510: Programming Languages

# Abstract Machines

David Walker

Fall, 2002

# What is a Computer?

State of the machine.

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

## What is a Computer?

Instructions for transforming the state.

- Must be **effective**, usually **simple**.

- Rules for determining order of execution.

Execution = instruction by instruction state transformation.

## 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 $\mapsto \subseteq S \times S$.

Execution = transition sequence.

# The M Machine

States: closed MinML expressions.

- Initial: any well-typed, closed expression.

- Final: closed values.

Transitions: $\mapsto_{\mathsf{M}}$ is given by SOS rules.

Evaluation: $e \mapsto^{*}_{\mathsf{M}} v$.

## 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 one or more premises that must be applied recursively.

- Interpreter is not tail recursive (iterative).

"Real" machines cannot rely on implicit storage management!

# Managing Control

For example,

$$\frac{e_1 \mapsto_{\mathsf{M}} e_1'}{\texttt{apply}(e_1, e_2) \mapsto_{\mathsf{M}} \texttt{apply}(e_1', e_2)}$$

To apply this rule, we must

1. Save the current state: $\texttt{apply}(\Box, e_2)$.

2. Execute a step in $e_1$, obtaining $e_1'$.

3. Restore the state: $\texttt{apply}(e_1', e_2)$.

## 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 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** $\mapsto_{\mathsf{C}}$ is defined inductively by a set of rules.

## The C Machine: Control Stacks

A **control stack**, or **continuation**, is a composition of **stack frames**:

$$\frac{}{\bullet\ \mathsf{stack}} \qquad \frac{f\ \mathsf{frame} \qquad k\ \mathsf{stack}}{f \triangleright k\ \mathsf{stack}}$$

Think of pushing a frame onto the control stack.

# The C Machine: Stack Frames

A **stack frame** records one pending computation:

$$\frac{e_2 \ \textsf{expr}}{+(\Box, e_2) \ \textsf{frame}} \qquad \frac{v_1 \ \textsf{value}}{+(v_1, \Box) \ \textsf{frame}}$$

$$\frac{e_1 \ \textsf{expr} \quad e_2 \ \textsf{expr}}{\texttt{if}_\tau \, \Box \, \texttt{then} \, e_1 \, \texttt{else} \, e_2 \, \texttt{fi} \ \textsf{frame}}$$

$$\frac{e_2 \ \textsf{expr}}{\texttt{apply}(\Box, e_2) \ \textsf{frame}} \qquad \frac{v_1 \ \textsf{value}}{\texttt{apply}(v_1, \Box) \ \textsf{frame}}$$

Think of a frame as an abstract return address; the "$\Box$" marks the return point.

## The C Machine: Numbers

We evaluate the first argument:

$$(k, +(e_1, e_2)) \mapsto_{\mathsf{C}} (+(\Box, e_2) \triangleright k, e_1)$$

... and then the second argument:

$$(+(\Box, e_2) \triangleright k, v_1) \mapsto_{\mathsf{C}} (+(v_1, \Box) \triangleright k, e_2)$$

... and then we execute the addition:

$$(+(n_1, \Box) \triangleright k, n_2) \mapsto_{\mathsf{C}} (k, n_1 + n_2)$$

# The C Machine: Booleans

First we evaluate the test:

$$(k, \texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}) \mapsto_{\mathsf{C}}$$
$$(\texttt{if}_\tau\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \triangleright k, e)$$

...and then decide how to proceed:

$$(\texttt{if}_\tau\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \triangleright k, \texttt{true}) \mapsto_{\mathsf{C}} (k, e_1)$$

$$(\texttt{if}_\tau\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \triangleright k, \texttt{false}) \mapsto_{\mathsf{C}} (k, e_2)$$

## The C Machine: Functions

First we evaluate the function:

$$(k, \texttt{apply}(e_1, e_2)) \mapsto_{\mathsf{C}} (\texttt{apply}(\Box, e_2) \triangleright k, e_1)$$

. . . and then the argument:

$$(\texttt{apply}(\Box, e_2) \triangleright k, v_1) \mapsto_{\mathsf{C}} (\texttt{apply}(v_1, \Box) \triangleright k, e_2)$$

. . . and then perform the application:

$$(\texttt{apply}(v_1, \Box) \triangleright k, v_2) \mapsto_{\mathsf{C}} (k, \{v_1, v_2 / f, x\} e)$$

where $v_1 = \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end}$.

