COMP3161/COMP9164 Supplementary Lecture Notes

Abstract Machines

Gabriele Keller, Liam O'Connor

October 20, 2019

1 Abstract Machine

Abstract machines are a theore
including initial and final states, and a set of machine operations, which manipulate the state of the machine. They are an important concept in theoretical computer science, for example to specify the operational semantics of a programming language (as we did with MinHS and TinyImp), or computational and complexity theory. Probably one of the best known examples of an abstract machine is the Turing machine, which was designed by Alan Turing in 1936 as a means to tackle the Entscheidungsproblem (Decision Problem).

Abstract Machines are closely related to Virtual Machines, which are basically Abstract Machines with an imple gh-level programming l Framework for the .NET framewo

In this course, we us
programming languages. We started with two simple machines, w
operations and states. We will gradually add more details to the machin
to a machine we could actually use as a basis for an implementation. We do th
make languages easier to reason about - we use the higher level specificat
step semantics) for reasoning about higher level properties such as safety, and use the lower level specification for reasoning about machine characteristics such as performance.

Technically, you have already seen two examples of abstract machines - in the structural operational semantics of a language (the "small step" semantics), and in the evaluation ("big step") semantics. While these technically constitute abstract machines, they are perhaps *too* abstract for our needs.

We seek to specify the evaluation of a language in a more "low-level" way. Should we start from the evaluation semantics, or the small-step semantics? Notice that the evaluation semantics do not even specify the *order* in which terms should be evaluated. Seeing as nondeterminism doesn't exist in real computers, we must specify such an order. Hence we could say that the small-step semantics are "lower-level" than the big-step. Therefore, it makes sense for us to start with the small-step semantics. We call the small-step semantics of MinHS the *M Machine*.

2 The *C Machine*

One thing still left rather abstract in the *M Machine* is control flow, specifically, the notion of a *runtime stack*. When we want to evaluate a term, say `Plus (Plus (Num 2) (Num 3)) (Num 4)`, the *M Machine* rules tell us that we must first evaluate the inner `Plus (Num 2) (Num 3)` and then return it to the original expression. This is akin to *pushing* the greater expression context onto a stack, evaluating the inner expression and then *popping* the context off the stack again, returning the evaluated expression to it.

We will introduce a new machine, the *C Machine*, that makes this stack explicit. In order to do this, we will need to formalise it, but we will get an informal intuition for it first.

2.1 *C Machine* States

In the *M Machine*, the state of the machine merely consists of the current expression to be evaluated - the notion of the stack is hidden in the deduction tree of the inference rules. In our *C Machine*, however, our state consists of three parts:

- The current expression to be evaluated
- A *stack* of expression *frames*
- A single binary flag that d … *g* an expression, or *returning* a value a

To start with, we will introduce s

$$\frac{}{\circ \ \textit{Stack}} \qquad \frac{s\ \textit{Stack} \quad x\ \textit{Frame}}{x \triangleright s\ \textit{Stack}}$$

So, $\circ$ represents the empty stack, and $x \triangleright s$ is a stack with a frame x on the top. But what is a frame? We define a frame as an expression with a hole in it, denoted by $\square$. For example:

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

Is a `Plus` expressi … *suspended computations*, whi

2.1.1 A sketch of a *C Machine*

Now that we have the stack, we can have a shot at representing the mac … want to evaluate our earlier example with the *C Mach*

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

To begin, we need to come up with an *initial state* for our expression. So, when we start, we have the empty stack ($\circ$), and the machine starts with the flag set to *evaluate* the expression (denoted by $\succ$).

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

The rules in the *M Machine* state that in order to evaluate a `Plus` expression, first the first subexpression should be evaluated, then the second. Hence, in this case, our *C Machine* will mirror the behaviour of *M Machine* and therefore should evaluate `Plus (Num 2) (Num 3)` first. To achieve this, a stack frame is pushed for `Plus`, with a $\square$ in place of the first subexpression, and the machine is set to evaluate the expression we just replaced:

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

Now the machine has to evaluate another `plus`, so another stack frame is pushed:

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

Now the machine simply has to evaluate a numeric literal. Seeing as no further evaluation need take place (the value associated with the expression can be inferred without further work), the machine switches to *return* (denoted by $\prec$) the value back into the awaiting stack frame:

$$\texttt{Plus}\ \square\ (\texttt{Num}\ 3) \triangleright \texttt{Plus}\ \square\ (\texttt{Num}\ 4) \triangleright \circ \quad \prec \quad 2$$

