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

SOLUTIONS TO PROBLEM SET 3

Problem 1: Disk Remapping (30)

A few people had a solution with the following property:

```
write(1000,"foo")     %remapped to 600
read(600)             %returns "foo"!
```

Such a solution was in error. The way to get around this is to allocate a fixed part of the disk for remapped sectors, and to reject reads of disk addresses in this allocated region. This would probably require modifications to `new`, `check`, and the abstraction function.

```
CLASS RemappingDisk EXPORT Byte, Data, DA, E, DBSize, read, write, size,
  check, Crash = % implements Disk

CONST
        N         := 5
        B         := 10
        remapSize := 100
VAR     udisk     :  Disk
        failures  :  DA -> Int
        remaps    :  DA -> DA := {}
        nextRemap :  DA

% size should return udisk.size() - R
APROC new(size: Int) -> Disk = <<
      self := StdNew(); udisk := udisk.new(size+remapSize);
      nextRemap := size + 1; RET self >>

APROC FreshRemap() -> DA RAISES {NoMoreSpace} = <<
      VAR q := nextRemap | nextRemap + := 1; RET q >>

% In these uncommon cases, it's acceptable to read a sector at a time.
PROC CorrectingRead(e) -> Data RAISES {notThere} =
      % I could have been much, much slicker using function composition.
      VAR data, i := e.da |
          DO i < e.da + e.size - 1 =>
          IF remaps!i  => data + := RemappingRead(remaps(i))
                     [*] data + := RemappingRead(i) FI;
          i := i + 1
          OD;
          RET data

PROC CorrectingWrite(da, data) RAISES {notThere} =
      VAR blockCount := DtoB(data).size, blocks := DtoB(data),
```

```
        j := 0 |
        DO j < blockCount =>
        IF remaps!(j+da)  => udisk.write(remaps(j+da), blocks(j))
                      [*] udisk.write(j+da, blocks(j)) FI;
        j := j + 1
        OD

% It turns out that it's pretty inconvenient to have an exception...
% Convert it to nil.
PROC readOrNil(e) -> Data + nil =
      RET udisk.read(E{da, 1}) EXCEPT error => nil

PROC RemappingRead(da) -> Data =
      VAR retryData : data := nil |
        DO retryData = nil /\ failures(da) < B => % free tries
            failures(da) + := 1;
            retryData := readOrNil(E{da, 1})
        OD
        IF retryData = nil /\ failures(da) > B => % must remap
            % try to read N times.
            VAR retryCount := 0 |
                DO retryData = nil /\ retryCount < N =>
                    retryCount + := 1;
                    retryData := readOrNil(E{da, 1})
                OD
                IF retryCount = N => RAISE error FI;
                VAR rda := FreshRemap() |
                    remaps(da) := rda;
                    write(rda, retryData);
        FI;
        RET retryData

PROC read(e) -> Data RAISES {notThere} =
      check(e);
      VAR data, i := e.da |  % see if already remapped
        DO i < e.da + e.size - 1 =>
        IF remaps!i => RET CorrectingRead(e) FI; i := i + 1
        OD;
      RET udisk.read(e) EXCEPT error =>
        VAR da := udisk.lastError(),
            retryData := RemappingRead(da),
            leftoverData := read(E{da+1, e.size - (da-e.da)}) |
            RET data.seq(0, da-e.da-1) + retryData + leftoverData

PROC write(da, data) RAISES {notThere} =
      VAR blockCount := DtoB(data).size, i := da |
        check(E{da, blockCount});
        DO i < da + blockCount - 1 =>
        IF remaps!i => CorrectingWrite(da, data); RET FI; i := i + 1
        OD;
        udisk.write(da, data)

APROC check(e) RAISES {notThere} =
    << e.das.rng <= udisk.disk.restrict(0..size) => RET [*] RAISE notThere >>
```

The abstraction function restricts the size of the disk and overlays the remapped sectors over the original disk.

```
AF disk.disk = udisk.disk.restrict(0..size) + (udisk.disk * remaps)
```

We use a standard forward simulation to prove that `RemappingDisk` implements the spec of `Disk`. Initial states correspond, since `remaps` starts out empty, and `udisk.disk` is clearly a valid state for `disk.disk`. We proceed by considering each of the (important) procs.

