

Abstract Machines

Thomas Sewell
UNSW
Term 3 2025

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

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

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

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?

```
mergesort([]) = []  
mergesort(xs) =  
  let (ys, zs) = partition xs;  
      ys' = mergesort ys;  
      zs' = mergesort zs  
  in merge ys' zs'
```

Big O

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

<code>mergesort([]) = []</code>	$f(0) = c_1$
<code>mergesort(xs) =</code>	$f(n) =$
let (ys, zs) = partition xs;	$c_2 * n +$
ys' = mergesort ys;	$f(n/2) +$
zs' = mergesort zs	$f(n/2) +$
in merge ys' zs'	$c_3 * n$

A: Define a **cost function** f , then find its closed form.

Big O

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

<code>mergesort([]) = []</code>	$f(0) = c_1$
<code>mergesort(xs) =</code>	$f(n) =$
let <code>(ys, zs) = partition xs;</code>	$c_2 * n +$
<code>ys' = mergesort ys;</code>	$f(n/2) +$
<code>zs' = mergesort zs</code>	$f(n/2) +$
in merge <code>ys' zs'</code>	$c_3 * n$

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?

Big O

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

<code>mergesort([]) = []</code>	$f(0) = c_1$
<code>mergesort(xs) =</code>	$f(n) =$
let <code>(ys, zs) = partition xs;</code>	$c_2 * n +$
<code>ys' = mergesort ys;</code>	$f(n/2) +$
<code>zs' = mergesort zs</code>	$f(n/2) +$
in merge <code>ys' zs'</code>	$c_3 * n$

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.

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:

$$\llbracket \cdot \rrbracket : \text{Program} \rightarrow \text{Cost}$$

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

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:

$$\begin{array}{c}
 \frac{e_1 \mapsto_M e'_1}{(\text{Plus } e_1 \ e_2) \mapsto_M (\text{Plus } e'_1 \ e_2)} \quad \dots \\
 \\
 \frac{e_1 \mapsto_M e'_1}{(\text{If } e_1 \ e_2 \ e_3) \mapsto_M (\text{If } e'_1 \ e_2 \ e_3)} \\
 \frac{}{(\text{If } (\text{Lit True}) \ e_2 \ e_3) \mapsto_M e_2} \quad \frac{}{(\text{If } (\text{Lit False}) \ e_2 \ e_3) \mapsto_M e_3} \\
 \\
 \frac{e_1 \mapsto_M e'_1}{(\text{Apply } e_1 \ e_2) \mapsto_M (\text{Apply } e'_1 \ e_2)} \\
 \\
 \frac{e_2 \mapsto_M e'_2}{(\text{Apply } (\text{Recfun } (f.x. \ e)) \ e_2) \mapsto_M (\text{Apply } (\text{Recfun } (f.x. \ e)) \ e'_2)} \\
 \\
 \frac{v \in F}{(\text{Apply } (\text{Recfun } (f.x. \ e)) \ v) \mapsto_M e[x := v, f := (\text{Recfun } (f.x. \ e))]}
 \end{array}$$

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.

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.

$$\text{eval} (\text{Num } n) = n$$

$$\text{eval } e = \text{eval} (\text{oneStep } e)$$

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

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

$$\text{oneStep} (\text{Plus } e_1 e_2) = \text{Plus} (\text{oneStep } e_1) e_2$$

...

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?

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!

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

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:

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

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:

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:

$$\frac{e_2 \text{ Expr}}{(\text{Plus } \square e_2) \text{ Frame}} \quad \frac{v_1 \text{ Value}}{(\text{Plus } v_1 \square) \text{ Frame}}$$

...

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:

$$\frac{e_2 \text{ Expr}}{(\text{Plus } \square e_2) \text{ Frame}} \quad \frac{v_1 \text{ Value}}{(\text{Plus } v_1 \square) \text{ Frame}}$$

...

Evaluating

There are three axioms about Plus now:

When evaluating a Plus expression, first evaluate the LHS:

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

Evaluating

There are three axioms about Plus now:

When evaluating a Plus expression, first evaluate the LHS:

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

Once the LHS is evaluated, switch to the RHS:

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

Evaluating

There are three axioms about `Plus` now:

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

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

Once the LHS is evaluated, switch to the RHS:

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

Once the RHS is evaluated, return the sum:

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

Evaluating

There are three axioms about `Plus` now:

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

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

Once the LHS is evaluated, switch to the RHS:

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

Once the RHS is evaluated, return the sum:

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

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

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

Example

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

Example

$\circ \succ (\text{Plus } (\text{Plus } (\text{Num } 2) (\text{Num } 3)) (\text{Num } 4))$
 $\mapsto_C (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Plus } (\text{Num } 2) (\text{Num } 3))$

Example

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

$\mapsto_C (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Plus } (\text{Num } 2) (\text{Num } 3))$

$\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Num } 2)$

Example

$\circ \succ (\text{Plus } (\text{Plus } (\text{Num } 2) (\text{Num } 3)) (\text{Num } 4))$
 $\mapsto_C (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Plus } (\text{Num } 2) (\text{Num } 3))$
 $\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Num } 2)$
 $\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \prec 2$

