Abstract Mach

Dr. Liam O'Connor
University of Edinburgh LFCS
UNSW, Term 3 2020

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 $x_0 \in \mathbb{R}$ and a coefficient m s

$$\forall x > x_0.\ f(x) \leq$$

What is the codomain of f?

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

Cost Models

A *cost model* is a mathematical model that tries to measure of the cost of executing a program.

There exist *de*

However in this course we will focus on *operatio*

Operational Cost Models

First, we define a program-evaluating *abstract machine*. We can determine the time cost by counting the number of steps taken by the abstract machine.

Abstract Machines

Abstract Machines

An *abstract ma*

1. A set of states
2. A set of initial s
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 structural operational (or small-step) semantics.

The M Machine

Is just our usual small-step rules:

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

$$\ldots \mapsto_M e_3$$

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

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

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

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. **Substitut** $\mathcal{O}(n)$ time.

2. **Control F** recursivel

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

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

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!

Key Idea: States will consist of a current expression to evalua computational contexts that situate it in the overall computa would be:

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

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

Initial states are th

$\circ \succ e$. Final state $\circ \prec v$.

Stack frames are expressions with holes or values in them:

$$\frac{e_2\ \mathsf{Expr}}{(\mathtt{Plus}\ \Box\ e_2)\ \mathsf{Frame}} \qquad \frac{}{(\mathtt{Pl} \quad _1}$$

$$\cdots$$

Evaluating

There are three axioms about `Plus` now:

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

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

$$\overline{\quad {}_2 \quad {}_1 \mapsto_C \quad {}_1 \quad {}_2}$$

Once it is evaluated, return the s

$$\overline{(\texttt{Plus}\ v_1\ \Box) \triangleright s \prec v_2 \mapsto}$$

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

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

Example

$$\circ \succ \texttt{(Plus (Plus (Num 2) (Num 3)) (Num 4))}$$

$$\mapsto_C \texttt{(Plus}\ \square\ \texttt{(Num 4))} \rhd \circ \succ \texttt{(Plus (Num 2) (Num 3))}$$

$$\mapsto \quad \texttt{m 2)}$$

$$\mapsto_C$$

$$\mapsto_C \texttt{(Plus 2}\ \square\texttt{)} \rhd \texttt{(Plus}\ \square\ \texttt{(Num}$$

$$\mapsto_C \texttt{(Plus}\ \square\ \texttt{(Num 4))} \rhd \circ \prec 5$$

$$\mapsto_C \texttt{(Plus 5}\ \square\texttt{)} \rhd \circ \succ \texttt{(Num 4)}$$

$$\mapsto_C \texttt{(Plus 5}\ \square\texttt{)} \rhd \circ \prec 4$$

$$\mapsto_C \circ \prec 9$$

Other Rules

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

$$\frac{}{\;}$$

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

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

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 applic

$$\frac{}{s \succ (\texttt{Apply}\ e_1\ e_2) \mapsto_C (\texttt{A}}$$

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

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

We are still using substitution for now.

What have we done?

- All the rules a
  while loo
- We have a low
  assembly language)
- Substitution is still a machine operation – we need to find a w

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 (*co* [illegible] evel (*abstract*) semantics if every possible execution in the low-level semantics has a corresponding execution in the high-level semantics. In our case:

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

Functional correctness properties are preserved by refinement, but security properties are not (cf. Dining Cryptographers).

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:

$$\circ \succ (+\ (+\ (\mathtt{N}\ 2)\ (\mathtt{N}\ 3))\ (\mathtt{N}\ 4)) \qquad\qquad (+\ (+\ (\mathtt{N}\ 2)\ (\mathtt{N}\ 3))\ (\mathtt{N}\ 4))$$

$$\mapsto_C \ ($$

$$\mapsto_C \ ($$

$$\mapsto_C \ ($$

$$\mapsto_C \ (+\ 2\ \square) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \succ (\mathtt{N}\ 3)$$

$$\mapsto_C \ (+\ 2\ \square) \triangleright (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \prec 3$$

$$\mapsto_C \ (+\ \square\ (\mathtt{N}\ 4)) \triangleright \circ \prec 5 \qquad\qquad \mapsto_M$$

$$\mapsto_C \ (+\ 5\ \square) \triangleright \circ \succ (\mathtt{N}\ 4)$$

$$\mapsto_C \ (+\ 5\ \square) \triangleright \circ \prec 4$$

$$\mapsto_C \ \circ \prec 9 \qquad\qquad \mapsto_M \ (\mathtt{N}\ 9)$$

How to Prove Refinement

1. Define an *abstraction function* $\mathcal{A} : \Sigma_C \to \Sigma_M$ that relates C-Machine states to M-Machi
2. Prove that fo $\mathcal{A}(\sigma) \in I_M$.

3. Prove for ea
   - the step is a no-op in the M-Machine and $(\sigma_1) = (\sigma_2)$, or
   - the step is replicated by the M-Machine $\mathcal{A}$

4. Prove that for all *final* states $\sigma \in F_C$ that the c $(\sigma) \in F_M$.

In general this abstraction function is called a *si* pe 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:

$\circ \succ$ (+ (+ (N 2) (N 3)) (N 4)) — (+ (+ (N 2) (N 3)) (N 4))

$\mapsto_C$ (+ $\square$ (N 4))

$\mapsto_C$ (+ $\square$ (N 3))

$\mapsto_C$ (+ $\square$ (N 3))

$\mapsto_C$ (+ 2 $\square$) $\triangleright$ (+ $\square$ (N 4)) $\triangleright \circ \succ$ (N 3)

$\mapsto_C$ (+ 2 $\square$) $\triangleright$ (+ $\square$ (N 4)) $\triangleright \circ \prec$ 3

$\mapsto_C$ (+ $\square$ (N 4)) $\triangleright \circ \prec$ 5 — $\mapsto_M$ (+

$\mapsto_C$ (+ 5 $\square$) $\triangleright \circ \succ$ (N 4)

$\mapsto_C$ (+ 5 $\square$) $\triangleright \circ \prec$ 4

$\mapsto_C$ $\circ \prec$ 9 — $\mapsto_M$ (N 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.

$$=$$

$$= \ ($$

$$\mathcal{A}((\texttt{Plus}\ \square\ e_2) \triangleright s \succ e_1) \quad = \quad \texttt{Plus}$$

etc.

By definition, all the initial/final states of the C-Machine are m

states of the M-Machine. So all that is left is the requirement for each transition.

Showing Refinement for Plus

This is a no-op in the M

$$\mathcal{A}(RHS) = \mathcal{A}((\texttt{Plus}$$
$$= \mathcal{A}(s \succ (\texttt{P}$$
$$= \mathcal{A}(LHS)$$

Showing Refinement for Plus

2

Another no-op in t

$$\begin{aligned}
\mathcal{A}(LHS) &= \mathcal{A}((\texttt{Plus}\ \square\ e\ ) \triangleright s\quad v\ )\\
&= \mathcal{A}(s \succ (\texttt{Pl}\\
&= \mathcal{A}((\texttt{Plus}\ v\\
&= \mathcal{A}(RHS)
\end{aligned}$$

Showing Refinement for Plus

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

This correspond

$$\begin{array}{cl} = & \mathcal{A}(s \succ (\texttt{Plus}\ (\texttt{N} \\ \mapsto_M & \mathcal{A}(s \succ (\texttt{Num}\ v_1 \\ = & \mathcal{A}(s \prec v_1 + v_2) \\ = & \mathcal{A}(RHS) \end{array}$$

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