Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2015
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

\[
\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'} \quad \frac{e_1 e_2 \rightarrow e_3}{??}
\]
Big Step OS for Imperative Programs

• The same techniques apply to programs with state
  – The big difference is that the configuration now includes state

• Example: IMP
  
  \[
  e := n \mid x \mid e_1 + e_2 \mid e_1 == e_2 \mid \text{True} \mid \text{False}
  \]

  \[
  c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \mid \text{skip}
  \]

• Now we need two types of judgments
  expressions result in values                      commands change the state

\[
\langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma'
\]
Big Step OS for Imperative Programs

- Rules for expressions are very similar to what we had before

\[
\begin{align*}
\langle N, \sigma \rangle &\rightarrow n \\
\hline
\langle e_1, \sigma \rangle &\rightarrow n_1 \quad \langle e_2, \sigma \rangle &\rightarrow n_2 \quad n = n_1 + n_2 \\
\hline
\langle e_1 + e_2, \sigma \rangle &\rightarrow n
\end{align*}
\]

- We need a rule to read values from variables

\[
\langle x, \sigma \rangle \rightarrow \sigma(x)
\]
Big Step OS for Imperative Programs

• Commands mutate the state

\[
\begin{align*}
\langle e, \sigma \rangle &\rightarrow e' \\
\langle X := e, \sigma \rangle &\rightarrow \sigma[X \rightarrow e']
\end{align*}
\]

\[
\begin{align*}
\langle c_1, \sigma \rangle &\rightarrow \sigma'' \\
\langle c_2, \sigma'' \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow false \\
\langle c_f, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle \rightarrow \sigma'
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow true \\
\langle c_t, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle \rightarrow \sigma'
\]

• What about loops?
Big Step OS for Imperative Programs

• The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle \rightarrow false & \quad \Rightarrow \quad \langle while \ e_1 \ then \ c, \sigma \rangle \rightarrow \sigma \\
\langle e_1, \sigma \rangle \rightarrow true \quad \langle c; \ while \ e_1 \ then \ c, \sigma \rangle \rightarrow \sigma' & \quad \Rightarrow \quad \langle while \ e_1 \ then \ c, \sigma \rangle \rightarrow \sigma' \\
\langle e_1, \sigma \rangle \rightarrow true \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle while \ e_1 \ then \ c, \sigma'' \rangle \rightarrow \sigma' & \quad \Rightarrow \quad \langle while \ e_1 \ then \ c, \sigma \rangle \rightarrow \sigma'
\end{align*}
\]
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

- 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: \((\lambda x. v) \ e_2\ , \ (\lambda x. e_1) \ e_2\)
Local reduction rules

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

- \( \langle x, \sigma[x = n] \rangle \rightarrow \langle n, \sigma \rangle \)
- \( \langle n_1 + n_2, \sigma \rangle \rightarrow \langle n, \sigma \rangle \) where \( n = n_1 + n_2 \)
- \( \langle x := n, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[x \rightarrow n] \rangle \)
- \( \langle \text{skip}; c, \sigma \rangle \rightarrow \langle c, \sigma \rangle \)
- \( \langle \text{if} \, \text{true} \, \text{then} \, c_1 \, \text{else} \, c_2, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle \)
- \( \langle \text{if} \, \text{false} \, \text{then} \, c_1 \, \text{else} \, c_2, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle \)
- \( \langle \text{while} \, b \, \text{do} \, c, \sigma \rangle \rightarrow \langle \text{if} \, b \, \text{then} \, (c; \, \text{while} \, b \, \text{do} \, c) \, \text{else} \, \text{skip}, \sigma \rangle \)
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

• 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, σ> ↦ <e, σ'> then <H[r], σ> ↦ <H[e], σ'>

• 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 = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>o</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

\[ H ::= o \mid n + H \mid H + e \mid x := H \]
\[ \mid \text{if } H \text{ then } c_1 \text{ else } c_2 \mid H; c \]

- 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?

• Decomposition theorem:
  If c is not “skip”, then there exist unique H and r such that c is H[r]
  – Exist guarantees progress
  – Unique guarantees determinism
ML Style References

- Adding references
  \[
  \tau ::= \ldots \mid \tau \text{ref}
  \]
  \[
  e ::= \ldots \mid \text{ref}\; e \mid e_1 := e_2 \mid e_1; e_2 \mid ! e
  \]

- Example:
  \[
  (\lambda f: \text{int} \to (\text{int ref}). \; ! (f\; 5))\; (\lambda x: \text{int}. \; \text{ref}\; x)
  \]
  \[
  (\lambda x: \text{int ref}. \; x := 7; ! x)\; \text{ref}\; x
  \]

- Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset \mid h, a \rightarrow \text{val}: \tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- **New contexts** (in addition to the ones before)
  - $H := \text{ref } H \mid H := e \mid \text{addrs} := H \mid !H$

- **No new local reduction rules**

- **New global reduction rules**
  - $\text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a]$
  
  - $\text{heap } h \text{ in } H[!a] \rightarrow \text{heap } h \text{ in } H[v]$
    - As long as $a \rightarrow v \in h$
    - $\text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*]$
Additional typing rules for references

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \Gamma \vdash (\text{ref } e : \tau) : \tau \text{ref} \\
\Gamma \vdash e : \tau \text{ref} & \quad \Gamma \vdash \! e : \tau \\
\Gamma \vdash e_1 : \tau \text{ref} \quad \Gamma \vdash e_2 : \tau & \quad \Gamma \vdash e_1 := e_2 : \text{unit}
\end{align*}
\]
References and polymorphism

\[
\begin{align*}
\text{let } x: \forall t. (t \to t)ref &= \Lambda t. \text{ref } (\lambda x: t. x) \\
\text{in } x[\text{bool}]: &= \lambda x: \text{bool. not } x \\
(\neg x[\text{int}]) 5
\end{align*}
\]

- This is a big problem
- Solution: Disallow side effects in let.