CSE 505 : Graduate Programming Languages

Lecture 8 : Lambda Calculus

So Far

- syntax, semantics, equivalence
  - all limited to IMP = loops + globals
- What is IMP missing?
  - scope, functions, data structures
  - (threads, I/O, exceptions, strings, ...)
- Let's look at expanding IMP

DATA + CODE

With higher order functions, we get both scope and data structures.

▷ Scope : not all ~~data~~ memory available to all code

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

let seven = add3 4
```

▷ Data : closures store data, e.g. alist (association list)

```
let empty = fun k -> raise Empty
let cons k v l = fun k' -> if k' = k then v else l k
let lkup k l = l k
```

Data Structures for IMP

Not so bad ...

$$e ::= c \mid x \mid e + e \mid e * e \mid (e, e) \mid e.1 \mid e.2$$

$$v ::= c \mid (v, v)$$

$$H ::= \cdot \mid H, x \to v$$

(no change to s)

$\boxed{H ; e \Downarrow c}$ ... all old stuff plus

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

(.2 similar)

Note:

- Pairs of values, not just pairs of ints!
- Can have "stuck" exp or stmt, e.g. $c.1$
  - What would C++, Scheme, Java, ML, Perl, do?
  - Division can also cause "stuckness"

Functions for IMP...?

Gets ugly fast!

$$e ::= \ldots \mid \text{fun } x \to s$$
$$v ::= \ldots \mid \text{fun } x \to s$$
$$s ::= \ldots \mid e(e)$$

woah! mutual dependence... Is that ok?

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

$$\boxed{H; s \to H'; s'}$$

... +

$$\frac{}{H; \text{fun } x \to s \Downarrow \text{fun } x \to s}$$

$$\frac{H; e_1 \Downarrow \text{fun } x \to s \qquad H; e_2 \Downarrow v}{H; e_1(e_2) \to H; x := v; s}$$

Sweet! Does this match our intuition?

... Uhh... no...

what about:

$$\left.\begin{array}{l} x := 1; \\ (\text{fun } x \to y := x)(2); \\ \text{ans} := x \end{array}\right\} \begin{array}{l} \text{yields } 2! \\ \text{we want } 1\ldots \end{array}$$

We care about <u>scope</u>, not variable <u>name</u>!

- locals should be "local"
- choice of local names should not escape function, should only have local consequences

OK, maybe just wrong semantics... what about

$$\frac{H; e_1 \Downarrow \text{fun } x \to s \quad H; e_2 \Downarrow v \quad \text{fresh}(\text{"}y\text{"})}{H; e_1(e_2) \to H; y = x;\ x := v;\ s;\ x := y}$$

preserves x!

Sorta weird:

- "fresh" not very IMP-ish, but ok (malloc)
- far from actual implementation
- inconvenient for reasoning about something basic as calling...

TOTALLY BROKEN!

- what if "f" calls another func that modifies global "x"?

