Abstract Machines

Johannes Åman Pohjola
UNSW
Term 3 2022
We all know that `MergeSort` has $O(n \log n)$ time complexity, and that `BubbleSort` has $O(n^2)$ time complexity, but what does that actually mean?

**Big O Notation**

Given functions $f, g : \mathbb{R} \rightarrow \mathbb{R}$, $f \in O(g)$ if and only if there exists a value $x_0 \in \mathbb{R}$ and a coefficient $m$ such that:

$$\forall x > x_0. \quad f(x) \leq m \cdot g(x)$$

What is the codomain of $f$?

When analysing algorithms, we don’t usually time how long they take to run on a real machine.
Big O

Q: How would you derive the complexity of this mergesort?

\[
\text{mergesort}([ ]) = [] \\
\text{mergesort}(xs) = f(n) = \\
\text{let } (ys, zs) = \text{partition } xs; c_2 \cdot n + \\
ys' = \text{mergesort } ys; f(n/2) + \\
zs' = \text{mergesort } zs f(n/2) + \\
\text{in merge } ys' zs' c_3 \cdot n
\]

A: Define a cost function \( f \), then find its closed form.

Q: Is there a formal connection between mergesort and \( f \), or did we just pull \( f \) out of thin air?

A: Well, um.
A *cost model* is a mathematical model that measures the cost of executing a program. There are *denotational* cost models, that assign a cost directly to syntax:

\[ [\cdot] : \text{Program} \rightarrow \text{Cost} \]

In this course, we will focus on *operational cost models*.

**Operational Cost Models**

First, we define a program-evaluating *abstract machine*. We determine the time cost by counting the number of steps it takes.
Abstract Machines

An abstract machine consists of:

1. A set of states $\Sigma$,
2. A set of initial states $I \subseteq \Sigma$,
3. A set of final states $F \subseteq \Sigma$, and
4. A transition relation $\rightarrow \subseteq \Sigma \times \Sigma$.

We’ve seen this before in structured operational (or small-step) semantics.
The M Machine

Is just our usual small-step rules:

\[
\begin{align*}
    e_1 & \mapsto_M e_1' \\
    (\text{Plus } e_1 e_2) & \mapsto_M (\text{Plus } e_1' e_2) \\
    e_1 & \mapsto_M e_1' \\
    (\text{If } e_1 e_2 e_3) & \mapsto_M (\text{If } e_1' e_2 e_3) \\
    (\text{If } (\text{Lit True}) e_2 e_3) & \mapsto_M e_2 \\
    (\text{If } (\text{Lit False}) e_2 e_3) & \mapsto_M e_3 \\
    e_1 & \mapsto_M e_1' \\
    (\text{Apply } e_1 e_2) & \mapsto_M (\text{Apply } e_1' e_2) \\
    e_2 & \mapsto_M e_2' \\
    (\text{Apply } (\text{Recfun } (f.x. e)) e_2) & \mapsto_M (\text{Apply } (\text{Recfun } (f.x. e)) e_2') \\
    v & \in F \\
    (\text{Apply } (\text{Recfun } (f.x. e)) v) & \mapsto_M e[x := v, f := (\text{Recfun } (f.x. e))] 
\end{align*}
\]

The M Machine is unsuitable as a basis for a cost model. Why?
Performance

One step in our machine should always only be $O(1)$ in our language implementation. Otherwise, counting steps will not get an accurate description of the time cost.

This makes for two potential problems:

1. **Substitution** occurs in function application, which is potentially $O(n)$ time.
2. **Control Flow** is not explicit – which subexpression to reduce is found by recursively descending the abstract syntax tree each time.

\[
\text{eval}(\text{Num } n) = n
\]

\[
\text{eval } e = \text{eval} (\text{oneStep } e)
\]

\[
\text{oneStep} (\text{Plus} (\text{Num } n) (\text{Num } m)) = \text{Num} (n + m)
\]

\[
\text{oneStep} (\text{Plus} (\text{Num } n) e_2) = \text{Plus} (\text{Num } n) (\text{oneStep } e_2)
\]

\[
\text{oneStep} (\text{Plus } e_1 e_2) = \text{Plus} (\text{oneStep } e_1) e_2
\]

...
The C Machine

We want to define a machine where all the rules are axioms, so there can be no recursive descent into subexpressions. How is recursion typically implemented?

