Introduction to typed and polymorphic languages

L.Cardelli & P. Wegner (1985)

Soumya D. Sanyal
and
Chris Hathhorn

Why study foundations?

- Allows you to reason about the abstractions you’re using.
- Know what’s a mistake and what’s not.
- Understand subtleties of programming practice.

Lambda Calculus

- Mathematical foundations of computation and programs, due to Alonzo Church.
- Untyped lambda calculus
  - Example implementation in Haskell
- Typed lambda calculus
  - Another example implementation in Haskell
- Code on
  - https://github.com/soumyadsanyal/lambdaloungetalk

Lambda Calculus

- Model of computation underlying functional programming languages
  - e.g.: Haskell is a simply typed lambda calculus with additional datatypes and constants
- Consists purely of applying functions
- Can be used to define arithmetic, boolean logic, pairs, tuples, projections maps, recursive functions, lists, trees,...

Lambda Calculus

- Syntax:
  - Terms t of the (untyped) **λ** calculus consist of:
    - variables x
    - abstractions **λ**x. t
    - applications t t
  - and that's it.

Lambda Calculus

- Examples:
  - id = $\lambda x.x$
  - plusone = $\lambda x.(x+1)$
  - twice = $\lambda f.\lambda x.(f(f(x))$
  - plus = $\lambda x.\lambda y.(x+y)$
  - zero = $\lambda s.\lambda z.z$
  - one = $\lambda s.\lambda z.\ s\ z$
  - two = $\lambda s.\lambda z.\ s\ (s\ z)$
  - three = $\lambda s.\lambda z.\ s\ (s\ (s\ z))$ ...

Lambda Calculus

- How to compute with these things?
- By β - reduction
- Essentially substitute:
  - (λx.t) s = [x/s]t
  - In plain English: (λx.t) is a function, s is the argument, apply the function to the argument by plugging in s for x in the body of the lambda expression t

Lambda Calculus

- Examples:
  - Easy:
    - $(\lambda x.x)(1) = [x/1]\ x = 1$
    - $(\lambda x.x+1)\ 100 = [x/100]\ (x+1) = 101$

Lambda Calculus

- Examples:
  - Higher order:
    - λf.λx.(f(f(x))) (λx.x+1) (1)
    - = [f/(λx.x+1)] λx.(f(f(x))) (1)
    - = λx.((λx.x+1)((λx.x+1)(x))) (1)
    - = λx.((λx.x+1)([x/x] x+1)) (1)
    - = λx.((λx.x+1)(x+1)) (1)
    - = λx.([x/x+1] x+1) (1)
    - = λx.((x+1)+1)) (1)
    - = [x/1] (x+1) + 1 = (1+1) + 1 = 3

Lambda Calculus

- Another example:
  - $(\lambda x.x)\ ((\lambda x.x)\ (\lambda z.\ (\lambda x.x)\ z))$
  - $= (\lambda x.x)\ ((\lambda x.x)\ (\lambda z.\ [x/z]\ x))$
  - $= (\lambda x.x)\ ((\lambda x.x)\ (\lambda z.\ z))$
  - $= (\lambda x.x)\ ([x/(\lambda z.\ z)]\ x\ )$
  - $= (\lambda x.x)\ (\lambda z.\ z)$
  - $= [x/(\lambda z.\ z)]\ x$
  - $= \lambda z.\ z$
  - which is the identity function, who knew!

Lambda Calculus

- A harder example:
  - Define plus = λm.λn.λs.λz. m s (n s z)
  - Work out out that:
    - plus one two = three
    - purely by β reduction!
  - Define times = λm. λn. m (plus n) zero
  - Work out that:
    - times two three = six
    - purely by β reduction!

Lambda Calculus

- Different evaluation strategies lead to different program behavior
  - Full beta reduction (anything can be reduced at any time)
  - Normal order strategy (left to right)
  - Call by name (no reductions inside the body of a lambda expression)
  - Call by value (only reduce when the argument is fully reduced to a value)

Lambda Calculus

- Variables can be bound or free
  - In the expression λx.(x+y) , x is a bound variable and y is a free variable
- Expressions in which no variables are free are called combinators
  - λx.x
  - omega = (λx. x x) (λx. x x) (divergent combinator)

