What is. . . type theory?

An introduction for mathematicians

Jesper Cockx

21 February 2017

What is a type system?

A type system is a set of rules that assign a property called type to various constructs a computer program consists of.

The main purpose of a type system is to reduce possibilities for bugs in computer programs.

(paraphrased from Wikipedia)

Type theory is…

Surprisingly theoretical:

- deep links with category theory and homotopy theory
- alternative foundation of mathematics

Type theory is...

Surprisingly theoretical:

- deep links with category theory and homotopy theory
- alternative foundation of mathematics

Surprisingly practical:

- new programming languages
- proof assistants based on type theory

Goals of this talk

Introduce you to the world of type systems

Goals of this talk

Introduce you to the world of type systems

Show some beautiful ideas from type theory

Goals of this talk

Introduce you to the world of type systems

Show some beautiful ideas from type theory

Explain what types can do for you as a mathematician

Type systems

Dependent types

Homotopy type theory

Type systems

Dependent types

Homotopy type theory

Types in programming languages

Goal: give meaning to computer programs without having to run them.

Types in programming languages

Goal: give meaning to computer programs without having to run them.

- Data types in C: `int x = 5;`

Types in programming languages

Goal: give meaning to computer programs without having to run them.

- Data types in C: `int x = 5;`
- Classes in C++ and Java:
  `class Triangle extends Shape`

Types in programming languages

Goal: give meaning to computer programs without having to run them.

- Data types in C: `int x = 5;`
- Classes in C++ and Java:
  `class Triangle extends Shape`
- Polymorphic function types in Haskell:
  $\texttt{zip} : [a] \to [b] \to [(a, b)]$

Static versus dynamic typing

Statically typed languages guarantee no type errors will occur while running a program.

e.g. C, Java, Haskell, ...

Static versus dynamic typing

Statically typed languages guarantee no type errors will occur while running a program.

e.g. C, Java, Haskell, ...

Dynamically typed languages don't give any such guarantees.

e.g. JavaScript, Python, MATLAB, ...

Syntax of type theory

$$a : T$$

"The term a has type T."

Simply typed lambda calculus

Function types: $A \to B$

Alonzo
Church

Simply typed lambda calculus

Function types: $A \rightarrow B$

Function application:

$$\frac{f : A \rightarrow B \qquad x : A}{f\ x : B}$$

Alonzo
Church

Simply typed lambda calculus

Function types: $A \to B$

Function application:

$$\frac{f : A \to B \qquad x : A}{f\ x : B}$$

Lambda abstraction:

$$\frac{x : A \vdash t : B}{\lambda x.\, t : A \to B}$$

Alonzo Church

The Curry-Howard correspondence

Haskell B. Curry

If we read a type $A \to B$ as an implication $A \Rightarrow B$, any given function corresponds to a proof of the implication and vice versa.

The Curry-Howard correspondence

Logic		Type system
proposition	A	type
proof	$a : A$	program
implication	$A \to B$	function type
conjunction	$A \times B$	pair type
disjunction	$A \uplus B$	sum type
true	$\top$	unit type
false	$\bot$	empty type

Constructive logic

Type theory is a *constructive* logic.

We can *run* a proof of "A or B" to get either a proof of A or a proof of B.

Luitzen E. J. Brouwer

Constructive logic

Type theory is a *constructive* logic.

We can *run* a proof of "A or B" to get either a proof of A or a proof of B.

In particular, the law of the excluded middle doesn't hold!

Luitzen E. J. Brouwer

The cake question

"Do you want the cake or can I have it?"

Classical mathematician: *"Yes!"*

Curry-Howard beyond propositional logic

Ideas from logic can be used in type systems, and vice versa:

- Modal logic / monads (Haskell)
- Linear logic / linear types (Rust)
- Classical logic / CPS (Scheme)
- Predicate logic / dependent types

Type systems

Dependent types

Homotopy type theory

Dependent types

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

E.g. `Vec` $A\ n$ is the type of vectors of length n.

Per Martin-Löf

Martin-Löf type theory

Logic		Type system
Universal quantification	$(x : A) \to B\ x$	Dependent function type
Existential quantification	$(x : A) \times B\ x$	Dependent pair type
Equality	$x \equiv_A y$	Identity type

Example: $(x : \mathbb{N}) \to (y : \mathbb{N}) \times (2 * x \equiv_{\mathbb{N}} y)$

Proof assistants

Thierry Coquand

Type theory provides a full alternative to both logic and set theory.

