Abstract Machines

Johannes Åman Pohjola
UNSW

Term 3 2022

Big O

We all know that MergeSort has $\mathcal{O}(n \log n)$ time complexity, and that BubbleSort has $\mathcal{O}(n^2)$ time complexity, but what does that actually mean?

Big O Notation

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

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

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`?

$$
\begin{array}{ll}
\texttt{mergesort}([]) = [] & f(0) = c_1 \\
\texttt{mergesort}(xs) = & f(n) = \\
\quad \text{let } (ys, zs) = \texttt{partition}\ xs; & \quad c_2 * n\ + \\
\qquad ys' = \texttt{mergesort}\ ys; & \quad f(n/2)\ + \\
\qquad zs' = \texttt{mergesort}\ zs & \quad f(n/2)\ + \\
\quad \text{in } \texttt{merge}\ ys'\ zs' & \quad c_3 * n
\end{array}
$$

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.

Cost Models

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:

$$[\![\cdot]\!] : \text{Program} \to \text{Cost}$$

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

Abstract Machines

An *abstract machine* consists of:

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

We've seen this before in structured operational (or small-step) semantics.

The M Machine

Is just our usual small-step rules:

$$\frac{e_1 \mapsto_M e_1'}{(\texttt{Plus}\ e_1\ e_2) \mapsto_M (\texttt{Plus}\ e_1'\ e_2)} \quad \cdots$$

$$\frac{e_1 \mapsto_M e_1'}{(\texttt{If}\ e_1\ e_2\ e_3) \mapsto_M (\texttt{If}\ e_1'\ e_2\ e_3)}$$

$$\frac{}{(\texttt{If}\ (\texttt{Lit True})\ e_2\ e_3) \mapsto_M e_2} \quad \frac{}{(\texttt{If}\ (\texttt{Lit False})\ e_2\ e_3) \mapsto_M e_3}$$

$$\frac{e_1 \mapsto_M e_1'}{(\texttt{Apply}\ e_1\ e_2) \mapsto_M (\texttt{Apply}\ e_1'\ e_2)}$$

$$\frac{e_2 \mapsto_M e_2'}{(\texttt{Apply}\ (\texttt{Recfun}\ (f.x.\ e))\ e_2) \mapsto_M (\texttt{Apply}\ (\texttt{Recfun}\ (f.x.\ e))\ e_2')}$$

$$\frac{v \in F}{(\texttt{Apply}\ (\texttt{Recfun}\ (f.x.\ e))\ v) \mapsto_M e[x := v, f := (\texttt{Recfun}\ (f.x.\ e))]}$$

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

Performance

One step in our machine should always only be $\mathcal{O}(1)$ 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 $\mathcal{O}(n)$ time.
2. **Control Flow** is not explicit – which subexpression to reduce is found by recursively descending the abstract syntax tree each time.

$$
\begin{array}{lcl}
\mathit{eval}\ (\texttt{Num}\ n) & = & n \\
\mathit{eval}\ e & = & \mathit{eval}\ (\mathit{oneStep}\ e)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{oneStep}\ (\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) & = & \texttt{Num}\ (n + m) \\
\mathit{oneStep}\ (\texttt{Plus}\ (\texttt{Num}\ n)\ e_2) & = & \texttt{Plus}\ (\texttt{Num}\ n)\ (\mathit{oneStep}\ e_2) \\
\mathit{oneStep}\ (\texttt{Plus}\ e_1\ e_2) & = & \texttt{Plus}\ (\mathit{oneStep}\ e_1)\ e_2
\end{array}
$$

$\ldots$

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!

$$\frac{}{\circ\ \mathsf{Stack}} \qquad \frac{f\ \mathsf{Frame} \quad s\ \mathsf{Stack}}{f \triangleright s\ \mathsf{Stack}}$$

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:

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

This represents the computational context:

$$(\texttt{Times}\ (\texttt{Plus}\ 3\ \square)\ (\texttt{Num}\ 2))$$

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:

$$\frac{e_2 \ \mathsf{Expr}}{(\texttt{Plus}\ \Box\ e_2)\ \mathsf{Frame}} \qquad \frac{v_1 \ \mathsf{Value}}{(\texttt{Plus}\ v_1\ \Box)\ \mathsf{Frame}}$$

$$\dots$$

Evaluating

There are three axioms about `Plus` now:

When evaluating a `Plus` expression, first evaluate the LHS:

$$\frac{}{s \succ (\texttt{Plus}\ e_1\ e_2) \quad \mapsto_C \quad (\texttt{Plus}\ \square\ e_2) \triangleright s \succ e_1}$$

Once the LHS is evaluated, switch to the RHS:

$$\frac{}{(\texttt{Plus}\ \square\ e_2) \triangleright s \prec v_1 \quad \mapsto_C \quad (\texttt{Plus}\ v_1\ \square) \triangleright s \succ e_2}$$

Once the RHS is evaluated, return the sum:

$$\frac{}{(\texttt{Plus}\ v_1\ \square) \triangleright s \prec v_2 \quad \mapsto_C \quad s \prec v_1 + v_2}$$

