510: Programming Languages

Abstract Machines

David Walker

Fall, 2003

What is an abstract machine?

An abstract machine has some state:

- Initial and final state.

Instructions for transforming the state.

- Rules for determining order of execution.

Execution = instruction by instruction state transformation.

What is an abstract machine?

An abstract machine is defined by a **state 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

You have already defined one sort of abstract machine.

States: closed MinML expressions.

- Initial: any well-typed, closed expression.

- Final: closed values.

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

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

The M Machine

The M Machine is very high-level, in at least two ways:

- **Control**. Complex search rules specify order of execution. Many abstract machines have an **explicit** stack to manage the order of operations.

- **Data**. Parameter passing is by substitution, which is complex and generates “new” code on the fly.

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** is defined inductively by a set of rules.

$(k, e) \mapsto_{\mathsf{C}} (k', e')$

The C Machine: Control Stacks

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

$$\text{stacks} \quad k \quad ::= \quad \bullet \mid f \triangleright k$$

Think of pushing a frame onto the control stack.

The C Machine: Stack Frames

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

$$\text{frames} \quad f \quad ::= \quad +(\Box, e_2) \mid +(v_1, \Box) \\ \mid \mathtt{if}_\tau\, \Box\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi} \\ \mid \mathtt{apply}(\Box, e_2) \mid \mathtt{apply}(v_1, \Box)$$

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:

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

. . . and then decide how to proceed:

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

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

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

Relating the C and the M Machines

Each frame of the stack corresponds to a search rule.

- For example, $\texttt{apply}(\Box, e_2)$ 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

What is the relationship between a C Machine state and an M Machine state?

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

$$k \mathbin{@} e = e'$$

The relationship will be inductively defined on the structure of k.

Relating the C and the M Machines

The details:

$$
\begin{array}{rcl}
\bullet \,@\, e & = & e \\
+(\Box, e_2) \triangleright k \,@\, e_1 & = & k \,@\, +(e_1, e_2) \\
+(v_1, \Box) \triangleright k \,@\, e_2 & = & k \,@\, +(v_1, e_2) \\
\mathtt{if}_\tau \,\Box\, \mathtt{then}\, e_1 \,\mathtt{else}\, e_2 \,\mathtt{fi} \triangleright k \,@\, e & = & \\
 & & k \,@\, \mathtt{if}_\tau \, e \, \mathtt{then}\, e_1 \,\mathtt{else}\, e_2 \,\mathtt{fi} \\
\mathtt{apply}(\Box, e_2) \triangleright k \,@\, e & = & k \,@\, \mathtt{apply}(e, e_2) \\
\mathtt{apply}(v_1, \Box) \triangleright k \,@\, e & = & k \,@\, \mathtt{apply}(v_1, e)
\end{array}
$$

Relating the C and the M Machines

Theorem 1

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

1. $k \circledcirc e = k' \circledcirc e'$, *or*

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

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

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

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

Relating the C and the M Machines

Then we calculate

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

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

Exceptions

The C Machine makes it very easy to specify the semantics of exceptions.

Add a primitive **failure** mechanism:

$$e \ ::= \ \ldots \mid \mathtt{fail} \mid \mathtt{try}\, e_1 \,\mathtt{ow}\, e_2$$

Think of `fail` as raising a fixed, anonymous exception.

Think of $\mathtt{try}\, e_1 \,\mathtt{ow}\, e_2$ as e_1 `handle _ =>` e_2.

Static Semantics of Exceptions

No surprises here:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\, e_1\, \texttt{ow}\, e_2 : \tau}$$

Failures have **any** type.

Normal and failure return for handler must have the **same** type.

Dynamic Semantics of Exceptions

Use the C Machine since it provides access to the control stack.

- Could use the M Machine (exercise).
- Another solution below relies on explicit control stack.

