LAB 2: USING THE STAINLESS VERIFICATION SYSTEM

Rodrigo Raya, École Polytechnique Fédérale de Lausanne 5/3/2017

The following persons and materials have been useful for completing this lab.

As a theoretical foundation, we have used chapters 5 and 6 from the textbook of the course [1]. For practical issues, we have to thank users Nicolas Voirol and OlivierBlanvillain on stack-overflow and users chi and D.W. on cs.stackexchange. The relevant links are [2],[3],[4]. Special thanks to user OlivierBlanvillain who recommended us the article in [5] which guided us in the solution for the second exercise. Also the relevant section of the documentation of Stainless [6] although the part about termination can be expanded.

Exercise 1

The first task for this lab was to verify that mutually recursive implementation of the QuickSort algorithm actually sorts the list on which it operates. More precisely, we need Stainless to proof the postcondition of quickSort function:

Listing 1: Original implementation to verify

```
object QuickSort {

  def isSorted(list: List[BigInt]): Boolean = list match {
    case Cons(x, xs @ Cons(y, _)) => x <= y && isSorted(xs)
    case _ => true
  }

  def quickSort(list: List[BigInt]): List[BigInt] = (list match {
    case Nil() => Nil[BigInt]()
    case Cons(x, xs) => par(x, Nil(), Nil(), xs)
  }) ensuring { res => isSorted(res) }

  def par(x: BigInt, l: List[BigInt], r: List[BigInt], ls: List[BigInt]): ↩
     List[BigInt] = {
   ls match {
     case Nil() => quickSort(l) ++ Cons(x, quickSort(r))
     case Cons(x2, xs2) =>
       if (x2 <= x) par(x, Cons(x2, l), r, xs2)
       else par(x, l, Cons(x2, r), xs2)
   }
  }
}
```

After speaking with the assistants, reading the Stainless documentation and asking in [2], we understood that in order to verify a program in Stainless we have to guide the system and provide him with the relevant facts to do the proof.

According to the textbook [1], a useful property to proof that the resulting list is ordered, is to state that the elements on the resulting left list of par are less than the pivot and that the elements on the resulting right list of par are greater than the pivot. This statement is better understood in the equivalent iterative version of our recursive method.

However, it remains to proof that the concatenation of the ordered left list, the pivot element and the ordered right list is itself an ordered list and for that purpose we had to introduce three auxiliary lemmas:

Listing 2: Auxiliar lemmas in our proof

```
def lowerBoundLemma(l1:List[BigInt],l2: List[BigInt],x:BigInt):Boolean = {
 require(l1.content == l2.content
         && forall((y : BigInt) => l1.content.contains(y) ==> y <= x))
 forall((z : BigInt) => l2.content.contains(z) ==> z <= x)
}.holds

def upperBoundLemma(l1:List[BigInt],l2:List[BigInt],x:BigInt):Boolean = {
 require(l1.content == l2.content
         && forall((y : BigInt) => l1.content.contains(y) ==> y >= x))
 forall((z : BigInt) => l2.content.contains(z) ==> z >= x)
}.holds

def concatLemma(l1:List[BigInt],l2:List[BigInt],pivot:BigInt):Boolean = {
 require(isSorted(l1) && isSorted(l2) &&
         forall((z : BigInt) => l1.content.contains(z) ==> z <= pivot) &&
         forall((z : BigInt) => l2.content.contains(z) ==> z >= pivot)
        )
 isSorted(l1++Cons(pivot,l2)) because{
   l1 match{
     case Nil() => isSorted(Cons(pivot,l2))
     case Cons(h,Nil()) => h <= pivot && isSorted(Cons(pivot,l2))
     case Cons(h,t) => isSorted(l1) && concatLemma(t,l2,pivot)
   }
 } &&
 Set(pivot)++l1.content++l2.content == (l1++Cons(pivot,l2)).content &&
 1+l1.size+l2.size == (l1++Cons(pivot,l2)).size
}.holds
```

The first and second lemmas state a fairly simple property, if two lists have the same contents and all the elements of the first list are bounded by a constant, then the elements of the second list are also bounded by that constant. Our intend here is to proof that concatenating the sorted lists with the pivot does not alter the property that the left list is less than the pivot and the right list is greater than the pivot.

Once we have proved this facts we can proof the main lemma. Assuming we are given two sorted list l1 and l2 where the first list has elements less than the pivot x and the second list has elements greater than the pivot then, the concatenation of the lists is a sorted list.

