50003 - Models Of Computation - (Prof Wiklicky) Lecture 5

Oliver Killane

04/04/22

Lambda Calculus

Lecture Recording

Lecture recording is available here

$$M ::= \overset{\text{Variable}}{x} \quad \overset{\text{Abstraction}}{\lambda x.\, M} \quad \overset{\text{Application}}{\underset{\text{Left associative}}{M\ M}}$$

$$((M)\ M)\ M$$

Syntax

- **Bound Variables** x is bound inside $\lambda x\ .\ M$ (it is bound within the scope of M)
- **Free Variables** y is free inside $\lambda x\ .\ M$ (it is not bound)
- **Closed Term** A λ-term with no free variables, e.g $\lambda x\ y\ z\ .\ x\ y$
- **Binding Occurences** The λ-term's parameters $\lambda x\ y\ z\ .\ (\dots)$, here the x, y and z before the dot.
- **Left Associativity** Lambda Terms are left associative, hence $A\ B\ C\ D \equiv (((A)\ (B))\ (C))\ (D)$

Bound and Free Formally

$$\begin{array}{lll} FreeVariables & (x) & = \{x\} \\ FreeVariables & (\lambda x\ .\ M) & = FreeVariables(M) \setminus \{x\} \\ FreeVariables & (M\ N) & = FreeVariables(M) \cup FreeVariables(N) \end{array}$$

Definition: α-equivalence

$M =_\alpha N$ if and only if N can be obtained from M by renaming bound variables (or vice-versa)

Hence the free variable set must be the same (not renamed).

Substitution

$$M[new/old] \text{ means replace free variable } old \text{ with } new \text{ in } M$$

Only free variables can be substituted. Formally we can describe this as:

$$x[M/y] = \begin{cases} M & x = y \\ x & x \neq y \end{cases}$$

$$(\lambda x\ .\ N)[M/y] = \begin{cases} \lambda x\ .\ N & x = y \text{ (} x \text{ will be bound inside, so cannot go further)} \\ \lambda z\ .\ N[z/x][M/y] & x \neq y \text{ (To avoid name conflicts with } M,\ z \notin ((FV(N) \setminus \{x\}) \cup FV(M) \cup \{y\})) \end{cases}$$

$$(A\ B)[M/y] = (A[M/y])\ (B[M/y])$$

- For variables, simply check if equal.

- For lambda abstractions, if the old term is bound, cannot go further, else, switch the bound term for some term not free inside, in the substitution, and not the new value replacing.
- For applications, simply substitute into both λ-terms.

Example: Basic Substitution

$$\begin{aligned} x[y/x] &= y \\ y[y/x] &= y \\ (x\ y)[y/x] &= y\ y \\ \lambda x\ .\ x\ y[y/x] &= \lambda x\ .\ x\ y \end{aligned}$$

Semantics

Lecture Recording

Lecture recording is available here

$$\frac{}{(\lambda x\ .\ M)\ N \to_\beta M[N/x]} \quad \frac{M \to_\beta M'}{\lambda x\ .\ M \to_\beta \lambda x\ .\ M'} \quad \frac{M \to_\beta M'}{M\ N \to_\beta M'\ N} \quad \frac{N \to_\beta N'}{M\ N \to_\beta M\ N'}$$

$$\frac{M =_\alpha M'\ \ M' \to_\beta N'\ \ N' =_\alpha N}{M \to_\beta N}$$

- A term of the form $(\lambda x\ .\ M)$ is called a **redex**.
- A λ-term may have several different reductions. These different reductions for a **derivation tree**.

Multi-Step Reductions

Steps can be combined using the transitive closure of $\to_\beta$ under α-conversion.

$$\frac{M =_\alpha M'}{M \to_\beta^* M'} \qquad \text{(Reflexivity of } \alpha\text{-conversion)}$$

$$\frac{M \to_\beta M'\ \ M' \to_\beta^* M''}{M \to_\beta^* M''} \qquad \text{(Transitivity)}$$

Definition: Confluence

All derivation paths in the derivation tree that reach some normal form, reach the same normal form.

$$\forall M, M_1, M_2.\ [M \to_\beta^* M_1 \land M \to_\beta^* M_2 \Rightarrow \exists M'.[M_1 \to_\beta^* M' \land M_2 \to_\beta^* M']]$$

$$\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr] & & M_2 \arrow[dl] \\
 & M' &
\end{tikzcd}$$

