CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2003

Lecture 8— Typed Lambda Calculus, “Simple” Extensions

(Covering an Amazing Breadth of Concepts)

Outline

- Finish safety proof
- Discuss the proof
  - Chart of lemma dependencies
  - Actually inverting derivations
- Extend ST$\boldsymbol{\lambda}$C
  (pairs, records, sums, recursion, . . .)
  - For each, sketch proof additions
  - At the end, discuss the general approach
- Not today: References, exceptions, polymorphism, lists, . . .

Lemma dependencies

- Safety (evaluation never gets stuck unless a value)
  - Preservation (to stay well-typed)
    - Substitution (so β-reduction stays well-typed)
      - Weakening (so substituting under nested $\boldsymbol{\lambda}$s well-typed)
      - Exchange (technical point)
  - Progress (so well-typed not stuck)
    - Canonical Forms (so primitive reductions apply)

Comments:

- Substitution strengthened to open terms for the proof
- When we add heaps, Preservation will use Weakening directly

Induction on derivations – Another Look

The app cases are really elegant and worth mastering:

$e = e_1\ e_2$. For Preservation, lemma assumes $\cdot \vdash e_1\ e_2 : \tau$.

Inverting the typing derivation ensures it has the form:

$$\frac{\dfrac{\mathcal{D}_1}{\Gamma \vdash e_1 : \tau' \to \tau} \qquad \dfrac{\mathcal{D}_2}{\Gamma \vdash e_2 : \tau'}}{\Gamma \vdash e_1\ e_2 : \tau}$$

1 Preservation subcase: If $e_1\ e_2 \to e_1'\ e_2$, inverting that derivation means:

$$\frac{\dfrac{\mathcal{D}}{e_1 \to e_1'}}{e_1\ e_2 \to e_1'\ e_2}$$

continued...

The inductive hypothesis means there a derivation of this form:

$$\frac{\mathcal{D}_3}{\Gamma \vdash e_1' : \tau' \rightarrow \tau}$$

So a derivation of this form exists:

$$\frac{\dfrac{\mathcal{D}_3}{\Gamma \vdash e_1' : \tau' \rightarrow \tau} \qquad \dfrac{\mathcal{D}_2}{\Gamma \vdash e_2 : \tau'}}{\Gamma \vdash e_1'\ e_2 : \tau}$$

Write out the app case of the Substitution Lemma this way (invoke induction twice at once to get the new derivation)

Adding Stuff

- Extend the syntax
- Extend the operational semantics
  - Derived forms (syntactic sugar) (with/without types)
  - Direct semantics
- Extend the type system
- Consider soundness (stuck states, proof changes)
  - That's what makes these extensions simple

Let bindings (CBV)

$$e ::= \ldots \mid \textsf{let } x = e_1 \textsf{ in } e_2$$

$$\frac{e_1 \rightarrow e_1'}{\textsf{let } x = e_1 \textsf{ in } e_2 \rightarrow \textsf{let } x = e_1' \textsf{ in } e_2}$$

$$\frac{}{\textsf{let } x = v \textsf{ in } e_2 \rightarrow e_2[v/x]}$$

$$\frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \textsf{let } x = e_1 \textsf{ in } e_2 : \tau}$$

Progress: If e is a let, 1 of the 2 rules apply (using induction).

Preservation: Uses Substitution Lemma

Substitution Lemma: Uses Weakening and Exchange

Derived forms

let seems just like $\boldsymbol{\lambda}$, so can make it a derived form:

let $x = e_1$ **in** e_2 a "macro" (derived form) $(\boldsymbol{\lambda x.}\ e_2)\ e_1$.

(Harder (?) if $\boldsymbol{\lambda}$ needs explicit type.)

Or just define the semantics to replace let with $\boldsymbol{\lambda}$:

$$\frac{}{\textbf{let } x = e_1 \textbf{ in } e_2 \rightarrow (\boldsymbol{\lambda x.}\ e_2)\ e_1}$$

These 3 semantics are *different* in the state-sequence sense ($e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n$).