Additional frame:

$$f \; ::= \; \cdots \mid \texttt{try}\,\square\,\texttt{ow}\,e_2$$

Dynamic Semantics of Exceptions

Evaluate "try" clause:

$$\frac{}{(K, \texttt{try}\, e_1\, \texttt{ow}\, e_2) \mapsto (\texttt{try}\, \Box\, \texttt{ow}\, e_2 \triangleright K, e_1)}$$

If it achieves a value, return it and drop handler:

$$\frac{}{(\texttt{try}\, \Box\, \texttt{ow}\, e_2 \triangleright K, v) \mapsto (K, v)}$$

Dynamic Semantics of Exceptions

If "try" clause fails, unwind stack to nearest enclosing handler.

$$\frac{(F \neq \texttt{try}\,\Box\,\texttt{ow}\,e_2)}{(F \triangleright K, \texttt{fail}) \mapsto (K, \texttt{fail})}$$

Then invoke pending handler.

$$\frac{}{(\texttt{try}\,\Box\,\texttt{ow}\,e_2 \triangleright K, \texttt{fail}) \mapsto (K, e_2)}$$

Safety for Exceptions

Well-formed machine states:

$$\frac{\vdash K : \tau\,\texttt{stack} \qquad \vdash e : \tau}{(K, e)\ \mathsf{ok}}$$

That is, it must be type correct to pass (the value of) e to K.

Stack Typing

A stack is a composition of frames:

$$\frac{}{\vdash \bullet : \tau\,\texttt{stack}}$$

$$\frac{\vdash F : (\tau,\tau')\ \texttt{frame} \qquad \vdash K : \tau'\,\texttt{stack}}{\vdash F \triangleright K : \tau\,\texttt{stack}}$$

Frame Typing

$$\frac{\vdash e_2 : \texttt{int}}{\vdash +(\Box, e_2) : \texttt{(int,int) frame}}$$

$$\frac{v_1 \ \mathsf{value} \qquad \vdash v_1 : \texttt{int}}{\vdash +(v_1, \Box) : \texttt{(int,int) frame}}$$

Frame Typing

$$\frac{\vdash e_2 : \tau_2}{\vdash \texttt{apply}(\Box, e_2) : (\tau_2 \rightarrow \tau, \tau)\ \texttt{frame}}$$

$$\frac{v_1\ \text{value} \qquad \vdash v_1 : \tau_2 \rightarrow \tau}{\vdash \texttt{apply}(v_1, \Box) : (\tau_2, \tau)\ \texttt{frame}}$$

Frame Typing

$$\frac{\vdash e_1 : \tau \qquad \vdash e_2 : \tau}{\vdash \texttt{if}_\tau\,\square\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} : \texttt{(bool,}\tau\texttt{) frame}}$$

$$\frac{\vdash e_2 : \tau}{\vdash \texttt{try}\,\square\,\texttt{ow}\,e_2 : \texttt{(}\tau\texttt{,}\tau'\texttt{) frame}}$$

Type Safety

Theorem 5 (Preservation)
If (K, e) ok *and* $(K, e) \mapsto (K', e')$, *then* (K', e') ok.

Theorem 6 (Progress)
If (K, e) ok *then either*

1. $K = \bullet$ *and* e value, *or*

2. $K = \bullet$ *and* $e = \texttt{fail}$, *or*

3. *there exists* K' *and* e' *such that* $(K, e) \mapsto (K', e')$.

A More Realistic Semantics

Raising an exception walks the stack to find nearest enclosing handler.

- Slow: linear in depth of stack, not depth of handler nesting.

- Penalizes code that doesn't raise exceptions by popping unused exception handlers.

- Requires run-time information to detect handler.

We expect a failure to jump **directly** to the handler and non-failing code to execute normally.

A More Realistic Semantics

Informal idea:

- Maintain a register containing **current handler**.
- On failure, **jump** directly to contents of this register.
- Restore **previous** handler before jump and before normal return.
- Restore control stack before invoking handler.

