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

Lecture 2: September 16, 1992

Lecturer: Bill Weihl *Scribe: Keith M. Swartz*

1 Administrative Note

Bring your old handouts to class, particularly those handed out in the last lecture.

2 Examples of Specifications and Implementations

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.

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*.[1] 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:

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

Inputs: seq int $\times$ int
Output: int $\cup$ {NotFound}

The relation itself is:

$$a, x \mapsto \{i | a[i] = x\} \cup \{\text{NotFound} | \nexists i \text{ s.t. } a[i] = x\}$$

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;
```

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`.

Here, the implementation represents a different input-output relation than the specification, namely,

$$a, x \mapsto \{\text{least } i | 0 \leq i < a.size \wedge a[i] = x\} \cup \{\text{NotFound} | \nexists i \text{ s.t. } a[i] = x\}$$

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.

2.3 Binary Search Implementation

The example above can be easily modified to allow for different details, such as the search method used. For instance, the user may wish to use a binary search. In such a case, however, the array must be sorted before the search can be carried out. In order to indicate this, the following must be added to the original specification:

```
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.

Another faulty specification would be:

```
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`.

3 Data Abstractions

Data abstraction allows us to encapsulate multiple operations as well as state. Abstraction removes 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.

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 state 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. 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.) 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.

There are five procedures in this example:

- Init (chooses an *arbitrary* TOTAL mapping, so that `m` is defined over all addresses).
- Reset (reset state of memory with value `d`, e.g.: for zeroing out space).[2]
- Read (returns the current state of the memory for the given address `a`).
- Write (writes value of `d` to address `a`).
- Swap (swaps the value of address `a` with the new value `d` and returns the old one).

It is required that `Init` be called before any of the other routines. Spec does not have a formal method of stating such requirements, so we simply list it as a comment. 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`.

This a simple example of data abstraction. For *procedures* we use relations between input and output in which the input and output may include state. 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.

A *transition* between states is labelled 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

[2]Introduces the mapping constructor `* ->`, which appears in the form `S{ * -> q}`, and maps all elements of sequence `S` to the value `q`.

values D is {a,b,c}, any single state may be represented as four values, each being one of a, b, c, or nothing.[3] Below are illustrations of a few possible transitions:

A	D		A	D		A	D
1	a		1	b		1	/
2	c	$\longrightarrow$	2	b	$\longleftarrow$	2	/
3	b	(Reset(b),RET)	3	b	(Reset(b),RET)	3	/
4	a		4	b		4	/

(Read(3),b)

Note how the resulting state is unchanged for operations without side effect (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. The implementation 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
2. otherwise return the value value stored in the main memory

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

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 state returned when a value is read from a memory address that is not cached. There are several possibilities as to which cache item is actually flushed. (See Figure ¿.) All of these instances must be restricted in order to have a correct implementation.[4]

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 set of traces for the above example might be:

[3] 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.

[4] Correctness: next lecture.

A	D	C
1	a	a
2	c	c
3	b	/
4	a	/

$\xrightarrow{\text{(Read(3),b)}}$

A	D	C
1	a	/
2	c	c
3	b	b
4	a	/

$\downarrow$ (Read(3),b)

A	D	C
1	a	a
2	c	/
3	b	b
4	a	/

Figure 1: Non-deterministic state transition of `Read`.

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 set of invocations, there is a single set 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 set 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.

Acknowledgements to Carl G. Heinzl for the excerpts in sections 3.1 and 3.2 that are from his notes last year.