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

Sebastian Fischer
Department of Computer Science
Christian-Albrechts-University of Kiel, Germany
sebf@informatik.uni-kiel.de

Herbert Kuchen
Department of Information Systems
University of Münster, Germany
kuchen@uni-muenster.de

Abstract

We employ the narrowing-based execution mechanism of the functional logic programming language Curry in order to automatically generate a system of test cases for glass-box testing of Curry programs. The test cases for a given function are computed by narrowing a call to that function with initially uninstantiated arguments. The generated test cases are produced w.r.t. a selected code-coverage criterion such as control-flow coverage. Besides an adaption of the notion of control-flow coverage to functional (logic) programming, we present a novel coverage criterion for this programming paradigm. A particular difficulty of the adaption is the handling of laziness.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages

Keywords Testing, Declarative Programming, Code Coverage

1. Introduction

Curry [14] is a programming language that aims at integrating different declarative programming paradigms into a single programming language. Its syntax is similar to Haskell [20] but it uses a different evaluation mechanism [2] and supports free variables and nondeterministic computations like logic languages. Declarative programming languages offer a high degree of abstraction which eases the development of complex systems. They help to write more readable and re-usable code [15] which, however, can still contain errors. Ultimate confidence in the correctness of a program can only be achieved by proofs with regard to a complete and formal specification. Declarative languages are usually based on a formal semantics, which helps to formally certify some properties of declarative programs. However, proving complex systems correct turns out to be too difficult and time consuming, even in the context of declarative programming. Therefore, heuristic approaches to expose errors gain importance also in this area. Recently, techniques have been proposed to systematically debug programs that are known to be erroneous [11, 8, 6, 5, 10]. However, little can be found in the literature about the systematic testing of functional logic programs. This paper intends to help filling this gap. As the functional part of Curry is basically Haskell98 without type classes, we can also apply our tool to generate test cases for Haskell programs.

1.1 Testing

Approaches to software testing can be divided into black-box testing, where test cases are deduced from a specification without taking the concrete implementation into account, and glass-box testing[1], which aims at a systematic coverage of the code. Both approaches do not exclude each other, but can be combined. Black-box testing is often used to evaluate larger parts of an application. Glass-box testing is preferred for testing small program units with a complex algorithmic structure, which makes it hard to deduce all possible behaviors from the specification. Since declarative programs typically consist of a sequence of small function definitions, glass-box testing is even more suited for declarative languages than for imperative languages. Moreover, the lack of side effects renders it possible to test parts of an algorithm independently. We will focus on glass-box testing in this paper.

Regardless of the way test cases have been generated, they can not only be used for testing a program unit once. More importantly, they can be used for so-called regression testing, i.e., the suite of collected test cases is automatically processed in order to check whether some change in the program affects the already implemented functionality.

1.2 Related Work

Existing tools for testing functional programs generate random test cases or deduce them from a specification [12, 16], i.e., they are based on black-box testing. Since they do not take the implementation into account, they cannot ensure that all parts of the program are actually executed by the test cases. Hence, errors in uncovered parts of the program may remain undetected.

In [18, 17] an approach for generating glass-box test cases for Java is presented. Techniques known from logic programming are incorporated into a symbolic Java virtual machine for code-based test-case generation. A similar approach based on a Prolog simulation of a Java Virtual Machine is presented in [1]. A related approach to test-case generation for logic programs is discussed in [19]. Here, test cases are not generated by executing the program but by first computing constraints on input arguments that correspond to an execution path and then solving these constraints to obtain test inputs that cover the corresponding path.

We transfer the approach for Java presented in [18, 17] to the functional logic programming language Curry. However, instead of

[1] Glass-box testing is often called white-box testing, although one can of course not look inside a white box.

PPDP'07 July 14–16, 2007, Wroclaw, Poland.

extending an abstract machine by components for backtracking and handling logic variables, we can just employ the usual execution mechanism of Curry, since it already provides these features. Actually, this approach to test-case generation seems to be tailor-made for functional logic languages. Our approach is based on a transformation of Curry programs and, hence, not restricted to a specific Curry implementation.

Besides testing, there are different approaches for debugging functional logic programs, which are remotely related to testing. *Trace debuggers* [4, 8, 6] record information about a specific (usually erroneous) program run and present it to the user in a structured way. Different kinds of views can be employed to analyze the recorded program trace. Algorithmic debuggers ask the user a sequence of questions about the correctness of subcomputations in order to find an error in a program [10]. Algorithmic debugging can be implemented as an interactive view on a program trace. With tools for *observational debugging* [4, 5] the programmer can annotate her code with observer functions to record parts of the computation. With this approach, only selected parts considered interesting by the programmer are recorded, which saves memory. Both tracing and observational debugging are approaches to locate an already present error. They are not aiming at but can be combined with testing.

The remainder of this paper is structured as follows. In Section 2 we recall the functional logic programming language Curry. In Section 3 we discuss our approach in detail. We consider two different code-coverage criteria and explain the generation of test cases. In Section 4 we discuss a prototypical implementation of our test tool. In Section 5 we report on practical experience before we conclude and point out directions for future work in Section 6.

2. The Curry Language

Curry [14] is a declarative programming language that aims at integrating the most important declarative programming paradigms – functional and logic programming. Due to the lack of space, we only sketch it here. Details can be found in [14]. Curry extends Haskell [20] by partial data structures and uses a different evaluation mechanism. For each function the user can fix (by using an annotation) either *needed narrowing* [2] or *residuation* as evaluation mechanism. Needed narrowing corresponds to lazy evaluation in Haskell, except for the fact that unification rather than pattern matching is used for parameter passing. If an argument of a function contains free variables that are required by a pattern, this may cause them to be bound to some term. Residuation will suspend the computation in such a situation until some concurrent computation has bound the variable. If no free variable is required by a pattern during the computation, both needed narrowing and residuation will behave just as lazy evaluation in functional languages.

2.1 Datatypes and Function Declarations

Curry supports algebraic datatypes that can be defined by the keyword `data` followed by the newly introduced type and a list of constructor declarations separated by the symbol "`|`". For example, the following two declarations introduce the predefined datatypes for boolean values and polymorphic lists, respectively. The latter has a special syntax in Curry and is usually written as `[a]`.

```
data Bool    = True | False
data List a  = []   | a : List a
```

Type synonyms can be declared with the keyword `type`. For example, sets of natural numbers can be represented as sorted lists of pairwise distinct numbers represented as lists of integers:

```
type NumSet = [Int]
```

Curry offers the usual primitive functions on integers and comparison operators defined in terms of a primitive function `compare`:

```
(+), (-), (*), div, mod :: Int -> Int -> Int
(<),(<=),(==),(/=),(>=),(>) :: Int -> Int -> Bool

data Ordering = LT | EQ | GT
compare :: Int -> Int -> Ordering
```

In fact, `compare` is (ad hoc) polymorphic in Curry. However, for the purpose of this description, we consider it to be only defined on integers.

Curry functions are defined by rules that are selected nondeterministically. If more than one rule matches a call to a defined function, one matching rule is applied nondeterministically – not necessarily the topmost one. Depending on the implementation other rules are tried after backtracking (as in most Curry implementations) or in parallel. Like Haskell, Curry also supports `case` expressions that are evaluated top down.

As an example of a function declaration in Curry, consider an operation that inserts an integer into a set represented as a sorted list of pairwise distinct integers:

```
insert :: Int -> NumSet -> NumSet
insert n [] = [n]
insert n (x:xs) = case compare n x of {
  LT -> n:x:xs;
  EQ -> x:xs;
  GT -> x:insert n xs }
```

The first line of the function declaration is an optional type signature. If it is omitted, it is inferred by a type inference algorithm.

