6.826 Principles of Computer Systems **Fall Semester, 1993**

Lecture 5: September 27, 1993

Lecturer: Bill Weihl *Scribe: Janey Hoe and Kin Hong Kan*[1]

Handouts:
Handout 12: History Variables
Handout 13: Rudiments of Logic
Handout 14: Atomic Semantics of Spec
Handout 15: Problem Set #3

1 Formal Semantics

We want to discuss formal semantics of Spec. This will allow us to give unambiguous definition of what a Spec program means. One way to approach this is to use denotational semantics, which is mathematical and good for doing proofs. However, we take an approach that is easier to understand for most people. We look at a Spec program as a state machine. When we are writing a Spec program, we are writing a state machine with states and transitions. Each instruction in the instruction set specifies a transition for the state machine. It is important to have a precise definition for what an instruction does in any language, so that two persons reading the same specification can agree on the effects of an instruction.

To elaborate, one purpose of this is to make sure that there is no doubt that we really do know what we are talking about. And, in general, notations or facilities that exist for describing what computer systems should do, are not in that state of grace.

If you read the pascal manual or the C manual you will come away, if you read it carefully, with a number of questions, about what exactly happens if I do this and this. Questions which the manual will not answer adequately. Questions with the property that two reasonably intelligent people who have studied the thing carefully can come to different conclusions. And argue for a long time and not be able to decide what is the right answer by reading the manual.

There is of course one class of mechanisms for saying what the computer should do that tends not to have that property, and that is the machines instruction set. The reason for this is that it really tends to just be a garden variety state machine, with fairly simple transitions, which is not beyond the power of the guy who is writing the manual to describe properly. Whereas a programming language is not like that. It has much more power and generality, and wonderfulness, and there tends to be a lot of confusion there.

One of the things we want to be able to do is to write specifications for computer systems, and one example of a computer system is a programming language. A notation for writing programs or a notation for writing specifications. We are going to learn how to write a spec for that particular class of computer systems.

[1] These notes are based on Fall 92 lecture 5 notes prepared by Mitchell Charity and Fall 92 lecture 6 notes prepared by Thomas Lee.

This is a very different application of spec from the one which we have been looking at in the last week, which has been file systems. For this purpose, spec is not the ideal descriptive notation. If you are in the business of writing the semantics of programming languages you wouldn't use spec. People have invented a large number of other notations and some of them are actually better than spec, although most of them are far worse. But it 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 trying to pick up a new one each time. For there are a lot of pitfalls involved in devising a new notation. So we are going to stick with this one and see what we can get out of it.

So those are the two things which will be going on for the next hour. We are going to get down to the foundations of spec, and we are going to see another, very different application of spec. Certainly a programming language is a pretty different kind of system than a file system.

For this lecture, we will only talk about the sequential semantics of Spec, the atomic semantics of Spec. We are not going to talk about concurrent semantics.

Consider the program:

```
                x,y = 0
thread 1:                    thread 2:
 << x := 3 >>                 << z := x + y >>
 << y := 4 >>
```

In the concurrent world, it is possible to get any of the answers 0, 3 or 7.

In the sequential world, which we are in today, the only possible answers are 0 and 7. It is a simpler world. Bill will be talking next week about the semantics of concurrency, which is unavoidably more complicated.

In a sequential spec, there are basically three constructs:

```
- Expressions
- Commands
- Routines
```

(I may occasionally accidentally substitute "statement" for "command".)

In order to describe what each of these things mean, you first of all have to have some notion of what kind of thing the meaning of an expression or command might be. And then we have to in detail explain exactly what the meaning of each possible kind of expression is.

The basic technique we are going to use for that is the standard method you always use in a situation where you have things which are made up out of smaller things. That is called structural induction.

2 Sequential Semantics of Spec

We will talk about sequential or atomic semantics. What does this restriction mean?

2.1 Example

Consider the following piece of code:

```
VAR x,y: Int := 0
Thread 1
        <<x:=3>>;
        <<y:=4>>
Thread 2
        <<z:=x+y>>
```

A sequential execution of the piece of code would give:

```
z = 0 or z = 7
```

A concurrent execution of the piece of code would give:

```
z = 0 or z = 3 or z = 7
```

We will talk about only sequential semantics. We can assume that once Thread 1 starts executing, nothing will come in between the sequential composition `<<x:=3>>; <<y:=4>>`.

2.2 Sequential Semantics

The sequential semantics of Spec is composed of:

1. expressions
2. commands
3. routines

The semantics is defined in the Spec manual using structural induction.

Structural induction specifies the meaning of an expression by defining the components of this expression inductively, using its sub-expressions that are already defined.

For example, looking at `x+y`, we would first find the meaning of sub-expression `x` and sub-expression `y` and then apply the operator `+`.

2.3 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.

State is a function of names to values.

