The λ Cube

@zerokarmaleft

#tulsalambdalounge

About Me

- Senior Software Engineer
  at Laureate Institute for Brain Research

- Language Nut (all shapes and sizes)

References

- Barendregt, Henk. “Introduction to generalized type systems.”
- Pierce, Benjamin. Types and Programming Languages.
- Bird, Richard. Introduction to Functional Programming in Haskell.
- Haskell Wiki. http://www.haskell.org/haskellwiki
- Harrah, Mark. “Type-level Programming in Scala.” http://apocalisp.wordpress.com/2010/06/08/type-level-programming-in-scala

Sample Code

https://github.com/zerokarmaleft/the-lambda-cube-presentation

Agenda

- Foundation
  - untyped lambda calculus
  - simply-typed lambda calculus
- Explore extensions to simply-typed lambda calculus from Barendregt's Cube (λ Cube)

Goals

- Re-examine preconceived notions of type systems
- Establish a jump-off point to other interesting topics
- A healthy dose of functional brain candy!

Anti-goals

- Don’t use a metric crapton of Γρεεκ letters
- Avoid arguing statically-typed languages vs. dynamically-typed languages
- Don’t use the “m” word

What do types give me?

- Detection of errors
- Abstraction
- Documentation
- Language safety
- Efficiency

Types describe data

```haskell
data Person = Person
  { firstName :: String
  , lastName  :: String
  , address   :: Address
  }

data Address = Address
  { street1 :: String
  , street2 :: String
  , city    :: String
  , state   :: String
  , zip     :: String
  }
```

