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)
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
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
  - Jave 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”

\(^a\)We’ll call this machine $M$-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)

\(^a\)We’ll call this machine M-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 M-machine
Implementing the single step semantics of MinHs in Haskell:

\[
\begin{align*}
\text{eval} \ (\text{Num} \ n) &= \text{Num} \ n \\
\text{eval} \ e &= \text{eval} \ (\text{evalSingle} \ e)
\end{align*}
\]

\[
\begin{align*}
\text{evalSingle} \ (\text{Plus} \ (\text{Num} \ n1, \ \text{Num} \ n2)) &= \\
&= \text{Num} \ (n1 + n2) \\
\text{evalSingle} \ (\text{Plus} \ (\text{Num} \ n1, \ e)) &= \\
&= \text{Plus} \ (\text{Num} \ n1, \ \text{evalSingle} \ e) \\
\text{evalSingle} \ (\text{Plus} \ (e1, \ e2)) &= \\
&= \text{Plus} \ (\text{evalSingle} \ e1, \ e2) \\
\text{evalSingle} \ (\text{Times} \ (\ldots \\
\end{align*}
\]

→ 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 machine state consists of
THE C-MACHINE

The machine state consists of

- the current expression

Initial states:

- stack is empty

Final states:

- current expression is a value
- stack is empty
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
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

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

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

→ addition (multiplication etc similarly):

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

→ if-expression:

\[
\begin{align*}
\text{if}(\square, e_1, e_2) & \quad \text{frame}
\end{align*}
\]

→ application:

\[
\begin{align*}
\text{apply}(\square, e) & \quad \text{frame}
\end{align*}
\]
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.

\[ \begin{align*}
\circ & \text{ stack} \\
\hline
f & \text{ frame} \quad s & \text{ stack} \\
\hline
f & \triangleright s & \text{ stack}
\end{align*} \]
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.

\[ f \text{ frame} \quad s \text{ stack} \]
\[ f \triangleright s \text{ stack} \]

States of the machine:
\[ S = S_1 \cup S_2, \text{ where} \]
\[ S_1 = \{ s \triangleright e, s \text{ stack}, e \text{ expr} \}: \text{ evaluate } e \text{ under stack } s \]
\[ S_2 = \{ s \prec v, s \text{ stack} \}: \text{ return } v \text{ to stack } s \]
\[ I = \{ \circ \triangleright e \} \]
\[ F = \{ \circ \prec v \} \]
TRANSITION RULES FOR MINHS

→ Values (integers, booleans, functions)

\[ s \succ v \mapsto_c \]

→ Addition
TRANSITION RULES FOR MINHS

➔ Values (integers, booleans, functions)

\[ s > v \rightarrow_c s < v \]

➔ Addition
**Transition Rules for MinHs**

- Values (integers, booleans, functions)

\[ s \succ v \quad \rightarrow_c \quad s \prec v \]

- Addition

\[ s \succ \text{plus}(e_1, e_2) \quad \rightarrow_c \]
TRANSITION RULES FOR MINHS

→ Values (integers, booleans, functions)

\[ s > v \iff_c s < v \]

→ Addition

\[ s > \text{plus}(e_1, e_2) \iff_c \text{plus}(\square, e_2) \triangleright s \succ e_1 \]

\[ \text{plus}(\square, e_2) \triangleright s < v \iff_c \]
**Transition Rules for MinHs**

→ Values (integers, booleans, functions)

\[
\begin{align*}
\text{Values} & \rightarrow v 
\end{align*}
\]

→ Addition

\[
\begin{align*}
\text{Addition} & \rightarrow \text{plus}(e_1, e_2) \\
\text{plus}(\square, e_2) & \rightarrow \text{plus}(\square, e_2) \triangleright s \succ e_1 \\
\text{plus}(\square, e_2) & \triangleright s \prec v \\
\text{plus}(v, \square) & \rightarrow \text{plus}(v, \square) \triangleright s \succ e_2
\end{align*}
\]
**Transition Rules for MinHs**

- **Values** (integers, booleans, functions)

  \[ s > v \rightarrow_c s < v \]

- **Addition**

  \[ s > \text{plus}(e_1, e_2) \rightarrow_c \text{plus}(\square, e_2) \triangleright s > e_1 \]

  \[ \text{plus}(\square, e_2) \triangleright s < v \rightarrow_c \text{plus}(v, \square) \triangleright s > e_2 \]

  \[ \text{plus}(\text{num}(n_1), \square) \triangleright s < \text{num}(n_2) \rightarrow_c \]
TRANSITION RULES FOR MINHS

→ Values (integers, booleans, functions)

\[ s \succ v \rightarrow_\mathsf{c} s \prec v \]

→ Addition

\[ s \succ \text{plus}(e_1, e_2) \rightarrow_\mathsf{c} \text{plus}(e_2, e_2) \succ s \succ e_1 \]

\[ \text{plus}(e_2, e_2) \succ s \prec v \rightarrow_\mathsf{c} \text{plus}(v, e_2) \succ s \succ e_2 \]

\[ \text{plus}(\text{num}(n_1), e_2) \succ s < \text{num}(n_2) \rightarrow_\mathsf{c} s < \text{num}(n_1 + n_2) \]
If-expressions

\[ s \Rightarrow \text{if}(\text{e}_1, \text{e}_2, \text{e}_3) \iff_c \text{if}(\text{\square}, \text{e}_2, \text{e}_3) \Rightarrow s \Rightarrow \text{e}_1 \]

\[ \text{if}(\text{\square}, \text{e}_2, \text{e}_3) \Rightarrow s < \text{true} \iff_c s \Rightarrow \text{e}_2 \]

\[ \text{if}(\text{\square}, \text{e}_2, \text{e}_3) \Rightarrow s < \text{false} \iff_c s \Rightarrow \text{e}_3 \]
Function application

\[
\begin{align*}
&s \triangleright apply(e_1, e_2) \mapsto_c apply(\square, e_2) \triangleright s \triangleright e_1 \\
&apply(\square, e_2) \triangleright s < v \mapsto_c apply(v, \square) \triangleright s \triangleright e_2 \\
&apply(fun(\tau_1, \tau_2, f, x, e), \square) \triangleright s < v \mapsto_c s \triangleright \{fun(\tau_1, \tau_2, f, x, e)/f\}{v/x}e
\end{align*}
\]
Observations:

- all the inference rules are axioms!
- the definition of single step evaluation in the C-machine is not recursive
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
How can we extend the C-machine to use an environment instead of substitution?
How can we extend the C-machine to use an environment instead of substitution?

→ We cannot just pass it along
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?
E-MACHINE

Frames: as before

Environment:

\[
\begin{align*}
\text{\texttt{env}} & \quad \text{\texttt{env}} \\
\ast \text{\texttt{env}} & \quad \eta \text{\texttt{env}} \\
\end{align*}
\]
\[x = v, \eta \text{\texttt{env}}\]

Stack:

\[
\begin{align*}
\text{\texttt{stack}} & \quad \text{\texttt{frame}} \\
\circ \text{\texttt{stack}} & \quad f \triangleright s \text{\texttt{stack}} \\
\eta \text{\texttt{env}} & \quad s \text{\texttt{stack}} \\
\end{align*}
\]

States:

\[
\begin{align*}
\rightarrow S & = \{ s \mid \eta \triangleright e \} \cup \{ s \mid \eta \prec v \} \\
\rightarrow \text{Initial States:} & \quad \{ \circ \mid \ast \triangleright e \} \\
\rightarrow \text{Final States:} & \quad \{ \circ \mid \ast \prec v \}
\end{align*}
\]
First Attempt:

→ Free variables:

\[ s \mid \eta \succ x \mapsto_{E} s \mid \eta \prec v \]

if \( x = v \in \eta \)

→ Application:

apply(fun(\( \tau_1, \tau_2, f.x.e \), \(\square\)) \(\triangleright s \mid \eta \prec v \mapsto_{E} \eta \triangleright s \mid f = \text{fun}(\ldots), x = v, \eta \succ e \))

→ Returning values:

\( \eta \triangleright s \mid \eta' \prec v \mapsto_{E} s \mid \eta \prec v \)
Are these rules correct? Let us try and see what happens when we evaluate the following two examples.

① a simple function application:

\[ \text{apply}(\text{fun}(\text{int, int, } f.x.\text{plus}(x, 1)), 3) \]

② and a nested application (corresponds to a function which accepts two arguments and returns the first one):

\[ \text{apply}(\text{apply}(\text{fun}(\text{int } \rightarrow \text{ int, int, } f.x.\text{fun}(\text{int, int, } g.y.x)), 3))4 \]

Since the type information is not relevant for the evaluation, we omit it in the following. To further save some space, we abbreviate \text{apply} to \text{app}, and write \( n \) instead of \text{num}(n)
Example 1:
\[
\begin{align*}
& o \mid * \triangleright app(fun(f.x.plus(x, 1)), 3) \\
\mapsto_{E} & app(\Box, 3) \mid * \triangleright fun(f.x.plus(x, 1)) \\
\mapsto_{E} & app(\Box, 3) \triangleright o \mid * \prec fun(f.x.plus(x, 1)) \\
\mapsto_{E} & app(fun(f.x.plus(x, 1)), \Box) \triangleright o \mid * \triangleright 3 \\
\mapsto_{E} & app(fun(f.x.plus(x, 1)), \Box) \triangleright o \mid * \prec 3 \\
\mapsto_{E} & * \triangleright o \mid x = 3, f = fun(f.x.plus(x, 1)), * \triangleright plus(x, 1) \\
\mapsto_{E} & plus(\Box, 1) \triangleright * \triangleright o \mid x = 3, f = fun(f.x.plus(x, 1)), * \triangleright x \\
\mapsto_{E} & plus(\Box, 1) \triangleright * \triangleright o \mid x = 3, f = fun(f.x.plus(x, 1)), * \prec 3 \\
\mapsto_{E} & plus(3, \Box) \triangleright * \triangleright o \mid x = 3, f = fun(f.x.plus(x, 1)), * \triangleright 1 \\
\mapsto_{E} & plus(3, \Box) \triangleright * \triangleright o \mid x = 3, f = fun(f.x.plus(x, 1)), * \prec 1 \\
\mapsto_{E} & * \triangleright o \mid x = 3, f = fun(f.x.plus(x, 1)), * \prec 4 \\
\mapsto_{E} & o \mid * \prec 4
\end{align*}
\]
something went wrong here! By returning the function value and restoring the old (empty) environment, we threw away the binding for the variable $x$. It now occurs freely in $g$!
Problem: incorrect results for partially applied functions!
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:

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

\[
s \mid \eta \succ \text{fun}(\tau_1, \tau_2, f.x.e) \leftrightarrow_E s \mid \eta \prec \langle \eta, \text{fun}(\tau_1, \tau_2, f.x.e) \rangle
\]

- Application of functions:

\[
\text{apply}(\langle \eta', \text{fun}(\tau_1, \tau_2, f.x.e) \rangle, \square) \triangleright s \mid \eta \prec v \leftrightarrow_E \eta \triangleright s \mid f = \text{fun}(\ldots), x = v, \eta' \succ e
\]