Lecture Notes on Dynamic Semantics

15-411: Compiler Design
Frank Pfenning, Rob Simmons, Jan Hoffmann

Lecture 13
October 11, 2016

1 Introduction

In the previous lecture we specified the *static semantics* of a small imperative language. In this lecture we proceed to discuss its *dynamic semantics*, that is, how programs execute. The relationship between the dynamic semantics for a language and what a compiler actually implements is usually much less direct than for the static semantics. That's because a compiler doesn't actually run the program. Instead, it translates it from some source language to a target language, and then the program in the target language is actually executed.

In our context, the purpose of writing down the dynamic semantics is primarily to precisely specify how programs are supposed to execute. Previously, we defined how programs were supposed to execute either informally ("use left-to-right evaluation") or by explicitly specifying the way an elaborated AST is compiled into a lower-level language where effects are explicitly ordered. As our language becomes more complex, it will become increasingly important to make our language specification formal in order to avoid the ambiguity inherent in informal descriptions.

We *could* continue to define our dynamic semantics by compiling the language into a simpler form. (In fact, we are doing this to some degree, because we are defining our dynamic semantics in terms of the elaborated AST, which is a simplification of the concrete syntax. There is a trade-off here: the more work we do before defining the dynamic semantics, the harder it is too see the relationship between the original program and the program whose behavior we are defining.) Writing this down the dynamic semantics in terms of the source language, or something close to it, has a number of advantages. The programmer is saved from needing to understand the behavior of the compiler in order to understand the specification of the dynamic semantics. Conversely, the compiler writer is able to implement the

dynamic semantics in any way that conforms to the specification, rather than being tied to a specific implementation strategy they have specified.

Defining both our formal dynamic semantics and static semantics over the same elaborated AST also facilitates mathematically proving properties of the programming language. Much of the theory of programming languages is concerned with just that and therefore requires an dynamics semantics. Furthermore, if we define an operational specification of our language both before and after compilation, we can consider proving that they always compute the same result. These topics, however, are outside the scope of this course.

2 Evaluating Expressions

When trying to specify the operational semantics of a programming language, there are a bewildering array of choices regarding the *style* of presentation. Some choices are natural semantics, structural operational semantics, abstract machines, substructural operational semantics, and many more. We use the mechanism of *abstract machines*, despite some of its shortcomings.

In an *abstract machine semantics*, which is a form of so-called *small-step operational semantics*, we step through the evaluation of an expression e until we have reached a value v. So the basic judgment might be written $e \longrightarrow e'$. However, this is much too simplistic. For example, it does not represent the call stack, or the current value of the variables that are recorded in an *environment*, or what to do with the eventual value. We will introduce such semantic artifacts one by one, as they are needed.

Consider the expression $e_1 + e_2$. By the left-to-right evaluation rule, we first have to evaluate e_1 and then e_2. So why we evaluate e_1 we have to "remember" that we still have to evaluate e_2 then sum up the value. The information on what we still have to do is collected in a so-called *continuation* K. We write the judgment as

$$e \rhd K$$

which we read as "*evaluate expression* e *and pass the result to the continuation* K". In the continuation there is a "hole" (written as an underscore character _) in which we plug in the value passed to it. So:

$$e_1 + e_2 \rhd K \qquad \longrightarrow \qquad e_1 \rhd (\_ + e_2\ , K)$$

When e_1 has been reduced to a value c_1, we plug it into the hole and evaluate e_2 next;

$$c_1 \rhd (\_ + e_2\ , K) \qquad \longrightarrow \qquad e_2 \rhd (c_1 + \_\ , K)$$

Finally, when e_2 has been reduced to a value c_2 we perform the actual addition and pass the result to K.

$$c_2 \rhd (c_1 + \_\ , K) \qquad \longrightarrow \qquad c \rhd K \qquad (c = c_1 + c_2 \bmod 2^{32})$$

In the last rule we appeal to the mathematical operation of addition modulo 2^{32} on two given integers modulo 2^{32}. Since we describe C0, we assume that constants are 32-bit words in two's complement representation. All other binary modular arithmetic operations $\oplus$ are handled in a similar way, so we summarize them as

$$\begin{array}{lcll} e_1 \oplus e_2 \rhd K & \longrightarrow & e_1 \rhd (\_ \oplus e_2\ , K) & \\ c_1 \rhd (\_ \oplus e_2\ , K) & \longrightarrow & e_2 \rhd (c_1 \oplus \_\ , K) & \\ c_2 \rhd (c_1 \oplus \_\ , K) & \longrightarrow & c \rhd K & (c = c_1 \oplus c_2 \bmod 2^{32}) \end{array}$$

