CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

Lecture 6— Lambda Calculus

Where we are

- Done: Modeling mutation and local control-flow
- Proving operational equivalence (hard!)
- Now: Didn't IMP leave some things out?

Time for a new model... (Pierce, chapter 5)

What we forgot

IMP is missing lots of things (threads, I/O, exceptions, strings, ...), but here are two really basic and interesting ones:

- Scope (all global variables, no functions or objects)
- Data structures (only integers, no records or pairs)

As we'll see, higher-order functions do both! (E.g., my heap implementation in hw1 used functions like a list)

Adding data structures

Extending IMP with data structures isn't too hard:

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid e+e \mid e*e \mid (e,e) \mid e.1 \mid e.2 \\
v & ::= & c \mid (v,v) \\
H & ::= & \cdot \mid H, x \mapsto v
\end{array}
$$

$$
\frac{H;e_1 \Downarrow v_1 \quad H;e_2 \Downarrow v_2}{H;(e_1,e_2) \Downarrow (v_1,v_2)} \qquad \frac{H;e \Downarrow (v_1,v_2)}{H;e.1 \Downarrow v_1} \qquad \frac{H;e \Downarrow (v_1,v_2)}{H;e.2 \Downarrow v_2}
$$

Note: We allow pairs of values, not just pairs of integers.

Note: We now have *stuck* programs (e.g., $c.1$) – what would C do? Scheme? ML? Java?

What about functions

But adding functions (or objects) does not work well:

$$e ::= \ldots \mid \texttt{fun}\ x \texttt{ -> } s \qquad s ::= \ldots \mid e(e)$$

$$\frac{}{H; \texttt{fun}\ x \texttt{ -> } s \Downarrow \texttt{fun}\ x \texttt{ -> } s}$$

$$\frac{H; e_1 \Downarrow \texttt{fun}\ x \texttt{ -> } s \qquad H; e_2 \Downarrow v}{H\ ;\ e_1(e_2) \rightarrow H\ ;\ x := v; s}$$

NO: Consider $(\texttt{fun}\ x \texttt{ -> } s)(y := x); ans := x$.

Scope matters; variable name doesn't.

Another try

$$\frac{H;e_1 \Downarrow \texttt{fun}\ x\ \texttt{->}\ s \qquad H;e_2 \Downarrow v \qquad y \text{ “fresh”}}{H\ ;\ e_1(e_2) \rightarrow H\ ;\ y := x; x := v; s; x := y}$$

- “fresh” is sloppy
- not a good match to how functions are implemented (round-peg, square-hole)
- yuck
- NO: wrong model for most functional and OO languages (okay for C, I think)

The wrong model

$$\frac{H;e_1 \Downarrow \texttt{fun}\ x\ \texttt{->}\ s \qquad H;e_2 \Downarrow v \qquad y\ \text{“fresh”}}{H\ ;\ e_1(e_2) \rightarrow H\ ;\ y := x; x := v; s; x := y}$$

$f_1 := \texttt{fun}\ x\ \texttt{->}\ f_2 := \texttt{fun}\ z\ \texttt{->}\ y := x + z;$

$f_1(2);$

$x := 3$

$f_2(4)$

```
let f1 = fun x -> (fun z -> x + z) in
let f2 = f1 2 in
let x  = 3 in
f2 4
```

Punch line

The way higher-order functions and objects work is not modeled by mutable global variables. So let's build a new model that focuses on this essential concept.

(Or just borrow a model from Alonzo Church.)

And drop mutation, conditionals, integers (!), and loops (!)

The Lambda Calculus:

$$\begin{array}{rcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \\ v & ::= & \lambda x.\ e \end{array}$$

You *apply* a function by *substituting* the argument for the *bound variable*.

Example Substitutions

$$\begin{aligned} e &::= \lambda x.\ e \mid x \mid e\ e \\ v &::= \lambda x.\ e \end{aligned}$$

Substitution is the key operation we were missing:

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

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

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

After substitution, the bound variable is gone, so its "name" was irrelevant. (Good!)

