EasyCheck — Test Data for Free*

Jan Christiansen and Sebastian Fischer

Department of Computing Science, University of Kiel, Germany
{jac,sebf}@informatik.uni-kiel.de

Abstract. We present a lightweight, automated tool for specification-based testing of declarative programs written in the functional logic programming language Curry and emphasize the usefulness of logic features in its implementation and use. Free variables, nondeterminism and encapsulated search turn out to be elegant and powerful means to express test-data generation.

Key words: Testing, Nondeterminism, Encapsulated Search, Curry

1 Introduction

Automatic test tools have to generate values of a certain type. For example, to test the function `reverse` which reverses a list we have to generate a variety of lists of values of some type.

We present the implementation of an automatic test tool for the functional logic programming language Curry. Functional logic languages like Curry extend the functional programming paradigm with *nondeterministic operations* and *free variables*. In [1] it was shown that a free variable can be seen as a nondeterministic generator that yields all values of its type. We argue that this result is not only of theoretical interest. We present a practical application of this new view on free variables. Instead of defining a test-case generator for lists, we can use a free variable of an appropriate list type. Moreover, the notion of nondeterminism greatly simplifies the implementation of custom generators.

In Curry, nondeterminism is introduced by operations with overlapping left hand sides. For example, the operation `bool` is nondeterministic because its left hand sides trivially overlap – they are identical. It is semantically equivalent to a free variable of type `Bool`.

```
bool = False
bool = True
```

The operation `bool` nondeterministically evaluates to `False` or `True`. The operation `bList` is semantically equivalent to a free variable of type `[Bool]`.

* Partially supported by the German Research Council (DFG) grant Ha 2457/5-2.

```
bList = []
bList = bool : bList
```

It yields an empty list or a list with a boolean head and a `bList` as tail. Therefore, `bList` nondeterministically yields all values of type `[Bool]`.

The above definitions are superfluous, because they evaluate to *every* value of their type and we can replace them by free variables. However, we can apply a similar technique to define custom generators that evaluate only to a subset of all possible values. For example, if we do not want to check `reverse` for empty lists, we can define an operation that nondeterministically yields all nonempty lists of type `[Bool]`.

```
neBList = bool : bList
```

We present an automatic test tool that uses nondeterministic operations for the generation of test data.

– We show that the generation of test data is already included in the concepts of functional logic programming. Therefore, the programmer does not have to learn a new syntax and the syntax for test data generation is very simple.
– We separate the generation of test data and its enumeration. Test data generators are nondeterministic operations. The nondeterminism is encapsulated [2] by an operation that yields a tree which contains all possible values. We present a new traversal strategy for such trees that serves well for the purpose of test data generation (Section 4). In contrast to other approaches, this enables us to ensure that every value is enumerated only once and that every value is eventually enumerated. The separation between generation and enumeration of test data allows a clear and flexible implementation of automatic testing.
– We extend the interface of test tools for functional languages with additional operations to specify properties of nondeterministic operations (Section 3).

2 Curry

Curry is a functional logic programming language whose syntax is similar to the syntax of the functional programming language Haskell [3]. In the following, we assume that the reader is familiar with the syntax of Haskell and only explain Curry specifics in detail. Apart from functional features (algebraic datatypes, higher-order functions, lazy evaluation), Curry provides the essential features of logic programming, viz., nondeterminism and free variables. Because of nondeterminism we use the term *operation* in the context of Curry instead of function. Free variables are introduced by the keyword `free` and nondeterminism by overlapping left hand sides. Curry does not follow a top-down strategy but evaluates every matching rule of an operation. For example, the binary operation `(?) :: a -> a -> a` nondeterministically yields one of its arguments.

```
x ? _ = x
_ ? x = x
```

In Curry you can encapsulate a nondeterministic value and get a deterministic tree that contains all possible values. In the following, we will use the term *search tree* when we talk about this tree structure. Note that this is not a search tree in the sense of an AVL or a Red Black tree. A search tree in our sense denotes a value of the following datatype.[1]

```
data SearchTree a = Value a | Or [SearchTree a]
```

