

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:
 - $\lambda x . x$
 - $\lambda z . \lambda s . s z$
 - $\lambda n . \text{snd } (n (\text{pair zero zero}) (\lambda x . \text{pair } (\text{succ } (\text{fst } x)) (\text{fst } x)))$
 - $\lambda f . (\lambda x . f (x x)) (\lambda x . f (x x))$

Variables

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

$$\text{varsOf } v = \{v\}$$

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

$$\text{varsOf } (\lambda x . e) = \{x\} \cup \text{varsOf } e$$

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

Examples

- $\text{varsOf } (\lambda x . x) = \{x\}$
- $\text{varsOf } (\lambda z . \lambda s . s z) = \{s, z\}$
- $\text{varsOf } (\lambda n . \text{snd } (n (\text{pair zero zero})$
 $(\lambda x . \text{pair } (\text{succ } (\text{fst } x)) (\text{fst } x))))$
 $= \{n, \text{snd}, \text{pair}, \text{zero}, x, \text{succ}, \text{fst}\}$

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

- $\text{freeOf } (\lambda z . \lambda s . s z) = \{ \}$
- $\text{freeOf } (\lambda n . \text{snd } (n (\text{pair zero zero})$
 $(\lambda x . \text{pair } (\text{succ } (\text{fst } x)) (\text{fst } x))))$
 $= \{\text{snd}, \text{pair}, \text{zero}, \text{succ}, \text{fst}\}$

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 $(\lambda y. y)$ for x in $(f\ x\ z)$:
$$\text{sub } x\ (\lambda y. y)\ (f\ x\ z) \rightarrow (f\ (\lambda y. y)\ z)$$
- Watch out! We must be careful if the term we are substituting into has a lambda inside

$$\begin{array}{ccc} \text{sub } x \text{ (g } y) \text{ } (\lambda y. f \text{ } x \text{ } y) & \rightarrow & (\lambda y. f \text{ (g } y) \text{ } y) \\ \parallel \parallel & & \not\parallel \\ \text{sub } x \text{ (g } y) \text{ } (\lambda w. f \text{ } x \text{ } w) & \rightarrow & (\lambda w. f \text{ (g } y) \text{ } w) \end{array}$$

Substitution

- We can substitute a term for a variable in a lambda-term,

- ▶ e.g., lets substitute $(\lambda y. y)$ for x in $(f x z)$:

$$\text{sub } x (\lambda y. y) (f x z) \rightarrow (f (\lambda y. y) z)$$

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

$$\begin{array}{lcl} \text{sub } x (g y) (\lambda y. f x y) & \rightarrow & \cancel{(\lambda y. f (g y) y)} \\ \quad \quad \quad \parallel & & \quad \quad \quad \parallel \\ \text{sub } x (g y) (\lambda w. f x w) & \rightarrow & (\lambda w. f (g y) w) \end{array}$$

Careful Substitution Algorithm

- ▶ $\text{sub } v_1 \text{ new } (v) = \text{if } v_1 = v \text{ then new else } v$
- ▶ $\text{sub } v_1 \text{ new } (x \ y) =$
 $(\text{sub } v_1 \text{ new } x) (\text{sub } v_1 \text{ new } y)$
- ▶ $\text{sub } v_1 \text{ new } (\lambda v . e) =$
 $\lambda \acute{v} . \text{sub } v_1 \text{ new } (\text{sub } v \acute{v} e)$

where $\acute{v}$ is a fresh variable not in the free variables of *new*

Example of Substitution

- $\text{sub } x (g \ y) (\lambda y. f \ x \ y) \rightarrow$
 $\lambda \acute{y}. \text{sub } x (g \ y) (\text{sub } y \ \acute{y} (f \ x \ y)) \rightarrow$
 $\lambda \acute{y}. \text{sub } x (g \ y) (f \ x \ \acute{y}) \rightarrow$
 $\lambda \acute{y}. f (g \ y) \ \acute{y}$

$\text{sub } v_1 \text{ new } (v) =$
 if $v_1 = v$ then new else v
 $\text{sub } v_1 \text{ new } (x \ y) =$
 $(\text{sub } v_1 \text{ new } x) (\text{sub } v_1 \text{ new } y)$
 $\text{sub } v_1 \text{ new } (\lambda v . e) =$
 $\lambda v' . \text{sub } v_1 \text{ new } (\text{sub } v \ v' e)$

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
$$\text{sub } x v e$$
- Formally,
$$(\lambda x . e) v \Leftrightarrow_{\beta} \text{sub } x v e$$
- In the $\Rightarrow$ direction, this is called reduction, because it gets rid of an application.

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

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
 - ▶ $\text{zero} = \lambda z . \lambda s . z$
 - ▶ $\text{one} = \lambda z . \lambda s . s z$
 - ▶ $\text{two} = \lambda z . \lambda s . s (s z)$
 - ▶ $\text{three} = \lambda z . \lambda s . s (s (s z))$
 - ▶ $\text{four} = \lambda z . \lambda s . s (s (s (s z)))$
- What is the pattern here?

Can we use this?

