Abstract Machines

Robert Harper
Carnegie Mellon University
Spring, 1999

What is a Computer?

- The current state of computation.
  - initial state = starting configuration
  - final state = ending configuration
- Some instructions that change state.
  - must be effectively executable
- Execution = transition from initial to final state.

What is a Computer?

- Example: x86 processor.
  - state = memory and register contents, program counter, etc.
  - instruction = change memory/registers according to current state

What is a Computer?

- In essence a computer is a transition system!
  - Not all are realized in silicon, e.g. JVM.
  - But all share the same fundamental pattern of step-by-step execution.
- We'll explore some abstract (aka virtual) machines derived from the MinML semantics.

The M Machine

- The M machine = MinML Dynamic Semantics.
  - state = expression e
  - instruction = transition $e \mapsto e'$
- Very high level instructions:
  - "search" rules for finding next instruction
  - function call by substitution

Deriving Abstract Machines

- Two issues:
  - control flow
  - data flow
- We'll derive two abstract machines:
  - C machine, which handles control flow
  - E machine, which handles data flow

The C Machine

- The M machine employs "search" rules to find the next instruction.
  - e.g., apply(e1,e2) ↦ apply(e1', e2), if e1 ↦ e1'.
- In implementation terms, the interpreter is not tail recursive.
  - Recursive call on e1 determines how to proceed with apply(e1,e2).

The C Machine

- Non-tail recursive procedures require a stack to keep track of pending calls.
  - Corresponds to an implicit requirement for additional state to implement the abstract machine.
- Idea: make this explicit and avoid the "meta-stack".
  - ∴ interpreter will be tail recursive!

The C Machine

- The state of the C machine is a pair (k, e), where
  - k is the control stack, and
  - e is the current expression
- The control stack records the pending computations to be performed after evaluation of e.
  - Control stack = current continuation.

The C Machine

- A control stack k is a sequence of frames f:
  - empty, ●
  - non-empty, f ▷ k
- A frame represents one "level" of pending computation.
  - expression with a "hole" in it, the "current position" within it

The C Machine

- Frames are inductively defined:
  - o(v1,...,vi-1,□,ei+1,...,en)
  - if □ then e1 else e2 fi
  - apply(□, e2)
  - apply(v1, □)
- The "□" indicates the "current" sub-expression on which we are working.
  - Note where values are required.

The C Machine

- Initial state: (●,e)
  - No surrounding context.
- Final state: (●,v)
  - All pending work completed.
- Transitions: (k,e) ↦ (k',e')
  - Given by a set of rules (to follow).

The C Machine

- Primitive operations
  - (k, o(e1,...,en)) ↦ (o(□,e2,...,en) ▷ k, e1)
  - (o(v1,...,vi-1,□,ei+1,...,en) ▷ k, vi) ↦ (o(v1,...,vi,□,ei+2,...,en) ▷ k, ei+1)
  - (o(v1,...,vn-1,□) ▷ k, vn) ↦ (k, v)
  - (where v is value of o on v1,...,vn)

The C Machine

- Conditional Expression
  - (k, if e then e1 else e2 fi) ↦ (if □ then e1 else e2 fi ▷ k, e)
  - (if □ then e1 else e2 fi ▷ k, true) ↦ (k, e1)
  - (if □ then e1 else e2 fi ▷ k, false) ↦ (k, e2)

The C Machine

- Applications
  - (k, apply(e1, e2)) ↦ (apply(□, e2) ▷ k, e1)
  - (apply(□, e2) ▷ k, v1) ↦ (apply(v1, □) ▷ k, e2)
  - (apply(v1, □) ▷ k, v2) ↦ (k, [v1, v2/f,x]e) (where v1 = fun f(x:τ):τ' is e end)

Key Observation

- None of the rules of the C machine have any premises!
  - They are all pure transitions; no "if ..., then ...".
- Consequently, the implementation is naturally tail recursive.
  - Pending computations are explicit in the C machine's state.

The C Machine

- Example: let F be factorial,
  - fun f(x:int):int is if x=0 then 1 else x*f(x-1) fi end
- (●,F(3)) ↦
- (□(3) ▷●, F) ↦
- (F(□) ▷●,3) ↦
- (●,if 3=0 then 1 else 3*F(3-1) fi) ↦
- (if □ then 1 else 3*F(3-1) fi ▷●, 3=0) ↦*

The C Machine

- (if □ then 1 else 3*F(3-1) fi ▷●, 3=0) ↦*
- (if □ then 1 else 3*F(3-1) fi ▷●, false) ↦
- (●,3*F(3-1)) ↦
- (□*F(3-1) ▷●, 3) ↦
- (3* □ ▷●, F(3-1)) ↦*
- (F(□) ▷ 3*□ ▷ ●, 3-1) ↦*
- (3*□ ▷ ●, 2*f(2-1)) ↦*
- (2*□ ▷ 3*□ ▷ ●, 1*f(1-1)) ↦*

The C Machine

- (1*□ ▹ 2*□ ▹ 3*□ ▹ ●, 1) ↦*
- (2*□ ▹ 3*□ ▹ ●, 1*1) ↦*
- (3*□ ▹ ●, 2*1) ↦*
- (●, 3*2) ↦*
- (●,6)

Relating the C to the M Machine

- The C machine and the M machine are equivalent: e ↦* v in the M machine, iff (●,e) ↦* (●,v) in the C machine.
- But how can we prove this?
  - It is surprisingly tricky to get this right!
  - Essentially a programming problem in disguise.

