QuickEval: An Interactive Tool for Coverage Based Testing of Haskell Programs

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology (M.Tech)

by

Subhash P. Kale

Supervised By

Dr. Amey Karkare

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June, 2012

[illegible]

CERTIFICATE

It is certified that the work contained in this thesis entitled "*QuickEval: An Interactive Tool for Coverage Based Testing of Haskell Programs*", by *Subhash P. Kale(Roll No. 10111016)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 7/6/2012

(Dr. Amey Karkare)

Department of Computer Science and Engineering,

Indian Institute of Technology Kanpur

Kanpur-208016

ABSTRACT

We present QuickEval, a tool for Haskell programs that can be used to evaluate and test the programs based on the coverage. QuickEval combines random testing and program coverage. QuickEval can be used to execute a function with random or user given input and finds out the unevaluated or unexecuted expressions of function. This information can be used to generate new tests to cover unexecuted/uncovered expressions until all the subexpressions of the function are covered. Thus, QuickEval can be used to perform the unit testing and to achieve the desired coverage of program. QuickEval can also be used effectively to perform the assignment evaluation process as it can create the test suite automatically and also helps to evaluate the function with all possible test cases which guarantees the proper evaluation of assignments. Coverage based testing also helps the developers to debug the programs easily.

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Dr. Amey Karkare for his support and encouragement. I thank him for providing an opportunity to work under his guidance. Thanks for all technical discussions which added value to my technical knowledge.

I thank all professors of CSE department for their insightful courses and giving me wonderful learning environment. I thank all my friends for their encouragement and support.

I am grateful to my parents for their enormous support and love. I want to thank my brother Sachin, who encouraged me a lot over the years and always helped me in taking the right decision.

Finally, I would like to thank IIT Kanpur, which provides such a magical environment of friendship and great platform for research.

Subhash P. Kale

Dedicated to

my parents and my brother.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Why QuickEval?

1.1.1 Need of a testing tool for Haskell

In the process of learning and using programming languages, software tools like debuggers, testing tools and profilers play important role. Lack of such tools for Haskell language is one of the main reasons that very few people use Haskell [26]. Lack of debuggers forces the programmers to use alternative means to verify and debug programs. Two popular tools for Haskell are QuickCheck and Haskell Program Coverage (HPC). QuickCheck is used to generate random inputs for a function to verify user defined properties of the function. HPC, on the other hand, is used to find the program coverage for a particular input to the program. QuickCheck is useful because it allows user to test a function with large number of tests without much effort. HPC is useful as it allows programmer to find expressions in a program that are not evaluated for the inputs supplied. The programmer can use this information to devise new inputs that can force the evaluation of these unevaluated expressions, and analyze the output to catch any bugs in the program.

1.1.2 Need of a tool for assignment evaluation

The process of student assessment is one of the key issues in education and assignment evaluation is one of the Teachers' important tasks in this process. It is always desirable that evaluation process should (a) be consistent for all the solutions submitted (b) be less time consuming. Consistency here means that, for programming assignments, the same test-suite should be used for the evaluation of all assignments. For the programming assignments, it is also preferable that teacher should try to evaluate all expressions/paths of the program including the corner cases. This helps to find out all the bugs present in the program and also the full coverage ensures the proper evaluation of the assignments.

As the number of students is increasing in higher education, such type of assignment evaluation process has become very challenging and time consuming [8]. If done manually, the evaluation process may become inconsistent and it may also fail to cover all the expressions/paths of the program. It is laborious task for the teachers to generate the test cases for each programming assignment solution such that all the paths of the program will be covered and tested. To avoid such painful and time consuming process, teachers may try to set easier, shorter, or MCQ type assignments. Such type of assignments are not always good choice to assess the students' knowledge.

1.2 Use of QuickEval

In this report, we present a tool named QuickEval that combines the features of QuickCheck and HPC. QuickEval can be useful to teachers for performing consistent assignment evaluation within less period of time and also to the developers for performing coverage based unit testing. QuickEval enables the teachers to complete assignment evaluation in shorter time or to evaluate more assignments in the available period of time. With QuickEval user can

1. generate random input or use user supplied input for a function and create

the test suite for that particular function

2. get the executed and unexecuted portions of a function for a set of random/user-supplied inputs, and

3. based on the observed coverage, generate new inputs in order to cover unevaluated portions of the function.

QuickEval is useful for Haskell users in different ways:

1. developers can use the coverage information along with random test generation of QuickEval to generate large number of effective inputs for a function to discover bugs.

2. Novice users can use it to understand the behavior of the programs by looking at and reasoning about the evaluated and unevaluated portions of the program for a given input.

3. Teachers of Haskell programming language can use it for the evaluation of programming assignments as it requires much less effort to generate a test-suite that can achieve the full coverage of the programs submitted by the students. Tool also allows the teachers to dynamically update the test-suite so that all the assignments can be tested with full coverage and also within less period of time.

The QuickEval tool has a very simple user interface, and is easy to use. Unlike QuickCheck, user does not need to generate any properties of the function. However, this also means that user has to verify outputs manually (or through scripts) to find any bugs. The detailed comparison of QuickEval with other tools is discussed in Chapter 6. Our initial experiments of QuickEval on different assignments are very exciting. The detailed discussion of results is given in Chapter 5.

1.3 Outline of the this report

Chapter[2] discusses the Haskell language, Code coverage and different tools available for Haskell language. Chapter[3] explains the use of QuickEval to evaluate the programming assignments in Haskell. Chapter [4] gives the implementation details of QuickEval and Chapter[5] discusses the experimental results. QuickEval is compared with different Haskell tools in Chapter [6]. Conclusion and future work are explained in Chapter[7].

Chapter 2

Background and Related Work

2.1 Programming Language Paradigms

Programming languages are generally classified as imperative and declarative languages. The programs in imperative languages have an *implicit state.* These programs manipulate the stored values using different commands provided by source language. To allow this manipulation (*side effects*), programs use explicit notion of *sequencing.* For example, assignment command is used in most imperative languages to alter the values of *variables.* C, C++, Java, Fortran, Algol are some of the popular imperative languages.

Programs in declarative languages have *no implicit state.* These programs mainly constitute *expressions* and not *commands.* Declarative languages are divided into functional and relational languages.

In a functional language, the basic method of computation is based upon the application of *function* to arguments. Some examples of functional languages are as follows.
Purely Functional languages : Haskell, FP, Miranda
Hybrid languages : Lisp, Scheme, SML

In the relational languages, the basic method of computation is based upon the mathematical concept of *relation.* Prolog, Parlog, KL1 are some of the relational languages.

2.2 Haskell

Haskell is a pure functional programming language. This section describes different features of Haskell language.

- **Referential Transparency**: *Referential Transparency* means that an*expression* (or function) in a functional program always represents the same value. Since Haskell does not support the *side effects*, the value of *expression* always remains same when used at two different places in the program. This may not be possible with programs in imperative languages. The functions in imperative languages may alter the values of variables and return the different values, when called at two different places in same program. Referential transparency help us to construct and reason about the functional programs in the same way as mathematical expressions [3].

- **Higher Order Functions**: A *higher order function* is a function which can take function as an argument or return function as result value. In Haskell language, functions are treated as *first class objects.* Functions can be stored in data structures (like lists), passed as an argument to other functions and returned as result value. [13].

- **Lazy Evaluation**: Haskell supports *Lazy evaluation* strategy. In this strategy, expression is not evaluated as long as it's actual values is not needed in computation. This is in contrast with *eager evaluation.* Consider a function `f` as shown below.

  ```
  f (x) = 3
  ```

  In *eager evaluation*, if we call the function `f` with argument value 1/0 , it will generate an error. However, the value of an argument `x` is not needed in body of function `f`, so in *lazy evaluation*, no error will be generated and function will return value 3. Lazy evaluation allows to use the infinite data structures (like list of natural numbers) in the programs. Referential transparency helps to achieve the *lazy evaluation.* In order to postpone the evaluation of some expression, we need to get the same result later as we would get them before and that is where *referential transparency* is used.

- **List Comprehension**: In Haskell, *list* is a primary data structure used to store and manipulate the data in computation [17]. Haskell provides the *list comprehension* technique which allows the generation of new lists by manipulating and filtering elements of one or more existing lists.