2.2 Narrowing

As mentioned above, Curry supports partial data structures that contain free variables. If an argument of a function call contains free variables, the unification of the left-hand side of a rule with the function call can cause each to be bound to the corresponding term of the left-hand side of the rule. Consider the function `size` that computes the cardinality of a finite set of numbers:

```
size []     = 0
size (x:xs) = 1 + size xs
```

If we call `size` with a free variable as argument, it nondeterministically computes any cardinality by binding the argument to a set of corresponding size:

```
> let xs free in size xs
Free variables in goal: xs
Result: 0, Bindings: xs=[]
Result: 1, Bindings: xs=[x0]
Result: 2, Bindings: xs=[x0,x1]
...
```

Note that there are infinitely many solutions to the goal and that the elements of `xs` are introduced as fresh variables. A *solution* consists of a constructor term as *result* and a *substitution* binding each free variable of the initial function call to a constructor term.

2.3 The Core Language FlatCurry

Every Curry program can be translated into a simplified core language FlatCurry, that is commonly preferred over Curry for analytical purposes because of its simplicity. The syntax of FlatCurry is depicted in Figure 1. We will describe our approach in terms of flat programs. A flat program is a list of function declarations and functions are defined by a single rule with pairwise distinct variables as arguments. The body of a function is an *expression*, which can be a

Program	P	::=	$D_1 \ldots D_m$	
Definition	D	::=	$f\ x_1\ \ldots\ x_n = e$	
Expression	e	::=	x	(variable)
		|	$c\ e_1\ \ldots\ e_n$	(constructor call)
		|	$f\ e_1\ \ldots\ e_n$	(function call)
		|	$let\ \overline{x_i = e_i}\ in\ e$	(local declarations)
		|	$e_1\ or\ e_2$	(disjunction)
		|	$case\ e\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$	(case distinction)
Pattern	p	::=	$c\ x_1\ \ldots\ x_n$	

Figure 1. Syntax of FlatCurry Programs

variable, a constructor or function application, a *let* expression, an *or* expression or a *case* expression. *Or* expressions nondeterministically evaluate to one of the arguments. The translation of the function `insert` introduced above into FlatCurry is:

```
insert n l = case l of {
  [] -> [n];
  (x:xs) -> case compare n x of
     { LT -> n:x:xs;
       EQ -> x:xs;
       GT -> x:insert n xs }}
```

The different rules of `insert` are combined into a single rule with variables as arguments.

Readers who are familiar with FlatCurry may miss the introduction of free variables. In fact we use a simplified version of FlatCurry which assumes that free variables have been eliminated by a transformation recently developed by Antoy and Hanus [3]. The main idea of this transformation is that free variables are replaced by generators for all values of the corresponding type.

3. Test-Case Generation

Before we explain our approach to generate test cases for Curry, let us briefly recall the basics of glass-box testing for imperative languages. For testing a function one collects a set of test cases which covers the possible behaviors of the function in a reasonable way. A *test case* is a pair of a function call and a corresponding expected result.

Since there are often infinitely many possible inputs and corresponding computation paths through a program, it is impossible to test all of them. But even if the number of paths was finite, many of them would cover the same control and data flow and would hence be equivalent from the point of view of testing. For cost effective testing, one is interested in a minimal set of test cases covering the code according to a selected coverage criterion.

Classical criteria known from testing imperative programs are coverage of the (nodes and) edges of the control-flow graph and the so-called def-use chain coverage [21]. The latter requires that each sequence of statements is covered, which starts with a statement computing a value and ends with a statement, where this value is used and where this value is not modified in between.

It is important to understand that no coverage criterion guarantees the absence of errors, including the criteria presented in this paper. They rather try to detect as many errors as possible with limited effort. It is always possible to find examples where a given coverage criterion fails to expose an error. Nevertheless coverage criteria are useful in order to find the majority of the errors, in particular in algorithmically complex code. Remaining errors can be eliminated, e.g., by black-box testing.

3.1 Code Coverage

Lazy declarative languages like Curry have no assignments and a rather complicated control-flow (due to laziness), which cannot easily be represented by a control-flow graph. Therefore, we cannot simply transfer the notions of code coverage from the imperative to the declarative world, but need adapted notions. Here, we will present two different coverage criteria: *Global Branch Coverage* (GBC) and *Function Coverage* (FC) which correspond both to variants of control-flow coverage in imperative languages. However, let us point out that our approach works with any coverage criterion. We only require that the coverage can be described by a set of *coverable items*. These items can represent control- and/or data-flow information.

3.1.1 Global Branch Coverage

In glass-box testing for imperative languages, typically only code sequences that are part of a single function or procedure declaration are considered. Due to control structures like loops present in imperative languages, there is often no need to consider more than one function to obtain interesting test cases.

In declarative programming, (recursive) function calls are used to express control structures and to replace loops. Since the functions in a declarative program are typically very small and consist of a few lines only, it is not sufficient in practice to cover only the branches of the function to be tested. Thus, we will aim at covering the branches of all the directly and indirectly called functions, too.

The main idea of this approach is that we label all the alternatives in *or* and *case* expressions with pairwise different labels and that we try to make sure that every labelled alternative will be executed at least once by some test case. This means that each alternative of an *or* expression and each e_i in a *case* expression

$$case\ e\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}$$

will be evaluated to head normal form. We extend the syntax of FlatCurry by labeled expressions that are written as

$$\langle l \rangle\ e$$

Here, l is a label and e an arbitrary FlatCurry expression. The labeling transformation σ depicted in Figure 2 shows how the labeling is done by a straightforward homomorphism on flat expressions. Initially, the transformation σ can be called with distinct labels for each defined function in order to obtain globally unique labels in a program. When applying σ to the empty sequence $\langle\rangle$ and the right-hand side of the `insert` function, we get:

```
insert n l = case l of {
  [] -> ⟨1⟩ [n];
  (x:xs) -> ⟨2⟩ case compare n x of {
    LT -> ⟨2.1⟩ n:x:xs;
    EQ -> ⟨2.2⟩ x:xs;
    GT -> ⟨2.3⟩ x:insert n xs }}
```

$$
\begin{array}{l}
\sigma(\langle l\rangle, x) = x \\
\sigma(\langle l\rangle, c\ e_1\ \ldots\ e_n) = c\ \sigma(\langle l.1\rangle, e_1)\ \ldots\ \sigma(\langle l.n\rangle, e_n) \\
\sigma(\langle l\rangle, f\ e_1\ \ldots\ e_n) = f\ \sigma(\langle l.1\rangle, e_1)\ \ldots\ \sigma(\langle l.n\rangle, e_n) \\
\sigma(\langle l\rangle, let\ x = e\ in\ e') = let\ x = \sigma(\langle l.0\rangle, e)\ in\ \sigma(\langle l.1\rangle, e') \\
\sigma(\langle l\rangle, e_1\ or\ e_2) = \langle l.1\rangle\ \sigma(\langle l.1\rangle, e_1)\ or\ \langle l.2\rangle\ \sigma(\langle l.2\rangle, e_2) \\
\sigma(\langle l\rangle, case\ e\ of\ \{p_1 \to e_1; \ldots; p_n \to e_n\}) \\
\quad = case\ \sigma(\langle l.0\rangle, e)\ of \\
\qquad \{p_1 \to \langle l.1\rangle\ \sigma(\langle l.1\rangle, e_1); \\
\qquad\quad \ldots \\
\qquad\ p_n \to \langle l.n\rangle\ \sigma(\langle l.n\rangle, e_n)\}
\end{array}
$$

Figure 2. Labeling Transformation

To test the definition of a function `f`, we can now compute the set of all functions reachable from `f` and generate test cases that cover all branches of these functions.

