Lecture 20: Other Models of Computability

CSCI 101
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
- Computation based on substituting actual parameter for formal parameters
  - *Must be careful about defining substitution!*

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) λx. M → λy. ([y/x] M), if y ∉ FV(M).

  (β) (λx. M) N → [N/x] M.

  (η) λx. (M x) → M. *Optional rule*

Computability

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

Lambda Encoding

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

Encoding Integers

- Integers:
  - 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)
  - 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: $\underline{Y} = \lambda f . (\lambda x. f (xx))(\lambda x. f (xx))$

Invented by Haskell Curry

- Claim for all g, $\underline{Y}\, g = g\, (\underline{Y}\, g)$

$\underline{Y}\, g = (\lambda f . (\lambda x. f (xx))(\lambda x. f (xx)))\, g$

$= (\lambda x. g(xx))\underline{(\lambda x. g(xx))}$

$= g(\underline{(\lambda x. g(xx))}\ \underline{(\lambda x. g(xx))})$

$= g\, (\underline{Y}\, g)$

- If let $x_o = \underline{Y}\, g$, then $g\, (x_o) = x_o$.

Factorial

- Recursive definition:

- let $\underline{F} = \lambda f. \lambda n.\ \underline{cond}\ (\underline{isZero}\ n)\ 1\ (\underline{Mult}\ n\ (f\ (\underline{Pred}\ n)))$
- let fact = $\underline{Y}\ \underline{F}$
- then F(fact) = fact *because Y always gives fixed points*

- Compute:

fact $\underline{0}$ = (F (fact)) $\underline{0}$ *because fact is a fixed point of F*

= cond (isZero $\underline{0}$) $\underline{1}$ (Mult $\underline{0}$ (fact (Pred $\underline{0}$)))

= $\underline{1}$ *by the definition of cond*

Computing Factorials

fact $\underline{1}$ = (F (fact)) $\underline{1}$ *because fact is a fixed point of F*

= $(\lambda n.\ \underline{cond}\ (\underline{isZero}\ n)\ \underline{1}\ (\underline{Mult}\ n\ (fact\ (\underline{Pred}\ n))))\ \underline{1}$ *expanding F*

= cond (isZero $\underline{1}$) $\underline{1}$ (Mult $\underline{1}$ (fact (Pred $\underline{1}$))) *applying it*

= Mult $\underline{1}$ (fact (Pred $\underline{1}$)) *by the definition of cond*

= fact $\underline{0}$ *by the definition of Mult and Pred*

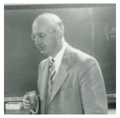
= $\underline{1}$ *by the above calculation*

-

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

Undecidability

- Convertibility problem for λ-calculus undecidable.
- Validity in first-order predicate logic undecidable.
- Proved independently year later by Turing.
  - First showed halting problem undecidable

Alan Turing

- Turing
  - 1936, in Cambridge, England, definition of Turing machine
  - 1936-38, in Princeton to get Ph.D. under Church.
  - 1937, first published fixed point combinator
    - $(\lambda x.\ \lambda y.\ (y\ (x\ x\ y)))\ (\lambda x.\ \lambda y.\ (y\ (x\ x\ y)))$
  - *Kleene did not use fixed-point operator in defining functions on natural numbers!*
  - Broke German enigma code in WW2, Turing test AI
  - Persecuted as homosexual, committed suicide in 1954

With More Work ...

- Show anything computable by TM is computable by lambda calculus ...
  - or by RAM, or WHILE language, or ...
- ... and vice-versa!

Writing Interpreters

Natural (*Operational*) Semantics

- Arithmetic expressions example on web page
  - ArithSemantics.hs
- How to interpret identifiers?
- Environment: Association list of id's & values.
- Semantics defined recursively on abstract syntax trees.

PCF

- Programming language for Computable Functions
- Includes recursive definitions
- Call-by-value (eager) semantics
- Function application as substitution
- Rewriting semantics

Semantics in English

- Semantics of `succ e`
  - Evaluate expression `e` to value v
    - return v+1
- Semantics of `if b then e1 else e2`
  - Evaluate `b`
    - if b evaluates to true return value of e1
otherwise return value of e2

PCF Syntax & Semantics

```
e ::= x | n | true | false | succ | pred | iszero |
      if e then e else e | (fn x => e) | (e e) |
      rec x => e | let x = e1 in e2 end
```

(1) `n ⇒ n` *for n an integer.*

(2) `true ⇒ true, false ⇒ false`

(3) `error ⇒ error`

(4) `succ ⇒ succ,` *and similarly for the other initial functions*

```
    b ⇒ true         e1 ⇒ v
(5) -------------------------
    if b then e1 else e2 ⇒ v
```

More PCF Semantics

```
    b ⇒ false        e2 ⇒ v
(6) -------------------------
    if b then e1 else e2 ⇒ v

    e1 ⇒ succ        e2 ⇒ n
(7) -------------------------
         (e1 e2) ⇒ (n+1)

    e1 ⇒ pred   e2 ⇒ 0        e1 ⇒pred    e2 ⇒ (n+1)
(8) ----------------------     ------------------------
        (e1 e2) ⇒ 0                (e1 e2) ⇒ n

    e1 ⇒ iszero   e2 ⇒ 0     e1 ⇒iszero   e2 ⇒ (n+1)
(9) -----------------------   -------------------------
       (e1 e2) ⇒ true             (e1 e2) ⇒ false
```

More PCF Semantics

```
(10)        (fn x => e) ⇒ (fn x => e)

     e1 ⇒ (fn x => e3)      e2 ⇒ v1    e3[x:=v1] ⇒ v
(11) --------------------------------------------------
                   (e1 e2) ⇒ v
```

Call by value!

```
                 e[x:=rec x => e] ⇒ v
(12)             ---------------------
                   (rec x => e) ⇒ v
```

Like Υ combinator!

Recursion

```
f n  = if (n == 0) then 1 else n*(f(n-1))
```

is written in PCF (assuming have already defined `mult`) as

```
rec f => fn n => if (isZero n) then 1
                               else mult n (f (pred n))
```

which is equivalent to

```
Y(λf. λn. cond (isZero n) 1 (mult n (f (pred n))))
```

Computed via unwinding.

Substitution-based Interpreter

```
data Term = AST_ID String | AST_NUM Int | AST_BOOL Bool
          | AST_SUCC | AST_PRED | AST_ISZERO
          | AST_IF (Term, Term, Term) | AST_ERROR String
          | AST_FUN (String, Term) | AST_APP (Term, Term)
          | AST_REC (String, Term)
```

- Key is to get right definition of substitution that matches static scope
- Interpreter code matches semantic rules
  - PCFSubstInterpreter.hs

PCF Semantics w/Environments

- Substitution slow & space consuming
- Can't handle terms w/free variables
- Environment allows to evaluate once.
- Meaning now separate set of values -- not just rewriting
- Meaning of function is closure, which carries around its environment of definition.

The Problem

- Program:
  - y = 4
  - f x = x + y
  - g (h) = let y = 5 in (h 2) + y
  - g(f)
- When evaluate (h 2), the needed y is out of scope!

Values of Answers

- Key difference w/ new interpreter
  - Update environment, not rewrite term!
  - Not destructive!
- Mutually recursive type definitions:

```
data Value = NUM Int | BOOL Bool | SUCC | PRED |
        ISZERO | CLOSURE (String, Term, Env) |
        THUNK (Term, Env) | ERROR (String, Value)
type Env = [(String, Value)]
```