Stacks!

<table>
<thead>
<tr>
<th>Stack</th>
<th>$f$ Frame $s$ Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\circ Stack$</td>
<td>$f \triangleright s$ Stack</td>
</tr>
</tbody>
</table>

**Key Idea**: States will consist of a current expression to evaluate and a stack of computational contexts that situate it in the overall computation. An example stack would be:

$$(\text{Plus } 3 \square) \triangleright (\text{Times } \square (\text{Num } 2)) \triangleright \circ$$

This represents the computational context:

$$(\text{Times } (\text{Plus } 3 \square )(\text{Num } 2))$$
The C Machine

Our states will consist of two modes:

1. **Evaluate** the current expression within stack $s$, written $s \succ e$.

2. **Return** a value $v$ (either a function, integer, or boolean) back into the context in $s$, written $s \prec v$.

*Initial states* start evaluation with an empty stack, i.e. $\circ \succ e$. *Final states* return a value to the empty stack, i.e. $\circ \prec v$.

*Stack frames* are expressions with holes or values in them:

\[
\begin{array}{c}
\text{e}_2 \text{ Expr} \\
(\text{Plus} \square \text{e}_2) \text{ Frame}
\end{array}
\quad
\begin{array}{c}
\text{v}_1 \text{ Value} \\
(\text{Plus} \text{v}_1 \square) \text{ Frame}
\end{array}
\]

\ldots
Evaluating

There are three axioms about \texttt{Plus} now:

When evaluating a \texttt{Plus} expression, first evaluate the LHS:

\[
\begin{align*}
  s \succ (\texttt{Plus } e_1 \; e_2) & \implies_C (\texttt{Plus } \square \; e_2) \triangleright s \succ e_1 \\
  (\texttt{Plus } \square \; e_2) \triangleright s \prec v_1 & \implies_C (\texttt{Plus } v_1 \; \square) \triangleright s \succ e_2 \\
  (\texttt{Plus } v_1 \; \square) \triangleright s \prec v_2 & \implies_C s \prec v_1 + v_2
\end{align*}
\]

Once the LHS is evaluated, switch to the RHS:

Once the RHS is evaluated, return the sum:

We also have a single rule about \texttt{Num} that just returns the value:

\[
  s \succ (\texttt{Num } n) \implies_C s \prec n
\]
Example

\[\circ \succ (\text{Plus}(\text{Plus}(\text{Num} \ 2)(\text{Num} \ 3))(\text{Num} \ 4))\]

\[\mapsto_C (\text{Plus} \ □(\text{Num} \ 4)) \triangleright \circ \succ (\text{Plus}(\text{Num} \ 2)(\text{Num} \ 3))\]

\[\mapsto_C (\text{Plus} \ □(\text{Num} \ 3)) \triangleright (\text{Plus} \ □(\text{Num} \ 4)) \triangleright \circ \succ (\text{Num} \ 2)\]

\[\mapsto_C (\text{Plus} \ □(\text{Num} \ 3)) \triangleright (\text{Plus} \ □(\text{Num} \ 4)) \triangleright \circ \prec 2\]

\[\mapsto_C (\text{Plus} \ 2 □) \triangleright (\text{Plus} \ □(\text{Num} \ 4)) \triangleright \circ \succ (\text{Num} \ 3)\]

\[\mapsto_C (\text{Plus} \ 2 □) \triangleright (\text{Plus} \ □(\text{Num} \ 4)) \triangleright \circ \prec 3\]

\[\mapsto_C (\text{Plus} \ □(\text{Num} \ 4)) \triangleright \circ \prec 5\]

\[\mapsto_C (\text{Plus} \ 5 □) \triangleright \circ \succ (\text{Num} \ 4)\]

\[\mapsto_C (\text{Plus} \ 5 □) \triangleright \circ \prec 4\]

\[\mapsto_C \circ \prec 9\]
Other Rules

We have similar rules for the other operators and for booleans. For If:

\[
s \succ (\text{If } e_1 \ e_2 \ e_3) \iff_C (\text{If } \square \ e_2 \ e_3) \succ s \succ e_1
\]

\[
(\text{If } \square \ e_2 \ e_3) \succ s \prec \text{True} \iff_C s \succ e_2
\]

\[
(\text{If } \square \ e_2 \ e_3) \succ s \prec \text{False} \iff_C s \succ e_3
\]
Functions