Having evaluated the first argument, the machine again suspends computation of the `plus` expression in order to evaluate the second subexpression, which proceeds similarly:

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

As the machine is *returning* the last value necessary for the `plus` frame, it *pops* the frame off the stack, performs a primitive addition operation, and returns the result 5:

$$\texttt{Plus}\ \square\ \texttt{Num}\ 4 \triangleright \circ \quad \prec \quad 5$$

The rest of the evaluation proceeds predictably:

$$\texttt{Plus}\ 5 \quad \triangleright \qquad \texttt{Num}\ 4$$

2.2 Formalising the *C Machine*

Now that we have an informal idea of what our *C Machine* looks like, we can begin to formalise the machine. An abstract machine in general consists of:

- A set of states Σ.
- A set of initial states $I \subseteq \Sigma$.
- A set of final stat
- A *state tran*

You have seen this before in small-step semantics - this is because small-step semantics are a form of abstract machine.

We can specify formally each of these components. The set of stat

$$\frac{s\ \mathit{Stack} \quad e\ \mathit{Expr}}{s \succ e \in \Sigma} \qquad \frac{s}{s \prec v \in \Sigma}$$

That is, Σ is comprised of all evaluating states and all returning states.

The initial states set I is defined as the set of all evaluating states with an empty stack:

$$\frac{e\ \mathit{Expr}}{\circ \succ e \in I}$$

And the final states F are defined as all returning states with an empty stack:

$$\frac{v\ \mathit{Value}}{\circ \prec v \in F}$$

Now we must define the state transition relation for our *C Machine*, $\mapsto_c$.

2.2.1 Literals

To begin, we will start on the easy part - Evaluating numeric and boolean literals:

$$\frac{}{s \succ \texttt{Num}\ v \mapsto_c s \prec v} \qquad \frac{}{s \succ \texttt{Bool}\ v \mapsto_c s \prec v}$$

So, the machine simply returns the corresponding values unchanged - no further computation is necessary. For function values, we must introduce some new notation, but the principle is the same as for numbers and booleans:

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

2.2.2 Primitive Operations

Now we can specify more complicated rules, such as that for `Plus`. When faced with an unevaluated `Plus` expression, the machine first evaluates the first subexpression, and pushes the rest on the stack:

$$\frac{}{s \succ \texttt{Plus}\ e_1\ e_2 \mapsto_c \texttt{Plus}\ \Box\ \ e_2 \rhd s \succ e_1}$$

Once that subexpression is evaluated, the machine will begin evaluating the second subexpression:

$$\frac{}{\texttt{Plus}\ \Box\ \ e_2 \rhd s \prec v \mapsto_c \texttt{Plus}\ v\ \Box \rhd s \succ e_2}$$

Finally, when both subexpressions are evaluated, the machine returns the resultant sum, computed via a primitive operation, `+`:

[illegible]

The definitions for `Time` [illegible] f different primitive machine operations.

2.2.3 Conditionals

Now, what about `if`? Recall that in the *M Machine*, the machine first evaluates the condition to some result, and, depending on the result, would evaluate just one branch of the conditional.

Similarly, when faced with an unevaluated `If` expression, the *C Machine* evalutes the condition first:

[illegible]

If the result is `Tr` [illegible]

$$\frac{}{\texttt{If}\ \Box\ \ t\ e \rhd s \prec \texttt{True} \mapsto_c s \succ t} \qquad \frac{}{\texttt{If}\ [illegible] \succ e}$$

2.2.4 Function Application

Finally, we must deal with function application. Recall that in the *M Machine*, first the function expression was evaluated, then the argument to the function, then finally the body of the function, substituting in the value of the argument. We employ a similar strategy here.

