Alexander J. Summers

Program Verification

Exercise Solutions 5: Encoding to SMT

Assignment 1 (Extensionality)

Extensionality can be expressed by the following axiom; the triggers chosen use the `isSequence` function discussed in the question:

```
axiom extensionality {
  forall s1:Sequence, s2:Sequence :: {isSequence(s1),isSequence(s2)}
    length(s1) == length(s2) && (forall i:Int :: {lookup(s1,i),lookup(s2,i)}
      lookup(s1,i) == lookup(s2,i))  ==>  s1==s2
}
```

This axiom will be instantiated for every pair of sequences in the problem (for which the `isSequence` assumption is added), i.e. there will be quadratically-many instantiations per ground sequence term. However, unlike in the previous exercise sheet, there isn't an obvious way to avoid this; there is no way to e.g. write an "inverse" function from the (unboundedly-many) sequence values to the sequence itself, which would be a way of characterising that a sequence is uniquely-determined by its values.

Assignment 2 (Axiomatising Maps)

The axiomatisation of maps was covered in the lectures, but is included here for the simple case described in the question. Adding the bulk-update operation requires a generalisation of the axioms for defining map-lookup over map-update (select-store axioms). The main technical difficulty is how to represent the condition for the bulk-update (defining which keys are to be updated). To represent a general condition seems to require passing a function as an argument to another function, which is not supported. Instead, we could represent these "filters" for the bulk-updates using maps from integers (keys) to booleans. We take the slightly simpler approach of *defunctionalisation*, here: we represent each desired filter function as an element of a new type `Filter`, which we equip with a `filter` function that models applying the filter to a particular key. We can then define, e.g. the `Filter` that is true exactly for even-number keys, by taking an unknown value of type `Filter` and defining its behaviour via a quantifier. We can then pass this `Filter` to our bulk-update operation:

```
domain Map {
  function select(m: Map, key: Int) : Int
  function store(m:Map, key: Int, value: Int) : Map
  function update_range(m: Map, from: Int, to: Int, value: Int) : Map
  function update_all(m:Map, f:Filter, v:Int) : Map

  axiom select_store_same {
    forall m: Map, k: Int, v: Int :: {select(store(m,k,v),k)}
      select(store(m,k,v),k) == v
  }

  axiom select_store_diff {
    forall m: Map, k1: Int, k2: Int, v : Int ::
    {select(store(m,k1,v),k2)} {select(m,k2),store(m,k1,v)}
      k1 != k2 ==> select(store(m,k1,v),k2) == select(m,k2)
  }

  axiom select_range_update {
      forall m: Map, from: Int, to: Int, v: Int, k: Int ::
      {select(update_range(m, from, to, v), k)}
      {select(m, k), update_range(m, from, to, v)}
      select(update_range(m, from, to, v), k) ==
        (from <= k && k < to ? v : select(m, k))
  }

  axiom select_bulk_update {
    forall m:Map, f:Filter, v:Int, k:Int ::
    {select(update_all(m,f,v),k)} {select(m,k), update_all(m,f,v)}
    select(update_all(m,f,v),k) ==
      (filter(f,k) ? v : select(m,k))
  }

}

domain Filter {
  function filter(f:Filter, i:Int) : Bool
}

method test(m : Map, f:Filter) {
  assume select(m,3) == 2;
  assume select(m,1) == 4;
  assume forall i:Int :: {filter(f,i)}
    filter(f,i) <==> i % 2 == 0
  assert select(update_range(m,2,4,1),3) == 1
  assert select(update_range(m,2,4,1),1) == 4
  assert select(update_all(m,f,5),3) == 2
```

```
  assert select(update_all(m,f,5),4) == 5
}
```

Assignment 3 (Sequence Take and Drop)

Here is a possible axiomatisation in Viper:

```
domain Sequence {
  function lookup(s:Sequence, i:Int) : Int
  function length(s:Sequence) : Int
  function take(s:Sequence, n: Int) : Sequence
  function drop(s:Sequence, n: Int) : Sequence

  axiom length_take {
    forall s:Sequence, n:Int ::
    {length(take(s,n))} {length(s),take(s,n)}
    length(take(s,n))==
      (n <= 0 ? 0 :
        (n >=length(s) ? length(s) : n))
  }
  axiom length_drop {
    forall s:Sequence, n:Int ::
    {length(drop(s,n))} {length(s),drop(s,n)}
    length(drop(s,n))==
      (n <= 0 ? length(s) :
        (n >=length(s) ? 0: length(s)-n))
  }
  axiom lookup_take {
    forall s:Sequence, n:Int, i:Int ::
      {lookup(take(s,n),i)} {lookup(s,i), take(s,n)}
    n > 0 && i < n && i < length(s) ==>
      lookup(take(s,n),i) == lookup(s,i)
  }
  axiom lookup_drop {
    forall s:Sequence, n:Int, i:Int :: {lookup(drop(s,n),i)}
    n < length(s) && i >= 0 && i < length(s)-n ==>
      lookup(drop(s,n),i) == lookup(s,i+n)
  }
  axiom lookup_drop_two { // as above for i == j-n
    forall s:Sequence, n:Int, j:Int :: {lookup(s,j), drop(s,n)}
    n < length(s) && j >= n && j < length(s) ==>
      lookup(drop(s,n),j-n) == lookup(s,j)
  }

  axiom length_pos {
```

```
    forall s: Sequence :: length(s) >= 0
  }
}

method test(s1: Sequence, s2:Sequence) {
  assume length(s1) >= 5
  assert lookup(take(s1,3),2) == lookup(drop(s1,2),0)

  assume take(s1,1) == take(s2,1)
  assert lookup(s1,0) == lookup(s2,0) // needs 2nd triggers on lookup_take

  assume drop(s1,1) == drop(s2,1)
  assert lookup(s1,1) == lookup(s2,1) // needs lookup_drop_two
}
```

With respect to potential incompletenesses, there is the usual extensionality issue; we might have two observationally-equivalent sequences that we cannot prove to be equal. Leaving aside sequence equality, the need for the second sets of triggers on the first three axioms `lookup_take` axiom, and for the `lookup_drop_two` axiom might not be immediately obvious. These allow the axiom to be instantiated in situations in which a lookup was performed on the original sequence, not the sequence after the `take` or `drop` operation; they are the "inverse" cases to those described by the first set of triggers. The `test` method illustrates an example in which the second triggers on `lookup_take` are necessary to prove the assertion. Similarly, the second axiom `lookup_drop_two` covers the analogous situation for `drop`. The reason this can't be directly achieved with an extra set of triggers on `lookup_drop` is that the analogous triggers to choose would be the terms {`lookup(s,i+n)`,`drop(s,n)`}, but the first term cannot be used in a trigger because of the integer + operator. Instead, the axiom `lookup_drop_two` expresses this "inverse" case of triggering by adjusting the range of the quantified variable to range over the index into `s` directly. This trick of rewriting axioms via arithmetic "shifts" to avoid problematic arithmetic operators in triggers is quite commonly-useful.