```
f := (fun x -> g:=(fun z -> ans := x + z));
f (2);
x := 3;
g (4)
```

WANT: ans = 6

- f(2) should make g a func which adds 2 to its arg and stores result in ans

REALITY: ans = 7

- f(2) sets g to a func which adds <u>current</u> value of x to its arg :(

PUNCH LINE

- can't properly model local scope w/ just a global heap of ints
  - funcs are not simply sugar for assigns to globals
- take a step back, figure out this core idea
  - add IMP features back later
- get rid of everything:
  - mutation, conditionals, loops (!), even integers (!!!)
- Someone thought of this long ago...

THE LAMBDA CALCULUS (the coolest)

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

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

"Whatever next 700 languages turn out to be, they will surely be variants of the lambda calculus." Landin '66 __apply__ a func by __substituting__ the argument for the __bound__ variable.

$$(\lambda \underbrace{x}_{\text{bound variable}} . \underbrace{e_1}_{\text{body}}) \underbrace{(e_2)}_{\text{argument}}$$

(we'll see other ways to think about this too...)

__Examples:__

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

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

$(\lambda x . x\ x)(\lambda x . x\ x) \to (\lambda x . x\ x)(\lambda x . x\ x)$

- subst was key idea we were missing!
- after subst bound var gone ...
    - so its name was irrelevant !!! (awesome)

Substitution (I know this notation is weird!) . . .

"$e_1[e_2/x]$" means "replace x with e_2 everywhere it occurs (free) in e_1"

(don't think sed/UNIX !!!)

Semantics

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

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

- small step, call by value (CBV), left-to-right
  - stops (terminates) when you get to some value (ie. $\lambda x.e$)

$$(\lambda a.\, \lambda b.\, a)(\lambda c.\, \lambda d.\, c)(\lambda e.\, \lambda f.\, f)$$

✓

×

$$(\lambda b.\, \lambda c.\, \lambda d.\, c)(\lambda e.\, \lambda f.\, f)$$

$$(\lambda a.\, \lambda b.\, a)(\lambda d.\, \lambda e.\, \lambda f.\, f)$$

$$\lambda c.\, \lambda d.\, c$$

$$\lambda b.\, \lambda d.\, \lambda e.\, \lambda f.\, f$$

ztatlock@nocatgw

2c.jpg.jpeg

Sun, 13 Oct 2013 13:29:04 -0700

ps541 / hp LaserJet 4200

But we can get stuck too... :(

- *free* var at "top level"
  - ↳ not bound under some λ

This is the heart, the asm, the core of languages like OCaml, Haskell, Agda, Coq, ...

- (though real implementations do something more efficient than substitution... but it's *equivalent* to subst [proofs!])

A Couple Notes on Concrete Syntax

Disambiguate:

▷ $\lambda x. e_1\ e_2$ is $(\lambda x. e_1\ e_2)$ not $(\lambda x. e_1)\ e_2$

▷ $e_1\ e_2\ e_3$ is $(e_1\ e_2)\ e_3$ not $e_1\ (e_2\ e_3)$

▷ Does it matter? YES! app not assoc!

In general:

- ▷ function bodies extend right until they hit a ")"
- ▷ application associates to the left

As in IMP, we assume AST under it all

- ▷ all non-leaves either λ or "app"
- ▷ weird syntax OK but it's stood test of time (70 years!)

* Don't even see the code, only see ...*

<u>GREAT</u>. So what have just done? (found problems impt functions)

- developed & formalized CBV λ-calc using substitution
- ... built something that can encode

ALL COMPUTATION !!! Wow!

... really? what about all those missing features...

OK, this is nuts. We left out numbers. We left out conditionals. We left out loops. How the heck is this thing Turing complete? How could I even convince you it is?

Lambda Encodings

We'll build up enough features to write programs somewhat intuitively. (Church-Turing Thesis does the rest...)

Could pull IMP trick, but this is cooler!

Why?

- Expand your mind. (Morpheus.) "Don't even see the code."
- Demonstrate model is realistic.
- We don't *really* need all those fancy modern features (yeah right)

Booleans

Any implementation of bool must provide 3 thgs

"true" $\equiv \lambda x. \lambda y. x$

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

conditional : take 3 args. if 1st arg true you get 2nd arg, otherwise 3rd

"if" b "then" t "else" f $= \lambda b. \lambda t. \lambda f.\ b\ t\ f$

just as good as any other impl!

Note: "if" "true" $v_1\ v_2 \xrightarrow{*} v_1$

Some boolean operators:

$$\text{not} = \lambda b.\ b\ \text{"false"}\ \text{"true"}$$

$$\text{and} = \lambda a.\ \lambda b.\ a\ b\ \text{"false"}$$

$$\text{or} = \lambda a.\ \lambda b.\ a\ \text{"true"}\ b$$

PAIRS (delicious)

Pairs require 3 things: constructor, fst, snd

$$\text{"mkpair"} = \lambda x.\ \lambda y.\ (\lambda z.\ z\ x\ y)$$

$$\text{"fst"} = \lambda p.\ p\ \underbrace{(\lambda x.\ \lambda y.\ x)}_{\text{"true"}}$$

$$\text{"snd"} = \lambda p.\ p\ \underbrace{(\lambda x.\ \lambda y.\ y)}_{\text{"false"}}$$

__Example:__ $\text{snd}\ (\text{fst}\ (\text{mkpair}\ v_1\ v_2)\ v_3) \xrightarrow{*} v_2$

Woah... fst = true and snd = false ?!

Uh, is that OK? Sure! We use the same bit pattern to mean all sorts of different things at the arch level: int, float, ptr...

Of course, just because we __can__ do something, doesn't mean we __should__. Beware the Turing tarpit!

LISTS

Turns out, we've alredy encoded enough to build lists at a higher level! Just use bools and pairs!

nil = mkpair false false

cons = $\lambda h.\ \lambda t.$ mkpair true (mkpair h t)

is-empty = fst

head = $\lambda \ell.$ fst (snd ℓ)

tail = $\lambda \ell.$ snd (snd ℓ)

Note: tail nil does weird stuff, but so ... does following null → next in C / Java etc...
(kinda how list work at asm lvl)

Doing Good!

bools, pairs, list :)

Numbers "Church Numerals"

$0 = \lambda s. \lambda z. z$

$1 = \lambda s. \lambda z. s\ z$

$2 = \lambda s. \lambda z. s\ (s\ z)$

$3 = \lambda s. \lambda z. s(s(s\ z))$

...