- `read`: Consider a concrete pre-state t and an extent e; the corresponding abstract pre-state is s. If no sectors of e are remapped in t and there are no errors, then `udisk.read(e)` is called and clearly the simulation holds. If some sectors of e are remapped, then the code for `CorrectingRead` returns the result of `udisk.read(`i`)` for unremapped sectors i and `udisk.read(remaps(`i'`)` for remapped sectors i' (which is identical to the result returned by `disk.read` on s). Finally, there's a lot of cruft around the remapping attempt, but what it boils down to is (assuming that `read` works correctly on sectors after the broken sector) either failing with `error`, as in the spec, or adding a mapping to `remaps` and copying the value, possibly recursively, but always preserving the abstract value of `disk.disk` at the broken sector:

  ```
  remaps(da) := rda;
  write(rda, retryData);
  ```

  Since `retryData` contains the value of `read(E{da, 1})`, there is no abstract state change caused by `read`. As for the return value, we can inductively assume that `leftoverData` is correct, as is the `data` prefix before `lastError`; `disk.disk(da)` is exactly the content of `retryData`, so we do return the same value.

- `write`: If there is no remapping in the sectors to be written, then `write` obviously implements its spec. In the event of remapping, then consider unremapped block j. Because it is unremapped, `remaps!j` is undefined, so that under the abstraction function, it is represented by `udisk.disk(j)`. That's where we write `blocks(j-da)`, so the abstraction of our concrete post-state is the same as the abstract execution on the abstract pre-state. For remapped block j, we concretely write to sector `remaps(j)`; our abstraction function says that the data for j is to be found there (because j was remapped), so the concrete and abstract transitions correspond.

Problem 2: Modelling a Hard Disk Head (20)

Here is a head implementation.

```
MODULE HeadImpl EXPORT Move, Position

  TYPE Place = [cyl: Int, sec: Int]

  VAR pos := Place(cyl := 0, sec := 0)

  APROC Move(dCyl: Int, dSec : Int) =
    << pos := Place(cyl := pos.cyl + dCyl, sec := pos.sec + dSec) >>

  FUNC sumCyl() = pos.cyl
```

```
  FUNC sumSec() = pos.sec
  FUNC Position() -> Place = Place(cyl:=sumCyl(), y:=sumSec())
END HeadImpl
```

Of course, we can't prove that this implementation satisfies the specification; we introduce the obvious history variable.

```
MODULE HeadImplH EXPORT Move, Position

  TYPE Place = [cyl: Int, sec: Int]
       Path = SEQ Place

  VAR pos := Place(cyl := 0, sec := 0)
  VAR pH := {}

  APROC Move(dCyl: Int, dSec : Int) =
    << pos := Place(cyl := pos.cyl + dCyl, sec := pos.sec + dSec);
       pH := pH + { Place(cyl := dCyl, sec := dSec)} >>

  FUNC sumCyl() = pos.cyl
  FUNC sumSec() = pos.sec
  FUNC Position() -> Place = Place(cyl:=sumCyl(), y:=sumSec())

  % AF Head.p = HeadImplH.pH
END HeadImplH
```

Note that `pH` is a history variable: the initial state of `HeadImplH` has `pH = {}`; `pH` disables no steps; and in fact, `pH` is a purely write-only variable, so that state never depends on history. We can therefore use `HeadImplH` in place of `HeadImpl`, because they have the same traces.