- **Powerful type system**: Type system is useful to detect the *forbidden errors* such as attempting to add a number and a character [17]. Haskell is *strongly typed language* as it checks and prevents all forbidden errors. It also does error checking statically (at compile time). The powerful type system of Haskell language allows the functions to be overloaded and polymorphic [17].

2.3 Code Coverage

In the software testing, it is always desirable to test all the paths or statements of the program. Code coverage is the term used in software testing to give information about various executed parts of the program(such as statements, functions, blocks etc.)when executed with some particular input values. Using this information programmer can generate a test-suite which makes sure that all the parts of the program will be executed. Code coverage can be classified as white box testing technique. In the code coverage technique of testing, the internal structure of the program is tested rather than functionality of the program [18].

There are different ways to measure the code coverage for a particular program. Below is the summary of some fundamental types of code coverage [23].

- **Statement Coverage**: This is also known as **Line Coverage**. This type of coverage describes the executed statements of the program when tested with some input values. As there is direct association between statements and source code lines, this type of code coverage information is helpful to the developers [18]. In **Basic Block Coverage**, unit of code measured is not a single statement but a sequence of non-branching statements.

- **Decision Coverage**: This type of coverage describes whether boolean expressions tested in different structures (like if-else or while statements) evaluated to true and false values. In Decision coverage, the whole boolean expression may be evaluated as true or false without executing all the parts of the expression. If short-circuit technique is used while evaluating the expression then some part of the expression may be unexecuted.

- **Condition Coverage**: In this coverage type, each conditional subexpression is evaluated as true or false. In **Multiple Condition Coverage**, the entire boolean expression as well as subexpressions are considered for coverage information.

- **Path Coverage**: There can be number of paths in a function from the start of the function to it's exit(return statement). For different input values to the function, different paths can be executed. Path coverage describes whether each path was evaluated during the execution of a program.

- **Function Coverage**: This type of coverage gives the information about the functions which are evaluated during the execution of program.

2.4 Automated Tools for Imperative Languages

There are numerous automated testing tools available for different imperative languages. There are also some online judging softwares(codechef, sphere online judge etc) which take programs written in different languages and execute them with large number of predefined test cases. Such judging softwares only produce the results as `pass` or `fail` based on the output values and the execution time of the program. These tools do not provide any information about coverage of the function.

2.4.1 Unit Testing Tools

In unit testing, individual modules of the program are tested with different test cases. For procedural languages module can be procedure or function. In Objected oriented languages module can be class or individual method. Unit testing is considered one of the important phases of software testing as large number of defects are identified during this phase.

There are various unit testing frameworks which are collectively called as xUnit. For example FUnit for Fortran, NUnit for .NET languages, AceUnit, CUnit, CTest for C language, C++test for C++ language, COBOLUnit for Cobol, EUnit for Erlang, JUnit for Java, JSUnit for JavaScript. Using these frameworks developers can easily generate the test cases, execute the test cases as part of build process and capture the output [2]. Some of theses frameworks provide the rich set of assertions for testing common data types [9].

`Check` is one more unit testing tool for C language which provides the interface for defining unit tests [4]. `Jtest` is unit testing, code review and runtime error detection tool for Java language. It generates and executes the test cases for Java and also provide the static code analysis and data flow analysis [22].

2.4.2 Random Testing Tools

Random testing is generally easy to implement and it can also generate very large number of test cases. There are numerous random testing tools available for different languages. DART is random testing tool for C language [12]. It can randomly generate the test values for the function which is to be tested. It also directs the values of test cases such that uncovered expressions will be executed in the next iteration. `Jartege` is a random testing tool for Java language which can randomly generate the test cases for Java classes. These test cases can contain various sequences of constructor and method calls for the classes which are to be tested with Jartege [20]. `Eclat` is a one another random testing technique which randomly generates the large number of test cases and then selects a small subset which likely to be find the bugs in program under test [21].

2.5 Haskell Tools

2.5.1 QuickCheck

As mentioned earlier, QuickCheck [6] is used for random testing of programs written in Haskell programming language. For testing a function with QuickCheck, user needs to create properties for the function. Typically these properties are boolean predicates. QuickCheck then generates large number of random inputs for the function, and checks if the property is satisfied for each of the input.

For example, consider a standard function `reverse`. This function reverses a list. Function `reverse` satisfies the following properties.

```
          reverse[x] = [x]
    reverse (xs ++ ys) = reverse ys ++ reverse xs
  reverse (reverse xs) = xs
```

To check these properties with QuickCheck, we need to represent them using Haskell functions. Below code shows the Haskell functions corresponding to above properties.

```
prop1_rev x =
  reverse [x] == [x]


prop2_rev xs ys =
  reverse (xs ++ ys) == reverse ys ++ reverse xs
```

```
prop3_rev xs =
  reverse (reverse xs) == xs
```

The QuickCheck automatically generates the random values for these functions and if function returns `True` for each possible value then the properties are satisfied. The interaction of quickCheck with theses functions is shown below.

```
*Main> quickCheck prop1_rev
+++ OK, passed 100 tests.


*Main> quickCheck prop2_rev
+++ OK, passed 100 tests.


*Main> quickCheck prop3_rev
+++ OK, passed 100 tests.
```

If function returns `False` value for some test cases, then QuickCheck reports these input values to the user. For example, function `prop2_rev` mistakenly defined as

```
prop2_rev :: [Int] -> [Int] -> Bool
prop2_rev xs ys =
   reverse (xs ++ ys) == reverse xs ++ reverse ys
```

QuickCheck has produced following output

```
*Main> quickCheck prop2_rev
*** Failed! Falsifiable (after 3 tests and 1 shrink) :
[0]
[1]
```

where `[0]` is input value for `xs` while `[1]` is for `ys`. Function `verboseCheck` is similar to `quickCheck`. In addition to the boolean result of properties, `verboseCheck` also displays the randomly generated test cases.

2.5.2 HPC

HPC is a tool used to record and display Haskell Program Coverage. HPC works in three phases [11].

1. generate the instrumented program to gather the coverage information of the program
2. run the instrumented program
3. display the coverage information in different formats

Two types of coverage information is gathered by HPC: source coverage and boolean-control coverage. Source coverage describes the coverage information about every part of the program and it is measured at three different levels: declaration(both top-level and local), alternatives or patterns(among several equations or case branches) and expression. Boolean coverage describes the degree to which values `True` and `False` are evaluated in every boolean context (i.e. guard, condition, qualifier).

Coverage information obtained by HPC is displayed in two different ways: in the form of summary statistics using textual reports and source-code with color markup [11]

2.5.2.1 A Small example: Intersection of Lists

Consider the following piece of code to calculate and display the intersection of two integer lists.

```
--belongsTo x xs = x ‘elem‘ xs
belongsTo :: Integer -> [Integer] -> Bool
belongsTo x [] = False
belongsTo x (y:ys) = if x == y then True
                     else belongsTo x ys
```

```
--Sort the elements of list
sort [] = []
sort (x:xs) = sort (filter (<=x) xs ) ++ [x] ++ sort (filter (>x) xs )


--Remove consecutive duplicate elements from list
uniqify :: [Integer] -> [Integer]
uniqify [] = []
uniqify [x] = [x]
uniqify (x:y:rest)
        | x==y = uniqify (y:rest)
        | otherwise = x : uniqify (y : rest)


intersection :: [Integer] -> [Integer] -> [Integer]
intersection [] ys = []
intersection (x:xs) ys = if x `belongsTo` ys
                            then uniqify (sort (x : intersection xs ys ))
                            else [] ++ intersection xs ys
main = do
  list1 <- readLn
  list2 <- readLn
  putStrLn (show $ (intersection list1 list2) )
~
```

HPC first generates the instrumented version of the above program (Intersection.hs) using `hpc-build` script.

```
$ hpc-build Intersection
transforming Intersection.hs into ./.hpc/Intersection.hs
ghc   -w -package base -package hpc-0.4  -I./.hpc
-i./.hpc  -c  -o Intersection.o Intersection.hs
```

```
ghc  -w -package base -package hpc-0.4  -I./.hpc
-i./.hpc   -o Intersection Intersection.o
```

