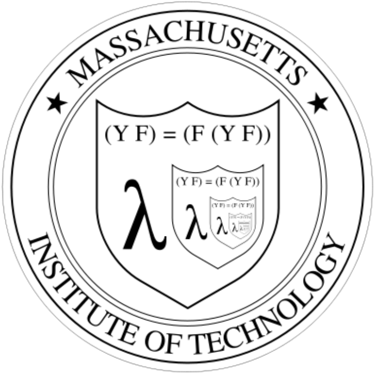
CIS 352

Names & Functions

A Second Attempt



Great Seal
of the
Knights of the λ-Calculus

Jim Royer

February 21, 2019

References

!!!! Slides for: *Language Semantics and Implementation Lectures 11 & 12*, by Richard Mayr and Colin Stirling, School of Informatics, University of Edinburgh, 2013. http://www.inf.ed.ac.uk/teaching/courses/lsi/13Lsi11-12.pdf from slide 9 onward

- Andrew Pitts' Lecture Notes on Semantics of Programming Languages: http://www.inf.ed.ac.uk/teaching/courses/lsi/sempl.pdf.
- *Semantics of programming languages Course Notes 2014-2015: Chapter 4, A simple functional language*, by Matthew Hennessy, Trinity College Dublin, University of Dublin, 2014. https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/LectureNotes/Notes14%20copy.pdf
- William Cook, Anatomy of Programming Languages, Chapter 3, http://www.cs.utexas.edu/~wcook/anatomy/anatomy.htm

---

✚ The Y in the equation $(Y\,F) = (F\,(Y\,F))$ is a *Y-combinator* (discussed later) is a program that builds programs. It was the inspiration of the well-known Y-Combinator start-up incubator (http://www.ycombinator.com) which is a business that builds businesses.

LFP = LC + λ + function application + variables

LFP Expressions

$$
\begin{aligned}
E ::= \; & n \mid b \mid \ell \mid E \; iop \; E \mid E \; cop \; E \mid \textbf{if } E \textbf{ then } E \textbf{ else } E \\
& \mid \,!E \mid E := E \mid \textbf{skip} \mid E;\, E \mid \textbf{while } E \textbf{ do } E \\
& \mid \underbrace{x \mid \lambda x.E \mid E\,E}_{\text{the } \lambda\text{-calculus}} \quad \big(\mid \textbf{let } X = E \textbf{ in } E \,\big)
\end{aligned}
$$

where

- $x \in \mathbb{V}$, an unlimited set of variables
- $n \in \mathbb{Z}$ (integers), $b \in \mathbb{B}$ (booleans), $\ell \in \mathbb{L}$ (locations)
- $iop \in$ (integer-valued binary operations: $+$, $-$, $*$, etc.)
- $cop \in$ (comparison operations: $==$, $<$, $\neq$, etc.)

$fv(E)$ = the set of *free variables* of LFP expression E

Definition:

$$\left.\begin{array}{r} fv(n),\; fv(b), \\ fv(\ell),\; fv(\mathbf{skip}) \end{array}\right\} = \emptyset.$$

$$fv(!E) = fv(E).$$

$$\left.\begin{array}{r} fv(E_1\, iop\, E_2),\; fv(E_1\, cop\, E_2), \\ fv(E_1 := E_2),\; fv(E_1;\; E_2), \\ fv(\mathbf{while}\ E_1\ \mathbf{do}\ E_2),\; fv(E_1\ E_2) \end{array}\right\} = fv(E_1) \cup fv(E_2).$$

$$fv(\mathbf{if}\ E_0\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2) = fv(E_0) \cup fv(E_1) \cup fv(E_2).$$

$$fv(x) = \{\, x \,\}.$$

$$fv(\lambda x.E) = fv(E) - \{\, x \,\}.$$

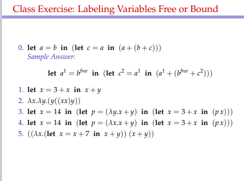
$$fv(\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2) = fv(E_1) \cup (fv(E_2) - \{\, x \,\})$$