The Curry system KiCS [4, 5] provides a primitive encapsulating operation `searchTree :: a -> SearchTree a` that takes a possibly nondeterministic value and yields the corresponding search tree. Encapsulating nondeterminism is still a topic of ongoing research. Nevertheless, all Curry implementations provide some kind of encapsulated search. The search tree for a deterministic value is a single `Value` leaf.

```
> searchTree True
Value True
```

Nondeterministic choices are reflected by `Or` nodes in the search tree.

```
> searchTree (False ? True ? False)
Or [Value False,Or [Value True,Value False]]
```

If we apply `searchTree` to `bList` or a free variable of type `[Bool]`, we obtain an infinite search tree because there are infinitely many values of type `[Bool]`. However, due to lazy evaluation, only those parts of the search tree are generated that are demanded by the surrounding computation. Therefore it is possible to guide the search by user defined traversal operations as the one presented in Section 4. Figure 1 visualizes the first six levels of the search tree that corresponds to a free variable of type `[Bool]`. Each inner node represents a nondeterministic choice for a constructor and its outgoing edges are labeled with the chosen constructor. The leaves of the tree are labeled with the corresponding lists of booleans.

[1] We do not consider *failure* throughout this paper, which could be expressed as `Or []`.

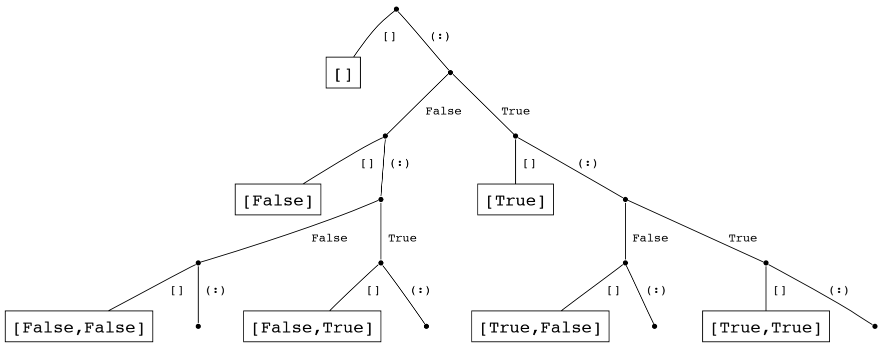

Fig. 1. Search tree for a free variable of type `[Bool]`

3 Using EasyCheck

The interface of EasyCheck is similar to the interface of QuickCheck [6] or G∀ST [7]. We provide a combinator `property :: Bool -> Property` that is satisfied if its argument deterministically evaluates to `True`. The deterministic equality operator `(-=-) :: a -> a -> Property` is satisfied if its arguments are deterministically evaluated to the same value.

However, EasyCheck is a test tool for a functional *logic* language and has special combinators to deal with nondeterminism. In this section we present the use of additional combinators to specify properties of nondeterministic operations.

We cannot use `(-=-)` to specify properties of nondeterministic operations because `(-=-)` demands its arguments to be deterministic. It would be questionable what a property like `(0?1) -=- 1` should mean and whether it should be equivalent to `1 -=- (0?1)`. We provide different combinators for nondeterministic operations that allow to address multiple values of an expression explicitly: `(~>), (<~), (<~>) :: a -> a -> Property`.

- The combinator `(~>)` demands that its left argument evaluates to every value of its right argument. The set of results of the left argument must be a *superset* of the set of results of the right argument.
- The combinator `(<~)` is dual to `(~>)` and demands that the set of results of its left argument is a *subset* of the set of results of the right one.
- Finally, `(<~>)` is satisfied if the sets of results of its arguments are *equal*. Note that `(<~>)` is not equivalent to `(-=-)` because the latter demands that the sets of results of its arguments are singleton sets.

