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 `t1 <= t2` and `t2 <= t1` without having `t1 = t2`, so that `"<="` is not enough to distinguish values of `T`. For instance, `T` might be the record type `[name:String, salary: Int]` with `"<="` comparison of the `salary` field. Several `T`'s can have different `name`s but the same `salary`.

```
TYPE S = SET T
     Q = SEQ T

APROC Sort(s) -> Q = <<
     VAR q | (ALL t | s.count(t) = q.count(t)) /\ Sorted(q) => RET q >>
```

This spec uses the auxiliary function `Sorted`, defined as follows.

```
FUNC Sorted(q) -> Bool = RET (ALL i :IN q.dom – {0} |  q(i-1) <= q(i))
```

If we made `Sort` a `FUNC` rather than a `PROC`, what would be wrong?[1] What could we change to make it a `FUNC`?

We could have written this more concisely as

```
APROC Sort(s) -> Q =
     << VAR q :IN a.perms | Sorted(q) => RET q >>
```

using the `perms` method for sets that returns a set of sequences that contains all the possible permutations of the set.

[1] Hint: a `FUNC` can't have side effects and must be deterministic (return the same value for the same arguments).

2. Searching

Search spec

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

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

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

```
APROC Search(q, t) -> Int RAISES {NotFound} =
     << IF VAR i :IN q.dom | q(i) = t => RET i [*] RAISE NotFound FI >>
```

Sequential search code

Here is code for the `Search` spec given above. It uses sequential search, starting at the first element of the input sequence.

```
APROC SeqSearch(q, t) -> Int RAISES {NotFound} = << VAR i := 0 |
     DO i < q.size => IF q(i) = t => RET i [*] i + := 1 FI OD; RAISE NotFound >>
```

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.

```
APROC Search1(q, t) -> Int RAISES {NotFound} = <<
     IF   ~Sorted(q) => HAVOC
     [*]  VAR i :IN q.dom | q(i) = t => RET i
     [*]  RAISE NotFound
     FI >>
```

Alternatively, the requirement could go in the type of the `q` argument:

```
APROC Search1(q: Q SUCHTHAT Sorted(this), t) -> Int RAISES {NotFound} = <<
     ... >>
```

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:

```
APROC Search2(q, t) -> Int RAISES {NotFound, NotSorted} = <<
     IF   ~Sorted(q) => RAISE NotSorted
     ...
```

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 `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 `Search` were part of an operating system interface, it would be intolerable to have `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 `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):

```
MODULE Register [V] EXPORT Read, Write =

VAR m: V                                         % arbitrary initial value

APROC Read() -> V = << RET m  >>
APROC Write(m)    = << m := v >>

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 `Reset` and `Swap` operations in addition to `Read` and `Write`.

```
MODULE Memory [A, V] EXPORT Read, Write, Reset, Swap =

TYPE M =  A -> V
VAR  m := Init()

APROC Init() -> M  = << VAR m' | (ALL a | m'!a) => RET m' >>
% Choose an arbitrary function that is defined everywhere.

FUNC  Read(a) -> V = << RET m(a)  >>
APROC Write(a, v)  = << m(a) := v >>

APROC Reset(v)     = << m := M{* -> v} >>
% Set all memory locations to v.

APROC Swap(a, v) -> V = << VAR v' := m(a) | m(a) := v; RET v' >>
% Set location a to the input value and return the previous value.

END Memory
```

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.

```
MODULE WBCache [A, V] EXPORT Read, Write, Reset, Swap =
% implements Memory

TYPE M          =  A -> V
     C          =  A -> V

CONST Csize     :  Int := ...                    % cache size

VAR  m          := InitM()
     c          := InitC()

APROC InitM() -> M = << VAR m' | (ALL a | m'!a)     => RET m' >>
% Returns a M with arbitrary values.

APROC InitC() -> C = << VAR c' | c'.dom.size = CSize => RET c' >>
% Returns a C that has exactly CSize entries defined, with arbitrary values.

APROC Read(a) -> V = << Load(a); RET c(a) >>
APROC Write(a, v) = << IF ~c!a => FlushOne() [*] SKIP FI; c(a) := v >>
% Makes room in the cache if necessary, then writes to the cache.

APROC Reset(v) = <<...>>                         % exercise for the reader

APROC Swap(a, v) -> V = << Load(a); VAR v' | v' := c(a); c(a) := v; RET v' >>

% Internal procedures.

APROC Load(a) = << IF ~c!a => FlushOne(); c(a) := m(a) [*] SKIP FI >>
% Ensures that address a appears in the cache.

APROC FlushOne() =
% Removes one (arbitrary) address from the cache, writing the data value back to main memory if necessary.
    << VAR a | c!a => IF Dirty(a) => m(a) := c(a) [*] SKIP FI; c := c{a -> } >>

FUNC Dirty(a) -> Bool = RET c!a /\ c(a) # m(a)
% Returns true if the cache is more up-to-date than the main memory.

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.

```
FUNC AF(m, c) -> M = RET (\ a | c!a => c(a) [*] m(a) )
```

We could have written this more concisely as

```
FUNC AF(m, c) -> M = RET m + c
```

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.

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

CONST nb        := A.hf.rng.max                          % Number of buckets

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}          % remove a pair with a from b
     [*]  SKIP FI;                                       % do nothing if there isn't one
     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`.

Of course `HashMemory` is not fully detailed code. In particular, it just treats the variable-length buckets as sets and doesn't explain how to maintain them, which is usually done with a linked list. However, the code does capture all the essential details.

6. Replicated memory

Our final code is based on some number `k ≥ 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.

```
MODULE MajReg [V] =                                 % implements Register

CONST k        =  5                                 % 5 copies

TYPE N         =  Nat
     C         =  IN 1 .. k                         % copies, ints between 1 and k
     Maj       =  SET C SUCHTHAT maj.size > k/2     % all majority subsets of C

TYPE P         =  [v, n] WITH {"<=":=PLEq}          % Pair of value and sequence number
     M         =  C -> P                            % Memory (really register) copies
     S         =  SET P

VAR  default   :  V                                 % arbitrary initial value
     m         := M{* -> P{v := default, n := 0}}

APROC Read() -> V = << RET ReadPair().v >>

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

% Internal procedures.

APROC ReadPair() -> P = << VAR s := ReadMaj() |
% Returns a pair with the largest sequence number from some majority of the copies.
     VAR p :IN s | p.n = s.max.n => RET p >>

APROC ReadMaj () -> S = << VAR maj | RET maj * m >>
% Returns the set of pairs belonging to some majority of the copies. maj * m  is {c :IN maj | | m(c)}

FUNC PLeq(p1, p2) -> Bool = RET p1.n <= p2.n

END MajReg
```

The following is a key invariant for `MajReg`.

```
FUNC Inv(m) -> Bool = RET
       (ALL p :IN m.rng, p' :IN m.rng | p.n = p'.n ==> p.v = p'.v)
     /\ (EXISTS maj | m.rng.max <= (maj * m).min)
```

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.

The following Spec function is an abstraction function. It says that the abstract register data value is the data component of a copy with the highest sequence number. Again, because of the invariants, there is only one `p.v` that will be returned.

```
FUNC AF(m) -> V = RET m.rng.max.v
```

We could have written the body of `ReadPair` as

```
    << VAR s := ReadMaj() | RET s.max >>
```

except that `max` always returns the same maximal `p` from the same `s`, whereas the `VAR` in `ReadPair` chooses one non-deterministically.