What is a Computer?

State of the machine.

- Internal registers, memory, etc..
- Initial and final state.

What is a Computer?

A computer is a transition system.

- Set of states $S$.
- Initial states $I \subseteq S$.
- Final states $F \subseteq S$.
- Relation $\mathrel{\rightsquigarrow} \subseteq S \times S$.

Execution $=$ transition sequence.

The M Machine

States: closed MinML expressions.

- Initial: any well-typed, closed expression.
- Final: closed values.

Transitions: $\rightsquigarrow^\text{M}$ is given by SOS rules.

Evaluation: $e \rightsquigarrow^\text{M} v$. 

The M Machine

The M Machine is very high-level, in two senses:

- **Control.** Complex search rules specify order of execution. Rely on a “metastack” to manage search.
- **Data.** Parameter passing is by substitution, which is complex and generates “new” code on the fly.
Managing Control
The M Machine “cheats” by relying on implicit storage management.

- Search rules have one or more premises that must be applied recursively.
- Interpreter is not tail recursive (iterative).

“Real” machines cannot rely on implicit storage management!

The C Machine
The state of the C Machine is a pair $(k, e)$, where
- $k$ is a control stack, or continuation;
- $e$ is a closed expression.

The transition relation $\rightarrow_C$ is defined inductively by a set of rules.

Managing Control
For example,
\[
\begin{align*}
e_1 \rightsquigarrow_M e'_1 \\
\text{apply}(e_1, e_2) \rightsquigarrow_M \text{apply}(e'_1, e_2)
\end{align*}
\]
To apply this rule, we must
1. Save the current state: $\text{apply}(\Box, e_2)$.
2. Execute a step in $e_1$, obtaining $e'_1$.
3. Restore the state: $\text{apply}(e'_1, e_2)$.

The C Machine: Control Stacks
A control stack, or continuation, is a composition of stack frames:

Think of pushing a frame onto the control stack.

C Machine States
The state of the C Machine takes one of two forms:
- an evaluation state $k > e$ corresponds to evaluating closed expression $e$ relative to control stack $k$
- a return state $k < v$ corresponds to evaluating stack $k$ relative to closed value $v$

The separator points to the focal entry of the state.
The C Machine: Functions

First we evaluate the function:

\[ k > \text{apply}(e_1, e_2) \rightarrow C \text{apply}(\square, e_2); k > e_1 \]

...and then the argument:

\[ \text{apply}(\square, e_2); k < v_1 \rightarrow C \text{apply}(v_1, \square); k > e_2 \]

...and then perform the application:

\[ \text{apply}(v_1, \square); k < v_2 \rightarrow C k > \{v_1, v_2/f, x\}e \]

where \( v_1 = \text{fun}(x:t_1); t_2 \text{ is end} \).

The C Machine: Stack Frames

A stack frame records one pending computation:

\[ e_2 \text{ expr} \quad \begin{array}{l} \text{frame} \\ +(\square, e_2) \quad \text{frame} \\ +e_1 \text{ expr} \\ \text{frame} \\ \text{frame} \end{array} \]

\[ e_2 \text{ expr} \quad \begin{array}{l} \text{frame} \\ \text{frame} \end{array} \]

\[ \text{v1 value} \quad \begin{array}{l} \text{frame} \\ \text{frame} \end{array} \]

Think of a frame as an abstract return address; the "\( \square \)" marks the return point.

The C Machine: Booleans

First we evaluate the test:

\[ k > \text{if}\tau e \text{then} e_1 \text{else} e_2 \text{fi} \rightarrow C \text{if}\tau \square \text{then} e_1 \text{else} e_2 \text{fi}; k > e \]

...and then decide how to proceed:

\[ \text{if}\tau \square \text{then} e_1 \text{else} e_2 \text{fi}; k < \text{true} \rightarrow C k > e_1 \]

\[ \text{if}\tau \square \text{then} e_1 \text{else} e_2 \text{fi}; k < \text{false} \rightarrow C k > e_2 \]

