9. Atomic Semantics of Spec

This handout defines the semantics of the atomic part of the Spec language fairly carefully. It tries to be precise about all difficult points, but is sloppy about some things that seem obvious in order to keep the description short and readable. For the syntax and an informal account of the semantics, see the Spec reference manual, handout 4.

There are three reasons for giving a careful semantics of Spec:

1. To give a clear and unambiguous meaning for Spec programs.
2. To make it clear that there is no magic in Spec; its meaning can be given fairly easily and without any exotic methods.
3. To show the versatility of Spec by using it to define itself, which is quite different from the way we use it in the rest of the course.

This handout is divided into two parts. In the first half we describe semi-formally the essential ideas and most of the important details. Then in the second half we present the complete atomic semantics precisely, with a small amount of accompanying explanation.

Semi-formal atomic semantics of Spec

Our purpose is to make it clear that there is no arm waving in the Spec notation that we have given you. A translation of this into fancy words is that we are going to study a formal semantics of the Spec language.

Now that is a formidable sounding term, and if you take a course on the semantics of programming languages (6.821—Gifford, 6.830J—Meyer) you will learn all kinds of fancy stuff about bottom and stack domains and fixed points and things like that. You are not going to see any of that here. We are going to do a very simple minded, garden-variety semantics. We are just going to explain, very carefully and clearly, how it is that every Spec construct can be understood, as a transition of a state machine. So if you understand state machines you should be able to understand all this without any trouble.

One reason for doing this is to make sure that we really do know what we are talking about. In general, descriptions of programming languages are not in that state of grace. If you read the Pascal manual or the C manual carefully you will come away with a number of questions about exactly what happens if I do this and this, questions which the manual will not answer adequately. Two reasonably intelligent people who have studied it carefully can come to different conclusions, argue for a long time, and not be able to decide what is the right answer by reading the manual.

There is one class of mechanisms for saying what the computer should do that often does answer your questions precisely, and that is the instruction sets of computers (or, in more modern language, the architecture). These specs are usually written as state machines with fairly simple transitions, which are not beyond the power of the guy who is writing the manual to describe properly. A programming language, on the other hand, is not like that. It has much more power, generality, and wonderfulness, and also much more room for confusion.

Another reason for doing this is to show you that our methods can be applied to a different kind of system than the ones we usually study, that is, to a programming language, a notation for writing programs or a notation for writing specs. We are going to learn how to write a spec for that particular class of computer systems. This is a very different application of Spec from the last one we looked at, which was file systems. For describing a programming language, Spec is not the ideal descriptive notation. If you were in the business of giving the semantics of programming languages, you wouldn’t use Spec. There are many other notations, some of them better than Spec (although most are far worse). But Spec is good enough; it will do the job. And there is a lot to be said for just having one notation you can use over and over again, as opposed to picking up a new one each time. There are many pitfalls in devising a new notation.

Those are the two themes of this lecture. We are going to get down to the foundations of Spec, and we are going to see another, very different application of Spec, a programming language rather than a file system.

For this lecture, we will only talk about the sequential or atomic semantics of Spec, not about concurrent semantics. Consider the program:

```
thread 1:
<< x := 3 >>
<< y := 4 >>
```

In the concurrent world, it is possible to get any of the values 0, 3, or 7 for \( z \). In the sequential world, which we are in today, the only possible values are 0 and 7. It is a simpler world. We will be talking later (in handout 17 on formal concurrency) about the semantics of concurrency, which is unavoidably more complicated.

In a sequential Spec program, there are three basic constructs (corresponding to sections 5, 6, and 7 of the reference manual):

- Expressions
- Commands
- Routines

For each of these we will give a meaning function, \( ME, MC, \) and \( MR \), that takes a fragment of Spec and yields its meaning as some sort of Spec value. We shall see shortly exactly what type of values these are.