There are *irreducible* expressions ($x\ e$). (maybe a problem)

A Programming Language

Given substitution ($e_1[e_2/x]$), we can give a semantics:

$$\frac{}{(\lambda x.\ e)\ v \rightarrow e[v/x]} \qquad \frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v\ e_2 \rightarrow v\ e_2'}$$

A small-step, *call-by-value (CBV)*, left-to-right semantics

Gets stuck exactly when there's a *free variable* at top-level
(Won't cheat because scope is what we're interested in)

This is the "heart" of functional languages like O'Caml
(but "real" implementations don't substitute—see hw3)

Where are we

- Motivation for a new model (done)
- CBV lambda calculus using substitution (done)
- Notes on concrete syntax
- Simple Lambda encodings
- Other reduction strategies
- Defining substitution

Syntax Revisited

We resolve concrete-syntax ambiguities as follows:

1. $\lambda x.\ e_1\ e_2$ is $(\lambda x.\ e_1\ e_2)$, not $(\lambda x.\ e_1)\ e_2$
2. $e_1\ e_2\ e_3$ is $(e_1\ e_2)\ e_3$, not $e_1\ (e_2\ e_3)$
   (Convince yourself application is not associative)

More generally:

1. Function bodies extend to an unmatched right parenthesis (e.g., $(\lambda x.\ y(\lambda z.\ z)w)q)$
2. Application associates to the left. E.g., $e_1\ e_2\ e_3\ e_4$ is $(((e_1\ e_2)\ e_3)\ e_4)$.

These strange-at-first rules are convenient (see O'Caml)

Simple encodings

It's fairly crazy we left out constants, conditionals, primitives, and data structures.

In fact, we're *Turing complete* and can *encode* whatever we need. Motivation for encodings:

- It's fun and mind-expanding
- It shows we aren't oversimplifying the model ("numbers are syntactic sugar")
- It can show languages are *too expressive* (e.g., unlimited C++ template instantiation)

Encoding booleans

There are two booleans and one conditional expression. The conditional takes 3 arguments (via currying). If the first is one boolean it evaluates to the second. If it's the other boolean it evaluates to the third.

Any 3 expressions meeting this specification (of "the boolean ADT") is an encoding of booleans.

"true" $\boldsymbol{\lambda x.\ \lambda y.\ x}$

"false" $\boldsymbol{\lambda x.\ \lambda y.\ y}$

"if" $\boldsymbol{\lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f}$

This is just one encoding. E.g.: "if" "true" $\boldsymbol{v_1\ v_2 \rightarrow^* v_1}$.

Evaluation Order Matters

Careful: With CBV we need to "thunk"...

"if" "true" $(\lambda x.\ x)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

diverges, but

$($"if" "true" $(\lambda z.\ \lambda x.\ x)\ (\lambda z.\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))))$
v

doesn't.

Encoding pairs

The "pair ADT" has a constructor taking two arguments and two selectors. The first selector returns the first argument passed to the constructor and the second selector returns the second.

"mkpair" $\boldsymbol{\lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y}$

"fst" $\boldsymbol{\lambda p.\ p(\lambda x.\ \lambda y.\ x)}$

"snd" $\boldsymbol{\lambda p.\ p(\lambda x.\ \lambda y.\ y)}$

Example:

"snd" ("fst" ("mkpair" ("mkpair" $\boldsymbol{v_1\ v_2}$) $\boldsymbol{v_3}$)) $\longrightarrow^* \boldsymbol{v_2}$

More encodings (see the text)

- "Church numerals": A natural number is $\mathbf{0}$ or the successor of a natural (and maybe a test for equality) Given the encoding, can define math operations (plus, exponentiation, ...)
- "lists": in an untyped world, booleans and pairs are enough for lists, one approach:
  - Empty list is "mkpair" "false" "false"
  - Non-empty list is "mkpair" "true" ("mkpair" $\boldsymbol{h}$ $\boldsymbol{t}$)

  (Not too far from how lists are implemented.)
- "recursion": useful looping by applying a function to itself and an argument (too intricate for lecture)

Reduction "Strategies"

Suppose we allowed any substitution to take place in any order:

$$\frac{}{(\boldsymbol{\lambda x.\ e})\ \boldsymbol{e'} \rightarrow \boldsymbol{e[e'/x]}} \qquad \frac{\boldsymbol{e_1 \rightarrow e_1'}}{\boldsymbol{e_1\ e_2 \rightarrow e_1'\ e_2}}$$

$$\frac{\boldsymbol{e_2 \rightarrow e_2'}}{\boldsymbol{e_1\ e_2 \rightarrow e_1\ e_2'}} \qquad \frac{\boldsymbol{e \rightarrow e'}}{\boldsymbol{\lambda x.\ e \rightarrow \lambda x.\ e'}}$$

Programming languages don't typically do this, but it has uses. . .

