Systematic Generation of Glass-Box Test Cases for Functional Logic Programs

Herbert Kuchen[1] and Sebastian Fischer[2]

[1]University of Münster, Germany
[2]University of Kiel, Germany

PPDP 2007, Wroclaw, Poland

I'm going to present joint work with Herbert Kuchen from Münster, Germany. We developed an approach to generate test-cases for glass-box testing declarative programs.
As opposed to black-box testing, glass-box testing refers to an approach that considers the implementation of the function under test. Current tools for testing functional programs are based on black-box testing because they generate test cases based on types and specifications.
Although we implemented our approach for the functional logic programming language Curry, it could be transfered to functional programs too.

Can you spot a bug?

```
data Nat = One | O Nat | I Nat

compare :: Nat -> Nat -> Ordering
compare One   One     = EQ
compare One   (O _)   = LT
compare One   (I _)   = LT
compare (O _) One     = GT
compare (O x) (O y)   = compare x y
compare (O x) (I y)   | cmpxy == EQ  = LT
                      | otherwise    = cmpxy
 where cmpxy = compare x y
compare (I _) One     = GT
compare (I x) (O y)   | cmpxy == EQ  = GT
                      | otherwise    = cmpxy
 where cmpxy = compare x y
compare (I x) (I y)   = compare y x
```

This is a Curry program. In Curry, datatypes can be defined by the keyword `data` followed by a list of constructors with argument types separated by vertical bars. The datatype shown here represents positive integers in binary notation.

Functions can be defined in Curry by multiple rules with pattern matching. The first line of the definition is an optional type signature that is inferred by type inference, if omitted. It states that `compare` is a binary function that takes two positive integers and returns a value that specifies their ordering.

You are not supposed to understand the definition of this function. Just note, that it might be difficult to find bugs in functions only by inspecting their source code.

Let's test!

```
kicsi compare.curry

compare> let n,m free in (n,m,compare n m)
(1,1,EQ) More? yes
(1,2,LT) More? yes
(1,3,LT) More? yes
(2,1,GT) More? yes
(3,1,GT) More? yes
(1,4,LT) More? yes
(1,6,LT) More? yes
(1,5,LT) More? yes
(1,7,LT) More? yes
(2,2,EQ) More? yes
(2,3,LT) More? yes
(4,1,GT) More? yes
(6,1,GT) More? no
```

In Curry, we can apply a function to free variables in order to observe its input-output relation. In fact, free variables serve very well as built-in test-case generators.
For example, we can apply the function `compare` to two free variables. A Curry system can then print one result after the other interactively.

For this talk, I generated 13 results due to obvious space restrictions.

It works! Does it?

Curry comes with **built-in test case generator**:

Narrowing

No bug visible in first 13 test cases

How do I know that my code is correct?

- Which parts where executed?
- Was every possible branch taken?

None of the shown results exposes a bug. Can we conclude that the definition of `compare` is correct?
Tests can never show the correctness of programs. They show bugs but not their absence.
In order to gain confidence in the quality of the produced set of test cases, we want to incorporate information about which parts of the definition of `compare` where executed and whether every possible branch was taken during the computation of one specific result. We need to consider the implementation of `compare` in order to decide whether it was tested thoroughly enough.

Tool Support

Benefits of tool support:

- Generate as much test cases as **necessary**
- Eliminate **redundant** test cases

How much test cases are **necessary**?

- Decide w.r.t. **code coverage**
- Different criteria possible (this talk presents two)

When is a test case **redundant**?

- No additional coverage
- set-covering problem (NP-complete!)

We have developed a tool that uses narrowing to generate as much test cases as necessary and eliminates redundant test cases that are equivalent to other already generated tests.
What does *as much as necessary* mean? We want to ensure that the code of the tested function is covered completely. There are different criteria possible for code to be *covered completely*. This talk presents two.
When is a test case redundant? If we associate a set of covered items to every test case, we can eliminate those with coverage that is already obtained by other test cases. To eliminate redundant test cases, we need to solve an instance SET-COVER problem, which is NP-complete. Therefore, we are happy with a heuristic that computes a small set of test cases and do not strive for the smallest possible.