State: Name $->$ Value

For example

```
VAR x: Int
    y: Int
```

If this is all there is, 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. So somehow the meaning we give to this whole construct has to express that.

Then 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 which changes the state, so that no matter what the x was before, it is 1 afterwards. Because thats what we want this assignment to mean.

Spec has been substantially simplified since last year. In particular, "references" (like pointers in C) have been ripped out. They complicate understanding of what is going on quite a bit, and they are not really necessary. When you are doing problems, if you feel the urge to call malloc, the correct thing to do is to make a sequence of whatever sort of thing you want to allocate, and then choose a new index in the sequence which isn't being used already. Make a sequence or make a function which maps an integer or some convenient sort of name into the values that you want to allocate.

So this is the state, just these name to value mappings.

2.3.1 Names

Spec has a module structure. The names can have two parts. When referring to a variable in another module, both parts must be used.

```
  MODULE a             MODULE b
    VAR x
      x := 3               a.x := 3

      a.x := 3
```

From the point of view of the semantics, we are going to use a.x as the name, everywhere. In order to apply the semantics, one of the things you have to do is go through the program first and every place you see an x, if that x was declared in the current module a, then you replace in with a.x. You convert all references to global variables to these two part names. This makes things simpler, but crufty to read.

All the global state is going to consist of these two part names.

However, local variables are not prefixed by the module name:

```
PROC
  VAR i | ... i
```

This is how we tell the global state apart from the local state. Global state has dots, local state does not.

Question: Can modules be nested?

No. The general parameters for the design of spec were to try to make it suitable for the kinds of specifications and implementations that we do in this course, some of which are moderately complex.

Spec is plenty good enough for everything we need to do in this course. In general the rule was, if it's not really needed to write the specifications of the kind we are doing in this course, then we left it out, to keep it simpler.

2.4 Expressions

Expressions have two properties:

1. They cannot change the state of a piece of Spec code.
2. They are deterministic.

Spec is different from other languages, such as C, which allow expressions such as the following:

`if (x=3) ...;` This is not allowed in Spec.

Besides, if expressions can be nondeterministic, we may have the possibility that the following evaluates to false:

```
(f(x)=f(x))
```

which is counter-intuitive.

The meaning of expressions is given by a partial function `ME` with type:

```
ME: E->(S->(V+X))
```

where `E` means expression, `S` means states, `V` means values and `X` means exceptions.

There are 3 kinds of expressions:

- literals, *e.g.*, `1`, which is the syntactic sugar for `LAMBDA(S)->(V+X) = RET 1`
- variables, *e.g.*, `x`, which is the syntactic sugar for `LAMBDA(S)->(V+X) = RET S("x")`
- invocations, *e.g.*, f(x)

2.5 Commands

The meaning of commands is given by a function `MC` with type:

```
MC: C->ATr, where Atr = [s,o] -> Bool.
```

where `ATr` relation for each command is a predicate, *i.e.*, a command relates state `S` to outcome `O` iff the predicate on `S` and `O` is true.

Examples of some outcomes `o`:

```
o("$x")= "" => normal return
o("$x")= "foo" => exception foo
```

A command can be non-deterministic.

Handout 14, p.6 gives a detailed definition of all the commands:

```
FUNC MC(c) -> ATr = ...
        [] VAR var,e | c = <<var:=e>> =>
        RET (LAMBDA (o,s) -> Bool =
                RET ...full predicate for this case...)
```

e.g., for an assignment statement,

```
var:=e    o = s{var->ME(e)(s)}
```

An alternate style is:

```
o(var) = ME(e)(s),
```

The above, however, does not say anything about the possible side-effects of the assignment statement, *i.e.*, the state values of other valuables are not specified.

Another example:

```
f(e)-> LAMBDA(s,o)-> Bool = RET ME(P)(S)(S{"$a"->ME(e)(s)},o)
```

Now let us go into the details of the semantics of some common commands. The following notes are extracted from Fall 92 lecture 5 and 6 notes.

2.5.1 Commands - assignment

x := 1, or in general:

```
variable := expression
```

What we have to come up with is one of these expressions which is

```
lambda s,o = RET  ...
```

How is o going to be related to s? By the function returning true. This is a slight uncomfortable way to write this. If we choose this way of describing a relation, we have to come up with a predicate on the state and the outcome, which is true exactly when we want to make a transition. There is not going to be any exceptional outcome. We are encoding a relation between states. We are encoding the relations as these functions from a state and outcome to a boolean. And the function is supposed to give back true, exactly when the relation holds.

So when does the relation hold for variable gets expression?

Well, perhaps when o(x) = exp?

```
RET o("x") = ME(e)(s)      ?
```

ME is the meaning function for expressions.

Well, the valid transition

```
x = 0       x = 1
       ->
y = 0       y = 0
```

would certainly be allowed.

