Program Design and Data Structures, 20.0 c

Course code: 1DL201, Report code: DL201, 67%, DAG, NML,
week: 44 - 02 Semester: Autumn 2017
week: 03 - 11 Semester: Spring 2018

LAB ASSIGNMENT 14

This information is not available in English. Now showing the Swedish version.

After this lab you should be able to ...

- write test cases for your programs
- use tracing to debug programs
- use the debugger to step through a program
- (optionally) write QuickCheck specifications

BEFORE THE LAB:

- Read the parts of lecture slides 20 devoted to testing, tracing, debugging and (optionally) QuickCheck.
- Put on your detective hats, you're going bug hunting (hmmm, that's a mixed metaphor).
- Read the whole assignment first. There are links to external web pages. Read these pages when needed.

BACKGROUND

The goal of this lab session is to explore debugging finding and removing bugs in your program. Four techniques will be explored: testing (which you've seen before), using trace functions, using a debugger, and (optionally) property-based testing using QuickCheck.

Testing describes how a function should behave in terms of its input and expected output. Testing is useful for finding that there are bugs in a program, though it is not immediately helpful in locating the bugs. The input-output pair that reveals the bug can be used to help locate the bug.

Recall that a program can be tested by importing Test.HUnit and by writing test cases as follows

```
import Test.HUnit

foo x = x * x

test1 = TestCase (assertEqual "for (foo 3)," 9 (foo 3))
test2 = TestCase (assertBool "foo 10 > 0" (foo 10 > 0))
```

Name the test cases and group them together:

```
tests = TestList [TestLabel "test1" test1, TestLabel "test2" test2]
```

Run tests as follows.

```
runTestTT tests
```

See https://wiki.haskell.org/HUnit_1.0_User%27s_Guide for more details.

One way of locating a bug in a program is by **manually tracing** the program. This means applying the program to some input and unfolding the equations, evaluating the program, until it gives a result. In principle, this is the same idea as evaluating arithmetic expressions that you've seen in maths courses, though instead of arithmetic expressions programs are used.

For example, consider

```
length [] = 0
length (a : as) = 1 + length as
```

This function applied to [1,2,3] can be traced as follows:

```
length [1,2,3]
   {- second pattern matching branch matches. a -> 1, as -> [2,3] -}
= 1 + length [2,3]
  {- second pattern matching branch matches. a -> 2, as -> [3] -}
= 1 + 1 + length [3]
  {- second pattern matching branch matches. a -> 3, as -> [] -}
= 1 + 1 + 1 + length []
  {- first pattern matching branch matches. -}
= 1 + 1 + 1 + 0
  {- arithmetic -}
= 3
```

Manually tracing a program can be quite tedious and only really works for small programs.

Another way is to **instrument your code** to print out intermediate results so that you can see the steps the program is taking. The function

```
trace :: String -> a -> a
```

from module Debug.Trace can be added in code to print out a string. The string can be used to indicate where the trace function is being called. It can also contain the value being passed to the trace function, using show, as in the examples below. The second argument to the function is some value, which is the value returned by the trace function. Thus

```
trace "Foo" (1 + 2)
```

prints "Foo" to the screen and returns 3.

Now a more elaborate example:

```
module Main where
```

```
import Debug.Trace

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = trace ("n: " ++ show n) $ fib (n - 1) + fib (n - 2)


main = putStrLn $ "fib 4: " ++ show (fib 4)
```

Traces the fibonaci function, by evaluating main, produces output

```
n: 4
n: 3
n: 2
n: 2
fib 4: 3
```

For more details of how to use this function see http://en.wikibooks.org/wiki/Haskell/Debugging.

The final technique for debugging a program is using a **debugger**. A debugger allows one to stop a running program at certain breakpoints and from there inspect the state of the program, and step through the program -- kind of like tracing by hand, but with help of the machine. Using a debugger is a bit more involved, so its use is left as a challenge exercise for this lab. In any case, it is very useful to become aware of the debugger and learn how to use it.

Read here for more details https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci-debugger.html

Documentation on QuickCheck can be found here: https://wiki.haskell.org/Introduction_to_QuickCheck1

TASKS

Form groups of 2 and take at the look at the following programs which are buggy. They have two deliberately inserted bugs.

