CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2005

Lecture 6— Lambda Calculus

Where we are

- Done: Syntax, semantics, and equivalence for loops and global integer variables
- 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!

- Scope: not all memory available to all code
- Data: For example, 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:

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

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

Does this match "the semantics we want" for function calls?

What about functions

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

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

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

NO: Consider `x := 1; (fun x -> y := x)(2); ans := x`.

Scope matters; variable name doesn't. That is:

- Local variables should "be local"
- Choice of local-variable names should have only local ramifications

Another try

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

- "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 (even wrong for C if s calls another function that accesses x)

The wrong model

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

$\texttt{f}_1 := (\texttt{fun x -> f}_2 := (\texttt{fun z -> ans} := \texttt{x} + \texttt{z}));$

$\texttt{f}_1(2);$

$\texttt{x} := 3;$

$\texttt{f}_2(4)$

“Should” set ans to 6:

- $\texttt{f}_1(2)$ should assign to $\texttt{f}_2$ a function that adds 2 to its argument and stores result in ans.

“Actually” sets ans to 7:

- $\texttt{f}_2(2)$ assigns to $\texttt{f}_2$ a function that adds *the current value of* x to its argument.

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:

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

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

Example Substitutions

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

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:

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

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 Caml (but "real" implementations don't substitute; they do something *equivalent*)

Where are we

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

Syntax Revisited

We (and Caml) resolve concrete-syntax ambiguities as follows:

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

More generally:

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

- These strange-at-first rules are convenient
- Like in IMP, we really have trees (with non-leaves labeled $\boldsymbol{\lambda}$ or "application")

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)

Encodings are also just "(re)definition via translation"

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" $\boldsymbol{v_1\ v_2} \longrightarrow^* \boldsymbol{v_1}$.

Evaluation Order Matters

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

"if" "true" $(\lambda x.\ x)\ \underbrace{((\lambda x.\ x\ x)(\lambda x.\ x\ x))}_{\text{an infinite loop}}$

diverges, but

("if" "true" $\lambda x.\ x)\ \underbrace{(\lambda z.\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))z))}_{\text{a value that when called diverges}}$

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" $\boldsymbol{v_1\ v_2}$) $\boldsymbol{v_3}$)) $\boldsymbol{\longrightarrow^*\ v_2}$

Encoding lists

Rather than start from scratch, notice that booleans and pairs are enough:

- Empty list is "mkpair" "false" "false"
- Non-empty list is "mkpair" "true" ("mkpair" h t)
- Is-empty is ...
- Head is ...
- Tail is ...

(Not too far from how lists are implemented.)

Encoding natural numbers

Known as "Church numerals" — see the text (or don't bother).

We can define the naturals as "zero", a "successor" function, an "is equal" function, a "plus" function, etc.

The encoding is correct if "is equal" always returns what it should, e.g., is-equal (plus (succ zero) (succ zero)) (succ(succ zero))
should evaluate to "true"

Recursion

We have programs diverge, but can we write useful loops? Yes!

To write a recursive function:

- Write a function that takes an f and calls it in place of recursion
  - Example (in enriched language):
    $\lambda f.\ \lambda x.\ \textbf{if}\ (x = 0)\ \textbf{then}\ 1\ \textbf{else}\ (x * f(x-1))$
- Then apply "fix" to it to get a recursive function:
  - "fix" $\lambda f.\ \lambda x.\ \textbf{if}\ (x = 0)\ \textbf{then}\ 1\ \textbf{else}\ (x * f(x-1))$
- "fix" $\lambda f.\ e$ will reduce to *something roughly equivalent to* $\lambda f.\ e[(\text{"fix"}\lambda f.\ e)/f]$, which is "unrolling the recursion once" (and further unrollings will happen as necessary).
- The details, especially for CBV, are icky; the point is it's possible and you define "fix" only once
- Not on exam: "fix" $\lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))(\lambda x.\ f\ (\lambda y.\ x\ x\ y))$

Reduction "Strategies"

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

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

$$\frac{e \longrightarrow e'}{\lambda x.\ e \longrightarrow \lambda x.\ e'}$$

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

Full Reduction

- Prove programs equivalent *algebraically* (also use $\boldsymbol{\lambda x.\ e \longrightarrow \lambda y.\ e'}$ where $\boldsymbol{e'}$ is $\boldsymbol{e}$ with free $\boldsymbol{x}$ replaced with $\boldsymbol{y}$ and $\boldsymbol{\lambda x.\ e\ x \longrightarrow e}$ where $\boldsymbol{x}$ not free in $\boldsymbol{e}$)
- 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 $\boldsymbol{e \longrightarrow^* e_1}$ and $\boldsymbol{e \longrightarrow^* e_2}$, then there exists an $\boldsymbol{e_3}$ such that $\boldsymbol{e_1 \longrightarrow^* e_3}$ and $\boldsymbol{e_2 \longrightarrow^* e_3}$.

"No strategy gets painted into a corner"

Some other common semantics

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

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

Claim: Without assignment, I/O, exceptions, ... you cannot distinguish left-to-right CBV from right-to-left CBV.

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

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

Even "smaller" than CBV!

Diverges strictly less often than CBV, e.g., $(\boldsymbol{\lambda y.}\ \boldsymbol{\lambda z.}\ z)e$. 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.