Full Reduction

- Prove programs equivalent *algebraically* (also use $\lambda x.\ e \rightarrow \lambda y.\ e'$ where e' is e with free x replaced with y and $\lambda x.\ e\ x \rightarrow e$ where x not free in e)
- Optimize/pessimize/partially-evaluate (even avoid an infinite loop)

What order you reduce is a "strategy"; equivalence is undecidable

Non-obvious fact ("Church-Rosser"): In this pure calculus, if $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$, then there exists an e_3 such that $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$.

"No strategy gets painted into a corner"

Some other common strategies

We have seen "full" and left-to-right CBV.

(O'Caml is unspecified order, but actually right-to-left.)

Claim: Without assignment, I/O, exceptions, ... you cannot distinguish evaluation order.

Another option is call-by-name (CBN):

$$\frac{}{(\lambda x.\ e)\ e' \rightarrow e[e'/x]} \qquad \frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}$$

Even "smaller" than CBV!

Diverges strictly less often than CBV, e.g., $(\lambda y.\ \lambda z.\ z)e$.
Can be faster (fewer steps), but not usually (reuse args).

More on evaluation order

In "purely functional" code, evaluation order "only" matters for performance and termination.

Example: Imagine CBV for conditionals!

```
let rec f n = if n=0 then 1 else n*(f (n-1))
```

Call-by-need or "lazy evaluation": "Best of both worlds"? (E.g.: Haskell) Evaluate the argument the first time it's used. Memoize the result. (Useful idiom for coders too.)

Can be formalized, but it's not pretty.

For purely functional code, total equivalence with CBN and same asymptotic time as CBV. (Note: *asymptotic*!) Hard to reason about with effects.

Formalism not done yet

For rest of course, assume CBV:

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

Need to define substitution—shockingly subtle.

Attempt 1:

$$\frac{}{x[e/x] = e} \qquad \frac{y \neq x}{y[e/x] = y} \qquad \frac{e_1[e/x] = e_1'}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'}$$

$$\frac{e_1[e/x] = e_1' \qquad e_2[e/x] = e_2'}{(e_1\ e_2)[e/x] = e_1'\ e_2'}$$

Getting substitution right

Attempt 2:

$$\frac{e_1[e/x] = e_1' \qquad y \neq x}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'} \qquad \frac{}{(\lambda x.\ e_1)[e/x] = \lambda x.\ e_1}$$

What if e is y or $\lambda z.\ y$ or, in general y is *free* in e? This *mistake* is called *capture*.

It doesn't happen under CBV/CBN *if* our source program has *no free variables*.

Can happen under full reduction.

Another Try

Attempt 3 (define $FV(e)$ inductively on e):

$$\frac{e_1[e/x] = e_1' \qquad y \neq x \qquad y \notin FV(e)}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'}$$

$$\frac{}{(\lambda x.\ e_1)[e/x] = \lambda x.\ e_1}$$

A *partial* definition because of the *syntactic accident* that y was used as a binder (should not be visible).

So we allow *implicit systematic renaming* (of a binding and all its bound occurrences). So the top rule can always apply.

In general, we *never* distinguish terms that differ only in the names of variables. (A key language-design principle!)

Summary and some jargon

- If everything is a function, every step is an application: $(\lambda x.\ e)e' \to e[e'/x])$ (called β-reduction)
- Substitution avoids capture via implicit renaming (called α-conversion)
- With full reduction, $\lambda x.\ e\ x \to e$ makes sense if $x \notin FV(e)$ (called η-reduction), for CBV it's "unthunking"

Most languages use CBV application, some use call-by-need.

Our Turing-complete language models functions and encodes everything else.

Where we're going

You should now have a better understanding of higher-order functions, functional languages, and evaluation order.

Which will make you a better programmer in all languages.

But the untyped lambda-calculus is like an "assembly language" where everything looks the same: closed programs always do something and everything is a function (cf. bits).

Next: We'll distinguish constants from functions and use a *type system* to prevent run-time errors.