Types for Imperative Programs
[redacted]

Adam Chlipala
MIT
Based on slides by Armando Solar-Lezama & George Necula

6.820
October 2, 2013
Related Reading

On operational semantics for “IMP”: Chapter 2 of Winksel, “Introduction to Operational Semantics”

On lambda-calculus with references: Chapter 13 of Pierce, “References”
Big-Step OS for $\lambda$ calculus

- Configuration is simply a lambda expression.
  - there is no state.
- Result is a different lambda expression.

- Inductive definition: Base case
  \[
  x \rightarrow x
  \]

- Inductive definition: recursive cases
  \[
  \frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'} \quad \frac{??}{e_1 e_2 \rightarrow e_3}
  \]
The same techniques apply to programs with state.
  - The big difference is that the configuration now includes state.

Example: IMP

\[ e := n \mid x \mid e_1 + e_2 \]
\[ c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]
• Rules for expressions are very similar to what we had before.

• We need a rule to assign values to variables.
Big-Step OS for Imperative Programs

- Commands mutate the state.

- What about loops?
Big-Step OS for Imperative Programs

• The definition for loops must be recursive.
Small-Step Semantics

• Many design decisions
  – How small is a step?
  – How do we select the next step?

• These decisions need to be defined formally.
Redex [review & extension]

• A redex is an expression that can be reduced in one atomic step.

• The first step in defining a small-step semantics is to define the redexes.

• Ex.
  - In IMP: \( n_1 + n_2 \mid x = n \mid \text{skip}; c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \)
  - In \( \lambda \)-calculus: \( (\lambda x. e_1) \, v \, , \, (\lambda x. e_1) \, e_2 \)
Local reduction rules

- One for each redex
  - show how to advance one step of the execution.

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\>
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\) where n = n1 + n2
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x \rightarrow n]>\>
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\>
- \(<\text{if true then } c1 \text{ else } c2, \sigma> \rightarrow <c1, \sigma>\>
- \(<\text{if false then } c1 \text{ else } c2, \sigma> \rightarrow <c2, \sigma>\>
- \(<\text{while } b \text{ do } c, \sigma> \rightarrow <\text{if } b \text{ then } (c; \text{ while } b \text{ do } c) \text{ else } \text{skip}, \sigma>\>
Global reduction rules

• A simple algorithm:
  - start with a program.
  - identify a redex.
  - reduce according to local reduction rules.
  - repeat until you can’t reduce anymore.

• We need rules to define the next redex.
**Contexts** [which should be rather familiar after PS2!]

- We use $H$ to refer to a context.
- $H[r]$ is a program fragment consisting of redex $r$ in context $H$.

- Global reduction rules can be defined from local reduction rules as flows.

- If $<r, \sigma> \rightarrow <e, \sigma'>$ then $<H[r], \sigma> \rightarrow <H[e], \sigma'>$.

- How we define the set of contexts will determine the order in which local reductions are applied.
Example

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := (x + 1) + 2, [x=2]&gt;</td>
<td>x = (□ + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = □ + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = □</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>□</td>
<td>x:=5</td>
</tr>
<tr>
<td>&lt;skip, [x=5]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole.
Contexts

• Contexts are defined by a grammar.

\[ H ::= \square | n + H | H + e | x:= H \]

• The grammar defines the evaluation order.
  - Note in \( a + b \), \( a \) is evaluated before \( b \).

• We can define redexes and contexts to
  - define the order of evaluation.
  - define short circuit behavior.
Contexts

• How do we know if our contexts and redexes are well-defined?
ML-Style References

\[
\begin{align*}
\tau & ::= \text{int} \mid \tau \rightarrow \tau \mid \tau \text{ ref} \\
\text{e} & ::= n \mid x \mid \text{e e} \mid \lambda x. \ e \mid \text{ref e} \mid \text{e := e} \mid \text{!e}
\end{align*}
\]

Examples:

\[
\begin{align*}
& (\lambda f : \text{int }\rightarrow (\text{int ref}). \ !(f \; 5)) (\lambda x : \text{int} . \ \text{ref } x) \\
& (\lambda x : \text{int ref}. \ x := 7; \ !x) (\text{ref } 42)
\end{align*}
\]

Big difference from pure lambda calculus:

Program behavior isn't "just algebra" anymore!
Modeling the Heap

Heaps $h$ are partial functions from $locations$ to values.

Reduction rules for redexes:

$$(h, (\lambda x. e) v) \rightarrow (h, [v/x]e)$$
Typing Rules

\[
\begin{align*}
\Gamma(x) &= \tau \\
\Gamma^h; \Gamma |- x : \tau &\quad \text{Assigns types to locations!} \\
\Gamma^h; \Gamma |- \ell : \tau \\
\Gamma^h; \Gamma |- e_1 e_2 : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma^h(\ell) &= \tau \\
\Gamma^h; \Gamma |- \ell : \tau \\
\Gamma^h; \Gamma, x : \tau' |- e : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma^h; \Gamma |- \lambda x. e : \tau' \rightarrow \tau
\end{align*}
\]
A Heap Typing Judgment

\[
\Gamma^h; \Gamma \vdash x : \tau
\]

Assigns types to locations!
The Key Invariant for Type Soundness

\[ \text{Type soundness proof recipe:} \]

**Progress:**
Every good state is either finished or not stuck.

**Preservation:**
Steps preserve goodness.

\[ \text{Good}(h, e, \tau) = \]
### Important Administrative Lemmas

Coq code for this lecture gives a formal type soundness proof. Highlights of the key lemmas:

<table>
<thead>
<tr>
<th><strong>Standard lemmas</strong></th>
<th><strong>References-specific lemmas</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weakening:</strong></td>
<td><strong>Heap Weakening:</strong></td>
</tr>
<tr>
<td>If ( \Gamma^h; \Gamma \vdash e : \tau ) and ( \Gamma \subseteq \Gamma' ) then ( \Gamma^h; \Gamma' \vdash e : \tau )</td>
<td>If ( \Gamma^h; \Gamma \vdash e : \tau ) and ( \Gamma^h \subseteq \Gamma^{h'} ) then ( \Gamma^{h'}; \Gamma \vdash e : \tau )</td>
</tr>
</tbody>
</table>

| **Substitution:**   | ...and plenty more fiddly lemmas about manipulation of heaps... |
| If \( \Gamma^h; \Gamma, x : \tau' \vdash e : \tau \) and \( \Gamma^h \vdash e' : \tau' \) then \( \Gamma^h; \Gamma \vdash [e'/x]e : \tau \) |
References and Polymorphism

Classic type soundness bug in ML
(a language family we'll learn about via OCaml shortly):