First evaluate the function value:

$$\frac{}{s \succ \texttt{Apply}\ f\ \ a \mapsto_c \texttt{Apply}\ \Box\ \ a \rhd s \succ f}$$

Then, evaluate the argument:

$$\frac{}{\texttt{Apply}\ \Box\ \ a \rhd s \prec \langle\!\langle f.x.\ e \rangle\!\rangle \mapsto_c \texttt{Apply}\ \langle\!\langle f.x.\ e \rangle\!\rangle\ \Box \rhd s \succ a}$$

Then finally evaluate the function body, substituting the value for the argument, and the function name in case the function is recursive:

$$\frac{}{\texttt{Apply}\ \langle\!\langle f.x.\ e \rangle\!\rangle\ \Box \rhd s \prec v_a \mapsto_c s \succ e[f := (\texttt{Fun}\ f.x.\ e), x = v_a]}$$

2.3 Example

Here we have a simple function that determines if the provided argument is even, applied to the argument 3. To make things shorter, we rename the `Num` abstract syntax expression to simply `N`, `Minus` to `Sub`, and `Apply` to `Ap`.

$$
\begin{array}{lrcl}
 & \circ & \succ & \texttt{Ap (Fun } f.x.\texttt{(If (LEq } x \texttt{ (N 0)) (Eq } x \texttt{ (N 0)) (Ap } f \texttt{ (Sub } x \texttt{ (N 2))))) (N 3)} \\
\mapsto_c & \texttt{Ap } \square \texttt{ (N 3)} \triangleright \circ & \succ & \texttt{Fun } f.x.\texttt{(If (LEq } x \texttt{ (N 0)) (Eq } x \texttt{ (N 0)) (Ap } f \texttt{ (Sub } x \texttt{ (N 2))))} \\
\mapsto_c & \texttt{Ap } \square \texttt{ (N 3)} \triangleright \circ & \prec & \langle\langle f.x.\texttt{(If (LEq } x \texttt{ (N 0)) (Eq } x \texttt{ (N 0)) (Ap } f \texttt{ (Sub } x \texttt{ (N 2))))} \rangle\rangle \\
\mapsto_c & \texttt{Ap } \langle\langle \cdots \rangle\rangle\ \square \triangleright \circ & \succ & \texttt{N 3} \\
\mapsto_c & \texttt{Ap } \langle\langle \cdots \rangle\rangle\ \square \triangleright \circ & \prec & \texttt{3} \\
\mapsto_c & \circ & \succ & \texttt{If (LEq (N 3) (N 0)) (Eq (N 3) (N 0)) (Ap (Fun } \cdots \texttt{) (Sub (N 3) (N 2)))} \\
\mapsto_c & \texttt{If } \square\ \cdots \triangleright \circ & \succ & \texttt{LEq (N 3) (N 0)} \\
\mapsto_c & \texttt{LEq } \square \texttt{ (N 0)} \triangleright \texttt{If } \cdots \triangleright \circ & \succ & \texttt{N 3} \\
\mapsto_c & \texttt{LEq } \square \texttt{ (N 0)} \triangleright \texttt{If } \cdots \triangleright \circ & \prec & \texttt{3} \\
\mapsto_c & \texttt{LEq 3 } \square \triangleright \texttt{If } \cdots \triangleright \circ & \succ & \texttt{N 0} \\
\mapsto_c & \texttt{LEq 3 } \square \triangleright \texttt{If } \cdots \triangleright \circ & \prec & \texttt{0} \\
\mapsto_c & \texttt{If } \square\ \cdots\ \texttt{(Ap } \cdots \texttt{)} \triangleright \circ & \prec & \texttt{False} \\
\mapsto_c & & & \texttt{(Ap (Fun }\quad\texttt{) (Sub (N 3) (N 2)))} \\
\mapsto_c & \texttt{Ap } \square \texttt{ (Su} & & \\
\mapsto_c & \texttt{Ap } \square \texttt{ (Su} & & \\
\mapsto_c & \texttt{Ap } \langle & & \\
\mapsto_c & \texttt{Sub } \square \texttt{ (N 2)} \triangleright & & \\
\mapsto_c & \texttt{Sub } \square \texttt{ (N 2)} \triangleright \texttt{Ap } \cdots \triangleright \circ & \prec & \texttt{3} \\
\mapsto_c & \texttt{Sub 3 } \square \triangleright \texttt{Ap } \cdots \triangleright \circ & \succ & \texttt{N 2} \\
