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 $\text{ST} \lambda C$
  (pairs, records, sums, recursion, \ldots)
  - For each, sketch proof additions
  - At the end, discuss the general approach
- Not today: References, exceptions, polymorphism, lists, \ldots
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 λs 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:

$$
\begin{align*}
\frac{D_1}{\Gamma \vdash e_1 : \tau' \rightarrow \tau} & \quad \frac{D_2}{\Gamma \vdash e_2 : \tau'} \\
& \quad \frac{\Gamma \vdash e_1 \ e_2 : \tau}
\end{align*}
$$

1 Preservation subcase: If $e_1 \ e_2 \rightarrow e'_1 \ e_2$, inverting that derivation means:

$$
\begin{align*}
\frac{D}{e_1 \rightarrow e'_1} & \quad \frac{e_1 \ e_2 \rightarrow e'_1 \ e_2}
\end{align*}
$$
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:

\[
\frac{\mathcal{D}_3}{\Gamma \vdash e'_1 : \tau' \rightarrow \tau} \quad \frac{\mathcal{D}_2}{\Gamma \vdash e_2 : \tau'} \quad \frac{}{\Gamma \vdash e'_1 \ e_2 : \tau}
\]

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)

\[ e ::= \ldots \mid \text{let } x = e_1 \text{ in } e_2 \]

\[
e_1 \rightarrow e_1' \\
\text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e_1' \text{ in } e_2
\]

\[
\text{let } x = v \text{ in } e_2 \rightarrow e_2[v/x]
\]

\[
\Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \vdash e_2 : \tau \\
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau
\]

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 \( \lambda \), so can make it a derived form:

\[
\text{let } x = e_1 \text{ in } e_2 \text{ a "macro" (derived form) } (\lambda x. \ e_2) \ e_1.
\]

(Harder (?) if \( \lambda \) needs explicit type.)

Or just define the semantics to replace let with \( \lambda \):

\[
\text{let } x = e_1 \text{ in } e_2 \rightarrow (\lambda x. \ e_2) \ e_1
\]

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 \( \lambda \) differently. (Later.)

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

\[ e ::= \ldots \mid \text{true} \mid \text{false} \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

\[ \tau ::= \ldots \mid \text{bool} \quad v ::= \ldots \mid \text{true} \mid \text{false} \]

\[ e_1 \rightarrow e_1' \]

\[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3 \]

\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \]

\[ \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \]

\[ \Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{false} : \text{bool} \]

Notes: CBN, new Canonical Forms case, all lemma cases easy
Pairs (CBV, left-right)

\[
\begin{align*}
e &::= \ldots \mid (e, e) \mid e.1 \mid e.2 \\
\tau &::= \ldots \mid \tau \ast \tau \\
v &::= \ldots \mid (v, v)
\end{align*}
\]

\[
\begin{align*}
e_1 \rightarrow e_1' &
\phantom{\rightarrow} \quad (e_1, e_2) \rightarrow (e_1', e_2) \\
(v_1, e_2) &\rightarrow (v_1, e_2')
\end{align*}
\]

\[
\begin{align*}
e &\rightarrow e' \\
\phantom{\rightarrow} \quad e_1 \rightarrow e_1' \\
(v_1, v_2).1 &\rightarrow v_1 \\
(v_1, v_2).2 &\rightarrow v_2
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2 \\
\Gamma \vdash e : \tau_1 \ast \tau_2 & \quad \Gamma \vdash e : \tau_1 \ast \tau_2 \\
\Gamma \vdash e.1 : \tau_1 & \quad \Gamma \vdash e.2 : \tau_2
\end{align*}
\]

Canonical Forms: If \( \cdot \vdash v : \tau_1 \ast \tau_2 \), 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 \textit{directly}.
Records

Records seem like pairs with *named fields*

\[
e ::= \ldots \mid \{l_1 = e_1; \ldots; l_n = e_n\} \mid e.1
\]

\[
\tau ::= \ldots \mid \{l_1 : \tau_1; \ldots; l_n : \tau_n\}
\]

\[
v ::= \ldots \mid \{l_1 = v_1; \ldots; l_n = v_n\}
\]

Fields do *not* \(\alpha\)-convert.