We use the following invariant of `HeadImplH` to describe how `HeadImplH.pH` is related to `HeadImplH.pos`.

```
INVARIANT
    HeadImplH.pos = Place(cyl := + : (pH * (\ place | place.cyl)),
                          sec := + : (pH * (\ place | place.sec)))
```

The invariant holds initially, because `pos` is $(0, 0)$ and `pH` is empty; furthermore, the implementation of `Move` maintains the invariant by changing `pH` and `pos` in consistent ways. Two obvious consequences of the invariant are:

```
    HeadImplH.pos.cyl = + : (HeadImplH.pH * (\ place | place.cyl))
    HeadImplH.pos.sec = + : (HeadImplH.pH * (\ place | place.sec))
```

We can now show that `AF` is an abstraction function from `HeadImplH` to `Head`. Clearly `AF` maps the initial state of `HeadImplH` to the initial state of `Head` (because both `Head.p` and `HeadImplH.pH` are `{}`). Consider `Move`. The concrete state of `HeadImplH` is transformed by

```
       pH := pH + { Place(cyl := dCyl, sec := dSec)}
```

(and stuff we ignore); under the abstraction function, this is equivalent to the statement

```
p := p + { Place(cyl := dCyl, sec := dSec};
```

which is actually executed by `Head`. Furthermore, `Position()` returns the same value in `Head` and `HeadImplH`: look at the consequences of the invariant and compare them to the definition of `Head.Position()`. Thus `HeadImplH` indeed implements `Head`.

Problem 3: Weighted Sums (30)

The implementation is fairly straightforward.

```
MODULE WeightedSumCalculatorImpl EXPORT sum, largestVal, add
  VAR isEmpty : bool := true,
      max : Int,
      total : Int := 0

  APROC add(val:Int, count:Int) = <<
      IF isEmpty => max := val FI
      isEmpty := false;
      total := total + val * count;
      IF val > max => max := val FI >>
  FUNC largestVal() -> Int RAISES {empty} =
                IF isEmpty => RAISE empty
                [*] max
  FUNC sum() -> Int RAISES {empty} =
                IF isEmpty => RAISE empty
                [*] total
End WeightedSumCalculatorImpl
```

This implementation adds back the abstract state in the form of history variables.

```
MODULE WeightedSumCalculatorImplH EXPORT sum, largestVal, add
  VAR pairsH : SEQ (Int, Int) := {},
      isEmpty : bool := true,
      max : Int,
      total : Int := 0

  APROC add(val:Int, count:Int) = << pairsH + := (val, count);
      isEmpty := false;
      total := val * count;
      IF val > max => max := val FI >>
  FUNC largestVal() -> Int RAISES {empty} =
                IF isEmpty => RAISE empty
                [*] max
  FUNC sum() -> Int RAISES {empty} =
                IF isEmpty => RAISE empty
                [*] total
End WeightedSumCalculatorH
```

The abstraction function simply maps `pairsH` to `pairs`.

```
AF WeightedSumCalculator.pairs = WeightedSumCalculatorImplH.pairsH
```

We also keep an invariant.

```
FUNC car (p : (Int, Int)) -> Int = p.seq.(0)
FUNC weightedVal (p : (Int, Int)) -> Int = p.seq(0) * p.seq(1)

INVARIANT
    pairs = {} => isEmpty
               [*] (max = (pairs * car).max
                    /\ total = + : (pairs * weightedVal))
```

Since we've stated the invariant, we'd better prove it. Initially, `pairs` is empty and `isEmpty` is true, so the invariant holds. After a call to `WeightedSumCalculatorH.add`, we see that `max` contains the largest number seen so far, and that `total` contains the weighted sum updated with the new entry: the difference between the old sum and the new sum is precisely `val * count`, the amount that we add to `total`. Nothing else changes the state, so we're done here.

