Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.826 Principles of Computer Systems

SOLUTIONS TO PROBLEM SET 1

Problem 0

```
APROC Factor(n : Int) -> (Int, Int) RAISES {NoFactor} =
  << VAR p, q : Int | n > 1 /\ p > 1 /\ q > 1 /\ p * q = n => RET (p, q)
         [*] RAISE NoFactor >>

APROC Factor(n : Int) -> (Int, Int) RAISES {NoFactor} =
  << n <= 1 => RAISE NoFactor
         [*] {p := 2 BY p+1 WHILE p <= n/2 | n // p = 0 | (p, n/p) }.head
             [*] RAISE NoFactor >>
```

Problem 1

```
MODULE PriorityQueueImpl[T WITH {"<="}]
                   EXPORTS Insert, FindMin, DeleteMin

    % we artificially make our array start at index 1 by
    % inserting a null element at the beginning:
    VAR c : SEQ T := {Null}

    APROC Insert(n : T) = << c := c + n;
                             VAR newIndex, parentIndex : Int;
                                 newIndex := c.size;
                                 parentIndex := newIndex/2;
                                 DO (parentIndex > 0 /\ n <= c(parentIndex)) =>
                                     VAR t := c(newIndex);
                                     c(newIndex) := c(parentIndex);
                                     c(parentIndex) := t;
                                     newIndex := parentIndex;
                                     parentIndex := newIndex / 2
                                 OD
                          >>

    APROC FindMin() -> T RAISES {Empty} =
                           << IF c.size = 1 => RAISE Empty [*] RET c(0) >>

    APROC DeleteMin() -> T RAISES {Empty} =
                          << VAR min := c(1);
                               c(1) := c(c.size-1); c := c.sub(0,c.size-2);
                               Heapify(1);
                               RET min
                          >>
```

```
    APROC Heapify(i : Int) =
                        << % no children:
                           IF i*2 > c.size-1 => RET FI

                           % if only left child exists...
                           IF i*2 = c.size-1 =>
                               IF c(i*2) <= c(i) => VAR j : T |
                                 j := c(i*2); c(i*2) := c(i); c(i) := j;
                               FI
                           RET FI

                           % now, check both left and right children.
                           VAR smaller := 0 |
                             IF c(i*2) <= c(i*2+1) =>
                                smaller := i*2
                             [*]
                                smaller := i*2+1
                             FI
                             IF c(smaller) <= c(i) => VAR j : T |
                                j := c(smaller); c(smaller) := c(i); c(i) := j;
                                Heapify(smaller);
                             FI
                           RET
                        >>

    % The principal invariant says that you are always less than your parent.
    FUNC Inv () -> Bool = RET (ALL i | (1 < i /\ i < c.size) ==>
                               c(i) <= c(i/2))

    % The simplest abstraction function you can write is:
    FUNC AF () -> (T -> Int) = (\ t | RET c.Count(t))

    % If you don't want to use Count, just iterate through the array
    % looking for t, and sum up how many times it occurs.
END PriorityQueueImpl
```

Problem 2

```
MODULE SetAssociative[A WITH {hf : A->Int}, V, N, B]
             EXPORT Read, Write, Reset, Swap =

TYPE M      = A -> V
     Pair   = [a, v]
     Bucket = SET Pair
     HashT  = SEQ Bucket

VAR m         := InitM()
    v0        := default;
    c : HashT := InitC(v0)

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

```
% The complicated initialization is so that the rep. invariant
% will hold even at initialization
APROC InitC(v0) -> HashT = << VAR c' : HashT | c'.size = B /\
                                  (ALL i :IN c'.dom | c'(i).size = N /\
                                      (ALL p :IN c'(i) | p.a.hf = i) /\
                                      (ALL p :IN c'(i) | p.v = v0) /\
                                      (ALL a | {p' :IN c'(i) | p'.a = a}.size <= 1))
                              => RET c' >>

% It's OK to use :IN here, although it was fine to provide more detail.
APROC Read(a) -> V = << VAR bucket := c(a.hf);
                        VAR p :IN bucket | p.a = a => RET p.v [*]
                           VAR p' :IN bucket | Flush(p');
                                               p'.a := a; p'.v = m(a);
                                               RET m(a) >>

APROC Write(a, v) = << VAR bucket := c(a.hf);
                       VAR p :IN bucket | p.a = a => p.v := v [*]
                          VAR p' :IN bucket | Flush(p');
                                              p'.a := a; p'.v = v >>

APROC Reset(v) = << c := InitC(v); VAR m' | (ALL a | m'(a)=v) => m := m' >>

APROC Swap(a, v) -> V = << VAR t := Read(a); Write(a, v); RET t >>

% Internal function: flush
APROC Flush(Pair p) = << m(p.a) := p.v >>
```

The abstraction function is similar to the one for `HashMemory`.