For an effectful operation such as division, the last step could also raise an arithmetic exception arith (`SIGFPE` on the x86 architecture family, numbered 8). How do we represent that? We would like to abort the computation entirely and go to a state where the final outcome is reported as an arithmetic exception. We describe this as follows:

$$\begin{array}{lcll} e_1 \oslash e_2 \rhd K & \longrightarrow & e_1 \rhd (\_ \oslash e_2\ , K) & \\ c_1 \rhd (\_ \oslash e_2\ , K) & \longrightarrow & e_2 \rhd (c_1 \oslash \_\ , K) & \\ c_2 \rhd (c_1 \oslash \_\ , K) & \longrightarrow & c \rhd K & (c = c_1 \oslash c_2) \\ c_2 \rhd (c_1 \oslash \_\ , K) & \longrightarrow & \mathsf{exception}(\mathsf{arith}) & (c_1 \oslash c_2 \text{ undefined}) \end{array}$$

Here, some care must be taken to define the value $c_1 \oslash c_2$ correctly in the cases of division and modulus, and the conditions under which the result is mathematically "undefined" (like division by zero) and therefore must raise an exception. We have specified this in previous lectures and assignments, so we won't detail the conditions here.

What happens when evaluation finishes normally? In the case of the empty continuation we stop the abstract machine and return $\mathsf{value}(c)$

$$c \rhd \cdot \quad \longrightarrow \quad \mathsf{value}(c)$$

Boolean expression and short-circuiting expressions work similarly.

$$\begin{array}{lcl} e_1 \texttt{ \&\& } e_2 \rhd K & \longrightarrow & e_1 \rhd (\_ \texttt{ \&\& } e_2\ , K) \\ \mathsf{false} \rhd (\_ \texttt{ \&\& } e_2\ , K) & \longrightarrow & \mathsf{false} \rhd K \\ \mathsf{true} \rhd (\_ \texttt{ \&\& } e_2\ , K) & \longrightarrow & e_2 \rhd K \end{array}$$

Notice how e_2 is ignored in case false is returned to the continuation $(\_ \texttt{ \&\& } e_2\ , K)$, which encodes the short-circuiting behavior of the conjunction. Also note that we now have two kinds of values: boolean values false and true as well as integer values c. We could have also defined boolean expressions as evaluating to integers with a rule like $\mathsf{true} \rhd K \longrightarrow 1 \rhd K$. By making true a value, we explicitly capture the fact that we expect integers and boolean expressions to not be confused at runtime.

Example Consider the expression $((4 + 5) * 10) + 2$. Using our evaluation rules, we obtion the following evaluation.

$$
\begin{array}{lll}
 & ((4+5)*10)+2 & \rhd \ \cdot \\
\longrightarrow & (4+5)*10 & \rhd \ \_ + 2 \\
\longrightarrow & (4+5)*10 & \rhd \ \_ + 2 \\
\longrightarrow & 4+5 & \rhd \ \_ * 10 \ , \ \_ + 2 \\
\longrightarrow & 4 & \rhd \ \_ + 5 \ , \ \_ * 10 \ , \ \_ + 2 \\
\longrightarrow & 5 & \rhd \ 4 + \_ \ , \ \_ * 10 \ , \ \_ + 2 \\
\longrightarrow & 9 & \rhd \ \_ * 10 \ , \ \_ + 2 \\
\longrightarrow & 10 & \rhd \ 9 * \_ \ , \ \_ + 2 \\
\longrightarrow & 90 & \rhd \ \_ + 2 \\
\longrightarrow & 2 & \rhd \ 90 + \_ \\
\longrightarrow & 92 & \rhd \ \cdot
\end{array}
$$

3 Variables

We can continue along this line, be we get stuck for variables. Where do their values come from? We need to add an *environment* η that maps variables to their values. We write

$$\eta ::= \cdot \mid \eta, x \mapsto v$$

and $\eta[x \mapsto v]$ for either adding $x \mapsto v$ to η or overwriting the current value of x by v (if $\eta(x)$ is already defined). The state of the abstract machine now contains the environment η. We separate by a turnstile ($\vdash$) from the expression to evaluate and its continuation.