Example

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

$\mapsto_C (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Plus } (\text{Num } 2) (\text{Num } 3))$

$\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Num } 2)$

$\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \prec 2$

$\mapsto_C (\text{Plus } 2 \square) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Num } 3)$

Example

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

$\mapsto_C (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Plus } (\text{Num } 2) (\text{Num } 3))$

$\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Num } 2)$

$\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \prec 2$

$\mapsto_C (\text{Plus } 2 \square) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Num } 3)$

$\mapsto_C (\text{Plus } 2 \square) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \prec 3$

Example

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

$\mapsto_C (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Plus } (\text{Num } 2) (\text{Num } 3))$

$\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Num } 2)$

$\mapsto_C (\text{Plus } \square (\text{Num } 3)) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \prec 2$

$\mapsto_C (\text{Plus } 2 \square) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \succ (\text{Num } 3)$

$\mapsto_C (\text{Plus } 2 \square) \triangleright (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \prec 3$

$\mapsto_C (\text{Plus } \square (\text{Num } 4)) \triangleright \circ \prec 5$

$\mapsto_C (\text{Plus } 5 \square) \triangleright \circ \succ (\text{Num } 4)$

$\mapsto_C (\text{Plus } 5 \square) \triangleright \circ \prec 4$

$\mapsto_C \circ \prec 9$

Other Rules

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

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

Other Rules

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

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

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

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

Functions

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

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

Functions

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

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

Function application is then handled similarly to Plus.

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

Functions

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

$$\frac{}{s \succ (\text{Fun } (f.x. e)) \mapsto_C s \prec \llbracket f.x. e \rrbracket}$$

Function application is then handled similarly to Plus.

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

$$\frac{}{(\text{Apply } \square e_2) \triangleright s \prec \llbracket f.x. e \rrbracket \mapsto_C (\text{Apply } \llbracket f.x. e \rrbracket \square) \triangleright s \succ e_2}$$

Functions

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

$$\frac{}{s \succ (\text{Fun } (f.x. e)) \mapsto_C s \prec \llbracket f.x. e \rrbracket}$$

Function application is then handled similarly to Plus.

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

$$\frac{}{(\text{Apply } \square e_2) \triangleright s \prec \llbracket f.x. e \rrbracket \mapsto_C (\text{Apply } \llbracket f.x. e \rrbracket \square) \triangleright s \succ e_2}$$

$$\frac{}{(\text{Apply } \llbracket f.x. e \rrbracket \square) \triangleright s \prec v \mapsto_C s \succ e[x := v, f := (\text{Fun } (f.x.e))]}$$

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:

$$\forall e, v. \frac{\begin{array}{ccc} \circ \succ e & \xrightarrow{*}_C & \circ \succ v \end{array}}{\begin{array}{ccc} e & \xrightarrow{*}_M & v \end{array}}$$

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:

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

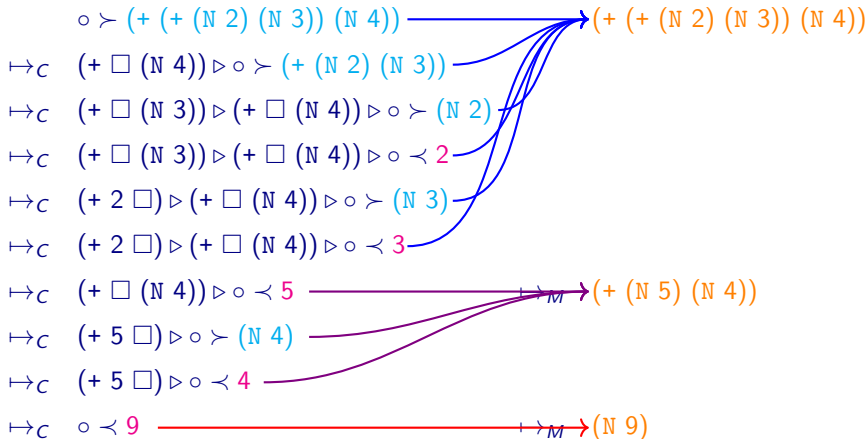
How to Prove Refinement

- 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:



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.

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

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

$$s \succ (\text{Plus } e_1 \ e_2) \quad \vdash_C \quad (\text{Plus } \square \ e_2) \triangleright s \succ e_1$$

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:

$$\begin{aligned} \mathcal{A}(RHS) &= \mathcal{A}((\text{Plus } \square \ e_2) \triangleright s \succ e_1) \\ &= \mathcal{A}(s \succ (\text{Plus } e_1 \ e_2)) \\ &= \mathcal{A}(LHS) \end{aligned}$$

Showing Refinement for Plus

$$(\text{Plus } \square e_2) \triangleright s \prec v_1 \quad \vdash_C \quad (\text{Plus } v_1 \square) \triangleright s \succ e_2$$

Showing Refinement for Plus

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

Another no-op in the M-Machine:

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

Showing Refinement for Plus

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

This corresponds to a M-Machine transition:

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

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