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

*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: \(\text{concat} : \text{string} \rightarrow \text{string} \rightarrow \text{string}\)
\(\text{toInt} : \text{float} \rightarrow \text{int}\)
\(\text{“hello”} : \text{string}\)

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:

\[
\text{type } t = A \mid B \text{ of } \text{int} \mid C \text{ of } \text{int} \times t
\]

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:

\[
\begin{align*}
e & ::= \ldots \mid \text{inl}(e) \mid \text{inr}(e) \mid \text{case } e \ x.e \mid x.e \\
\tau & ::= \ldots \mid \tau_1 + \tau_2 \\
v & ::= \ldots \mid \text{inl}(v) \mid \text{inr}(v)
\end{align*}
\]
Sum semantics

\[
\text{case } \text{inl}(v) \ x.e_1 | x.e_2 \rightarrow e_1[v/x]
\]

\[
\text{case } \text{inr}(v) \ x.e_1 | x.e_2 \rightarrow e_2[v/x]
\]

\[
e \rightarrow e' \\
\text{inl}(e) \rightarrow \text{inl}(e')
\]

\[
e \rightarrow e' \\
\text{inr}(e) \rightarrow \text{inr}(e')
\]

\[
e \rightarrow e' \\
\text{case } e \ x.e_1 | x.e_2 \rightarrow \text{case } e' \ x.e_1 | x.e_2
\]

case has binding occurrences, just like pattern-matching.
Sum Type-checking

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

\[
\begin{align*}
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash \text{inl}(e) : \tau_1 + \tau_2 \\
\Gamma \vdash e : \tau_2 \\
\Gamma \vdash \text{inr}(e) : \tau_1 + \tau_2 \\
\Gamma \vdash e : \tau_1 + \tau_2 \\
\Gamma, x:\tau_1 \vdash e_1 : \tau \\
\Gamma, x:\tau_2 \vdash e_2 : \tau \\
\Gamma \vdash \text{case } e \ x.e_1 \mid x.e_2 : \tau
\end{align*}
\]

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

The rest is induction and substitution.

Can encode booleans with sums. E.g., \( \text{bool} = \text{int} + \text{int} \), \( \text{true} = \text{inl}(0) \), \( \text{false} = \text{inr}(0) \).
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 ST\(\lambda\)C.

E.g., let \texttt{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:

\[
e ::= \ldots | \text{fix } e
\]

\[
\begin{align*}
e \rightarrow e' & \\
\text{fix } e \rightarrow \text{fix } e' & \\
\text{fix } \lambda x. e \rightarrow e[\text{fix } \lambda x. e/x]
\end{align*}
\]
Using fix

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

\[
\text{fix } \lambda f. \lambda n. \text{ if } n < 1 \text{ then } 1 \text{ else } n \ast (f(n - 1))
\]

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 \( \lambda f. \, \lambda n. \, \text{if } n < 1 \text{ then } 1 \text{ else } n \ast (f(n - 1)) \).

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 \((\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})\). The fix-point of \( g \) is the function that computes factorial for all natural numbers.

And \textbf{fix } \( g \) is equivalent to that function. That is, \textbf{fix } \( g \) is the fix-point of \( g \).
Typing fix

\[ \Gamma \vdash e : \tau \rightarrow \tau \]
\[ \Gamma \vdash \text{fix } e : \tau \]

Math explanation: If \( e \) is a function from \( \tau \) to \( \tau \), then \( \text{fix } e \), the fixed-point of \( e \), is some \( \tau \) with the fixed-point property. So it’s something with type \( \tau \).

Operational explanation: \( \text{fix } \lambda x. e' \) becomes \( e'[\text{fix } \lambda x. e'/x] \). The substitution means \( x \) and \( \text{fix } \lambda x. e' \) better have the same type. And the result means \( e' \) and \( \text{fix } \lambda x. e' \) 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 $\tau$ there are:

- Introduction forms (ways to make values of type $\tau$)
- Elimination forms (ways to use values of type $\tau$)

What are these forms for functions? Pairs? Sums?

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