CS311 Computational Structures

Church's Lambda Calculus

Lecture 14

Andrew Black
based on material from Tim Sheard

Other Notions of Computability

- Many other notions of computability have been proposed, e.g.
  - Grammars
  - Partial Recursive Functions
  - **Lambda calculus**
  - Markov Algorithms
  - Post Algorithms
  - Post Canonical Systems
  - Simple programming language with while loops
- All have been shown equivalent to Turing machines by simulation proofs

The Lambda Calculus

- The Lambda calculus
  - Powerful computation mechanism
  - 3 simple formation rules
  - 2 simple operations
  - extremely expressive

Syntax

A *term* in the calculus has one of the following three forms:

- Let t be a term, and v be a variable
- Then

v	is a term (a variable, an element of a countable set)
$t_1\ t_2$	is a term (an application)
$\lambda\ v\ .\ t$	is a term (an abstraction)

- By convention, the scope of the λ stretches as far to the right as possible, and application is left-associative
- Examples:
  - λ x . x
  - λ z . λ s . s z
  - λ n . snd (n (pair zero zero)
(λ x . pair (succ (fst x)) (fst x)))
  - λ f . (λ x . f (x x)) (λ x . f (x x))

Variables

- The variables in a term can be computed using the following algorithm

  varsOf v = {v}

  varsOf (x y) = varsOf x ∪ varsOf y

  varsOf (λ x . e) = {x} ∪ varsOf e

- Note the form of this algorithm: a *structural induction*

Examples

- varsOf (λ x . x) = {x}

- varsOf (λ z . λ s . s z) = {s, z}

- varsOf (λ n . snd (n (pair zero zero)
  (λ x . pair (succ (fst x)) (fst x))))
  = {n, snd, pair, zero, x, succ, fst}

Free Variables

- The free variables of a term can be computed using the following algorithm

$$\text{freeOf } v = \{v\}$$

$$\text{freeOf } (x\ y) = \text{freeOf } x \cup \text{freeOf } y$$

$$\text{freeOf } (\lambda\ x\ .\ e) = \text{freeOf } e - \{x\}$$

Examples

- freeOf (λ z . λ s . s z) = { }

- freeOf (λ n . snd (n (pair zero zero)
  (λ x . pair (succ (fst x)) (fst x))))
  = {snd, pair, zero, succ, fst}

Alpha conversion

- Terms that differ only in the name of their bound variables are considered equal.

  $(\lambda\ z.\ \lambda\ s.\ s\ z) = (\lambda\ a.\ \lambda\ b.\ b\ a)$

- Also called α-renaming or α-reduction

Substitution

- We can substitute a term for a variable in a lambda-term,
  - e.g., lets substitute (λ y. y) for x in (f x z):

    sub x (λ y. y) (f x z) → (f (λ y. y) z)

- Watch out! We must be careful if the term we are substituting into has a lambda inside

  sub x (g y) (λ y. f x y) → (λ y . f (g y) y)

  sub x (g y) (λ w. f x w) → (λ w . f (g y) w)

Substitution

- We can substitute a term for a variable in a lambda-term,
  - e.g., lets substitute (λ y. y) for x in (f x z):

    sub x (λ y. y) (f x z) → (f (λ y. y) z)

- Watch out! We must be careful if the term we are substituting into has a lambda inside

  sub x (g y) (λ y. f x y) → ~~(λ y . f (g y) y)~~

  sub x (g y) (λ w. f x w) → (λ w . f (g y) w)

Careful Substitution Algorithm

- sub v_1 new (v) = if v_1 = v then new else v
- sub v_1 new (x y) =
  (sub v_1 new x) (sub v_1 new y)
- sub v_1 new (λ v . e) =
  λ v́ . sub v_1 new (sub v v́ e)

  where v́ is a fresh variable not in the free variables of *new*

Example of Substitution

- sub x (g y) (λ y. f x y) →
  λ ý. sub x (g y) (sub y ý (f x y)) →
  λ ý. sub x (g y) (f x ý) →
  λ ý. f (g y) ý

sub v_1 new (v) =
if v_1 = v then new else v
sub v_1 new (x y) =
(sub v_1 new x) (sub v_1 new y)
sub v_1 new (λ v . e) =
λ v´. sub v_1 new (sub v v´e)

Portland State University

Alpha conversion

- Now we can formally define α-conversion:
- If y is not free in X, then

$$(\lambda\ x.\ X) \Leftrightarrow_{\alpha} (\lambda\ y.\ \text{sub}\ x\ y\ X)$$

Beta-conversion

- If we have a term with the form
  $(\lambda\ x\ .\ e)\ v$
  then we can take a β−step to get
  sub x v e

- Formally,
  $(\lambda\ x\ .\ e)\ v \Leftrightarrow_{\beta}$ sub x v e

- In the $\Rightarrow$ direction, this is called reduction, because it gets rid of an application.