Global Branch Coverage (GBC)

Every pattern matching introduces a branching

```
[] ++ ys = ys                                    app1
(x:xs) ++ ys = x : (xs++ys)                      app2

reverse [] = []                                  rev1
reverse (x:xs) = reverse xs ++ [x]               rev2
```

Branches **reachable** from `reverse`: app1,app2,rev1,rev2

- all branches of `reverse` and all called functions
- $\rightsquigarrow$ dependency graph

Let's continue with discussing different coverage criteria for declarative programs.
Global Branch Coverage is a straightforward attempt to monitor which parts of a function have been executed. For every function, we label each branch that is introduced by pattern matching and demand every branch to be executed by at least one test case.
Here we show the definition of the naive `reverse` function and the append function on lists, which is written in infix notation. Each function has two branches which are labeled *app1*, *app2*, *rev1* and *rev2*.
Usually, very few test cases suffice to cover the branches of a single function. In order to test more thoroughly, we also want to execute the branches of other functions reachable in the dependency graph of the program, i.e., those that are directly or indirectly called by the tested function.

In the example, the branches *app1*, *app2*, *rev1* and *rev2* are reachable from the definition of `reverse`.

Global Branch Coverage (GBC)

```
[] ++ ys = ys                                  app1
(x:xs) ++ ys = x : (xs++ys)                    app2

reverse [] = []                                rev1
reverse (x:xs) = reverse xs ++ [x]             rev2
```

test case	covered branches
`reverse [] = []`	rev1
`reverse [x] = [x]`	app1,rev1,rev2
`reverse [x,y] = [y,x]`	app1,app2,rev1,rev2
`reverse [x,y,z] = [z,y,x]`	app1,app2,rev1,rev2

Table: test cases for `reverse`

This table shows some possible test cases for the function reverse along with the covered branches.
We can observe that we need a list with at least two elements to cover the second branch of `(++)`. On the other hand, a test case with a list of two elements also covers all other reachable branches, so the other test cases are redundant.

GBC for `compare`

generating test cases for function compare..

```
obtained coverage with 38 generated test cases
eliminating redundant test cases..

compare 6 2 = GT
compare 2 1 = GT
compare 1 3 = LT
compare 3 3 = EQ
compare 3 2 = GT
compare 2 3 = LT
compare 3 4 = LT
compare 6 3 = GT
```

- 8 test cases selected out of 38
- no bug visible

If we ask our tool to generate a set of test cases w.r.t. Global Branch Coverage, 38 test cases are generated and 8 of them selected after redundancy elimination.
The test cases are shown on the slide. Convince yourself, that none of them exposes a bug.
But was the definition of `compare` really correct?

compare is a complex function!

9 rules, 4 recursive calls

```
compare One   One     = EQ
compare One   (O _)   = LT
compare One   (I _)   = LT
compare (O _) One     = GT
compare (O x) (O y)   = compare x y
compare (O x) (I y)   | cmpxy == EQ  = LT
                      | otherwise    = cmpxy
 where cmpxy = compare x y
compare (I _) One     = GT
compare (I x) (O y)   | cmpxy == EQ  = GT
                      | otherwise    = cmpxy
 where cmpxy = compare x y
compare (I x) (I y)   = compare y x
```

compare is a quite complex function with 9 rules and 4 recursive calls. Can we be sure that the behavior of compare was thoroughly tested because every branch was taken during the evaluation of one the 8 test cases?

Simple Example

Although every branch is covered,
a functions behavior might not be tested sufficiently:

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

A single test case

```
test [x] [] = True
```

covers all branches of `reverse` and `(++)`

This test also succeeds for an identity function called `reverse`

We should be sceptical about this. I'll show you why, with are simpler example.
The shown property states that the concatenation of two reversed lists is equal to the reversed concatenation of the lists in reversed order. The definition of the function `test` includes three calls to `reverse` and two to `(++)`. Observe that a single test case with a singleton and an empty list covers all branches of `reverse` and `(++)`.
This test also succeeds for an identity function called `reverse`. The property is not true in general for the identity function. So, the behavior of `reverse` is not tested sufficiently by Global Branch Coverage.
That's because different calls to the same function only need to cover the branches of this function together, not individually.