3.1.2 Function Coverage

Covering all branches of all reachable functions may produce a lot of overhead. Moreover, it may be impossible to cover all branches of the called functions, since the parameters passed to them do not permit this. If, for instance, the body of the considered function `f` contains a call `insert 1 []`, only branch $\langle 1\rangle$ of `insert` can be executed. Thus, we are interested in a criterion which focuses on the code of the function to be tested. As mentioned, a simple coverage of all branches of the considered function will expose too few errors in practice. Thus, we extend this approach slightly. We will ensure that in addition to all branches of the original call to the considered function, also all branches of all recursive calls to that function have to be executed. As shown in Section 5, the resulting criterion called *Function Coverage* (FC) works quite well in practice. If we assume that all functions called by the considered function `f` have been tested before, FC allows to find strictly more errors than GBC, since FC will then not only ensure that every branch is executed once, but that every branch will be checked in every call to the considered function.

As for GBC, it may be sometimes impossible to cover all branches for each recursive call, since the actual parameters do not permit this. In this case, we will confine ourselves to execute the reachable branches.

As a simple example consider the labeled definition of the append function `(++)` for lists:

```
l ++ ys = case l of {
  [] -> ⟨1⟩ ys;
  (x:xs) -> ⟨2⟩ x : (xs++xs) }
```

Here, the test case `[0] ++ [] = [0]` suffices to reach global branch coverage without exposing the error in branch $\langle 2\rangle$ (`xs++xs` should be `xs++ys`).

On the other hand, there are two calls of `(++)` to be covered with FC, the initial call and the recursive call in branch $\langle 2\rangle$. We require both calls to execute all branches of `(++)`. With the above test case, the recursive call does not execute branch $\langle 2\rangle$ and our tool also generates the test case `[0] ++ [1] = [0]` that exposes the error.

If we fail to fix the error and write `ys ++ xs`, then the test cases `[] ++ [] = []` and `[0] ++ [1] = [0,1]` suffice to fulfill both GBC and FC. None of the coverage criteria exposes this error which hints at the incomplete nature of every coverage criterion discussed at the beginning of this section.

In Section 5, we will compare the presented criteria experimentally. The following subsection describes our approach to generating test cases informally before we present its implementation in Section 4.

3.2 Generating Test Cases

Let us now consider, how we can generate a system of test cases for some coverage criterion. For each test case, we need to find a sequence of parameters with a corresponding expected result. Moreover, we would like the set of test cases to cover all coverable items according to the selected criterion.

A naive way of producing a set of test cases in a functional logic language is to call the function `f` to be tested with a sequence of unbound logic variables `x1,...,xn` as parameters and to compute all possible solutions of `f x1...xn`. Each computation will bind the logic variables to some terms such that the function called with these terms as parameters causes the desired coverage. The result of this computation will be the desired expected result for the test case. Note, that we do not need to integrate a constraint solver like [17] because in a functional-logic language free variables can be bound by the built-in narrowing mechanism. In order to obtain a list of test cases for a function `f`, we could collect the results of the test-case generation with the primitive function `allValues` using encapsulated search [7]:

```
allValues (let x1,...,xn free
               in ((x1,...,xn), f x1 ... xn))
```

Unfortunately, this naive approach will in general fail to produce the desired minimal set of test cases. Typically, it will even generate an infinite number of them. This does not mean that this narrowing-based generation of test cases cannot be used at all. We rather have to make sure that the computation is controlled in such a way that not too many test cases are generated.

In our approach, we record the set of covered items during the computation along with every computed result. Note that this approach is independent of the selected coverage criterion. Given this additional information, we can demand further results until we obtain the desired coverage. Thus, we need to be able to compute the result of encapsulated search lazily. Moreover we have to rely on a fair search strategy, as, e.g., offered by KiCS [9], that ensures that all results are eventually computed.

With a non-fair depth-first search, the overall computation would try to find an infinite amount of solutions for some subexpression before considering an alternative which causes the missing items to be covered. In particular in situations, where the desired coverage cannot be achieved (as explained in Subsubsection 3.1.2), additional means for controlling the computation are required. One possibility is to limit the recursion depth based on an additional parameter which keeps track of it. Alternatively, the computation could be stopped, if the last n generated test cases do not cover any new coverable item (n has to be configured appropriately). The simplest means for controlling the computation is to limit the amount of generated test cases to some fixed n (which has to be configured appropriately). Our system can be combined with any of these alternatives.

In the case of the `insert` function shown above, our tool will compute the following (not yet minimal) set of test cases with GBC (ignoring the built-in function `compare`):

function call	expected result	covered branches
`insert 0 []`	`[0]`	$\langle 1\rangle$
`insert 1 []`	`[1]`	$\langle 1\rangle$
`insert -1 []`	`[-1]`	$\langle 1\rangle$
`insert 0 [1]`	`[0,1]`	$\langle 2\rangle$, $\langle 2.1\rangle$
`insert 0 [0]`	`[0]`	$\langle 2\rangle$, $\langle 2.2\rangle$
`insert 0 [-1]`	`[-1,0]`	$\langle 2\rangle$, $\langle 2.3\rangle$, $\langle 1\rangle$

The program transformation leading to these test cases will be explained in the next section. The reader may wonder, why we need to use the generated test cases at all for testing, since they reflect the actual behavior of the system and one can hence observe an

erroneous behavior by just looking at a generated test case. The reason is that the test cases are needed for regression testing, i.e., in order to check whether a change of the system does not destroy the already working functionality.

We can observe that the approach described so far does not ensure a minimal set of test cases. Here, the first three test cases are redundant, since their sets of covered items are contained in the corresponding sets of the other test cases (in fact they are already subsumed by the last test case). In order to get a minimal set of test cases, we need an additional step which removes redundant test cases. Obviously, this problem is the set covering problem which is known to be NP-complete [13]. Since it is not essential in our context that we really find a minimal solution, we are happy with any heuristic producing a small solution. Sometimes, a larger set of smaller test cases may be preferred over a smaller set of larger test cases because small test cases are usually easier to verify by humans. However, for regression testing the smaller set is always cheaper to check.

Not all Curry implementations support the guessing of numbers in arithmetic operations. Currently only KiCS [9] does it – by implementing numbers as algebraic datatype (cf. Section 5). In order to be able to handle guessing in arithmetic operations in other Curry implementations as well, we employ the system of constraint solvers presented in [17]. During the computation, generated constraints are checked for consistency against other already generated constraints in order to select valid computation paths. After the computation, we solve the generated constraints and instantiate numerical variables according to the computed solution in order to produce a test case. We do not describe the integration of the constraint solver in detail, because it is not an essential part of our approach to generating test cases. Using KiCS, we do not have to integrate a constraint solver at all.

3.3 Custom Input Data

The approach presented so far may produce some test cases which the user does not expect. Some of them are helpful, since they show a possible behavior which the user has not carefully thought about. Others are less helpful, since they use parameters which do not meet preconditions of the corresponding function. For instance, the following predicate `subset` assumes its arguments to be sorted lists of pairwise distinct integers.

```
subset [] _ = True
subset (_:_) [] = False
subset (x:xs) (y:ys) = case compare x y of {
  LT -> False;
  EQ -> subset xs ys;
  GT -> subset (x:xs) ys }
```

Applied to `subset`, our tool will deliver besides a couple of desired test cases the following uninteresting one consisting of the call `subset [0] [0,0]` and expected result `True`. The second argument of the function call is invalid as it contains the value `0` twice. In order to eliminate such test cases, custom *generators* can be supplied by the user and are then used as arguments instead of free variables. These generators should only produce values that meet the specification. In our example, we would replace the initial call `let xs,ys free in subset xs ys` by

```
let xs = numSet; ys = numSet in subset xs ys
```

where `numSet` computes valid list representations of sets of integers:

```
numSet = []
numSet = insert x numSet where x free
```

Using a call to `numSet` instead of a free variable, effectively prevents the generation of the invalid test case.

