Types for Imperative Programs

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
  \[ e \rightarrow e' \]
  \[ \lambda x. e \rightarrow \lambda x. e' \]
  \[ e_1 e_2 \rightarrow e_3 \]
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 | x | e + e
  c := x := e | c ; c | if e then c else c | while e do c | 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.

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

• We need a rule to assign values to variables.

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

- Commands mutate the state.

\[
\frac{\langle e, \sigma \rangle \rightarrow e'}{\langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e']}
\]

Read as “0”

\[
\frac{\langle e_1, \sigma \rangle \rightarrow false \quad \langle c_f, \sigma \rangle \rightarrow \sigma'}{\langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle \rightarrow \sigma'}
\]

- What about loops?

\[
\frac{\langle c_1, \sigma \rangle \rightarrow \sigma'' \quad \langle c_2, \sigma'' \rangle \rightarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma'}
\]

Read as “nonzero”

\[
\frac{\langle e_1, \sigma \rangle \rightarrow true \quad \langle c_t, \sigma \rangle \rightarrow \sigma'}{\langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle \rightarrow \sigma'}
\]
Big-Step OS for Imperative Programs

• The definition for loops must be recursive.

\[
\langle e_1, \sigma \rangle \rightarrow \text{false} \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma
\]

\[
\langle e_1, \sigma \rangle \rightarrow \text{true} \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'
\]

\[
\langle e_1, \sigma \rangle \rightarrow \text{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \text{while } e_1 \text{ then } c, \sigma'' \rangle \rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'
\]
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, σ> → <σ(x), σ>`
  - `<n1+n2, σ> → <n, σ>` \( \text{where } n = n1 + n2 \)
  - `<x := n, σ> → <skip, σ[x→ n]>`
  - `<skip; c, σ> → <c, σ>`
  - `<if true then c1 else c2, σ> → <c1, σ>`
  - `<if false then c1 else c2, σ> → <c2, σ>`
  - `<while b do c, σ> → <if b then (c; while b do c) else skip, σ>`
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

The context is a program with a hole.

<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 := (\square + 1) + 2)</td>
<td>(x)</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2])&gt;</td>
<td>(x := \square + 2)</td>
<td>(2 + 1)</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2])&gt;</td>
<td>(x := \square)</td>
<td>(3 + 2)</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2])&gt;</td>
<td>(\square)</td>
<td>(x:=5)</td>
</tr>
<tr>
<td>(&lt;\text{skip, [x=5]})&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contexts

• Contexts are defined by a grammar.

\[ H ::= □ | n + H | H + e | x := H \\
   | \text{if} \ H \ \text{then} \ c_1 \ \text{else} \ c_2 | H; c \\
   | \text{while} \ H \ \text{do} \ c \]

• 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

\[ \tau ::= \text{int} | \tau \rightarrow \tau | \tau \text{ ref} \]

\[ e ::= n | x | e \ e | \lambda x. \ e | \text{ref } e | e ::= e | !e \]

Examples:

\[ (\lambda f : \text{int } \rightarrow (\text{int ref}). ! (f \ 5)) (\lambda x : \text{int}. \text{ref } x) \]

\[ (\lambda x : \text{int ref}. \ x ::= 7; !x) (\text{ref } 42) \]

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:
- $(h, (\lambda x. e) \, v) \rightarrow (h, [v/x]e)$
- $(h, \text{ref } v) \rightarrow (h[\ell = v], \ell)$ if $\ell \not\in \text{dom}(h)$
- $(h, !\ell) \rightarrow (h, h(\ell))$ if $\ell \in \text{dom}(h)$
- $(h, \ell := v) \rightarrow (h[\ell = v], v)$ if $\ell \in \text{dom}(h)$

Evaluation contexts (call-by-value):
$$C ::= \square | C \, e | v \, C \mid \text{ref } C \mid !C \mid C := e \mid v := C$$

We add locations as a special case of expressions!
(They aren't allowed to appear in initial programs.)

Top-level step rule:
$$\frac{}{(h, e) \rightarrow (h', e')}$$
$$\frac{}{(h, C[e]) \rightarrow (h', C[e'])}$$
Typing Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Logical Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma(x) = \tau )</td>
<td>( \Gamma^h; \Gamma</td>
</tr>
<tr>
<td>( \Gamma^h; \Gamma</td>
<td>- \ell : \tau )</td>
</tr>
<tr>
<td>( \Gamma^h; \Gamma</td>
<td>- e_1 e_2 : \tau )</td>
</tr>
<tr>
<td>( \Gamma^h; \Gamma</td>
<td>- e_1 : \tau )</td>
</tr>
<tr>
<td>( \Gamma^h; \Gamma</td>
<td>- ref e : \tau )</td>
</tr>
<tr>
<td>( \Gamma^h; \Gamma</td>
<td>- e_1 : \tau )</td>
</tr>
<tr>
<td>( \Gamma^h; \Gamma</td>
<td>- e_2 : \tau )</td>
</tr>
<tr>
<td>( \Gamma^h; \Gamma</td>
<td>- \lambda x. e : \tau' \rightarrow \tau )</td>
</tr>
</tbody>
</table>
A Heap Typing Judgment

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

Assigns types to locations!

Notice this circularity within the heap!

\[ \text{dom}(\Gamma^h) = \text{dom}(h) \quad \forall \ell \in \text{dom}(\Gamma^h). \quad \Gamma^h; \cdot \vdash h(\ell) : \Gamma^h(\ell) \]

\[ \Gamma^h \vdash h \]

Sidebar: how to use said circularity to implement \textbf{recursion}

let r = ref (\lambda x: int. x) in
r := (\lambda x. !r(x));
!r(0)
The Key Invariant for Type Soundness

Good(h, e, τ) =
\exists \Gamma^h. \Gamma^h \vdash h \land \Gamma^h; \cdot \vdash e : \tau

Execution has finished safely when:
e is a value, as usual
(but now locations are values, too).

We capture “non-stuckness” as:
\exists h', e'. (h, e) \rightarrow (h', e')

Type soundness proof recipe:
Progress:
Every good state is either finished or not stuck.
Preservation:
Steps preserve goodness.
Important Administrative Lemmas

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

**Standard lemmas**

**Weakening:**
If $\Gamma^h; \Gamma |- e : \tau$
and $\Gamma \subseteq \Gamma'$
then $\Gamma^h; \Gamma' |- e : \tau$

**Substitution:**
If $\Gamma^h; \Gamma, x : \tau' |- e : \tau$
and $\Gamma^h; \cdot |- e' : \tau'$
then $\Gamma^h; \Gamma |- [e'/x]e : \tau$

**References-specific lemmas**

**Heap Weakening:**
If $\Gamma^h; \Gamma |- e : \tau$
and $\Gamma^h \subseteq \Gamma^{h'}$
then $\Gamma^{h'}; \Gamma |- e : \tau$

...and plenty more fiddly lemmas about manipulation of heaps...
References and Polymorphism

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

```
let x: ∀t. (t → t) ref = ref (λx. x) in
x := λx: bool. not x;
!x(5)
```

ML solution, the **value restriction**: Only generalize τ in “let x : τ = e in ...” as polymorphic when e is a **value**.

This is an *overapproximation* of the real requirement: e can't be allowed to have certain kinds of side effects.