Function Coverage (FC)

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

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

rev1, rev2, **rev1, rev2**, **rev1, rev2**, **rev1, rev2**

- local criterion
- behavior tested more thoroughly

test case	covered branches
`test []  []  = True`	**rev1**, **rev1**, **rev1**
`test [x] []  = True`	rev1, **rev2**, **rev1**, **rev2**
`test []  [x] = True`	rev1, **rev1**, **rev2**, **rev2**
`test [x] [y] = True`	rev1, rev2, **rev2**, **rev2**, **rev2**

Table: test cases for `test`

To overcome these deficiencies, we present Function Coverage. Function Coverage differs from Global Branch Coverage in two respects:

1. the branches of the tested function must be covered individually by each call syntactically present in the program, and
2. due to the more thorough testing we do not need to cover branches of reachable functions

In contrast to Global Branch Coverage, Function Coverage is a local criterion.
The table shows again some test cases along with the covered items. For the grey call to `reverse`, the second branch is only covered by the last test case. Two of the shown test cases can be dropped because all covered branches are also covered by other test cases. Which?

FC for `compare`

```
generating test cases for function compare..

obtained coverage with 294 generated test cases
eliminating redundant test cases..

compare 3 1 = GT
compare 2 1 = GT
compare 1 3 = LT
compare 1 2 = LT
compare 1 1 = EQ
compare 5 3 = LT
...
```

- 36 test cases selected out of 294
- 8 test cases expose a bug

If we use our tool to generate test cases w.r.t. Function Coverage, 294 tests are generated and 36 remain after redundancy elimination.
For Global Branch Coverage we had 8 test cases out of 38, so Function Coverage really enforces a more thorough testing. Each of the four recursive calls to `compare` needs to cover all branches individually.
In fact, 8 of the 36 test cases expose a bug in the definition of compare which remained undetected before.

Bug in last rule!

```
compare One   One     = EQ
compare One   (O _)   = LT
compare One   (I _)   = LT
compare (O _) One     = GT
compare (O x) (O y)   = compare x y
compare (O x) (I y)   | cmpxy == EQ  = LT
                      | otherwise    = cmpxy
 where cmpxy = compare x y
compare (I _) One     = GT
compare (I x) (O y)   | cmpxy == EQ  = GT
                      | otherwise    = cmpxy
 where cmpxy = compare x y

compare (I x) (I y)   = compare y x
```

If we analyze the coverage information for the falsifying test cases, we see that they all use the last rule of compare. This rule is indeed faulty, as a close inspection reveals: the arguments of the recursive call have been swapped.

Not detected with GBC

```
compare (I x) (I y) = compare y x

compare 6 2 = GT
compare 2 1 = GT
compare 1 3 = LT

compare 3 3 = EQ

compare 3 2 = GT
compare 2 3 = LT
compare 3 4 = LT
compare 6 3 = GT
```

```
compare (I One) (I One) -> compare One One -> EQ
```

Why was this bug not detected before?
Let's reconsider the test cases that were generated w.r.t. Global Branch Coverage: The highlighted test case is the only one that has executed the last rule of `compare`. If we take a closer look at its execution, we recognize that the recursive call in the last rule was only executed with the arguments `One` and `One`. Obviously, swapping these remains undetected. Other calls to this specific recursive call are not enforced by Global Branch Coverage, so we miss the error.

Program Transformation

Augmented program collects information when executed

Dirty approach

```
collect :: [Info] -> a -> a
collect info x = unsafePerformIO(writeInfo info>>return x)
```

Pure approach

```
data I a = ...
collect :: [Info] -> I a -> I a
```

`I a` is `a` with attached information

`reverse :: [a] -> [a]` $\rightsquigarrow$ `reverse :: I [a] -> I [a]`

Details tricky because of laziness

We have seen two different coverage criteria for declarative programs – Global Branch Coverage and Function Coverage. The remainder of this talk shows how coverage information can be collected during the computation.
We need to modify the source program such that information about the executed branches is collected. A simple approach is to use `unsafePerformIO` to record the information as a side effect.
In order to get a pure implementation of our tool, we developed a different approach. The idea is to replace every value of type `a` with a wrapped value of type `I a` with attached information. The concrete type `Info` of collected information is irrelevant here.