Portland State UNIVERSITY

Example

$(\lambda\ n\ .\ \lambda\ z\ .\ \lambda\ s\ .\ n\ (s\ z)\ s)\ \ (\lambda\ z\ .\ \lambda\ s\ .\ z)$

$\Rightarrow \lambda\ z\ .\ \lambda\ s\ .\ (\lambda\ z\ .\ \lambda\ s\ .\ z)\ \ (s\ z)\ s$

$\Rightarrow \lambda\ z\ .\ \lambda\ s\ .\ (\lambda\ z\ .\ \lambda\ s\ .\ z)\ (s\ z)\ s$

$\Rightarrow \lambda\ z\ .\ \lambda\ s\ .\ (\lambda\ s0\ .\ s\ z)\ s$

$\Rightarrow \lambda\ z\ .\ \lambda\ s\ .\ s\ z$

Eta Conversion

- There is one other way to remove a application — when it doesn’t achieve anything, because the argument isn’t used.
- Example: If x does not appear in f, then

  $$(\lambda\ x.\ f\ x)\ g = f\ g$$

  so we allow $(\lambda\ x.\ f\ x) \rightarrow f$
- Formally: if x is not free in f, then

  $$(\lambda\ x.\ f\ x) \Leftrightarrow_{\eta} f$$

Portland State UNIVERSITY

What good is this?

How can we possibly compute with the λ-calculus when we have no data to manipulate!

1. no numbers
2. no data-structures
3. no control structures (if-then-else, loops)

Answer:

Use what we have to build these from scratch!

We use λ-combinators: terms without free variables

The Church numerals

- We can encode the natural numbers
  - zero = λ z . λ s . z
  - one = λ z . λ s . s z
  - two = λ z . λ s . s (s z)
  - three = λ z . λ s . s (s (s z))
  - four = λ z . λ s . s (s (s (s z)))
- What is the pattern here?

Can we use this?

- The succ function:

  succ one → two

- succ (λ z . λ s . s z) → (λ z . λ s . s (s z))

- Can we write this? Let's try
  - succ = λ n . ???

The succ Combinator

- succ = λ n . λ z . λ s . n (s z) s

- succ one $\rightarrow$

  (λ n . λ z . λ s . n (s z) s) one $\rightarrow$

  λ z . λ s . one (s z) s $\rightarrow$

  λ z . λ s . (λ z . λ s . s z) (s z) s $\rightarrow$

  λ z . λ s . (λ s0 . s0 (s z)) s $\rightarrow$

  λ z . λ s . s (s z)

Can we write the add function?

$$add = \lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ \lambda\ s\ .\ x\ (y\ z\ s)\ s$$

- what about multiply?

Can we build the booleans?

- We'll need
  - true: Bool
  - false: Bool
  - if: Bool → x → x → x

- true = λ t . λ f . t

- false = λ t . λ f . f

- if = λ b . λ then . λ else . b then else

- Lets try it out:

  if false two one

What about pairs?

- we'll need
  - pair: a → b → Pair a b
  - fst: Pair a b → a
  - snd: Pair a b → b
- Define:
  - pair = λ x . λ y . λ k . k x y
  - fst = λ p . p (λ x . λ y . x)
  - snd = λ p . p (λ x . λ y . y)

Can we write the pred function?

pred = λ n . snd

(n (pair zero zero)

(λ x . pair (succ (fst x)) (fst x)))

- How does this work?

Can we write the pred function?

pred = λ n . snd

(n (pair zero zero)

(λ x . pair (succ (fst x)) (fst x)))

- How does this work?

the pair ⟨a, b⟩ encodes the fact that (pred a) = b

What about primitive recursion?

- Recall:
  - #0 = λ z . λ s . z
  - #1 = λ z . λ s . s z
  - #2 = λ z . λ s . s (s z)
  - …
  - #n = λ z . λ s . s^n z
- The *n* th Church-numeral already has the power to apply s to z *n* times!
- We also need to use *pair* to carry around the intermediate result.

Factorial

- Defined (using the primitive recursion scheme) as a base case and an inductive step:

base = pair #0 #1 — base = $\langle 0, 1\rangle$

step = λp . pair (succ (fst p)) (mult (succ (fst p)) (snd p))

— step (n, r) = $\langle n+1, (n+1) \times r\rangle$

fact = λn. snd (n base step) — fact n = step^n base $\downarrow 2$

Think about this:

$(\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)$

The Y combinator

The Y combinator

- Define

The Y combinator

- Define

  $y = \lambda\, f_0 \,.\, (\lambda\, x \,.\, f_0\, (x\ x))\ (\lambda\, x \,.\, f_0\, (x\ x))$

The Y combinator

- Define

  $y = \lambda f_0 . (\lambda x . f_0 (x x)) (\lambda x . f_0 (x x))$

- what happens if we apply y to f ?