Program can be executed with some input values as follows.

```
$ ./Intersection
[4,6,1,8,9]
[1,2,3,4,5]
[1,4]
$:~/hprogs$
```

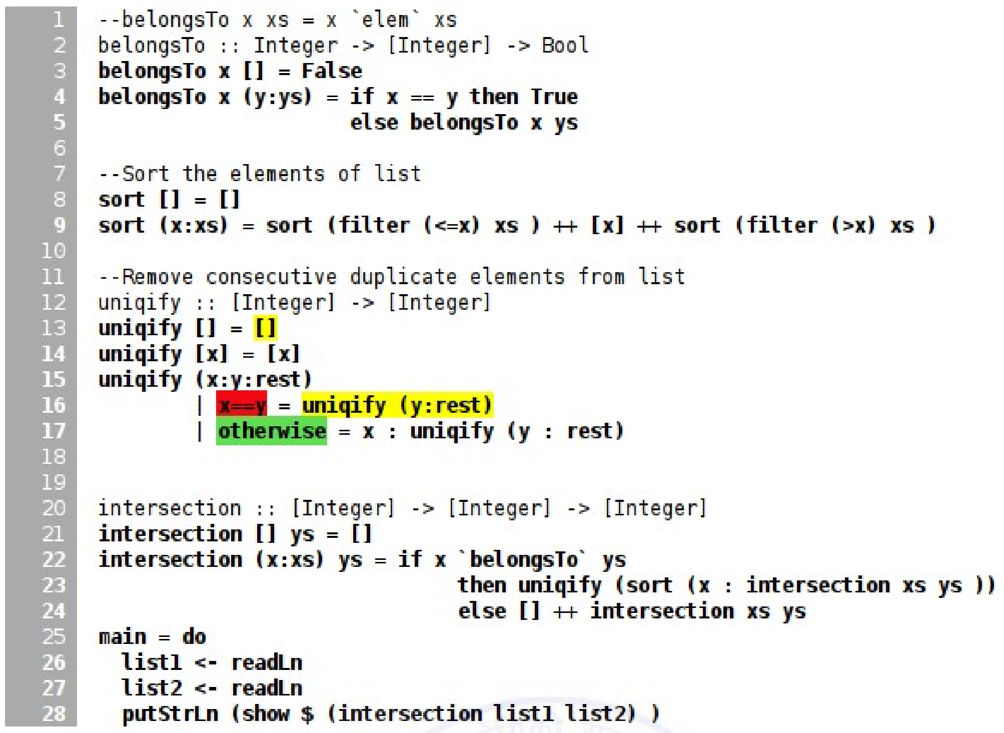
The textual summary of the coverage information can be obtained as follows.

```
$:~/hprogs$ hpc-report Intersection
 91% expressions used (73/80)
 50% boolean coverage (2/4)
       0% guards (0/2), 1 always True, 1 always False
     100% 'if' conditions (2/2)
     100% qualifiers (0/0)
 85% alternatives used (12/14)
100% local declarations used (0/0)
100% top-level declarations used (5/5)
```

Finally, the mark up version of coverage can be generated as follows.

```
$:~/hprogs$ hpc-markup Intersection
Writing: Intersection.hs.html
Writing: hpc_index.html
Writing: hpc_index_fun.html
Writing: hpc_index_alt.html
Writing: hpc_index_exp.html
```

Using the HTML browser we can see the mark-up version of the coverage and it looks like as shown in figure 2.1



```haskell
--belongsTo x xs = x `elem` xs
belongsTo :: Integer -> [Integer] -> Bool
belongsTo x [] = False
belongsTo x (y:ys) = if x == y then True
                     else belongsTo x ys

--Sort the elements of list
sort [] = []
sort (x:xs) = sort (filter (<=x) xs ) ++ [x] ++ sort (filter (>x) xs )

--Remove consecutive duplicate elements from list
uniqify :: [Integer] -> [Integer]
uniqify [] = []
uniqify [x] = [x]
uniqify (x:y:rest)
        | x==y = uniqify (y:rest)
        | otherwise = x : uniqify (y : rest)


intersection :: [Integer] -> [Integer] -> [Integer]
intersection [] ys = []
intersection (x:xs) ys = if x `belongsTo` ys
                              then uniqify (sort (x : intersection xs ys ))
                              else [] ++ intersection xs ys
main = do
  list1 <- readLn
  list2 <- readLn
  putStrLn (show $ (intersection list1 list2) )
```

Figure 2.1: `hpc-markup` output

2.5.3 HUnit

HUnit [16] is one another testing tool for Haskell. In HUnit, user needs to create the test cases, combine them into groups and then execute them. In the test cases, user needs to provide the input values and corresponding expected output values. The drawback of HUnit over QuickEval is that it can not generate the input values randomly, but relies on the user to provide them. HUnit also does not provide the coverage information of function. Generation of user test cases is an overhead for user that QuickEval tries to minimize.

2.5.4 Hood

Hood [10] is a tool used for debugging Haskell programs by observing intermediate data structures. Hood provides a library with various combinators that can be used by a user to mark different parts of a function. In the below example 'observe' is used to mark the expression.

```
value = use_result . observe "Result" . generate_result
```

Here `observe` is inserted between generation of result and use of result. `observe` stores the *result* and then returns it. After the program termination, user can use the browser to view the stored values. Any object stored by `observe` is viewed by browser. It takes the name of object and then displays it's value. User explicitly needs to mark the expressions and this tool does not provide information about coverage of program but can only provide the output of marked expressions.

2.5.5 Freja

Freja [19] is an interactive debugger for Haskell. To debug a program, Freja interactively asks the questions to user about the validity of the evaluation of various expressions. Each function concerns a function application and user has to answer the questions by a *yes* or a *no*. This is useful to find discrepancy in output to locate buggy expressions. The limitations of Freja are that it works only for a subset of Haskell and only on SPARC machines.

Chapter 3

A Tour of QuickEval

3.1 QuickEval for assignment evaluation

This section describes how to use the QuickEval to evaluate the assignments.

Consider that a teacher has given a programming assignment in Haskell to students, and she wants to evaluate the assignments. There are different Haskell functions defined in each assignment solution for different problem statements and these functions can be independent of each other. The problem statement for one of such functions in assignment is given below.

Problem Statement(1): Write a function `isSublist` which takes two (finite) lists, and returns true if the first list is a sublist of the second, false otherwise.
A list $l_{\texttt{sub}}$ is a sublist of a list $l_{\texttt{super}}$ if there exists $l_{\texttt{before}}$ and $l_{\texttt{after}}$, such that $l_{\texttt{super}} = l_{\texttt{before}} ++ l_{\texttt{sub}} ++ l_{\texttt{after}}$.

3.1.1 Correcting the first solution

Teacher can randomly pick up any assignment solution and start to evaluate it using QuickEval. Further, if a model solution is available(created by TA or teacher herself), it can be taken as first solution as it will probably be the most complete solution coming from the experienced programmer. Below code shows the Haskell function `isSublist` for problem statement (1) from the first assignment solution that teacher has picked up randomly to evaluate.

```
isSublist :: [Int] -> [Int] -> Bool
isSublist subList superList
          | length superList < length subList = False
          | subList == [x | (x,y)<- (zip superList subList)] = True
          | otherwise = isSublist subList (tail superList)
```