$$\eta \vdash e \rhd K$$

The rules so far just carry this along. For example:

$$
\begin{array}{lll}
\eta \vdash e_1 \oplus e_2 \rhd K & \longrightarrow & \eta \vdash e_1 \rhd (\_ \oplus e_2 \ , \ K) \\
\eta \vdash c_1 \rhd (\_ \oplus e_2 \ , \ K) & \longrightarrow & \eta \vdash e_2 \rhd (c_1 \oplus \_ \ , \ K) \\
\eta \vdash c_2 \rhd (c_1 \oplus \_ \ , \ K) & \longrightarrow & \eta \vdash c \rhd K \qquad (c = c_1 \oplus c_2 \bmod 2^{32})
\end{array}
$$

Variables are just looked up in the environment.

$$\eta \vdash x \rhd K \qquad \longrightarrow \qquad \eta \vdash \eta(x) \rhd K$$

Because we are interested in evaluating only expression that have already passed all static semantic checks of the language, we know that $\eta(x)$ will be defined (all variables must be initialized before they are used).

4 Executing Statements

Executing statements in L3, the fragment of C0 we have considered so far, can either complete normally, return from the current function with a return statement, or raise an exception. The "normal" execution of a statement does not pass a value to its continuation; instead it has an effect on its environment by assigning to its existing variables or declaring new ones. We write this as

$$\eta \vdash s \blacktriangleright K$$

where the continuation K should start with a statement. Statements don't return values, so every statement that terminates eventually becomes a nop. For example:

$$\begin{array}{lcl} \eta \vdash \mathsf{seq}(s_1, s_2) \blacktriangleright K & \longrightarrow & \eta \vdash s_1 \blacktriangleright (s_2\ , K) \\ \eta \vdash \mathsf{nop} \blacktriangleright (s\ , K) & \longrightarrow & \eta \vdash s \blacktriangleright K \end{array}$$

The last line codifies that if there no further statement to execution, we grab the first statement from the continuation. When executing an assignment we first have to evaluate the assignment, then change the variable value in the environment.

$$\begin{array}{lcl} \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K & \longrightarrow & \eta \vdash e \rhd (\mathsf{assign}(x, \_)\ , K) \\ \eta \vdash v \rhd (\mathsf{assign}(x, \_)\ , K) & \longrightarrow & \eta[x \mapsto v] \vdash \mathsf{nop} \blacktriangleright K \end{array}$$

Conditionals follow the pattern of the short-circuiting conjunction.

$$\begin{array}{lcl} \eta \vdash \mathsf{if}(e, s_1, s_2) \blacktriangleright K & \longrightarrow & \eta \vdash e \rhd (\mathsf{if}(\_, s_1, s_2)\ , K) \\ \eta \vdash \mathsf{true} \rhd (\mathsf{if}(\_, s_1, s_2), K) & \longrightarrow & \eta \vdash s_1 \blacktriangleright K \\ \eta \vdash \mathsf{false} \rhd (\mathsf{if}(\_, s_1, s_2), K) & \longrightarrow & \eta \vdash s_2 \blacktriangleright K \end{array}$$

While loops are a bit more complicated. We take a slight shortcut by using the identity

$$\mathsf{while}(e, s) \equiv \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop})$$

to avoid writing out several rules implementing the right-hand side of this identity directly.

$$\eta \vdash \mathsf{while}(e, s) \blacktriangleright K \quad \longrightarrow \quad \eta \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop}) \blacktriangleright K$$

Loops bring up the question of non-termination, which is modeled naturally: we just have abstract machine transitions $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots$ without ever arriving at a final state. The final states are just $\mathsf{nop} \blacktriangleright \cdot$ and $\mathsf{exception}(E)$, where E can currently be either arith or the abort exception caused by a failing assert statement.

$$\begin{array}{lcl} \eta \vdash \mathsf{assert}(e) \blacktriangleright K & \longrightarrow & \eta \vdash e \rhd (\mathsf{assert}(\_), K) \\ \eta \vdash \mathsf{true} \rhd (\mathsf{assert}(\_), K) & \longrightarrow & \eta \vdash \mathsf{nop} \blacktriangleright K \\ \eta \vdash \mathsf{false} \rhd (\mathsf{assert}(\_), K) & \longrightarrow & \mathsf{exception}(\mathsf{abort}) \end{array}$$