Recfun (here abbreviated to Fun) evaluates to a function value:

\[ s \succ (\text{Fun} (f \cdot x \cdot e)) \leftrightarrow_c s \prec \langle\langle f \cdot x \cdot e \rangle\rangle \]

Function application is then handled similarly to Plus.

\[ s \succ (\text{Apply} \ e_1 \ e_2) \leftrightarrow_c \ (\text{Apply} \ \square \ e_2) \triangleright s \succ e_1 \]

\[ (\text{Apply} \ \square \ e_2) \triangleright s \prec \langle\langle f \cdot x \cdot e \rangle\rangle \leftrightarrow_c \ (\text{Apply} \ \langle\langle f \cdot x \cdot e \rangle\rangle \ \square) \triangleright s \succ e_2 \]

\[ (\text{Apply} \ \langle\langle f \cdot x \cdot e \rangle\rangle \ \square) \triangleright s \prec v \leftrightarrow_c s \succ e[x := v, f := (\text{Fun} (f \cdot x \cdot e))] \]

We are still using substitution for now.
What have we done?

- All the rules are axioms – we can now implement the evaluator with a simple `while` loop (or a `tail recursive` function).
- We have a lower-level specification – helps with code generation (e.g. in an assembly language)
- Substitution is still a machine operation – we need to find a way to eliminate that.
Correctness

While the M-Machine is reasonably straightforward definition of the language's semantics, the C-Machine is much more detailed. We wish to prove a theorem that tells us that the C-Machine behaves analogously to the M-Machine.

Refinement

A low-level (concrete) semantics of a program is a refinement of a high-level (abstract) semantics if every possible execution in the low-level semantics has a corresponding execution in the high-level semantics. In our case:

\[ \forall e, v. \quad \bigcirc \triangleright e \quad \xrightarrow{*} C \quad \bigcirc \triangleleft v \quad \xrightarrow{*} e \quad \xrightarrow{*} M \quad v \]

Functional correctness properties are preserved by refinement, but security properties are not.
How to Prove Refinement

We can’t get away with simply proving that each C machine step has a corresponding step in the M-Machine, because the C-Machine makes multiple steps that are no-ops in the M-Machine:

\[ \circ \succ (+ (+ (N \ 2) (N \ 3)) (N \ 4)) \succ (+ (+ (N \ 2) (N \ 3)) (N \ 4)) \]

\[ \mapsto_C (+ \square (N \ 4)) \bowtie \circ \succ (+ (N \ 2) (N \ 3)) \]

\[ \mapsto_C (+ \square (N \ 3)) \bowtie (+ \square (N \ 4)) \bowtie \circ \succ (N \ 2) \]

\[ \mapsto_C (+ \square (N \ 3)) \bowtie (+ \square (N \ 4)) \bowtie \circ \prec 2 \]

\[ \mapsto_C (+ 2 \square) \bowtie (+ \square (N \ 4)) \bowtie \circ \succ (N \ 3) \]

\[ \mapsto_C (+ 2 \square) \bowtie (+ \square (N \ 4)) \bowtie \circ \prec 3 \]

\[ \mapsto_C (+ \square (N \ 4)) \bowtie \circ \prec 5 \]

\[ \mapsto_C (+ 5 \square) \bowtie \circ \succ (N \ 4) \]

\[ \mapsto_C (+ 5 \square) \bowtie \circ \prec 4 \]

\[ \mapsto_C \circ \prec 9 \]

\[ \mapsto_M (+ (N \ 5) (N \ 4)) \]

\[ \mapsto_C \prec M (N \ 9) \]
How to Prove Refinement

1. Define an *abstraction function* $A : \Sigma_C \rightarrow \Sigma_M$ that relates C-Machine states to M-Machine states, describing how they “correspond”.

