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 $\trianglerighteq S \times S$.

Execution $= \text{transition sequence}$.

The M Machine

States: closed MinML expressions.

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

Transitions: $\trianglerighteq$ is given by SOS rules.

Evaluation: $\epsilon \trianglerighteq v$.

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 C Machine makes the flow of control explicit in the state of the machine.

- Explicit control stack, or continuation, that manages the flow of control.
- Transition rules will have no premises — fully iterative (tail recursive) implementation.

The state of the C Machine is a pair $(k, \epsilon)$, where

- $k$ is a control stack, or continuation;
- $\epsilon$ is a closed expression.

The transition relation $\trianglerighteq$ is defined inductively by a set of rules.
Dynamic vs. Static Scope
Consider the following ML code:

```ml
val apply_pair = fn f => fn x => f (x, x)
val constantly = fn x => fn y => x
val k3 = constantly 3
val g = apply_pair k3
val z = g 4
```

What is the value of z?

Managing Binding
It is unrealistic to substitute arguments for parameters in function bodies.

- Replicates data structures, rather than sharing them.
- Corresponds to run-time code generation, which is not the typical implementation technique.

Idea: make the management of variable binding explicit.

Managing Binding
A complication: values are no longer just expressions!

- Integers, booleans remain as before.
- Functions are represented as closures = code + environment.

This distinction will also affect the representation of the control stack, which will also involve closures.

Managing Binding
To evaluate an application `apply(fun f (x:int):int is e end, e)`,

1. **Bind** `x` to `e` in the environment.
2. Execute `e` in the extended environment.
3. Restore the environment by popping the binding for `x`.

The environment is sometimes called the **data stack**; it is often combined with the control stack into a unified control and data stack.

Managing Binding
But this is a fallacy!

- The environment is not necessarily a stack.
- The **locus classicus** of this mistake is Lisp, where it is called dynamic scoping.

The environment must sometimes **persist** beyond the lifetime of a function call.

Dynamic vs. Static Scope
Employing the M Machine, we calculate:

- Substituting 3 for `x` in constantly, we obtain `fn y => 3` for `k3`.
- Substituting `fn y=>3` for `f` in `apply_pair`, we obtain `fn x => (fn y => 3) (x, x)` for `g`.
- Substituting 4 for `x` in `g`, we obtain `(fn y => 3) (4, 4)`.
- Evaluating, we obtain 3 for `z`.
Dynamic vs. Static Scope

Employing the environment model, we proceed as follows:

- Binding 3 to x in constantly, we obtain fn y => x for k3.
- Popping the binding for x, we bind f to fn y => x in apply.pair to obtain fn x => f(x, x) for g.
- Popping the binding for f, we bind x to 4 in g to obtain f(x, x) for z.
- But now we are in serious trouble! There is no binding for f!

Dynamic vs. Static Scope

Continuing, ...

- Evaluating (x, x) we obtain (4, 4).
- Binding (4, 4) to y we evaluate x.
- Evaluating x, we obtain 4 for z.

This is not the right answer!

Dynamic vs. Static Scope

First moral: we cannot drop bindings of parameters!

- Binding x to 3 in constantly, we obtain fn y => x for k3.
- Binding f to fn y => x in apply.pair, we obtain fn x => f(x, x) for g.
- Binding x to 4 in g, we obtain f(x, x).

Dynamic vs. Static Scope

Second moral: we must be careful to distinguish different variables with the same name!

- The second binding for x captured the occurrence of x within the binding of k3!
- If we rename x to y in the definition of apply.pair, the problem disappears.

Dynamic vs. Static Scope

Static scope:

- Resolution of scope is performed by a syntactic analysis (nearest enclosing binding).
- Evaluation respects α-conversion, hence modular.
Dynamic vs. Static Scope

Dynamic scope:

- Resolution of scope is performed dynamically, during execution. The most recent binding in the execution order is used.
- Does not respect α-conversion, and hence non-modular.

This is what makes lisp, sb, tcl, and tex code so hard to use and maintain.

The E Machine: Values

The set of machine values is inductively defined by these rules:

- \( \text{true, false, mvalue} \)
- \( \text{fun f (x:τc):τd is e end}[^{\eta}] \text{ mvalue} \)

The E Machine: Environments

An E Machine environment, \( \eta \), is a finite function mapping variables to machine values.

