Abstract Machines

What is an abstract machine?
- a set of legal states
  - final and initial states as subsets
- a set of instructions altering the states of the machine
- it should be possible to emulate the operations on a real machine in a finite number of steps

Why use abstract machines at all?
- useful to exactly specify the semantics of a programming language (SOS)
- can be used to facilitate porting to other architectures
- mobile code
  - Java Virtual Machine

Implementing the single step semantics of MinHs in Haskell:

Slide 3

The C-machine

We define a new abstract machine with
- explicit control stack
- explicit handling of control flow
called the C-machine

Variable binding is still handled by substitution.

Note: this is a variant of the C-machine defined in the textbook
The C-machine

The machine state consists of

- the current expression
- a control stack of subcomputations (frames) which have to be performed before the computation terminates

**Initial states:**
- stack is empty

**Final states:**
- current expression is a value
- stack is empty

---

**Example: Addition**

1. Evaluate first argument
   - first argument becomes current expression
   - remember to continue with computation, result as first argument

2. Evaluate second argument
   - second argument becomes current expression
   - remember to continue with computation, result as second argument

---

**Term-representation of frames:**

The term

\[
\text{plus}(v_1, v_2)
\]

represents a suspended computation of an addition, waiting for the value of its first argument.

---

**Syntax of Frames:**

- addition (multiplication etc similarly):

  \[
  e \text{ expr} \\
  \text{plus}(\emptyset, e) \text{ frame}
  \]

- if-expression:

  \[
  c_1 \text{ expr} \ c_2 \text{ expr} \\
  \text{if}(\emptyset, c_1, c_2) \text{ frame}
  \]

- application:

  \[
  v \text{ value} \ e \text{ expr} \\
  \text{apply}(\emptyset, v) \text{ frame}
  \]
Syntax of Stacks: We write $f_1 \triangleright f_2 \triangleright \circ$ to denote a stack with frame $f_1$ as the top-most frame, $f_2$ as second.

- \( \circ \) stack

\[
\begin{array}{c}
 f \text{ frame } \circ \text{ stack } \\
 f \triangleright s \text{ stack }
\end{array}
\]

States of the machine:
- \( S = S_1 \cup S_2 \), where
  - \( S_1 = \{ s > e, s \text{ stack}, e \text{ expr} \}: \text{ evaluate } e \text{ under stack } s \)
  - \( S_2 = \{ s < v, s \text{ stack} \}: \text{ return } v \text{ to stack } s \)
- \( I = \{ o > e \} \)
- \( F = \{ o < v \} \)

Transition Rules for MinHS
- Values (integers, booleans, functions)

\( s > v \rightarrow s < v \)

Slide 9

Slide 10

Transition Rules for MinHS
- Addition

\[
\begin{align*}
 s > \text{plus}(e_1, e_2) &\rightarrow s \triangleright \text{plus}(f_1, e_2) \\
 \text{plus}(f_1, e_2) &\rightarrow s < v \rightarrow \text{plus}(f_1, f_2) \\
 \text{plus}(f_1, f_2) &\rightarrow s > e_1 \\
 \text{plus}(f_1, f_2) &\rightarrow s < \text{num}(e_1) \rightarrow s < \text{num}(e_1 + e_2)
\end{align*}
\]

- If-expressions

\[
\begin{align*}
 s > \text{if}(e_1, e_2, e_3) &\rightarrow s \triangleright \text{if}(f_1, e_2, e_3) \\
 \text{if}(f_1, e_2, e_3) &\rightarrow s < \text{true} \rightarrow s \triangleright e_1 \\
 \text{if}(f_1, e_2, e_3) &\rightarrow s < \text{false} \rightarrow s \triangleright e_2
\end{align*}
\]

Slide 11

Function application

\[
\begin{align*}
 s > \text{apply}(e_1, e_2) &\rightarrow s \triangleright \text{apply}(f_1, e_2) \\
 \text{apply}(f_1, e_2) &\rightarrow s < \text{apply}(f_1, f_2) \\
 \text{apply}(f_1, f_2) &\rightarrow s > e_1 \\
 \text{apply}(f_1, f_2) &\rightarrow s < \text{apply}(f_1, f_2) \\
 \text{apply}(f_1, f_2) &\rightarrow s > \{ \text{fun}(\tau_1, \tau_2, f, x, e), f \} \{ v / x \}
\end{align*}
\]

Slide 12
Observations:
- all the inference rules are axioms!
- the definition of single step evaluation in the C-machine is not recursive
- the full evaluator is only tail recursive

**Slide 14**

**Environments**
How can we extend the C-machine to use an environment instead of substitution?
- We cannot just pass it along
- Can we use the stack to keep track of the environment?

