Purely Functional Algorithm Specification
Exercises Day 3 — With Answers

Jan van Eijck
CWI & ILLC, Amsterdam

ESSLLI, Opole, August 8, 2012

`homepages.cwi.nl/~jve/courses/12/esslli12/`

```
module Answers3

where
import List
import AssertiveCoding
```

Exercise

`merge` can be used as follows, to create a function for list sorting:

```
mergeSrt ::  Ord a => [a] -> [a]
mergeSrt [] = []
mergeSrt (x:xs) = merge [x] (mergeSrt xs)
```

Find a suitable assertion, and write an assertive version of this.

Answer: an obvious choice is the postcondition stating that the result of `mergeSrt` is sorted.

Wrapping this around the code gives:

```
mergeSrtA ::  Ord a => [a] -> [a]
mergeSrtA = post1 sorted mergeSrt
```

Follow-up Exercise

Another approach to merge sort is to start by splitting the list to be sorted in equal parts, recursively sort the parts, next merge.

Implement this, using the following split function.

```
split :: [a] -> ([a],[a])
split xs = let
   n = (length xs) `div` 2
  in
   (take n xs, drop n xs)
```

Answer:

```
mrgSrt ::  Ord a => [a] -> [a]
mrgSrt []  = []
mrgSrt [x] = [x]
mrgSrt xs  = let
    (ys,zs) = split xs
 in
   merge (mrgSrt ys) (mrgSrt zs)
```

Next, find a suitable assertion, and write an assertive version.

Answer: same postcondition as before:

```
mrgSrtA :: Ord a => [a] -> [a]
mrgSrtA = post1 sorted mrgSrt
```

Testing Euclid’s Algorithm

Suppose we want to write code for performing automated tests on Euclid’s algorithm for large ranges of values, as follows:

```
myEuclid :: Integer -> Bool
myEuclid k = let
    triv = (\_ -> True)
  in
    and [ triv $ (assert2 isGCD euclid) n m |
                           n <- [1..k], m <- [1..k] ]
```

Use your Haskell system to check if this is a reasonable test.

If there is something wrong, can you correct it?

Answer:

There is something wrong.

This does not work, for the Haskell compiler is too clever: it knows that it does not have to evaluate f on its arguments in order to find out that `triv` $\cdot$ f will yield True.

This can be remedied by replacing the trivial function by a function that depends on its input, e.g., the function $\lambda n.n > 0$. The result is the test that was given in the lecture slides:

```
testEuclid :: Integer -> Bool
testEuclid k =
  and [ (assert2 isGCD euclid) n m > 0 |
                        n <- [1..k], m <- [1..k] ]
```