Definition: β Normal Forms

A λ-term is in β-normal form if it contains no **redexes**, and hence cannot be further reduced.

$$\text{is in normal form}(M) \triangleq \forall N.\ M \not\to_\beta N$$
$$\text{has a normal form}(M) \triangleq \exists M'.\ M \to_\beta^* M' \land \text{is in normal form}(M)$$

If a normal form exists, it is unique.

$$\forall M, N_1, N_2, .\ [M \to_\beta^* N_1 \land M \to_\beta^* N_2 \land \text{is-norm-form}(N_1) \land \text{is-norm-form}(N_2) \Rightarrow N_1 =_\alpha N_2]$$

Definition: β-equivalence

An equivalence relation for $\to_\beta$.

$$M =_\beta N \Leftrightarrow \exists T.\ [M \to_\beta^* T \land N \to_\beta^* T]$$

Reduction Order

For a **redex** $E = (\lambda x\ .\ M)\ N$:

- Any **redex** in M or N is inside of E
- E is outside of any **redex** in M or N

Definition: Innermost Redex

A **Redex** with no **redexes** inside of it.

Definition: Outermost Redex

A **Redex** with no **redexes** outside of it.

We can choose several different orders by which to reduce.

- **Normal Order**
  - Reduce the **leftmost outermost redex** first.
  - This always reduces a λ-term to its normal form if one exists.
  - Can perform computations on unevaluated function bodies.
  - Not used in any programming languages.
- **Call By Name**
  - Reduce the **leftmost outermost** first.
  - Does not reduce the inside of λ-abstractions.
  - Does not always reduce a λ-term to its normal form.
  - Passes unevaluated function parameters into function body. Only evaluating a parameter when it is used.
  - Used with some variation by haskell, R, and LaTeX.
- **Call By Values**
  - Reduce the **leftmost innermost redex** first.
  - Does not reduce the inside of λ-abstractions.
  - Does not always reduce a λ-term to its normal form.
  - Evaluate parameters before passing them to function body.
  - Terminates less often than **call by name** (e.g if a parameter cannot be normalised, but is never used), but evaluated the parameters only once.
  - Used by C, Rust, Java, etc.

Definition: η-equivalence

Captures equality better than $=_\beta$.

$$\frac{x \notin FV(M)}{\lambda x\ .\ M\ x =_\eta M} \qquad \frac{\forall N.\ M\ N =_{\eta^+} M'\ N}{M =_{\eta^+} M'}$$

Namely if the application of M to another λ-term is equivalent to M' applied to the same λ-terms then M and M' are equivalent.

For example with the basic application of f:

$$\lambda x\ .\ f\ x \neq_\beta f \quad \text{however} \quad (\lambda x\ .\ f\ x)\ M =_\beta f\ M \quad \text{and} \quad \lambda x\ .\ f\ x \neq_\eta f$$

Definability

Lecture Recording

Lecture recording is available here

Definition: λ-definable

Partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is **λ-definable** if and only if there is a **closed** λ-term M where:

$$f(x_1, \dots, x_n) = y \Leftrightarrow M\ \underline{x_1}\ \dots\ \underline{x_n} =_\beta y$$

And

$$f(x_1, \dots, x_n) \uparrow \Leftrightarrow M\ \underline{x_1}\ \dots\ \underline{x_n} \text{ has no } \textbf{normal form}$$

λ-definable specifies what can be computed by the lambda calculus, and is equivalent to **Register Machine Computable** or **Turing Machine Computable**.

Encoding Mathematics

Encoding Numbers

We represent natural numbers as **Church Numerals**. These are n repeated applications of some function f.

$$\underline{n} \triangleq \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{n \text{ times}}\dots) \text{ with } n \text{ applications of } f$$

$$\begin{aligned}
\underline{0} &\triangleq \lambda f\ .\ \lambda x\ .\ x \\
\underline{1} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ x \\
\underline{2} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ f\ x \\
\underline{3} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ f\ f\ x \\
\underline{4} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ f\ f\ f\ x \\
\underline{5} &\triangleq \lambda f\ .\ \lambda x\ .\ f\ f\ f\ f\ f\ x \\
&\vdots
\end{aligned}$$

Encoding Addition

Addition is represented as a function application:

$$\underline{m} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{m \text{ times}}\dots) \quad \underline{n} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{n \text{ times}}\dots)$$

$$\underline{m+n} \triangleq \underbrace{(\lambda m\ .\ \lambda n\ .\ \lambda f\ .\ \lambda x\ .\ m\ f\ (n\ f\ x))}_{+}\ \underline{m}\ \underline{n}$$

By applying the functions, we have f applied $m + n$ times, representing the **Church Numeral** $\underline{m+n}$.