**Slide 15**

**E-MACHINE**

Frames: as before

Environment:

<table>
<thead>
<tr>
<th>* env</th>
<th>v env</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = v</td>
<td></td>
</tr>
</tbody>
</table>

Stack:

<table>
<thead>
<tr>
<th>o stack</th>
<th>s stack</th>
<th>f frame</th>
<th>f s stack</th>
<th>η env</th>
<th>s stack</th>
<th>η s stack</th>
</tr>
</thead>
</table>

States:
- $S = \{ s \mid η > e \} \cup \{ s \mid η < v \}$
- Initial States: $\{ o \mid * > e \}$
- Final States: $\{ o \mid * < v \}$

**First Attempt:**
- Free variables:
  
  $s \mid η > x \mapsto s \mid η < v$

  If $x = v \in η$

  - Application:
    
    $\text{apply}(\text{fun}(\tau_1, \tau_2, f.x.v), \square) \mapsto s \mid η < v \mapsto s \mid f = \text{fun}(\ldots), x = v, η > e$

  - Returning values:
    
    $\eta \mapsto s \mid η' < v \mapsto s \mid η < v$
Are these rules correct? Let us try and see what happens when we evaluate the following two examples.

1. a simple function application:
   \[
   \text{apply(fun(int, int, f.x.plus(x, 1))), 3}
   \]

2. and a nested application (corresponds to a function which accepts two arguments and returns the first one):
   \[
   \text{apply(apply(fun(int -> int, f.x.fun(int, int, g.y.x)), 3))}
   \]

Since the type information is not relevant for the evaluation, we omit it in the following. To further save some space, we abbreviate apply to app, and write \(n\) instead of \(\text{num}(n)\)

---

**Example 1:**

\[
\begin{align*}
   & o \mid \ast \Rightarrow \text{app}(\text{fun}(f.x.plus(x, 1)), 3) \\
   & \text{app}([\square, 3]) \mid \ast \Rightarrow \text{fun}(f.x.plus(x, 1)) \\
   & \text{app}([\square, 3], o) \mid \ast \leftarrow \text{fun}(f.x.plus(x, 1)) \\
   & \text{app}(\text{fun}(f.x.plus(x, 1)), o) \mid \ast \leftarrow 3 \\
   & \ast \triangleright o \mid x = 3, f = \text{fun}(f.x.plus(x, 1)) \mid \ast \leftarrow \text{plus}(x, 1) \\
   & \text{plus}([\square, 1]) \triangleright o \mid x = 3, f = \text{fun}(f.x.plus(x, 1)) \mid \ast \leftarrow x \\
   & \text{plus}([\square, 1], o) \triangleright o \mid x = 3, f = \text{fun}(f.x.plus(x, 1)) \mid \ast \leftarrow 3 \\
   & \text{plus}(3, [\square]) \triangleright o \mid x = 3, f = \text{fun}(f.x.plus(x, 1)) \mid \ast \leftarrow 1 \\
   & \text{plus}(3, [\square], o) \mid x = 3, f = \text{fun}(f.x.plus(x, 1)) \mid \ast \leftarrow 1 \\
   & \ast \triangleright o \mid x = 3, f = \text{fun}(f.x.plus(x, 1)) \mid \ast \leftarrow 4 \\
   & o \mid \ast \leftarrow 4
\end{align*}
\]

---

**Problem:** Incorrect results for partially applied functions!

**Solution:** Bundle functions with the current environment

We add a new type of expression which is only used in the operational semantics:

\[
\langle \eta, \text{fun}(\tau_1, \tau_2, f.x.e) \rangle
\]

**New rules:**

- Returning function values:
  \[
  s \mid \eta \Rightarrow \text{fun}(\tau_1, \tau_2, f.x.e) \Rightarrow_\eta s \mid \eta \leftarrow \langle \eta, \text{fun}(\tau_1, \tau_2, f.x.e) \rangle
  \]

- Application of functions:
  \[
  \text{apply}([\langle \eta', \text{fun}(\tau_1, \tau_2, f.x.e) \rangle], [\square]) \triangleright s \mid \eta \leftarrow v \Rightarrow \eta \triangleright s \mid f = \text{fun}(...) \mid x = v, \eta' \triangleright e
  \]