- The succ function:
succ one $\rightarrow$ two
- $\text{succ } (\lambda z . \lambda s . s z) \rightarrow (\lambda z . \lambda s . s (s z))$
- Can we write this? Let's try
 - $\text{succ} = \lambda n . ???$

The succ Combinator

- $\text{succ} = \lambda n . \lambda z . \lambda s . n (s z) s$
- $\text{succ one} \rightarrow$
 $(\lambda n . \lambda z . \lambda s . n (s z) s) \text{ one} \rightarrow$
 $\lambda z . \lambda s . \text{one} (s z) s \rightarrow$
 $\lambda z . \lambda s . (\lambda z . \lambda s . s z) (s z) s \rightarrow$
 $\lambda z . \lambda s . (\lambda s0 . s0 (s z)) s \rightarrow$
 $\lambda z . \lambda s . s (s z)$

Can we write the add function?

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

- what about multiply?

Can we build the booleans?

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

- $\text{true} = \lambda t . \lambda f . t$
- $\text{false} = \lambda t . \lambda f . f$
- $\text{if} = \lambda b . \lambda \text{then} . \lambda \text{else} . b \text{ then else}$

- Lets try it out:
if false two one

What about pairs?

- we'll need
 - ▶ $\text{pair}: a \rightarrow b \rightarrow \text{Pair } a \ b$
 - ▶ $\text{fst}: \text{Pair } a \ b \rightarrow a$
 - ▶ $\text{snd}: \text{Pair } a \ b \rightarrow b$
- Define:
 - ▶ $\text{pair} = \lambda x . \lambda y . \lambda k . k \ x \ y$
 - ▶ $\text{fst} = \lambda p . p \ (\lambda x . \lambda y . x)$
 - ▶ $\text{snd} = \lambda p . p \ (\lambda x . \lambda y . y)$

Can we write the pred function?

$\text{pred} = \lambda n . \text{snd}$

$(n (\text{pair zero zero})$

$(\lambda x . \text{pair} (\text{succ} (\text{fst } x)) (\text{fst } x)))$

- How does this work?

Can we write the pred function?

$\text{pred} = \lambda n . \text{snd}$

$(n (\text{pair zero zero})$

$(\lambda x . \text{pair} (\text{succ} (\text{fst } x)) (\text{fst } x)))$

- How does this work?

the pair $\langle a, b \rangle$ encodes the fact that $(\text{pred } a) = b$

What about primitive recursion?

- Recall:
 - ▶ $\#0 = \lambda z . \lambda s . z$
 - ▶ $\#1 = \lambda z . \lambda s . s z$
 - ▶ $\#2 = \lambda z . \lambda s . s (s z)$
 - ▶ ...
 - ▶ $\#n = \lambda z . \lambda s . s^n z$
- 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:

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

$\text{step} = \lambda p . \text{pair } (\text{succ } (\text{fst } p)) (\text{mult } (\text{succ } (\text{fst } p)) (\text{snd } p))$

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

$\text{fact} = \lambda n . \text{snd } (n \text{ base step})$ — $\text{fact } n = \text{step}^n \text{ base} \downarrow 2$

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 ?

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

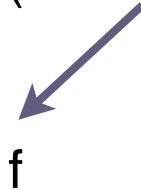
The Y combinator

- 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 ?

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



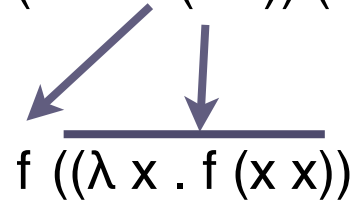
The Y combinator

- 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 ?

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



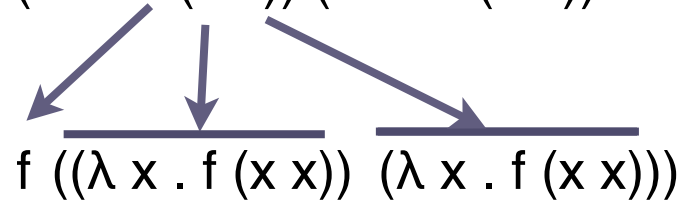
The Y combinator

- 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 ?

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



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 ?

$$\begin{aligned} y f &\rightarrow (\lambda x . f (x x)) (\lambda x . f (x x)) \\ &\quad \swarrow \quad \downarrow \quad \searrow \\ &\quad \underline{f ((\lambda x . f (x x)) (\lambda x . f (x x)))} \\ &= f (y f) \end{aligned}$$

Fixed points

- The *fixed point* of a function $f: \text{Nat} \rightarrow \text{Nat}$ is a value $x \in \text{Nat}$ 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:

$$A(0, y) = y + 1$$

$$A(1, y) = y + 2$$

$$A(2, y) = 2y + 3$$

$$A(3, y) = 2^{y+3} - 3$$

$$A(4, y) = \underbrace{2^{2^{\dots^2}}}_{y+3} - 3.$$

- 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:
- $\text{ackermannGen} = \lambda \text{ ackermann} . \lambda x . \lambda y$
ifZero x (succ y)
 (ifZero y
 (ackermann (pred x) one)
 (ackermann (pred x) (ackermann x (pred y)))))

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>
```

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