\[ (\forall r \cdot \text{not} (\text{snd } r)) \Rightarrow \vDash \text{not} (\text{fst } r) \]

\[ \text{head } = \vDash \text{if } \text{is-empty } \text{then } \text{fst } \text{else } \text{nil} \]

\[ \text{cons } = \vDash \text{if } \text{is-empty } \text{then } \text{false } \text{else } \text{match } x \cdot \text{true} (\text{match } y \cdot \text{false}) \]

Books and Pairs!

Turns out, we've already encoded enough to build lists at a higher level! Just use

\[ \text{null} \]

\[ \text{fst} \]

\[ \text{snd} \]
$3 = \lambda z. \lambda z. s (s (s z))$

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

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

$0 = \lambda z. z$

Church Numerals

Ding (good)

Note: tail will do weird stuff, but so
applicable at first and one more time

\[ \text{succ} = \lambda n. n \cdot \lambda z. \text{sz} (n \cdot \text{sz} z) \]

Right bending for \( s \) and \( z \)

A implementation arithmetic by clearly passing in

A "\( s \)" stands for successor

A "\( z \)" stands for zero

Starting with its second and

\( \alpha \cdot \text{compose} \) its first and \( \alpha \) itself \( \alpha \) times,

\( \alpha \) numbers encoded with 2-ary times
\[ \text{A (Add n) n times to zero.} \]

"success" in the first.

\[ A \text{ times } \frac{A}{n} \text{ m, n, m (plus m) zero.} \]

\[ \text{N+m times to its 2nd ary} \]

A result = a function which applies its 1st ary "zero" for the first.

\[ \text{N times to the first ary} \]

\[ \text{A times to the two numbers and use me one as the} \]

\[ \text{plus = A \times A \times A \times A \times A} \]
is_zero = \lambda n. n (\lambda x. \text{false}) \text{ true}

▷ think about it

- We can also do pred, minus, div, is_equal, etc.
- Notice we've done everything without loops!

**Loops** (well... recursion)

OK, almost there, but how do we repeat actions? Can we write divergent progs? Yes!
How about useful ones? ... let's see...
(turns out doing this clearly in full detail takes time... sketch only here)
A Fix is a function itself

\textit{Fix (a \rightarrow A \times \text{if} (x = 0) \text{then } \text{else } (x \times f (x-1)))}

A next apply a Fix to get a recursive thing.

\textit{Next we could just pass in the function itself, which be good...}

\textit{A if \ X \ \text{if} (x = 0) \text{then } \text{else } (x \times f (x-1))}

-e.g. if we had a few more features:

in place of recursion:

\textbf{Write func that \textbf{fixes} \ f and calls that}
If true \((\forall x \cdot x^2)\) \(\forall z \cdot (\forall x \cdot x^2)(\forall x \cdot x^2)\) if true \((\forall x \cdot x^2)\) if true \((\forall x \cdot x^2)(\forall x \cdot x^2)\)

may need "funct"

what about:

Any problems wi CBV?

--- everything

implement numbers as lists of books (o nil)

with fix we can do all sorts of stuff easier:
Next: more reduction strategies, hypotheses

Turing Complete? The hot topic!

Let $x = e_1 \in e_2^2$ (A X. C2) &

We can even do sets!
Reduction Strategies, Substitution

\[ \lambda \text{-calculus - syntax:} \]

\[ e ::= \lambda x.e \mid e \mid e \]

\[ (\lambda x.e) y \rightarrow e[y/x] \]

\[ (\lambda x.e) y \rightarrow e[y/x] \]

\[ \frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'} \]

\[ \frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'} \]

\[ \frac{y \rightarrow y'}{\lambda x.e \rightarrow \lambda x.e'} \]

\[ \frac{y \rightarrow y'}{\lambda x.e \rightarrow \lambda x.e'} \]

\[ \text{CBV, } \lambda \rightarrow \text{-X semantics:} \]

\[ \frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'} \]

\[ \frac{y \rightarrow y'}{\lambda x.e \rightarrow \lambda x.e'} \]

\[ \text{We've done a lot!} \]
What if we tweak semantics to throw out order?

Other Reduction Strategies

A the verbase version will be handy later

A more common, but...

\[ (\lambda x.e) \gamma \vdash e, \varepsilon \]

Last time we saw the first axiom as:

\[ (\lambda x.e) \varepsilon \vdash e, \varepsilon \]
What does this mean?

then \( E \) is \( e, e', e'' \), and \( e' \rightarrow e_2 \).

