Abstract Machines

Johannes Åman Pohjola
UNSW
Term 3 2022
Big O

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

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

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

What is the codomain of $f$?

When analysing algorithms, we don't usually time how long they take to run on a real machine.
Q: How would you derive the complexity of this mergesort?

\[
\text{mergesort}([],) = [] \\
\text{mergesort}(xs) = \\
\text{let } (ys, zs) = \text{partition } xs; \\
\quad ys' = \text{mergesort } ys; \\
\quad zs' = \text{mergesort } zs \\
\text{in } \text{merge } ys' zs'
\]
Q: How would you derive the complexity of this mergesort?

mergesort([]) = [] \quad f(0) = c_1
mergesort(xs) = \quad f(n) =
  let (ys, zs) = partition xs;
    ys' = mergesort ys;
    zs' = mergesort zs
  in merge ys' zs'
    c_2 \times n + f(n/2) + f(n/2) + c_3 \times n

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

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

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

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

mergesort([]) = []  \quad f(0) = c_1
mergesort(xs) =  \quad f(n) =
    let (ys, zs) = partition xs;
        ys' = mergesort ys;
        zs' = mergesort zs
    in merge ys' zs'
        c_2 \times n + f(n/2) + c_3 \times 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:

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

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 $\mapsto \subseteq \Sigma \times \Sigma$.

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

Is just our usual small-step rules:

\[
\begin{align*}
  e_1 & \mapsto_M e'_1 \\
  (\text{Plus } e_1 e_2) & \mapsto_M (\text{Plus } e'_1 e_2) \\
  (\text{If } e_1 e_2 e_3) & \mapsto_M (\text{If } e'_1 e_2 e_3) \\
  (\text{If } (\text{Lit True}) e_2 e_3) & \mapsto_M e_2 \\
  (\text{If } (\text{Lit False}) e_2 e_3) & \mapsto_M e_3 \\
  (\text{Apply } e_1 e_2) & \mapsto_M (\text{Apply } e'_1 e_2) \\
  e_2 & \mapsto_M e'_2 \\
  (\text{Apply } (\text{Recfun } (f.x. e)) e_2) & \mapsto_M (\text{Apply } (\text{Recfun } (f.x. e)) e'_2) \\
  v \in F \\
  (\text{Apply } (\text{Recfun } (f.x. e)) v) & \mapsto_M e[x := v, f := (\text{Recfun } (f.x. e))] 
\end{align*}
\]

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.
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.

\[
\begin{align*}
\text{eval (Num } n\text{)} & = n \\
\text{eval } e & = \text{eval (oneStep } e) \\
\text{oneStep (Plus (Num } n\text{) (Num } m\text{))} & = \text{Num (} n + m\text{)} \\
\text{oneStep (Plus (Num } n\text{) } e_2\text{)} & = \text{Plus (Num } n\text{) (oneStep } e_2\text{)} \\
\text{oneStep (Plus } e_1\text{ } e_2\text{)} & = \text{Plus (oneStep } e_1\text{) } e_2 \\
\end{align*}
\]

...
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?
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!**

\[ \text{Stack} \quad f \text{ Frame} \quad s \text{ Stack} \]

\[ \text{Stack} \quad f \triangleright s \text{ Stack} \]

**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:

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

This represents the computational context:

\[(\text{Times (Plus 3 } \square) (\text{Num 2}))\]
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:
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:

\[
\begin{align*}
\text{e}_2 \text{ Expr} & \quad \text{v}_1 \text{ Value} \\
(\text{Plus} \ [ \ [ e_2 ] ] \text{ Frame}) & \quad (\text{Plus} \ [ \ [ v_1 ] ] \text{ Frame}) \quad \ldots
\end{align*}
\]
The C Machine

Our states will consist of two modes:

1. **Evaluate** the current expression within stack $s$, written $s \triangleright e$.
2. **Return** a value $v$ (either a function, integer, or boolean) back into the context in $s$, written $s \triangleleft v$.

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

Stack frames are expressions with holes or values in them:

\[
\begin{align*}
\text{e}_2 & \quad \text{Expr} \\
\text{(Plus } \square \text{ e}_2) & \quad \text{Frame} \\
\text{v}_1 & \quad \text{Value} \\
\text{(Plus v}_1 \square \text{)} & \quad \text{Frame} \\
\vdots
\end{align*}
\]
Evaluating

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

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

\[
\begin{align*}
  & s \succ (\texttt{Plus} \ e_1 \ e_2) \quad \rightarrow_C \quad (\texttt{Plus} \ \Box \ e_2) \triangleright s \succ e_1 \\
\end{align*}
\]
Evaluating

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

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

\[
s \succ (\texttt{Plus} \; e_1 \; e_2) \implies_C (\texttt{Plus} \; \square e_2) \triangleright s \succ e_1
\]

Once the LHS is evaluated, switch to the RHS:

\[
(\texttt{Plus} \; \square e_2) \triangleright s \prec v_1 \implies_C (\texttt{Plus} \; v_1 \; \square) \triangleright s \succ e_2
\]
Evaluating

