Lecture 2: September 15, 1993

Lecturer: Bill Weihl *Scribes: Christopher Small and Joanna Bryson*[1]

1 Administrative Notes

- The TAs for the course will be:

  David Maw (maw@ginger.lcs.mit.edu), and
  David Wald (wald@theory.lcs.mit.edu)

- To request extra handouts, see the secretary (Lisa Eastman, NE43-251). Please check the file cabinets *outside* her office before going in and bothering her!

2 Specifications and Implementations

We began by discussing specifications and implementations of procedures. There are two main concepts:

- We model specifications and implementations as a relation between inputs and the outputs.
- The relation for the implementation should be a (possibly equal) subset of the relation for the specification. To force non-trivial implementations we add the constraint that the domain set of inputs for the implementation should be equivalent to the domain of the specification.

2.1 Search Specification

The first example (from Handout 5) is a procedure for searching through an array for a given element:

```
APROC Search (a: SEQ Int, x: Int) -> Int RAISES {NotFound} =
   << VAR i: Int | (0 <= i /\ i < a.size /\ a[i]=x) => RET i
      [*] RAISE NotFound >>
```

It is important to note the distinction between this—a specification—and an implementation. A specification may generally be non-deterministic: for example, the behavior of this code is undefined for the case where the element `x` appears more than once in the array. The specification indicates that *any* `i` satisfying these constraints may be returned.

[1] These notes are an updated copy of those of Keith M. Swartz from Fall, 1992. Swartz acknowledges Carl G. Heinzl for portions of sections 4.1 and 4.2 from Fall, 1991.

The first line of the specification is the *header*. This contains the name of the procedure or function, the names and types of its arguments, the type of its result, and any exceptions it may raise. The next two lines form the *body*, which is enclosed in *atomicity brackets*.[2] The body is of the form:

```
VAR i:  Int | ...(a)...
=> ...(b)...
[*] ...(c)...
```

In English, this may be read as "Choose an integer `i`, such that (a), then execute (b) if one is found, otherwise execute (c)."

The important question raised by this specification is what the actual relation is between input and output. The type of relation may be denoted as:

Inputs: $\texttt{SEQ Int} \times \texttt{Int}$
Output: $\texttt{Int} \cup \{\texttt{NotFound}\}$

The relation itself, where `(a,x)` is the input pair and `r` is the result, is:

$$R_{search} = \{\langle(a,x),r\rangle \mid a[r] = x \vee [(\forall r'.a[r'] \neq x) \wedge r = \texttt{NotFound}]\,\}$$

Note that in general we want to reason about programs at this level of abstraction, not at the level of the Spec text.

2.2 Search Implementation

Whereas a specification merely lists the constraints on the results, an implementation actually determines how to *compute* the results. An implementation may be non-deterministic too, but in the example below, it is deterministic and returns the first result it finds, starting at the beginning of the array, and progressing to the end.

```
APROC Search (a: SEQ[INT], x: INT) -> INT RAISES {NotFound} =
  << VAR i: INT := 0 |
     DO
       i < a.size /\ a[i]#x => i:=i+1
     OD;
     i = a.size => RAISE Notfound
     [*] RET i
  >>
```

This example introduces a `DO` loop. In general, it is of the form:

```
DO
  S
OD;
```

[2] Until the course turns toward concurrency and crashes, all operations may be considered to be atomic.

If `S` has an outcome in the current state, that is, if it relates the current state to some final state, it is executed. This is done repeatedly, until `S` fails, at which point the loop is exited. A common form in which the loop appears is:

```
DO
  P => S'
OD;
```

This is equivalent to a "while" loop: as long as `P` is true and `S'` has an outcome in the current state, it is executed. At the end of the loop in the example, *either* `i` is equal to `a.size`, which raises the exception `NotFound`, or $a[i] = x$, when it returns `i`.

The `P => S` construct is very common in Spec programs. The predicate `P` is called the *guard*. Two other forms commonly used are:

```
[*] S
```

and

```
S1 [] S2
```

In the first case, `S` is an "else clause"; `P => S [*] S1` is read "if `P` then `S` else `S1`". The latter clause is an "or", one of the statements is chosen non-deterministically.

2.2.1 Specificity

In the example code above, the implementation represents a more specific input-output relation than the specification; it returns the first match found. More rigorously,

$$a, x \mapsto \{\text{LEAST}\ i \mid 0 \le i < a.size \wedge a[i] = x\} \cup \{\texttt{NotFound} \mid \nexists i \text{ s.t. } a[i] = x\}$$

The spec allows several possible answers, but the implementation only returns one. Implementations are *allowed* to restrict output, to make it deterministic.