Relating the C and M Machines

- Let's start with the simpler case: if (●,e) ↦* (●,v), then e ↦* v.
- Might try: if (●,e) ↦ (●,e'), then e ↦ e'. But ...
  - the stack grows and shrinks
  - steps don't correspond one-for-one
- Must strengthen the theorem (prove more).

Relating the C and M Machines

- Observation 1: each step of the C machine corresponds to zero or more steps of the M machine.
  - The steps that push frames onto the stack are implicit bookkeeping in the M machine.
  - The genuine computation steps are instructions in the M machine.

Relating the C and M Machines

- Observation 2: we can reconstruct the current program from the state of the C machine.
  - "Unwind" the stack, starting with the current expression.
  - The stack k records "where we are" in a complex expression.
  - Notation: k @ e.

Unwinding the Stack

- Define k@e by induction on k:
  - ● @ e = e
  - (o(v1,...,vi-1,□,ei+1,...,en)▹k) @ e = k @ o(v1,...,vi-1,e,ei+1,...,en)
  - (if □ then e1 else e2 fi ▹ k) @ e = k @ if e then e1 else e2 fi
  - (apply(□, e2) ▹ k) @ e = k @ apply(e, e2)
  - (apply(v1, □) ▹ k) @ e = k @ apply(v1, e)

Relating the C and M Machines

- Thm: If (k,e) ↦ (k', e'), then either k@e = k'@e', or k@e ↦ k'@e'.
  - Proof is by induction on C machine transition relation
  - NB: the next M machine instruction to execute in k@e must lie within e!
  - It follows that if (k,e) ↦ (k', e'), then k@e ↦* k'@e' (the * is 0 or 1 here).

Relating the C and M Machines

- Corollary: If (●,e) ↦* (●,v), then e ↦* v.
  - We have (●,e) ↦ (k',e') ↦ ... ↦ (●,v).
  - By the theorem, e = ●@e ↦* k'@e' ↦* ... ↦* ●@v = v.
  - Notice that each starred step is reall either zero or one steps!

Relating the C and M Machines

- (k,apply(e1, e2)) ↦ (apply(□,e2) ▷ k, e1)
  - No work for the M machine in this case.
  - k @ apply(e1, e2) = (apply(□, e2) ▷ k) @ e1
- (apply(v1,□) ▷ k, v2) ↦ (k, [v1,v2/f,x]e)
  - A genuine instruction!
  - (apply(v1,□) ▷ k) @ v2 = k @ apply(v1, v2) ↦ k @ [v1,v2/f,x]e
  - (Check: why is the last step true?)

Relating the C and M Machines

- The converse is much harder!
  - The difficulty is in stating the theorem.
  - The proof is easy after that ....
- Key problem: one M transition may correspond to many C transitions.
  - Cannot relate transitions step-by-step.
  - Hint: recall the relation between evaluation semantics and step semantics.

Relating the C and M Machines

- If e ↦ e' and (k, e') ↦* (●,v), then (k, e) ↦* (●, v).
- That is, if the M machine takes a step from e to e', ...
- and the C machine runs from (k,e') to completion, ...
- then the C machine runs from (k,e) to completion.

Relating the C and M Machines

- Hence e ↦* v implies (●,e) ↦* (●,v), by induction on the # of M steps.
- Suppose e1 ↦ e2 ↦ ... ↦ en = v.
- Clearly (●, en) ↦* (●,v).
- So (●,en-1) ↦* (●,v), and so (●, en-2) ↦* (●,v), ..., and so (●,e1) ↦* (●,v).

Relating the C and M Machines

- See the course notes for a detailed proof of correspondence.
- Proof is by induction on the M machine rules.
  - Consider each M rule, piece together a C transition from it.
- Observe that many C steps are needed for some M steps.

Data Flow

- The C machine makes control flow explicit.
  - Makes control state explicit using a stack.
- But "data flow" remains implicit.
  - Substitution [v1,v2/f,x]e is a "heavyweight" operation.
  - Corresponds to self-modifying code!

The E Machine

- Idea: avoid substitution by recording the bindings of variables in an environment.
  - Retrieve the binding of a variable whenever it is used.
  - Environment is stack-like, but is not really a stack!

