Implementing Type Theory

Daniel Gratzer¹  Jonathan Sterling²  Lars Birkedal¹

May 24, 2019

¹This University 😊
²Not This University 😊
Languages classify expressions into different types (\texttt{int}, \texttt{string}, \texttt{char}).

**Type System**  The rules for what expressions belong to which types.

**Type-Checker**  The program that makes sure we follow the rules.
Setting the Scene

What is type theory? Type theory is a:

- programming language with a rich type system.
- framework for reasoning about mathematical objects.
What is type theory? Type theory is a...

- programming language with a rich type system.
- framework for reasoning about mathematical objects.
Set aside the questions of *mathematics* and *programming* for a second.

Type theory has functions

**Example**

```
useful_function(important_argument)
```

When is this application well-typed?
Set aside the questions of mathematics and programming for a second.

Type theory has functions

Example

```
useful_function(important_argument)
```

must have type

\[ A \rightarrow B \]
Set aside the questions of mathematics and programming for a second.

Type theory has functions

**Example**

useful_function(important_argument)

must have type $A \rightarrow B$

must have type $C$
Set aside the questions of mathematics and programming for a second.

Type theory has functions

**Example**

```
useful_function(important_argument)
```

must have type  

$A \rightarrow B$

We must also have  

$A = C$

must have type  

$C$
How Hard is Type-Checking?

What should we take away from this example?

1. In order to type-check, we must check if two types are equal.
2. So we need a program checking type equality.
Deciding type equality is always a problem but we have fancier types:

\[
\text{Vec}(A, n) \quad \text{A list of } A \text{s of length } n
\]
Deciding type equality is always a problem but we have fancier types:

\[
\text{Vec}(A, n) \quad \text{A list of As of length } n
\]

We need more than type equality... we need term equality too!

\[
\text{Vec}(A, 2 \times n) \equiv \text{Vec}(A, n + n)
\]
In order to implement type theory we must check the equality of terms.

1. This is completely impossible in a Turing-complete language\(^1\).
2. Actually it’s impossible in many Turing-\textit{incomplete} languages as well.
3. Many equalities we expect are impossible to automatically check:

\[ f = g \iff \text{for all } x, \ f(x) = g(x) \]

---

\(^1\)Python, Java, C, C++, PostScript, and Magic the Gathering are all Turing-complete
The central balancing act is then defining an equality relation which is

- strong enough to match our mathematical intuitions.
- simple enough that we can implement it.
We designed a theory of equality for a particular modal type theory.

- The type theory was mathematically motivated.
- But it is still interesting for programming.

In both cases, having an implementation was important!
The Process:\n
1. Write down the rules of the type system. \hfill (2 pages)\n2. Prove the decidability of type-checking. \hfill (90 pages)\n3. Implement the type-checker. \hfill (300 lines)\n
See our paper: \url{https://jozefg.github.io/modal.pdf}

\footnote{Elided: the coffee & false starts, or where I get distracted by random Wikipedia articles.}
I cut out a lot of cool stuff in this talk:

- Using type theory, we can “run” math proofs.
- We can use computer science to explore mathematics.
- We can use maths to inspire better PLs.

Many unexplored and interesting questions remain...
The LogSem Group

If this sounds interesting, please come talk to us!

[Images of group members]