity proofs* (1994).
- Mike Shulman, *A tentative proposal for a general syntax of higher inductive types* (2012).
- Dan Licata and Robert Harper, *Canonicity for 2-dimensional type theory* (draft).
- Dan Licata, *Programming in Homotopy Type Theory* (talk at IAS).