4. Program Transformation

In this section, we discuss the implementation of our approach to test-case generation. We first motivate the ideas behind our program transformation in Subsection 4.1 and then present an algorithm that transforms FlatCurry programs in Subsection 4.2. We consider higher-order functions in Subsection 4.3 and show how to extend the algorithm to different coverage criteria in Subsection 4.4.

As mentioned in Subsection 3.2, we transform a labelled program such that the computation collects information w.r.t. a coverage criterion. Hence, the transformed program needs to record which of its parts have been reached during the execution. To determine which parts of the program really have been executed is a subtle task due to laziness. If the program transformation changes the evaluation order of the original program, the collected information might not correspond to the original execution. Even worse, a computation of the transformed program may not even terminate, if the transformation destroys laziness.

4.1 Tracking Computations

In this subsection we will develop a program transformation to collect information about lazy computations. We start with a simple monadic approach, identify its problems and refine it to preserve laziness. We will observe that a transformation into monadic code has a strong influence on the evaluation order and is therefore not suited for our purposes. Nevertheless, we present the approach to point out the importance of preserving laziness.

4.1.1 A Too Simple Approach

We aim at transforming a program such that it computes a list of items covered by the computation along with the original result. We can even generalize this aim and compute information that can be mapped to items that correspond to different coverage criteria instead of computing specific items for a single coverage criterion. For now, we will restrict us to collect labels that represent branches taken by a computation. As labels are represented as lists of numbers, we define the type `Info` of collected information as `[Int]`. The result of a computation should be augmented with such information, so we define a type

```
data I a = I a [Info]
```

that represents an augmented original result of some type `a`. We can easily define a monad that hides the collection of information:

```
return :: a -> I a
return x = I x []

(>>=) :: I a -> (a -> I b) -> I b
I a ia >>= f = let I b ib = f a in I b (ia++ib)

(>>) :: I a -> I b -> I b
ia >> ib = ia >>= \_ -> ib

ap :: I (a -> b) -> I a -> I b
ig `ap` ix
  = ig >>= \g -> ix >>= \x -> return (g x)
```

Using these monad operations we could transform each function of a program into a monadic version that collects the branches that were taken during its execution. For this purpose, we define a monad operation `collect` that adds a value of type `Info` to the collected information:

```
collect :: Info -> I ()
collect l = I () [l]
```

To demonstrate the shortcomings of this approach, we will present a program along with its transformed version and observe for some

computations whether the collected information reflects the original program behavior. The transformation performs every subcomputation that is given as argument to another function inside the monad to collect the covered items and supplies the computed result in the original argument position. Every label of a labeled expression is transformed into a call to `collect`. The result of the `collect`-operation is always discarded. We use the operation only to collect the covered items in the background. Consider the following definitions:

```
test :: Bool
test = isIn 1 (interval 1 3)

interval :: Int -> Int -> NumSet
interval n m = case n == m of {
  True -> [n];
  False -> n : interval (n+1) m }

isIn :: Int -> NumSet -> Bool
isIn n ns = case ns of {
  [] -> False;
  (m:ms) -> case n == m of {
    True -> True;
    False -> isIn n ms }}
```

In the monadic version of this program additional calls to `collect` are used to record the executed branches:

```
test :: I Bool
test = interval 1 3 >>= isIn 1

interval :: Int -> Int -> I NumSet
interval n m = case n == m of {
  True -> collect [1,1] >> return [n];
  False -> collect [1,2] >>
           return (n:) `ap` interval (n+1) m }

isIn :: Int -> NumSet -> I Bool
isIn n ns = case ns of {
  [] -> collect [2,1] >> return False;
  (m:ms) -> collect [2,2] >> case n == m of {
    True -> collect [2,2,1] >> return True;
    False -> collect [2,2,2] >> isIn n ms }}
```

The result of executing the transformed version of `test` is:

```
I True [[1,2],[1,2],[1,1],[2,2],[2,2,1]]
```

The computed result is `True` since 1 is an element of the set $\{1, 2, 3\}$ and every but the first and the last branch of `isIn` has been collected by the monadic version of the program. In particular, all branches of the function `interval` have been collected although the first is never executed in a lazy computation of the original program. If we test whether 1 is an element of the infinite set of positive numbers[2] by evaluating `interval 1 0 >>= isIn 1`, the computation loops forever. The evaluation of `isIn 1 (interval 1 0)` in the original program would terminate due to lazy evaluation.

The presented monad is not suited to compute information covered by a lazy computation. Originally uncovered items are collected and programs that rely on lazy evaluation may not terminate in the transformed version. In the remainder of this section we present a more elaborate program transformation without the presented drawbacks.

[2] computed by (mis-)use of the function `interval` where the first argument is greater than the second

4.1.2 Preserving Laziness

The key to preserving laziness is to associate a set of covered items with every constructor instead of computing one set for the result of the computation. If each constructor has an associated set of items, then exactly this set can be discarded, if the corresponding constructor is not demanded by the evaluation. With this approach, only items attached to demanded constructors are computed which is essential to obtain coverage information for a lazy computation. One possibility to store information at every constructor is to extend every constructor of the program with an additional argument. However, with this approach it would be difficult to obtain the original value from a value augmented with coverage information, which is useful for two reasons: 1. the generated test case should be presented to the user without attached coverage information and 2. external functions cannot be called with augmented values. In the monadic approach, we could just remove the attached list from the computed result. This is an advantage we do not want to dismiss in our adapted approach.

Therefore, instead of a list of items we store a tree of them along with every value. At the root of this tree, we store the items that are covered to compute the head-normal form of the corresponding value. A subtree of items is added to the root for every argument of the head-constructor. We redefine a type `I a` to collect (unranked) trees of information next to an arbitrary value.

```
data I a = I a (Tree [Info])
data Tree a = Tree a [Tree a]
```

We define an operation `wrap` to wrap values with an empty tree of items and redefine the operations `collect` and `ap`. The function `collect` will be used to collect covered items and `ap` to apply constructors to wrapped arguments:

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

collect :: [Info] -> I a -> I a
collect xs (I a (Tree ys ts))
  = I a (Tree (xs++ys) ts)

ap :: I (a -> b) -> I a -> I b
ic `ap` ix = I (c x) (Tree xs (ts++[t]))
 where
  I c (Tree xs ts) = ic
  I x t = ix
```

Note that the type of `collect` has changed in two respects: the first argument is of type `[Info]` instead of `Info` and we add a second argument of type `I a` whose root information is augmented with the information given as first argument. The function `ap` is used to apply a constructor to an additional argument. Therefore, we add the tree of items from the argument to the child trees of the constructor. We use *lazy pattern matching* for the definition of `ap`: the pattern matching on the arguments is performed in a local declaration and therefore not performed until the matched components are demanded. For example, the argument `ix` is not evaluated until `x` or `t` is demanded by the computation. To enhance readability, we use local where-declarations that are not allowed in FlatCurry but are legal Curry syntax and can be easily eliminated. We can use the functions `wrap` and `ap` to construct complex data structures from simpler ones. For example, the list `[1,2]` - which is equivalent to `1:2:[]` or `((:) 1 ((:) 2 []))` - is represented by the expression

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

This expression evaluates to

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

The term structure of the value `[1,2]` is reflected by the structure of the attached tree.

We now describe our program transformation using the example introduced in Subsubsection 4.1.1. In contrast to the monadic approach, not only the result types but also the argument types of the functions are modified. The definition of `test` is transformed as follows:

```
test :: I Bool
test = isIn (wrap 1) (interval (wrap 1) (wrap 3))
```

The result of the call to `interval` can be directly passed to the function `isIn`, which takes a value of type `I NumSet` as second argument. We first discuss the transformation of `interval`. The call to `(==)` in the original definition is replaced by a call to the (omitted) function `eq :: I Int -> I Int -> I Bool`. Analogously, the call to `(+)` is replaced by a call to `add`.