The abstraction relation is as follows.

```
AR(s,t) -> Bool = RET s.pairs = {} => t.isEmpty
                                \/ (t.max = (s.pairs * car).max
                                   /\ t.total = + : (s.pairs * weightedVal))
```

Problem 4: Mind-Reading Vending Machine (20)[1]

We modify `DispenserImpl` so that the proofs go through:

```
MODULE DispenserImpl EXPORT insertCoin, dispense =
    VAR coinInserted : Bool := false, selection : String := "soda"

    APROC insertCoin() = << coinInserted := true; >>
    APROC dispense() -> String =
        << coinInserted => coinInserted := false;
            selection := "soda" [] selection := "pop";
            RET selection; >>
END DispenserImpl
```

a) Here's `DispenserImplP`:

```
MODULE DispenserImplP EXPORT insertCoin, dispense =
    VAR coinInserted : Bool := false, selection : String := "soda",
        pSelection : String := "soda"

    APROC insertCoin() = << coinInserted := true;
                            pSelection = "soda" [] "pop" >>
```

[1]Special thanks to Patrick Lam, last term's TA, for this detailed solution.

```
    APROC dispense() -> String =
        << coinInserted => coinInserted := false;
            pSelection = "soda" => selection := "soda" []
            pSelection = "pop" => selection := "pop";
            RET selection >>
END DispenserImplP
```

b) The abstraction relation from `DispenserImpl` to `DispenserImplP` is:

```
DispenserImplP.coinInserted = DispenserImpl.coinInserted
DispenserImplP.selection = DispenserImpl.selection
DispenserImplP.pSelection = *
```

The abstraction function from `DispenserImplP` to `MindReadingDispenser` is

```
MindReadingDispenser.coinInserted = DispenserImpl.coinInserted
MindReadingDispenser.selection = DispenserImplP.pSelection
```

c) To prove that `DispenserImpl` implements `DispenserImplP`, you could either prove that `pSelection` was indeed a prophecy variable, which guarantees that the prophecy-enabled implementation has the same traces as the prophecy-lacking implementation, or you can prove the backward simulation directly. I will do the latter, because it provides more insight into what's actually going on.

By the way, the conditions in the notes for proving that something is a prophecy variable aren't quite correct. In particular, condition 3 is wrong: it says "Same condition", but that clearly can't be the case: it cannot possibly be true that you don't read the value of the prophecy variable! What does need to hold is that prophecy variables may only be copied into other variables.

The invariant on `DispenserImplP` you need is as follows.

```
coinInserted => (pSelection = "pop" \/ pSelection "soda")
             [*] pSelection = selection
```

We show the invariant by arguing over forward executions: clearly it holds initially, because `coinInserted` is false and `pSelection` equals `selection`; after `insertCoin`, we explicitly establish `coinInserted` and `pSelection = "..."`; finally, after `dispense`, we set `coinInserted` to false and copy `selection` from `pSelection`, ensuring the invariant.

A related invariant clearly holds on `DispenserImpl`:

```
selection = "pop" \/ selection = "soda"
```

We proceed with the backward simulation proof. I found the explanation in the handout to be unsatisfactory; here's my attempt to explain the difference between forward and backward simulation, at least with respect to transitions.

Terminology: t is a concrete pre-state, t' is a concrete post-state (so that (t, π, t') is a transition of T). s is an abstract pre-state, s' is an abstract post-state, (s, π, s') is a transition of S. We'll always have $(s, t) \in R$ and $(s', t') \in R$. The difference is in the quantification.

- Forward simulation: $(\forall t)(\forall t')(\forall s)$ show that there exists at least one abstract post-state s' that is both a post-state of s and an abstraction of t'.
- Backward simulation: $(\forall t)(\forall t')(\forall s')$ show that there exists at least one abstract pre-state s that is both a pre-state of s and an abstraction of t[2].

With respect to reachability in condition 2: in forward simulation, recall that you only need to show for all t reachable. In backward simulation, conceptually you'd think that you need to show for all t' reachable. But if t' is reachable, so is t, by its very definition; we use $\forall$ (reachable) for both t and t'. So the only difference is (with respect to initial states and) whether you need to exhibit s or s'.