There are three axioms about $\text{Plus}$ now:

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

\[
\begin{align*}
  s \succ (\text{Plus} \ e_1 \ e_2) & \mapsto_C \ (\text{Plus} \ □ \ e_2) \triangleright s \succ e_1 \\
\end{align*}
\]

Once the LHS is evaluated, switch to the RHS:

\[
\begin{align*}
  (\text{Plus} \ □ \ e_2) \triangleright s \prec v_1 & \mapsto_C \ (\text{Plus} \ v_1 \ □) \triangleright s \succ e_2 \\
\end{align*}
\]

Once the RHS is evaluated, return the sum:

\[
\begin{align*}
  (\text{Plus} \ v_1 \ □) \triangleright s \prec v_2 & \mapsto_C \ s \prec v_1 + v_2 \\
\end{align*}
\]
Evaluating

There are three axioms about Plus now:

When evaluating a Plus expression, first evaluate the LHS:

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

Once the LHS is evaluated, switch to the RHS:

\[ (\text{Plus } \Box e_2) \triangleright s \prec v_1 \mapsto_C (\text{Plus } v_1 \Box) \triangleright s \succ e_2 \]

Once the RHS is evaluated, return the sum:

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

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

\[ s \succ (\text{Num } n) \mapsto_C s \prec n \]
Example

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

\( \circ \succ (Plus (Plus (Num 2) (Num 3)) (Num 4)) \)

\( \mapsto_C (Plus \ □ (Num 4)) \triangleright \circ \succ (Plus (Num 2) (Num 3)) \)
Example

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

\[ \mapsto_c (\text{Plus} \Box (\text{Num} 4)) \triangleright o \succ (\text{Plus} (\text{Num} 2) (\text{Num} 3)) \]

\[ \mapsto_c (\text{Plus} \Box (\text{Num} 3)) \triangleright (\text{Plus} \Box (\text{Num} 4)) \triangleright o \succ (\text{Num} 2) \]
Example

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

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

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

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

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

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

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

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

\[\mapsto_C \ (Plus \ 2 \ □) \triangleright (Plus \ □ \ (Num \ 4)) \triangleright \circ \succ (Num \ 3)\]
Example

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

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

\[ \mapsto_C \quad (\text{Plus} \boxminus (\text{Num} 3)) \triangleright (\text{Plus} \boxminus (\text{Num} 4)) \triangleright \circ \preceq (\text{Num} 2) \]

\[ \mapsto_C \quad (\text{Plus} \boxminus (\text{Num} 3)) \triangleright (\text{Plus} \boxminus (\text{Num} 4)) \triangleright \circ \preceq 2 \]

\[ \mapsto_C \quad (\text{Plus} 2 \boxminus) \triangleright (\text{Plus} \boxminus (\text{Num} 4)) \triangleright \circ \preceq 3 \]

\[ \mapsto_C \quad (\text{Plus} 2 \boxminus) \triangleright (\text{Plus} \boxminus (\text{Num} 4)) \triangleright \circ \preceq 4 \]

\[ \mapsto_C \quad \circ \preceq 5 \]

\[ \mapsto_C \quad \circ \succ 2 \]

\[ \mapsto_C \quad \circ \succ 3 \]
Example

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

\[ \mapsto C (\text{Plus} \square (\text{Num} 4)) \rhd \circ \succ (\text{Plus} (\text{Num} 2) (\text{Num} 3)) \]

\[ \mapsto C (\text{Plus} \square (\text{Num} 3)) \rhd (\text{Plus} \square (\text{Num} 4)) \rhd \circ \succ (\text{Num} 2) \]

\[ \mapsto C (\text{Plus} \square (\text{Num} 3)) \rhd (\text{Plus} \square (\text{Num} 4)) \rhd \circ \prec 2 \]

\[ \mapsto C (\text{Plus} 2 \square) \rhd (\text{Plus} \square (\text{Num} 4)) \rhd \circ \succ (\text{Num} 3) \]

\[ \mapsto C (\text{Plus} 2 \square) \rhd (\text{Plus} \square (\text{Num} 4)) \rhd \circ \prec 3 \]

\[ \mapsto C (\text{Plus} \square (\text{Num} 4)) \rhd \circ \prec 5 \]

\[ \mapsto C (\text{Plus} 5 \square) \rhd \circ \succ (\text{Num} 4) \]

\[ \mapsto C (\text{Plus} 5 \square) \rhd \circ \prec 4 \]

\[ \mapsto C \circ \prec 9 \]
Other Rules

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

\[
s \succ (\text{If } e_1 \ e_2 \ e_3) \mapsto_C (\text{If } \Box e_2 \ e_3) \triangleright s \succ e_1
\]
Other Rules

