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. Sections 4 and 5 contain code 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 \( \leq \) comparison method. We do not assume that \( \leq \) 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 \( \leq \) is not enough to distinguish values of \( T \). For instance, \( T \) might be the record type \([\text{name}:\text{String}, \text{salary}: \text{Int}]\) with \( \leq \) comparison of the salary field. Several \( T \)'s can have different \text{name}s but the same \text{salary}.

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

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

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

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

We could have written this more concisely as

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

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 APROC rather than a FUNC because there can be several allowable results for the same inputs.

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

```spec
module Register [V]
export Read, Write =
var m: V % arbitrary initial value
proc Read() -> V = << ret m >>
proc 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`.

```spec
module Memory [A, V]
export Read, Write, Reset, Swap =
type M = A -> V
var m := Init()
proc 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) >>
proc Write(a, v) = << if ~c!a => flushOne() [*] skip fi; c(a) := v >> % Makes room in the cache if necessary, then writes to the cache.
proc Reset(v) = <<...>> % exercise for the reader
proc Swap(a, v) -> V = << var v' | load(a); v' := c(a); c(a) := v; ret v' >>
end 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.

```spec
module WBCache [A, V]
export Read, Write, Reset, Swap =
type M = A -> V
c = A -> V
const Csize : int := ...
var m := InitM()
c := InitC()
proc InitM() -> M = << var m' | (all a | m'!a) => ret m' >> % Returns a \( M \) with arbitrary values.
proc InitC() -> C = << var c' | c'.dom.size = CSize => ret c' >> % Returns a \( C \) that has exactly \( CSize \) entries defined, with arbitrary values.
proc Read(a) -> V = << load(a); ret c(a) >>
proc Write(a, v) = << if ~c!a => flushOne(); c(a) := m(a) [*] skip fi; c(a) := v >> % Ensures that address \( a \) appears in the cache.
proc Reset(v) = <<...>> % exercise for the reader
proc Swap(a, v) -> V = << var v' | load(a); v' := c(a); c(a) := v; ret v' >>
end WBCache
```

The following `Spec` function is an abstraction function mapping a state of the `WBCache` module to a state of the `Memory` module. It's written to live inside the module. It says that the contents of location \( a \) is \( c(a) \) if \( a \) is in the cache, and \( m(a) \) otherwise.

```spec
func AF() -> M = ret (all a | c!a => c(a) [*] m(a))
```
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.
% The module expects that the hash function A.hf is total and that its range is 0 .. n for some n.

**5.1 Hash table code**

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

```plaintext
FUNC AF() -> M = RET
(LAMBDA(a) -> V =
  IF VAR i :IN m.dom, p :IN m(i).rng | 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. Its main deficiency is that it doesn’t explain how to maintain the variable-length bucket sequences, 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 to a memory location determines the number of copies.

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

```plaintext
CONST k = 5

TYPE N = Nat
C = IN 1 .. k
Maj = SET C
S = SET P

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

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

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.
% 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 { c :IN maj | | m(c) } >> % Returns the set of pairs belonging to some majority of the copies.
FUNC PLeq(p1, p2) = RET p1.n <= p2.n
END MajReg

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.

The following is a key invariant for MajReg.

FUNC Inv(m: M) -> Bool = RET
    (ALL p :IN m.rng, p' :IN m.rng | p.n = p'.n ==> p.v = p'.v)
    /
    (EXISTS maj | (ALL c :IN maj, p :IN m.rng | m(c).n >= p.n)))

The first conjunct says that any two pairs having the same sequence number also have the same
data. The second conjunct says that the highest sequence number appears in some majority of the
copies.

The following Spec function is an abstraction function between states of the MajReg module and
states of the Register module.

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

That is, 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.