\mapsto_c & \texttt{Sub 3 } \square \triangleright \texttt{Ap } \cdots \triangleright \circ & \prec & \texttt{2} \\
\mapsto_c & \texttt{Ap } \langle\langle \cdots \rangle\rangle\ \square \triangleright \circ & \prec & \texttt{1} \\
\mapsto_c & \circ & \succ & \texttt{If (LEq (N 1) (N 0)) (Eq (N 1) (N 0)) (Ap (Fun } \cdots \texttt{) (Sub (N 1) (N 2)))} \\
\mapsto_c & \texttt{If } \square\ \cdots \triangleright \circ & \succ & \texttt{LEq (N 1) (N 0)} \\
\mapsto_c & \texttt{LEq } \square \texttt{ (N 0)} \triangleright \texttt{If } \cdots \triangleright \circ & \succ & \texttt{N 1} \\
\mapsto_c & \texttt{LEq } \square \texttt{ (N 0)} \triangleright \texttt{If }\quad \triangleright & & \texttt{1} \\
\mapsto_c & \texttt{L} & & \\
\mapsto_c & \texttt{L} & & \\
\mapsto_c & \texttt{If} & & \\
\mapsto_c & & & \\
\mapsto_c & \texttt{Ap } \square \texttt{ (Sub } \cdots \texttt{)} \triangleright \circ & \succ & \texttt{(Fun } \cdots \texttt{)} \\
\mapsto_c & \texttt{Ap } \square \texttt{ (Sub } \cdots \texttt{)} \triangleright \circ & \prec & \langle\langle \cdots \rangle\rangle \\
\mapsto_c & \texttt{Ap } \langle\langle \cdots \rangle\rangle\ \square \triangleright \circ & \succ & \texttt{Sub (N 1) (N 2)} \\
\mapsto_c & \texttt{Sub } \square \texttt{ (N 2)} \triangleright \texttt{Ap } \cdots \triangleright \circ & \succ & \texttt{N 1} \\
\mapsto_c & \texttt{Sub } \square \texttt{ (N 2)} \triangleright \texttt{Ap } \cdots \triangleright \circ & \prec & \texttt{1} \\
\mapsto_c & \texttt{Sub 1 } \square \triangleright \texttt{Ap } \cdots \triangleright \circ & \succ & \texttt{N 2} \\
\mapsto_c & \texttt{Sub 1 } \square \triangleright \texttt{Ap } \cdots \triangleright \circ & \prec & \texttt{2} \\
\mapsto_c & \texttt{Ap } \langle\langle \cdots \rangle\rangle\ \square \triangleright \circ & \prec & \texttt{-1} \\
\mapsto_c & \circ & \succ & \texttt{If (LEq (N -1) (N 0)) (Eq (N -1) (N 0)) (Ap (Fun } \cdots \texttt{) (Sub (N -1) (N 2)))} \\
\mapsto_c & \texttt{If } \square\ \cdots \triangleright \circ & \succ & \texttt{LEq (N -1) (N 0)} \\
\mapsto_c & \texttt{LEq } \square \texttt{ (N 0)} \triangleright \texttt{If } \cdots \triangleright \circ & \succ & \texttt{N -1} \\
\mapsto_c & \texttt{LEq } \square \texttt{ (N 0)} \triangleright \texttt{If } \cdots \triangleright \circ & \prec & \texttt{-1} \\
\mapsto_c & \texttt{LEq -1 } \square \triangleright \texttt{If } \cdots \triangleright \circ & \succ & \texttt{N 0} \\
\mapsto_c & \texttt{LEq -1 } \square \triangleright \texttt{If } \cdots \triangleright \circ & \prec & \texttt{0} \\
\mapsto_c & \texttt{If } \square \texttt{ (Eq (N -1) (N 0))}\ \cdots \triangleright \circ & \prec & \texttt{True} \\
\mapsto_c & \circ & \succ & \texttt{Eq (N -1) (N 0)} \\
\mapsto_c & \texttt{Eq } \square \texttt{ (N 0)} \triangleright \circ & \succ & \texttt{N -1} \\
\mapsto_c & \texttt{Eq } \square \texttt{ (N 0)} \triangleright \circ & \prec & \texttt{-1} \\
\mapsto_c & \texttt{Eq -1 } \square \triangleright \circ & \succ & \texttt{N 0} \\
\mapsto_c & \texttt{Eq -1 } \square \triangleright \circ & \prec & \texttt{0} \\
\mapsto_c & \circ & \prec & \texttt{False}
\end{array}
$$