## 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:

$$\bullet \mathbin{@} e \;=\; e$$

$$+(\Box, e_2) \triangleright k \mathbin{@} e_1 \;=\; k \mathbin{@} +(e_1, e_2)$$

$$+(v_1, \Box) \triangleright k \mathbin{@} e_2 \;=\; k \mathbin{@} +(v_1, e_2)$$

$$\texttt{if}_\tau\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \triangleright k \mathbin{@} e \;=\; k \mathbin{@} \texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}$$

$$\texttt{apply}(\Box, e_2) \triangleright k \mathbin{@} e \;=\; k \mathbin{@} \texttt{apply}(e, e_2)$$

$$\texttt{apply}(v_1, \Box) \triangleright k \mathbin{@} e \;=\; k \mathbin{@} \texttt{apply}(v_1, e)$$

## Relating the C and the M Machines

Each frame of the stack corresponds to a search rule.

- For example, $\texttt{apply}(\square, e_2)$ corresponds to searching within the function position.

- All frames have this property.

## 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.

## 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

**Theorem 1**

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

1. $k \mathbin{@} e = k' \mathbin{@} e'$, *or*

2. $k \mathbin{@} e \mapsto_{\mathsf{M}} k \mathbin{@} e'$.

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

**Corollary 2**

*If* $(\bullet, e) \mapsto^{*}_{\mathsf{C}} (\bullet, v)$, *then* $e \mapsto^{*}_{\mathsf{M}} v$.

## Relating the C and the M Machines

What about the converse?

Suppose that $e \mapsto_{\mathsf{M}} e'$ — 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

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

**Theorem 3**
*If* $e \mapsto_{\mathsf{M}} e'$ *and* $(k, e') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$*, then* $(k, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$*.*

**Corollary 4**
*If* $e \mapsto_{\mathsf{M}}^{*} v$*, then* $(\bullet, e) \mapsto_{\mathsf{C}}^{*} (\bullet, v)$*.*

## Relating the C and the M Machines

Suppose that $e = \texttt{apply(}e_1, e_2\texttt{)}$ and $e' = \texttt{apply(}e_1', e_2\texttt{)}$ where $e_1 \mapsto_{\mathsf{M}} e_1'$.

Suppose that $(k, e') \mapsto_{\mathsf{C}}^{*} (\bullet, v)$. Therefore we have

$$\begin{array}{rcl} (k, e') & \mapsto_{\mathsf{C}} & (\texttt{apply(}\Box, e_2\texttt{)} \triangleright k, e_1') \\ & \mapsto_{\mathsf{C}}^{*} & (\bullet, v) \end{array}$$

## Relating the C and the M Machines

Then we calculate

$$
\begin{array}{rcl}
(k, e) & \mapsto_{\mathsf{C}} & (\mathtt{apply}(\Box, e_2) \triangleright k, e_1) \\
& \mapsto^*_{\mathsf{C}} & (\mathtt{apply}(\Box, e_2) \triangleright k, e'_1) \\
& \mapsto^*_{\mathsf{C}} & (\bullet, v)
\end{array}
$$

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

## Managing Binding

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

- Replicates data structures, rather than shares 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

$$\texttt{apply(fun}\,f\,\texttt{(}x\texttt{:int):int is}\,e\,\texttt{end},3\texttt{)}$$