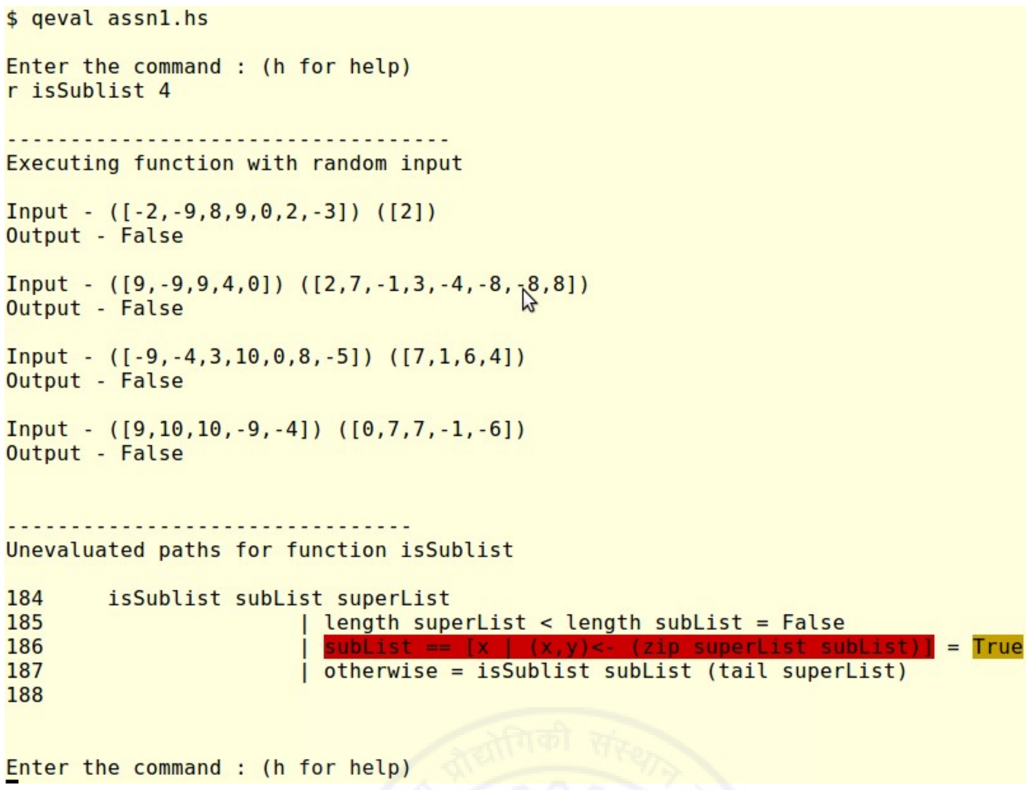
Using QuickEval, teacher can generate the test cases for each function in assignment solution. QuickEval allows the user to either supply an input (command 'i') or generate a random input (command 'r'). If test-suite is available for some function, then instead of generating input values randomly or manually, teacher can directly use the input values from test-suite (command 'f').

Since this is the first assignment solution teacher is evaluating, the test-suite for function `isSublist` would be empty. In such a situation, teacher would prefer to randomly generate input values for function `isSublist` using command 'r'. Figure 3.1 shows the interaction of teacher with QuickEval when she chooses a command 'r' with count 4, so that 4 random input values are generated[1]. The corresponding output values for the random inputs and the coverage information of function `isSublist` is shown in the same Figure.

These randomly generated input values are saved in a test-suite for function `isSublist` for later use. The saved test cases of function `isSublist` can be displayed using command 'd' as shown below.

```
Enter the command : (h for help)
```

[1]Note that, we are not generating large number of random test cases, though it is not impossible. Our main emphasis is not only on the coverage of function but also on the output of the function. Teacher should be able to check the output of function with generated input values. If we generate large number of test cases, then also QuickEval will show the accumulated coverage, but it will be laborious task to check the output values manually.



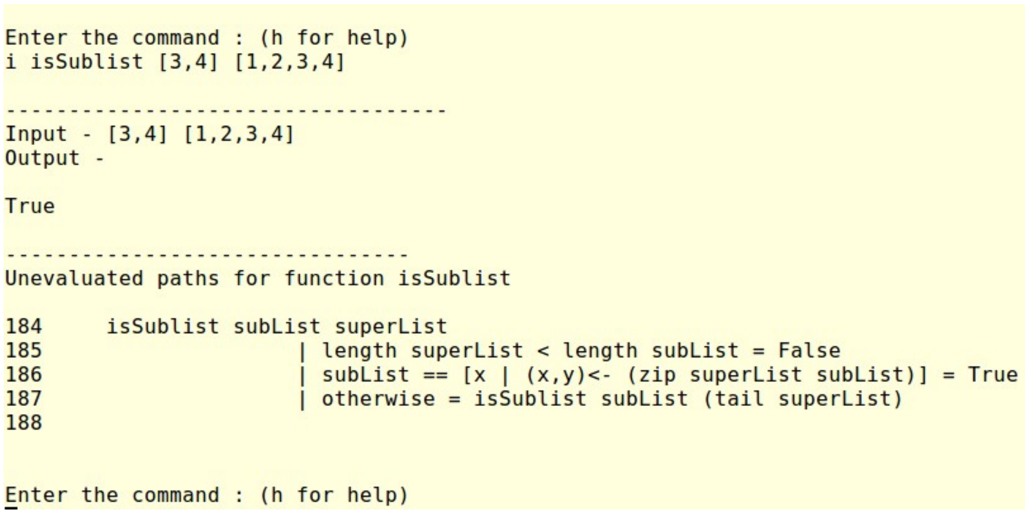
```
$ qeval assn1.hs

Enter the command : (h for help)
r isSublist 4

-----------------------------------
Executing function with random input

Input - ([-2,-9,8,9,0,2,-3]) ([2])
Output - False

Input - ([9,-9,9,4,0]) ([2,7,-1,3,-4,-8,-8,8])
Output - False

Input - ([-9,-4,3,10,0,8,-5]) ([7,1,6,4])
Output - False

Input - ([9,10,10,-9,-4]) ([0,7,7,-1,-6])
Output - False


--------------------------------
Unevaluated paths for function isSublist

        isSublist subList superList
                       | length superList < length subList = False
                       | subList == [x | (x,y)<- (zip superList subList)] = True
                       | otherwise = isSublist subList (tail superList)


Enter the command : (h for help)
```

Figure 3.1: `isSublist function with random input from QuickEval`

```
d isSublist


isSublist ([-2,-9,8,9,0,2,-3]) ([2])

isSublist ([9,-9,9,4,0]) ([2,7,-1,3,-4,-8,-8,8])

isSublist ([-9,-4,3,10,0,8,-5]) ([7,1,6,4])

isSublist ([9,10,10,-9,-4]) ([0,7,7,-1,-6])
```

The coverage information displayed by QuickEval is annoted source code of all the functions which are invoked in the particular iteration. The annotation is in forms of colors over different parts of codes. The color scheme is inspired by HPC [11]:green color indicates the part of code that was always evaluated as `True`, red color means that part of code was always evaluated as `False` and yellow color indicates that the part of code was not evaluated for the provided input values.

From Figure 3.1, we can see that all the parts of the function `isSublist` are not covered. In such a case, teacher can repeat the process and QuickEval will



```
Enter the command : (h for help)
i isSublist [3,4] [1,2,3,4]

-----------------------------------
Input - [3,4] [1,2,3,4]
Output -

True

--------------------------------
Unevaluated paths for function isSublist

184     isSublist subList superList
185                     | length superList < length subList = False
186                     | subList == [x | (x,y)<- (zip superList subList)] = True
187                     | otherwise = isSublist subList (tail superList)
188


Enter the command : (h for help)
```

Figure 3.2: `isSublist function with input supplied by user`

execute the function `isSublist` again, with different set of random values. In case teacher decides that it is unlikely that unevaluated expressions will be evaluated using random tests, she can provide the input manually. Figure 3.2 shows the case where user has specified the input values.

After second interaction of teacher with QuickEval, test-suite for function `isSublist` is modified as below.

```
Enter the command : (h for help)
d isSublist


isSublist ([-2,-9,8,9,0,2,-3]) ([2])
isSublist ([9,-9,9,4,0]) ([2,7,-1,3,-4,-8,-8,8])
isSublist ([-9,-4,3,10,0,8,-5]) ([7,1,6,4])
isSublist ([9,10,10,-9,-4]) ([0,7,7,-1,-6])
isSublist [3,4] [1,2,3,4]
```

The inputs (random or user-supplied) can be saved automatically to create a test-suite for later testing. This process makes sure that the tests added to the test-suite cover new expressions in the program, and do not only cover parts that are already covered by other existing tests.