Declarations are also pretty straightforward, since they just add a new variable with an undefined or useless value to the environment.

$$\eta \vdash \mathsf{decl}(x, \tau, s) \blacktriangleright K \qquad \longrightarrow \qquad \eta[x \mapsto \mathsf{nothing}] \vdash s \blacktriangleright K$$

In a language that permits shadowing of variables, we would have to save the current value of x and restore it after s has finished executing. Or we would think of η as a list where $\eta(x)$ refers to the value of the rightmost occurrence, which would be removed when we leave the scope of declaration. Or we could statically rename the variables during elaboration so that no shadowing can occur during execution.

At this point we have handled all salient statements except return statements that are tied to function calls. We discuss them in the next section.

Example Consider the statement $\mathsf{while}(x > 0, \mathsf{assign}(x, x + 1))$ and $\eta = [x{\mapsto}1]$. Using rules for statement execution, we obtion the following execution; where $s \equiv x = x + 1$.

$$\begin{array}{lll}
 & [x{\mapsto}1] \vdash \mathsf{while}(x > 0, s) & \blacktriangleright \ \cdot \\
\longrightarrow & [x{\mapsto}1] \vdash \mathsf{if}(x{>}0, \mathsf{seq}(s, \mathsf{while}(x{>}0, s)), \mathsf{nop}) & \blacktriangleright \ \cdot \\
\longrightarrow & [x{\mapsto}1] \vdash x > 0 & \triangleright \ \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\
\longrightarrow & [x{\mapsto}1] \vdash x & \triangleright \ _ > 0; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\
\longrightarrow & [x{\mapsto}1] \vdash 1 & \triangleright \ _ > 0; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\
\longrightarrow & [x{\mapsto}1] \vdash 0 & \triangleright \ 1 > _; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\
\longrightarrow & [x{\mapsto}1] \vdash \mathsf{true} & \triangleright \ \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\
\longrightarrow & [x{\mapsto}1] \vdash \mathsf{seq}(s, \mathsf{while}(x > 0, s)) & \blacktriangleright \ \cdot \\
\longrightarrow & [x{\mapsto}1] \vdash \mathsf{assign}(x, x + 1)) & \blacktriangleright \ \mathsf{while}(x > 0, \mathsf{assign}(x, x + 1)) \\
\longrightarrow & [x{\mapsto}1] \vdash x + 1 & \triangleright \ \mathsf{assign}(x, _)); \mathsf{while}(x > 0, s) \\
\longrightarrow & [x{\mapsto}1] \vdash x & \triangleright \ _ + 1; \mathsf{assign}(x, _)); \mathsf{while}(x > 0, s) \\
\longrightarrow & [x{\mapsto}1] \vdash 1 & \triangleright \ _ + 1; \mathsf{assign}(x, _)); \mathsf{while}(x > 0, s) \\
\longrightarrow & [x{\mapsto}1] \vdash 1 & \triangleright \ 1 + _; \mathsf{assign}(x, _)); \mathsf{while}(x > 0, s) \\
\longrightarrow & [x{\mapsto}1] \vdash 2 & \triangleright \ \mathsf{assign}(x, _)); \mathsf{while}(x > 0, s) \\
\longrightarrow & [x{\mapsto}2] \vdash \mathsf{nop} & \blacktriangleright \ \mathsf{while}(x > 0, s) \\
\longrightarrow & [x{\mapsto}2] \vdash \mathsf{while}(x > 0, s) & \blacktriangleright \ \cdot \\
\ldots & &
\end{array}$$

5 Function Calls

A function call first has to evaluate the function arguments, from left to right. Then we invoke the function, whose body starts to execute in an environment that maps that formal parameters of the function to the argument values. But meanwhile we have to save the current environment of the caller somewhere. Similarly, we also have to save the continuation of the caller, so that when the callee returns we pass it the return value. So a stack frame $\langle \eta, K \rangle$ consists of an environment η and a

continuation K.

$$\text{Call stack} \quad S \quad ::= \quad \cdot \mid S \,,\, \langle \eta, K \rangle$$

Now states representing evaluation of expression and execution of statements have the forms

$$\begin{array}{ll} \text{Evaluation} & S \,;\, \eta \vdash e \rhd K \\ \text{Execution} & S \,;\, \eta \vdash s \blacktriangleright K \end{array}$$