The E Machine

- State: (K, E, e)
  - K is a control stack
  - E is an environment : Variables -> Values
  - e is an expression st FV(e) $\subseteq$ Dom(E)
- The bindings of variables are in E.
  - Look up binding on use of a variable.
  - Bindings are introduced by function calls.

The E Machine

- Function calling sequence (roughly):
  - To call the function v = fun f(x) is e end with argument v',
  - bind f to v and x to v' in E
  - evaluate e to value v''
  - return v''

The E Machine

- But watch out!
  - e.g., fun f(x) is (fun g(y) is x+y end) end
  - if we call f on 3, we bind x to 3, then evaluate body to get fun g(y) is x+y end
  - if we re-bind x for some other reason, the x in g gets "captured"!

Dynamic Binding

- Lisp 1.5 made the mistake of dropping the binding for x, yielding a function with a "dangling" x.
- What happens if we apply it? We get whatever binding for x happens to be around at the time, if any.
- This is called dynamic binding.

Dynamic Binding

- Here's what happens in original Lisp:
  - (define f
    (lambda (x) (lambda (y) (+ x y))))
  - (define g (f 3))
  - (define x 5) ; bind x
  - (g 3) yields 8, not 6!
  - Without the top-level binding for x, (g 3) yields 6!

Dynamic Binding

- Dynamic binding seems like a bad idea!
  - Violates modularity: you cannot predict the behavior of (f 3) by looking at its code.
  - e.g., if you pass (f 3) to library code, it will pick up whatever binding for x happens to be laying around
- But: dynamic binding is the basis for inheritance in oop ....

Static Binding

- How can we avoid dynamic binding?
  - When yielding a function as result, package it with the current bindings for its free variables. This is called a closure.
  - When applying a function, reconstitute the bindings in the closure, throwing away any "current" bindings.

The E Machine

- Closures complicate things since they not just expressions.
- Must distinguish machine values from expression values.
  - integers and booleans as before,
  - function values are replaced by closures

The E Machine

- Environments map variables to machine values.
  - e.g., a variable may be bound to a closure
- Control stacks must involve closures to record bindings of variables in pending computations.
  - otherwise similar to C machine stacks

The E Machine

- Machine values V
  - constants: n, true, false
  - closures: (fun f(x) is e)[E]
- Machine Stack Frames
  - o(V1,...,Vi-1,□,ei+1,...,en) [E]
  - if □ then e1 else e2 fi [E]
  - apply(□, e2) [E]
  - apply(V1, □)

The E Machine

- Two kinds of states
  - (K, E, e) evaluate e relative to K and E
  - (K, V) return a value V to K
- Transitions have four forms
  - (K, E, e) ↦ (K', E', e')
  - (K, E, e) ↦ (K', V')
  - (K, V) ↦ (K', V')
  - (K, V) ↦ (K', E', e')

The E Machine

- Variables
  - (K, E, x) ↦ (K, E(x))
- Constants
  - (K, E, n) ↦ (K, n)
  - (K, E, true) ↦ (K, true)
  - (K, E, false) ↦ (K, false)

The E Machine

- Functions (form a closure)
  - (K, E, fun f(x) is e end) ↦
    (K, fun f(x) is e end [E])
- Primitives
  - (K, E, o(e1,...,en)) ↦
    (o(□,e2,...,en)[E] ▷ K, E, e1)
  - (o(V1,...,Vi-1,□,ei+1,...,en)[E] ▷ K, Vi) ↦
    (o(V1,...,Vi,□,ei+2,...,en)[E] ▷ K, E, ei+1)

The E Machine

- Primitives, cont'd
  - (o(V1,...,Vn-1,□)[E] ▷ K, Vn) ↦ (K, V)
    (where V is value of o on V1,...,Vn)
- Conditionals
  - (K, E, if e then e1 else e2 fi) ↦
    (if □ then e1 else e2 fi [E] ▷ K, E, e)
  - (if □ then e1 else e2 fi [E] ▷ K, true) ↦ (K, E, e1)

The E Machine

- Conditionals, cont'd
  - (if □ then e1 else e2 fi [E] ▷ K, false) ↦ (K, E, e2)
- Applications
  - (K, E, apply(e1, e2)) ↦
    (apply(□, e2)[E] ▷ K, E, e1)
  - (apply(□, e2)[E] ▷ K, V1) ↦
    (apply(V1, □) ▷ K, E, e2)

The E Machine

- Applications, cont'd
  - (apply(V1, □) ▷ K, V2) ↦
    (K, E[f=V1][x=V2], e)
    (where V1 = fun f(x:τ):τ' is e end[E])

Important Points

- The use of stacks to model control flow.
  - Corresponds to a run-time stack.
- The use of environments to model data flow.
  - Much closer to real implementations.

Important Points

- The distinction between static and dynamic scope for variables.
  - More on this later ....
- The use of closures to implement static scope.
  - Package functions with their environments.
  - Reconstitute environment when called.