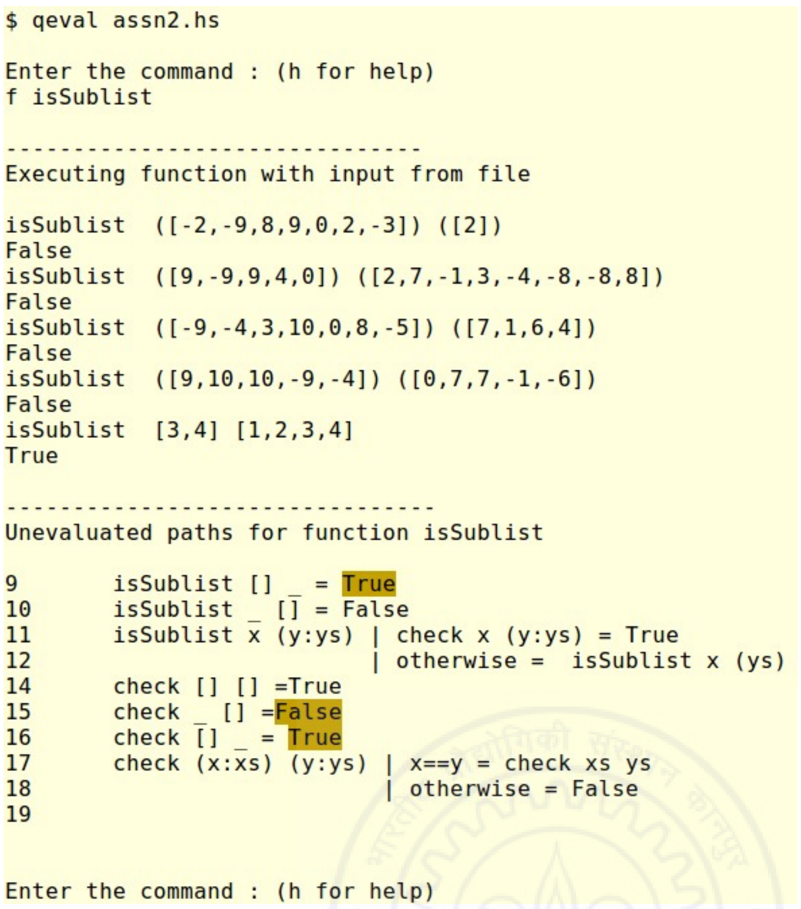
Thus, QuickEval enables teacher to perform random testing as long as she wishes and then if there are any unexecuted expressions, user can provide the inputs to cover such expressions. After testing one function, user can focus on another function from the same assignment solution and test it in the same way. With QuickEval, testing can be performed in a systematic way. Now since, all the expressions of the function `isSublist` are covered and output values of function for various input values are checked, teacher can take another assignment solution for evaluation.

3.1.2 Correcting the subsequent solutions

Below code shows the solution submitted by another student for the problem statement(1).

```
-- check whether a list is a sublist of second list
isSublist:: [Int]->[Int]-> Bool
isSublist [] _ = True
isSublist _ [] = False
isSublist x (y:ys) | check x (y:ys) = True
                   | otherwise =  isSublist x (ys)

-- comparisons are done by check function
check [] [] =True
check _ [] =False
check [] _ = True
check (x:xs) (y:ys) | x==y = check xs ys
                    | otherwise = False
```

Since the test-suite for function `isSublist` is available , teacher does not need to generate the values randomly or manually. Here teacher can use command 'f'



```
$ qeval assn2.hs

Enter the command : (h for help)
f isSublist

-------------------------------
Executing function with input from file

isSublist  ([-2,-9,8,9,0,2,-3]) ([2])
False
isSublist  ([9,-9,9,4,0]) ([2,7,-1,3,-4,-8,-8,8])
False
isSublist  ([-9,-4,3,10,0,8,-5]) ([7,1,6,4])
False
isSublist  ([9,10,10,-9,-4]) ([0,7,7,-1,-6])
False
isSublist  [3,4] [1,2,3,4]
True

--------------------------------
Unevaluated paths for function isSublist

9       isSublist [] _ = True
10      isSublist _ [] = False
11      isSublist x (y:ys) | check x (y:ys) = True
12                         | otherwise =  isSublist x (ys)
14      check [] [] =True
15      check _ [] =False
16      check [] _ = True
17      check (x:xs) (y:ys) | x==y = check xs ys
18                          | otherwise = False
19


Enter the command : (h for help)
```

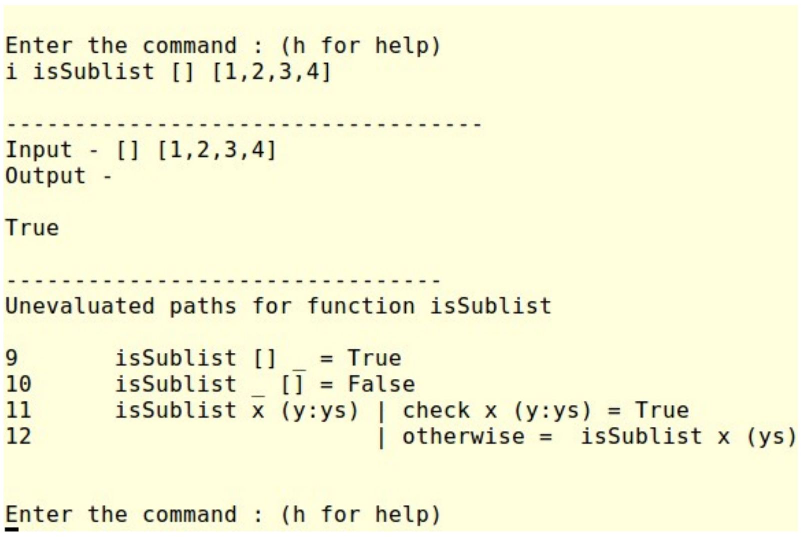
Figure 3.3: `isSublist function with input supplied from the test suit`

to use the test cases available in test-suite. Figure 3.3 shows interaction of teacher with QuickEval using command 'f'.

From Figure 3.3, we can see that most of the part of the function `isSublist` is covered by the already generated test cases in test-suite. Such type of automatically generated test-suites considerably reduces assignment evaluation time. To cover the remaining part of function, teacher can generate the input value manually as shown in Figure 3.4. Our experimental results have shown that only for first few assignments, teacher needs to add input values to test suite. Experimental results are discussed in section 5.

After manually generating input value for function `isSublist` test-suite is modified as shown below.

```
Enter the command : (h for help)

d isSublist
```



```
Enter the command : (h for help)
i isSublist [] [1,2,3,4]

-----------------------------------
Input - [] [1,2,3,4]
Output -

True

--------------------------------
Unevaluated paths for function isSublist

9       isSublist [] _ = True
10      isSublist _ [] = False
11      isSublist x (y:ys) | check x (y:ys) = True
12                         | otherwise =  isSublist x (ys)


Enter the command : (h for help)
```

Figure 3.4: `isSublist function with input supplied by the user`

```
isSublist ([-2,-9,8,9,0,2,-3]) ([2])

isSublist ([9,-9,9,4,0]) ([2,7,-1,3,-4,-8,-8,8])

isSublist ([-9,-4,3,10,0,8,-5]) ([7,1,6,4])

isSublist ([9,10,10,-9,-4]) ([0,7,7,-1,-6])

isSublist [3,4] [1,2,3,4]

isSublist [] [1,2,3,4]
```

3.2 QuickEval as a testing tool for Developers

Developers of Haskell programs can use the QuickEval as coverage based testing tool. Starting with random inputs, user can locate the unexecuted expressions. User can then manually generate the input values such that unexecuted expressions will be evaluated. Beside the coverage information of function, user can also check the output of function for that particular input and find the bugs in output value if there are any. With the help of QuickEval, user can generate the input values randomly or manually, such that full coverage can be achieved. Since, QuickEval is interactive tool, user can perform unit testing of different functions present in the same program with less efforts and in less time.

Chapter 4

Internals of QuickEval

We now describe various parts of QuickEval implementation.

4.1 Random test generator

The front-end of our tool is based on the QuickCheck [6]. The main task of the front-end is to generate random input values for the function to be tested and find the corresponding output. We can not use QuickCheck directly because it does not work directly on the function which is to be tested, but on the user-defined properties of the function.