We only show the special case of evaluation function calls with two and zero arguments. This gets the point across while avoiding nitpicky details of working with multiple-argument functions.

$$\begin{array}{lcll} S \,;\, \eta \vdash f(e_1, e_2) \rhd K & \longrightarrow & S \,;\, \eta \vdash e_1 \rhd (f(\_, e_2) \,,\, K) & \\ S \,;\, \eta \vdash c_1 \rhd (f(\_, e_2) \,,\, K) & \longrightarrow & S \,;\, \eta \vdash e_2 \rhd (f(c_1, \_) \,,\, K) & \\ S \,;\, \eta \vdash c_2 \rhd (f(c_1, \_) \,,\, K) & \longrightarrow & (S \,,\, \langle \eta, K \rangle) \,;\, [x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright \cdot & \\ & & \quad \textit{(given that } f \textit{ is defined as } \; f(x_1, x_2)\{s\}) & \\ S \,;\, \eta \vdash f(\,) \rhd K & \longrightarrow & (S \,,\, \langle \eta, K \rangle) \,;\, \cdot \vdash s \blacktriangleright \cdot & \\ & & \quad \textit{(given that } f \textit{ is defined as } \; f(\,)\{s\}) & \end{array}$$

In the next-to-last rule we see a new environment with values for x_1 and x_2 and a new stack frame save the caller's environment η and continuation K.

When executing a return statement we simply have to restore the caller's environment and continuation from the stack and pass the return value to the caller's continuation.

$$\begin{array}{lcl} S \,;\, \eta \vdash \mathsf{return}(e) \blacktriangleright K & \longrightarrow & S \,;\, \eta \vdash e \rhd (\mathsf{return}(\_) \,,\, K) \\ S \,,\, \langle \eta', K' \rangle \,;\, \eta \vdash v \rhd (\mathsf{return}(\_) \,,\, K) & \longrightarrow & S \,;\, \eta' \vdash v \rhd K' \end{array}$$

In order to support functions returning void, we can use a useless value, nothing, and elaborate "`return;`" as return(nothing). We also assume that elaboration adds an additional return(nothing) statement to the end of every `void` function. As an alternative, we could explicitly add a rule for function calls that finish without a `return` statement:

$$S \,,\, \langle \eta', K' \rangle \,;\, \eta \vdash \mathsf{nop} \blacktriangleright \cdot \quad \longrightarrow \quad S \,;\, \eta' \vdash \mathsf{nothing} \rhd K'$$

6 Statics, Dynamics, and Safety

We start the machine initially in a state where we call the main function, and we stop the abstract machine if we reach this continuation.

$$\begin{array}{lcll} \cdot \,;\, \cdot \vdash \mathsf{main}(\,) \rhd \cdot & & & \text{(initial state)} \\ \cdot \,;\, \eta \vdash c \rhd \cdot & \longrightarrow & \mathsf{value}(c) & \text{(final state)} \end{array}$$

which will eventually step to value(c), where c is returned by the main function. We have defined four kinds of *machine state* $\mathcal{ST}$:

- $S\ ;\ \eta \vdash e \rhd K$ – Evaluating the expression e with the continuation K
- $S\ ;\ \eta \vdash s \blacktriangleright K$ – Evaluating the statement s with the continuation K
- $\mathsf{value}(c)$ – Final state, return a value
- $\mathsf{exception}(E)$ – Final state, report an error

We've also written a bunch of transition rules $\mathcal{ST} \longrightarrow \mathcal{ST}'$. Can we say anything about whether those rules are reasonable or not?

Obviously we don't expect any final state to be on the left-hand side of a transition. On that basis we could say that any rule like $\mathsf{value}(3) \longrightarrow \mathsf{value}(4)$ is an unreasonable rule. We might also care that the language is deterministic: that every machine state $\mathcal{ST}$ has at most one state $\mathcal{ST}'$ such that $\mathcal{ST} \longrightarrow \mathcal{ST}'$.