We also have a single rule about `Num` that just returns the value:

$$\frac{}{s \succ (\texttt{Num}\ n) \mapsto_C s \prec n}$$

Example

$$
\begin{array}{ll}
 & \circ \succ (\texttt{Plus}\ (\texttt{Plus}\ (\texttt{Num}\ 2)\ (\texttt{Num}\ 3))\ (\texttt{Num}\ 4)) \\
\mapsto_C & (\texttt{Plus}\ \square\ (\texttt{Num}\ 4)) \triangleright \circ \succ (\texttt{Plus}\ (\texttt{Num}\ 2)\ (\texttt{Num}\ 3)) \\
\mapsto_C & (\texttt{Plus}\ \square\ (\texttt{Num}\ 3)) \triangleright (\texttt{Plus}\ \square\ (\texttt{Num}\ 4)) \triangleright \circ \succ (\texttt{Num}\ 2) \\
\mapsto_C & (\texttt{Plus}\ \square\ (\texttt{Num}\ 3)) \triangleright (\texttt{Plus}\ \square\ (\texttt{Num}\ 4)) \triangleright \circ \prec 2 \\
\mapsto_C & (\texttt{Plus}\ 2\ \square) \triangleright (\texttt{Plus}\ \square\ (\texttt{Num}\ 4)) \triangleright \circ \succ (\texttt{Num}\ 3) \\
\mapsto_C & (\texttt{Plus}\ 2\ \square) \triangleright (\texttt{Plus}\ \square\ (\texttt{Num}\ 4)) \triangleright \circ \prec 3 \\
\mapsto_C & (\texttt{Plus}\ \square\ (\texttt{Num}\ 4)) \triangleright \circ \prec 5 \\
\mapsto_C & (\texttt{Plus}\ 5\ \square) \triangleright \circ \succ (\texttt{Num}\ 4) \\
\mapsto_C & (\texttt{Plus}\ 5\ \square) \triangleright \circ \prec 4 \\
\mapsto_C & \circ \prec 9
\end{array}
$$

Other Rules

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

$$\frac{}{s \succ (\texttt{If}\ e_1\ e_2\ e_3) \quad \mapsto_C \quad (\texttt{If}\ \square\ e_2\ e_3) \triangleright s \succ e_1}$$

$$\frac{}{(\texttt{If}\ \square\ e_2\ e_3) \triangleright s \prec \texttt{True} \quad \mapsto_C \quad s \succ e_2}$$

$$\frac{}{(\texttt{If}\ \square\ e_2\ e_3) \triangleright s \prec \texttt{False} \quad \mapsto_C \quad s \succ e_3}$$

Functions

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

$$\frac{}{s \succ (\texttt{Fun}\ (f.x.\ e)) \ \mapsto_C \ s \prec \langle\langle f.x.\ e \rangle\rangle}$$

Function application is then handled similarly to Plus.

$$\frac{}{s \succ (\texttt{Apply}\ e_1\ e_2) \quad \mapsto_C \quad (\texttt{Apply}\ \square\ e_2) \triangleright s \succ e_1}$$

$$\frac{}{(\texttt{Apply}\ \square\ e_2) \triangleright s \prec \langle\langle f.x.\ e \rangle\rangle \quad \mapsto_C \quad (\texttt{Apply}\ \langle\langle f.x.\ e \rangle\rangle\ \square) \triangleright s \succ e_2}$$

$$\frac{}{(\texttt{Apply}\ \langle\langle f.x.\ e \rangle\rangle\ \square) \triangleright s \prec v \quad \mapsto_C \quad s \prec e[x := v, f := (\texttt{Fun}\ (f.x.e))]}$$

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:

$$\forall e, v.\ \frac{\circ \succ e \quad \overset{\star}{\mapsto}_C \quad \circ \prec v}{e \quad \overset{\star}{\mapsto}_M \quad v}$$

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:

$$
\begin{array}{llll}
 & \circ \succ (+\ (+\ (\mathtt{N}\ 2)\ (\mathtt{N}\ 3))\ (\mathtt{N}\ 4)) & & (+\ (+\ (\mathtt{N}\ 2)\ (\mathtt{N}\ 3))\ (\mathtt{N}\ 4)) \\
\mapsto_C & (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \succ (+\ (\mathtt{N}\ 2)\ (\mathtt{N}\ 3)) & & \\
\mapsto_C & (+\ \square\ (\mathtt{N}\ 3)) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \succ (\mathtt{N}\ 2) & & \\
\mapsto_C & (+\ \square\ (\mathtt{N}\ 3)) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \prec 2 & & \\
\mapsto_C & (+\ 2\ \square) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \succ (\mathtt{N}\ 3) & & \\
\mapsto_C & (+\ 2\ \square) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \prec 3 & & \\
\mapsto_C & (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \prec 5 & \mapsto_M & (+\ (\mathtt{N}\ 5)\ (\mathtt{N}\ 4)) \\
\mapsto_C & (+\ 5\ \square) \triangleright \circ \succ (\mathtt{N}\ 4) & & \\
\mapsto_C & (+\ 5\ \square) \triangleright \circ \prec 4 & & \\
\mapsto_C & \circ \prec 9 & \mapsto_M & (\mathtt{N}\ 9)
\end{array}
$$

