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
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 \( \lambda \) 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 \textit{structural induction}
Examples

• \( \text{varsOf} \ (\lambda \ x \ . \ x) = \{x\} \)

• \( \text{varsOf} \ (\lambda \ z \ . \ \lambda \ s \ . \ s \ z) = \{s, z\} \)

• \( \text{varsOf} \ (\lambda \ n \ . \ \text{snd} \ (n \ (\text{pair} \ \text{zero} \ \text{zero}) \ (\lambda \ x \ . \ \text{pair} \ (\text{succ} \ (\text{fst} \ x)) \ (\text{fst} \ x)))) = \{n, \ \text{snd}, \ \text{pair}, \ \text{zero}, \ x, \ \text{succ}, \ \text{fst}\} \)
Free Variables

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

\[
\begin{align*}
\text{freeOf } v &= \{v\} \\
\text{freeOf } (x \ y) &= \text{freeOf } x \cup \text{freeOf } y \\
\text{freeOf } (\lambda \ x \ . \ e) &= \text{freeOf } e - \{x\}
\end{align*}
\]
Examples

- \( \text{freeOf} (\lambda z . \lambda s . s z) = \{ \} \)

- \( \text{freeOf} (\lambda n . \text{snd} (n \ (\text{pair} \ \text{zero} \ \text{zero}) \ (\lambda x . \text{pair} (\text{succ} (\text{fst} x)) (\text{fst} x)))) = \{\text{snd}, \text{pair}, \text{zero}, \text{succ}, \text{fst}\} \)
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 $\alpha$-renaming or $\alpha$-reduction
Substitution

• We can substitute a term for a variable in a lambda-term,
  ▶ e.g., let's substitute \((\lambda y. y)\) for \(x\) in \((f \times z)\):
    \[
    \text{sub } x (\lambda y. y) (f \times z) \rightarrow (f (\lambda y. y) z)
    \]

• Watch out! We must be careful if the term we are substituting into has a lambda inside
  \[
  \begin{align*}
  \text{sub } x (g y) (\lambda y. f \times y) & \rightarrow (\lambda y. f (g y) y) \\
  \text{sub } x (g y) (\lambda w. f \times w) & \rightarrow (\lambda w. f (g y) w)
  \end{align*}
  \]
Substitution

• We can substitute a term for a variable in a lambda-term,
  ◦ e.g., let's substitute \((\lambda \ y. \ y)\) for \(x\) in \((f \ x \ z)\):
    \[
    \text{sub } x \ (\lambda \ y. \ y) \ (f \ x \ z) \to (f \ (\lambda \ y. \ y) \ z)
    \]

• Watch out! We must be careful if the term we are substituting into has a lambda inside
  \[
  \text{sub } x \ (g \ y) \ (\lambda \ y. \ f \ x \ y) \to \begin{array}{c}
  (\lambda \ y. \ f \ (g \ y) \ y) \\
  \text{not this!}
  \end{array}
  \]
  \[
  \text{sub } x \ (g \ y) \ (\lambda \ w. \ f \ x \ w) \to (\lambda \ w. \ f \ (g \ y) \ w)
  \]
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 \( \text{new} \)
Example of Substitution

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

\[
\begin{align*}
\text{sub } v_1 \ \text{new } (v) &= \text{if } v_1 = v \text{ then } \text{new} \text{ else } v \\
\text{sub } v_1 \ \text{new } (x \ y) &= (\text{sub } v_1 \ \text{new } x) \ (\text{sub } v_1 \ \text{new } y) \\
\text{sub } v_1 \ \text{new } (\lambda \ v. \ e) &= \lambda \ v. \ \text{sub } v_1 \ \text{new } (\text{sub } v \ v \ e)
\end{align*}
\]
Alpha conversion

• Now we can formally define $\alpha$-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
  \[(\lambda x . e) \_\]
  then we can take a $\beta$-step to get
  \[\text{sub } x \_ e\]

- Formally,
  \[(\lambda x . e) \_ \iff_{\beta} \text{sub } x \_ e\]

- 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 \[(\lambda x. f x) \eta f\]

• Formally: if x is not free in f, then
  \[(\lambda x. f x) \equiv_{\eta} f\]
What good is this?

