Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 11
Dependent types

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Dependent types

The final frontier?

- **How do we know a program is correct?**
  - **We might write it, then verify the correctness**
  - **Or, we might prove that a solution exists and then extract a program from that.**
  - **Or, we might write the program in a language that allows us to state properties**

Dependent types

Types matter to us because they let us check (automatically) properties of programs.

```
map :: (a -> b) -> [a] -> [b]
```

The richer the type structure the more one can say in those properties

- **So, the more flexibility one has to write programs while preventing errors**
- **Types represent static guarantees**

Dependent types

The Simply Typed Lambda Calculus parallels the deduction rules of zeroth-order logic

- **So we could say types correspond to propositions**
- **Terms correspond to proofs**
- **In the Hindley-Milner type system we have more:**
- **We can abstract over propositions (polymorphism) and introduce axiom schemes (parameterised datatypes)**

Dependent types

Theorem proving in types

If we take this view then:

- Encode our propositions as types
- If the program type checks (i.e. if the checker accepts our definition “f :: T” then we can conclude that “T” is proven)

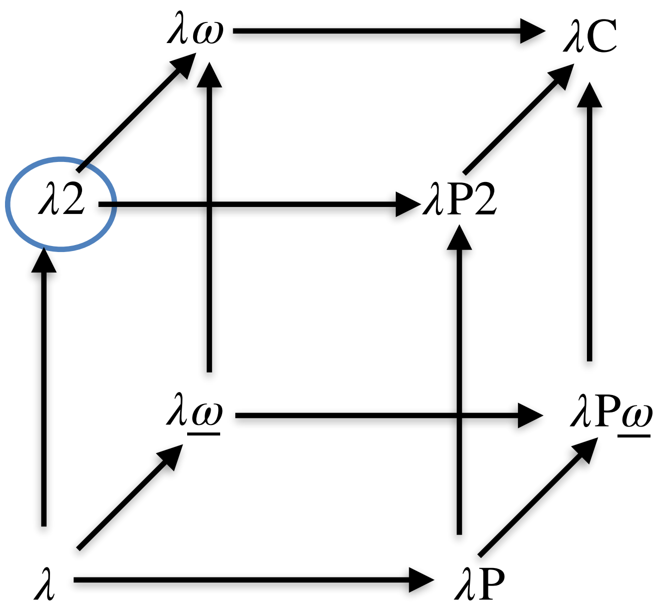
However, we can't express predicates, our logic is not expressive enough

Dependent types

Theorem proving in types

- **Terms can depend on Terms**
  Function definitions
- **Types can depend on Types**
  Parameterised data types
- **Terms can depend on Types**
  Polymorphic terms

- **The missing part of the grid is a way for Types to depend on Terms**
  This is a *dependent* type system



Barendregt's Lambda Cube

Dependent types

Theorem proving in types

Examples of systems that offer dependent typing:

- **Agda**
- **Coq**
- **Idris**

We will take a brief look at Agda and Idris which have some familiarity for functional programmers.

Dependent types

Agda

Agda is

- **"a dependently typed functional programming language"**
- **Haskell-like, but with many important differences too**
- **"a proof assistant"**
- **A tool for writing and checking constructive proofs.**
- **Today we will give a very brief taste of Agda, to demonstrate dependent types**

Dependent types

Agda

The usual example in dependent types is the vector (list) which tracks length.

```
data ℕ: Set where
  zero : ℕ
  suc  : ℕ  → ℕ
```

```
data Vec (A : Set) : ℕ → Set where
  []    : Vec A zero
  _::_  : ∀ {n} → A → Vec A n → Vec A (suc n)
```

We can use this to create, for example, a definition of "map" that records the fact that length is invariant under mapping.

```
map : ∀ {A B n} → (A → B) → Vec A n → Vec B n
map f []        = []
map f (x :: xs) = f x :: map f
```

Dependent types

Agda

We could get that far using GADTs. We needed type families to define this:

```
_+_ : ℕ → ℕ  → ℕ
zero + n = n
(suc n) + m = suc (n + m)
```

Which gives us list concatenation:

```
_++_ : ∀ {A n m} → Vec A n → Vec A m → Vec A (n + m)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Dependent types

Agda

That's not all we can do. Here's a solution to the “printf” problem we saw described in the “Functional Unparsing" paper.

```
data ValidFormat : Set₁ where
   argument : (A : Set) → (A → String) → ValidFormat
   literal  : Char        → ValidFormat
```

```
data Format : Set₁ where
   valid   : List ValidFormat → Format
   invalid : Format
```

Dependent types

Agda

We can convert a string containing a format string into a "Format" value using the "parse" function. This will produce a *valid* format for any correct format string, and an *invalid* format otherwise

```
parse : String → Format
parse s = parse' [] (toList s)
  where
    parse' : List ValidFormat → List Char → Format
    parse' l ('%' :: 's' :: fmt) = parse' (argument String (λ x → x) :: l) fmt
    parse' l ('%' :: 'c' :: fmt) = parse' (argument Char (λ x → fromList [ x ]) :: l) fmt
    parse' l ('%' :: 'd' :: fmt) = parse' (argument ℕ showNat :: l) fmt
    parse' l ('%' :: '%' :: fmt) = parse' (literal '%' :: l) fmt
    parse' l ('%' :: c   :: fmt) = invalid
    parse' l (c          :: fmt) = parse' (literal c :: l) fmt
    parse' l []                  = valid (reverse l)
```

Dependent types

Agda

A function that converts a format string to a *type*:

```
Args : Format → Set
Args invalid                          = ⊥ → String
Args (valid (argument t _ :: r)) = t → (Args (valid r))
Args (valid (literal _ :: r))     = Args (valid r)
Args (valid [])                       = String
```

```
FormatArgs : String → Set
FormatArgs f = Args (parse f)
```

Dependent types

Agda

The actual “printf” function. Note that the *type* is computed from the input parameter f

```
sprintf : (f : String) → FormatArgs f
sprintf = sprintf' "" ∘ parse
  where
    sprintf' : String → (f : Format) → Args f
    sprintf' accum invalid                      = λ t → ""
    sprintf' accum (valid [])                   = accum
    sprintf' accum (valid (argument _ s :: l)) = λ t →  (sprintf' (accum ++ s t) (valid l))
    sprintf' accum (valid (literal c :: l))    = sprintf' (accum ++ fromList [ c ]) (valid l)
```

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 1

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/