We have made some changes in the source code of QuickCheck so that actual function is executed with randomly generated input values. Further, the output of the function is displayed to the user. To do so, we have changed the type `Result` as follows:

```
data Result = Result { arguments::[String], ans :: String }
               deriving Show
```

4.2 Coverage generator

The back-end of our tool is based on HPC [11]. The main task of the back-end is to find out the unevaluated or unexecuted subexpressions of a function based on

```
data Mix
  = Mix
    FilePath   -- location of
               -- original file
    Integer    -- time (in seconds)
               -- of original file
    Hash
    Int
    [MixEntry] -- entries
```

```
data Tix = Tix [TixModule]
   deriving (Read, Show)

data TixModule
   = TixModule
     String    --module name
     Hash
     Int -- length of tix list
     [Integer] -- actual tics
 deriving (Read,Show, Eq)
```

```
type MixEntry = (SourcePosition, BoxLabel)

data BoxLabel = ExpBox Bool
              | TopLevelBox [String]
              | LocalBox [String]
              | BinBox BoolCxt Bool
```

```
data BoolCxt = GuardBinBox
             | CondBinBox
             | QualBinBox
```

Figure 4.1: `Data types used by HPC and QuickEval (definitions borrowed from HPC [11])`

random or user-supplied input values. To understand the working of QuickEval, we need to understand the working of HPC. We present a simplified view of working of HPC next. The exact details differ in minor ways and are described elsewhere [11].

HPC translates the original program into an instrumented program. The instrumented program associates a unique number called *tickBox* with each expression of original program. Initially the tickBox value is zero for each expression. At the runtime, if an expression is evaluated then the corresponding tickBox value is increased. Thus, tickBox value can be used to decide whether an expression is evaluated by an execution of the program.

Along with the instrumented program, HPC also generates Mix (Module Index) file. The Mix file is used to record the location of source code associated with each tickBox introduced in the instrumented program. Figure 4.1 shows the important data types used by HPC, such as Tix and Mix.

The `TopLevelBox` values can be used to find out the start and end locations of the functions in a program. These locations are required to display the exact source code of functions as output of QuickEval.

QuickEval differs from HPC in that we support user-supplied inputs as well as random generation of inputs. To do so, QuickEval creates two versions of instrumented program and mix file for a given program. Each of the generated instrumented program contains an automatically generated `main` function, which in turn calls a *pilot* function. Pilot function is a dummy function that is replaced by the function to be tested during the testing phase.

The two versions of instrumented program differ in the way `main` function calls the pilot function. For random testing, `main` calls the pilot function under the control of random input generator (described in Section 4.1). For testing with user-supplied input, `main` function calls the pilot function directly, with user-supplied input. In each case the `main` is appended at the end of the instrumented file. Therefore, in the generated Mix files, the locations corresponding to the start and end of all the functions in the instrumented program remain same as those in original program. Since QuickEval does not apply tickBoxes to the `main` function, its location is not of interest.

The combination of Tix and Mix files gives us the information about locations of expressions and their evaluation status. Since the source locations of all functions except `main` function are same in both versions of Mix file, we get the correct location and evaluation status of the expressions.

Chapter 5

Our Experiments with QuickEval

In this section we present the results of several experiments performed with QuickEval on different assignments. We used QuickEval to evaluate the few assignments written by students. While evaluating the assignments, we randomly picked up any assignment to evaluate using QuickEval. Our experimental results show that, complete test-suite can be generated by evaluating first few assignments only and this test-suite can be used to evaluate later assignments without user generating input values randomly or manually.

5.1 Function `BelongsTo`

Problem Statement

```
Write a function belongsTo which takes an element and
a list of elements and returns True if element
is member of a list and False otherwise.
```

Below table shows the different input values generated to test the `belongsTo` function for different assignment solutions and comments about the output of function.

Assignment	Function (belongsTo)	Comments

Assn1	`Newly generated input :`
	`10 [],`
	`2 [-7,-8],`
	`-5 [-2,-5,-8,-9,-10,-1,-2],`
	`-8 [10,-10,-7,0,8,8],`
	`10 [-8,8,9,-5,7,-9,-3,9,5],`	

Assn2	No new input generated	

Assn3	No new input generate	`Error with input`
		`belongsTo 10 []`

Assn4	No new input generated	
Assn5	No new input generated	
Assn6	No new input generated	
Assn7	No new input generated	
Assn8	No new input generated	
Assn9	No new input generated	
Assn10	No new input generated	
Assn11	No new input generated	
Assn12	No new input generated	

Table 5.1: `Evaluation of belongsTo with QuickEval`

The above results show that, after evaluating the first assignment, there were five input values in test-suite for `belongsTo` function. This test-suite was sufficient to evaluate same function from other assignment solutions without generating new input values. While evaluating one assignment user could also detect the wrong output value for some particular input values.

5.2 Function `setDifference`

Problem statement:

```
Write a function setDifference which takes two lists and return

a list with elements from first list which are not

in the second list.
```

Assignment	Function (SetDifference)	Comments
Assn1	Newly generated input : [-4,0,3,0,-1,-4,5] [-9,1], [10,-2] [], [-9,-1,2,6] [-6,-5,5,9,4], [1,2,3] [1,3],	
Assn2	No new input generated	
Assn3	Newly generated input [] []	
Assn4	No new input generated	
Assn5	No new input generated	
Assn6	No new input generated	
Assn7	No new input generated	
Assn8	No new input generated	
Assn9	No new input generated	
Assn10	No new input generated	
Assn11	No new input generated	
Assn12	No new input generated	

Table 5.2: `Evaluation of setDifference with QuickEval`

For this function user generated input values for two assignment solutions and other assignment solutions were evaluated using test-suite only.

5.3 Function `isSublist`

Problem statement:

```
Write a function isSublist which takes two (finite) lists,

and returns true if the first list is a sublist of the second,

false otherwise.
```

`A list` l_{sub} `is a sublist of a list` l_{super} `if there exists` l_{before} `and` l_{after},

`such that` $l_{super} = l_{before}$ `++` l_{sub} `++` l_{after}.

Assignment	Function (isSublist)	Comments

Assn1	`Newly generated input :`
	`[-4,-10,-5,10,8,-3,5,-3]`
	`[2,-8,6,-9,-5],`
	`[-10,9,6] [-1,4,7,4,-5,-7],`
	`[3,4,5] [1,2,3,4,5,6]`	

Assn2	`Newly generated input`
	`[] [1]`	

Assn3	`Newly generated input`
	`[] []`
	`[1] []`
	`[1,2] [1,2]`	

Assn4	No new input generated	

Assn5	No new input generated	`Error on inputs`
		`isSublist [1] []`

Assn6	No new input generated	
Assn7	No new input generated	
Assn8	No new input generated	
Assn9	No new input generated	
Assn10	No new input generated	
Assn11	No new input generated	

Table 5.3: `Evaluation of isSublist with QuickEval`

5.4 Function `errno`

Problem Statement:

`A well balanced parenthesis string (WBPS)`

`is defined by the grammar` $S \rightarrow (S)\ S\ |\epsilon$

`As examples, the string (), (()), and (())() are WBPS and`

`the strings )( and (() are not. A string which is not WBPS`

```
can be converted to a WBPS by removing some characters from it.
e.g. ((() gets converted to (). Of course, by removing all
characters from ((() we can also get a WBPS, but we want
to remove a minimum number of characters.

Define a function errno, which when given a parenthesized string,
would return minimum number of characters, removal of which would
give a WBPS. As example, errno ")(" returns 2, errno "())()))"
returns 3, and errno "(())()" is, of course 0.
```

Assignment	Function (errno)	Comments

Assn1	`Newly generated input :`
	`"((()()"`
	`"))()()"`
	`"()()"`	

Assn2	No new input generated	
Assn3	No new input generated	
Assn4	No new input generated	
Assn5	No new input generated	
Assn6	No new input generated	
Assn7	No new input generated	
Assn8	No new input generated	
Assn9	No new input generated	
Assn10	No new input generated	
Assn11	No new input generated	