1. **Bind** $x$ to 3 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

Consider the following ML code:

```
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`?

## 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

**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

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

**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 `w` in the definition of `apply_pair`, the problem disappears.

## Dynamic vs. Static Scope

The environment model (as described so far) does not respect change of bound variable names!

This violates the fundamental principles of higher-order abstract syntax, which ensures that bound variable names may be chosen arbitrarily.

## Dynamic vs. Static Scope

**Static scope**:

- Resolution of scope is performed by a syntactic analysis (nearest enclosing binding).

- Evaluation respects $\alpha$-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 $\alpha$-conversion, and hence non-modular.

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

## The E Machine

To implement static scope we must attach environments to expressions during evaluation.

- Pairs of expressions and environments are called **closures**.
- Function values are closures (to ensure that free variables remain bound).
- Stack frames employ closures.

## The E Machine: Values

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

$$\frac{}{n \ \mathsf{mvalue}} \qquad \frac{}{\mathtt{true}\ \mathsf{mvalue}} \qquad \frac{}{\mathtt{false}\ \mathsf{mvalue}}$$

$$\frac{}{\mathtt{fun}\, f\ \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\, \mathtt{is}\, e\, \mathtt{end}[E]\ \ \mathsf{mvalue}}$$

An **environment**, $E$, is a finite function mapping variables to machine values.

## The E Machine: Stacks

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

$$\frac{}{\bullet\ \mathsf{mstack}} \qquad \frac{F\ \mathsf{mframe} \qquad K\ \mathsf{mstack}}{F \triangleright K\ \mathsf{mstack}}.$$

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:

$$\frac{}{+(\Box, e_2)[E] \ \textsf{mframe}} \qquad \frac{}{+(V_1, \Box) \ \textsf{mframe}}$$

$$\frac{}{\texttt{if}_\tau \, \Box \, \texttt{then} \, e_1 \, \texttt{else} \, e_2 \, \texttt{fi}[E] \ \textsf{mframe}}$$

$$\frac{}{\texttt{apply}(\Box, e_2)[E] \ \textsf{mframe}} \qquad \frac{}{\texttt{apply}(V_1, \Box) \ \textsf{mframe}}$$

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 is a triple $(K, E, e)$ where

- $K$ is a machine stack;
- $E$ is an environment;
- $e$ is an expression whose free variables are bound by $E$.

## The E Machine: States

We also need an **auxiliary states** $(K, V)$ where

- $K$ is a machine stack;
- $V$ is a machine value.

## The E Machine: Transitions

Types of transitions:

$(K, E, e) \mapsto_{\mathsf{E}} (K', E', e')$
*process expression*

$(K, E, v) \mapsto_{\mathsf{E}} (V', K')$
*pass value from expression to stack*

$(V, K) \mapsto_{\mathsf{E}} (V', K')$
*pass value to stack*

$(V, K) \mapsto_{\mathsf{E}} (K', E', e')$
*process pending expression*

## The E Machine: Variables

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

$$(K, E, x) \mapsto_{\mathsf{E}} (E(x), K)$$

## The E Machine: Numbers

$$(K, E, n) \mapsto_{\mathsf{E}} (n, K)$$

$$(K, E, +(e_1, e_2)) \mapsto_{\mathsf{E}} (+(\square, e_2)[E] \triangleright K, E, e_1)$$

$$(V_1, +(\square, e_2)[E] \triangleright K) \mapsto_{\mathsf{E}} (+(V_1, \square) \triangleright K, E, e_2)$$

$$(n_2, +(n_1, \square) \triangleright K) \mapsto_{\mathsf{E}} (n_1 + n_2, K)$$

# The E Machine: Booleans

$$(K, E, \texttt{true}) \mapsto_{\mathsf{E}} (\texttt{true}, K)$$

$$(K, E, \texttt{false}) \mapsto_{\mathsf{E}} (\texttt{false}, K)$$

$$\begin{array}{l}(K, E, \texttt{if}_\tau\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}) \mapsto_{\mathsf{E}} \\ (\texttt{if}_\tau\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}[E] \triangleright K, E, e)\end{array}$$

$$(\texttt{true}, \texttt{if}_\tau\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}[E] \triangleright K) \mapsto_{\mathsf{E}} (K, E, e_1)$$

$$(\texttt{false}, \texttt{if}_\tau\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}[E] \triangleright K) \mapsto_{\mathsf{E}} (K, E, e_2)$$

## The E Machine: Functions

$$(K, E, \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e\ \texttt{end}) \mapsto_{\mathsf{E}}$$
$$(\texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e\ \texttt{end}[E], K)$$

$$(K, E, \texttt{apply(}e_1, e_2\texttt{)}) \mapsto_{\mathsf{E}} (\texttt{apply(}\Box, e_2\texttt{)}[E] \triangleright K, E, e_1)$$

$$(V, \texttt{apply(}\Box, e_2\texttt{)}[E] \triangleright K) \mapsto_{\mathsf{E}} (\texttt{apply(}V, \Box\texttt{)} \triangleright K, E, e_2)$$

$$(V_2, \texttt{apply(}V, \Box\texttt{)} \triangleright K) \mapsto_{\mathsf{E}} (K, E[f \mapsto V][x \mapsto V_2], e)$$

where

$$V = \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e\ \texttt{end}[E].$$

## The E Machine: Final States

The final states of the E machine have the form $(V, \bullet)$, with final result $V$.

If $V$ is a number or boolean, it can just be printed. If it is a closure, ML just prints `fn`, rather than attempting to expand the environment and print the code.

## The E and the M Machines

**Proposition 5**
*For closed expressions* $e$ *of base type,* $e \mapsto^*_{\mathsf{M}} v$ *iff* $(\bullet, \emptyset, e) \mapsto^*_{\mathsf{E}} (\bullet, \emptyset, v)$.

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.