But what others would be allowed? What about:

```
            x = 1
       ->
            y = 94
```

So this can't be quite right. Half right, but missing something important. You have to say that you don't mess around with the rest of the state.

The way you do that is to say that the outcome is equal to the state except at the variable.

```
 o = s{"var":= ME(e)(s)}
```

This is just a function constructor, as in f{arg:=value}.

2.5.2 Commands - an alternate encoding

As we said before, there are many ways to code the command relation. One way we mentioned was:

```
- Commands    S -> SET S
```

I tried that when writing the semantics for spec, but, either I did something stupid, or things just come out cruftier. But it certainly works.

There is another way to write this, which has a lot of advantages. That is to 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 this

```
(x' = 1 /\ y' = y)
```

is another way of writing

```
o = s{"x":=1}
```

In fact, this approach is another way of writing programs. I could write everything just as predicates. Of course, I could write everything as this o = s{...} sort of cruft, but that would look pretty awful. It is more conceivable that we would want to write things as predicates, because it doesn't look so bad.

Sometimes it's actually nice to do this. Say you want to write the predicate which says you can have any value at all for x. The spec

```
VAR z | x:= z
```

is just

```
(y':=y)
```

(in the simple world where the only state variables are x and y). This is much simpler that 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 these y'=y stuff clutter things up an awful lot.

This was just an aside, but sometimes I will take advantage of this observation, and when talking generally about what kinds of things can go on in a specification, I will be willing to describe it using predicates, rather than using these functions of states to boolean.

2.5.3 Commands - routine invocation f(x)

What are the semantics of routine application/invocation?

Well, it has to do something with s.

What about the argument? There are many ways to deal with the argument. The way we do it here, and in the handout, is to use another pseudo-variable, $a. $a is used to pass the argument and get back the result.

The meaning of f(e) is going to be

```
f(e)
  lambda s,o = RET           s                        | Take the state,
                              {"$a":=ME(e)(s)}        | append the argument,
                  ME(f)(s)                            | get the function,
                          (                      ,o)  | and apply it.

  lambda s,o = RET ME(f)(s)(s{"$a":=ME(e)(s)},o)
```

So what does this say? This invocation relates a state to an outcome, if, when you take that state, and modify its $a component to be equal to the value of the argument, the meaning of the function relates that state to the outcome.

Another way of writing this, which isn't so nested, and might be clearer, would be to introduce some intermediate state s':

```
  VAR s' = s{"$a":=ME(e)(s)} | RET ME(f)(s)(s',o)
```

These two are exactly the same thing. The invocation relates s to o, iff the function relates s' to o, where s' is just s with the argument passing component modified.

$a is just a way of communicating the argument value to the function.

Question: Why use ME(f)(s) rather than MR? I would use MR if I was looking at a FUNC. But f is just a variable (which had better be bound to a function, or this wont typecheck).

An alternate encoding

Here is a different way of communicating the argument value to the function. You could take the view, that when I write the procedure

```
PROC p (i: Int)
```

what we are doing is defining a whole flock of different commands, one for every possible argument value.

Then we need 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:

```
ME(f)(s) ( ME(e)(s) ) (s,o)
```

This says, first, get the meaning of f. f is not going to be 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 function, namely what that function does when given that particular argument value.

So we pass it the argument value, and apply the resulting transition.

Encodings contrasted

These two alternatives are based on different choices about how to code the meaning of procedures. If you code the meaning of a routine simply as a transition, that is spec picks up the argument value out of this magic $a variable.

But there is nothing mystical going on here. Setting $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 $a.

The second approach is a little bit more mystical. We are taking more advantage of all 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, one computes the factorial of 1, another the factorial of 2, another the factorial of 3, and so forth.

So in $ME(f)(s)(ME(e)(s))(s,o)$, $ME(f)(s)$ is the infinite supply, $ME(e)(s)$ is the argument that picks out a particular function, to which we finally pass (s,o).

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, and they don't really make that much difference, but they can create a lot of confusion. Firstly, because a bad choice can leave you in a thicket of notation. And secondly, because you can get confused about what choice was made.

So, while this

```
RET ME(f)(s)(s{"$a":=ME(e)(s)},o)
```

and this,

```
VAR s' = s{"$a":=ME(e)(s)} | RET ME(f)(s)(s',o)
```

are two ways of writing exactly the same thing, this,

```
ME(f)(s) ( ME(e)(s) ) (s,o)
```

is different, and only makes sense with a different choice about what the meaning of a function is.

So while I argue that $ME(f)(s)(ME(e)(s))(s,o)$ is more beautiful, I propose to use $lambda$ $s,o = RETME(f)(s)(s\{``\$a'' := ME(e)(s)\},o)$ because I think it 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 chose isn't that important.