In order to describe what each of these things means, we first of all need some notion of what kind of thing the meaning of an expression or command might be. Then we have to explain in detail the exact meaning of each possible kind of expression. The basic technique we use is the standard one for a situation where you have things that are made up out of smaller things: structural induction.

The idea of structural induction is this. If you have something which is made up of an \( A \) and a \( B \), and you know the meaning of each, and have a way to put them together, you know how to get the meaning of the bigger thing.

Some ways to put things together in Spec:
STATE

What are the meanings going to be? Our basic notion is that what we are doing when writing a Spec program is describing a state machine. The central properties of a state machine are that it has states and it has transitions.

A state is a function from names to values: $\text{State}: \text{Name} \rightarrow \text{Value}$. For example:

\begin{verbatim}
VAR x: Int
y: Int
\end{verbatim}

If there are no other variables, the state simply consists of the mapping of the names "x" and "y" to their corresponding values. Initially, we don’t know what their values are. Somehow the meaning we give to this whole construct has to express that.

Next, if we write $x := 1$, after that the value of $x$ is 1. So the meaning of this had better look something like a transition that changes the state, so that no matter what the $x$ was before, it is 1 afterwards. That’s what we want this assignment to mean.

Spec is much simpler than C. In particular, it does not have “references” or “pointers”. When you are doing problems, if you feel the urge to call malloc, the correct thing to do is to make a function whose range is whatever sort of thing you want to allocate, and then choose a new element of the domain that isn’t being used already. You can use the integers or any convenient sort of name for the domain, that is, to name the values. If you define a CLASS, Spec will do this for you automatically.

So the state is just these name-to-value mappings.

NAMES

Spec has a module structure, so that names have two parts, the module name and the simple name. When referring to a variable in another module, you need both parts.

\begin{verbatim}
MODULE M
VAR x
x := 3
M.x := 3
\end{verbatim}

To simplify the semantics, we will use $M.x$ as the name everywhere. In other words, to apply the semantics you first must go through the program and replace every $x$ declared in the current module $M$ with $M.x$. This converts all references to global variables into these two part names, so that each name refers to exactly one thing. This transformation makes things simpler to describe and understand, but uglier to read. It doesn’t change the meaning of the program, which could have been written with two part names in the first place.

All the global variables have these two part names. However, local variables are not prefixed by the module name:

\begin{verbatim}
A, B
A := B
a + b
A [] B
\end{verbatim}
What about \( x + y \)? This is just shorthand for \( T.\text{+}(x, y) \), where \( T \) is the type of \( x \). Everything that is not a constant or a variable is an invocation. This should be a familiar concept for those of you who know Scheme.

**Semantics of function invocation**

What are the semantics of function invocation? Given a function \( T \rightarrow U \), the correct type of its meaning is \( (T, S) 
\rightarrow U \), since the function can read the state but not modify it. Next, how are we going to attach a meaning to an invocation \( f(x) \)? Remember the rule of structural induction. In order to explain the meaning of a complicated thing, you are supposed to build it out of the meaning of simpler things. We know the meaning of \( x \) and of \( f \). We need to come up with a map from states to values that is the meaning of \( f(x) \). That is, we get our hands on the meaning of \( f \) and the meaning of \( x \), and then put them together appropriately. What is the meaning of \( f \)? It is \( o(f) \).

\[
\begin{align*}
\text{\( f(x) \) means} & \quad \text{\( s\("f"\)} \quad \text{\( s\("x"\)} \quad \text{...} \\
\text{How are we going to put it together, remembering the type we want for \( f(x) \), which is \( S \rightarrow U? \) } & \\
\text{\( f(x) \) means} & \quad \\text{\( (\\{ s | s\("f"\) = (s\("x"), s)) \)} \\
\end{align*}
\]

Now this could be complete nonsense, for instance if \( s\("f"\) \) evaluates to an integer. If \( s\("f"\) \) isn’t a function then this doesn’t typecheck. But there is no doubt about what this means if it is legal. It means invoke the function.