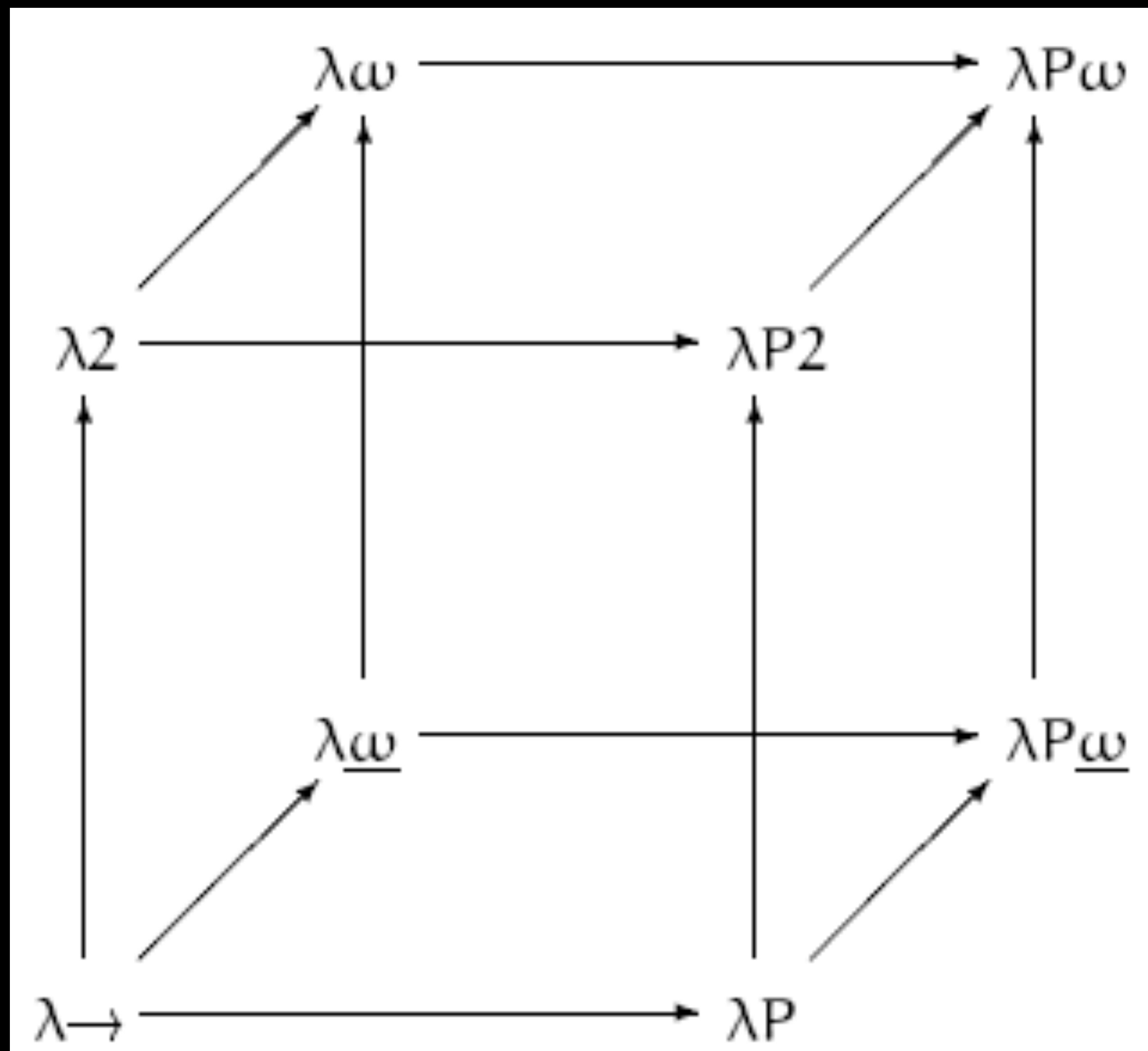
Types describe behavior

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

filter :: (a -> Bool) -> [a] -> [a]

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

Berendregt's Cube



untyped lambda calculus (λ)

- calculus - a system of computation in a special notation
- lambda (λ) - a function abstraction which can be applied to carry out computation
- untyped - without type verification

λ

- `x` - variables
- `λx.t` - abstractions
- `t t` - applications

λ

```
(def true  (fn [t] (fn [f] t)))

(def false (fn [t] (fn [f] f)))
```

λ

```
(def test
  (fn [b]
    (fn [v]
      (fn [w] ((b v) w)))))

(def and
  (fn [b1]
    (fn [b2]
      ((b1 b2) false))))

(def or
  (fn [b1]
    (fn [b2]
      ((b1 true) b2))))

(def not
  (fn [b]
    ((b false) b)))
```

λ

```
(def 0 (fn [s] (fn [z] z)))
(def 1 (fn [s] (fn [z] (s z))))
(def 2 (fn [s] (fn [z] (s (s z)))))
(def 3 (fn [s] (fn [z] (s (s (s z))))))
…

(def succ
  (fn [n]
    (fn [s]
      (fn [z]
        (s ((n s) z))))))
…
```

simply-typed lambda calculus ($\lambda_{\rightarrow}$)

- add simple typing relation to primitives (e.g. Booleans, numbers, etc.)
- add simple typing relation to functions (the $\rightarrow$ type)
- typing may be explicit or implicit

$\lambda_{\rightarrow}$

```
data Bool = True | False

data Nat = Zero | Succ Nat | Pred Nat
```

$\lambda_{\to}$

```
if (Pred Zero) then True else False

Succ True
```

λ→

```
isZero :: Nat -> Bool
```

System F ($\lambda 2$) polymorphism

- parametric
  - function defined over a range of types, with the same behavior for each type
- ad-hoc
  - function defined over several types, with different behavior for each type

System F
parametric polymorphism

```
doubleInt :: (Int -> Int) -> (Int -> Int)
doubleInt f = λx -> f (f x)
```

System F
parametric polymorphism

```
doubleBool :: (Bool -> Bool) -> (Bool -> Bool)
doubleBool f = λx -> f (f x)
```

System F
parametric polymorphism

```
doubleInt :: (Int -> Int) -> (Int -> Int)
doubleInt f = λx -> f (f x)

doubleBool :: (Bool -> Bool) -> (Bool -> Bool)
doubleBool f = λx -> f (f x)
```

System F
parametric polymorphism

```
doubleInt :: (Int -> Int) -> (Int -> Int)
doubleInt f = λx -> f (f x)

doubleBool :: (Bool -> Bool) -> (Bool -> Bool)
doubleBool f = λx -> f (f x)

double :: (a -> a) -> (a -> a)
double f = λx -> f (f x)
```