Testing whether or not the implementation satisfies the specification requires passing two constraints:

- $R_I \subseteq R_S$
- $\text{Dom}(I) = \text{Dom}(S)$

In other words,

1. the set of all possible outcomes of the implementation must be a subset of the outcomes the specification allows, and
2. the domains of the implementation and specification should be equivalent.

In this example, the implementation does satisfy the specification, by passing these two requirements. However, whether or not it is the *right* specification cannot be determined: this is wholly dependent on what the client *wants*, *i.e.*, does he want any or the least or the greatest `i`, *etc*.

It will be assumed in future lectures that we are able to read Spec code and reason about it in these terms.

NB: We will be expected to be able to abstract the I/O pairs from any specification or implementation for the course.

NB: We aren't talking about properties like termination yet—we are only dealing with partial correctness. These will get discussed in a few weeks.

2.3 Binary Search Implementation

It's important to understand that you can't make assumptions in the implementation beyond what is available in the specification. For example, you couldn't just use a binary search, because that would assume the array is sorted, and this isn't specified.

How would you modify the previous section to allow for binary search? You could add the following to the original specification to ensure the array is sorted:

```
APROC Search (a: SEQ INT, x: INT) -> INT RAISES {NotFound} =
   << ~Sorted(a) => HAVOC
      [*] VAR i: INT | (0 <= i /\ i < a.size /\ a[i]=x) => RET i
          [*] RAISE NotFound >>
```

Note that the routine will cause `HAVOC` if the array is not sorted first. (The `Sorted` function can be found in Handout 5.) `HAVOC` implies that literally anything can happen. This is the chief method for specifying requirements or preconditions on the caller. We don't want to check the array to make sure it's sorted, because this would throw away the advantage of using a binary search algorithm since the check itself would take linear time. The use of `HAVOC` is present to make it clear to the caller that the procedure makes this assumption and makes no guarantees of its behavior if the assumption is false. The condition is on the caller—the caller must make sure that the condition holds.[3]

Another (faulty) solution would be to move the following into the spec:

```
Sorted(A) ==> VAR i:  INT | ...
```

This specifies that no outcome be produced in the event the array is not sorted—it may not return an `Int` or signal `NotFound`. It moves the test from the caller into the procedure. The problem in this case is, the check would be as long as the worst-case linear search you are trying to avoid. In general, you need to balance between loss of efficiency and trusting the caller.

[3] `HAVOC` is a pretty clear flag, but it's kind of bad style, you could probably give a better idea of what's really going to happen. Depends on what you are using the spec for.

3 Data Abstractions

Data abstraction allows us to encapsulate multiple operations as well as state. Abstraction hides unnecessary details from the users of the system; *i.e.*, they need not be concerned with the details of the implementation, only the manner in which the data structures change according to the specification. Without data abstraction it would be quite difficult to understand systems as they become larger and their data structures grow in complexity.

A data abstraction differs from a procedure in that it can have more than one operation. More importantly, it has data, and therefore *internal state*.

3.1 The Simple Memory Module

As an example, consider the simple memory module (Example 4 in Handout 5) that keeps track of a data value for each of some domain of addresses. In Spec, we have several different *levels* in which to write. Modules are essentially a building block for name and procedure encapsulation. Modules may have the following:

- Types
- Variables (containing the *states* of the module)
- Operations (procedures and functions)

Modules may be *parameterized*, which implies that the module may be instantiated with different values for the different type variables; in this case, A and D represent the address and data types in the simple memory module. We have a type `M` that maps addresses to data. (`TYPE M = A -> D`) The state of the simple memory module is simply one of these functions—at any point in time one can think of the state as being a function of addresses to data.[4]

There are five procedures in this example:

- `Init` chooses an arbitrary *total* mapping, so that `m` is defined over all addresses.
- `Reset(d)` resets the state of memory with value `d`, *e.g.*, for zeroing out space.[5]
- `Read(a)` returns the current state of the memory for the given address `a`.
- `Write(a,d)` writes value of `d` to address `a`.
- `Swap(a,d)` swaps the value of address `a` with the new value `d` and returns the old value.

[4] Declarations of variables, unless explicitly specified, take on the (implicit) type of its capitalized equivalent, *e.g.*, `m` takes on the type `M`. (This is simply a convention of SPEC; for further information, read the SPEC Reference Manual—Handout 4.)