In the other direction, there are many examples of non-final states that have *no* transition: $S; \eta \vdash 42 \rhd (\mathsf{if}(\_, s_1, s_2); K)$ is one example, and $\cdot;\cdot \vdash \mathsf{nop} \blacktriangleright \cdot$ is another. Such states are called *stuck* – it is literally undefined what the program should do in such a state. The central relationship between the static semantics and dynamic semantics is that any program that passes the static semantics should be *free of undefined behavior*, that is, free of stuck states. This can be expressed mathematically:

Theorem 1 (No undefined behavior) *If a program passes all the static semantics, and*

$$\cdot;\cdot \vdash \mathsf{main}(\,) \longrightarrow \mathcal{ST}_1 \longrightarrow \ldots \longrightarrow \mathcal{ST}_n$$

then either $\mathcal{ST}_n$ is a final state or else $\mathcal{ST}_n$ is not-stuck because there exists a state $\mathcal{ST}'$ such that $\mathcal{ST}_n \longrightarrow \mathcal{ST}'$.

In a course like 15-312, we would learn how to prove these sorts of theorems, but just stating the theorem is still useful as a specification. If we can find a *counterexample*, a program that passes the static semantics and yet gets stuck in a non-final state according to the dynamic semantics, then we know that we need to change either or static or dynamic semantics.

7 Summary

We use $\odot$ to stand for either a pure operation $\oplus$, or a potentially effectful operation $\oslash$ as well as shift, comparison, and bitwise operators.

Expressions	e	::=	$c \mid e_1 \odot e_2 \mid \mathsf{true} \mid \mathsf{false} \mid e_1 \;\&\&\; e_2 \mid x \mid f(e_1, e_2) \mid f(\,)$
Statements	s	::=	$\mathsf{nop} \mid \mathsf{seq}(s_1, s_2) \mid \mathsf{assign}(x, e) \mid \mathsf{decl}(x, \tau, s)$
		$\mid$	$\mathsf{if}(e, s_1, s_2) \mid \mathsf{while}(e, s) \mid \mathsf{return}(e) \mid \mathsf{assert}(e)$
Values	v	::=	$c \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{nothing}$
Environments	η	::=	$\cdot \mid \eta, x \mapsto c$
Stacks	S	::=	$\cdot \mid S \;,\; \langle \eta, K \rangle$
Cont. frames	ϕ	::=	$\_ \odot e \mid c \odot \_ \mid \_ \;\&\&\; e \mid f(\_, e) \mid f(c, \_)$
		$\mid$	$s \mid \mathsf{assign}(x, \_) \mid \mathsf{if}(\_, s_1, s_2) \mid \mathsf{return}(\_) \mid \mathsf{assert}(\_)$
Continuations	K	::=	$\cdot \mid \phi \;,\; K$
Exceptions	E	::=	$\mathsf{arith} \mid \mathsf{abort} \mid \mathsf{mem}$

$$
\begin{array}{lcl}
S\,;\,\eta \vdash e_1 \odot e_2 \rhd K & \longrightarrow & S\,;\,\eta \vdash e_1 \rhd (\_ \odot e_2\ ,\ K) \\
S\,;\,\eta \vdash c_1 \rhd (\_ \odot e_2\ ,\ K) & \longrightarrow & S\,;\,\eta \vdash e_2 \rhd (c_1 \odot \_\ ,\ K) \\
S\,;\,\eta \vdash c_2 \rhd (c_1 \odot \_\ ,\ K) & \longrightarrow & S\,;\,\eta \vdash c \rhd K \qquad (c = c_1 \odot c_2) \\
S\,;\,\eta \vdash c_2 \rhd (c_1 \odot \_\ ,\ K) & \longrightarrow & \mathsf{exception}(\mathsf{arith}) \qquad (c_1 \odot c_2 \text{ undefined})
\end{array}
$$

$$
\begin{array}{lcl}
S\,;\,\eta \vdash e_1\ \texttt{\&\&}\ e_2 \rhd K & \longrightarrow & S\,;\,\eta \vdash e_1 \rhd (\_\ \texttt{\&\&}\ e_2\ ,\ K) \\
S\,;\,\eta \vdash \mathsf{false} \rhd (\_\ \texttt{\&\&}\ e_2\ ,\ K) & \longrightarrow & S\,;\,\eta \vdash \mathsf{false} \rhd K \\
S\,;\,\eta \vdash \mathsf{true} \rhd (\_\ \texttt{\&\&}\ e_2\ ,\ K) & \longrightarrow & S\,;\,\eta \vdash e_2 \rhd K
\end{array}
$$

$$
S\,;\,\eta \vdash x \rhd K \quad \longrightarrow \quad S\,;\,\eta \vdash \eta(x) \rhd K
$$