A More Realistic Semantics

This semantics requires:

- A **handler stack** to enclose pending handlers. Current handler is top of stack. Stack is modified during execution.

- A means of **freeze-drying** and **thawing** stacks to save stack appropriate to handler.

A More Realistic Semantics

Machine state: (H, K, e), where

- H is a handler stack;
- K is a control stack;
- e is a closed expression

A More Realistic Semantics

Handler stacks:

$$\text{handler stack} \quad H \quad ::= \quad \bullet \mid (K, e) \triangleright H$$

Store exception handler paired with control stack.

A More Realistic Semantics

Handlers preserve state on handler stack:

$$\overline{(H, K, \texttt{try}\, e_1\, \texttt{ow}\, e_2) \mapsto ((K, e_2) \triangleright H, \texttt{try}\, \Box\, \texttt{ow}\, e_2 \triangleright K, e_1)}$$

On failure, pop the handler stack:

$$\overline{((K', e') \triangleright H, K, \texttt{fail}) \mapsto (H, K', e')}$$

If no handler, stop execution immediately:

$$\overline{(\bullet, K, \texttt{fail}) \mapsto (\bullet, \bullet, \texttt{fail})}$$

A More Realistic Semantics

If guarded expression evaluates normally, we must remember to pop the handler stack:

$$\frac{}{((K, e_2) \triangleright H, \texttt{try}\,\Box\,\texttt{ow}\,e_2 \triangleright K, v_1) \mapsto (H, K, v_1)}$$

A More Realistic Semantics

Is freeze-drying a stack realistic?

- Sounds like a heavyweight operation.
- Can we "hot swap" stacks on a running program?

A More Realistic Semantics

Lemma 7

Let (H, K, e) be a state of the abstract machine. If $H = (K', e') \triangleright H'$, then K' is an initial segment of K. Moreover, the same property holds of (H', K, e).

Thus stored stacks may be represented as "fingers" pointing into the control stack.

- Restore by multi-pop to a given point earlier in the stack.
- *cf.* `setjmp/longjmp` in C.

Value-Passing Exceptions

It's important to be able to distinguish different sorts of failures.

- Division-by-zero, arithmetic overflow.
- Match and bind failures.
- User-defined failures.

Solution: pass values along with exceptions.

Value-Passing Exceptions

A schematic formulation:

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}(e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau_{\mathsf{exn}} \to \tau}{\Gamma \vdash \mathtt{try}\, e_1\, \mathtt{ow}\, e_2 : \tau}$$

Question: how to choose τ_{exn}?

Value-Passing Exceptions

Observation: there must be **one** choice governing **all** exceptions.

- Handler cannot tell which exception will be raised.
- Handler usually analyzes value associated with the exception.

Value-Passing Exceptions

One possiblity: $\tau_{\mathsf{exn}} = \mathtt{exc}$.

```
Datatype exc = Div | Overflow | Match | ···
```

Then we can easily distinguish exceptions using pattern matching:

```
fun div (m, 0) = raise Div
  | div (m, n) = ... raise Overflow ...
fun hdlr Div = ...
  | hdlr Overflow = ...
```

Value-Passing Exceptions

Requires that we **fix in advance** the set of exceptions.

- Non-modular. Makes writing libraries difficult.

- Non-extensible. No user-defined exceptions.

Value-Passing Exceptions

Better: a **dynamically extensible** datatype.

- Called `exn` in ML.
- Add a **new** constructor using `exception` declaration.
- Pattern match to dispatch on exception.

```
exception Div
exception Overflow
fun div (m, 0) = raise Div
  | div (m, n) = ... raise Overflow ...
```

Value-Passing Exceptions

The `exn` type can be used independently of the exception handling mechanisms!

- Useful on its own as an extensible type.
- Exception mechanism passes values of type `exn`.

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.

It's easy to define the semantics of Exceptions using abstract machines.