Preserving Laziness

Info Trees

```
data I a = I a (Tree [Info])   -- attach info tree
data Tree a = Tree a [Tree a]  -- unranked trees

wrap :: a -> I a
wrap x = I x (Tree [] [])

ap :: I (a -> b) -> I a -> I b
ap = ...
```

- attach tree of information to every value
- `wrap` attaches empty tree
- ap used to build wrapped compound terms

The details of the transformation are nontrivial because we are not allowed to destroy the laziness of the original program. Otherwise, terminating programs could be translated into nonterminating programs and coverage information may not reflect a lazy execution.
In order to be able to discard information of sub terms, if they are not used by the computation, we associate a tree of information to every value. The purpose of these trees will be apparent in a minute. We provide functions `wrap` and ap to build compound terms with attached information trees.

Preserving Laziness

```
1 -> wrap 1 = I 1 (Tree [] [])

[] -> wrap [] = I [] (Tree [] [])

[1,2] = (1:(2:[])) = (:) 1 ((:) 2 [])

-> wrap (:) `ap` wrap 1
            `ap` (wrap (:) `ap` wrap 2
                           `ap` wrap [])

= I [1,2] (Tree [] [Tree [] []
                    ,Tree [] [Tree [] []
                             ,Tree [] []]])
```

term structure $\longleftrightarrow$ info tree structure
$\leadsto$ associate information to constructors

Here, we have a few examples of wrapped terms and how they are computed. Constructors without arguments are simply wrapped using the function `wrap`. Compound terms are created using both `wrap` and `ap`. In the resulting wrapped values, the structure of the attached info tree reflects the term structure of the associated value. Therefore, we can associate specific information to every single constructor of a compound term in the associated tree.

Preserving Laziness

- root of info tree
  - information collected during computation of head constructor
- child trees
  - information collected during computation of sub terms
- one-to-one correspondence: sub terms $\leftrightarrow$ child trees

pattern matching: discard info associated to unused sub terms

essential to preserve laziness

The root of the tree stores information associated to the head constructor of a value. The subtrees store information for each sub term of the head constructor.

Due to the tree structure we can discard information associated to unused sub terms during pattern matching.

Example Transformation

Flat append function

```
(++) :: [a] -> [a] -> [a]
l ++ ys = case l of
            []     -> ys                                app1
            (x:xs) -> x : (xs++ys)                      app2
```

pattern matching by case expression

different branches easily recognized

As an example for our program transformation, consider a flat version of the append function, where multiple rules with pattern matching have been replaced by a single rule and a case expression.

Example Transformation

Change type

```
(++) :: I [a] -> I [a] -> I [a]
l ++ ys = case l of
            []      -> ys                         app1
            (x:xs)  -> x : (xs++ys)               app2
```

- not well typed anymore
- first argument cannot be matched against list constructors
- $\rightsquigarrow$ modify pattern matching

We want to generate a function that takes and returns wrapped lists instead of lists. Therefore, we need to adapt the pattern matching.

Example Transformation

Match wrapped value

```
(++) :: I [a] -> I [a] -> I [a]
l ++ ys = case l' of
            []      -> ys                                  app1
            (x:xs)  -> x : (xs++ys)                        app2
 where
  I l' (Tree info ts) = l
```

- `info` of `l` discarded
- $\rightsquigarrow$ `collect` it for current result

We match the wrapped list in a local declaration and extract the list `l'` to match it with the original case expression.
We should not discard the information collected for the head constructor of the list, but should collect it for the result of the current computation.

Example Transformation

Collect root information

```
(++) :: I [a] -> I [a] -> I [a]
l ++ ys
  = collect info
     (case l' of
        []      -> ys                                  app1
        (x:xs)  -> x : (xs++ys))                       app2
 where
  I l' (Tree info ts) = l
```

```
collect info (I x (Tree info' ts))
  = I x (Tree (info++info') ts)
```

We need to collect this information, as the head constructor is demanded by the current computation. We do this with the function `collect` which simply attaches the given information to the root of the computed info tree.