System F
parametric polymorphism

```
id :: a -> a

length :: [a] -> Int

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

System F
parametric polymorphism

```
type Cbool a = a -> a -> a

true, false :: Cbool a
true t f = t
false t f = f

testBool :: Cbool Bool -> Bool
testBool b = b True False
```

System F
parametric polymorphism

```
not :: Cbool (Cbool a) -> Cbool a
not b = b false true

and :: Cbool (Cbool a) -> Cbool a
and b1 b2 = b1 b2 true

or :: Cbool (Cbool a) -> Cbool a
or b1 b2 = b1 true b2
```

System F
parametric polymorphism

```
type Cnum a = (a -> a) -> (a -> a)

c0 :: Cnum a
c0 f = id

c1 :: Cnum a
c1 f = f

c2 :: Cnum a
c2 f = f . f
```

System F
parametric polymorphism

```
testNum :: Cnum Int -> Int
testNum n = n f 0

csucc :: Cnum a -> Cnum a
csucc n = λs -> s . n s
```

System F
parametric polymorphism

```
testNum :: Cnum Int -> Int
testNum n = n f 0

csucc :: Cnum a -> Cnum a
csucc n = λs -> s . n s
```

λ_ω ($\lambda_{\underline{\omega}}$)
what's the type?

```
λ > :t []
[] :: ???

λ > :t Just
Just :: ???

λ > :t Nothing
Nothing :: ???
```

λ_ω
what's the type?

```
λ > :t []
[] :: [a]

λ > :t Just
Just :: a -> Maybe a

λ > :t Nothing
Nothing :: Maybe a
```

λ_ω

what's the type?

```
λ > :t (,)
(,) :: ???

λ > :t Left
Left :: a -> ???

λ > :t Right
Right :: b -> ???
```

λ_ω

what's the type?

```
λ > :t (,)
(,) :: a -> b -> (a, b)

λ > :t Left
Left :: a -> Either a b

λ > :t Right
Right :: b -> Either a b
```

λ_ω

type constructors

- `[]` and `Maybe` are both unary type constructors
- `(,)` and `Either` are both binary type constructors
- type constructors are abstract types that evaluate to a concrete type

λ_ω

kinds

- types are sets of values
  (e.g. `Bool`, `Int`, `Char`)

- kinds are sets of types

λ_ω

kinds

```
λ > :k Bool
Bool :: *

λ > :k Int
Int :: *

λ > :k Char
Char :: *
```

λ_ω

kinds

```
λ > :k []
[] :: * -> *

λ > :k Maybe
Maybe :: * -> *

λ > :k (,)
(,) :: * -> * -> *

λ > :k Either
Either :: * -> * -> *
```

λ_ω

kinds

functions from proper types to proper types

```
λ > :k []
[] :: * -> *

λ > :k Maybe
Maybe :: * -> *
```

λ_ω

kinds

functions from proper types to type operators

```
λ > :k (,)
(,) :: * -> * -> *

λ > :k Either
Either :: * -> * -> *
```

System F_ω (λ_ω)

it's turtles all the way up

- λ adds variables, function abstraction, and function application at the value level
- $\lambda_\to$ adds type verification at the term level
- System F adds parametric polymorphism on types

System F_ω (λ_ω)

it's turtles all the way up

- λ_ω adds variables, function abstraction, and function application at the type level
- $\lambda_{\omega\rightarrow}$ adds kind verification at the type level
- System F_ω adds parametric polymorphism on kinds

And Beyond

- System $F_{<:}$ - adds sub typing, bounded quantification
- System $F^{\omega}_{<:}$ - combines System F_{ω} and System $F_{<:}$
- dependent types - Agda, Idris
- calculus of constructions - Coq

And Beyond

- System $F_{<:}$ - adds sub typing, bounded quantification
- System $F^{\omega}_{<:}$ - combines System F_{ω} and System $F_{<:}$

And Beyond

- dependent types - Agda, Idris
- calculus of constructions - Coq