Encoding Multiplication

$$\underline{m} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{m \text{ times}}\dots) \quad \underline{n} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{n \text{ times}}\dots)$$

$$\underline{m \times n} \triangleq \underbrace{(\lambda m\ .\ \lambda n\ .\ \lambda f\ .\ m\ (n\ f))}_{\times}\ \underline{m}\ \underline{n}$$

Each application of the f inside m is substituted for n applications of f, using the above λ-abstraction we get $m \times n$ applications of f.

Exponentiation

$$\underline{m} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{m \text{ times}}\dots) \quad \underline{n} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{n \text{ times}}\dots)$$

$$\underline{m^n} \triangleq \underbrace{(\lambda m\ .\ \lambda n\ .\ n\ m)}_{\text{exponential}}\ \underline{m}\ \underline{n}$$

Conditional

$$\underline{m} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{m \text{ times}}\dots)$$

$$\text{if } m = 0 \text{ then } x_1 \text{ else } x_2 \triangleq \underbrace{(\lambda m\ .\ \lambda x_1\ .\ \lambda x_2\ .\ m\ (\lambda z\ .\ x_2)\ x_1)}_{\text{if zero}}\ \underline{m}$$

If $\underline{m} = \underline{0} = \lambda f\ .\ \lambda x\ .\ x$ then x is returned, which will be x_1.

If not zero, then the f applied returns x_2, so any number of applications of f, results in x_2.

Successor

$$\underline{m} = \lambda f\ .\ \lambda x\ .\ \underbrace{f(\dots(f\ x)}_{m \text{ times}}\dots)$$

We simply take $\underline{m}$ and apply f one more time

$$\underline{m+1} \triangleq \underbrace{(\lambda m\ .\ \lambda f\ .\ \lambda x\ .\ f\ (m\ f\ x))}_{\text{succ}}\ \underline{m}$$

Pairs

We can encode pairs as a function, with a selector s function. Hence by supplying $first$ or $second$ as the selector, we can use the pair.

$$newpair(a, b) \triangleq \underbrace{(\lambda a\ .\ \lambda b\ .\ \lambda s\ .\ s\ a\ b)}_{\text{newpair}}\ a\ b \equiv \underbrace{(\lambda a\ b\ s\ .\ s\ a\ b)}_{\text{newpair}}\ a\ b$$

$$first(p) \triangleq p\ \underbrace{(\lambda x\ .\ \lambda y\ .\ x)}_{\text{first}} \equiv p\ \underbrace{(\lambda x\ y\ .\ x)}_{\text{first}}$$

$$second(p) \triangleq p\ \underbrace{(\lambda x\ .\ \lambda y\ .\ y)}_{\text{second}} \equiv p\ \underbrace{(\lambda x\ y\ .\ y)}_{\text{second}}$$

Predecessor

$$\underline{m} = \lambda f \ . \ \lambda x \ . \ \underbrace{f(\dots(f\ x)}_{m \text{ times}}\dots)$$

We cannot remove applications of f, however we can use a pair to count up until the successor is $\underline{m}$.

Hence we first need a function to get the next pair from the current:

$$\mathit{transition}\ p \triangleq \underbrace{(\lambda n\ .\ \mathit{newpair}\ (\mathit{second}\ n)\ ((\mathit{second}\ n) + 1))}_{\text{transition function}}\ p$$

We can then simply run the transition n times on a pair starting by using $f = \mathit{transition}$ and $x = \mathit{newpair}\ \underline{0}\ \underline{0}$.

$$\mathit{pred}(n) \triangleq \begin{cases} 0 & n = 0 \\ n - 1 & \mathit{otherwise} \end{cases}$$

$$\mathit{pred}(n) \triangleq \underbrace{(\lambda n\ .\ n\ \mathit{transition}\ (\mathit{newpair}\ \underline{0}\ \underline{0})\ \mathit{first})}_{\text{predecessor}}\ \underline{n}$$