How can we possibly compute with the \(\lambda\)-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 \(\lambda\)-combinators: terms without free variables
The Church numerals

- We can encode the natural numbers
  - zero = \( \lambda z . \lambda s . z \)
  - one = \( \lambda z . \lambda s . s \, z \)
  - two = \( \lambda z . \lambda s . s \, (s \, z) \)
  - three = \( \lambda z . \lambda s . s \, (s \, (s \, z)) \)
  - 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

• succ ($\lambda$ z . $\lambda$ s . s z) $\rightarrow$ ($\lambda$ z . $\lambda$ s . s (s z))

• Can we write this? Let’s try
  ▶ succ = $\lambda$ n . ????
The succ Combinator

- \textbf{succ} = \lambda \ n . \lambda \ z . \lambda \ s . \ n \ (s \ z) \ s

- succ one →
  
  (\lambda \ n . \lambda \ z . \lambda \ s . \ n \ (s \ z) \ s) \ one →
  
  \lambda \ z . \lambda \ s . \one \ (s \ z) \ s →
  
  \lambda \ z . \lambda \ s . \ (\lambda \ z . \lambda \ s . \ s \ z) \ (s \ z) \ s →
  
  \lambda \ z . \lambda \ s . \ (\lambda \ s0 . \ s0 \ (s \ z)) \ s →
  
  \lambda \ z . \lambda \ 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 = \lambda t . \lambda f . t

• false = \lambda t . \lambda f . f

• if = \lambda b . \lambda then . \lambda else . b then else
• Lets try it out:
  
  if false two one
What about pairs?

• we’ll need
  ‣ pair: \( a \rightarrow b \rightarrow \text{Pair } a \ b \)
  ‣ fst: \( \text{Pair } a \ b \rightarrow a \)
  ‣ snd: \( \text{Pair } a \ b \rightarrow b \)

• Define:
  ‣ pair = \( \lambda \ x \ . \ \lambda \ y \ . \ \lambda \ k \ . \ k \ x \ y \)
  ‣ fst = \( \lambda \ p \ . \ p \ (\lambda \ x \ . \ \lambda \ y \ . \ x) \)
  ‣ snd = \( \lambda \ p \ . \ p \ (\lambda \ x \ . \ \lambda \ y \ . \ y) \)
Can we write the pred function?

\[ \text{pred} = \lambda \ n \ . \ \text{snd} \]
\[(n \ (\text{pair} \ \text{zero} \ \text{zero}) ) \]
\[ (\lambda \ x \ . \ \text{pair} \ (\text{succ} \ (\text{fst} \ x)) \ (\text{fst} \ x))) \]

- How does this work?
Can we write the pred function?

pred = \lambda n . \text{snd}

(n (\text{pair zero zero})

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

• How does this work?

the pair \langle a, b \rangle encodes the fact that (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 \( \textit{pair} \) 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 = ⟨0, 1⟩

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

   — step (n, r) = ⟨n+1, (n+1) × r⟩

  fact = λn. snd (n base step)  — fact n = stepⁿ base ↓2
Think about this:

\((\lambda x \cdot x x) (\lambda x \cdot x x)\)
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 \( f \) ?

\[
y f \rightarrow (\lambda x . f (x x)) (\lambda x . f (x x))
\]

\( f \)
The Y combinator

• Define

\[ y = \lambda f_0 \cdot (\lambda x \cdot f_0 (x x)) (\lambda x \cdot f_0 (x x)) \]

• what happens if we apply \( y \) to \( f \)?

\[ y f \rightarrow (\lambda x \cdot f (x x)) (\lambda x \cdot f (x x)) \]

\[ \underbrace{f ((\lambda x \cdot f (x x))} \]
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?
  \[ y f \to (\lambda x . f (x x)) (\lambda x . f (x x)) \]
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 $f : \text{Nat} \to \text{Nat}$ 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 $x$. 
The Ackermann function $A(x, y)$ is defined for integers $x$ and $y$ by

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

Special values for $x$ include the following:

- $A(0, y) = y + 1$
- $A(1, y) = y + 2$
- $A(2, y) = 2y + 3$
- $A(3, y) = 2^{y+3} - 3$
- $A(4, y) = 2^{2^{y+3} - 3}$
• 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 \( \lambda \)-calculus?
Ackermann’s function in the $\lambda$-calculus

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

• Using the Y combinator, we can define Ackerman’s function in the $\lambda$-calculus, even though it is not primitive recursive!

• First define a function `ackermannGen`, whose fixed-point will be `ackermann`:

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

- The $\lambda$-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