In order to demonstrate nondeterministic testing, we consider an operation that inserts an element at an arbitrary position in a list.

```
insert :: a -> [a] -> [a]
insert x xs     = x : xs
insert x (y:ys) = y : insert x ys
```

The following property states that `insert` should insert the given element (at least) at the first and last position of the given list.

```
insertAsFirstOrLast :: Int -> [Int] -> Property
insertAsFirstOrLast x xs = insert x xs ~> (x:xs ? xs++[x])
```

To check a polymorphic property we have to annotate a type to determine possible test cases. For example `insertAsFirstOrLast` is tested for integers and lists of integers. We can use `easyCheck2` to verify that `insert` satisfies this property for the first 1,000 generated test cases.

```
> easyCheck2 insertAsFirstOrLast
OK, passed 1000 tests.
```

We provide operations `easyCheck`n to test properties of arity n for every reasonable n. As Curry does not support type classes, we cannot provide an operation `easyCheck` that handles properties of arbitrary arities.

We can employ `insert` to define a nondeterministic operation `perm` that computes all permutations of a given list.

```
perm :: [a] -> [a]
perm = foldr insert []
```

In order to test `perm`, we use one of the nondeterministic counterparts of the operation `property`, namely `always, eventually :: Bool -> Property`. These operations do not demand their arguments to be deterministic and are satisfied if all and any of the nondeterministic results of the argument are satisfied respectively. Assuming a predicate `sorted :: [Int] -> Bool` we can define a test for `perm` as follows.

```
permIsEventuallySorted :: [Int] -> Property
permIsEventuallySorted xs = eventually (sorted (perm xs))

> easyCheck1 permIsEventuallySorted
OK, passed 1000 tests.
```

The presented combinators are a relatively straightforward generalization of those found in QuickCheck for nondeterministic operations. We did not present all available combinators in this section. We introduce additional combinators in later sections when we use them.

4 Enumerating Test Cases

The primitive operation `searchTree :: a -> SearchTree a` encapsulates nondeterminism. It takes a possibly nondeterministic expression as argument and deterministically yields a search tree that contains all possible values of this expression. The standard libraries of Curry provide two operations to traverse a search tree and enumerate its values in depth- or breadth-first order. However, for test-case generation, we need a *complete*, *advancing* and *balanced* enumeration.

- We call an enumeration *complete* if every value is eventually enumerated. This property allows us to *prove* properties that only involve datatypes with finitely many values. Moreover, it implies that any node of an infinite search tree is reached in finite time.
- Furthermore, it is desirable to obtain reasonably large test cases early in order to avoid numerous trivial test cases. Therefore we want to visit the first node of the n-th level of a search tree after $p(n)$ other nodes where p is a polynomial. We call an enumeration with this property *advancing*.
- We call an enumeration *balanced* if the enumerated values are independent of the order of child trees in branch nodes. Balance is important in order to obtain diverse test cases.

Neither depth- nor breadth-first search fulfills all properties. Depth-first search is advancing[2] but incomplete and unbalanced. Breadth-first search is complete and almost balanced but not advancing because it generates a lot of small values before larger ones. Therefore, we present a new search tree traversal that is better suited for test-case generation.

4.1 Level Diagonalization

The following operation yields the list of levels of a search forest.

```
levels :: [SearchTree a] -> [[SearchTree a]]
levels ts  | null ts    = []
           | otherwise  = ts : levels [ u | Or us <- ts, u <- us ]
```

Note that not only values but *all* nodes of the forest are enumerated. If we would only enumerate leaves, we might need to process large sequences of inner nodes without being able to yield a node. As a consequence, large parts of levels would be visited what we aim to avoid due to performance reasons. By yielding also the inner nodes of the tree, we are able to process all levels incrementally. The operation `levelDiag` merges the different levels and extracts the values from the resulting enumeration.

```
levelDiag :: SearchTree a -> [a]
levelDiag t = [ x | Value x <- diagonal (levels [t]) ]
```