[5] This definition introduces the mapping constructor `* ->`, which appears in the form `S{ * -> q}`, and returns a function that maps all domain elements to the value `q`.

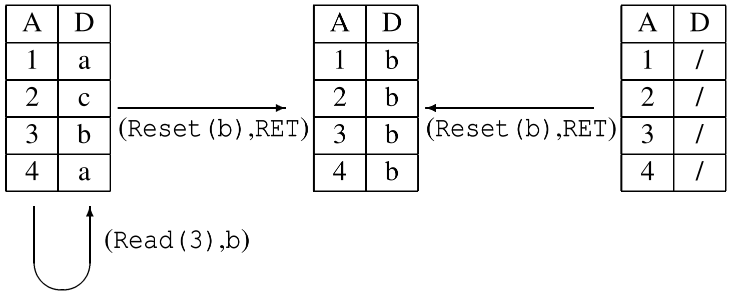

Figure 1: Examples of state transitions.

The initialization of m happens before any of the other procedures are called. Note that a write can be expressed by the expression `m := m{a->d}`, which is a function constructor that takes in a function m and sets it to another function that has the same values as the original m except with the value of a mapped to the value d.

We can't use the kind of abstract model of just I/O relations we used for procedures for data abstractions, because of these internal states. Instead, the right way to think about a data abstraction is as a *state machine* (but not necessarily a finite one). What is the abstract behavior, or what do the users *depend* on? Users care about the responses to a sequence of invocations of the various routines (*e.g.*, initialize followed by some sequence of reads, writes, and swaps). Initialize and write simply return without passing any information back to the user, while read and swap pass back data to the user. So the thing you need to worry about is the sequence of transitions between states. (see Handout 8, Figure 1)

A *transition* between states is labeled by the invocation/response pair, and a *trace* is a sequence of such transitions. In a configuration where A is the set $\{1,2,3,4\}$ and the domain of possible data values D is $\{a,b,c\}$, any single state may be represented as four values, each being one of a, b, c, or nothing.[6] Figure 1 shows a few possible transitions. Note how the resulting state is unchanged for operations without side effects (*e.g.*, `Read`).

3.2 Memory Module with Write-back Cache Implementation

The memory module specification may now be used to create an implementation; particularly, one that uses a write-back cache. (one where you write to memory, it goes to a cache first, doesn't necessarily get written to main memory, the cache may differ and if it does it's more recent) The implementation (Example 5 on the handout) requires that the number of entries defined in the cache (effectively, the size of the cache) be a built-in constant; for the purposes of this example, we will assume that size (`CSIZE`) is 2. The implementation basically says:

1. if the data corresponding to the requested memory address is in the cache, return that value;

[6] Actually, it can be shown that no reachable state ever has a value of "nothing"; a valid initial state has all locations full, and no function exists for "erasing" a location.

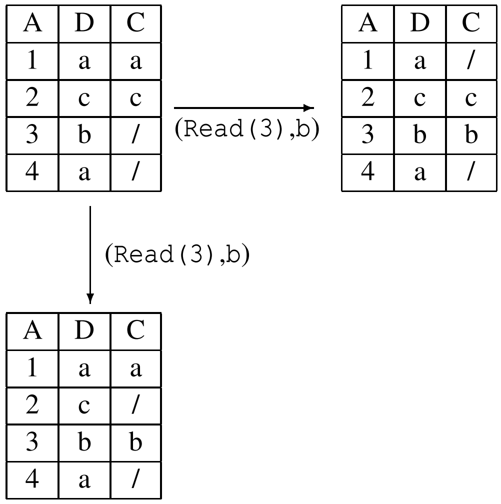

Figure 2: Non-deterministic state transition of `Read`.

2. otherwise move the value from main memory to the cache and return it.

Similarly, the cache is written to first, then this value is moved into memory when the cache needs to be flushed.

See also Figure 2 of Handout 8. Convince yourself that Example 5 is a correct implementation of Example 4. Why? Because if you only use the four routines provided by the spec, you'll never know about the caching.

How do we convince ourselves this is true systematically? Using traces. So our new principle is:

$$\mathit{Traces}_I \subseteq \mathit{Traces}_S$$

Again, the specification is a list of *allowable* things. But because traces are arbitrarily long and infinite in number, we have to use inductive proofs. . .

