Lecture 8: Lambda Calculus

CSC 131
Spring, 2019

Kim Bruce

Pure Lambda Calculus

- Terms of pure lambda calculus
  - M ::= v | (M M) | λv. M
  - Impure versions add constants, but not necessary!
  - Turing-complete
- Left associative: M N P = (M N) P.
- Computation based on substituting actual parameter for formal parameters

Free Variables

- Substitution easy to mess up!
- Def: If M is a term, then FV(M), the collection of free variables of M, is defined as follows:
  - FV(x) = { x }
  - FV(M N) = FV(M) ∪ FV(N)
  - FV(λv. M) = FV(M) - {v}

Substitution

- Write [N/x] M to denote result of replacing all free occurrences of x by N in expression M.
  - [N/x] x = N,
  - [N/x] y = y, if y ≠ x,
  - [N/x] (L M) = ([N/x] L) ([N/x] M),
  - [N/x] (λy. M) = λy. ([N/x] M), if y≠ x and y ∉ FV(N),
  - [N/x] (λx. M) = λx. M.

Computation Rules

- Reduction rules for lambda calculus:

  (α) $\lambda x.\ M \rightarrow \lambda y.\ ([y/x]\ M)$, if $y \notin FV(M)$.

  change name of parameters if new not capture old

  (β) $(\lambda x.\ M)\ N \rightarrow [N/x]\ M$.

  computation by subst function argument for formal parameter

  (η) $\lambda x.\ (M\ x) \rightarrow M$.

  Optional rule to get rid of excess λ's

Why so complicated?

illegal substitution

$\underline{(\lambda f.\ \lambda z.\ f\ (f\ z))\ (\lambda x.\ x + z)} \rightarrow \lambda z.\ (\lambda x.\ x + z)(\underline{(\lambda x.\ x + z)\ z})$

$\rightarrow \lambda z.\ \underline{(\lambda x.\ x + z)(z + z)}$

$\rightarrow \lambda z.\ (z + z) + z = \lambda z.\ 3z.$

- *rather than the correct*

$\underline{(\lambda f.\ \lambda y.\ f\ (f\ y))\ (\lambda x.\ x + z))} \rightarrow \lambda y.\ (\lambda x.\ x + z)(\underline{(\lambda x.\ x + z)\ y})$

$\rightarrow \lambda y.\ \underline{(\lambda x.\ x + z)(y + z)}$

$\rightarrow \lambda y.\ (y + z) + z = \lambda y.\ y + 2z.$

Normal Forms

- A term M is in normal form if no reduction rules apply, even after applications of α.
- Not all terms have normal forms
  - $\Omega = (\lambda x.\ (x\ x))(\lambda x.\ (x\ x))$

How to evaluate

- Many strategies:
  - $(\lambda x.\ x + 32)(\underline{(\lambda y.\ y * 3)\ 5}) \rightarrow \underline{(\lambda x.\ x + 32)\ 15} \rightarrow 47$ *Inside-out*
  - versus
  - $\underline{(\lambda x.\ x + 32)((\lambda y.\ y * 3)\ 5)} \rightarrow (\underline{(\lambda y.\ y * 3)\ 5}) + 32 \rightarrow 47$ *Outside-in*
- Confluence: If M can be reduced to a normal form, then there is only one such normal form.
- However, not all strategies give a normal form:
  - $(\lambda x.\ 47)\ \Omega$

Computability

- Can encode all computable functions in pure untyped lambda calculus.
  - true = λ u. λ v. u
    - true a b = a
  - false = λ u. λ v. v
    - false a b = b
  - cond = λ u. λ v. λ w. u v w
    - cond true a b = ?   cond false a b = ?

Lambda Encoding

- Pairing:
  - Pair = λ m. λ n. λ b. cond b m n.
  - fst = λp. p true
    - fst (Pair a b) = ?
  - snd = λp. p false
    - snd (Pair a b) = ?

