5. Examples of Specs and Code

This handout is a supplement for the first two lectures. It contains several example specs and code, all written using Spec.

Section 1 contains a spec for sorting a sequence. Section 2 contains two specs and one code for searching for an element in a sequence. Section 3 contains specs for a read/write memory. Sections 4 and 5 contain code for a read/write memory based on caching and hashing, respectively. Finally, Section 6 contains code based on replicated copies.

1. Sorting

The following spec describes the behavior required of a program that sorts sets of some type \( T \) with a "\(<\)" comparison method. We do not assume that "\(<\)" is antisymmetric; in other words, we can have \( t_1 \leq t_2 \) and \( t_2 \leq t_1 \) without having \( t_1 = t_2 \), so that "\(<\)" is not enough to distinguish values of \( T \). For instance, \( T \) might be the record type \([\text{name: String}, \text{salary: Int}]\) with "\(<\)" comparison of the \( \text{salary} \) field. Several \( T \)’s can have different \( \text{names} \) but the same \( \text{salary} \).

\[
\text{TYPE } S = \text{SET } T \\
\text{Q = SEQ } T \\
\text{APROC } \text{Sort}(s) \rightarrow \text{Q} = \langle \text{VAR q | (ALL t | s.count(t) = q.count(t))} \& \& \text{Sorted}(q) \Rightarrow \text{RET q} >
\]

This spec uses the auxiliary function \( \text{Sorted} \), defined as follows.

\[
\text{FUNC } \text{Sorted}(q) \rightarrow \text{Bool} = \text{RET (ALL i :IN q.dom – {0} | q(i-1) \leq q(i))}
\]

If we made \( \text{Sort} \) a \( \text{FUNC} \) rather than a \( \text{PROC} \), what would be wrong? What could we change to make it a \( \text{FUNC} \)?

We could have written this more concisely as

\[
\text{APROC } \text{Sort}(s) \rightarrow \text{Q} = \langle \text{VAR q :IN a.perms | Sorted(q) \Rightarrow \text{RET q} >}
\]

using the \( \text{perms} \) method for sets that returns a set of sequences that contains all the possible permutations of the set.

2. Searching

Search spec

We begin with a spec for a procedure to search an array for a given element. Again, this is an \( \text{APROC} \) rather than a \( \text{FUNC} \) because there can be several allowable results for the same inputs.

\[
\text{APROC } \text{Search}(q, t) \rightarrow \text{Int RAISES } \{\text{NotFound}\} = \langle \text{IF } \text{VAR i :IN q.dom | q(i) = t} \Rightarrow \text{RET i} \rangle
\]

Or, equivalently but slightly more concisely, and highlighting the changes with boxes:

\[
\text{APROC } \text{Search}(q, t) \rightarrow \text{Int RAISES } \{\text{NotFound}\} = \langle \text{IF VAR i :IN q.dom | q(i) = t} \Rightarrow \text{RET i} \rangle
\]

Sequential search code

Here is code for the \( \text{Search} \) spec given above. It uses sequential search, starting at the first element of the input sequence.

\[
\text{APROC } \text{SeqSearch}(q, t) \rightarrow \text{Int RAISES } \{\text{NotFound}\} = \langle \text{VAR i := 0 | DO i < q.size => IF q(i) = t} \Rightarrow \text{RET i} \rangle
\]

Alternative search spec

Some searching algorithms, for example, binary search, assume that the input argument sequence is sorted. Such algorithms require a different spec, one that expresses this requirement.

\[
\text{APROC } \text{Search1}(q, t) \rightarrow \text{Int RAISES } \{\text{NotFound}\} = \langle \text{IF } \text{~Sorted}(q)} \Rightarrow \text{HAVOC} \rangle
\]

Alternatively, the requirement could go in the type of the \( q \) argument:

\[
\text{APROC } \text{Search1}(q: \text{Q SUCHTHAT Sorted(this)}, t) \rightarrow \text{Int RAISES } \{\text{NotFound}\} = \langle ...angle
\]

This is farther from code, since proving that a sequence is sorted is likely to be too hard for the code’s compiler.