That takes care of expressions, because there are no other expressions besides these. Structural induction says you work your way through all the different ways to put little things together to make big things, and when you have done them all, you are finished.

**Question:** What about undefined functions?

Then the \( (T, S) 
\rightarrow U \) mapping is partial.

**Question:** Is \( f(x) = f(x) \) if \( f(x) \) is undefined?

No, it’s undefined. But those are deep waters and I propose to stay out of them.

**Commands**

What is the type of the meaning of a command? Well, we have states and values to play with, and we have used up \( S \rightarrow V \) on expressions. What sort of thing is a command? It’s a transition from one state to another.

**Expressions:** \( S \rightarrow V \)

**Commands:** \( S \rightarrow S? \)

This is good for a subset of commands. But what about this one?

\[
\begin{align*}
\text{\( x := 1 \)} & \quad \text{\( x := 2 \)} \\
\end{align*}
\]

Is its meaning a function from states to states? No, from states to *sets* of states. It can’t just be a function. It has to be a relation. Of course, there are lots of ways to code relations as functions. The way we use is:

\[
\begin{align*}
\text{\( o = s\("x" \rightarrow ME(e)(s)) \)} \\
\end{align*}
\]
This is just a Spec function constructor, of the form \( f\{arg \rightarrow value\} \). Note that we are using the semantics of expressions that we defined in the previous section.

**Aside—an alternate encoding for commands**

As we said before, there are many ways to code the command relation. Another possibility is:

Commands: \( S \rightarrow \text{SET} \, S \)

This encoding seems to make the meanings of commands clumsier to write, though it is entirely equivalent to the one we have chosen.

There is a third approach, which has a lot of advantages: write predicates on the state values. If \( x \) and \( y \) are the state variables in the pre-state, and \( x' \) and \( y' \) the state variables in the post-state, then

\[
(x' = 1 \land y' = y)
\]

is another way of writing

\[
o = s\{"a" \rightarrow 1\}
\]

In fact, this approach is another way of writing programs. You could write everything just as predicates. (Of course, you could also write everything in the ugly \( o = s\{...\} \) form, but that would look pretty awful. The predicates don’t look so bad.)

Sometimes it’s actually nice to do this. Say you want to write the predicate that says you can have any value at all for \( x \). The Spec

\[
\text{VAR } z \mid x := z
\]

is just

\[
(y' = y)
\]

(in the simple world where the only state variables are \( x \) and \( y \)). This is much simpler than the previous, rather inscrutable, piece of program. So sometimes this predicate way of doing things can be a lot nicer, but in general it seems to be not as satisfactory, mainly because the \( y' = y \) stuff clutters things up a lot.

That was just an aside, to let you know that sometimes it’s convenient to describe the things that can go on in a spec using predicates rather than functions from state pairs to \( \text{Bool} \).

**Commands — routine invocation \( p(x) \)**

What are the semantics of routine invocation? Well, it has to do something with \( s \). The idea is that \( p \) is an abstraction of a state transition, so its meaning will be a relation of type \( \text{ATr} \). What about the argument \( x \)? There are many ways to deal with it. Our way is to use another pseudo-variable \( s_a \) to pass the argument and get back the result.

The meaning of \( p(e) \) is going to be

\[
\{ s \, | \, ME(p)(s) \{ \text{"$a" \rightarrow ME(e)(s)\} \} \}
\]

This says, first get \( ME(p) \), the meaning of \( p \). This is no longer a transition but a function from argument values to transitions, because the idea is that for every possible argument value, we are going to get a different meaning for the routine, namely what that routine does when given that particular argument value. So we pass it the argument value \( ME(e)(s) \), and invoke the resulting transition.

These two alternatives are based on different choices about how to code the meaning of routines. If you code the meaning of a routine simply as a transition, then Spec picks up the argument value out of the magic \( s_a \) variable. But there is nothing mystical going on here. Setting \( s_a \) corresponds exactly to what we would do if we were designing a calling sequence. We would say “I am going to pass the argument in register 1.” Here, register 1 is \( s_a \).