1. Pick one of the following programs.
2. Apply it to various inputs to see whether you can spot a bug -- this should be easy, the programs are very broken.
3. Write test cases for the programs using HUnit. Run your test cases. Write other test cases that you expect to succeed that test the boundary conditions.
4. Use the trace functionality to better understand what's going on inside the buggy function. Does this help you locate the bug?
5. Run the debugger on the code for some buggy input. Step through the program. Print out intermediate values to see whether they are what you expect (in general, try out all the commands available in the debugger). Do you understand the output the debugger gives?
6. (Optional) Try to use QuickCheck to automatically find bugs in your program. Read the documentation, formulate some properties, try to code them up, and run QuickCheck warning,

QuickCheck is not easy, but it is a very useful tool.

7. If you find the bugs, try to fix them (this is not the most important part of the lab) and run your test case on them.
8. When you've had enough of one program, try the other one.
9. Discuss the following points:

- Which of these tools could you use in which circumstances?
- What strategies for bug-finding did you use?
- How well do you imagine that these tools could be applied to larger programs?

Tips

- Passing Lab does not require finding the bugs. It requires trying out all techniques and discussing the process you went through first with your fellow lab mates, then with the TA. Try to discuss with TA early and often to avoid being stuck waiting at the end.
- Share strategies and tips with the team next you.
- Ask the TA questions.
- Read documentation online about HUnit, trace functions, Haskell's debugger and (optionally) QuickCheck.
- Please do **not** try to find the correct code on the internet to find the bug. The purpose of the lab is to exercise the tools, not finding the bug.

Program the First

The first program determines whether a string consists of balanced brackets. That is, that string consists entirely of pairs of opening ("[") and closing ("]") brackets, all of which are nested correctly.

Examples:

```
(empty)   OK
[]        OK
][        NOT OK
[][]      OK
][][      NOT OK
[[][]]    OK
[]][[]    NOT OK
```

When there is an error, the program indicates which position the mismatch was found.

```
import Text.Printf -- for printing stuff out

-- Return whether a string contains balanced brackets.  Nothing indicates a
-- balanced string, while (Just i) means an imbalance was found at, or just
```

```
-- after, the i'th bracket.  We assume the string contains only brackets.
isBalanced :: String -> Maybe Int
isBalanced = bal (-1) 0
   where bal :: Int -> Int -> String -> Maybe Int
         bal _   0      []  = Nothing
         bal i   _      []  = Just i
         bal i (-1)      _  = Nothing
         bal i   n ('[':bs) = bal (i+1) (n+1) bs
         bal i   n (']':bs) = bal (i+1) (n+1) bs


-- Print a string, indicating whether it contains balanced brackets.  If not,
-- indicate the bracket at which the imbalance was found.
check :: String -> IO ()
check s = maybe (good s) (bad s) (isBalanced s)
   where good s   = printf "Good \"%s\"\n" s
         bad  s n = printf "Bad  \"%s\"\n%*s^\n" s (n+6) " "
```

There are two bugs, both in the function isBalanced.

Program the Second

The second program is a well-known sorting algorithm.

```
-- sort the input list into increasing order
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ qsort [y | y <- xs, y > x]
```

WHEN YOU ARE DONE

When you are done with all problems (the explorations below are optional), raise your hand or approach an assistant to have your solution graded.

If you pass this lab at least 30 minutes early and other groups are still working on it, we ask you to **help one other group**. Do not simply share your solution with them, but try to understand the (partial) solutions that they have developed so far (which may be different from yours) and the difficulties that they have. Assist them in coming up with their own solutions. Once the group that you are helping completes the lab assignment, you may leave. (If it is still early, the group that received your help should then stay to help another group.)

EXPLORATIONS

If you have finished the lab exercises and are waiting to be graded, or if you wish to explore programming in Haskell further at any time, or want to get some extra practice, have a look at the Explorations page.

Exploration problems are optional and should only be attempted after other problems are completed.
You don't have to show these answers to the lab assistants for grading.