You might consider writing the spec to raise an exception when the array is not sorted:

\[
\text{APROC } \text{Search2}(q, t) \rightarrow \text{Int RAISES } \{\text{NotFound}, \text{NotSorted}\} = \langle \text{IF } \text{~Sorted(q)} \Rightarrow \text{RAISE NotSorted} \rangle
\]

This is not a good idea. The whole point of binary search is to obtain \( O(\log n) \) time performance (for a sorted input sequence). But any code for the \( \text{Search2} \) spec requires an \( O(n) \) check, even for a sorted input sequence, in order to verify that the input sequence is in fact sorted.

This is a simple but instructive example of the difference between defensive programming and efficiency. If \( \text{Search} \) were part of an operating system interface, it would be intolerable to have \( \text{HAVOC} \) as a possible transition, because the operating system is not supposed to go off the deep
end no matter how it is called (though it might be OK to return the wrong answer if the input isn’t sorted; what would that spec be?). On the other hand, the efficiency of a program often depends on assumptions that one part of it makes about another, and it’s appropriate to express such an assumption in a spec by saying that you get \texttt{HAVOC} if it is violated. We don’t care to be more specific about what happens because we intend to ensure that it doesn’t happen. Obviously a program written in this style will be more prone to undetected or obscure errors than one that checks the assumptions, as well as more efficient.

3. Read/write memory

The simplest form of read/write memory is a single read/write register, say of type \(V\) (for value), with arbitrary initial value. The following Spec module describes this (a lot of boilerplate for a simple variable, but we can extend it in many interesting ways):

\[
\text{MODULE Register} \ [V] \ \text{EXPORT} \ \text{Read, Write} = \\
\text{VAR} \ m: V \quad \% \text{arbitrary initial value} \\
\text{APROC} \ \text{Read}() \to V = \langle \text{RET} \ m \rangle \\
\text{APROC} \ \text{Write}(m) = \langle \text{m := v} \rangle \\
\text{END Register}
\]

Now we give a spec for a simple addressable memory with elements of type \(V\). This is like a collection of read/write registers, one for each address in a set \(A\). In other words, it’s a function from addresses to data values. For variety, we include new \text{Reset} and \text{Swap} operations in addition to \text{Read} and \text{Write}.

\[
\text{MODULE Memory} \ [A, V] \ \text{EXPORT} \ \text{Read, Write, Reset, Swap} = \\
\text{TYPE M} = A \to V \\
\text{VAR} \ m := \text{InitM}() \\
\text{APROC} \ \text{InitM}() \to M = \langle \text{VAR} \ m' | \ (\text{ALL} \ a | m'!a) \Rightarrow \text{RET} \ m' \rangle \\
\% \text{Returns a M with arbitrary values.} \\
\text{FUNC} \ \text{InitC}() \to C = \langle \text{VAR} \ c' | c'.\text{dom.size} = \text{CSize} \Rightarrow \text{RET} \ c' \rangle \\
\% \text{Returns a C that has exactly CSize entries defined, with arbitrary values.} \\
\text{APROC} \ \text{Read}(a) \to V = \langle \text{Load(a)}; \text{RET} \ c(a) \rangle \\
\text{APROC} \ \text{Write}(a, v) = \langle \text{IF} \sim c!a \Rightarrow \text{FlushOne(); SKIP FI; c(a) := v} \rangle \\
\% \text{Makes room in the cache if necessary, then writes to the cache.} \\
\text{APROC} \ \text{Reset}(v) = \langle \ldots \rangle \\
\% \text{exercise for the reader} \\
\text{APROC} \ \text{Swap}(a, v) = \langle \ldots \rangle \\
\% \text{override the function m with the function c wherever c is defined.}
\]

The next three sections describe code for Memory.