[2] not always, however, because it is incomplete

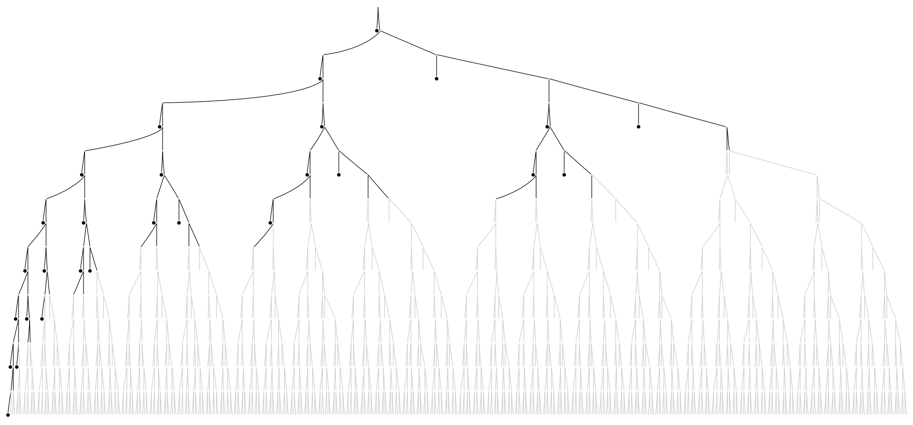

Fig. 2. Level diagonalization for `[Bool]` values

We do not simply concatenate the levels like in breadth-first search but use a list diagonalization operation reminiscent to the diagonalizing list comprehensions of Miranda [8]. The operation `diagonal` takes a list of lists and yields a list that contains all elements of the inner lists in a diagonally interleaved order.

```
diagonal :: [[a]] -> [a]
diagonal = concat . foldr diags []
 where diags []      ys   = ys
       diags (x:xs)  ys   = [x] : merge xs ys

       merge []        ys      = ys
       merge xs@(_:_)  []      = map (:[]) xs
       merge (x:xs)    (y:ys)  = (x:y) : merge xs ys
```

We can use `diagonal` to merge an infinite list of infinite lists:

```
> take 10 (diagonal [[ (i,j) | j <- [1..]] | i <- [1..]])
[(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),(4,1)]
```

You can think of the list of lists as a matrix. Note that the first element of the nth inner list is returned as $(n(n+1)/2)$th element of the result (or earlier). This means that `levelDiag` visits the first node of level n after visiting only $O(n^2)$ other nodes (compared to $O(2^n)$ for breadth-first search and $O(n)$ for depth-first search). Figure 2 shows part of the search tree that represents lists of booleans. The first 100 nodes that are visited by `levelDiag` are highlighted. Thanks to lazy evaluation, only a small part of the tree is computed. Observe that all values of lower levels are enumerated and, therefore, boundary cases are covered completely. Moreover, large values are enumerated reasonably early, i.e., `levelDiag` is

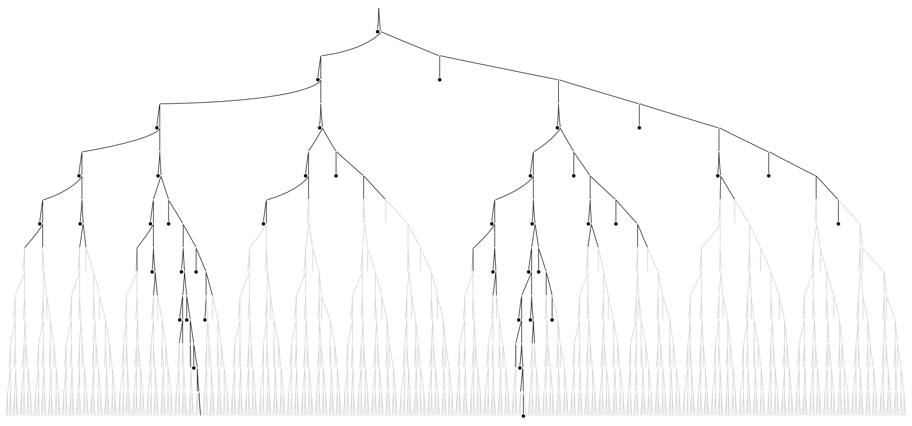

Fig. 3. Combined randomized level diagonalization for `[Bool]` values

advancing. It is also complete since every value is eventually enumerated. However, it is not balanced because it prefers left branches. Also, most of the visited nodes are in the left part of the tree.