The second approach is a little bit more mystical. We are taking more advantage of the wonderful abstract power and generality that we have. If someone writes a factorial function, we will treat it as an infinite supply of different functions with no arguments; one computes the factorial of 1, another the factorial of 2, another the factorial of 3, and so forth. In

\[
\{ s, o \rightarrow (s, ME(e)(s)) \}
\]

we have to pick out the right one based on the argument value we have. If we coded it this way (and it is merely a coding thing) we would get:

\[
\{ s, o \rightarrow ME(p)(s) \}
\]

This says, first get \( ME(p) \), the meaning of \( p \). This is no longer a transition but a function from argument values to transitions, because the idea is that for every possible argument value, we are going to get a different meaning for the routine, namely what that routine does when given that particular argument value. So we pass the argument value \( ME(e)(s) \), and invoke the resulting transition.
However, there are lots of other ways to do this. One of the things which makes the semantics game hard is that there are many choices you can make. They don’t really make that much difference, but they can create a lot of confusion, because

- a bad choice can leave you in a briar patch of notation,
- you can get confused about what choice was made, and
- every author uses a slightly different scheme.

So, while this

\[
\text{RET ME(p)}(\text{S}) (S("a" \rightarrow \text{ME(e)} (s)),o)
\]

and this

\[
\text{VAR } s' := s{"a" \rightarrow \text{ME(e)}(s)} \mid \text{RET ME(p)}(s) (s',o)
\]

are two ways of writing exactly the same thing, this

\[
\text{RET ME(p)}(s) (\text{ME(e)}(s)) (s,o)
\]

is different, and only makes sense with a different choice about what the meaning of a function is. The latter is more elegant, but we use the former because it is less confusing.

Stepping back from these technical details, what the meaning function is doing is taking an expression and producing its meaning. The expression is a piece of syntax, and there are a lot of possible ways of coding the syntax. Which exact way we choose isn’t that important.

Now we return to the meanings of Spec commands.

**Commands** — **SKIP**

\[
\text{MC} (\text{c1}) \rightarrow (s, o | s = o)
\]

In other words, the outcome after **SKIP** is the same as the pre-state. Later on, in the formal half of the handout, we give a table for the commands which takes advantage of the fact that there is a lot of boilerplate—the \[ \ldots \] stuff is always the same, and so is the treatment of exceptions. So the table just shows, for each syntactic form, what goes after the |.

**Commands** — **HAVOC**

\[
\text{MC} (\text{c1}) \rightarrow (s, o | \text{true})
\]

In other words, after **HAVOC** you can have any outcome. Actually this isn’t quite good enough, since we want to be able to have any sequence of outcomes. We deal with this by introducing another magic state component $\text{havoc}$ with a Bool value. Once $\text{havoc}$ is true, any transition can happen, including one that leaves it true and therefore allows havoc to continue. We express this by adding to the command boilerplate the disjunction $s("\text{havoc}")$, so that if $\text{havoc}$ is true in $s$, any command relates $s$ to any $o$.

Now for the compound commands.

**Commands** — **c1 ; c2**

\[
\text{MC}(c1)(s, o) \rightarrow \text{MC}(c2)(s, o)
\]
Formal atomic semantics of Spec

In the rest of the handout, we describe the meaning of atomic Spec commands in complete detail, except that we do not give precise meanings for the various expression forms other than lambda expressions; for the most part these are drawn from mathematics, and their meanings should be clear. We also omit the detailed semantics of modules, which is complicated and uninteresting.

Overview

The semantics of Spec are defined in three stages: expressions, atomic commands, and non-atomic commands (treated in handout 17 on formal concurrency). For the first two there is no concurrency: expressions and atomic commands are atomic. This makes it possible to give their meanings quite simply:

Expressions as functions from states to results, that is, values or exceptions.

Atomic commands as relations between states and outcomes: a command relates an initial state to every possible outcome of executing the command in the initial state.