2. Prove, for all initial states $\sigma \in I_C$, that the corresponding state $A(\sigma) \in I_M$.

3. Prove for each step in the C-Machine $\sigma_1 \leftrightarrow_C \sigma_2$, either:
   - the step is a no-op in the M-Machine and $A(\sigma_1) = A(\sigma_2)$, or
   - the step is replicated by the M-Machine $A(\sigma_1) \leftrightarrow_M A(\sigma_2)$.

4. Prove, for all final states $\sigma \in F_C$, that $A(\sigma) \in F_M$.

In general this abstraction function is called a *simulation relation* and this type of proof is called a *simulation* proof.
The Abstraction Function

Our abstraction function $A$ will need to relate states such that each transition that corresponds to a no-op in the M-Machine will move between $A$-equivalent states:

$\circ \succ (+ (+ (N \ 2) (N \ 3)) (N \ 4))$  
$\implies C (+ \Box (N \ 4)) \triangleright \circ \succ (+ (N \ 2) (N \ 3))$

$\implies C (+ \Box (N \ 3)) \triangleright (+ \Box (N \ 4)) \triangleright \circ \succ (N \ 2)$

$\implies C (+ \Box (N \ 3)) \triangleright (+ \Box (N \ 4)) \triangleright \circ \prec 2$

$\implies C (+ \Box (N \ 4)) \triangleright (+ \Box (N \ 4)) \triangleright \circ \succ (N \ 3)$

$\implies C (+ \Box (N \ 4)) \triangleright \circ \prec 3$

$\implies C (+ \Box (N \ 4)) \triangleright \circ \prec 5$

$\implies C (+ \Box (N \ 4)) \triangleright \circ \succ (N \ 4)$

$\implies C (+ \Box (N \ 4)) \triangleright \circ \prec 4$

$\implies C \circ \prec 9$
Abstraction Function

Given a C-Machine state with a stack and a current expression (or value), we reconstruct the overall expression to get the corresponding M-Machine state.

- $A(\circ \succ e) = e$
- $A(\circ \prec v) = (\text{Num } v)$
- $A((\text{Plus } e_2) \triangleright s \succ e_1) = A(s \succ (\text{Plus } e_1 e_2))$
- etc.

By definition, all the initial/final states of the C-Machine are mapped to initial/final states of the M-Machine. So all that is left is the requirement for each transition.
Showing Refinement for Plus

\[ s \succ (\text{Plus } e_1 e_2) \quad \mapsto_C \quad (\text{Plus } \square e_2) \triangleright s \succ e_1 \]

This is a no-op in the M-Machine:

\[
\mathcal{A}(\text{RHS}) = \mathcal{A}((\text{Plus } \square e_2) \triangleright s \succ e_1) \\
= \mathcal{A}(s \succ (\text{Plus } e_1 e_2)) \\
= \mathcal{A}(\text{LHS})
\]
Showing Refinement for Plus

\[(\text{Plus} \ □ \ e_2) \triangleright s \prec v_1 \quad \overset{C}{\implies} \quad (\text{Plus} \ v_1 \ □) \triangleright s \succ e_2\]

Another no-op in the M-Machine:

\[\mathcal{A}(LHS) = \mathcal{A}((\text{Plus} \ □ \ e_2) \triangleright s \prec v_1)\]
\[= \mathcal{A}(s \succ (\text{Plus} (\text{Num} \ v_1) \ e_2))\]
\[= \mathcal{A}((\text{Plus} \ v_1 \ □) \triangleright s \succ e_2)\]
\[= \mathcal{A}(RHS)\]
Showing Refinement for Plus

\[(\text{Plus } v_1 \square) \triangleright s \prec v_2 \quad \mapsto_C \quad s \prec v_1 + v_2\]

This corresponds to a M-Machine transition:

\[
\begin{align*}
\mathcal{A}(&LHS) = \mathcal{A}( (\text{Plus } v_1 \square) \triangleright s \prec v_2) \\
= & \mathcal{A}( s \succ (\text{Plus } (\text{Num } v_1)(\text{Num } v_2))) \\
\mapsto_M & \mathcal{A}( s \succ (\text{Num } (v_1 + v_2))) \\
= & \mathcal{A}( s \prec v_1 + v_2) \\
= & \mathcal{A}(RHS)
\end{align*}
\]

Technically the reduction step \((\ast)\) requires induction on the stack.