Example Transformation

Collect branch labels

```
(++) :: I [a] -> I [a] -> I [a]
l ++ ys
  = collect info
     (case l’ of
        []      -> collect ["app1"] ys
        (x:xs) -> collect ["app2"] (x : (xs++ys)))
 where
  I l’ (Tree info ts) = l
```

- type `Info` of collected information represents branch labels
- $\rightsquigarrow$ associate child trees to sub terms

In this example, the type `Info` of collected information represents branch labels. We introduce additional calls to `collect` in order to record the executed branches.

What should we do with the child trees of the info tree associated to the matched list?

Example Transformation

Associate child trees

```
(++) :: I [a] -> I [a] -> I [a]
l ++ ys
  = collect info
     (case l’ of
        [] -> collect ["app1"] ys
        (x’:xs’)
           -> collect ["app2"]
               (let x  = I x’  (ts!!0)
                    xs = I xs’ (ts!!1)
                 in x : (xs++ys)))
 where
  I l’ (Tree info ts) = l
```

We attach it to the corresponding sub terms of the matched list in a local declaration.

Example Transformation

Transformed append function

```
(++) :: I [a] -> I [a] -> I [a]
l ++ ys
  = collect info
     (case l’ of
        [] -> collect ["app1"] ys
        (x’:xs’)
           -> collect ["app2"]
               (let x  = I x’  (ts!!0)
                    xs = I xs’ (ts!!1)
                 in wrap (:) ‘ap‘ x ‘ap‘ (xs++ys)))
 where
  I l’ (Tree info ts) = l
```

Finally, we wrap the constructed value in the second rule using `wrap` and `ap` and the transformation is finished.

Example Applications

Which branches are executed?

```
wrap [] ++ (wrap (:) ‘ap‘ wrap 42 ‘ap‘ wrap [])
 = I [42] (Tree ["app1"] [Tree [] [],Tree [] []])

(wrap (:) ‘ap‘ wrap 47 ‘ap‘ wrap [])
 ++ (wrap (:) ‘ap‘ wrap 11 ‘ap‘ wrap [])
 = I  [47,11]
      (Tree ["app2"]  [Tree [] []
                      ,Tree ["app1"]  [Tree [] []
                                      ,Tree [] []]])
```

sufficient to compute GBC
Paper: extension to FC and higher-order functions

Now, we can observe for applications of (++) which branches are taken during its execution. The second example shows that only the second branch needs to be executed in order obtain the head constructor of the result. If we demand the whole list, both branches are executed.
The presented transformation suffices to collect information w.r.t. Global Branch Coverage. In the paper, we show how to extend it to support Function Coverage and higher-order functions.

Conclusions

Narrowing serves well to generate – a lot of (redundant) – test cases

More confidence with proper **code coverage**

- Global Branch Coverage
- Function Coverage

Program Transformation to collect coverage info

- no impure features
- preserves laziness

Test-Case Generation not

- difficult,
- tedious and
- error-prone

anymore

Experiments

	GBC		FC	
	# test cases	seconds	# test cases	seconds
`add`	6 (15)	0.04+0.04	23 (>219)	0.83+1.63
`sub`	7 (20)	0.07+0.08	17 (137)	0.55+0.27
`mul`	4 (>141)	0.52+0.78	5 (17)	0.03+0.02
`cmpNat`	8 (38)	0.09+0.21	36 (294)	1.50+8.47
`(+)`	20 (168)	0.79+4.02	11 (26)	0.04+0.04
`(-)`	19 (>212)	1.11+6.55	3 (5)	0.01+0.01
`(*)`	8 (>213)	0.58+1.26	7 (17)	0.01+0.01
`cmpInt`	15 (110)	0.24+0.54	9 (17)	0.02+0.03

Figure: Number of test cases generated and required runtimes for both considered coverage criteria, GBC and FC, for selected operations of the arithmetic library.