An outcome maps names (treated as strings) to values. It also maps three special strings that are not program names (we call them pseudo-names): 

- $\$a$, which is used to pass argument and result values in an invocation;
- $\$x$, which records an exceptional outcome;
- $\$havoc$, which is true if any sequence of later outcomes is possible.

A state is a normal outcome, that is, an outcome which is not exceptional; it has $\$x=\text{noX}$. The looping outcome of a command is encoded as the exception $\$\text{loop}$; since this is not an identifier, you can’t write it in a handler.

The state is divided into a global state that maps variables of the form m.id (for which id is declared at the top level in module m) and a local state that maps variables of the form id (those whose scope is a Var command or a routine). Routines share only the global state; the ones defined by Lambda also have an initial local state, while the ones declared in a routineDecl start with an empty local state. We leave as an exercise for the reader the explicit construction of the global state from the collection of modules that makes up the program.

We give the meaning of a Spec program using Spec itself, by defining functions ME, MC, and MR that return the meaning of an expression, command, and routine. However, we use only the functional part of Spec. Spec is not ideally suited for this job, but it is serviceable and by using it we avoid introducing a new notation. Also, it is instructive to see how the task is writing this particular kind of spec can be handled in Spec.

You might wonder how this spec is related to code for Spec, that is, to a compiler or interpreter. It does look a lot like an interpreter. As with other specs written in Spec, however, this one is not practical code because it uses existential quantifiers and other forms of non-determinism too freely. Most of these quantifiers are just there for clarity and could be replaced by explicit computations of the needed values without much difficulty. Unfortunately, the quantifier in the definition of Var does not have this property; it actually requires a search of all the values of the specified type. Since you have already seen that we don’t know how to give practical code for Spec, it shouldn’t be surprising that this handout doesn’t contain one.
Note that before applying these rules to a Spec program, you must apply the syntactic rewriting rules for constructs like \texttt{VAR id := e and CLASS} that are given in the reference manual. You must also replace all global names with their fully qualified forms, which include the defining module, or \texttt{Global} for names declared globally (see section 8 of the reference manual).

**Terminology**

We begin by giving the types and special values used to represent the Spec program whose meaning is being defined. We use two methods of functions, + (overlay) and \texttt{restrict}, that are defined in section 9 of the reference manual.

**Expressions**

An expression maps a state to a value or exception. Evaluating an expression does not change the state. Thus the meaning of expressions is given by a partial function \( ME \) with type \( E \rightarrow S \rightarrow (V + X) \); that is, given an expression, \( ME \) returns a function from states \( s \) to results (values \( V \) or exceptions \( X \)). \( ME \) is defined informally for all of the expression forms in section 5 of the reference manual. The possible expression forms are literal, variable, and invocation. We give formal definitions only for invocations and \texttt{LAMBDA} literals; they are written in terms of the meaning of commands, so we postpone them to the next section.

**Type checking**

For type checking to work we need to ensure that the value of an expression always has the type of the expression (that is, is a member of the set of values that is the meaning of the type). We do this by structural induction, considering each kind of expression. The type checking of return values ensures that the result of an invocation will have its declared type. Literals are trivial, and the only other expression form is a variable. A variable declared with \texttt{VAR} is initialized to a value \( e \) requires that the value of \( e \) have the type of \( \texttt{VAR} \). If the type of \( e \) is not equal to the type of \( \texttt{VAR} \) because it involves a union or a \texttt{SUCHTHAT}, this check can’t be done statically. To take account of this and to ensure that the meaning of expressions is independent of the static type checking, we assume that in the context \( \texttt{VAR} := e \) the expression \( e \) is replaced by \( e \texttt{AS t} \), where \( t \) is the declared type of \( \texttt{VAR} \). The meaning of \( e \texttt{AS t} \) in state \( s \) is \( ME(e)(s) \) if that is in \( t \) (the set of values of type \( t \)), and the exception \( \texttt{typeX} \) otherwise; this exception can’t be handled because it is not named by an identifier and is therefore a fatal error.