The Y combinator

- Define

  $y = \lambda\ f_0\ .\ (\lambda\ x\ .\ f_0\ (x\ x))\ (\lambda\ x\ .\ f_0\ (x\ x))$

- what happens if we apply y to f ?

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

The Y combinator

- Define

  $y = \lambda\, f_0 \,.\, (\lambda\, x \,.\, f_0\ (x\ x))\ (\lambda\, x \,.\, f_0\ (x\ x))$

- what happens if we apply y to f ?

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

  f

Portland State UNIVERSITY

The Y combinator

- Define

  $y = \lambda\, f_0\, .\, (\lambda\, x\, .\, f_0\, (x\, x))\, (\lambda\, x\, .\, f_0\, (x\, x))$

- what happens if we apply y to f ?

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

  $f\, (\overline{(\lambda\, x\, .\, f\, (x\, x)}))$

The Y combinator

- Define

  $y = \lambda\, f_0 \,.\, (\lambda\, x \,.\, f_0\, (x\; x))\; (\lambda\, x \,.\, f_0\, (x\; x))$

- what happens if we apply y to f ?

  $y\; f \rightarrow (\lambda\, x \,.\, f\, (x\; x))\; (\lambda\, x \,.\, f\, (x\; x))$

  $f\; ((\lambda\, x \,.\, f\, (x\; x))\;\; (\lambda\, x \,.\, f\, (x\; x)))$

The Y combinator

- Define

$$y = \lambda\, f_0 \,.\, (\lambda\, x \,.\, f_0\, (x\ x))\ (\lambda\, x \,.\, f_0\, (x\ x))$$

- what happens if we apply y to f ?

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

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

$$=\ f\ (y\ f)$$

Fixed points

- The *fixed point* of a function f : Nat → Nat is a value x ∈ Nat such that
  - f x = x
- Note that y f = f (y f)
  - so (y f) is a fixed point of the function f
  - y is called the fixed point combinator. When y is applied to a function, it answers a value x in that function's domain. When you apply the function to x, you get x.

Ackermann's function

The Ackermann function $A(x, y)$ is defined for integers x and y by

$$A(x, y) \equiv \begin{cases} y + 1 & \text{if } x = 0 \\ A(x-1, 1) & \text{if } y = 0 \\ A(x-1, A(x, y-1)) & \text{otherwise.} \end{cases}$$

Special values for x include the following:

$$\begin{aligned} A(0, y) &= y + 1 \\ A(1, y) &= y + 2 \\ A(2, y) &= 2y + 3 \\ A(3, y) &= 2^{y+3} - 3 \\ A(4, y) &= \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{y+3} - 3. \end{aligned}$$

Portland State UNIVERSITY

- Ackermann's function grows faster than any primitive recursive function, that is:

  for *any* primitive recursive function f, there is an n such that

$$A(n, x) > f\,x$$

- So A *can't* be primitive recursive
- Can we define A in the λ-calculus?

Ackermann's function in the λ-calculus

$$A(x, y) \equiv \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1, 1) & \text{if } y = 0 \\ A(x-1, A(x, y-1)) & \text{otherwise.} \end{cases}$$

- Using the Y combinator, we can define Ackerman's function in the λ-calculus, even though it is not primitive recursive!
- First define a function ackermannGen, whose fixed-point will be ackermann:
- ackermannGen = λ ackermann . λ x . λ y
  ifZero x (succ y)
      (ifZero y
          (ackermann (pred x) one)
          (ackermann (pred x) (ackermann x (pred y)))))

Portland State UNIVERSITY

Ackermann's function in the λ-calculus

- Then take the fixed point of that function:
- acc = (y acckermanGen)
- Try it:

```
prompt> acc #1 #1
acc #1 #1
prompt> :b
\ z . \ s . s (s (s z))
prompt> acc #2 #1
acc #2 #1
prompt> :b
\ z . \ s . s (s (s (s (s z))))
prompt>
```

Portland State UNIVERSITY

Summary: the λ-calculus

- The λ-calculus is "Turing complete":
  - It can compute anything that can be computed by a Turing machine
- Augmented with arithmetic, it is the basis for many programming languages, from Algol 60 to modern functional languages.
- Lambda notion, as a compact way of defining anonymous functions, is in most programming languages, including C#, Ruby and Smalltalk