4.2 Randomization

We employ `shuffle :: Int -> [a] -> [a]` in order to choose every branch with equal probability. It takes a random seed and yields a random permutation of its argument. We use this operation to shuffle search trees.

```
shuffleTree :: Int -> SearchTree a -> SearchTree a
shuffleTree _   (Value x)  = Value x
shuffleTree rnd (Or ts)    = Or (shuffle r (zipWith shuffleTree rs ts))
  where r:rs = split rnd
```

The function `split` computes an infinite list of uncorrelated random seeds from a given random seed. We can combine `shuffleTree` and `levelDiag` in order to obtain a complete, advancing and balanced search tree traversal. If we start the search with a fixed random seed, we obtain reproducible test cases. This is important because it is difficult to track down a bug if the same property fails in one execution and succeeds in another. Instead of shuffling only the children of nodes we could as well shuffle whole levels. This would give a good distribution but result in unacceptable performance since it causes the evaluation of large parts of the search tree.

Neither left nor right branches are preferred by randomized level diagonalization. But still large parts of the visited nodes are in the same part of the tree. This is desirable from a performance point of view because unvisited parts need not be computed. However, there is also an undesirable consequence: the larger the computed values are the more they resemble each other. QuickCheck does not show this behaviour because its test cases are independent of each other. But as a consequence the probability for enumerating a small value twice is very high.

Finding an efficient enumeration scheme that is complete, balanced and generates *sufficiently different* large values *early* deserves future work. In a first attempt, we apply randomized level diagonalization to different subtrees of the initial tree and combine the individual results. Figure 3 visualizes the effect of combining two randomized level diagonalizations.

5 Case Study

In this section we demonstrate how to test a heap implementation with EasyCheck. A heap is a tree that satisfies the *heap property*: the sequence of labels along any path from the root to a leaf must be non-decreasing. We want to evaluate test-case distribution for a complex datatype with multiple recursive components. Therefore, we use a custom datatype for natural numbers in binary notation as heap entries (cf. [9]).

```
data Heap = Empty | Fork Nat [Heap]
data Nat  = One | O Nat | I Nat
```

The heap implementation provides operations `empty :: Heap` to create an empty heap, `insert :: Nat -> Heap -> Heap` to add an element to a heap and `splitMin :: Heap -> (Nat,Heap)` to get the minimum of a nonempty heap and the heap without the minimum. The property `minIsLeqInserted` states that the minimum of a heap after inserting an element is less than or equal to the new entry.

```
minIsLeqInserted :: Nat -> Heap -> Property
minIsLeqInserted v h = property (m<=v)
  where (m,_) = splitMin (insert v h)
```

EasyCheck reports that this property is satisfied for 1000 test cases. The test uses a free variable to generate heaps. Because the test cases are generated by a free variable they do not necessarily satisfy the heap property. To count the number of valid heaps we employ the operation `classify :: Bool -> String -> Property -> Property` and a predicate `valid` on heaps. 505 of the first 1,000 heaps generated by a free variable are valid. We could use the operator `(==>) :: Bool -> Property -> Property` to reject invalid heaps like it is often done in QuickCheck. In this case all test cases – valid and invalid ones – are generated. This is insufficient if the percentage of valid test cases is small.