But (totally) *equivalent* and you could prove it (not hard).

Note: ML type-checks let and $\boldsymbol{\lambda}$ differently. (Later.)

Note: Don't desugar early if it hurts error messages!

Booleans and Conditionals

$$e \;::=\; \ldots \mid \textbf{true} \mid \textbf{false} \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$$

$$\tau \;::=\; \ldots \mid \textbf{bool} \qquad v \;::=\; \ldots \mid \textbf{true} \mid \textbf{false}$$

$$\frac{e_1 \to e_1'}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \to \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}$$

$$\frac{}{\textbf{if true then } e_2 \textbf{ else } e_3 \to e_2}$$

$$\frac{}{\textbf{if false then } e_2 \textbf{ else } e_3 \to e_3}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

$$\frac{}{\Gamma \vdash \textbf{true} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \textbf{false} : \textbf{bool}}$$

Notes: CBN, new Canonical Forms case, all lemma cases easy

Pairs (CBV, left-right)

$$
\begin{array}{rcl}
e & ::= & \ldots \mid (e, e) \mid e.1 \mid e.2 \\
\tau & ::= & \ldots \mid \tau * \tau \\
v & ::= & \ldots \mid (v, v)
\end{array}
$$

$$\frac{e_1 \rightarrow e_1'}{(e_1, e_2) \rightarrow (e_1', e_2)} \qquad \frac{e_2 \rightarrow e_2'}{(v_1, e_2) \rightarrow (v_1, e_2')}$$

$$\frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \qquad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2}$$

$$\frac{}{(v_1, v_2).1 \rightarrow v_1} \qquad \frac{}{(v_1, v_2).2 \rightarrow v_2}$$

Small-step can be a pain (more concise notation exists)

Pairs continued

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

Canonical Forms: If $\cdot \vdash v : \tau_1 * \tau_2$, then v has the form (v_1, v_2).

Progress: New cases using C.F. are $v.1$ and $v.2$.

Preservation: For primitive reductions, inversion gives the result *directly*.

Records

Records seem like pairs with *named fields*

$$\begin{array}{rcl} e & ::= & \ldots \mid \{l_1 = e_1; \ldots; l_n = e_n\} \mid e.l \\ \tau & ::= & \ldots \mid \{l_1 : \tau_1; \ldots; l_n : \tau_n\} \\ v & ::= & \ldots \mid \{l_1 = v_1; \ldots; l_n = v_n\} \end{array}$$

Fields do *not* α-convert.

Names might let us reorder fields, e.g.,
$\cdot \vdash \{l_1 = 42; l_2 = \textbf{true}\} : \{l_2 : \textbf{bool}; l_1 : \textbf{int}\}$.

Nothing wrong with this, but many languages disallow it.
(Why? Run-time efficiency and/or type inference)

(O'Caml has only *named* record types with *disjoint* fields.)

More on this when we study *subtyping*

Base Types, in general

What about floats, strings, enums, ...? Could add them all or do something more general...

Parameterize our language/semantics by a collection of *base types* $(b_1, \ldots, b_n)$ and *primitives* $(c_1 : \tau_1, \ldots, c_n : \tau_n)$.

Examples: concat : string→string→string
toInt : float→int
"hello" : string

For each primitive, *assume* if applied to values of the right types it produces a value of the right type.

Together the types and assumed steps tell us how to type-check and evaluate $c_i\ e$ where c_i is a primitive.

We can prove soundness *once and for all* given the assumptions.

Sums

What about ML-style datatypes:

```
type t = A | B of int | C of int*t
```

1. Tagged variants (i.e., discriminated unions)
2. Recursive types
3. Type constructors (e.g., `type 'a bst = ...`)
4. Names the type

Today we'll model just (1) with (anonymous) sum types:

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid \mathsf{case}\ e\ x.e \mid x.e \\
\tau & ::= & \ldots \mid \tau_1 + \tau_2 \\
v & ::= & \ldots \mid \mathsf{inl}(v) \mid \mathsf{inr}(v)
\end{array}
$$

Sum semantics

$$\overline{\textsf{case inl}(v)\ x.e_1 \mid x.e_2 \rightarrow e_1[v/x]}$$

$$\overline{\textsf{case inr}(v)\ x.e_1 \mid x.e_2 \rightarrow e_2[v/x]}$$

$$\frac{e \rightarrow e'}{\textsf{inl}(e) \rightarrow \textsf{inl}(e')} \qquad \frac{e \rightarrow e'}{\textsf{inr}(e) \rightarrow \textsf{inr}(e')}$$

$$\frac{e \rightarrow e'}{\textsf{case}\ e\ x.e_1 \mid x.e_2 \rightarrow \textsf{case}\ e'\ x.e_1 \mid x.e_2}$$

case has binding occurrences, just like pattern-matching.

Sum Type-checking

Inference version (not trivial to infer; can require annotations)

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \qquad \Gamma, x{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ e\ x.e_1 \mid x.e_2 : \tau}$$

C.F.: If $\cdot \vdash v : \tau_1 + \tau_2$, then either v has the form $\mathsf{inl}(v_1)$ and $\cdot \vdash v_1 : \tau_1$ or ...

The rest is induction and substitution.

Can encode booleans with sums. E.g., $\mathsf{bool} = \mathsf{int} + \mathsf{int}$, $\mathsf{true} = \mathsf{inl}(0)$, $\mathsf{false} = \mathsf{inr}(0)$.

Recursion

We won't prove it, but every extension so far preserves termination. A Turing-complete language needs some sort of loop. What we add won't be encodable in STλC.

E.g., `let rec f x =` e

Do typed recursive functions need to be bound to variables or can they be anonymous?

In O'Caml, you need variables, but it's unnecessary:

$$e ::= \ldots \mid \textsf{fix}\ e$$

$$\frac{e \to e'}{\textsf{fix}\ e \to \textsf{fix}\ e'} \qquad \frac{}{\textsf{fix}\ \lambda x.\ e \to e[\textsf{fix}\ \lambda x.\ e/x]}$$

Using fix

It works just like `let rec`, e.g.,

$$\textbf{fix } \lambda f.\ \lambda n.\ \textbf{if } n < 1 \textbf{ then } 1 \textbf{ else } n * (f(n-1))$$

Note: You can use it for mutual recursion too.

Pseudo-math digression

Why is it called fix? In math, a fixed-point of a function g is an x such that $g(x) = x$.

Let g be $\lambda f.\ \lambda n.$ **if** $n < 1$ **then** 1 **else** $n * (f(n-1))$.

If g is applied to a function that computes factorial for arguments $\leq m$, then g returns a function that computes factorial for arguments $\leq m + 1$.

Now g has type $(\textbf{int} \rightarrow \textbf{int}) \rightarrow (\textbf{int} \rightarrow \textbf{int})$. The fix-point of g is the function that computes factorial for *all* natural numbers.

And **fix** g is equivalent to that function. That is, **fix** g is the fix-point of g.

Typing fix

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \textbf{fix } e : \tau}$$

Math explanation: If e is a function from τ to τ, then **fix** e, the fixed-point of e, is some τ with the fixed-point property. So it's something with type τ.

Operational explanation: **fix** $\lambda x.\ e'$ becomes $e'[$**fix** $\lambda x.\ e'/x]$. The substitution means x and **fix** $\lambda x.\ e'$ better have the same type. And the result means e' and **fix** $\lambda x.\ e'$ better have the same type.

Note: Proving soundness is straightforward!

General approach

We added lets, booleans, pairs, records, sums, and fix. Let was syntactic sugar. Fix made us Turing-complete by "baking in" self-application. The others *added types*.

Whenever we add a new form of type τ there are:

- Introduction forms (ways to make values of type τ)
- Elimination forms (ways to use values of type τ)

What are these forms for functions? Pairs? Sums?

When you add a new type, think "what are the intro and elim forms"?