6.826 Principles of Computer Systems **Fall Semester, 1992**

Lecture 6: September 30, 1992

Lecturer: Professor Butler Lampson *Scribe: Thomas Lee*

1 Handouts

The handouts distributed today were:

- Handout 17 Basics of Performance Analysis
- Handout 18 "Queueing in Networks of Computers" by Denning

Other handouts relevant to this lecture include:

- Handouts 14 *Atomic Semantics of Spec*
- Handouts 16 *History Variables*

Denning's article, in particular, was recommended as an article for review of the topic as well as for identifying articles for further reading, as the course itself will only briefly touch on performance.

2 Overview

- finish semantics of spec.
- interlude re: use of dummy variables in proofs of an implementation.
- performance issues.

3 Semantics of Spec (cont'd)

In the previous lecture, we completed the semantics for **Else**. Recall that the meaning of a command `C` is given by `MC(C)` of type `lambda (State, Outcome) -> Bool`. To complete the semantics we study the meaning of some other kinds of commands.

3.1 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 intepret 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 descibe 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 occurance, 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))
```

3.2 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') \/ ~o'("$x") in XS) /\ o' = o)
            \/ IsX(o') /\ o'("$x") in XS) /\ MC(c2)(o'{"$x" -> ""}, o)
            )
)
```

So, given this semantics for handling exceptions, Given:

(c1 EXCEPT XS $\Rightarrow$ c2); c3)

if normal	do c1, no c2, do c3
if exception, handled	do c1, do c2, do c3
if exception and not handled	do c1, no c2, no c3

3.3 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 ouput 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 ouput state O where *id* is undefined.

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

3.4 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 reults? 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.

4 History Variables

Our technique for proving that an implementation satisfies a specification entails deriving an abstraction function (AF) and, perhaps, a rep invariant, to relate the implementation and the specification. But what happens when we cannot find an AF?

4.1 The Need for Dummy Variables

Consider the following example:

SPECIFICATION: Values are inserted as SEQ INT and the mean is returned.

v: SEQ Int := $\{\}$
Add(i) $\Rightarrow$ v := v ++ i

$$Mean() \Rightarrow Int = \frac{\sum_{\forall v} v}{v.size}$$

IMPLEMENTATION:

total, n: Int := 0
Add(i) = total := total + i; n := n + 1

$$Mean = RET\frac{total}{n}$$

In this example, we can't find an AF because there is no way of extracting the set of integers in the specification (SEQ INT) from the second integer in the implementation (n). The problem is that the specification contains more state than is necessary. But criticizing the specification for being poorly written is not an acceptable response. The purpose of the specification, after all, is to make life easy for the client — it is not supposed to be written to simplify the task of proving that a particular implementation satisfies it (although it would be nice if it did).

4.2 History Variables

The solution is to use a dummy variable. Though we may not always be able to find an AF directly from the implementation, it is ALWAYS possible to modify the implementation by adding dummy variables to that we may find an AF. Because dummy varibles do not actually affect what is happening within the implementation — they don't change anything, we need not implement anything.

Our general strategy will then be to create a new implementation, I_H by adding history variables to our original implementation I and then prove by conventional means, that I_H satisfies the specification. If I_H satisfies the specification, we may be certain that I does as well, because we construct I_H in a manner explicitly to insure that I is equivalent to I_H.

Our new implementation I_H will therefore be modified in the following way:

Add to the implementation var v: SEQ Int: I$\{\}$, v.size = n.
Change: Add(I) = total := total + i; n := n + 1; v := v ++ i

Here, we simply substituted a variable for the entire abstract state of the specification. This technique ALWAYS works. However, such a drastic measure is usually unecessary. A smaller change will usually suffice.

In general, the rules for history variables (HVs) are such that:

- Ordinary variables may afect HVs but HVs may NEVER affect ordinary VARS
- There may be no HVs in the right hand side of an assignment statement to a normal VAR.
- HVs should raise no exceptions.
- Nothing should return an HV (as the value of a RET).
- An HV may not occur in a guard.
- HVs should be "total" in that their being undefined should not disallow an otherwise possible transition in the implementation.