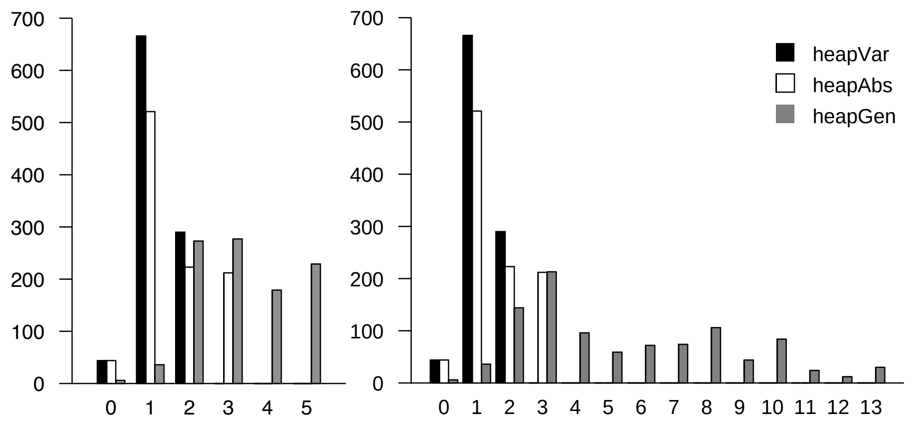

Fig. 4. Depth and size of generated `Heap` values

5.1 Generating Only Valid Heaps

In this subsection we discuss approaches to generate only valid heaps. In Subsection 5.2, we compare the distribution of test data with similar QuickCheck generators.

– We constrain a free variable by the predicate `valid`,
– we employ the abstract heap constructors `empty` and `insert`, and
– we explicitly generate an adequate subset of valid heaps.

For each of these approaches we measure the depth and the size of the generated heaps, i.e., the length of the longest path from the root to a leaf and the number of entries, respectively. The results of these measurements are depicted in Figure 4.

We can define a generator for valid heaps by narrowing a free variable.

```
heapVar | valid h = h where h free
```

This is a very simple and elegant way to generate only valid heaps. The test tool SparseCheck [10] employs the same idea, cf. Section 6 on related work. This definition is more efficient compared to the approach using `(==>)`. Thanks to laziness, invalid heaps are only generated as far as necessary to detect that they are invalid. If we use `(==>)` all test cases are generated completely and tested afterwards. Narrowing a free variable w.r.t. a predicate only generates valid heaps. Figure 4 shows that the majority of test cases that are generated by `heapVar` are small heaps. The probability that a larger heap that is generated by a free variable is valid is very small. Therefore it is much more effective to directly generate valid heaps. We can directly generate only valid heaps by using the abstract heap constructors `empty` and `insert`.

```
heapAbs = empty
heapAbs = insert unknown heapAbs
```

With `heapAbs` the average depth and size of the generated heaps increase noticably. More than 200 heaps of depth and size 3 are generated in 1,000 trials compared to none with `heapVar`.

Finally, we explicitly generate a subset of valid heaps, in order to further improve test-data distribution.

```
heapGen = Empty
heapGen = fork One

fork n = Fork m (heapList m) where m = n + smallNat

heapList _ = []
heapList n = fork n : heapList n

smallNat = One ? O One ? I One ? O (O One) ? I (O One) ? O (I One)
```

The number that is passed to `fork` defines the minimum size of all entries of the generated heap. That way, we assure that the generated heaps are valid. We also restrict the entries of the generated heaps: the difference of a label at any node and its children is at most 6. In our experiments, `heapGen` generates heaps up to depth 5 with up to 13 entries in the first 1,000 test cases. This is a significant improvement over the previously presented generators.

In order to evaluate the benefits of the presented search tree traversal, we have enumerated 1,000 results of the custom generator `heapGen` in breadth-first order. The largest heap of the first 1,000 results generated by breadth-first search has 5 entries. With combined randomized level diagonalization, the largest heap has 13 entries. The practical results documented in this section show that the new search tree traversal is an improvement over existing ones in the context of test-case generation. We have shown that EasyCheck serves well to generate sufficiently complex test data of non-trivial recursive datatypes like `Heap`. The generation of 1,000 heaps with the presented generators takes about one second on average using KiCS [4, 5] on a 2.2 GHz Apple MacBook™.

5.2 Comparison with QuickCheck

Due to lack of space, we cannot provide an extensive comparison with all important test tools for declarative languages. