CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2007

Lecture 7— Substitution; Simply Typed Lambda Calculus

Where we are

- Introduced $\boldsymbol{\lambda}$-calculus to model scope and functions.
- CBV $\boldsymbol{\lambda}$-calculus models higher-order functions in languages like ML and Scheme very well (and functions/function-pointers in C).
  - Call-by-need deserves more airtime than I am giving it.
- Still need to define substitution.
- Then 2–3 weeks on type systems.
- Plus a digression about stack-machines and *continuations*
- Then concurrency.
- Then objects.

Review

λ-calculus syntax:

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

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

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

Call-By-Name Small-Step Operational Semantics:

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

Call-By-Need in theory "optimizes" Call-By-Name.

For most of course, assume CBV Left-Right.

Formalism not done yet

Need to define substitution—shockingly subtle.

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

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" $FV(e)$:

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

Now define substitution with these rules for functions:

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

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: $(\lambda x.\ e)e' \rightarrow e[e'/x]$ (called β-reduction)

- Substitution avoids capture via implicit renaming (called α-conversion)

- With full reduction, $(\lambda x.\ e\ x) \rightarrow e$ makes sense if $x \notin FV(e)$ (called η-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* (whether or not we remain Turing complete):

1. Catch "simple" mistakes (e.g., "if" applied to "mkpair") early (too early? not usually)
2. (Safety) Prevent getting stuck (e.g., $x\ e$) (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
4. Assuming well-typedness allows faster implementations
   - E.g., don't have to encode constants and plus as functions

- 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. Novel uses in vogue (e.g., prevent data races)

We'll mostly focus on (2) with informal investigation of (3)

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.

Plan for a couple weeks

- Simply typed $\boldsymbol{\lambda}$ calculus (ST$\boldsymbol{\lambda}$C)
- (Syntactic) Type Soundness (i.e., safety)
- Extensions (pairs, sums, lists, recursion)
- Type variables ($\forall$, $\exists$, $\boldsymbol{\mu}$)
- Inference (not needing to write types)
- Later: References and exceptions (interesting even w/o types)
- Relation to ML (throughout)

And some other cool stuff as time permits...

Adding constants

Let's add integers to our CBV small-step λ-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 paramterize "programs" by them: $\lambda plus.\ e$. (Like Pervasives in Caml.)

(Could do the same with constants, but there are lots of them)

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

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

Wrong Attempt

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

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

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

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): $\Gamma ::= \cdot \mid \Gamma, x : \tau$ (a "compile-time heap"??) and $\Gamma \vdash e : \tau$.

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

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

Concretely, $\rightarrow$ is right-associative $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ is $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$.

STλC Type System

$$\Gamma \vdash e : \tau \qquad\qquad \begin{array}{rcl} \tau & ::= & \mathsf{int} \mid \tau \rightarrow \tau \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

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

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

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

A closer look

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

1. Where did τ_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 τ.
2. Can make Γ an abstract *partial function* if $x \notin \mathbf{Dom}(\Gamma)$. Systematic renaming ($\alpha$-conversion) allows it.
3. Still "too restrictive". E.g.: $\lambda x.\ (x\ (\lambda y.\ y))\ (x\ 3)$ applied to $\lambda z.\ z$ does not get stuck.

Always restrictive

"gets stuck" undecidable: If e has no constants or free variables, then $e\ (3\ 4)$ (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.

Also: let compilers *assume* well-typedness (enable transformations)

Evaluating ST$\boldsymbol{\lambda}$C

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

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

That is, the language dictator deemed $\boldsymbol{c}\ \boldsymbol{e}$ 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 $e \rightarrow^n v$ for an n and v such that $\cdot \vdash v : \tau$.

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

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

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

Prove Progress today; Preservation next time...

Progress

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

Proof: We first prove this lemma:

Lemma (Canonical Forms): If $\cdot \vdash v : \tau$, then:

- if τ is **int**, then v is some c
- if τ has the form $\tau_1 \rightarrow \tau_2$ then v has the form $\lambda x.\ e$.

Proof: By inspection of the form of values and typing rules.

We now prove Progress by structural induction (syntax height) on e...

Progress continued

The structure of e has one of these forms:

- x — impossible because $\cdot \vdash e : \tau$.
- c — then e is a value
- $\lambda x.\ e'$ — then e is a value
- $e_1\ e_2$ — By induction either e_1 is some v_1 or can become some e_1'. If it becomes e_1', then $e_1\ e_2 \rightarrow e_1'\ e_2$. Else by induction either e_2 is some v_2 or can become some e_2'. If to becomes e_2', then $v_1\ e_2 \rightarrow v_1\ e_2'$. Else e is $v_1\ v_2$. *Inverting the assumed typing derivation* ensures $\cdot \vdash v_1 : \tau' \rightarrow \tau$ for some τ'. So Canonical Forms ensures v_1 has the form $\lambda x.\ e'$. So $v_1\ v_2 \rightarrow e'[v_2/x]$.

Note: If we add $+$, we need the other part of Canonical Forms.