2.5.4 Commands - HAVOC

λ s,o = RET true

2.5.5 Commands - SKIP

λ s,o = RET s = o

In the handout, what I have done for the commands is to notice that there is a lot of boilerplating - this $\lambda s, o = RET$ stuff, is always the same. So I have given a table which shows for each syntactic form, what goes after the RET.

Now for the compound commands.

2.5.6 Commands - c1 [] c2

```
MC(c1)          MC(c2)
      (s,o)           (s,o)
            \/

MC(c1)(s,o) \/ MC(c2)(s,o)
```

2.5.7 Commands - c1 [*] c2

Well, it is clear we should begin with

```
MC(c1)(s,o) \/
```

But what next?

One possibility would be

```
                ~MC(c1)(s,o) ==> ...
```

This is in the right direction, but not correct. Else means that if there is no POSSIBLE outcome of c1, then you get to try c2. So there are two possible ways for an else to relate a state to an outcome. One is for c1 to relate the state to the outcome, the other is that there is no possible way to make progress with c1 in the state, and c2 to relates the state to the outcome.

So, the correct encoding is

```
MC(c1)(s,o) \/ (ALL o' | ~MC(c1)(s,o')) /\  MC(c2)(s,o) )
```

By the way, I handed out a small handout (#13) which I labeled "Rudiments of Logic". This is in response to my discovering last year that there were people who did not know what De Morgan's laws were.

2.6 Semicolon

c1 **;** c2

Although the meaning of semicolon may seem intuitively obvious, it is much more complex than one might first suspect — much more complicated than **or**, for instance. We interpret the statement : c1 or c2 as MC(c1) $\vee$ MC(c2). Because semicolon is a sequential composition, it requires that our semantics move through a median state.

If these were functions (if we could describe the commands as functions) then we could simply describe a sequential composition as:

(F2 (F1 S))

However, because Spec is not a functional language in this respect, we need to establish an intermediate state as a precursor to the final output state. As a first attempt at a semantics, we might suggest:

```
(lambda (s, o) -> Bool = RET
    (EXISTS o′ | MC(c1)(s, o′) /\ MC(c2)(o′,o)))
```

In words, this says that you can get from S to O if there exists an intermediate state O'. But is this always the case?

What if c1 produces an exception? Given the same situation as above, (x $<0 \Rightarrow$ c1); c2, when c1 produces an exception, we should *not* execute c2. Our semantics does not capture this possibility. To correct for such an occurrence, we need to verify that O' is a legitimate state. The correct semantics should therefore read:

```
(EXISTS o′ | MC(c1)(s, o′) /\ (˜IsX(o′) /\ MC(c2)
                                \/ IsX(o′) /\ o′ = o))
```

2.7 EXCEPT

Now, what if we have a handler for the exception? If we assume (for simplicity) that all exceptions are handled, one possible solution might be: c1 EXCEPT XS $\Rightarrow$ c2. In this case, we would simply implement the complement of the semicolon case. If we get an exception, then do c2. If there is no exception, do NOT do c2. We also need to include an additional check to insure that the exception considered is an element of the exception set — that is to say that it is a handled exception.

```
(EXISTS o′ | MC(c1)(s, o′) /\
            ( ((˜IsX(o′) \/ õ′("$x") in XS) /\ o′ = o)
            \/ IsX(o′) /\ o′("$x") in XS) /\ MC(c2)(o′{"$x" -> ""}, o)
            )
)
```

So, given this semantics for handling exceptions,

	if normal	do c1, no c2, do c3
Given: (c1 EXCEPT XS $\Rightarrow$ c2); c3	if exception, handled	do c1, do c2, do c3
	if exception and not handled	do c1, no c2, no c3

2.8 VAR

VAR id: T | c0

The general concept here is to introduce an existential variable as a quantifier. Therefore, our intuition might suggest a semantics of the following sort:

```
(EXISTS v | v in T /\ MC(c0)(s{id->v},o))
```

However, if we look carefully, we will realize that *id* is left defined in the output state O. Why is this bad? To resolve this omission, we need to introduce an intermediate state O' from which we may arrive at the final output state O where *id* is undefined.

```
(EXISTS v, o' | v in T /\ MC(c0)(s{id->v}, o') /\ o = o'{id ->})
```

2.9 Routines

As a last comment on semantics, we examined routines. In SPEC, routines include functions, atomic procedures, and procedures. For utility, the lecture focused on atomic procedures. How do we think about APROCs?

We know that APROCs describe transitions from their input state to their output state. Given this transition, then,how does SPEC handle the results? SPEC introduces a pseudo name $a to which a procedure's argument value is bound. SPEC then collects the value from $a. Please refer to the handout for a more complete discussion. In reality, SPEC is much more complex because it attempts to make RET more convenient. In particular, the intention was to permit RET to be called from anywhere and to raise the exception $RET when called.