We do not give practical code for the type check itself, that is, the check that a value actually is a member of the set of values of a given type. Such code would require too many details about how values are represented. Note that what many people mean by “type checking” is a proof that every expression in a program always has a result of the correct type. This kind of completely static type checking is not possible for Spec; the presence of unions and \texttt{SUCHTHAT} makes it undecidable. Sections 4 and 5 of the reference manual define what it means for one type to fit another and for a type to be suitable. These definitions are a sketch of how to code as much static type checking as Spec easily admits.
**Command**

<table>
<thead>
<tr>
<th>Command</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIP</td>
<td>$o = s$</td>
</tr>
<tr>
<td>HAVOC</td>
<td>true</td>
</tr>
<tr>
<td>RET e</td>
<td>$o = s { $x \rightarrow \text{retX}, $a \rightarrow \text{ME(e)}(s) }$</td>
</tr>
<tr>
<td>RET</td>
<td>$o = s { $x \rightarrow \text{retX} }$</td>
</tr>
<tr>
<td>RAISE id</td>
<td>$o = s { $x \rightarrow \text{id} }$</td>
</tr>
</tbody>
</table>

**Atomic commands**

An atomic command relates a state to an outcome; in other words, it is defined by an ATr (atomic transition) relation. Thus the meaning of commands is given by a function MC with type C→ATr, where ATr = (S, O) → Bool. We can define the ATr relation for each command by a predicate: a command relates state $s$ to outcome $o$ iff the predicate on $s$ and $o$ is true. We give the predicates in table 1 and explain them informally below; the predicates apply provided there are no exceptions.

Here are the details of how to handle exceptions and how to actually define the MC function. You might want to look at the predicates first, since the meat of the semantics is there.

Table 1: The predicates that define MC(command)(s, o) when there are no exceptions raised by expressions at the top level in command, and $\text{havoc}$ is false.

---

[1] The first case for assignment applies only if the right side is not an invocation of an APROC. Because an invocation of an APROC can have side effects, it needs different treatment.

---

**Table 1**

<table>
<thead>
<tr>
<th>Predicate</th>
</tr>
</thead>
</table>

- $\text{VALUE retX}$ and leaves the returned value in $\$a$.
- RAISE yields an exceptional outcome which records the exception id in $\$x$.
- An invocation relates $s$ to $o$ iff the routine which is the value of $e_1$ (produced by $\text{ME(e1)}(s)$) does so after $s$ is modified to bind $\$a$ to the actual argument; thus $\$a$ is used to communicate the value of the actual to the routine.
- An assignment leaves the state unchanged except for the variable denoted by the left side, which gets the value denoted by the right side. Recall that assignment to a
component of a function, sequence, or record variable is shorthand for assignment of a suitable constructor to the entire variable, as described in the reference manual. If the right side is an invocation of a procedure, the value assigned is the value of $a$ in the outcome of the invocation; thus $a$ also communicates the result of the invocation back to the invoker.

Now for the command components; their meaning is defined in terms of the meaning of their subcommands.

A guarded command $e$ $\rightarrow c$ has the same meaning as $c$ except that $e$ must be true.

A choice relates $s$ to $o$ if either part does.

An else $c_1$ $[*]$ $c_2$ relates $s$ to $o$ if $c_1$ does or if $c_1$ has no outcome and $c_2$ does.

A sequential composition $c_1$ $; c_2$ relates $s$ to $o$ if there is a suitable intermediate state, or if $o$ is an exceptional outcome of $c_1$.

$c_1$ EXCEPT $xs$=>$c_2$ is the same as $c_1$ for a normal outcome or an exceptional outcome not in the exception set $xs$. For an exceptional outcome $o'$ in $xs$, $c_2$ must relate $o'$ as a normal state to $o$. This is the dual of the meaning of $c_1$; $c_2$ if $xs$ includes all exceptions.