A simpler definition of predecessor is:

$$\mathit{pred}(n) \triangleq \underbrace{(\lambda n\ .\ \lambda f\ .\ \lambda x\ .\ n\ (\lambda g\ .\ \lambda h\ .\ h\ (g\ f))\ (\lambda u\ .\ x)\ (\lambda u\ .\ u))}_{\text{predecessor}}\ \underline{n}$$

Subtraction

We can use the predecessor function for subtraction. By applying the predecessor function $\underline{n}$ times on some number $\underline{m}$ we get $\underline{m-n}$.

$$\underline{m-n} \triangleq \underbrace{(\lambda m\ .\ \lambda n\ .\ m\ \mathit{pred}\ n)}_{\text{subtract}}\ \underline{m}\ \underline{n}$$

Combinators

Definition: Combinator

A **closed** λ-term (no free variables), usually denoted by capital letters that describe

I	$\triangleq$	$\lambda x\ .\ x$	$I(x)$	$\triangleq$	x
K	$\triangleq$	$\lambda x\ y\ .\ x$	$K(x,y)$	$\triangleq$	x
S	$\triangleq$	$\lambda x\ y\ z\ .\ x\ z\ (y\ z)$	$S(x,y,z)$	$\triangleq$	$x(z)(y(z))$
T	$\triangleq$	$\lambda x\ y\ .\ y\ x$	$T(x,y)$	$\triangleq$	$y(x)$
C	$\triangleq$	$\lambda x\ y\ z\ .\ x\ z\ y$	$C(x,y,z)$	$\triangleq$	$x(z)(y)$
V	$\triangleq$	$\lambda x\ y\ z\ .\ z\ x\ y$	$V(x,y,z)$	$\triangleq$	$z(x)(y)$
B	$\triangleq$	$\lambda x\ y\ z\ .\ x\ (y\ z)$	$B(x,y,z)$	$\triangleq$	$x(y(z))$
B'	$\triangleq$	$\lambda x\ y\ z\ .\ y\ (x\ z)$	$B'(x,y,z)$	$\triangleq$	$y(x(z))$
W	$\triangleq$	$\lambda x\ y\ .\ x\ y\ y$	$W(x,y)$	$\triangleq$	$x(y)(y)$
Y	$\triangleq$	$\lambda g\ .\ (\lambda x\ .\ g\ (x\ x))\ (\lambda x\ .\ g\ (x\ x))$	$Y(f)$	$\triangleq$	$(\lambda x \to f(x(x)))(\lambda x \to f(x(x)))$

Only SKI are required to define any **computable function** (can remove even λ-abstraction, this is called SKI-**Combinator Calculus**).

The Y-Combinator is used for recursion. In one step of β-reduction:

$$Y\ f \to_\beta f\ (Y\ f)$$

We cannot define λ-terms in terms of themselves, as the λ-term is not yet defined, and infinitely large λ-terms are not allowed.

We can use the $Y - Combinator$ to create recursion in the absence of recursive λ-term definitions.

Definition: Fixed-Point Combinator

A higher order function (e.g fix) that returns some function of itself:

$$\begin{aligned} &fix\ f = f(fix\ f) \\ &fix\ f = f(f(\dots f(fix\ f)\dots))\ \text{(after repeated application)} \end{aligned}$$

Example: Factorial

$$fact(n) = \begin{cases} 1 & n = 0 \\ n \times fact(n-1) & otherwise \end{cases}$$

If recursive definitions for λ-terms were allows, we could express this as:

$$\begin{aligned} fact &\triangleq \lambda n\ .\ \text{if zero}\ n\ \underline{1}\ (multiply\ n\ (fact\ (pred\ n))) \\ &\triangleq (\lambda f\ .\ \lambda n\ .\ \text{if zero}\ n\ \underline{1}\ (multiply\ n\ (f\ (pred\ n))))\ fact \end{aligned}$$

Since we can use the above form (higher order function applied to itself) with the Y combinator.

$$fact \triangleq Y(\lambda f\ .\ \lambda n\ .\ \text{if zero}\ n\ \underline{1}\ (multiply\ n\ (f\ (pred\ n))))$$