4. Write-back cache code

Our first code is based on two memory mappings, a main memory \(m\) and a write-back cache \(c\). The code maintains the invariant that the number of addresses at which \(c\) is defined is constant. A real cache would probably maintain a weaker invariant, perhaps bounding the number of addresses at which \(c\) is defined.

\[
\text{MODULE WBCache} \ [A, V] \ \text{EXPORT} \ \text{Read, Write, Reset, Swap} = \\
\text{% implements Memory} \\
\text{TYPE M} = A \to V \\
\text{C} = A \to V \\
\text{CONST Csize} : \text{Int} := \ldots \\
\% \text{cache size} \\
\text{VAR} \ m := \text{InitM}() \\
\text{VAR} \ c := \text{InitC()} \\
\text{APROC} \ \text{InitM}() \to M = \langle \text{VAR} \ m' | (\text{ALL} \ a | m'!a) \Rightarrow \text{RET} \ m' \rangle \\
\% \text{exercise for the reader} \\
\text{APROC} \ \text{InitC}() \to C = \langle \text{VAR} \ c' | c'.\text{dom.size} = \text{CSize} \Rightarrow \text{RET} \ c' \rangle \\
\% \text{exercise for the reader} \\
\text{APROC} \ \text{Load}(a) = \langle \ldots \rangle \\
\% \text{Exercise for the reader} \\
\text{APROC} \ \text{FlushOne(); SKIP FI; c(a) := v} \\
\% \text{Exercise for the reader} \\
\text{APROC} \ \text{Dirty(a)} \to \text{Bool} = \langle \ldots \rangle \\
\% \text{Exercise for the reader} \\
\text{END WBCache}
\]

The following Spec function is an abstraction function mapping a state of the WBCache module to a state of the Memory module. Unlike our usual practice we have written it explicitly as a function from the state of HashMemory to the state of Memory. It says that the contents of location \(a\) is \(c(a)\) if \(a\) is in the cache, and \(m(a)\) otherwise.

\[
\text{FUNC} \ \text{AF}(m, c) \to M = \text{RET} (\langle \text{IF} \sim c!a \Rightarrow \text{FlushOne(); c(a) := m(a) [*] SKIP FI} \rangle) \\
\% \text{Exercise for the reader} \\
\text{FUN} \ \text{AF}(c, m) \to M = \text{RET} \ m + c \\
\% \text{Exercise for the reader}
\]

That is, override the function \(m\) with the function \(c\) wherever \(c\) is defined.
5. Hash table code

Our second code for Memory uses a hash table for the representation. It is different enough from the spec that it wouldn’t be helpful to highlight the changes.

**MODULE HashMemory** [A WITH {hf: A->Int}, V] EXPORT Read, Write, Reset, Swap = % Implements Memory. Expects that the hash function A.hf is total and that its range is 0 .. n for some n.

```plaintext
TYPE Pair = [a, v] % Bucket in hash table
B = SET Pair % Bucket in hash table
HashT = SEQ B % Bucket in hash table

VAR m: HashT := {i :IN 1 .. nb | {}} % Memory hash table; initially empty
default : V % default value, initially arbitrary

APROC Read(a) -> V = << VAR p :IN m(a.hf) | p.a = a => RET p.v [*] RET default >>

APROC Write(a, v) = << VAR b := m(a.hf) |
IF VAR p :IN b | p.a = a => b := b – {p} [*] SKIP FI;

m(a.hf) := b / {Pair{a, v}} >> % and add the new pair

APROC Reset(v) = << m := {i :IN 1 .. nb | {}}; default := v >>

APROC Swap(a, v) -> V = << VAR v' | v' := Read(a); Write(a, v); RET v' >> END HashMemory
```

The following is a key invariant that holds between invocations of the operations of HashMemory:

```
FUNC Inv(m: HashT, nb: Int) -> Bool = RET
{ m.size = nb %
  (ALL i :IN m.dom, p :IN m(i) | p.a.hf = i)
  (ALL a | (p :IN m(a.hf) | p.a = a ).size <= 1) )
```

This says that the hash function maps all addresses to actual buckets, that a pair containing address a appears only in the bucket at index a.hf in m, and that at most one pair for an address appears in the bucket for that address. Note that these conditions imply that in any reachable state of HashMemory, each address appears in at most one pair in the entire memory.

