Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 5 September 13, 1993

Example Specifications and Implementations

This handout is a supplement for the first two lectures. It contains several example specifications of modules as well as implementations.

1 Search Specification

We begin with a specification for a procedure to search 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 >>
```

2 Search Implementation

Here is an implementation of the specification for search given above.

```
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
   >>
```

3 Alternative Search Specification

If we want to use a faster implementation for search, for example based on binary search, we need a different specification that ensures that the argument sequence is sorted. We do this by putting a requirement on the caller:

```
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 >>

% Return true if and only if the elements of a are in ascending order.
FUNC Sorted (a: SEQ INT) -> BOOL =
   VAR i: INT |
       (0 <= i /\ (i < a.size - 1) /\ a[i] > a[i+1]) => RET false
   [*] RET true
```

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

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

Why is this a bad idea?

4 Memory Specification

Here is a specification for a simple addressable memory.

```
MODULE SimpleMemory [ A, D ] =
      % A is addresses, D is data

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

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

APROC Reset (d) =
    << m := M { * -> d } >>

APROC Read (a) -> (D) =
    << RET m(a) >>

APROC Write (a, d) =
    << m(a) := d >>
```

```
APROC Swap (a, d) -> (D) =
    << VAR d' | d' := m(a); m(a) := d; RET d' >>

END SimpleMemory
```

5 Write-back Cache Implementation

Now let's consider an implementation that uses a write-back cache. An abstraction function is given after the implementation. The implementation maintains the invariant that the number of locations at which c is defined is constant; for a hardware cache the motivation for this should be clear. A real cache would maintain a more complex invariant, perhaps bounding the number of addresses at which c is defined.

```
MODULE WBCache [ A, D ] =

TYPE M = A -> D
     C = A -> D

VAR CSIZE: INT := ...    % number of entries in cache; constant

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

APROC InitM () -> M =
   << VAR m' | (ALL a | m'!a) => RET m' >>

APROC InitC () -> C =
   << c := M { * -> };
      % Initialize exactly CSIZE addresses in c.
      DO VAR a | ~c!a /\ (Set[A].Size({a' | c!a'}) < CSIZE)
           => VAR d | c(a) := d
      OD >>

FUNC Dirty (a) -> (BOOL) =
    RET c!a /\ c(a)#m(a)

APROC FlushOne () =   % Remove one location from the cache.
   << VAR a | c!a =>
            BEGIN Dirty(a) => m(a) := c(a) [*] SKIP END;
            c := c{a -> } >>

APROC Load (a) =    % Make sure a appears in the cache.
```

```
   << ~c!a => FlushOne(); c(a) := m(a)
      [*] SKIP >>

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

APROC Read (a) -> (D) =
   << Load(a); RET c(a) >>

APROC Write (a, d) =
   << BEGIN ~c!a => FlushOne() [*] SKIP END;
      c(a) := d >>

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

END WBCache
```

The abstraction function maps a state of the WBCache module to a state of the SimpleMemory module; it is given by the following Spec function:

```
FUNC AF (m, c, CSIZE: INT) -> (A->D) =
    RET ( LAMBDA (a) -> D =
              c!a => RET c(a) [*] RET m(a) )
```

6 Hash Table Implementation

This section contains an alternative implementation of SimpleMemory that uses a hash table for the representation. It illustrates the use of rep invariants in correctness proofs.

```
MODULE HashMemory [ A WITH {hf: A->INT}, D ] =

% The range of the method A.hf should be between 0 and
% some maximum; negative values are not allowed.

TYPE HashT = SEQ Bucket
     Bucket = SEQ Pair
     Pair = [addr: A, data: D]

VAR m: HashT := Sequence[Bucket].Fill(Bucket{},sz)
    init: D                 % initial value (arbitrary)
    sz: INT := InitSz()     % number of buckets
```

```
% Check to ensure that hash function has valid range,
% and return number of buckets needed if range is okay;
% otherwise signal.
FUNC InitSz () -> INT RAISES {BadHF} =
   VAR lw: INT, hi: INT |
     lw = {a: INT | true | a.hf}.min
    /\ hi = {a: INT | true | a.hf}.max
    /\ lw >= 0
      => RET hi+1
   [*] RAISE BadHF

% If a appears in a pair in the bucket, returns the index where
% it appears; otherwise raises NotFound.  Assumes a appears in at
% most one pair (otherwise wouldn't be a well-defined function).
FUNC FindEntry (a, bucket) -> INT RAISES (NotFound) =
    VAR i: INT | 0 <= i /\ i < bucket.size /\ bucket[i].addr = a
            => RET i
    [*] RAISE NotFound

% Returns the Bucket obtained from the second argument by
% removing any pair containing a.  Assumes at most one pair
% contains a.
FUNC DeleteEntry (a, bucket) -> Bucket =
    i: INT := FindEntry(a,bucket)
          EXCEPT NotFound => RET bucket END;
    RET bucket.sub(0,i-1) + bucket.sub(i+1,bucket.size-1)

APROC Reset (d) =
   << m := Init(); init := d >>

APROC Read (a) -> D =
   << bucket := m[a.hf];
      i: INT := FindEntry(a,bucket)
            EXCEPT NotFound => RET init END;
      RET bucket[i].data >>

APROC Write (a, d) =
   << bucket := DeleteEntry(a,m[a.hf]);
      m[a.hf] := bucket ++ Pair{addr:=a,data:=d};
      RET >>

APROC Swap (a, d) -> D =
```

```
   << VAR d' | d' := Read(a); Write(a,d); RET d' >>

END HashMemory
```

The abstraction function maps a state of the HashMemory module to a state of the SimpleMemory module; it is given by the following Spec function:

```
FUNC AF (m: HashT, init: D, sz: INT) -> (A->D) =
  RET ( LAMBDA (a) -> D =
         VAR i: INT, j: INT |
                 0<=i /\ i<m.size /\ 0<=j /\ j<m[i].size
                      /\ m[i][j].addr = a => RET m[i][j].data
         [*] RET init )
```

Notice that the abstraction function makes sense only when there is at most one pair in the entire rep that contains a given address.

The representation invariant is true whenever no process is executing one of the procedures in the module; it is given by the following Spec predicate:

```
FUNC RepInv (m: HashT, init: D, sz: INT) -> BOOL =
  RET ( sz > 0 /\ m.size = sz
    /\ ( ALL a, i: INT, j: INT |
           0<=i /\ i<m.size /\ 0<=j /\ j<m[i].size
             /\ m[i][j].addr = a ==> i = a.hf )
    /\ ( ALL a, j1: INT, j2: INT | 0<=j1 /\ 0<=j2
             /\ j1<m[a.hf].size /\ j2<m[a.hf].size
             /\ m[i][j1].addr = a /\ m[i][j2].addr = a
                  ==> j1 = j2 )
    /\ ( ALL a | 0<=a.hf /\ a.hf<sz ) )
```

Informally, this says that the number of buckets is greater than 0, a pair containing address `a` appears only in the bucket at index `a.hf` in `m`, that at most one pair appears in that bucket for each address, and that the range of the hash function is appropriate for the number of buckets allocated.