# CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2009

Lecture 7— Reduction Strategies; Substitution;
Simply Typed Lambda Calculus

# Review

$\lambda$-calculus syntax:

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

Call-By-Value Left-Right Small-Step Operational Semantics:

$\boxed{e \longrightarrow e'}$

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

# 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:

$\boxed{e \longrightarrow e'}$

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

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

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

- 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 $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$, then there exists an $e_3$ such that $e_1 \longrightarrow^* e_3$ and $e_2 \longrightarrow^* e_3$.

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

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):

$\boxed{e \longrightarrow e'}$

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

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.

# More on Call-By-Need

This course will mostly assume Call-By-Value.

Haskell uses Call-By-Need.

Example:

```
four = length (9:(8+5):17:42:[])
eight = four + four
main = do { putStrLn (show eight) }
```

Example:

```
ones = 1 : ones
nats_from x = x : (nats_from (x + 1))
```

# Formalism not done yet

Need to define substitution—shockingly subtle

Informally: $e[e'/x]$ " replaces occurrences of $x$ in $e$ with $e'$ "

$$\boxed{e_1[e_2/x] = e_3}$$

Attempt 1:

$$\frac{}{x[e/x] = e} \qquad \frac{y \neq x}{y[e/x] = y} \qquad \frac{e_1[e/x] = e_1'}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'}$$

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

# Getting substitution right

Attempt 2:

$$\frac{e_1[e/x] = e_1' \qquad y \neq x}{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'} \qquad \frac{}{(\lambda x.\ e_1)[e/x] = \lambda x.\ e_1}$$

What if $e$ is $y$ or $\lambda z.\ y$ or, in general $y$ is *free* in $e$? This *mistake* is called *capture*.

It doesn't happen under CBV/CBN *if* our source program has *no free variables*.

Can happen under full reduction.

# Another Try

Attempt 3:

First define the "free variables of an expression" $\boldsymbol{FV(e)}$:

$$\begin{aligned} \boldsymbol{FV(x)} &= \boldsymbol{\{x\}} \\ \boldsymbol{FV(e_1\ e_2)} &= \boldsymbol{FV(e_1) \cup FV(e_2)} \\ \boldsymbol{FV(\lambda x.\ e)} &= \boldsymbol{FV(e) - \{x\}} \end{aligned}$$

Now define substitution with these rules for functions:

$$\frac{\boldsymbol{e_1[e/x] = e_1' \qquad y \neq x \qquad y \notin FV(e)}}{\boldsymbol{(\lambda y.\ e_1)[e/x] = \lambda y.\ e_1'}} \qquad \frac{}{\boldsymbol{(\lambda x.\ e_1)[e/x] = \lambda x.\ e_1}}$$

But a *partial* definition (as stands, could get stuck because there is no substitution).

# Implicit Renaming

- A *partial* definition because of the *syntactic accident* that $y$ was used as a binder (should not be visible – local names shouldn't matter).
- So we allow *implicit systematic renaming* (of a binding and all its bound occurrences).
- So the left rule can always apply (can drop the right rule).
- In general, we *never* distinguish terms that differ only in the names of variables. (A key language-design principle!)
- So now even "different syntax trees" can be the "same term".

# Summary and some jargon

- If everything is a function, every step involves an application: $(\boldsymbol{\lambda x.\ e})e' \longrightarrow e[e'/x]$ (called $\boldsymbol{\beta}$-reduction)
- Substitution avoids capture via implicit renaming (called $\boldsymbol{\alpha}$-conversion)
- With full reduction, $(\boldsymbol{\lambda x.\ e\ x}) \longrightarrow \boldsymbol{e}$ makes sense if $\boldsymbol{x \notin FV(e)}$ (called $\boldsymbol{\eta}$-reduction), for CBV it can change termination behavior
  - But advanced Camlers scoff at `fun x -> f x`, since that's equivalent to `f`.

Most languages use CBV application, some use call-by-need.

Our Turing-complete language models functions and encodes everything else.

# Why types?

Our *untyped* $\boldsymbol{\lambda}$*-calculus* is universal, like assembly language. But we might want to allow *fewer programs*

1. Catch "simple" mistakes (e.g., "if" applied to "mkpair") early (but a decidable type system must be conservative)
2. (Safety) Prevent getting stuck (e.g., $\boldsymbol{x}\ \boldsymbol{e}$) (but for pure $\boldsymbol{\lambda}$-calculus, just need to prevent free variables)
3. Enforce encapsulation (an *abstract type*)
   - clients can't break invariants
   - clients can't assume an implementation
   - requires safety

Continued...

# Why types? continued

4. Assuming well-typedness allows faster implementations
   - smaller interfaces enable optimizations
   - don't have to check for being stuck
   - orthogonal to safety (e.g., C)
5. Syntactic overloading (not too interesting)
   - "late binding" (via run-time types) very interesting
6. Detect other errors via extensions (often "effect systems")
   - dangling pointers, data races, uncaught exceptions, tainted data, ... analysis, ...

   (Deep similarities in analyses suggest type systems a, "good way to think-about/define/prove what you're checking")

We'll really focus on (1), (2), and (3) though (plus (6) if have time)

# What is a type system?

Er, uh, you know it when you see it. Some clues:

- A decidable (?) judgment for classifying programs (e.g., $e_1 + e_2$ has type int if $e_1$ and $e_2$ have type int else it *has no type*)
- Fairly syntax directed (non-example??: $e$ terminates within 100 steps)
- A sound (?) abstraction of computation (e.g., if $e_1 + e_2$ has type int, then evaluation produces an int (with caveats!))