If \( e < e' \) and \( e_2 \rightarrow e' \).

In our order-less semantics:

Amazing Fact (Church-Rosser Theorem)

expressions is called a "reduction strategy".

The order in which you evaluate

- Freeness in equivalence proofs
- Optimizations / partial evaluation
- but provides some advantages:

1. NIs is sort of weird for a PL to do!
Convenience rules. They're often super.
Let's add a couple more.

Equivalence via Rewriting

How would you prove this?

Property:

In general, any rewriting system with this
property is said to have the "Church-Rosser"
property. It cannot pick a "wrong way" to go about eval.
\[
z \text{ is a literalmap (\text{true} \rightarrow \text{false}) \ if \ } z \text{ is a literalmap}\]

\[
\text{if } e \text{ is true then clause false}\ 
\]

\[
\text{if } x \not\in e \text{, then } e x \rightarrow e
\]

\[
\text{if } x \not\in e \text{, then } e x \rightarrow e
\]

\[
\text{occur "free" in } e\]

\[
\text{2. Replace } x \rightarrow e \rightarrow e \text{ with } e \text{ if } x \text{ does not occur "free" in } e
\]

\[
\text{1. Replace } x \rightarrow e \rightarrow e \text{ with } e \text{ if } x \text{ does not occur "free" in } e
\]

\[
Y. \text{ (assumed } Y \text{ not already used in } e)\]

\[
E. \text{ as } z \text{ with all } \text{ "free" } x \text{ replaced by } x. \text{ Where, when } x \in e, \text{ replace } x \rightarrow e \rightarrow e \text{ with } e.\]
If this rewriting can show e and e denote the same thing.

(Basically treat lambda as functions.)

Under natural denotational semantics, distinguishing be shown, we can show anything that can run them "backward" (rewrite right side with all these rules), plus ability to

That's all folks!
Just one more for Proof Equivalence? No!

So: the decision of expressions equality, just

(need set D, something to D → D)

(General) be natural computational semantics for H-CAE

A true equivalence.

We never have to add more rules to show

Our rules are complete, meaning

the respect the semantics

So our rules are sound, meaning
\( \text{\( e \in e_2 \rightarrow e \in e_2 \)} \)

\( e \in e_1 \)  

\( \exists x \in e \rightarrow e \in [e/x] \)

\( e \rightarrow e_1 \)

---

* even smaller than CBV!

"call by name" (CBN)

Semantics vs. (couple lectures back)

Remember, we did every proof for

How would you prove?

L-to-R CBV from \( \neg \text{to-} \neg \text{ CBV}\)

... (in short: effects)

Claim: without assignment (mutation), I/O, exceptions

Seen: "full reduction", L-to-R CBV

Other popular semantics
n * f (n-1)

else

if n < 1 then

let rec f n =

Imagine if OCaml "let" was CBV!

and termination.

Only matters for "performance" (# of steps)

If our case was an effect, then order

why re-evaluates args?

But may take more steps.

why? Only evals on demand.

Note: we actually barely less often than CBV
- Side effects get tricky
- Slower than C or C++
- For pure code, asymptotically no
  Best of Both Worlds?

Anyways, like C3N, but doesn't re-entrant!
It's used, then remember & reuse result
- Only need on one first time
  - also "lazy evaluation"
  - "call by need"

Another strategy
Also, roll your own control flow

foo = take 10 ones
ones = 1 : ones

Example:

\[ x + x = y \]
\[ x = \text{factorial 2} \]

Example:

Haskell programmers are lazy

Haskell uses call-by-value
\[ (x \cdot x) = (x \cdot x \cdot x) / x = x^2 \cdot x \]

Examples:

\[ x \cdot x \cdot y = x \cdot y \cdot x \]

Informally: e [e/x] "replaces each x in e with e"...

Sort of where we hid all the complexity...

Surprisingly, quite subtle - how hard can it be?

