**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

**Control Flow**

We defined the transition system of the single step semantics for MinHs in terms of a very high-level abstract machine\(^a\)

- substitution as "machine operations"
- can be avoided by using environment (see TinyC)
- control flow not explicit

- search rules determine next subexpression to be evaluated
- finding the next evaluable subexpression can be expensive

\(^a\)We’ll call this machine \textit{M-machine}

**Implementing the single step semantics of MinHs in Haskell:**

\[
\text{eval (Num n)} = \text{Num n} \\
\text{eval e} = \text{eval (evalSingle e)} \\
\text{evalSingle (Plus (Num n1, Num n2))} = \text{Num (n1 + n2)} \\
\text{evalSingle (Plus (Num n1, e))} = \text{Plus (Num n1, evalSingle e)} \\
\text{evalSingle (Plus (e1, e2))} = \text{Plus (evalSingle e1, e2)} \\
\text{evalSingle (Times \ldots)}
\]

- for each step, the expression has to be traversed
- makes heavy use of the Haskell runtime stack

**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

\[
\text{plus}([\square], e_2)
\]

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

Syntax of Frames:

- addition (multiplication etc similarly):

\[
\begin{align*}
\text{expr} & \quad \text{plus}([\square], e) \quad \text{frame} \\
\text{v} & \quad \text{plus}([\text{v}, e]) \quad \text{frame}
\end{align*}
\]

- if-expression:

\[
\begin{align*}
\text{c_1} & \quad \text{expr} & \quad \text{c_2} & \quad \text{expr} \\
\text{if}([\square], c_1, c_2) & \quad \text{frame}
\end{align*}
\]

- application:

\[
\begin{align*}
\text{v} & \quad \text{expr} \\
\text{apply}([\square], \text{v}) & \quad \text{frame}
\end{align*}
\]
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
- \( f \) frame \( \triangleright \) stack

States of the machine:

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

Transition Rules for MinHs

- Values (integers, booleans, functions)
  - \( s > v \rightleftharpoons s < v \)

- Addition
  - \( s > \text{plus}(e_1, e_2) \rightleftharpoons \text{plus}(e_1, e_2) \triangleright s > e_1 \)
  - \( \text{plus}(e_1, e_2) \triangleright s < v \rightleftharpoons \text{plus}(e_1, e_2) \triangleright s > e_2 \)
  - \( \text{plus} / (n_1). \triangleright s < \text{num}(n_2) \rightleftharpoons s < \text{num}(n_1 + n_2) \)

- If-expressions
  - \( s > \text{if}(e_1, e_2, e_3) \rightleftharpoons \text{if}(e_1, e_2, e_3) \triangleright s > e_1 \)
  - \( \text{if}(e_1, e_2, e_3) \triangleright s < \text{true} \rightleftharpoons s > e_2 \)
  - \( \text{if}(e_1, e_2, e_3) \triangleright s < \text{false} \rightleftharpoons s > e_3 \)

- Function application
  - \( s > \text{app}(e_1, e_2) \rightleftharpoons \text{app}(e_1, e_2) \triangleright s > e_1 \)
  - \( \text{app}(e_1, e_2) \triangleright s < v \rightleftharpoons \text{app}(e_1, e_2) \triangleright s > e_2 \)
  - \( \text{app}(\text{fun}(n_1, n_2, f, x, c), \triangleright s < v \rightleftharpoons \{ \text{fun}(n_1, n_2, f, x, c) / f \}{v/x}c \)
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 13**

**E-MACHINE**

Frames: as before

Environment:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Frame</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\circ \stack$</td>
<td>$f \triangleright s \stack$</td>
<td>$\eta \triangleright s \stack$</td>
</tr>
</tbody>
</table>

States:
- $S = \{ s \mid \eta \triangleright \epsilon \} \cup \{ s \mid \eta \triangleright v \}$
- Initial States: $\{ \circ \mid \star \triangleright \epsilon \}$
- Final States: $\{ \circ \mid \star \triangleright v \}$

**Slide 15**

First Attempt:
- Free variables:

  $s \mid \eta \triangleright x \mapsto q \mid \eta \triangleright v$

  If $x = v \in \eta$

  Application:

  $\text{apply}(\text{fun}(\tau_1, \tau_2, f; x; e), [\square] \triangleright s \mid \eta \triangleright v \mapsto \eta \triangleright s \mid f = \text{fun}(\ldots), x = v, \eta \triangleright e$

  Returning values:

  $\eta \triangleright s \mid \eta' \triangleright v \mapsto q \mid \eta \triangleright v$
Are these rules correct? Let us try and see what happens when we evaluate the following two examples.

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

2. and a nested application (corresponds to a function which accepts two arguments and returns the first one):
   \[
   \text{apply}(\text{apply}(\text{fun(int \rightarrow int, f.x fun(int, int, g.y.x)), 3)) 4)
   \]

Since the type information is not relevant for the evaluation, we abbreviate apply to app, and write n instead of \text{num}(n)

Example 1:
\[
O | * > \text{app}(\text{fun(f.x plus(x, 1)), 3)}
\]
\[
\text{app}([n, 3]) | * > \text{fun(f.x plus(x, 1))}
\]
\[
\text{app}([n, 3]) o | * < \text{fun(f.x plus(x, 1))}
\]
\[
\text{app}(\text{fun(f.x plus(x, 1)), [n, n]} o | * > 3
\]
\[
* o | x = 3, f = \text{fun(f.x plus(x, 1))}, * > \text{plus(x, 1)}
\]
\[
\text{plus}([n, 1]) o | * o | x = 3, f = \text{fun(f.x plus(x, 1))}, * > x
\]
\[
\text{plus}([n, 1]) o | * o | x = 3, f = \text{fun(f.x plus(x, 1))}, * > x
\]
\[
\text{plus}([3, 1]) o | * o | x = 3, f = \text{fun(f.x plus(x, 1))}, * > 3
\]
\[
\text{plus}(3, [n, 1]) o | * o | x = 3, f = \text{fun(f.x plus(x, 1))}, * > 1
\]
\[
\text{plus}(3, [n, 1]) o | * o | x = 3, f = \text{fun(f.x plus(x, 1))}, * > 1
\]
\[
* o | x = 3, f = \text{fun(f.x plus(x, 1))}, * < 4
\]
\[
o | * < 4
\]

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:
  \[
  s | \eta > \text{fun(\tau_1, \tau_2, f.x.e)} \leftarrow s | \eta < \langle \eta, \text{fun(\tau_1, \tau_2, f.x.e)} \rangle
  \]
- Application of functions:
  \[
  \text{apply}(\langle \eta', \text{fun(\tau_1, \tau_2, f.x.e)} \rangle, [n]) o | \eta < v \leftarrow \eta o | f = \text{fun(...), x = v, \eta' \succ e}
  \]