This is a CS-centric, PL-centric view. Foundational type theory has more rigorous answers (type systems are proof systems for logics)

# Plan for a couple weeks

- Simply typed $\lambda$ calculus (ST$\lambda$C)
- (Syntactic) Type Soundness (i.e., safety)
- Extensions (pairs, sums, lists, recursion)

*Then break for the Curry-Howard isomorphism; continuations; midterm*

- Subtyping
- Polymorphic types (generics)
- Effect systems (?)
- Recursive types
- Abstract types

Homework: Adding back mutation

Omitted: Type inference

# Adding constants

Let's add integers to our CBV small-step $\lambda$-calculus:

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

We could add + and other *primitives* or just parameterize "programs" by them: $\lambda plus.\ e$. (Like Pervasives in Caml.)

$\boxed{e \longrightarrow e'}$

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

What are the *stuck* states? Why don't we want them?

# Wrong Attempt

$$\tau \quad ::= \quad \textbf{int} \mid \textbf{fn}$$

$\boxed{\vdash e : \tau}$

$$\frac{}{\vdash \lambda x.\ e : \textbf{fn}} \qquad \frac{}{\vdash c : \textbf{int}} \qquad \frac{\vdash e_1 : \textbf{fn} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1\ e_2 : \textbf{int}}$$

1. NO: can get stuck, $(\lambda x.\ y)\ 3$
2. NO: too restrictive, $(\lambda x.\ x\ 3)\ (\lambda y.\ y)$
3. NO: types not preserved, $(\lambda x.\ \lambda y.\ y)\ 3$

# Getting it right

1. Need to type-check function bodies, which have free variables
2. Need to distinguish functions according to argument and result types

For (1): $\Gamma ::= \cdot \mid \Gamma, x : \tau$ and $\Gamma \vdash e : \tau$.

For (2): $\tau ::= \textbf{int} \mid \tau \rightarrow \tau$ (an infinite number of types)

E.g.s: $\textbf{int} \rightarrow \textbf{int}$, $(\textbf{int} \rightarrow \textbf{int}) \rightarrow \textbf{int}$, $\textbf{int} \rightarrow (\textbf{int} \rightarrow \textbf{int})$.

Concrete syntax note: $\rightarrow$ is right-associative, so $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ is $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

# STλC Type System

$$\begin{array}{rcl} \tau & ::= & \textsf{int} \mid \tau \rightarrow \tau \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash c : \textsf{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

The *function-introduction* rule is the interesting one...

# A closer look

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

1. Where did $\tau_1$ come from?
   - Our rule "inferred" or "guessed" it.
   - To be syntax directed, change $\lambda x.\ e$ to $\lambda x : \tau.\ e$ and use that $\tau$.
2. Can think of "adding $x$" as shadowing or requiring $x \notin \mathbf{Dom}(\Gamma)$. Systematic renaming ($\alpha$-conversion) ensures $x \notin \mathbf{Dom}(\Gamma)$ is not a problem.
3. Still "too restrictive". E.g.: $(\lambda x.\ (x\ (\lambda y.\ y))\ (x\ 3))\ \lambda z.\ z$ does not get stuck, but doesn't type-check
   - $((\lambda z.\ z)(\lambda y.\ y))((\lambda z.\ z)\ 3)$ type-checks though)

# Always restrictive

"gets stuck" undecidable: If $e$ has no constants or free variables, then $e\ (\mathbf{3}\ \mathbf{4})$ (or $e\ x$) gets stuck iff $e$ terminates.

Old conclusion: "Strong types for weak minds" – need back door (unchecked cast)

Modern conclusion: Make "false positives" (reject safe program) rare and "false negatives" (allow unsafe program) impossible. Be Turing-complete and convenient even when having to "work around" a false positive.

Justification: false negatives too expensive, have resources to use fancy type systems to make "rare" a reality.

# Evaluating ST$\lambda$C

1. Does ST$\lambda$C prevent false negatives? Yes.
2. Does ST$\lambda$C make false positives rare? No. (A starting point)

Big note: "Getting stuck" depends on the semantics. If we add $c\ v \longrightarrow 0$ and $x\ v \longrightarrow 42$ we "don't need" a type system. Or we could say $c\ v$ and $x\ v$ "are values".

That is, the language dictator deemed $c\ e$ and free variables bad (not "answers" and not "reducible"). Our type system is a conservative checker that they won't occur.

# Type Soundness

We will take a *syntactic* (operational) approach to soundness/safety (the popular way since the early 90s)...

Thm (Type Safety): If $\cdot \vdash e : \tau$ then $e$ diverges or $e \longrightarrow^n v$ for an $n$ and $v$ such that $\cdot \vdash v : \tau$.

- That is, if $\cdot \vdash e : \tau$ and $e \longrightarrow^n e'$, then $e'$ is not stuck (it might be a value).

Proof: By induction on $n$ using the next two lemmas.

Lemma (Preservation): If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$, then $\cdot \vdash e' : \tau$.

Lemma (Progress): If $\cdot \vdash e : \tau$, then $e$ is a value or there exists an $e'$ such that $e \longrightarrow e'$.