- used in the rule for Call

- still haven't nailed down substitution

One little detail!
Recursively replace every $x$ that is not $\in \mathcal{E}$ with $\epsilon$.

\[
\text{if } x = e \epsilon \text{ or } e \epsilon x \epsilon \text{ then } y = \epsilon \epsilon \epsilon \epsilon \epsilon \epsilon 
\]

\[
\forall E \epsilon x \epsilon \in \mathcal{E} \quad x \epsilon E \epsilon x \epsilon = e 
\]

\[
\forall E \epsilon x \epsilon \in \mathcal{E} \quad e \epsilon x \epsilon = e 
\]

\[
\forall E \epsilon x \epsilon \in \mathcal{E} \quad x \epsilon E \epsilon x \epsilon = e 
\]

\[
\forall E \epsilon x \epsilon \in \mathcal{E} \quad e \epsilon x \epsilon = e 
\]

\[
\forall E \epsilon x \epsilon \in \mathcal{E} \quad x \epsilon E \epsilon x \epsilon = e 
\]

\[
\forall E \epsilon x \epsilon \in \mathcal{E} \quad e \epsilon x \epsilon = e 
\]

\[
\forall E \epsilon x \epsilon \in \mathcal{E} \quad x \epsilon E \epsilon x \epsilon = e 
\]
Okay, let's try again...

Yes!

Consider: \( y \cdot x \cdot x \cdot x \cdot y \) 42

Outer func (shadows).

Inner function binds same var as...

Wrong for nested functions, if the...
If a function body uses an "outer" \( y \) these rules will capture it.

-Still wrong! It's a function body.

-For your function: stop when you hit blanks.

Respect shadowing: stop when you hit blanks.

\[
\begin{align*}
&\forall x \in \mathbb{R}, \quad e^x = e^x \\
&\exists x \in \mathbb{R}, \quad \exists y \in \mathbb{R}, \quad x \neq y \\
&(x, e^x) \in \mathbb{R} \\
&x \neq y \\
&y \in \mathbb{R} \\
&x \in \mathbb{R}
\end{align*}
\]
FV (x \cdot e) = FV (e) \cup \{x\}

FV (e \cdot c \circ) = FV (e) \cup FV (c) \cup FV (e \circ)

FV (x) = 3 \times 3

Free Variables need to know what's bounded

Can arise under full reduction

(\lambda \cdot \text{a} \cdot \text{b} \cdot \text{c}) \cdot (\text{a} \cdot \text{b} \cdot \text{c} \cdot \text{d})

\not\Rightarrow (\lambda \cdot \text{a} \cdot \text{b} \cdot (\lambda \cdot \text{c} \cdot \text{d}) \cdot \text{e})

(\lambda \cdot \text{x} \cdot \text{y} \cdot \text{x} \cdot \text{y}) \not\Rightarrow (\lambda \cdot \text{z} \cdot \text{y})
\[
x \Gamma \vdash e \vDash e \quad \frac{\gamma \neq x}{\gamma[e/x] = \gamma}
\]

\[
e_1[e/x] = e' \quad \gamma \neq x \quad \gamma \notin \text{fv}(e)
\]

\[
(\lambda y. e_1)[e/x] = \lambda y. e_1'
\]

\[
e_1[e/x] = e' \quad e_2[e/x] = e_2'
\]

\[
(e, e_2)[e/x] = e_1 e_2'
\]

OK... but could get stuck (no rule applies)

Implicit Renaming

- our defn only partial if y "accidentally" used as a binder
- whole point was we don't care how local vars are named!
- Different ASTs considered the same

- Key design principle based on how their vars are named. Generally, never distinguish between terms

- We must lose x shadowing rule

- By renaming, the x rule can always apply

- To handle this, we allow implicit renaming of a binding & all its occurrences.
\[ (x \cdot e) \cdot [e/x] = x \cdot e \]

\[ e \cdot [e/x] = e \quad y \neq x \quad y \in \text{Face} \]

\[ \begin{array}{c}
\frac{(c \cdot e) \cdot (c \cdot x)}{e \cdot \text{Face} = e} = e \cdot e \cdot e = e \\
\frac{y \cdot [e/x]}{e \cdot [e/x] = e} = e \\
\frac{x \cdot [e/x]}{e \cdot [e/x] = e} = e \\
\frac{e \cdot [e/x]}{e \cdot [e/x] = e} = e \\
\end{array} \]

Assume implicit systematic renaming, then...
- always find some 2
- global count in compiler

\[(\forall y \in e, e[y/x] = x) \subseteq \forall e, e[y/x] = e\]

\[\forall x \in e, e[x/y] = e, e[x/y] = e, e[x/y] = e\]

- Can implement w/ this verbose rule too:

- Google "capture avoiding substitution"

- Notoriously annoying problem in PL
- \( \lambda x.x \rightarrow e \) called "\( \beta \)-reduction",
- \( \lambda x.e \rightarrow e[L_x/e] \) called "\( \gamma \)-reduction",
- \( \gamma \) - reduces to \( \gamma \) - conversion,
- \( \gamma \) - reduces to \( \gamma \) - conversion,