Class Exercise: Labeling Variables Free or Bound

0. $\textbf{let}\ \ a = b\ \ \textbf{in}\ \ (\textbf{let}\ \ c = a\ \ \textbf{in}\ \ (a + (b + c)))$
   Sample Answer:

$$\textbf{let}\ \ a^1 = b^{free}\ \ \textbf{in}\ \ (\textbf{let}\ \ c^2 = a^1\ \ \textbf{in}\ \ (a^1 + (b^{free} + c^2)))$$

1. $\textbf{let}\ \ x = 3 + x\ \ \textbf{in}\ \ x + y$
2. $\lambda x.\lambda y.(y((xx)y))$
3. $\textbf{let}\ \ x = 14\ \ \textbf{in}\ \ (\textbf{let}\ \ p = (\lambda y.x + y)\ \ \textbf{in}\ \ (\textbf{let}\ \ x = 3 + x\ \ \textbf{in}\ \ (p\,x)))$
4. $\textbf{let}\ \ x = 14\ \ \textbf{in}\ \ (\textbf{let}\ \ p = (\lambda x.x + y)\ \ \textbf{in}\ \ (\textbf{let}\ \ x = 3 + x\ \ \textbf{in}\ \ (p\,x)))$
5. $((\lambda x.(\textbf{let}\ \ x = x + 7\ \ \textbf{in}\ \ x + y))\ (x + y))$



1. $\textbf{let } x^1 = 3 + x^{free} \textbf{ in } x^1 + y^{free}$
2. $\lambda x^1.\lambda y^2.(y^2((x^1 x^1) y^2))$
3. $\textbf{let } x^1 = 14 \textbf{ in } (\textbf{let } p^2 = (\lambda y^3.x^1 + y^3) \textbf{ in } (\textbf{let } x^4 = 3 + x^1 \textbf{ in } (p^2\ x^4)))$
4. $\textbf{let } x^1 = 14 \textbf{ in } (\textbf{let } p^2 = (\lambda x^3.x^3 + y^{free}) \textbf{ in } (\textbf{let } x^4 = 3 + x^1 \textbf{ in } (p^2\ x^4)))$
5. $((\lambda x^1.(\textbf{let } x^2 = x^1 + 7 \textbf{ in } x^2 + y^{free}))\ (x^{free} + y^{free}))$

Defining LFP substitution (the easy/boring cases)

$$V[P/x] = V \qquad (*)$$

$$(E_1 \ \mathsf{op}\ E_2)[P/x] = (E_1[P/x]) \ \mathsf{op}\ (E_2[P/x]) \qquad (\ddagger)$$

$$(\ell := E)[P/x] = (\ell := (E[P/x])$$

$$(C_1;\ C_2)[P/x] = (C_1[P/x]);\ (C_2[P/x])$$

$$(\mathbf{while}\ B\ \mathbf{do}\ C)[P/x] = \mathbf{while}\ (B[P/x])\ \mathbf{do}\ (C[P/x])$$

$$(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2)[P/x] = \mathbf{if}\ (B[P/x])\ \mathbf{then}\ (C_1[P/x])\ \mathbf{else}\ (C_2[P/x])$$

$$(E_1\ E_2)[P/x] = (E_1[P/x])\ E_2[P/x])$$

$(*)$ V is a number, boolean value, location, or a **skip**.