We can build proof assistants based on type theory! e.g. Coq, Agda, NuPRL

What can you do with a proof assistant?

- Check that your proofs are correct

What can you do with a proof assistant?

- Check that your proofs are correct
- Look at the current assumptions and goal while writing a proof

What can you do with a proof assistant?

- Check that your proofs are correct
- Look at the current assumptions and goal while writing a proof
- Write scripts that search a proof automatically (*tactics*)

Why no proof assistants based on set theory?

- Type membership is decidable, set membership is not.

Why no proof assistants based on set theory?

- Type membership is decidable, set membership is not.
- Terms in type theory can be *run* to simplify them.

Why no proof assistants based on set theory?

- Type membership is decidable, set membership is not.
- Terms in type theory can be *run* to simplify them.
- Set theory has only one type: *Set*.

Inductive families of datatypes

You can define new data types, for example:

$$
\begin{aligned}
&\textbf{data}\ \mathbb{N} : \texttt{Type}\ \textbf{where} \\
&\quad \texttt{z} : \mathbb{N} \\
&\quad \texttt{s} : \mathbb{N} \to \mathbb{N}
\end{aligned}
$$

Inductive families of datatypes

You can define new data types, for example:

$$
\begin{aligned}
&\textbf{data}\ \mathbb{N} : \mathtt{Type}\ \textbf{where} \\
&\quad \mathtt{z} : \mathbb{N} \\
&\quad \mathtt{s} : \mathbb{N} \to \mathbb{N}
\end{aligned}
$$

You can also define predicates and relations:

$$
\begin{aligned}
&\textbf{data}\ \_ \leq \_ : \mathbb{N} \to \mathbb{N} \to \mathtt{Type}\ \textbf{where} \\
&\quad \mathtt{lz} : (n : \mathbb{N}) \to \mathtt{z} \leq n \\
&\quad \mathtt{ls} : (m\ n : \mathbb{N}) \to m \leq n \to \mathtt{s}\ m \leq \mathtt{s}\ n
\end{aligned}
$$

Dependent pattern matching

You can define functions by pattern matching:

$$
\begin{aligned}
&\_ + \_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathtt{z} + y \qquad = y \\
&(\mathtt{s}\ x) + y = \mathtt{s}\ (x + y)
\end{aligned}
$$

Dependent pattern matching

You can define functions by pattern matching:

$$
\begin{aligned}
&\_ + \_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathtt{z} + y \quad\quad\;\; = y \\
&(\mathtt{s}\ x) + y = \mathtt{s}\ (x + y)
\end{aligned}
$$

You can also prove things by pattern matching:

$$
\begin{aligned}
&\mathtt{assoc} : (x : \mathbb{N}) \to (y : \mathbb{N}) \to (z : \mathbb{N}) \to \\
&\qquad\qquad x + (y + z) \equiv_{\mathbb{N}} (x + y) + z \\
&\mathtt{assoc}\ \mathtt{z}\ y\ z \quad\quad\; = \mathtt{refl} \\
&\mathtt{assoc}\ (\mathtt{s}\ x)\ y\ z = \mathtt{cong}\ \mathtt{s}\ (\mathtt{assoc}\ x\ y\ z)
\end{aligned}
$$

Type systems

Dependent types

Homotopy type theory

Open questions about Martin-Löf's type theory

What is the structure of the identity type?

Open questions about Martin-Löf's type theory

What is the structure of the identity type?

What is the equivalent of subsets and set quotient in type theory?

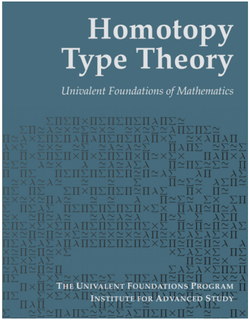
Open questions about Martin-Löf's type theory

What is the structure of the identity type?

What is the equivalent of subsets and set quotient in type theory?

How to do reasoning 'up to isomorphism'?

Enter homotopy type theory



The HoTT book.

We can interpret types as topological spaces:

- Elements $x : A$ are points of the space A.
- Functions $A \to B$ are continuous maps from A to B.
- Proofs of $x \equiv_A y$ are *paths* from x to y.