The C Machine: Values

When a closed expression has been reduced to a closed value, the evaluation state steps to a return state:

\[ k > n \rightarrow C k < n \]

\[ k > \text{true} \rightarrow C k < \text{true} \]

\[ k > \text{false} \rightarrow C k < \text{false} \]

\[ k > \text{fun} f(x:t_1); t_2 \text{ is end} \rightarrow C k < \text{fun} f(x:t_1); t_2 \text{ is end} \]

The C Machine: Numbers

We evaluate the first argument:

\[ k > +(e_1, e_2) \rightarrow C +(\square, e_2); k > e_1 \]

...and then the second argument:

\[ +(\square, e_2); k < v_1 \rightarrow C +(v_1, \square); k > e_2 \]

...and then we execute the addition:

\[ +(n_1, \square); k < n_2 \rightarrow C k < n_1 + n_2 \]

Relating the C and the M Machines

To recover the whole program from the C Machine state we unwind the stack around the current expression:

\[ e \circ e = e \]

\[ +(\square, e_2); k \circ e_1 = k \circ +(e_1, e_2) \]

\[ +(v_1, \square); k \circ e_2 = k \circ +(v_1, e_2) \]

\[ \text{if}\tau \square \text{then} e_1 \text{else} e_2 \text{fi}; k \circ e = k \circ \text{if}\tau e_1 \text{else} e_2 \text{fi} \]

\[ \text{apply}(\square, e_2); k \circ e = k \circ \text{apply}(e, e_2) \]

\[ \text{apply}(e_1, \square); k \circ e = k \circ \text{apply}(e_1, e) \]
Relating the C and the M Machines

Each frame of the stack corresponds to a search rule.

- For example, \( \text{apply}(\square, e) \) corresponds to searching within the function position.
- All frames have this property.

Relating the C and the M Machines

Instruction execution on the C machine corresponds to

- Finding that instruction using the M machine search rules.
- Executing that instruction "in place".
- Resuming execution from there.

Relating the C and the M Machines

What about the converse?

Suppose that \( e \mapsto_{M} e' \) — we've executed an instruction somewhere inside of \( e \). But where?

Moreover, this step may occur as a subgoal of executing some other step, \( i.e. \), as the premise of a search rule.

Relating the C and the M Machines

Pushing a frame onto the stack does not change the complete program.

- We're just shifting program fragments onto the stack.
- Unwinding the stack results in the same program.

Theorem 1

If \( (k, e) \mapsto_{C} (k', e') \), then either

1. \( k \otimes e = k' \otimes e' \), or
2. \( k \otimes e \mapsto_{M} k' \otimes e' \).

The first case covers pushing frames onto the control stack; the second covers instruction execution.

Corollary 2

If \( e > e \mapsto_{C} e < v \), then \( e \mapsto_{M} v \).

Relating the C and the M Machines

Idea: if \( e \mapsto_{M} e' \), then any complete execution from \( e' \) in any context determines a corresponding complete execution from \( e \) in that context.

Theorem 3

If \( e \mapsto_{M} e' \) and \( k > e' \mapsto_{C} e < v \), then \( k > e \mapsto_{C} e < v \).

Corollary 4

If \( e \mapsto_{M} v \), then \( e > e \mapsto_{C} e < v \).
Relating the C and the M Machines

Suppose that \( \epsilon = \text{apply}(e_1, e_2) \) and \( \epsilon' = \text{apply}(e'_1, e_2) \) where \( e_1 \mapsto_M e'_1 \).

Suppose that \( k > \epsilon' \mapsto_{\epsilon} \epsilon < v \). Therefore we have

\[
\begin{align*}
\epsilon > \epsilon' &\mapsto_{\epsilon} \text{apply}(\square, e_2); k > e'_1 \\
\mapsto_{\epsilon} &\epsilon < v 
\end{align*}
\]

Then we calculate

\[
\begin{align*}
k > \epsilon &\mapsto_{\epsilon} \text{apply}(\square, e_2); k > e'_1 \\
\mapsto_{\epsilon} &\epsilon < v 
\end{align*}
\]

The second step is by the inductive hypothesis, using the enlarged stack.

Summary

Abstract machines come in all shapes and sizes. They differ in how many details of execution are made explicit.

The C Machine manages control explicitly.