Table 5.4: `Evaluation of errno with QuickEval`

Chapter 6

Comparing QuickEval with Other Haskell Tools

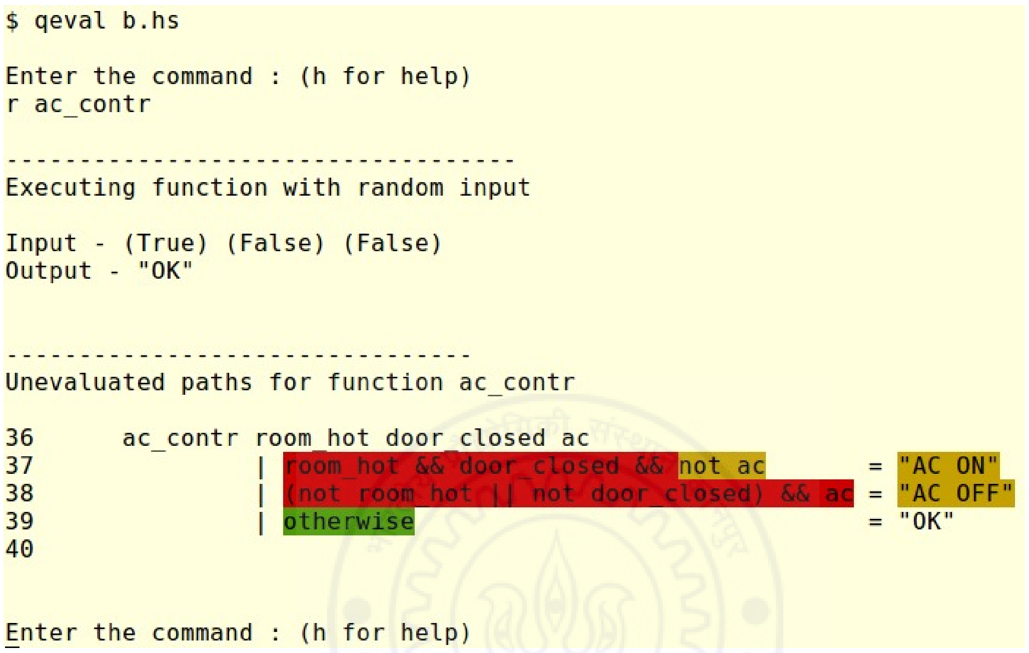
6.1 Comparison between QuickCheck and QuickEval

There are two drawbacks of QuickCheck. First, with random testing it is easy to catch bugs that are highly likely to occur, but it is very difficult to catch bugs that occur very infrequently. QuickCheck suffers from the same problem. Secondly, many times it is difficult and sometimes even impossible to generate properties of a function that can guarantee its correctness. Because QuickCheck works on boolean properties, it does not provide the output of the function which is being tested, but only success and failure message. However, in case the property fails to hold for a random input generated by QuickCheck, it is possible to get the (failing) input.

QuickEval differs from QuickCheck in that it does not require any property to test a function and that it reports the output of the function corresponding to the input provided (randomly or by user). This feature is useful for the novice users as they can quickly learn the behavior of the program using the tool and get help in understanding the Haskell programming language. For expert users, it helps in observing the inputs and corresponding outputs as applied to the function to be

```
ac_contr :: Bool->Bool->Bool->String
ac_contr room_hot door_closed ac
         | room_hot && door_closed && not ac        = "AC ON"
         | (not room_hot || not room_closed) && ac = "AC OFF"
         | otherwise                                = "OK"
```

Figure 6.1: `ac_contr function, adapted from Godefroid et. al. [12]`



```
$ qeval b.hs

Enter the command : (h for help)
r ac_contr

-----------------------------------
Executing function with random input

Input - (True) (False) (False)
Output - "OK"


--------------------------------
Unevaluated paths for function ac_contr

        ac_contr room_hot door_closed ac
                 | room_hot && door_closed && not ac        = "AC ON"
                 | (not room_hot || not door_closed) && ac = "AC OFF"
                 | otherwise                                = "OK"


Enter the command : (h for help)
```

Figure 6.2: `The function ac_contr tested with random input`

tested. However, this also means that the verification of the output has to be done externally.

To see how QuickEval is different from QuickCheck, consider the Haskell function `ac_contr` defined in Figure 6.1 (this function is adapted from a C function given by Godefroid et. al. [12]). The function checks different conditions for an AC controller and gives the output in the form of a status message. It is difficult to test this function with QuickCheck for the following 2 reasons: 1) It is not easily clear what properties describe the correct behavior of the controller, and 2) without looking at the inputs and the corresponding outputs, it is difficult to ascertain the correctness of the behavior of the function. With QuickEval, the user will get the actual output of the function i.e. the message corresponding to AC condition, for any given input conditions. One such interaction is shown in Figure 6.2.

6.2 QuickCheck with Hat

Hat is a tracing tool that can be used to generate traces of evaluation of Haskell program. QuickCheck and Hat together can be used to find out errors in a program more effectively [7]. This combination is very practical as testing with QuickCheck can find out one (or more) failing cases and then Hat can be used to find the cause of those errors. The combination is limited by the same issues as discussed for QuickCheck (Section 6.1). We believe that QuickEval can be integrated with Hat easily, and the integration will enable the user to discover hard to find bugs and their causes.

6.3 Comparison between QuickEval and HPC

HPC is not an interactive tool. HPC does not provide the feature of the random generation of input. With HPC, user always have to provide the input. With QuickEval, the user can either supply the inputs or randomly generate them. Thus, QuickEval is more interactive than HPC. Note that QuickEval borrows a lot from the implementation of HPC, and can be used in a mode that mimics HPC.

Chapter 7

Conclusion and Future Work

7.1 Conclusions and Future Work

In this report we presented QuickEval, a tool for testing Haskell programs based on coverage. The tool helps user in understanding the behavior of a program using random test generation along with user-specified tests. The proposed usage of QuickEval includes use by novice users to understand programs, by expert users to test programs and to generate test-suites automatically , and by teachers of Haskell language to evaluate assignments by creating and dynamically updating just enough test cases to test all submissions.

QuickEval is still a work in progress. To make it more effective, we plan to add direct automated random testing (DART [12]) technique to our tool. DART is a testing technique (and also a tool of the same name) used for directed automatic testing of the programs written in imperative languages. It works in three phases

1. Identification of types of arguments of function which is to be tested;
2. Automatic generation of random values for these arguments to perform the random testing; and
3. Automatic generation of new test input values to direct the execution of program to unexecuted paths.

QuickEval already implements the first two phases of DART: it can identify the types of arguments of function and automatically generate the random values for these arguments to perform the random testing. QuickEval also stores the unevaluated expressions or expressions with `False` values in one file for further processing. Third phase of QuickEval differs from DART as the user can not direct the input automatically but has to generate it manually. Improving this part, will make QuickEval completely automatic and will require less user interaction to generate test suite. This will make it more effective for large programs with many functions.

QuickEval can also be improved to check the output of the assignment solutions automatically. This task can be performed in following phases.

1. Teacher would create the master solution for assignment problems
2. QuickEval will generate the input values (randomly or manually) to test the function from student's assignment solution such that all the paths in function will be covered. The output values will be stored in one file.
3. QuickEval will use the same set of input values to test the same function from master solution and store the output values in one file
4. QuickEval will compare these two output files and produce the result as *pass* or *fail*.

Such type of improvement in QuickEval will minimize the manual checking of output and will further reduce the assignment evaluation time.

7.2 Acknowledgments

We are grateful to Andy Gill and Colin Runciman for sharing source code of HPC-0.4 with us.

Bibliography

[1] John W. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[2] Kent Beck. *Extreme Programming Explained.*

[3] Richard S. Bird and Philip Wadler. *Introduction to functional programming.* Prentice Hall International series in computer science. Prentice Hall, 1988.

[4] Check. A Unit Testing Framework for C. `http://check.sourceforge.net/`, June 2012 (last accessed).

[5] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, hat and hood - a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Implementation of Functional Languages*, pages 176–193, 2000.

[6] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[7] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and tracing lazy functional programs using quickcheck and hat. In *Advanced Functional Programming*, pages 59–99, 2002.

[8] Computer-Assisted Assessment. Implementing Learning Technology. `www.icbl.hw.ac.uk/ltdi/implementing-it/using.html`, May 2012 (last accessed).

[9] CUnit. A Unit Testing Framework for C. `http://cunit.sourceforge.net/`, June 2012 (last accessed).

[10] Andy Gill. Debugging haskell by observing intermediate data structures. *Electr. Notes Theor. Comput. Sci.*, 41(1):1, 2000.

[11] Andy Gill and Colin Runciman. Haskell program coverage. In *Haskell 2007*, pages 1–12, 2007.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI 2005*, pages 213–223, 2005.