The following Spec function is an abstraction function between states of the HashMemory module and states of the Memory module.

```
FUNC AF(m: HashT, default) -> M = RET
{ LAMBDA(a) -> V =
  IF VAR i :IN m.dom, p :IN m(i) | p.a = a => RET p.v [*] RET default FI
```

That is, the data value for address a is any value associated with address a in the hash table; if there is none, the data value is the default value. Spec says that a function is undefined at an argument if its body can yield more than one result value. The invariants given above ensure that the LAMBDA is actually single-valued for all the reachable states of HashMemory.
6. Replicated memory

Our final code is based on some number \( k \geq 1 \) of copies of each memory location. Initially, all copies have the same default value. A write operation only modifies an arbitrary majority of the copies. A read reads an arbitrary majority, and selects and returns the most recent of the values it sees. In order to allow the read to determine which value is the most recent, each write records not only its value, but also a sequence number. The crucial property of a majority is that any two majorities have a non-empty intersection; this ensures that a read will see at least one copy written by the most recent write.

For simplicity, we just show the module for a single read/write register. The constant \( k \) determines the number of copies.

\[ \text{MODULE MajReg} \]

\[ [V] \quad \text{\% implements Register} \]

\[ \text{CONST} \quad k = 5 \quad \text{\% 5 copies} \]

\[ \text{TYPE N} \quad \text{\% copies, ints between 1 and k} \]

\[ C = \text{IN} \ 1 \ldots \ k \]

\[ \text{Maj} = \text{SET} \ C \ \text{SUCHTHAT} \ \text{maj.size} > k/2 \quad \text{\% all majority subsets of C} \]

\[ \text{TYPE P} = [v, n] \ \text{WITH} \ \{"\leq":\text{PLEq}\} \quad \text{\% Pair of value and sequence number} \]

\[ M = C \rightarrow P \quad \text{\% Memory (really register) copies} \]

\[ S = \text{SET} \ P \]

\[ \text{VAR default} : V \quad \text{\% arbitrary initial value} \]

\[ m := M[* \rightarrow P{v := \text{default}, n := 0}] \]

\[ \text{APROC Read()} \rightarrow V = \langle \text{RET ReadPair().v} \rangle \]

\[ \text{APROC Write(v)} = \langle \text{VAR n := ReadPair().n, maj} | \%
\text{Determines the highest sequence number n, then writes v paired with n+1 to some majority maj of the copies.} \%
\text{n := n + 1;} \%
\text{DO VAR j :IN maj | m(j).n \# n => m(j) := \{v, n\} OD} \rangle \%
\text{\% Internal procedures.} \]

\[ \text{APROC ReadPair()} \rightarrow P = \langle \text{RET ReadMaj() \rangle} \%
\text{\% Returns a pair with the largest sequence number from some majority of the copies.} \%
\text{VAR p :IN s | p.n = s.max.n \rightarrow RET p} \rangle \%
\text{APROC ReadMaj()} \rightarrow S = \langle \text{RET maj * m} \rangle \%
\text{\% Returns the set of pairs belonging to some majority of the copies. maj * m is \{c :IN maj | \ | m(c)\}} \%
\text{FUNC PLeq(p1, p2) \rightarrow Bool = RET p1.n \leq p2.n} \%
\text{END MajReg} \]

The following is a key invariant for MajReg.

\[ \text{FUNC Inv(m)} \rightarrow \text{Bool = RET} \%
\text{\{ALL p :IN m.rng, p' :IN m.rng | p.n = p'.n \rightarrow p.v = p'.v\}} \%
\text{\{/ \text{EXISTS maj | m.rng.max \leq \{maj * m\}.min\}}} \%
\text{The first conjunct says that any two pairs with the same sequence number also have the same data. The second says that for some majority of the copies every pair has the highest sequence number.} \]