Encoding Natural Numbers

- Natural numbers:
  - 0 = λ s. λ z. z.
  - 1 = λ s. λ z. s z.
  - 2 = λ s. λ z. s (s z).
- Integers encode repetition:
  - 2 f x = f (f x)
  - 3 f x = f(f (f x))
  - n f x = $f^{(n)}$ (x)

Arithmetic

- Succ = λ n. λ s. λ z. s (n s z)
  - Succ n = λ s. λ z. s (n s z) = λ s. λ z. s ($s^{(n)}$ z) = n+1
- Plus = λ n. λ m. λ s. λ z. m s (n s z).
- Mult = λ n. λ m. (m (Plus n) 0).
- isZero = λ n. n (λ x. false) true
- Subtraction is hard!!

Predecessor

- PZero = <0,0> = Pair 0 0
- PSucc = λ n. Pair (snd n) (Succ (snd n))
  - PSucc PZero = <0,1>
  - n PSucc PZero = <n-1,n> *for n > 0*
- Pred = λ n. fst (n PSucc PZero)
  - Pred n = n - 1, *for n > 0,*
  - Pred 0 = 0

Recursion

- Recursive definitions are handy
  - fact = λn. cond (isZero n) 1 (Mult n (fact (Pred n)))
  - *Not* a legal definition in lambda calculus because can't name functions!
- Compute by expanding:
  - fact 2
  - = cond (isZero 2) 1 (Mult 2 (fact (Pred 2)))
  - = Mult 2 (fact 1)
  - = Mult 2 (cond (isZero 1) 1 (Mult 1 (fact (Pred 1)))
  - = Mult 2 (Mult 1 (fact 0)) = ... = Mult 2 (Mult 1 1) = 2

Recursion

- A different perspective: Start with
  - fact = λn. cond (isZero n) 1 (Mult n (fact (Pred n)))
- Let F stand for the closed term:
  - λf. λn. cond (isZero n) 1 (Mult n (f (Pred n)))
  - Notice F(fact) = fact.
  - fact is a *fixed point* of F
  - To find fact, need only find fixed point of F!
- Easy w/ g(x) = x * x, but F????

Fixed Points

- Several fixed point operators:
  - Ex: Y = λf . (λx. f (xx))(λx. f (xx))

Invented by Haskell Curry

- Claim for all g, Y g = g (Y g)

  Y g = (λf . (λx. f (xx))(λx. f (xx))) g

  = (λx. g(xx))(λx. g(xx))

  = g((λx. g(xx)) (λx. g(xx)))

  = g (Y g)

- If let x_0 = Y g, then g (x_0) = x_0.

Factorial

- Recursive definition:
  - let F = λf. λn. cond (isZero n) 1 (Mult n (f (Pred n)))
  - let fact = Y F
  - then F(fact) = fact *because Y always gives fixed points*
- Compute:

fact 0 = (F (fact)) 0 *because fact is a fixed point of F*

= cond (isZero 0) 1 (Mult 0 (fact (Pred 0)))

= 1 *by the definition of cond*

Computing Factorials

fact 1 = (F (fact)) 1 *because fact is a fixed point of F*

= (λn. cond (isZero n) 1 (Mult n (fact (Pred n)))) 1 *expanding F*

= cond (isZero 1) 1 (Mult 1 (fact (Pred 1))) *applying it*

= Mult 1 (fact (Pred 1)) *by the definition of cond*

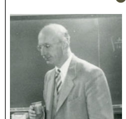
= fact 0 *by the definition of Mult and Pred*

= 1 *by the above calculation*

Lambda Calculus

- λ-calculus invented in 1928 by Church in Princeton & first published in 1932.
- Goal to provide a foundation for logic
- First to state explicit conversion rules.
- Original version inconsistent, but corrected
  - "If this sentence is true then 1 = 2" *problematic!!*
- 1933, definition of natural numbers

Collaborators



- 1931-1934: Grad students:
  - J. Barkley Rosser and Stephen Kleene
  - Church-Rosser confluence theorem ensured consistency (earlier version inconsistent)
  - Kleene showed λ-definable functions very rich
    - Equivalent to Herbrand-Gödel recursive functions
    - Equivalent to Turing-computable functions.
    - Founder of recursion theory, invented regular expressions

- Church's thesis:
  - λ-definability ≡ effectively computable