CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2008

Lecture 6— Lambda Calculus

Where we are

- Done: Syntax, semantics, and equivalence
  - As long as all you have is loops and global variables
- Now: Didn't IMP leave some things out?
  - Particularly scope, functions, and data structures
  - (Not to mention threads, I/O, exceptions, strings, ...)

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

Data + Code

Higher-order functions work well for scope *and* data structures.

- Scope: not all memory available to all code

```
let x = 1
let add3 y =
   let z = 2 in
   x + y + z
let seven = add3 4
```

- Data: Function closures store data. Example: Association "list"

```
let empty = (fun k -> raise Empty)
let cons k v lst = (fun k' -> if k'=k then v else lst k')
let lookup k lst = lst k
```

(Later: Objects do both too)

Adding data structures

Extending IMP with data structures isn't too hard:

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

$\boxed{H;e \Downarrow c}$

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

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? Perl?

Note: Division also causes stuckness

What about functions

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

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

$$s \ ::= \ \ldots \mid e(e)$$

$\boxed{H ; e \Downarrow c}$ $\boxed{H \ ; \ s \rightarrow H' \ ; \ s'}$

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

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

What about functions

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

$$e \;::=\; \ldots \mid \texttt{fun}\ x\ \texttt{->}\ s$$

$$s \;::=\; \ldots \mid e(e)$$

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

NO: Consider $\texttt{x} := 1; (\texttt{fun x -> y} := \texttt{x})(2); \texttt{ans} := \texttt{x}$.

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

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

- “fresh” isn't very IMP-like but okay (think malloc)
- not a good match to how functions are implemented
- 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

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

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

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

$\texttt{x} := 3;$

$\texttt{f}_2(4)$

"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 (can add other IMP features back later).

(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*.

(There's an equivalent *environment* definition not unlike heap-copying; see future homework.)

Example Substitutions

$$e ::= \lambda x.\ e \mid x \mid e\ e$$

$$v ::= \lambda x.\ e$$

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 ($x\ e$)

A Programming Language

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

$$\boxed{e \rightarrow e'}$$

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

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

- Terminates when the "whole program" is some $\lambda x.\ e$

But (also) gets stuck when there's a *free variable* "at top-level"
(Won't "cheat" like we did with $H(x)$ in IMP 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. $\lambda x.\ e_1\ e_2$ 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
   Example: $(\lambda x.\ y(\lambda z.\ z)w)q$
2. Application associates to the left
   Example: $e_1\ e_2\ e_3\ e_4$ is $(((e_1\ e_2)\ e_3)\ e_4)$.

- These strange-at-first rules are convenient
- Like in IMP, we really have trees
  (with non-leaves labeled λ or "application")

Simple encodings

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:

- Fun and mind-expanding
- Shows we aren't oversimplifying the model ("numbers are syntactic sugar")
- 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" $\lambda x.\ \lambda y.\ x$

"false" $\lambda x.\ \lambda y.\ y$

"if" $\lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$

This is just one encoding.

E.g.: "if" "true" $v_1\ v_2 \rightarrow^* v_1$.

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

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

Some 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* $e[(\text{"fix"}\lambda f.\ e)/f]$, 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" $\lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))(\lambda x.\ f\ (\lambda y.\ x\ x\ y))$

Where are we

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

Reduction "Strategies"

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

$\boxed{e \to e'}$

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

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

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

- Optimize/pessimize/partially evaluate programs
- Prove programs equivalent by reducing them to the same term

Church-Rosser

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

Non-obvious fact ("Confluence" or "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"

- Useful: No rewriting via the full-reduction rules prevents you from getting an answer (Wow!)

Any *rewriting system* with this property is said to, "have the Church-Rosser property."

Some more equivalences

We can add two more rewritings:

- Replace $\lambda x.\ e$ with $\lambda y.\ e'$ where e' is e with "free" x replaced with y.
- Replace $\lambda x.\ e\ x$ with e if x does not occur "free" in e.

With these, plus full reduction, plus "letting rules run either direction" we have a "complete" rewriting system for equivalence.

- Under the accepted denotational semantics (not in 505), two expressions denote the same thing if and only if this rewriting system can turn one into the other. (Wow!)

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):

$$\boxed{e \rightarrow e'}$$

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

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 if language has side-effects.