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

\[
s ≻ (\text{If } e_1 e_2 e_3) \mapsto_C (\text{If } \square e_2 e_3) \triangleright s ≻ e_1
\]

\[
(\text{If } \square e_2 e_3) \triangleright s \prec \text{True} \mapsto_C s ≻ e_2
\]

\[
(\text{If } \square e_2 e_3) \triangleright s \prec \text{False} \mapsto_C s ≻ e_3
\]
Functions

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

\[ s \succ (\text{Fun } (f.x. e)) \mapsto_C s \prec \langle f.x. e \rangle \]
Functions

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

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

Function application is then handled similarly to Plus.

\[
s \succ (\text{Apply} \ e_1 \ e_2) \mapsto_C (\text{Apply} \ □ e_2) \triangleright s \succ e_1
\]
Functions

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

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

Function application is then handled similarly to Plus:

\[ s \succ (\text{Apply} \, e_1 \, e_2) \rightarrow_C (\text{Apply} \, \Box \, e_2) \triangleright s \succ e_1 \]

\[ (\text{Apply} \, \Box \, e_2) \triangleright s \prec \langle \langle f \cdot x \cdot e \rangle \rangle \rightarrow_C (\text{Apply} \, \langle \langle f \cdot x \cdot e \rangle \rangle \, \Box) \triangleright s \succ e_2 \]
Functions

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

\[
s 	riangleright (\text{Fun } (f.x. e)) \mapsto_C s \leftarrow (f.x. e)
\]

Function application is then handled similarly to Plus:

\[
s 	riangleright (\text{Apply } e_1 e_2) \mapsto_C (\text{Apply } \square e_2) \triangleright s 	riangleright e_1
\]

\[
(\text{Apply } \square e_2) \triangleright s \leftarrow (f.x. e) \mapsto_C (\text{Apply } (f.x. e) \square) \triangleright s 	riangleright e_2
\]

\[
(\text{Apply } (f.x. e) \square) \triangleright s \leftarrow v \mapsto_C s 	riangleright e[x := v, f := (\text{Fun } (f.x.e))]
\]

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.
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:

$$\forall e, v. \quad e \xrightarrow{\ast} M \quad \Rightarrow \quad e \xrightarrow{\ast} C \quad \Rightarrow \quad e \xrightarrow{\ast} M \quad \Rightarrow \quad v$$

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:

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

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

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

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

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

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

\[
\mapsto_C (+ \ \square (N \ 4)) \triangleright \circ \prec 5 \quad \mapsto_M (+ (N \ 5) (N \ 4))
\]

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

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

\[
\mapsto_C \circ \prec 9 \quad \mapsto_M (N \ 9)
\]
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 $\sigma_1 \mapsto_C \sigma_2$, 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 $A(\sigma_1) \mapsto_M A(\sigma_2)$.

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 $\mathcal{A}$ 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:

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

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

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

\[
\rightarrow_C (+ \square (N 3)) \triangleright (+ \square (N 4)) \triangleright \circ \prec 2
\]

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

\[
\rightarrow_C (+ 2 \square) \triangleright (+ \square (N 4)) \triangleright \circ \prec 3
\]

\[
\rightarrow_C (+ \square (N 4)) \triangleright \circ \prec 5 \rightarrow \rightarrow_C (+ (N 5) (N 4))
\]

\[
\rightarrow_C (+ 5 \square) \triangleright \circ \succ (N 4)
\]

\[
\rightarrow_C (+ 5 \square) \triangleright \circ \prec 4
\]

\[
\rightarrow_C \circ \prec 9 \rightarrow \rightarrow_C (N 9)
\]

\[
\rightarrow_C \circ \prec 40
\]
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.

\[ A(\circ \succ e) = e \]
\[ A(\circ \prec v) = (\text{Num } v) \]
\[ A((\text{Plus } \Box e_2) \succ s \succ e_1) = A(s \succ (\text{Plus } e_1 e_2)) \]

\[ etc. \]

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

\[ s \succ (\text{Plus } e_1 e_2) \xrightarrow{C} (\text{Plus } \bigcirc e_2) \succ s \succ e_1 \]
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:

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

\[(\text{Plus } \Box e_2) \triangleright s \prec v_1 \quad \mapsto_C \quad (\text{Plus } v_1 \Box) \triangleright s \succ e_2\]
Showing Refinement for Plus

\[(\text{Plus } □ e_2) \triangleright s ≺ v_1 \quad \mapsto_\mathcal{C} \quad (\text{Plus } v_1 □) \triangleright s ≻ e_2\]

Another no-op in the M-Machine:

\[
\begin{align*}
\mathcal{A}(LHS) & = \mathcal{A}((\text{Plus } □ e_2) \triangleright s ≺ v_1) \\
& = \mathcal{A}(s ≻ (\text{Plus } (\text{Num } v_1) e_2)) \\
& = \mathcal{A}((\text{Plus } v_1 □) \triangleright s ≻ e_2) \\
& = \mathcal{A}(RHS)
\end{align*}
\]
Showing Refinement for Plus

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

This corresponds to a M-Machine transition:

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

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