```
1 interval :: I Int -> I Int -> I NumSet
2 interval n m =
3   let I b (Tree xs ts) = eq n m
4    in collect xs
5     (case b of {
6       True -> collect [[1,1]]
7               (wrap (:) `ap` n `ap` wrap []);
8       False -> collect [[1,2]]
9                (wrap (:)
10                `ap` n
11                `ap` interval (add n (wrap 1)) m)
12    })
```

To match the result of the equality test `n == m`, we select it as `b` from the result of `eq n m` in line 3. As the pattern matching demands the evaluation of the head-normal form of `b`, the items that were collected to compute this head-normal form are covered to compute the head-normal form of the result of the current computation. Therefore, we collect the items `xs` at the root of the tree corresponding to `b` in line 4. The list of child trees `ts` is not demanded. In fact, it will always be empty, because neither `True` nor `False` have arguments. We will see how to proceed with child trees later, when we transform the function `isIn`. The two branches of the function `interval` are augmented with calls to `collect` in lines 6 and 8. The functions `wrap` and `ap` are employed to wrap and apply the constructors. Functions are applied directly without a special combinator.

Consider the transformation of the function `isIn`:

```
1 isIn :: I Int -> I NumSet -> I Bool
2 isIn n ns
3   = let I ns' (Tree xs ts) = ns
4      in collect xs (case ns' of {
5       [] -> collect [[2,1]] (wrap False);
6       (m':ms') -> let m  = I m' (ts!!0)
7                       ms = I ms' (ts!!1) in
8         collect [[2,2]]
9          (let I b (Tree ys us) = eq n m
10           in collect ys (case b of {
11            True -> collect [[2,2,1]]
12                    (wrap True);
13            False -> collect [[2,2,2]]
14                     (isIn n ms)})})})
```

The lines 3 and 4 and the lines 9 and 10 are similar to the lines 3,4 and 5 in the transformation of `interval`: a value is selected to be matched in a case expression and the items at the root of the matched value are collected for the current computation. Also, every branch is augmented with an adequate call to `collect`. In line 6, the variables `m'` and `ms'` are selected as arguments of the constructor `(:)`. They are employed to build the wrapped values `m` and `ms` which are later used in this branch. These values are wrapped with the corresponding child trees that were attached to `ns`. We use the function `(!!) :: [a] -> Int -> a` to select the correct tree using its position in the list `ts` of child trees. Note that if the branch in line 11 is taken, then the items attached to `ms` are not collected for the result of the call to `isIn`. Therefore, they can be discarded which is essential to prevent the collection of uncovered items and avoid infinite loops. The tree of items that reflects the term structure of every value allows us to collect exactly those items that are covered by a lazy execution of the original program.

4.2 Formalization

In this section, we formalize our program transformation as mapping on labeled FlatCurry expressions. Each function of a program is transformed into another function by applying a mapping τ to its body. Each labeled expression is replaced by a call to `collect` applied to information computed from the label and the recursively transformed expression:

$$\tau(\langle l\rangle\ e) = \texttt{collect}\ [info(l)]\ \tau(e)$$

The term $info(l)$[3] denotes collected information that is computed from a label. In the previous example, we just collected the labels themselves.

We have already seen that the most interesting part of the transformation is concerned with constructor applications and case expressions. In fact, everything else is left unchanged:

$$\begin{array}{rll} \tau(x) = x & & (x \text{ variable}) \\ \tau(f\ e_1\ \ldots\ e_n) = f\ \tau(e_1)\ \ldots\ \tau(e_n) & & (f \text{ function}) \\ \tau(let\ \overline{x_i = e_i}\ in\ e) = let\ \overline{x_i = \tau(e_i)}\ in\ \tau(e) & & \\ \tau(e_1\ or\ e_2) = \tau(e_1)\ or\ \tau(e_2) & & \end{array}$$

In case of an *or* expression, both arguments will have an attached label. Therefore, their transformation will introduce corresponding calls to the function `collect`.

Since constructors do not take wrapped values as arguments, they are applied using the `wrap` and `ap` combinators:

$$\tau(c\ e_1\ \ldots\ e_n) = \texttt{ap}\ (\ldots(\texttt{ap}\ (\texttt{wrap}\ c)\ \tau(e_1))\ldots)\ \tau(e_n)$$

Note that a constructor c without arguments is just represented as $(\texttt{wrap}\ c)$.

To complete the definition of τ we now show how to transform case expressions. Since case expressions demand evaluation, we have to carefully associate collected labels with the computed result. The transformation of case expressions is made up of different parts. We need to

1. select the value to be matched from its wrapped representation,
2. collect the items that were covered to compute its head-normal form, and
3. wrap the arguments of the matched constructor with the corresponding item trees taken from the matched value.

The last rule of our transformation performs all these tasks. We do not strictly adhere to the syntax defined in Subsection 2.3 and use pattern matching in a local declaration to enhance readability. This can be easily eliminated – in fact, we could as well use additional *case*-expressions to decompose the wrapped value. However, this would bloat further the already quite complex rule.

[3] In Subsection 4.4 we will collect information that is not only computed from the label.

$$
\begin{array}{l}
\tau(case\ e\ of\ \{\ldots(c_i\ x_{i_1}\ \ldots\ x_{i_k}) \rightarrow e_i;\ldots\}) = \\
\quad let\ \texttt{I}\ e'\ \texttt{(Tree}\ xs\ ts\texttt{)} = e\ in \\
\quad\ \texttt{collect}\ xs\ (case\ e'\ of\{ \\
\quad\quad \vdots \\
\quad\quad c_i\ x'_{i_1}\ \ldots\ x'_{i_k} \rightarrow \\
\quad\quad\quad let\ x_{i_1} = \texttt{I}\ x'_{i_1}\ (ts\texttt{!!}0) \\
\quad\quad\quad\quad \vdots \\
\quad\quad\quad\quad x_{i_k} = \texttt{I}\ x'_{i_k}\ (ts\texttt{!!}(k-1)) \\
\quad\quad\quad in\ \tau(e_i); \\
\quad\quad \vdots \\
\quad\ \})
\end{array}
$$

In this definition, $e', xs, ts, x'_{i_1}, \ldots, x'_{i_k}$ are fresh variable names. The first let binding selects the value e' to be matched by the case expression, the items xs covered to compute the head normal form of e' and the item trees ts corresponding to the arguments of the head-constructor of e'. The call to `collect` collects the items xs because the head-normal form of e' is demanded by the following case expression. In each branch of this case expression the original argument variables are defined in terms of fresh ones together with the corresponding item trees selected from ts. Finally the result of the case branch is transformed recursively. As each branch has an attached label, this will result in another call to the function `collect`.

The formal definition of our program transformation can be directly transferred into an implementation. In fact, we have extended our approach to support higher-order functions and primitive operations and implemented this extension. However, its formalization is not within the scope of this description. We informally explain the extension of our approach to higher-order functions in the next subsection.

4.3 Higher-Order Functions

In FlatCurry there are no lambda abstractions. Every lambda abstraction in a Curry program is replaced by a newly introduced function by so called *lambda lifting*. Furthermore, applications of variables are eliminated using a primitive function

```
apply :: (a -> b) -> a -> b
```

For example, the following definition of function composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

is transformed into[4]

```
f . g = aux f g

aux :: (b -> c) -> (a -> b) -> a -> c
aux f g x = apply f (apply g x)
```

The primitive function `apply` evaluates its first argument to a *partial application* and extends this partial application with the additional argument. A partial application can be seen as constructor term. For each function or constructor of arity n there are n constructor symbols of arity $0, \ldots, n-1$ to construct partial applications. The arity of a function (constructor) is the number of argument variables (type arguments) in its definition, respectively. For example, the arity of the function `(.)` is two and the arity of the function `aux` is three, although both functions have the same type.

In the approach presented in the previous subsections, the argument- and result-types of functions are wrapped with the type constructor `I`. Functional argument types have to be transformed in the same way to allow transformed functions to be passed to other functions. To illustrate this, we transform the following program:

```
inclist :: [Int] -> [Int]
inclist = map (1+)

map :: (a -> b) -> [a] -> [b]
map f l = case l of {
            [] -> [];
            x:xs -> apply f x : map f xs }
```

The function `inclist` is defined as partial application of the function `map` to a partial application of the function `(+)`. In the definition of `map` the application of `f` to `x` is expressed using `apply`.

The function `(+) :: Int -> Int -> Int` is transformed into a function of type `I Int -> I Int -> I Int`. The transformed version of `map` has a type that is slightly surprising: `I (I a -> I b) -> I [a] -> I [b]`. The functional type of the first argument of `map` is wrapped by `I` which enables to collect items that are covered to compute this argument. The argument and result type of the first argument of `map` are also wrapped with `I` as functions passed to `map` have such types.

If we take a closer look at the involved types, we see that in the transformed program the partial application `(wrap 1 +)` would have the type `I Int -> I Int` which does not match the type `I (I a -> I b)` of the first argument of `map`. We need to wrap partial applications using `wrap`:

```
wrap (wrap 1 +) :: I (I Int -> I Int)
```

Now we can apply the transformed version of `map` to this wrapped partial application. The result is a partial application of `map` which therefore has to be wrapped with `wrap` itself:

```
inclist :: I (I [Int] -> I [Int])
inclist = wrap (map (wrap (wrap 1 +)))
```

We use the function `app` to apply such wrapped functions:

```
app :: I (I a -> I b) -> I a -> I b
app ig ix = I y (Tree (xs++ys) ts)
 where
  I g (Tree xs _) = ig
  I y (Tree ys ts) = g ix
```

The items covered to compute the function `g` are added to the items covered to compute the head-normal-form of the result `y` of the application.

The transformation of the definition of `map` is as follows[5]:

```
map :: I (I a -> I b) -> I [a] -> I [b]
map f l
  = let I l' (Tree _ ts) = l
    in case l' of {
         [] -> wrap [];
         x':xs' -> let  x = I x' (ts!!0)
                        xs = I xs' (ts!!1)
                   in wrap (:)  `ap` app f x
                                `ap` map f xs }
```

In general, all calls to `apply` are replaced with calls to `app` and all partial applications are wrapped using `wrap`. In the previous example, we have only seen partial applications with one missing argument. How do we have to wrap partial applications with more of them? For example, the partial application `(+)` without any arguments has two missing arguments and needs to be transformed into something of type `I (I Int -> I (I Int -> I Int))`. A

[4] Of course, a Curry compiler does not use `aux` as name for the auxiliary function as we did but takes care of name conflicts.

[5] We omit all calls to `collect` for brevity.

simple application of `wrap` is not sufficient here – the resulting type is `I (I Int -> I Int -> I Int)`. We need to stack calls to `wrap` using function composition `(.)` to get a wrapper of the correct type[6]:

```
wrap :: (I a -> I b) -> I (I a -> I b)
wrap.(wrap.)
  :: (I a -> I b -> I c)
  -> I (I a -> I (I b -> I c))
wrap.(wrap.).((wrap.).)
  :: (I a -> I b -> I c -> I d)
  -> I (I a -> I (I b -> I (I c -> I d)))
...
```

We can use this scheme to wrap partial applications of functions with any number of missing arguments. Constructors initially do not take wrapped arguments like functions do. The following family of functions corrects this for constructors of arbitrary arity greater than zero:

```
ap :: I (a -> b) -> I a -> I b
(ap.).ap :: I (a -> b -> c) -> I a -> I b -> I c
((ap.).).(ap.).ap
  :: I (a -> b -> c -> d)
  -> I a -> I b -> I c -> I d
...
```

If we first apply such a combination of `ap` and `(.)` and then a similar combination of `wrap` and `(.)`, we can wrap partial applications of constructors of any arity.

We could only briefly explain how we transform higher-order-functions. Space restrictions prevent us from formally presenting the approach. For the same reason, we omit the discussion of functional arguments of data constructors.

4.4 Different Coverage Criteria

In Section 3 we have introduced GBC to cover all branches of reachable functions and FC to cover the branches of a considered function for every reachable call to it. The transformation presented so far collects the labels that are attached to the branches in a FlatCurry expression. This information is sufficient to implement GBC. The set of covered labels is enough to decide which of the reachable branches have been executed.

However, our approach is not restricted to collecting branch labels. As an example, we show how to extend the collected information in order to support the implementation of FC. It turns out that this extension suffices to implement many variations of the presented coverage criteria. We only describe this extension informally by examples because it is far less complicated than the transformation presented so far.

GBC collects branch labels without keeping track of the function call which caused the execution of the branch. In order to check FC, we need to associate a call position to each collected branch. We can represent a call position of a function `g` by the name of the calling function (e.g., `f`) together with a number that distinguishes multiple calls to `g` in `f`. The new type of collected information stores a call position together with a label:

```
type CallPos = (String,Int)

type Info = (CallPos,[Int])
```

We can add a new parameter of type `CallPos` to every defined function and use this parameter to collect values of the adjusted `Info` type. We only show the transformation of the function `interval` to clarify the idea. The other functions can be transformed similarly.

[6] Note that the real types are more general then the presented types.

```
1  interval :: CallPos -> I Int -> I Int -> I NumSet
2  interval cp n m =
3    let I b (Tree xs ts) = eq ("interval",1) n m
4     in collect xs
5      (case b of {
6        True -> collect [(cp,[1,1])]
7                (wrap (:) `ap` n `ap` wrap []);
8        False -> collect [(cp,[1,2])]
9                (wrap (:)
10                 `ap` n
11                 `ap` interval ("interval",1)
12                       (add  ("interval",1)
13                          n  (wrap 1))
14                        m)
15     })
```

Every function called by `interval` is extended with a call position that consists of the name `"interval"` and a number enumerating different calls of the same function (here lines 3, 11 and 12). As no function is called twice, this number is always 1. The calls to `collect` in the branches of the case expression are modified such that the parameter `cp` is attached to every collected label (lines 6 and 8).

Using the extended version of the program transformation, the transformed program collects items that specify which branches have been executed due to which calls of the corresponding function. This information can be used to check GBC and FC. However, also modified versions of these coverage criteria could be checked. For example, we could check the coverage of any subset of all reachable branches. Or we could check for an arbitrary set of functions whether every call to one of these functions is covered separately by the generated test cases.

5. Practical Experience

We have implemented a prototype of our approach to evaluate its applicability and usefulness. Our benchmarks were run on an AMD Athlon™ XP 3000+ with 2 GHz, 512 KB cache and 3 GB main memory. We used KiCS [9] for our experiments, because it is the only Curry system that supports fair encapsulation on demand and guessing in arithmetic operations. The latter is due to the fact that values of type `Int` are represented by an algebraic datatype:

```
data Int = Neg Nat | Zero | Pos Nat
```

The datatype `Nat` defines positive integers in binary notation with the least significant bit first. The most significant bit is always 1 to avoid ambiguities and therefore denoted `IHi`:

```
data Nat = IHi | O Nat | I Nat
```

In this representation, e.g., the number 42 is represented as

