Why Type Theory Matters

λ Days 2019 in Krakow

Thorsten Altenkirch

Functional Programming Laboratory
School of Computer Science
University of Nottingham

February 22, 2019

My programming languages CV

1975 - 1982	BASIC *Bettina-von-Arnim Oberschule*
1980 - 1983	Z80 Assembler C *Nixdorf Microprocessor Engineering*
1983- 1989	Common Lisp Scheme *INPRO, Fraunhofer Institute, Expertise GmbH*
1986 - 1995	ML *Technical University of Berlin*
1989 - now	Type Theory LEGO, ALF, Agda *Universities of Edinburgh, Munich, Gothenburg and Nottingham*
2013 - now	Homotopy Type Theory (HoTT) cubical, cubical Agda

Overview

Well typed programs don't go wrong . . .

```
(!!) :: [a] -> Int -> a

> [1,2,3] !! 4
*** Exception: !!: index too large
```

Lookup in agda

```
_!!_ : Vec A n → Fin n → A

[] !! ()
(x :: as) !! zero = x
(x :: as) !! suc i = as !! i

(1 :: 2 :: 3 :: []) !! 4
```

is not well typed.

What is the type of add ?

```
x : ℕ
x = add 2 3 4 5
-- evaluates to 14

y : ℕ
y = add 1 2
-- evaluates to 3
```

It depends ...

```
NAdd : ℕ → Set
NAdd 0 = ℕ
NAdd (suc n) = ℕ → NAdd n

nadd : {n : ℕ} → ℕ → NAdd n
nadd {0} s = s
nadd {suc n} s i = nadd {n} (i + s)

add : {n : ℕ} → NAdd n
add {n} = nadd {n} 0
```

Let's do some logic . . .

$$\mathtt{P} \land (\mathtt{Q} \lor \mathtt{R}) \leftrightarrow \mathtt{P} \land \mathtt{Q} \lor \mathtt{P} \land \mathtt{R}$$

- Is this a tautology?
- How do we know?
- Use a truth table?

Write a program!

```
P ∧ Q = P × Q
P ∨ Q = P ⊎ Q
P ↔ Q = (P → Q) ∧ (Q → P)

distr : P ∧ (Q ∨ R) ↔ P ∧ Q ∨ P ∧ R

proj₁ distr (p , inj₁ q) = inj₁ (p , q)
proj₁ distr (p , inj₂ r) = inj₂ (p , r)
proj₂ distr (inj₁ (p , q)) = p , inj₁ q
proj₂ distr (inj₂ (p , r)) = p , inj₂ r
```

Classical vs Intuitionistic

Classical *A proposition is something that is either true or false.*

`Prop = Bool`

Intuitionistic *A proposition is something for which we can have evidence*

`Prop = Set`

Propositions as types

The classical lie

We can implement classical propositional logic using Bool:

```
_&&_ : Bool → Bool → Bool
false && c = false
true  && c = c

_||_ : Bool → Bool → Bool
false || c = c
true  || c = true
```

but what about predicate logic?

```
all : {A : Set}(P : A → Bool) → Bool
```

Propositions as types

```
all : {A : Set}(P : A → Set) → Set
all {A} P = (x : A) → P x

all∧ :
     ((x : A) → P x ∧ Q x)
  ↔ ((x : A) → P x) ∧ ((x : A) → Q x)

proj₁ all∧ f         = (λ x → proj₁ (f x)) ,
                        (λ x → proj₂ (f x))
proj₂ all∧ (g , h) = λ x → (g x) , (h x)
```

Why should we care?

- We can express any logical condition just using the type system.
- In practice proofs and programs are not clearly separated.

```
record Sort (inp : List ℕ) : Set where
  field
    out : List ℕ
    sorted : Sorted out
    perm : inp ≅ out

sort : (inp : List ℕ) → Sort inp
```

Totalitarianism

- Programs that hang are buggy.
- We only deal with programs that are total.
- We don't reason about programs that have syntax errors either.
- Programs that produce infinite structures are fine. They need to be productive.
- However, it can be hard work to convince a compiler that your program is total.
- You can work with programs that are not explicitly total by saying *trust me*.
- However, there are some places where you shouldn't do that.

Where we need to be total

Coercions

```
coe : {A B : Set} → A ≡ B → (A → B)
coe refl = λ x → x
```

- We never need to evaluate the proof of $A \equiv B$.
- But we need to know that it terminates.
- Otherwise type soundness would fail.

Certificates

- If the component `Sorted outp` is partial, then `sort` could produce output list that is not sorted.

Hiding of implementation details

- Consider two implementations of the natural numbers:

```
data ℕ₁ : Set where
  1   : ℕ₁
  1+_ : ℕ₁ → ℕ₁

data ℕ₂ : Set where
  1      : ℕ₂
  2×_    : ℕ₂ → ℕ₂
  1+2×_ : ℕ₂ → ℕ₂
```

- There is no predicate that can distinguish them.
- We can consistently replace one with the other.
- That is not true for set theory!
- Type Theory supports hiding of implementation details.
- However, Intensional Type Theory doesn't identify them.

Univalence in HoTT

- We can define equivalence of types:

```
record _≅_ (A B : Set) : Set where
  field
    f : A → B
    g : B → A
    gf : (a : A) → g (f a) ≡ a
    fg : (b : B) → f (g b) ≡ b
    coh : ...
```

- In particular we can show:

```
equiv : ℕ₁ ≅ ℕ₂
```

- The univalence principle states that equivalence is equivalent to equality:

```
unival : (A ≅ B) ≅ (A ≡ B)
```

- Hence we can derive:

```
equal : ℕ₁ ≡ ℕ₂
```

Higher inductive types in HoTT

- Every type comes with its own equality.
- Hence in datatype definitions we can also have constructors for equality.
- For example we can define the integers:

```
data ℤ : Set where
  0 : ℤ
  _+1 : ℤ → ℤ
  _-1 : ℤ → ℤ
  +- : ∀ i → (i +1) -1 ≡ i
  -+ : ∀ i → (i -1) +1 ≡ i
  coh : ...
```

Cubical type theory

- Voevodsky introduced HoTT based on ideas from homotopy theory.
- He constructed a mathematical model using *simplicial sets*.
- However, he used classical logic ...
- As a consequence it wasn't clear how to **run** definitions in HoTT in general.
- Later this issue was fixed by Thierry Coquand and his team.
- They provided a constructive semantics using *cubical sets*.
- This was also implemented in a proof of concept system called `cubical`.
- There is now a prototypical implementation of cubical agda available.

Take home messages

- If you never liked Maths but you like functional programming here is a new chance because **Maths is just functional programming!**
- If you like Maths anyway, here is a new Maths which is much better than the old one, **because** it is based on functional programming.
- If you don't care about Maths either way, here is a way to write programs that say what they do on the tin (i.e. in the type) and **the compiler will check that you are not lying.**
- If you are thinking that the tool chain isn't very good yet, you are probably right. **Help us to change this!**