$(\ddagger)$ $\mathsf{op} = +, -, *, \leq, \ldots$

Defining LFP substitution (the harder cases)

$$y[P/x] = \begin{cases} P, & \text{if } x = y \\ y, & \text{if } x \neq y \end{cases}$$

$$(\lambda y.P')[P/x] = \begin{cases} (\lambda y.P'), & \text{if } x = y; \\ (\lambda z.P'''), & \text{o/w, where } (*) \end{cases}$$

$$(\mathbf{let}\ y = P_1\ \ \mathbf{in}\ \ P_2)[P/x] = \begin{cases} \mathbf{let}\ y = (P_1[P/x])\ \mathbf{in}\ P_2, & \text{if } x = y; \\ \mathbf{let}\ z = (P_1[P/x])\ \mathbf{in}\ P_2'', & \text{o/w, where } (\dagger) \end{cases}$$

$(*)$ $z \notin (\mathit{freeVars}(P) \cup \mathit{freeVars}(P') \cup \{\, x \,\})$
$P'' = P'[z/y] \qquad P''' = P''[P/x]$

$(\dagger)$ $z \notin (\mathit{freeVars}(P) \cup \mathit{freeVars}(P_2) \cup \{\, x \,\})$
$P_2' = P_2[z/y] \qquad P_2'' = P_2'[P/x]$

(*Why all the fuss?*)

Recall: Capturing a variable (in C)

```
#define INCI(i) { int a=0; ++i; }
int main(void)
{
    int a = 0, b = 0;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d", a, b);
    return 0;
}
```

Running the above through the C preprocessor produces:

```
int main(void)
{
    int a = 0, b = 0;
    { int a=0; ++a; };
    { int a=0; ++b; };
    printf("a is now %d, b is now %d", a, b);
    return 0;
}
```

Class Exercise: Substitutions

(a) $(z\ y)[(t\ v)/y]$

(b) $(z\ y)[(t\ v)/w]$

(c) $((z\ y)\ z)[(y\ z)/y]$

(d) $(\lambda y.(z\ y))[(t\ v)/y]$

(e) $(\lambda t.(z\ y))[(tv)/y]$

(f) $(\lambda z.(x\ y))[(\lambda x.x)/y]$

(g) $((\lambda t.(u\ t))(\lambda w.(t\ w)))[(t\ u)/u]$

(h) $((\lambda y.(\lambda z.(w\ z)))(\lambda x.(y\ (w\ x))))[(x\ (y\ z))/w]$

└─Class Exercise: Substitutions

Class Exercise: Substitutions

(a) $(z\ y)[(t\ v)/y]$
(b) $(z\ y)[(t\ v)/w]$
(c) $((z\ y)\ z)[(y\ z)/y]$
(d) $(\lambda y.(z\ y))[(t\ v)/y]$
(e) $(\lambda t.(z\ y))[(tv)/y]$
(f) $(\lambda z.(x\ y))[(\lambda x.x)/y]$
(g) $((\lambda t.(u\ t))(\lambda w.(t\ w)))[(t\ u)/u]$
(h) $((\lambda y.(\lambda z.(w\ z)))(\lambda x.(y\ (w\ x))))[(x\ (y\ z))/w]$

(a) $(z\ y)[(t\ v)/y]$
answer: $(z\ (t\ v))$

(b) $(z\ y)[(t\ v)/w]$
answer: $(z\ y)$

(c) $((z\ y)\ z)[(y\ z)/y]$
answer: $((z\ (y\ z))\ z)$

(d) $(\lambda y.(z\ y))[(t\ v)/y]$
answer: $(\lambda y.(z\ y))$

(e) $(\lambda t.(z\ y))[(tv)/y]$
answer: $(\lambda a.(z\ (t\ v)))$

(f) $(\lambda z.(x\ y))[(\lambda x.x)/y]$
answer: $(\lambda z.(x(\lambda x.x)))$

(g) $((\lambda t.(u\ t))(\lambda w.(t\ w)))[(t\ u)/u]$
answer: $((\lambda a.((t\ u)\ a))(\lambda w.(t\ w)))$

(h) $((\lambda y.(\lambda z.(w\ z)))(\lambda x.(y\ (w\ x))))[(x\ (y\ z))/w]$
answer: $((\lambda a.(\lambda b.((x\ (y\ z))\ b)))(\lambda c.(y\ ((x\ (y\ z))\ c))))$

A big-step semantics for LFP, 1

The hold-overs from LC have the same rules as before:

$\Downarrow$-*Values:* $$\frac{}{\langle V, s \rangle \Downarrow \langle V, s \rangle} \quad (V \text{ is a value})$$

$\Downarrow$-$\circledast$: $$\frac{\langle E_1, s \rangle \Downarrow \langle n_1, s' \rangle \quad \langle E_2, s' \rangle \Downarrow \langle n_2, s'' \rangle}{\langle E_1 \circledast E_2, s \rangle \Downarrow \langle c, s'' \rangle} \quad (c = n_1 \circledast n_2)$$

$$\vdots$$

Values consist of numbers, `tt`, `ff`, locations, **skip**, *and* λ-expressions.

A big-step semantics for LFP, 2

For function application we have two choices:

Call by name

$$\Downarrow\text{-}cbn\!: \quad \frac{\langle E_1, s\rangle \Downarrow \langle \lambda x.E_1', s'\rangle \quad \langle E_1'[E_2/x], s'\rangle \Downarrow \langle V, s''\rangle}{\langle (E_1\ E_2), s\rangle \Downarrow \langle V, s''\rangle}$$

Call by value

$$\Downarrow\text{-}cbv\!: \quad \frac{\langle E_1, s\rangle \Downarrow \langle \lambda x.E_1', s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle V_2, s''\rangle \quad \langle E_1'[V_2/x], s''\rangle \Downarrow \langle V, s'''\rangle}{\langle (E_1\ E_2), s\rangle \Downarrow \langle V, s'''\rangle}$$

- $\Downarrow_{\mathsf{N}}$, the call-by-name evaluation relation
- $\Downarrow_{\mathsf{V}}$, the call-by-value evaluation relation
- $\langle E, s\rangle \not\Downarrow_{\mathsf{N}}$ means **not** $(\exists\langle V, s'\rangle)[\langle E, s\rangle \Downarrow_{\mathsf{N}} \langle V.s'\rangle]$
- $\langle E, s\rangle \not\Downarrow_{\mathsf{V}}$ means **not** $(\exists\langle V, s'\rangle)[\langle E, s\rangle \Downarrow_{\mathsf{V}} \langle V.s'\rangle]$

(We handle **let** *later.)*

Call-by-name and call-by-value are incompatable

Let:

$$\begin{aligned} \mathit{Boom} &=_{\mathit{def}} \textbf{while true do skip} \\ C_1 &=_{\mathit{def}} (\lambda x.\ \textbf{skip})\ \mathit{Boom} \\ C_2 &=_{\mathit{def}} (\lambda x.\ \textbf{if } !\ell = 0\ \ \textbf{then skip else } \mathit{Boom})(\ell := 0) \end{aligned}$$

Then:

$$\langle C_1, s \rangle \Downarrow_{\mathsf{N}} \langle \textbf{skip}, s \rangle \qquad \text{(for any } s\text{)}$$

$$\langle C_1, s \rangle \not\Downarrow_{\mathsf{V}}$$

$$\langle C_2, \{ \ell \mapsto 1 \} \rangle \not\Downarrow_{\mathsf{N}}$$

$$\langle C_2, \{ \ell \mapsto 1 \} \rangle \Downarrow_{\mathsf{V}} \langle \textbf{skip}, \{ \ell \mapsto 0 \} \rangle$$

(We'll do a closer comparison when we look at environment models for evaluation.)

Recursion, 1

What goes wrong?

$$\begin{array}{l}\textbf{let}\, f = \lambda n.\ \textbf{if}\ n \leq 0\ \textbf{then}\ 1\ \textbf{else}\ n * (f\ (n-1)) \\ \quad \textbf{in}\ (f\ 4)\end{array}$$

Recursion, 1

What goes wrong?

$$\textbf{let } f = \lambda n.\ \textbf{if } n \leq 0 \textbf{ then } 1 \textbf{ else } n * (f\ (n-1))$$
$$\quad \textbf{in } (f\ 4)$$

$$\textbf{let } f^1 = \lambda n^2.\ \textbf{if } n^2 \leq 0 \textbf{ then } 1 \textbf{ else } n^2 * (f^{free}\ (n^2 - 1))$$
$$\quad \textbf{in } (f^1\ 4)$$

Recursion, 2

LFP$^+$ = LFP + a recursion operator

$$E ::= \ldots \mid \mathbf{rec}\ x.E$$

Informally: "$\mathbf{rec}\ x.E$" reads *recursively define x to be E*.