How to Prove Refinement

1. Define an *abstraction function* $\mathcal{A} : \Sigma_C \rightarrow \Sigma_M$ 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 $\mathcal{A}(\sigma) \in I_M$.
3. Prove for each step in the C-Machine $\sigma_1 \mapsto_C \sigma_2$, either:
   - the step is a no-op in the M-Machine and $\mathcal{A}(\sigma_1) = \mathcal{A}(\sigma_2)$, or
   - the step is replicated by the M-Machine $\mathcal{A}(\sigma_1) \mapsto_M \mathcal{A}(\sigma_2)$.
4. Prove, for all final states $\sigma \in F_C$, that $\mathcal{A}(\sigma) \in F_M$.

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 $\mathcal{A}$ will need to relate states such that each transition that corresponds to a no-op in the M-Machine will move between $\mathcal{A}$-equivalent states:

$$
\begin{array}{lll}
 & \circ \succ (+\ (+\ (\mathtt{N}\ 2)\ (\mathtt{N}\ 3))\ (\mathtt{N}\ 4)) & (+\ (+\ (\mathtt{N}\ 2)\ (\mathtt{N}\ 3))\ (\mathtt{N}\ 4)) \\
\mapsto_C & (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \succ (+\ (\mathtt{N}\ 2)\ (\mathtt{N}\ 3)) & \\
\mapsto_C & (+\ \square\ (\mathtt{N}\ 3)) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \succ (\mathtt{N}\ 2) & \\
\mapsto_C & (+\ \square\ (\mathtt{N}\ 3)) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \prec 2 & \\
\mapsto_C & (+\ 2\ \square) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \succ (\mathtt{N}\ 3) & \\
\mapsto_C & (+\ 2\ \square) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \prec 3 & \\
\mapsto_C & (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \prec 5 & \mapsto_M (+\ (\mathtt{N}\ 5)\ (\mathtt{N}\ 4)) \\
\mapsto_C & (+\ 5\ \square) \triangleright \circ \succ (\mathtt{N}\ 4) & \\
\mapsto_C & (+\ 5\ \square) \triangleright \circ \prec 4 & \\
\mapsto_C & \circ \prec 9 & \mapsto_M (\mathtt{N}\ 9)
\end{array}
$$

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.

$$
\begin{array}{lcl}
\mathcal{A}(\circ \succ e) & = & e \\
\mathcal{A}(\circ \prec v) & = & (\texttt{Num}\ v) \\
\mathcal{A}((\texttt{Plus}\ \square\ e_2) \triangleright s \succ e_1) & = & \mathcal{A}(s \succ (\texttt{Plus}\ e_1\ e_2)) \\
\textit{etc.} & &
\end{array}
$$

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

$$\frac{}{s \succ (\texttt{Plus}\ e_1\ e_2) \quad \mapsto_C \quad (\texttt{Plus}\ \Box\ e_2) \triangleright s \succ e_1}$$

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

$$\begin{array}{rcl} \mathcal{A}(RHS) & = & \mathcal{A}((\texttt{Plus}\ \Box\ e_2) \triangleright s \succ e_1) \\ & = & \mathcal{A}(s \succ (\texttt{Plus}\ e_1\ e_2)) \\ & = & \mathcal{A}(LHS) \end{array}$$

Showing Refinement for Plus

$$\frac{}{(\texttt{Plus}\ \square\ e_2) \triangleright s \prec v_1 \quad \mapsto_C \quad (\texttt{Plus}\ v_1\ \square) \triangleright s \succ e_2}$$

Another no-op in the M-Machine:

$$\begin{array}{rcl} \mathcal{A}(LHS) & = & \mathcal{A}((\texttt{Plus}\ \square\ e_2) \triangleright s \prec v_1) \\ & = & \mathcal{A}(s \succ (\texttt{Plus}\ (\texttt{Num}\ v_1)\ e_2)) \\ & = & \mathcal{A}((\texttt{Plus}\ v_1\ \square) \triangleright s \succ e_2) \\ & = & \mathcal{A}(RHS) \end{array}$$

Showing Refinement for Plus

$$\frac{}{(\texttt{Plus}\ v_1\ \square) \triangleright s \prec v_2 \quad \mapsto_C \quad s \prec v_1 + v_2}$$

This corresponds to a M-Machine transition:

$$\begin{array}{rcll}
\mathcal{A}(LHS) & = & \mathcal{A}((\texttt{Plus}\ v_1\ \square) \triangleright s \prec v_2) & \\
& = & \mathcal{A}(s \succ (\texttt{Plus}\ (\texttt{Num}\ v_1)\ (\texttt{Num}\ v_2))) & \\
& \mapsto_M & \mathcal{A}(s \succ (\texttt{Num}\ (v_1 + v_2))) & (*) \\
& = & \mathcal{A}(s \prec v_1 + v_2) & \\
& = & \mathcal{A}(RHS) &
\end{array}$$

Technically the reduction step $(*)$ requires induction on the stack.