Finally our annotated functions look like this:

Listing 3: Annotated quickSort and par functions

```
def quickSort(list: List[BigInt]): List[BigInt] = (list match {
   case Nil() => Nil[BigInt]()
   case Cons(x, xs) => par(x, Nil(), Nil(), xs)
}) ensuring {
   res => isSorted(res) &&
         res.content == list.content &&
         res.size == list.size
}

def par(x : BigInt,l : List[BigInt],r : List[BigInt],ls : List[BigInt]): ↩
    List[BigInt] = {
  require(forall((z : BigInt) => l.content.contains(z) ==> z <= x)  &&
          forall((z : BigInt) => r.content.contains(z) ==> z >= x) )
  ls match {
    case Nil() =>
      quickSort(l) ++ Cons(x,quickSort(r))
    case Cons(x2, xs2) =>
      if (x2 <= x) par(x, Cons(x2, l), r, xs2)
      else par(x, l, Cons(x2, r), xs2)
  }
}ensuring{res => isSorted(res)
              because{
                lowerBoundLemma(l,quickSort(l),x) &&
                upperBoundLemma(r,quickSort(r),x) &&
                concatIsSortedLemma(quickSort(l),quickSort(r),x)
              } &&
              Set(x)++l.content++r.content++ls.content == res.content &&
              1+l.size+r.size+ls.size == res.size
}
```

We have to note that in this verification we are not proving that the resulting list has the same elements which requires the notion of permutation. This notion must be weakened in order to proof our result. In some way the extra conditions on the contents of the list and size play this role.

We note also that the process we followed in order to proof our proposition is similar to the theoretical procedure given in [1] based on basic paths and the precondition method. All the time we where going from annotation to annotation (here preconditions and postconditions) trying to figure out what Stainless would need to proof the following annotation. These are the basic paths mentioned in the book.

Exercise 2

The second task for this lab was to verify that the implementation of the QuickSort algorithm terminates for all inputs it accepts. We take as a reference for the ranking functions the ones given in [5].

As the preconditions and postconditions are not part of the implementation we can remove them. However, it is curious that removing all preconditions and postconditions results in Stainless not being able of proving the termination. On the other hand, leaving all the preconditions and postconditions also results in Stainless not being able of proving the termination. We conclude that the system is using the hints we provide to proof termination. These hints have been verified separately so that we can be sure that they hold even in this context.

Listing 4: Using decreases construct to proof termination

```scala
def quickSort(list: List[BigInt]): List[BigInt] = {
  decreases((list.size,BigInt(0)))
  list match {
    case Nil() => Nil[BigInt]()
    case Cons(x, xs) => par(x, Nil(), Nil(), xs)
  }
} ensuring { res => res.content == list.content && res.size == list.size }

def par(x: BigInt, l: List[BigInt], r: List[BigInt], ls: List[BigInt]): ↩
   List[BigInt] = {
  decreases((l.size+r.size+ls.size,ls.size+1))
  ls match {
    case Nil() => quickSort(l) ++ Cons(x,quickSort(r))
    case Cons(x2, xs2) =>
      if (x2 <= x) par(x, Cons(x2, l), r, xs2)
      else par(x, l, Cons(x2, r), xs2)
  }
}ensuring{
  res => Set(x)++l.content++r.content++ls.content == res.content &&
          1+l.size+r.size+ls.size == res.size
}
```

References

[1] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer, 2007.

[2] OlivierBlanvillain. *Proof the concatenation of ordered list is an ordered list in Stainless*. 2017. URL: `http://stackoverflow.com/questions/42582153/proof-the-concatenation-of-ordered-list-is-an-ordered-list-in-stainless` (visited on 03/05/2017).

[3] Nicolas Voirol. *Use of the forall construct in Stainless*. 2017. URL: `http://stackoverflow.com/questions/42596092/use-of-the-forall-construct-in-stainless` (visited on 03/05/2017).

[4] chi and D.W. *Understanding type annotations for termination verification*. 2017. URL: `http://cs.stackexchange.com/questions/71138/understanding-type-annotations-for-termination-verification` (visited on 03/05/2017).

[5] Hongwei Xi. "Dependent Types for Program Termination Verification". In: *Higher-Order and Symbolic Computation* (2002), pp. 91–131.

[6] Several authors. *Leon documentation: Proving theorems*. 2017. URL: `https://leon.epfl.ch/doc/neon.html` (visited on 03/05/2017).