The big-step operational semantics is given by:

$$\textit{unfolding:}\quad \frac{\langle E[(\mathbf{rec}\ x.E)/x], s \rangle \Downarrow \langle V, s' \rangle}{\langle \mathbf{rec}\ x.E, s \rangle \Downarrow \langle V, s' \rangle}$$

Recursion, 3

Examples:

- **rec** $x.x$

  Try: **rec** $x.x$

- **rec** $f.(\lambda x.\ \textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * f(x-1))$

  Try: $((\textbf{rec}\ f.(\lambda x.\ \textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * f(x-1)))\ 2)$

- **rec** $z.(\textbf{if}\ E\ \textbf{then}\ (E';\ z)\ \textbf{else skip})$

  Try: $\langle\, \textbf{rec}\ z.(\textbf{if}\ !\ell > 0\ \textbf{then}\ (\ell := !\ell - 1;\ z)\ \textbf{else skip}), \{\, \ell \mapsto 2 \,\}\, \rangle$

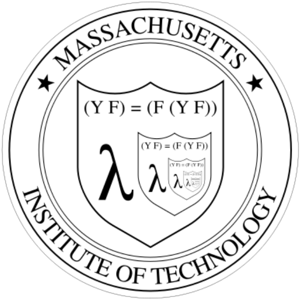
rec is an example of a fixed point combinator.

Haskell Curry's Y:	$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.
Alan Turing's (cbn):	$(\lambda x.\lambda y.(y(xxy)))(\lambda x.\lambda y.(y(xxy)))$
Alan Turing's (cbv):	$(\lambda x.\lambda y.(y(\lambda z.xxyz)))(\lambda x.\lambda y.(y(\lambda z.xxyz)))$

Digression on Y

$Y = \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$ *(requires call-by-name)*

$$
\begin{aligned}
Y\,g &= (\lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x)))\,g \\
&= (\lambda x.g(x\,x))(\lambda x.g(x\,x)) \\
&= (\lambda y.g(y\,y))(\lambda x.g(x\,x)) \\
&= g((\lambda x.g(x\,x))(\lambda x.g(x\,x))) \\
&= g(Y\,g)
\end{aligned}
$$



- For other fixed point combinators, see: `http://en.wikipedia.org/wiki/Fixed-point_combinator#Other_fixed-point_combinators`
- The key point: There are all sorts of recursions hiding in the (untyped) λ-calculus: $E ::= x \mid \lambda x.E \mid (E\,E')$.

LFP$^+$'s properties and problems

- Under call-by-value, LFP$^+$ expressions can have side-effects.

$$\langle\, (\lambda x.0)\ (\ell := 1), \{\, \ell \mapsto 0 \,\} \,\rangle \Downarrow_{\mathsf{V}} \langle\, 0, \{\, \ell \mapsto 1 \,\} \,\rangle.$$

[Define side-effect]

- However, under call-by-name

$$\langle\, (\lambda x.0)\ (\ell := 1), \{\, \ell \mapsto 0 \,\} \,\rangle \Downarrow_{\mathsf{N}} \langle\, 0, \{\, \ell \mapsto 0 \,\} \,\rangle.$$

- Under call-by-name, there are no side-effecting integer expression. *(We need to define what the integer expressions are to nail this down.)*
- LFP$^+$ is determinate under both call-by-name and call-by-value. [Spell this out]
- Subject reduction (i.e., a type-τ expression evaluates to a type-τ value) holds, but requires a typed versions of LFP$^+$.