[13] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[14] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *SIGPLAN Notices*, 27(5):1–, 1992.

[15] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[16] HUnit. Haskell Unit Testing. `http://hunit.sourceforge.net`, March 2012 (last accessed).

[17] Graham Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[18] Lasse Koskela. Introduction to Code Coverage. `http://www.javaranch.com/journal/2004/01/IntroToCodeCoverage.html`, May 2012 (last accessed).

[19] Henrik Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *J. Funct. Program.*, 11(6):629–671, 2001.

[20] Catherine Oriat. Jartege: A tool for random generation of unit tests for java classes. In *QoSA/SOQUA*, pages 242–256, 2005.

[21] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.

[22] Parasoft. Jtest. `http://www.parasoft.com/jsp/products/jtest.jsp`, June 2012 (last accessed).

[23] Steve Cornett. Coden Coverage Analysis. `http://www.bullseye.com/coverage.html#morell1990`, May 2012 (last accessed).

[24] Tutorial. Learn You a Haskell for Great Good. `learnyouahaskell.com`, May 2012 (last accessed).

[25] Kartick Vaddadi. Improving haskell debuggers. *MTech Thesis Dissertation.*

[26] Philip Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, 1998.

Appendix A

QuickCheck Implementaion

This chapter discusses the implementation of QuickCheck and material here is heavily borrowed from [6].

A.1 Generation of Arbitrary data

In QuickCheck random test data is generated depending on the type. If type is an instance of `Arbitrary` class, then only it is possible to generate random test values of that type.

```
class Arbitrary a where
   arbitrary :: Gen a
```

For those types which are instances of this class, data generators are provided by QuickCheck. Here `Gen a` is an abstract type and it represents the data generator for type `a`.

```
newtype Gen a = Gen (Rand -> a)
```

`rand` is random number seed.

Primitive generator function is defined as

```
choose :: (Int, Int) -> Gen Int
```

`choose` function picks up a random number in an interval. Data generators of different types are defined in terms of `choose`. For example, generators for integers and pairs are defined as follows.

```
instance Arbitrary Int where
   arbitrary = choose (-20, 20)


instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
   arbitrary = liftM2 (,) arbitrary arbitrary
```

For most of the Haskell primitive types, QuickCheck has declared such instances.

A.2 Data generation of User-Defined Types

QuickCheck can not generate test cases for user defined data types. To generate test values for such data types user has to create the instance of `Arbitrary` class for that particular data type. QuickCheck also provides the combinators for programmers to define data generators for user-defined data types.

The following example shows the use of combinator `oneof` to define the data generator.

```
data Fruits = Apple | Orange | Mango


instance Arbitrary Fruits  where
   arbitrary = oneof
      [return Apple, return Orange, return Mango]
```

A.3 Implementation of QuickCheck

To handle the functions with varying number of arguments and different result types, QuickCheck defines new type `Property` and class `Testable`.

```
newtype Property = Prop (Gen Result)


data Result =

Result {ok :: Maybe Bool, stamp :: [String], arguments :: [String]}


class Testable a where

   property :: a -> Property
```

Here `Property` plays the multiple roles. It handles boolean results of the properties of the function, the classification of test data and the arguments used in the generated test cases. Following examples show the role of `Property`. In the first example type `Bool` is tested and in second example the function for which arguments are to be generated is tested.

```
instance Testable Bool where

   property b = Prop (return (resultBool b))


instance (Arbitrary a, Show a, Testable b) =>

                           Testable (a->b) where

   property f = forall arbitrary f
```

Using the function `property`, the type of `quickCheck` becomes as follows.

```
quickCheck :: Testable a => a -> IO ()
```

Appendix B

QuickEval Implementation

This chapters discusses the implementation of QuickEval. To test the function with QuickEval and to display the coverage information we have modified some part of the source code of HPC. Generation of instrumented program, generation of tix and mix files are similar to that of HPC. Other changes are discussed in the following sections.

B.1 Generation of annoted source code of function

As a output QuickEval displays the source code of a function under test in different color patterns. This has been achieved by making some changes in main program for the hpc-markup tool of the original HPC package. In QuickEval package this program is renamed as Main_text.hs

The original program of hpc-markup, generates a list called as `info` by combining the data from tix and mix files. The entries in list `info` contains the positions of the expressions in Haskell program and also their evaluation status.

Using this list `info` , we created another list called `findModPos` which contains the positions of all functions in program and their evaluation status as (IsTicked or NotTicked).

```
findModPos [] = []


findModPos ((pos, theMark , TopLevelBox mods) : xs)
  | not ((head mods) ‘elem‘ ["strRes", "exec_func"])
                  = (pos , theMark) : findModPos xs
  | otherwise = findModPos xs


findModPos (_ : xs) = findModPos xs
```

To display the source code of function, it is necessary to find the starting and ending position of a function. Function `firstLastPos` finds the start and end positions of only those functions which were evaluated during the program execution(marked as Isticked).

```
firstLastPos []  = []


firstLastPos [x] = []


firstLastPos ((a_pos,IsTicked):(b_pos,_): xs)
              =  (a_pos, new_pos) : firstLastPos xs
              where (w, x, y, z) = fromHpcPos b_pos
                    new_pos      = toHpcPos (w -1, x, y, z)


firstLastPos (_ : b : xs) = firstLastPos (b:xs)
```

Now, every position of expression in list `info` is checked against the starting and ending positions of the functions which were evaluated during the program execution and theses expressions with their evaluation status are saved in another list. This newly created list is used to mark the expressions with different colors according to their evaluation status.

```
positions [] = []


positions ((pos, TickedOnlyFalse, _) : xs)
    | isPart (first_ele $ fromHpcPos $ pos) modPos
         = (pos, TickedOnlyFalse) : positions xs
    | otherwise = positions xs


positions ((pos, NotTicked, _) : xs)
    | isPart (first_ele $ fromHpcPos $ pos) modPos
         = (pos, NotTicked) : positions xs
    | otherwise = positions xs


positions ((pos, TickedOnlyTrue, _) : xs)
    | isPart (first_ele $ fromHpcPos $ pos) modPos
         = (pos, TickedOnlyTrue ) : positions xs
    | otherwise = positions xs
positions ((pos, _, _) : info) = positions info
```

QuickEval displays the coverage information in the accumulated form. If any expression is showed as unevaluated (marked in yellow) color, it means that this expression is not evaluated in any previous iteration. To achieve this feature, QuickEval stores the last output of function in a file. This output is compared with current output of function. If any expression is not evaluated in current output and it is also not evaluated in last output, in that case only it is marked as unevaluated. Otherwise if expression is unevaluated in current output, but evaluated in last output then the expression is displayed as evaluated. Same scenario is used to mark the expressions as always false. Program `Output.hs` defines various functions and data types to achieve this feature.

The expressions which are evaluated as `False` are saved in separate file.

```
grabReqText [] = []

grabReqText ((pos, TickedOnlyFalse) : as)
          = (grabHpcPos hsMap pos) : grabReqText as

grabReqText ((pos, _) : as) = grabReqText as

getText = unlines . grabReqText

--store the expressions with False boolen value
--in a file for further processing
writeFile (".auto/" ++ "result_" ++ head modNames ) $
"\n" ++ getText accum_info ++ "\n"
```

B.2 Compilation and sequence of execution

`quickeval` script handles the compilation and execution of different modules related to QuickEval. QuickEval creates two copies of the program under test. Original program is appended with two different `main` functions. Depending upon the type of input generation (random or manual) one of the two copies of original program are used for execution. Both `main` functions (appended to original program) call the pilot function (`__function_name__`) which is replaced by the actual function which is to be tested with QuickEval. For the random generation of input values `main` function also calls the `quickC` function.

Each copy of the original program is converted to instrumented program using tool `HpcTrans`. Along with the instrumented programs mix files are also generated for both copies. These instrumented programs are then compiled and executed to display the output of function and also to generate the tix file. When function is called with bulk of input values (using command ’r’ and count or using command 'f') then it is necessary to display all the functions which are called for these input values. To achieve this feature tix file for last input values is saved and it is merged with current tix file. This is done by the program `tixMerge`

Finally this tix file and function name is provided to the `Main_text` script, which gives the output in the form of annoted source code of all the functions which were called during that particular iteration.