4.3 Equivalence of I and I_H

To formally prove that I and I_H are equivalent, we extend the state space of I (s) to (s, h) by applying the following rules:

1. if s $\in$ $Init$ (the set of initial states)
   then (s,h) $\in$ $Init_H$ (initial states of I_H)

2. if s $\rightarrow$ s' is a transition in I,
   then $\forall$ h $|$ reachable(s, h) $\rightarrow$ ($\exists$ h' $|$ (s, h) $\rightarrow$ (s', h') is a transition in I_H)
   This is what was implied by requiring that HVs be total.

5 Performance

What do we mean when we use the word "performance"? Performance typically has two components:

1. Bandwidth (aka. thruput) — how many per second?
2. Latency (aka. response time) — how many seconds?

5.1 Examples where Bandwidth and Latency are used

1. Pipeline, as in the execution of instructions.
2. Main memory of a computer.
3. Disk.
4. Remote procedure call (RPC) of request-response.
5. Airline reservation system.

6. Professional journals. Publication of papers. Latency to the author is measured in the number of months or years taken to review an article for publication. Thruput from the perspective of the journal may be measured in papers published per issue or per year.

5.2 The Golden Rules of Optimization

1. Know how good of an approximation you need.
   10% is generally good enough. 1% is highly unlikely. The task of optimization may be greatly simplified if you know what can be neglected and what must be included.

2. Costs *do* total up. In calculating the cost of a complex sum, people tend to over-estimate system effects and ignore aggregate estimates. To get the performance of T, sum the performance of T's components:
   $$T = \sum_{\forall i} T_i$$
   The caveat to this rule is that it is OK to simplify complex tasks to optimize or evalute them.

3. Look for Common Cases (80/20 rule). On average, your system will spend 80%of it's time on 20% of the code. In fact, more often than not, the number is more like 3% of the code. Therefore, much of the system may be ignored when calculating efficiency and attempting to optimize.

4. Benchmark. Measure performance. Use counters to find the common cases. The canonical example here is **caching**.
   Look for things that are overwhelmingly frequent or are easy to implement.

   Time total:
   $$T = t_f P_f + t_s P_s$$
   $$= t_f(1 + [\frac{t_s}{t_f} - 1]P_s)$$
   P_f = prob of fast path, P_s = prob of slow path, $P_f + P_s = 1$
   t_f = fast time, and t_s = slow time.

   In most cases,
   $$\frac{t_s}{t_f}$$
   is very large and subtracting 1 makes no difference overall.

5.3 Bottlenecks

Bottlenecks are an extreme form of the "common case" rule noted above. The classic example of a bottleneck is in a network where one machine can restrict flow throughout the network. Two solutions are to either place a higher performance machine at the bottleneck or attempt to distribute the load at the bottleneck amongst several machines.

In general, it is not a good idea to run nodes at 100% thruput because nodes will inevitably become backlogged.

Consider an engine servicing requests from a queue. If we assume a request every 100 ms. and the ability to process a request that quickly, then the engine will be able to keep the size of the queue at either 1 or 0 depending upon how the counter is incremented. However, once requests become distributed randomly — for instance, each individual request is a random variable that obeys the Poisson distribution — then the engine alternates between dead time where it idles because there are no requests in the queue to process, and a series of requests that arrive one after another resulting in a backlog. This worsens the latency.

Typically, roughly 50% utilization is acceptable. At utilization over 80%, the backlog may build up to unmanageable levels, resulting in long response time.

5.4 General Comments on Performance

Optimization is not, in general — and with specific reference to utilization, very rewarding work. Not only is it very difficult, but it is also very complex. Moreover, the rapid pace of innovation in the computer industry contributes directly to the futility of large investments into optimization. This is not to say that optimization is bad or unnecessary. It is merely to suggest that people should consider carefully whether the high costs of engaging in optimization are justified in each particular case.