That is, an environment is a sequence of bindings of variables to machine values with no variable bound more than once.

<table>
<thead>
<tr>
<th>( \eta \text{ menv} )</th>
<th>( x # \eta )</th>
<th>( V \text{ mvalue} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon \text{ menv} )</td>
<td>( \eta, x = V \text{ menv} )</td>
<td></td>
</tr>
</tbody>
</table>

The E Machine: Stacks

The set of machine stacks is inductively defined by these rules:

- \( \epsilon \text{ mstack} \)
- \( F \text{ mframe K mstack} \)
- \( \text{F m K mstack} . \)

A machine stack is therefore a sequence of machine frames.

The E Machine: Frames

The set of machine frames is inductively defined by these rules:

- \( \epsilon \text{ mframe} \)
- \( +((\text{mframe}[c_2][\eta]) \text{ mframe} \)
- \( +(V_1, \text{mframe}) \text{ mframe} \)
- \( \text{if} \text{f then e_1 else e_2}[\eta] \text{ mframe} \)
- \( \text{apply}((\text{mframe}[c_2][\eta]) \text{ mframe} \)
- \( \text{apply}(V_1, \text{mframe}) \text{ mframe} \)

The main difference compared to the C Machine is the association of environments to frames that contain unevaluated expressions.
The E Machine: States

The state of the E Machine takes one of two forms:

- an evaluation state \( K > e \ [\eta] \) corresponds to evaluating expression \( e \) on stack \( K' \) relative to environment \( \eta \).

- a return state \( K < V \) corresponds to returning the value \( V \) to the stack \( K \).

The E Machine: Numbers

\[ K > n \ [\eta] \mapsto E K < n \]

\[ K > WRec, edS \ [\eta] \mapsto E WR\Box, edS[\eta]m K > ec \ [\eta] \]

\[ WR\Box, edS[\eta]m K < Vc \mapsto E WRVc, \BoxSm K > ed \ [\eta] \]

\[ WRnc, \BoxSm K < nd \mapsto E K < nc W nd \]

The E Machine: Functions

\[ K > \text{fun} f(x: \tau_c): \tau_d \text{is e end} \ [\eta] \mapsto E K < \text{fun} f(x: \tau_c): \tau_d \text{is e end}[\eta] \]

\[ K > \text{apply}(ec, ed) \ [\eta] \mapsto E \]

\[ \text{apply}(V, \Box) \ [\eta] \mapsto E \]

\[ \text{where} V = \text{fun} f(x: \tau_1): \tau_2 \text{is e end}[\eta]. \]

The E Machine: Variables

To evaluate a variable \( x \), we look up its binding and pass it to the stack:

\[ K > x \ [\eta] \mapsto E K < \eta(x) \]

The E Machine: Booleans

\[ K > \text{true} \ [\eta] \mapsto E K < \text{true} \]

\[ K > \text{false} \ [\eta] \mapsto E K < \text{false} \]

\[ K > \text{if} _r e \ \text{then} \ e_1 \ \text{else} \ e_2 \ [\eta] \mapsto E \]

\[ \text{if} _r \Box \text{then} e_1 \text{else} e_2 fi[\eta]; K > e \ [\eta] \]

\[ \text{if} _r \Box \text{then} e_1 \text{else} e_2 fi[\eta]; K < \text{true} \mapsto E K > e_1 \ [\eta] \]

\[ \text{if} _r \Box \text{then} e_1 \text{else} e_2 fi[\eta]; K < \text{false} \mapsto E K > e_2 \ [\eta] \]

The E Machine: States

The state of the w Machine takes one of two forms:

- an evaluation state \( \eta \)

The result closure of the environment evaluates to the stack.

The E Machine: Final States

The final states of the E machine have the form \( \epsilon < V \), with final result \( V \) where \( V \) value.

If \( V \) is a number or boolean, it can just be printed. If it is a closure, ML just prints \( \epsilon \), rather than attempting to expand the environment and print the code.
The E and the M Machines

Proposition 1

For closed expressions \( e \) of base type, \( e \xrightarrow{M} v \iff e \mid [v] \xrightarrow{E} e < v \)

The proof is complex, and is therefore omitted.

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, the E Machine manages both control and binding.

Dynamic scope is a bad idea because it violates fundamental principles of modularity.