$$
\begin{array}{lcl}
S\,;\,\eta \vdash \mathsf{nop} \blacktriangleright (s\ ,\ K) & \longrightarrow & S\,;\,\eta \vdash s \blacktriangleright K \\
S\,;\,\eta \vdash \mathsf{assign}(x, e) \blacktriangleright K & \longrightarrow & S\,;\,\eta \vdash e \rhd (\mathsf{assign}(x, \_)\ ,\ K) \\
S\,;\,\eta \vdash c \rhd (\mathsf{assign}(x, \_)\ ,\ K) & \longrightarrow & S\,;\,\eta[x \mapsto c] \vdash \mathsf{nop} \blacktriangleright K
\end{array}
$$

$$
S\,;\,\eta \vdash \mathsf{decl}(x, \tau, s) \blacktriangleright K \quad \longrightarrow \quad S\,;\,\eta[x \mapsto \mathsf{nothing}] \vdash s \blacktriangleright K
$$

$$
\begin{array}{lcl}
S\,;\,\eta \vdash \mathsf{assert}(e) \blacktriangleright K & \longrightarrow & S\,;\,\eta \vdash e \rhd (\mathsf{assert}(\_)\ ,\ K) \\
S\,;\,\eta \vdash \mathsf{true} \rhd (\mathsf{assert}(\_)\ ,\ K) & \longrightarrow & S\,;\,\eta \vdash \mathsf{nop} \blacktriangleright K \\
S\,;\,\eta \vdash \mathsf{false} \rhd (\mathsf{assert}(\_)\ ,\ K) & \longrightarrow & \mathsf{exception}(\mathsf{abort})
\end{array}
$$

$$
\begin{array}{lcl}
S\,;\,\eta \vdash \mathsf{if}(e, s_1, s_2) \blacktriangleright K & \longrightarrow & S\,;\,\eta \vdash e \rhd (\mathsf{if}(\_, s_1, s_2)\ ,\ K) \\
S\,;\,\eta \vdash \mathsf{true} \rhd (\mathsf{if}(\_, s_1, s_2), K) & \longrightarrow & S\,;\,\eta \vdash s_1 \blacktriangleright K \\
S\,;\,\eta \vdash \mathsf{false} \rhd (\mathsf{if}(\_, s_1, s_2), K) & \longrightarrow & S\,;\,\eta \vdash s_2 \blacktriangleright K
\end{array}
$$

$$
S\,;\,\eta \vdash \mathsf{while}(e, s) \blacktriangleright K \quad \longrightarrow \quad S\,;\,\eta \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop}) \blacktriangleright K
$$

$$
\begin{array}{lcl}
S\,;\,\eta \vdash f(e_1, e_2) \rhd K & \longrightarrow & S\,;\,\eta \vdash e_1 \rhd (f(\_, e_2)\ ,\ K) \\
S\,;\,\eta \vdash c_1 \rhd (f(\_, e_2)\ ,\ K) & \longrightarrow & S\,;\,\eta \vdash e_2 \rhd (f(c_1, \_)\ ,\ K) \\
S\,;\,\eta \vdash c_2 \rhd (f(c_1, \_)\ ,\ K) & \longrightarrow & (S\ ,\ \langle \eta, K \rangle)\,;\,[x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright \cdot \\
 & & \textit{(given that } f \textit{ is defined as } \ f(x_1, x_2)\{s\}\textit{)} \\
S\,;\,\eta \vdash f(\,) \rhd K & \longrightarrow & (S\ ,\ \langle \eta, K \rangle)\,;\,\cdot \vdash s \blacktriangleright \cdot \\
 & & \textit{(given that } f \textit{ is defined as } \ f(\,)\{s\}\textit{)}
\end{array}
$$

$$
\begin{array}{lcl}
S\,;\,\eta \vdash \mathsf{return}(e) \blacktriangleright K & \longrightarrow & S\,;\,\eta \vdash e \rhd (\mathsf{return}(\_)\ ,\ K) \\
(S\ ,\ \langle \eta', K' \rangle)\,;\,\eta \vdash v \rhd (\mathsf{return}(\_)\ ,\ K) & \longrightarrow & S\,;\,\eta' \vdash v \rhd K' \\
\cdot\,;\,\eta \vdash c \rhd (\mathsf{return}(\_)\ ,\ K) & \longrightarrow & \mathsf{value}(c)
\end{array}
$$