Because it would be impossible to enumerate all sequences, proofs are done inductively. The *state machine* is a useful abstraction. For the memory example, suppose that `A` and `D` are defined as before, and we have a two-location cache. The state of memory prior to initialization could also be undefined. It is interesting to note that there is non-determinism in the system; for example, the new state resulting from reading a value from a memory address that is not cached. There are several possibilities as to which cache item is actually flushed. All of these instances must be restricted in order to have a correct implementation.[7]

Note that the Read procedure can modify the state of the Write-Back Cache; it may flush an entry from the cache and replace it with another. But as long as callers can use only the abstract interface (Read, Reset, Write, Swap) they are unable to tell whether the implementation is Simple Memory or Write-Back Cache Memory.

[7] Correctness: another lecture.

One way of viewing these state machines "abstractly" is simply as a set of traces. These traces describe the behavior of the interface between the user and the module. The user has no knowledge about how these results are generated and cannot access the internal state of the module. A sample trace for the above example might be:

Invocation/Response	Description
`(Reset(c),RET)`	Reset all values to `c` (returns nothing)
`(Read(1),c)`	Read location `1` (returns `c`)
`(Write(2,b),RET)`	Write `b` to location `2` (returns nothing)
`(Read(2),b)`	Read location `2` (returns `b`)

This specification is deterministic in that, given a sequence of invocations, there is a single sequence of responses that is acceptable. This is *not* true in the general case—there may often be multiple responses. So, the specification can be thought of as a relation of invocation sequences to response sequence. In real life, this will be executed as one invocation at a time, but abstractly it is possible to model it as giving the module a sequence of invocations and looking at the sequence of return values.

So, if we consider a module as just a set of traces of input sequences to output sequences, it is very abstract. This is useful when we think about *correctness*—our abstract view of our module, and exactly what the specification denotes—which is also the information provided to a client (*i.e.*, what the client can depend on). The text of the implementation can be abstracted in the same way to a set of traces that the implementation will generate.

3.3 Proofs using Traces

The trace view is the only public view of the module. It is always possible to take an example trace and answer the question "is this a valid trace of the module?"

Note that the traces of a correct implementation are a subset of the traces of the specification. How can we convince ourselves that all possible traces of an implementation are traces of a specification? Possible traces are arbitrarily long and there are an infinite number of them.

We can use induction and case analysis. We return to the view of the module as a state machine, and define two functions on the machine.

- Abstraction Function
- Representation (Rep) Invariant.

An *abstraction function* maps states of an implementation to states of a specification. The states of an implementation correspond to those of the specification; the abstraction function defines the correspondence. (Note that several implementation states can map to a single specification state.) See Figure 3 in Handout 8.

A *rep. invariant* is a predicate on the state of the implementation. It defines the set of reachable, valid states. A rep invariant captures properties of the procedures in a module that other procedures in the module depend upon. A rep. invariant can be violated only while moving from one state to another (while "between states"), but at the interface it is never violated.

3.3.1 Abstraction Functions

An abstraction function shows you how to think of the implementation as if it were the specification. The function is often many implementation states to one specification state.

The goal is to prove that every implementation trace is a valid specification trace. We do so by first showing the abstraction function is a homomorphism on state machines. Our eventual goal is to show that the implementation is simulating the specification.

You need to be able to show two things:

1. If there exists a transition T from state S_1 to S_2 in the implementation, then T can take[8] $AF(S_1)$ to $AF(S_2)$ in the specification.

   AF may map S_1 and S_2 to the same state in the specification. For example, the Cache Memory Read procedure can modify the state of the module, where in the Simple Memory example it does not.

2. $AF(Init_{impl}) \subseteq Init_{spec}$; every initial state of the implementation is an initial state of the specification.

With these two, we can show inductively that the specification is simulated by the implementation.

3.3.2 Rep. Invariants

However, for some implementations, we need more than the abstraction function to show correctness. We also need some representation invariant to limit the domain of the abstraction function to sensible states of the implementation. The example given in lecture was of an implementation of a memory using a hash table. The hash table lookup fails if a location is stored in the wrong hash bucket. An invariant function for a hash table implementation would be true only for states in which all addresses are in the correct bucket since it is known that the store operation for a hash table implementation will never store an address in the wrong bucket. The invariant simplifies the correctness proof by limiting the implementation states that must be considered.[9]

for its address, or more than one value for a given address. If either

[8] T may be non-deterministic

[9] This paragraph was based on the Lecture 3 notes written by Ken Mackenzie in Fall, 1992.