LC syntax Outline 11/2/16

CBV left-to-right

Q: how to right-to-left?

Q: write a program that "behaves" differently in LR vs RL CBV

CBN

Q: difference? A: sometimes more work, sometimes less (usually more)

Q: Write a program that computes a different value in CBN vs CBV?

A: modulo termination (hey, kind of a big deal), no!

Q: How to prove? A: Full reduction + confluence. (draw diamond)

With "impure" features, can distinguish many more eval strategies

Add "beep" to LC (see example)

Call by need / Laziness (...) Takeaway: different! Good pattern to know!

Substitution

Examples on side board (TDD haha)

Principle of renaming bound variables

Proposed defn — counterexample etc.

...

Implement in Coq

505 LC Eval order + Substitution 11/2/16

Recall (left-to-right) CBV semantics for LC $\quad e ::= x \mid \lambda x.e \mid e\,e$

$v ::= \lambda x.e$ (closed)

$\boxed{e \to_{cbv} e'}$

$$\frac{}{(\lambda x.e_1)\,v \to_{cbv} e_1[v/x]} \qquad \frac{e_1 \to_{cbv} e_1'}{e_1\,e_2 \to_{cbv} e_1'\,e_2} \qquad \frac{e_2 \to_{cbv} e_2'}{v\,e_2 \to_{cbv} v\,e_2'}$$

Aside: You will see 1000 different notations for substitution. Here are all the ones I could think of off the top of my head

say: "e_1 with e_2 for x"

$e_1[e_2/x] \qquad [e_2/x]e_1 \qquad e_1[x := e_2] \qquad e_1[x \mapsto e_2]$

$e_1[e_2]$ (if x is known from context)

(please, make up your own!)

mnemonic: the slash is falling over to crush the x, replacing it with e_2. (e_2 is left sitting on top after slash falls :-))

★ Question: Can you write a program the evals differently in right to left? ★ Give an example of such a prog. or prove impossible.

Also the call-by-name semantics

$\boxed{e \to_{cbn} e'}$

$$\frac{}{(\lambda x.e_1)\,e_2 \to_{cbn} e_1[e_2/x]} \qquad \frac{e_1 \to_{cbn} e_1'}{e_1\,e_2 \to_{cbn} e_1'\,e_2}$$

What's the difference between these semantics? Any progs eval differently? Yes!

~~(λf. λx. f x x)~~ $(\lambda f.\ \lambda x.\ f\,x\,x)$ "plus" <expensive 7>

~~λ(λx. (λy.y)~~

~~λx.~~

under cbv, compute <expensive 7> first, then plug in 7 for x.

under cbn, make two copies of <expensive 7> and evaluate them separately during "plus" : "twice as slow"

$(\lambda x.\lambda y.\ y)$ (<infinite loop>) $(\lambda z.z)$

under cbv, loops

under cbn, $\to^* \lambda z.z$: "infinitely faster"

In pure LC, termination (and performance) is the only difference among the semantics. You can never get a different answer just by computing things in a different order.

With "impure" features, you can tell the difference.

(This is actually a pretty good definition of purity, which can be tricky to define directly.)

Another way to say this is that purity is about values, while impurity lets a little bit of "how you got there" leak into the meaning of programs. Think: heap in IMP, I/O, etc.

If we added "beep" to LC, then (beep then becomes $\lambda x.x$)

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

beeps once in CBV but zero times in CBN, → thus it's

$(\lambda x.\lambda y.yxx)$ beep $(\lambda a.\lambda b.\lambda z.z)$

beeps once in CBV.

OKay, so CBV sometimes does more work because it evals arguments even if they're not used. But CBN sometimes does more work b/c it evals arguments each time they're used.

The systems person in me is like: cache that shit!

That's a thing, and it's called "call-by-need", or "lazy" evaluation.

Formal semantics for this is a bit tricky because we have to say when exactly we "need" something. But intuitively we go like CBN, except that when we substitute, we plug in a "pointer" and make it point to the unevaluated argument. When ever we need the value behind the pointer, we evaluate it and replace it w/ the value. Later re-evals will be cheap.

Need for LC is just ~~left-most~~ when you have a variable applied to arg (i.e. pointer to lazy val)

Seems good because "no wasted work."

One tricky thing is side effects, though. Now, whether your arg gets evaluated (and to what extent) is implementation defined.

Depends not only on callee, but on the enclosing context's "need"

Haskell famously uses call-by-need, and doesn't have "real" side-effects for this reason.

$(\lambda four.$ ~~[illegible]~~ plus four four$)$ (length $(0 :: 1 :: beep\ 2 :: 4+3 :: nil)$)

evaluates length of list once, when plus asks for it
none of the elements of the list are evaluated
because length doesn't need them

People always say "call-by-need" makes it harder to reason about side effects. There's some truth to that, but I think it's mostly just different. It can be hard to reason about efficiency (esp. space) because it's hard to tell when a computation will need something, so it tends to get kept around.

Substitution, formally

Crazy thing about LC: everything boils down to β reduction.

$$(\lambda x.\, e_1)\, e_2 \longmapsto e_1[e_2/x]$$

Crazy thing about β reduction: it all boils down to substitution.