Lambda Calculus

- The free variables of a term can be computed using the formulas:
  - $FV(x) = \{x\}$
  - $FV(\lambda x.t) = FV(t) - \{x\}$
  - $FV(t\ s)$ = union of $FV(t)$ and $FV(s)$
- Free and bound variables require different treatment under substitution

Lambda Calculus

- Can substitute variables for terms or other variable names at will if the variables are free
  - $[x/s]\ x = s$
  - $[x/s]\ y = y$ (if $x != y$)
  - $[x/s]\ \lambda y.t = \lambda y.\ [x/s]t$
  - $[x/s]\ (u\ v) = ([x/s]\ u)\ ([x/s]\ v)$
  - $[x/(\lambda z.\ z\ w)]\ (\lambda y.\ x) = \lambda y.\ \lambda z.\ z\ w$

Lambda Calculus

- But not if the variable is bound
  - [x/y] (λx.x) = λx.y
    - This changed the identity function into a constant function!
- What if we just don't allow ourselves to replace a bound variable?
  - [x/z] (λz.x) = λz.z
    - This changed a constant function into an identity function!

Lambda Calculus

- It turns out the correct method of substitution is to make sure that:
  - you don’t substitute for a bound variable, and
  - the term you substitute in does not have any free variables that conflict with the bound variables of the destination term

Lambda Calculus

- Untyped lambda calculus
  - is Turing complete - it can compute any computable function
  - but this power is dangerous and allows pathological computations
    - $f = \lambda x.x$
    - What is f(f) ?
    - $omega = \lambda x.\ x\ (x)$
    - What is omega(omega) ?

Lambda Calculus

- $f = \lambda x.x$
  - $f(x) = \lambda x.x\ (x) = [x/x]\ x = x$ for any x
  - $f(f) = f$
- $omega = \lambda x.\ x\ (x)$
  - $omega(omega)$
  - $= \lambda x.\ x\ (x)\ (\lambda x.\ x\ (x))$
  - $= [x/(\lambda x.\ x\ (x))]\ x\ (x)$
  - $= \lambda x.\ x\ (x)\ (\lambda x.\ x\ (x)) = omega(omega)$

Lambda Calculus

- Simply typed lambda calculus
  - Always fully specify the type of every expression
  - No longer Turing complete
  - But all computations halt

Kinds of Types

- Basic types
- Structured types (using type constructors)
- Recursive types

Types

- Basic types
  - Int, Char, Bool, ...
- Structured types (using type constructors)
  - Array, finite Cartesian product, …
    - data Pair = Pair Int Int
- Recursion
  - Trees
    - data Tree' = Empty | Leaf Int | Node Int Tree' Tree'

What is a type system?

- Types provide constraints upon data and methods.
- Enforce contracts (anything illegal is a type error).

Static, strong typing

- Type systems can be static (all expressions can be typed upon static analysis at compile time), strong (all expressions are guaranteed to be type-consistent, even if not determined at compile time).
- Static => strong
- Type inference
  - Haskell supports type inference

Type Polymorphism

Allows more flexible behavior and programming than monomorphic type systems, where every value can have only ONE type.

Type Polymorphism

- Universal
  - Parametric
  - Inclusion
- Ad-hoc
  - Overloading
  - Coercion

Universal polymorphism

- Parametric:
  - Functions have implicit or explicit type parameters
  - data List' a = Nil | Cons a (List' a)
- Inclusion:
  - Methods can operate on various objects related by inclusion/inheritance

Ad-hoc polymorphism

- Overloading:
  - 2+3
  - “Soumya”+” Sanyal”
  - eval (Plus (Constant (VNat (Succ Zero))) (Constant (VNat (Succ (Succ (Succ Zero))))))
  - eval (Plus (Constant (VInt 0)) (Constant (VNat (Succ (Succ (Succ Zero))))))

Ad-hoc polymorphism

- Coercion:
  - 3+4
  - 3.0+4=?
  - 1+” Soumya”=?
  - eval (Times (Constant (VNat (Succ (Succ (Succ Zero))))) (Constant (VInt 100)))

Ad-hoc polymorphism

- The essential difference between ad-hoc polymorphism and monomorphism is a bit blurred.
- One could argue in some cases that overloading is essentially syntactic flexibility within monomorphic type systems.

For another talk ...

- Quantification over types (universal, existential, bounded, mixed)
- Polymorphism via subtyping, inheritance
- Parametric types
- Type inference

Thanks!