INF 102
ANALYSIS OF PROG. LANGS
LAMBDA CALCULUS

Instructors: Kaj Dreef

History

- Formal mathematical system
- Simplest programming language
- Intended for studying functions, recursion
- Invented in 1936 by Alonzo Church (1903-1995)
  - Same year as Turing's paper

Warning

- May seem trivial and/or irrelevant ***now***
- Had a tremendous influence in PLs
  - λ-calculus → Lisp → everything
- Context in the early 60s:
  - Assembly languages
  - Cobol
  - Unstructured programming

What is Calculus?

Calculus is a branch of mathematics that deals with limits and the differentiation and integration of functions of one or more variables

Real Definition

- A *calculus* is just a bunch of rules for manipulating symbols.
- People can give meaning to those symbols, but that's not part of the calculus.
- Differential calculus is a bunch of rules for manipulating symbols. There is an interpretation of those symbols corresponds with physics, slopes, etc.

Syntax

$$
\begin{array}{llll}
M ::= & x & & \text{(variable)} \\
 & \mid & \lambda x.M & \text{(abstraction)} \\
 & \mid & MM & \text{(application)}
\end{array}
$$

Nothing else!

- No numbers
- No arithmetic operations
- No loops
- No etc.

Symbolic computation

Syntax reminder

anonymous functions

λx.M ➔ function(x) { M }

LM, e.g. $\underbrace{\lambda x.N}_{L}\ \underbrace{y}_{M}$ ➔ apply L to M

Terminology – bound variables

λx.M

The *binding operator* λ **binds** *the variable x in the* λ-*term x.M*

- M is called the *scope of x*
- *x* is said to be a *bound variable*

Terminology – free variables

Free variables are all symbols that aren't bound (duh)

$$FV(x) = \{x\}$$
$$FV(MN) = FV(M) \cup FV(N)$$
$$FV(x.M) = FV(M) - x$$

Renaming of bound variables

$\lambda x.\mathrm{M} = \lambda y.([y/x]\mathrm{M})$ if y not in FV(M)

i.e. you can replace x with y
aka “renaming”

α-conversion

Operational Semantics

- Evaluating function application: $(\lambda x.e_1)\ e_2$
  - Replace every *x in* e_1 *with* e_2
  - Evaluate the resulting term
  - Return the result of the evaluation
- Formally: "β-reduction" (aka "substitution")
  - $(\lambda\ x.e_1)\ e_2 \rightarrow_\beta e_1[e_2/x]$
  - A term that can be β-reduced is a *redex (reducible expression)*
  - We omit β when obvious

Note again

- Computation = pure symbolic manipulation
  - Replace some symbols with other symbols

Scoping etc.

- Scope of λ extends as far to the right as possible
  - *λx.λy.xy* *is* *λx.(λy.(x y))*
- Function application is left-associative
  - *xyz means (xy)z*
- Possible syntactic sugar for declarations
  - (λ*x*.N)*M* is **let** *x* = M **in** N
  - (λx.(x + 1))10 is **let** *x*=10 **in** (x+1)

Multiple arguments

- $\lambda(x,y).e$???
  - Doesn't exist
- Solution: $\lambda x.\lambda y.e$ *[remember, $(\lambda x.(\lambda y.e))$]*
  - A function that takes *x* and returns another function that takes *y* and returns *e*
  - $(\lambda x.\lambda y.e)\ a\ b \rightarrow (\lambda y.e[a/x])\ b \rightarrow e[a/x][b/y]$
  - "Currying" after Curry: transformation of multi-arg functions into higher-order functions
- Multiple argument functions are nothing but syntactic sugar

Boolean Values and Conditionals

- True = $\lambda x.\lambda y.x$
- False = $\lambda x.\lambda y.y$
- If-then-else = $\lambda a.\lambda b.\lambda c.\ a\ b\ c = a\ b\ c$
- For example:
  - If-then-else true b c $\rightarrow(\lambda x.\lambda y.x)\ b\ c\rightarrow(\lambda y.b)\ c\rightarrow b$
  - If-then-else false b c $\rightarrow(\lambda x.\lambda y.y)\ b\ c\rightarrow(\lambda y.y)\ c\rightarrow c$

Boolean Values and Conditionals

- If True M N = $(\underbrace{\lambda a.\lambda b.\lambda c.abc}_{\text{If}})$ True M N

$$\rightarrow \quad (\lambda b.\lambda c.\,True\ b\ c)\ \mathrm{M\ N}$$
$$\rightarrow \quad (\lambda c.\,True\ M\ c)\ \mathrm{N}$$
$$\rightarrow \quad True\ M\ N$$
$$= (\lambda x.\lambda y.x)\ M\ N$$
$$\rightarrow \quad (\lambda y.M)\ N$$
$$\rightarrow \quad \mathrm{M}$$

Numbers…

- Numbers are counts of things, any things. Like function applications!

  - $0 = \lambda f.\ \lambda x.\ x$
  - $1 = \lambda f.\ \lambda x.\ (f\ x)$
  - $2 = \lambda f.\ \lambda x.\ (f\ (f\ x))$
  - $3 = \lambda f.\ \lambda x.\ (f\ (f\ (f\ x)))$
  - …
  - $N = \lambda f.\ \lambda x.\ (f^N\ x)$

Church numerals

Successor

- succ = $\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
  - Want to try it on succ(1)?
  - $\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)\ \underbrace{(\lambda f.\ \lambda x.\ (f\ x))}_{1}$

$\rightarrow\ \lambda f.\ \lambda x.\ f\ ((\lambda f.\ \lambda x.\ (f\ x))\ f\ x)$

$\rightarrow\ \underbrace{\lambda f.\ \lambda x.\ f\ (f\ x)}_{2\ !}$

There's more

- Reading materials

Recursion ???

```
(λn.
    (if (zero? n)
      1
    (* n (f (sub1 n)))))
```

???
Free variable

Recursion – The Y Combinator

$$Y = \lambda t.\ (\lambda x.\ t\ (x\ x))\ (\lambda x.\ t\ (x\ x))$$

```
Y a = λt. (λx. t (x x)) (λx. t (x x)) a
    = (λx. a (x x)) (λx. a (x x))
    = a ((λx. a (x x)) (λx. a (x x)))
    = a (Y a)
```

Y *a* = *a* applied to itself!

Y a = a (Y a) = a (a (Y a)) = a (a (a (Y a))) = ...

Factorial again

~~λn.~~
~~(if (zero? n)~~
~~1~~
~~(* n (f (sub1 n))))~~

λf.λn.
(if (zero? n)
1
(* n (f (sub1 n))))

F

Now it's bound!

Y F

Does it work?

F takes one function and one number as arguments

```
(Y F) 2 = F (Y F) 2
        = λf.λn.(if (zero? n) 1 (* n (f (sub1 n))))
          ((λt.(λx.t (x x)) (λx.t (x x))
           (λf.λn.(if (zero? n) 1 (* n (f (sub1 n))))))
          2
        = if (zero? 2) 1 (* 2 (Y F (sub1 2)))
        = (* 2 (Y F (sub1 2)))
        = (* 2 (Y F 1))
        = ...
        = (* 2 1)
        = 2
```

F

(Y F)

Points to take home

- Model of computation completely different from Turing Machine
  - pure functions, no commands
- Church-Turing thesis: the two models are equivalent
  - What you can compute with one can be computed with the other
- Inspiration behind Lisp (late 1950s)
- Foundation of all "functional programming" languages