Wow, that long just to compute if three is even! No wonder we prefer evaluation semantics! Computers, however, certainly would prefer the *C Machine* - note that every state transition for the *C Machine* is an axiom. This means we can implement it as a single tight `while` loop that moves from state to state until it reaches a state in F.

Note: In an exam situation, you may be asked to present a derivation like the above. It is not necessary to write out every single step, just those steps you believe to be most important.

3 The *E Machine*

Now that we've made control flow more explicit, it becomes easier to see how we would implement the language efficiently on a real computer.

Let's take a look at our primitive machine operations so far:

- **The Numeric Operators** - `+`, `*` etc.
- **Comparison Operators** - `==`, `<` etc.
- **Logical Operators** - `&&`, `||`, `!`
- **Substitution** - $e[x := y]$

The great thing about most of thes computers, so they can be imple

The one operation that tion is of complexity $O(n)$ in the s an efficient machine instruction!

So, we are going to extend our machine once more, to include *environments* in the machine state. We will call our new machine the *E Machine*. This way, we make the machine look up variables in the environment rather than rely on substitution.

We extend our states as follows:

$$\frac{s\ \mathit{Stack} \quad \Gamma\ \mathit{Env} \quad e\ \mathit{Expr}}{} \qquad \frac{s\ \mathit{Stack} \quad \Gamma\ \mathit{Env} \quad e\ \mathit{Expr}}{}$$

Note this is exactly the sa ed. How do we denote environmen

$$\frac{}{\bullet\ \mathit{Env}} \qquad \frac{}{x =}$$

So, $\bullet$ is the empty environment, and we add new bindings to the envi $x = v$.

Now we need to add some rules for dealing with variables. In the *hine*, a variable occurring by itself was a stuck state - the only way for such a situation to arise is if the variable is free, which makes it an invalid expression. In the *M Machine*, variables are to be expected - *if* they occur in the environment:

$$\frac{}{s \mid x = v; \Gamma \succ x \mapsto_E s \mid x = v; \Gamma \prec v}$$

Now, what happens when we call a function? Naturally, we'd want to introduce new bindings to our environment, for the argument and the recursive name. When the function returns, however, we want to *remove* these bindings, as they are no longer in scope. The way we achieve this is somewhat unusual. We extend our stack to be able to include *environments* as well as frames:

$$\frac{}{\circ\ \mathit{Stack}} \qquad \frac{s\ \mathit{Stack} \quad x\ \mathit{Frame}}{x \rhd s\ \mathit{Stack}} \qquad \frac{s\ \mathit{Stack} \quad \Gamma\ \mathit{Env}}{\Gamma \rhd s\ \mathit{Stack}}$$

Then, when we apply a function, we add the bindings, and push the *old* environment to the stack:

$$\frac{}{\texttt{Apply}\ \langle\!\langle f.x.e \rangle\!\rangle\ \square \rhd s \mid \Gamma \prec v \mapsto_E \Gamma \rhd s \mid x = v; f = \langle\!\langle f.x.e \rangle\!\rangle; \Gamma \succ e}$$

When the function returns, we *restore* the environment from the stack, which has the effect of removing the unwanted bindings:

$$\frac{}{\Gamma \rhd s \mid \Delta \prec v \mapsto_E s \mid \Gamma \prec v}$$

All other state transition rules are identical to the *C Machine*, preserving the environment across the transition.

The initial and final states are also unchanged, except that they now include the empty environment:

$$\frac{e\ \mathit{Expr}}{\circ \mid \bullet \succ e \in I} \qquad \frac{v\ \mathit{Value}}{\circ \mid \bullet \prec v \in F}$$

3.1 Closures

3.1.1 The Problem

Suppose we had a function that depended on variables that are free inside the function body, but in scope where the function is d
functions - like in the following (s

$$\begin{array}{lrcl}
 & \circ \mid \bullet & \succ & \texttt{App}\ \ \texttt{App}\ \ \texttt{Fun}\quad \texttt{Fun}\ g.y.x))\ 3)\ 4 \\
\mapsto_E & \texttt{App}\ \square\ \ 4 \rhd \circ \mid \bullet & \succ & \texttt{App}\ (\texttt{Fun}\ f.x.(\texttt{Fun}\ g.y.x))\ 3 \\
\mapsto_E & \texttt{App}\ \square\ \ 3 \rhd \texttt{App}\ \square\ \ 4 \rhd \circ \mid \bullet & \succ & \texttt{Fun}\ f.x.(\texttt{Fun}\ g.y.x) \\
 & & \vdots & \\
\mapsto_E & \texttt{App}\ \langle\!\langle f.x.(\texttt{Fun}\ g.y.x)\rangle\!\rangle\ \square \rhd \texttt{App}\ \square\ \ 4 \rhd \circ \mid \bullet & \prec & 3 \\
\mapsto_E & \bullet \rhd \texttt{App}\quad 4 \rhd \circ \mid x = 3; f = \langle\!\langle \cdots \rangle\!\rangle; \bullet & \succ & \texttt{Fun}\ g.y.x \\
\mapsto_E & \bullet \rhd \texttt{App}\quad 4 \rhd \circ \mid x = 3; f = \langle\!\langle \cdots \rangle\!\rangle; \bullet & \prec & \langle\!\langle g.y.x \rangle\!\rangle \\
\mapsto_E & & & \\
 & & & \\
\mapsto_E & & & \\
\mapsto_E & \bullet \rhd \circ \mid y = 4; g = \langle\!\langle g.y.x \rangle\!\rangle; \bullet & \succ & x
\end{array}$$