Since LC Turing complete, in some sense, all computation boils down to substitution!

In other words, if you can implement substitution, you can implement an arbitrary PL.

Substitution seems simple + intuitive. But maybe it's needy seen power should scare us. In fact substitution's effects are intuitive but implementation / details extremely subtle.

This is ~~sort~~ one of the "things." in PL.

Okay, suppose we forget that it's supposed to be hard and just go for it.

Goal is to define a function

$$e_1[e_2/x]$$

it takes three args: two exprs and a var name and returns an expr

Start w/ some examples: (on side board)

$x[\lambda y.\, y/x] = \lambda y.\, y$

$x[e/x] = e$ for any e.

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

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

Go by recursion on e_1:

$$x[e_2/x] = e_2$$

$$y[e_2/x] = y \quad (y \neq x)$$

$$(\lambda y.\, e)[e_2/x] = \lambda y.\, e[e_2/x]$$

$$(e_3\ e_4)[e_2/x] = e_3[e_2/x]\ e_4[e_2/x]$$

Desired Facts about subst

- α
- comm
- preservation of free vars

Recall principle of ~~α-equivalence~~ renaming bound var (aka. bound var name irrelevance)
- We should not depend on actual names of vars.
- Just "place holders"

We'll want to make sure our definition of substitution plays nice w/ α equiv.

Okay back to proposed definition.
- works okay on all previous examples
- λ case is not correct b/c does not respect shadowing

$(\lambda x.\ \lambda x.\ x)\ 42$

w/ our proposed definition, $\longrightarrow \lambda x.\ 42$

but "should" be $\lambda x.\ x$

The outer x is shadowed by inner λ, so var x inside refers to inner binding. This is why it can be useful to think of vars as place holders or pointers to their binding site

Okay try again but fix λ case to handle shadowing.

if λ binds same var, just stop. any reference to that ~~string~~ name inside will refer to λ and not us.

$$x[e_2/x] = e_2$$
$$y[e_2/x] = y \quad (y \neq x)$$
$$(\lambda x.\ e_1)[e_2/x] = \lambda x.\ e_1$$
$$(\lambda y.\ e_1)[e_2/x] = \lambda y.\ e_1[e_2/x] \quad (y \neq x)$$
$$(e_1\ e_3)[e_2/x] = e_1[e_2/x]\ e_3[e_2/x]$$

Works by checking if λ shadows our var.
- fixes previous bug and still works on all previous examples

Now we have a new problem

if e_2 uses a name that is bound by λ in e_1, then it is "captured".

This is the actual hard part of subst. (Not the previous problem.)

Example: $(\lambda x.\, \lambda y.\, x)\ y \longrightarrow \lambda y.\, y$ Changed "where y points" from "no where/outside" to that λ!

feels "wrong"

more formally, violates principle of renaming since changing ~~of~~ bound y to a gives

$(\lambda x.\, \lambda a.\, x)\ y \longrightarrow \lambda a.\, y$ Note y still free.

Desired properties of substitution:

- Preserves α-equivalence: $e_1 \sim_\alpha e_2 \Rightarrow e_1[e/x] \sim_\alpha e_2[e/x]$

Side note: this definition works fine if we assume e is closed (no free vars) For example both CBV and CBN never perform a substitution where e_2 has FV, so we're fine. Can get pretty far like this, but not "real" substitution.

One Fix: check for this case too. λ case becomes

$$(\lambda y.\, e_1)[e_2/x] = \lambda y.\, e_1[e_2/x] \quad (y \neq x \text{ and } y \notin FV(e_2))$$

Downside: subst is partial. Gets stuck if capture would occur.

since whether capture occurs or not depends on bound var names this is bad.

"Paper PL fix": implicitly assume that y has been renamed to avoid capture then ignore this problem for ever.

So in our example

~~$(\lambda x.\, (\lambda y.\, x))$~~

$(\lambda x.\, \lambda y.\, x)\ (\lambda z.\, y)$ ← implicitly rename → $(\lambda x.\, \lambda w.\, x)\ (\lambda z.\, y)$ ← not renamed!

Then substitution will work.

If we want to actually implement this stuff, implicit "conventions" aren't going to cut it.

Simplest approach is to make the implicit explicit by doing renaming.

$$
\begin{aligned}
x\,[e_2/x] &= e_2 \\
y\,[e_2/x] &= y \quad (y \neq x) \\
(\lambda y.\, e_1)\,[e_2/x] &= \text{let } z = \text{fresh}(e_1, e_2) \text{ in} \\
&\quad \text{let } e_1' = \text{rename}(y \mapsto z, e_1) \text{ in} \\
&\quad \lambda z.\, e_1'\,[e_2/x] \\
(e_1\ e_3)\,[e_2/x] &= (e_1\,[e_2/x]\ \ e_3\,[e_2/x])
\end{aligned}
$$

We can even reuse $e_1[z/y]$ as our renaming function.

This works great in OCaml, eg.

Unfortunately, not structurally recursive since we first rename and then recurse.

Excellent excercise (hard/interesting!!!) to think about how to fix this

(Only ways I know of are De Bruijn and multisubst)

Getting this right in practice w/ explicit names is horrible