```
FUNC AF() -> M = RET
     (LAMBDA(a) -> V = VAR b := c(a.hf);
                     IF VAR p :IN b | p.a = a => RET p.v [*] RET m(a)
                     FI)
```

The proof is fairly straightforward.

To show Condition 1, let t be an initial state of `SetAssociative`. Then `AF`(t) is some state of `Memory`. All memory states are allowable in initial states of `Memory`, so `AF`(t) is is an initial state of `Memory`, as needed. Condition 2 is a case analysis; let t and `AF`(t) be states of `SetAssociative` and `Memory`, and let (t, π, t') be a step of `SetAssociative`. We need to show that there is a step of `Memory` from `AF`(t) to `AF`(t') having the same trace.

- Read: Consider some state t of `SetAssociative` and the corresponding state `AF`(t) of `Memory`. Observe that `Read` does not change the abstract memory: if there is a cache hit, then it just returns a value from the cache, and otherwise, it just copies the value in main memory into the cache, possibly flushing an old value to memory; in both cases, the abstract memory is preserved. We next show that both `Read`s return the same value. The `Read` from `SetAssociative` returns `p.v` if `p.a = a`, otherwise it returns `m(a)`. The `Read` from `Memory` returns the value of `AF`(t)`.m(a)`, which is `p.v` if `p.a = a`, and `m(a)` otherwise. Since these are equal, we conclude that Condition 2 is satisfied in this case.

- Write: The abstract state is modified in one place: location `a` now contains `v`. Consider `SetAssociative`. If `a` was cached, then it is overwritten with `v`, which conforms to the change in the abstract state. If `a` was not cached, then some other location is flushed (no change in abstract state), and the cache is augmented with a mapping from `a` to `v`, also changing the abstract state as required.
- Reset: After executing `Reset`, all cache and memory locations of `SetAssociative` map to `v`. Therefore, the abstraction function will map all locations in the post-state to `v`. After a `Reset`, the post-state of `Memory` is the state where all locations contain `v`. Thus $\mathtt{AF}(t')$ is the same as the post-state of `Memory`.
- Swap: Since both `Read` and `Write` are correct, then `Swap` is clearly correct as well.

Problem 3

```
Module FileVersionImpl =
  TYPE FileRecord = [n:String, f:String, v:Int]

  % In my implementation, I maintain an in-memory map of strings to
  % version numbers.  This is OK because it is easy (but takes time
  % linear in the number of records written to disk) to reconstruct
  % this map completely from disk.  It was not OK if your implementation
  % needed some state that was in memory.  I preferred solutions where
  % you started looking for a version from the end of a disk.
  VAR diskEnd : Int = 0,
      disk : WriteOnceMemory[FileRecord],  % disk is dummy variable; need this line to
                                           % initialize the type T in WriteOnceMemory
      vs : String -> Int := {}

  APROC Write(n:String, f:String) = <<
      ~(n :IN vs.dom) => vs(n) := 0;
      diskEnd := WriteOnceMemory.append(FileRecord{n:=n, f:=f, v:=vs(n)});
      vs(n) := vs(n)+1 >>

  APROC Read(n:String, i:Int) -> String RAISES {NoFile} =
    << VAR q := diskEnd;
       DO q >= 0 =>
          (VAR d := WriteOnceMemory.fetch(q)
                       EXCEPT {NotThere} => RAISE NoFile |
                    IF d.n = n /\ d.v = i => RET d.f);
          q := q - 1
       OD
       RAISE NoFile >>

  APROC ReadLatest(n:String) -> String RAISES {NoFile} =
    << ~(n :IN vs.dom) => RAISE NoFile [*] RET Read(n, vs(n)-1) >>
END FileVersionImpl
```