We will abbreviate $T =$ `DispenserImpl` by `DI` and $S =$ `DispenserImplP` by `DIP`.

1. State correspondence: Consider an initial state of `DI`, where `DI.coinInserted = false`, `DI.selection` arbitrary. This maps to a corresponding initial `DIP` state where `DIP.coinInserted = false` and `DIP.pSelection`, `DIP.selection` both arbitrary, which is an initial state of `DIP`. For arbitrary states of `DI`, our choice of abstraction function allows us to choose corresponding states of `DIP`, so that for any reachable state of `DI`, we'll be able to exhibit a state of `DIP`.
2. Transitions: We consider each proc of `DispenserImpl`.

   `insertCoin`: Let t be a concrete pre-state of `DI`; say $b =$ `DI.coinInserted` and $x =$ `DI.selection`, both arbitrary. Let t' be a concrete post-state of `DI`. In particular, t' has `DI.coinInserted = true`, and $x =$ `DI.selection` unchanged. Then t' abstracts to the state s' where `DIP.coinInserted = true`, and $x =$ `DIP.selection`. Also, $y =$ `DIP.pSelection` also seems arbitrary. However, we happen to know that our invariant about `DIP` holds, so that for reachable abstract states, `DIP.pSelection` is in fact either `soda` or `pop`[3]. Now, consider pre-state s where $b =$ `DIP.coinInserted`, $x =$ `DIP.selection`, $y =$ `DIP.pSelection`. It satisfies both the conditions that $(s, t) \in R$ and that $(s, \pi, s') \in S$, so this case is OK. (You should draw a picture.)

   `dispense`: Let t be a concrete pre-state of `DI`, and t' its post-state. Here, t has `DI.coinInserted = true` and $x =$ `DI.selection` $=$ `"soda"` or `"pop"`, while t' has `DI.coinInserted = false` and $x =$ `DI.selection = "soda"` or `DI.selection = "pop"`. Mapping to abstract states, the induced post-state s' has `DIP.coinInserted = false`, $x =$ `DIP.selection = "soda"` or `"pop"`, and (by the invariant) `DIP.pSelection = "soda"` or "pop" $= x$. Now consider state s with `DIP.coinInserted = true`, $x =$ `DIP.pSelection` $=$ `DIP.selection`; clearly it satisfies the invariant on `DIP`. Furthermore, it is a valid pre-state of s' and it abstracts t. Hence we have a backward simulation.

d) It is rather obvious that `DispenserImplP` and `MindReadingDispenser` have the same traces. Nevertheless, here's a short (forward simulation) proof that `DispenserImplP` implements `MindReadingDispenser`. Clearly the arbitrary initial state of `DispenserImplP` maps to an arbitrary initial state of `MindReadingDispenser`. Furthermore, `insertCoin` in `DispenserImplP` relates an arbitrary state to the (`DIP.coinInserted = true`, `DIP.pSelection` chosen) state; this matches the state of `MindReadingDispenser` after executing its abstract `MRD.insertCoin` proc and taking the same choice (namely, (`MRD.coinInserted = true`, `MRD.selection = DIP.pSelection` (by AF) with same choice). Finally, `dispense` in `DispenserImplP` may execute when `DIP.coinInserted = true`, makes it (and thus `MRD.coinInserted`) false, causes no other change in abstract state, and returns `DIP.pSelection` (`= MRD.selection` by AF). This thus corresponds to the precondition of, change caused by, and return value of `MRD.dispense`.

[2] You'd think that you'd have to show that for all reachable pre-states s such that $(s, \pi, s') \in S$, s is an abstraction of t. But you don't!

[3] If we didn't have the invariant, our proof would fail, because the post-state `DIP.pSelection = train` has no possible abstract pre-state.