```
Pos (O (I (O (I (O IHi)))))
```

and -4 as

```
Neg (O (O IHi))
```

Defining the usual arithmetic operations on the `Int` and `Nat` datatypes is error prone because the binary notation is not very appealing to a human eye. On the other hand, a correct implementation is crucial here as it lies at the heart of every Curry program that involves arithmetic operations. As these operations are not yet formally verified, we wanted to gain confidence in their correctness by generating test cases for them. Binary arithmetic operations serve well to evaluate the usefulness of the presented coverage criteria because they can be implemented as small but complex functions. As mentioned in Section 1.1 this is a typical situation to apply Glass-Box Testing.

	GBC		FC	
	# test cases	seconds	# test cases	seconds
add	6 (15)	0.04+0.01	20 (>219)	0.71+0.24
sub	7 (20)	0.09+0.03	17 (137)	0.62+0.13
mul	2 (>141)	0.75+0.17	5 (17)	0.03+0.01
cmpNat	8 (38)	0.08+0.05	33 (294)	1.30+1.10
(+)	18 (184)	0.83+0.51	11 (42)	0.06+0.02
(-)	15 (>280)	1.51+0.94	3 (3)	0.01+0.01
(*)	7 (>171)	0.39+0.14	7 (1)	0.01+0.01
cmpInt	15 (178)	0.41+0.18	9 (15)	0.02+0.01

Figure 3. Number of test cases generated and required runtimes for both considered coverage criteria, GBC and FC, for selected operations of the arithmetic library.

5.1 Applicability

We will introduce some of the arithmetic operations in order of increasing complexity. First we will introduce simple operations on `Nat` values and later proceed with `Int` operations which are built upon the simpler ones. We will always generate test cases using the different coverage criteria introduced in Subsection 3.1 to evaluate these criteria. We do not yet aim at finding bugs but want to measure how expensive our approach to generating test cases is for the presented functions.

The addition on positive integers is defined as follows:

```
add :: Nat -> Nat -> Nat
add IHi   y      = succ y
add (O x) IHi    = I x
add (O x) (O y) = O (add x y)
add (O x) (I y) = I (add x y)
add (I x) IHi    = O (succ x)
add (I x) (O y) = I (add x y)
add (I x) (I y) = O (add (succ x) y)

succ :: Nat -> Nat
succ IHi    = O IHi
succ (O bs) = I bs
succ (I bs) = O (succ bs)
```

Six test cases suffice to cover all branches in the functions `add` and `succ`:

```
add (I IHi) IHi          = O (O IHi)  -- 3+1 = 4
add IHi     (I (O IHi))  = O (I IHi)  -- 1+5 = 6
add (I IHi) (O IHi)      = I (O IHi)  -- 3+2 = 5
add (O IHi) (I IHi)      = I (O IHi)  -- 2+3 = 5
add (O IHi) (O IHi)      = O (O IHi)  -- 2+2 = 4
add (I IHi) (I IHi)      = O (I IHi)  -- 3+3 = 6
```

The test cases were selected by our tool out of a set of 15 generated test cases that were computed in about 40 milliseconds. Nine redundant test cases were eliminated which took even less time.

Note that it is not possible to cover all branches of `add` in every call to it. The recursive call to `add` in the last rule never executes the first rule since the result of `succ x` in the first argument can never be `IHi`. Trying to generate a set of covering test cases according to FC (cf. Subsubsection 3.1.2), reveals exactly this information. Our tool generated 219 test cases in approximately one second before it gave up to satisfy the coverage criterion because 100 test cases were generated without additional coverage. After eliminating redundant test cases, which took less than half a second, a set of 20 test cases remained, which cover all reachable branches. With FC, the function `add` was tested more thoroughly than with GBC. 20 non-equivalent test-cases remain, compared to 6 using GBC. The function `add` only depends on one other function and it contains four recursive calls. Therefore, it is not surprising that it is more difficult to satisfy FC in this example.

Let us now consider the function `(+)` on `Int` values which is defined in terms of `add`, `sub` and `cmpNat`. We will not discuss the subtraction function `sub :: Nat -> Nat -> Nat` in detail. The results of generating test cases for `sub` are comparable to the results obtained with `add` although it is possible to satisfy FC. We will discuss the comparison function

```
cmpNat :: Nat -> Nat -> Ordering
```

later and proceed with the definition of `(+)`:

```
(+) :: Int -> Int -> Int
Zero  + x     = x
Pos x + Zero  = Pos x
Pos x + Pos y = Pos (add x y)
Pos x + Neg y = case cmpNat x y of
  LT -> Neg (sub y x)
  EQ -> Zero
  GT -> Pos (sub x y)
Neg x + Zero  = Neg x
Neg x + Neg y = Neg (add x y)
Neg x + Pos y = case cmpNat x y of
  LT -> Pos (sub y x)
  EQ -> Zero
  GT -> Neg (sub x y)
```

Here, the situation is different compared to the first example: the function `(+)` does not contain any recursive call but it depends on a number of other functions some of which depend on other functions themselves. Consequently, 18 test cases are required to satisfy GBC, while 11 suffice to satisfy FC.

The function `mul` on positive integers is an example of a recursive function that also depends on other functions:

```
mul :: Nat -> Nat -> Nat
mul IHi   y = y
mul (O x) y = O (mul x y)
mul (I x) y = add y (O (mul x y))
```

Not all branches of `add` can be covered by a call to `mul`: the second argument of `add` is never `IHi` (even in recursive calls of `add`), because in the call to `add` in `mul` the second argument is always bigger than the first. As GBC cannot be obtained, 141 test cases are generated where the last 100 do not cover additional branches. Our system selects two test cases that suffice to cover all reachable branches:

```
mul (I (I IHi)) (I (I IHi))  = I(O(O(O(I IHi))))
mul (O IHi)     IHi          = O IHi
```

If the coverage criterion cannot be satisfied, the stopping criterion determines which test cases will be (generated and) selected. For example, one single test case suffices to cover all branches reachable from `mul`:

```
mul (O (I (I IHi))) (I (I IHi))  -- 14*7
  = O (I (O (O (O (I IHi)))))    -- = 98
```

This test case is only generated, when our system generates larger and larger test cases trying to satisfy the coverage criterion. Here, the larger set of smaller test cases is preferable if it should be verified by a human.

Covering only the branches of `mul` in every single call is easily possible. Only 17 test cases need to be generated and 5 of them can be selected to satisfy FC.

The results of all our experiments are depicted in Figure 3. The two coverage criteria are compared w.r.t. the number of generated test cases and the time necessary to compute them. We show the number after eliminating redundant test cases along with the total number of generated test cases after which the coverage criterion was satisfied in parentheses. A greater sign ($>$) indicates that the coverage criterion could not be satisfied due to the actual parameters of the called functions and that the computation has been stopped, since a previously set limit[7] of (here) 100 test cases without covering new branches was reached. If this limit is big enough, these test cases will cover the reachable branches. The presented times are split in order to distinguish the time needed to compute all test cases (first number) from the time used for redundancy elimination (second number).

We can see that it is more difficult to satisfy GBC for functions that depend on other complex functions. FC could be satisfied easier. Here, the number of test cases generated to cover all calls to the considered function generally corresponds to the complexity of its definition, regardless whether it depends on other complex functions or not. FC leads to a more thorough testing of the branches of a function than GBC, provided that all other reachable functions have been tested separately.

Because we chose small functions that serve well to compare the presented coverage criteria, we can see little about the applicability of our approach to larger applications. To check whether our tool is useful in practice, we need to apply it to code that combines library functions from different modules. Therefore we generated test cases for the function that we use for redundancy elimination. We implemented a heuristic for the set-covering problem that uses standard libraries for list processing and balanced search trees. Our tool needs almost 30 seconds to generate 227 test cases for GBC and about 10 seconds to select 10 non-equivalent test cases. We also generated test cases for FC but aimed at covering separately all reachable functions in the same module instead of only the tested function. Here, 192 test cases were generated in 15 seconds and two of them were selected in almost 2 seconds. The measured times are significantly larger than those for the arithmetic functions. Nevertheless, they are acceptable and the number of generated test cases is manageable. We suspect that we introduce significant overhead by the way we collect coverage information that is not yet tuned for performance. Without monitoring coverage, thousands of test cases for this function can be generated within a few seconds. Therefore we are optimistic to be able to achieve better performance also for large applications in the future.

5.2 Usefulness

Until now, we have only compared, how expensive it is to satisfy the different coverage criteria. In this subsection, we will show, how effectively the criteria can expose errors.

We have introduced bugs into some of the arithmetic functions and checked whether the test cases generated w.r.t the different coverage criteria exposed them. In order to obtain more reliable results, we have not only introduced the same bugs at different positions but have chosen from three different kinds of bugs:

1. We have replaced a constructor in the right-hand side of a rule by another constructor of the same type. For example, the term `O x` could be replaced by `I x` to introduce a bug of this kind.
2. We have replaced variables in the right-hand side of a rule by another variable of the same type. For example, the term `add x y` could be replaced by `add x x` to introduce a bug of this kind.
3. We have swapped the arguments in applications of operations that are not commutative. For example, the term `cmpNat x y` could be replaced by `cmpNat y x` to introduce a bug of this kind.

Some bugs have been introduced only into a function that is called by the tested function. From a total number of 20 introduced bugs, 15 were detected by GBC and 19 by FC. The bug not exposed by FC has been introduced in a function called by the tested function which therefore did not have to be covered according to FC. FC exposes the bug, if we generate test cases for the buggy function. However, this does not mean that FC detects all bugs in general. We have already seen an example for a bug that is not detected by FC in Subsubsection 3.1.2. GBC failed to expose four introduced bugs of kind (2) and one introduced bug of kind (3). In order to explain the difference between GBC and FC and to understand why FC tests a function more thoroughly, we will discuss the test-case generation for one example in more detail.

According to GBC, the covered branches are not distinguished by the call that caused its execution. If one recursive call executes the first rule of the tested function, all other recursive calls do not need to execute this rule in order to satisfy GBC. According to FC, every recursive call needs to execute every branch of the tested function. Consider the following definition of the comparison function `cmpNat`:

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

The test cases that satisfy GBC do not expose an error:

```
cmpNat (O (I IHi)) (O IHi)      = GT  -- 6 > 2
cmpNat (O IHi)     IHi          = GT  -- 2 > 1
cmpNat IHi         (I IHi)      = LT  -- 1 < 3
cmpNat (I IHi)     (I IHi)      = EQ  -- 3 == 3
cmpNat (I IHi)     (O IHi)      = GT  -- 3 > 2
cmpNat (O IHi)     (I IHi)      = LT  -- 2 < 3
cmpNat (I IHi)     (O (O IHi))  = LT  -- 3 < 4
cmpNat (O (I IHi)) (I IHi)      = GT  -- 6 > 3
```

But is the definition correct? The function `cmpNat` is quite complex: there are 9 rules with 4 recursive calls. Covering all these calls individually requires 33 test cases. Among them are eight that expose an error:

```
cmpNat (I (O IHi)) (I IHi)          = LT  -- 5<3
cmpNat (I IHi)     (I (I IHi))      = GT  -- 3>7
cmpNat (I IHi)     (I (O IHi))      = GT  -- 3>5
cmpNat (I (I IHi)) (I (O IHi))      = LT  -- 7<5
cmpNat (I (O IHi)) (I (I IHi))      = GT  -- 5>7
cmpNat (I (I IHi)) (I (O (O IHi))) = GT  -- 7>9
cmpNat (I (O (O IHi))) (I (I IHi)) = LT  -- 9<7
cmpNat (O (I (I IHi))) (I (I IHi)) = LT  -- 14<7
```

[7] specified by the user

We can observe that all of them use the last rule of `cmpNat` which is indeed faulty: we have flipped the arguments in the recursive call. GBC did not expose the error, since the last rule was only executed with equal arguments. In fact, it was only executed once with the arguments `I IHi` and `I IHi`. The recursive call in the last rule was not covered completely. It was only executed with the arguments `IHi` and `IHi`. Swapping these two arguments obviously has no effect.

6. Conclusions and Future Work

We have shown how glass-box testing based on systematic coverage of the code can be adapted from the imperative world to a functional logic programming language.

We have developed two coverage criteria for the functional (logic) programming paradigm and presented a tool which generates a system of test cases automatically according to a selected coverage criterion. This tool employs the narrowing-based execution mechanism of Curry in order to generate test-cases. The computation is controlled by the set of items to be covered and redundant test cases are eliminated by a heuristic for the set covering problem.

The implementation of our tool is based on a program transformation which adds labels and additional parameters to the functions to be tested in order to compute the coverable items along with the result. Special care had to be taken to handle laziness properly and we have shown that it is not necessary to introduce impure features in order to compute the coverage information.

Practical experiments show that our approach is applicable to realistic algorithms and useful to expose bugs in their definition. Complete sets of test cases could be generated within a few seconds. Even for small but complex applications as the considered arithmetic library, it is very unlikely that a human could sucessfully generate such complete sets of test cases for a given coverage by hand, taking into account that these sets consist of up to 33 elements. This shows the value of our tool for the development of software of high quality. We have demonstrated that *Function Coverage* exposes errors that can remain undetected in test cases that satisfy *Global Branch Coverage*. On the other hand, it usually does not expose errors in reachable functions, so these need to be tested separately.

As future work, we plan to investigate the notion of data-flow coverage in the context of declarative programming. Section 5 indicates that it is sometimes hard to tell whether the generated test cases for a function correspond to its intended meaning. Therefore, we plan to integrate specifications that are employed to automatically verify the generated test cases.

Acknowledgement

This work was partially supported by the German Research Council (DFG) grant Ha 2457/5-2. We would also like to thank Christoph Lembeck and Roger Müller for their support of this work and several fruitful discussions about the approach. Bernd Brassel and Frank Huch made valuable suggestions concerning the coverage criteria and a draft version of this paper.

References

[1] E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, LNCS. Springer-Verlag, January 2007. To appear.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[3] S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.

[4] ART. Hat – the Haskell tracer (version 2.04). Available at URL `http://haskell.org/hat/`, 2005.

[5] B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing functional logic computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 193–208. Springer LNCS 3057, 2004.

[6] B. Braßel, S. Fischer, and F. Huch. A program transformation for tracing functional logic computations. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, LNCS. Springer, 2006. To appear.

[7] B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

[8] B. Braßel, M. Hanus, F. Huch, and G. Vidal. A semantics for tracing declarative multi-paradigm programs. In *Proc. of the 6th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'04)*, pages 179–190. ACM Press, 2004.

[9] B. Braßel and F. Huch. Translating Curry to Haskell. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 60–65. ACM Press, 2005.

[10] R. Caballero and M. Rodríguez-Artalejo. Ddt: a declarative debugging tool for functional-logic languages. In *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, pages 70–84. Springer LNCS 2998, 2004.

[11] O. Chitil, C. Runciman, and M. Wallace. Freja, hat and hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pages 176–193. Springer LNCS 2011, 2001.

[12] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, September 2000.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.

[14] M. Hanus et al. Curry: An integrated functional logic language (version 0.8.2). Available at URL `http://www.informatik.uni-kiel.de/~curry`, 2006.

[15] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[16] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña, editor, *The 14th International workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Madrid, Spain, Springer, September 2002.

[17] C. Lembeck, R. Caballero, R. Müller, and H. Kuchen. Constraint solving for generating glass-box test cases. In *Proceedings of International Workshop on Functional and (Constraint) Logic Programming (WFLP)*, pages 19–32, 2004.

[18] R. Müller, C. Lembeck, and H. Kuchen. A symbolic Java virtual machine for test-case generation. In *Proceedings IASTED*, 2004.

[19] N. Mweze and W. Vanhoof. Automatic generation of test inputs for Mercury programs. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, LNCS. Springer, 2006. To appear.

[20] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[21] R. S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw-Hill, Inc., 1992.