VAR $id$: $t$ $\mid c$ relates $s$ to $o$ if there is a value $v$ of type $t$ such that $c$ relates ($s$ with $id$ bound to $v$) to an $o'$ which is the same as $o$ except that $id$ is undefined in $o$. It is this existential quantifier that makes the spec useless as an interpreter for Spec.

The meaning of $DO$ $c$ $OD$ can’t be given so easily. It is the fixed point of the sequence of longer and longer repetitions of $c$. It is possible for $DO$ $c$ $OD$ to loop indefinitely; in this case it relates $s$ to $s$ with "$x"$->loopX$. This is not the same as relating $s$ to no outcome, as $false$ $\rightarrow$ $SKIP$ does.

The multiple occurrences of declInit and var in VAR declInit* and (varList):=exp are left as boring exercises, along with routines that have several formals.

**Routines**

Now for the meaning of a routine. We define a meaning function $MR$ for a routineDecl that relates the meaning of the routine to the meaning of the routine’s body; since the body is a command, we can get its meaning from MC. The idea is that the meaning of the routine should be a relation of states to outcomes just like the meaning of a command. In this relation, the pseudo-name $a$ holds the argument in the initial state and the result in the outcome. For technical reasons, however, we define $MR$ to yield not an $S$-$\rightarrow$ATr, but an $S$-$\rightarrow$ATr; a local state (static below) must be supplied to get the transition relation for the routine. For a LAMBDA this local state is the current state of its containing command. For a routine declared at top level in a module this state is empty.

The $MR$ function works in the obvious way:


---

1. Check that the argument value in $a$ has the type of the formal.
2. Remove local names from the state, since a routine shares only global state with its invoker.
3. Bind the value to the formal.
4. Find out using MC how the routine body relates the resulting state to an outcome.
5. Make the invoker’s outcome from the invoker’s local state and the routine’s final global state.
6. Deal with the various exceptions in that outcome.

A retX outcome results in a normal outcome for the invocation if the result has the result type of the routine, and a typeX outcome otherwise.

A normal outcome is converted to typeX, a type error, since the routine didn’t supply a result of the correct type.

An exception raised in the body is passed on.

**Invocation and LAMBDA expressions**

We have already given in MC the meaning of invocations in commands, so we can use MC to deal with invocations in expressions. Here is the fragment of the definition of ME that deals with an E that is an invocation $e1(e2)$ of a function. It is written in terms of the meaning MC(C(e1(e2)) of the invocation as a command, which is defined above. The meaning of the command is an atomic transition $aTr$, a predicate on an initial state and an outcome of the routine. In the outcome the value of the pseudo-name $a$ is the value returned by the function. The definition given here discards any side-effects of the function; in fact, in a legal Spec program there can be no side-effects, since functions are not allowed to assign to non-local variables or call procedures.
FUNC ME(e) -> (S -> (V + X)) =
  IF
    [ ] VAR e1, e2 | e = E« e1(e2) » =>
      % if E is an invocation its meaning is this function from states to values
      VAR aTr := MC(C« e1(e2) ») |
        RET ( LAMBDA (s) -> V =
            aTr(func(s)) |
            RET (~o.isX => o("$a") [*] o("$x")) )
  FI

The result of the expression is the value of $a$ in the outcome if it is normal, the value of $x$ if it is exceptional. If the invocation has no outcome or more than one outcome, ME(e)(s) is undefined.

The fragment of ME for LAMBDA uses MR to get the meaning of a FUNC with the same signature and body. As we explained earlier, this meaning is a function from a state to a transition function, and it is the value of ME((LAMBDA ...)). The value of (LAMBDA ...), like the value of any expression, is the result of evaluating ME((LAMBDA ...)) on the current state. This yields a transition function as we expect, and that function captures the local state of the LAMBDA expression; this is standard static scoping.

IF
  [ ] VAR signature, c0 | e = E« (LAMBDA signature = c0) » =>
    RET MR(RE FUNC id1 signature = c0 »)
  FI