Implementing Type Theory

Daniel Gratzer[1] Jonathan Sterling[2] Lars Birkedal[1]

May 24, 2019

[1]This University ☺
[2]Not This University ☹

Some Terminology

Languages classify expressions into different *types* (`int`, `string`, `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

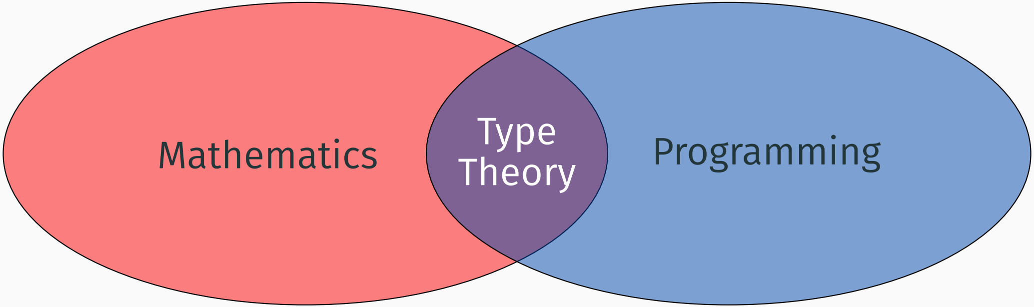
What is type theory? Type theory is a....

- programming language with a rich type system.
- framework for reasoning about mathematical objects.

Setting the Scene

What is type theory? Type theory is a....

- programming language with a rich type system.
- framework for reasoning about mathematical objects.



A Concrete Issue

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?

A Concrete Issue

Set aside the questions of mathematics and programming for a second.

Type theory has functions

Example

```
useful_function(important_argument)
```

must have type $A \longrightarrow B$

A Concrete Issue

Set aside the questions of mathematics and programming for a second.

Type theory has functions

Example

```
useful_function(important_argument)
```

must have type $A \longrightarrow B$

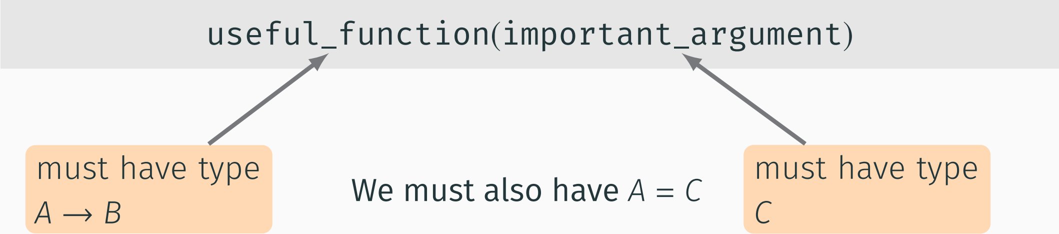
must have type C

A Concrete Issue

Set aside the questions of mathematics and programming for a second.

Type theory has functions

Example



```
useful_function(important_argument)
```

must have type $A \longrightarrow B$

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.

Just Type Equality?

Deciding type equality is always a problem but we have fancier types:

$\mathsf{Vec}(A, n)$	A list of As of length n

Just Type Equality?

Deciding type equality is always a problem but we have fancier types:

$\mathsf{Vec}(\mathsf{A}, \mathsf{n})$ A list of As of length n

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

$$\mathsf{Vec}(\mathsf{A}, 2 * \mathsf{n}) \overset{?}{=} \mathsf{Vec}(\mathsf{A}, \mathsf{n} + \mathsf{n})$$

The Mess We're In

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-*incomplete* languages as well.
3. Many equalities we expect are impossible to automatically check:

$$\mathtt{f} = \mathtt{g} \iff \text{for all } \mathtt{x},\ \mathtt{f}(\mathtt{x}) = \mathtt{g}(\mathtt{x})$$

---

[1] Python, Java, C, C++, PostScript, and Magic the Gathering are all Turing-complete

Modern Type Theory

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.

Our Work

We designed a theory of equality for a particular *modal* type theory.

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

In both cases, having an implementation was important!

Implementing Modal Type Theory

The Process[2]:

1. Write down the rules of the type system. (2 pages)
2. Prove the decidability of type-checking. (90 pages)
3. Implement the type-checker. (300 lines)

See our paper: `https://jozefg.github.io/modal.pdf`

---

[2] Elided: the coffee & false starts, or where I get distracted by random Wikipedia articles.

Conclusions (Some of the Stuff I Skipped)

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!