Higher inductive types

You can define new types by giving their points *and paths between their points*:

$$
\begin{array}{l}
\textbf{data}\ \mathtt{I} : \mathtt{Type}\ \textbf{where} \\
\quad \mathtt{start} \quad\ \ : \mathtt{I} \\
\quad \mathtt{end} \qquad\ \ : \mathtt{I} \\
\quad \mathtt{segment} : \mathtt{start} \equiv_{\mathtt{I}} \mathtt{end}
\end{array}
$$

This way we can define quotients and subsets.

Synthetic homotopy theory

We can define topological spaces inductively:

$$
\begin{array}{l}
\textbf{data}\ \mathtt{S}^1 : \mathtt{Type}\ \textbf{where} \\
\quad \mathtt{base} : \mathtt{S}^1 \\
\quad \mathtt{loop} : \mathtt{base} \equiv_{\mathtt{S}^1} \mathtt{base}
\end{array}
$$

Synthetic homotopy theory

We can define topological spaces inductively:

```
data S¹ : Type where
  base : S¹
  loop : base ≡_S¹ base
```

Now we can do synthetic homotopy theory!

Synthetic homotopy theory

We can define topological spaces inductively:

data $\mathtt{S}^1$: `Type` **where**
  `base` : $\mathtt{S}^1$
  `loop` : `base` $\equiv_{\mathtt{S}^1}$ `base`

Now we can do synthetic homotopy theory!

We can prove that (`base` $\equiv_{\mathtt{S}^1}$ `base`) $\simeq \mathbb{Z}$.

Equivalence of types

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

Two types are equivalent ($A \simeq B$) if there is an equivalence between them.

The univalence axiom

$$(A \equiv_{\texttt{Type}} B) \simeq (A \simeq B)$$

"Isomorphic structures can be identified"

Vladimir Voevodsky

Cubical type theory

Problem: Univalence is an *axiom*, i.e. a program that doesn't compute.

Cubical type theory

Problem: Univalence is an *axiom*, i.e. a program that doesn't compute.

Very recently (2016), this has been solved by cubical type theory, based on ideas from cubical sets in category theory.

Cubical type theory

Problem: Univalence is an *axiom*, i.e. a program that doesn't compute.

Very recently (2016), this has been solved by cubical type theory, based on ideas from cubical sets in category theory.

Now we can actually run univalence to transport constructions between isomorphic structures!

Pattern matching in HoTT

Problem: With pattern matching, we can prove that any two paths are equal:

$$\mathtt{UIP} : (p\ q : x \equiv_A y) \to p \equiv_{x \equiv_A y} q$$
$$\mathtt{UIP}\ \mathtt{refl}\ \mathtt{refl} = \mathtt{refl}$$

Pattern matching in HoTT

Problem: With pattern matching, we can prove that any two paths are equal:

$$\mathtt{UIP} : (p\ q : x \equiv_A y) \to p \equiv_{x \equiv_A y} q$$
$$\mathtt{UIP}\ \mathtt{refl}\ \mathtt{refl} = \mathtt{refl}$$

My PhD: how to make dependent pattern matching and HoTT work together?

Conclusion

Type theory is both surprisingly theoretical and surprisingly practical.

Conclusion

Type theory is both surprisingly theoretical and surprisingly practical.

Proof assistants based on type theory can make your life as a mathematician easier.

Conclusion

Type theory is both surprisingly theoretical and surprisingly practical.

Proof assistants based on type theory can make your life as a mathematician easier.

Lots of exciting things are happening right now, I've only scratched the surface!

References

Howard: *The formulae-as-types notion of construction* (1969).

Martin-Löf: *An intuitionistic theory of types* (1972).

Hofmann and Streicher: *The groupoid model refutes uniqueness of identity proofs* (1994).

Licata and Shulman: *Calculating the fundamental group of the circle in homotopy type theory* (2013).

Cohen, Coquand, Huber and Mörtberg: *Cubical type theory: a constructive interpretation of the univalence axiom* (2016).

Links

The HoTT book:

`homotopytypetheory.org/book`

The Coq proof assistant:

`coq.inria.fr`

The Agda proof assistant:

`wiki.portal.chalmers.se/agda`