The problem with these situations is now apparent - we can't evaluat s x is not in our environment! But, where the function g is defined, x oblem here is that the function value for g *escaped* the scope in which it was defined.

3.1.2 The Solution

To fix this problem, we must revise our definition of a function value. Instead of essentially being an expression with some bound variables, we will instead make it a *pair* of the expression for the body of the function, and the *environment* in which it is defined, i.e:

$$\frac{}{s \mid \Gamma \succ \texttt{Fun}\ (f.x.e) \ \mapsto_E\ s \mid \Gamma \prec \langle\!\langle \Gamma, f.x.e \rangle\!\rangle}$$

This pairing of a function body and an environment is called a *closure*.

When we apply a closure with an argument, it's very similar to our rules before, except instead of simply augmenting the *current* environment with the argument values and recursive name, we first set the current environment to be contents of the closure:

$$\frac{}{\texttt{App}\ \langle\!\langle \Delta, f.x.e \rangle\!\rangle\ \square \rhd s \mid \Gamma \prec v \ \mapsto_E\ \Gamma \rhd s \mid x = v; f = \langle\!\langle \Delta, f.x.e \rangle\!\rangle; \Delta \succ e}$$

With this environment capture in place, we can now evaluate the example above successfully:

$$
\begin{array}{lrcl}
 & \circ \mid \bullet & \succ & \texttt{App}\ (\texttt{App}\ (\texttt{Fun}\ f.x.(\texttt{Fun}\ g.y.x))\ 3)\ 4 \\
\mapsto_E & \texttt{App}\ \ \square\ \ 4 \rhd \circ \mid \bullet & \succ & \texttt{App}\ (\texttt{Fun}\ f.x.(\texttt{Fun}\ g.y.x))\ 3 \\
\mapsto_E & \texttt{App}\ \ \square\ \ 3 \rhd \texttt{App}\ \ \square\ \ 4 \rhd \circ \mid \bullet & \succ & \texttt{Fun}\ f.x.(\texttt{Fun}\ g.y.x) \\
 & & \vdots & \\
\mapsto_E & \texttt{App}\ \langle\!\langle \bullet, f.x.(\texttt{Fun}\ g.y.x) \rangle\!\rangle\ \square \rhd \texttt{App}\ \ \square\ \ 4 \rhd \circ \mid \bullet & \prec & 3 \\
\mapsto_E & \bullet \rhd \texttt{App}\ \ \square\ \ 4 \rhd \circ \mid x = 3; f = \langle\!\langle \cdots \rangle\!\rangle; \bullet & \succ & \texttt{Fun}\ g.y.x \\
\mapsto_E & \bullet \rhd \texttt{App}\ \ \square\ \ 4 \rhd \circ \mid x = 3; f = \langle\!\langle \cdots \rangle\!\rangle; \bullet & \prec & \langle\!\langle x = 3; f; \bullet, g.y.x \rangle\!\rangle \\
\mapsto_E & \texttt{App}\ \ \square\ \ 4 \rhd \circ \mid \bullet & \prec & \langle\!\langle x = 3; f; \bullet, g.y.x \rangle\!\rangle \\
 & & \vdots & \\
\mapsto_E & \texttt{App}\ \langle\!\langle x = 3; f; \bullet\ g.y.x \rangle\!\rangle \quad \rhd \circ \mid \bullet & \prec & 4 \\
\mapsto_E & \bullet \rhd \circ \mid y = 4; g = \langle\!\langle \cdots \rangle\!\rangle; x = 3; f = \langle\!\langle \cdots \rangle\!\rangle; \bullet & \succ & x \\
\mapsto_E & \bullet \rhd \circ \mid y = 4; & & \\
\mapsto_E & & &
\end{array}
$$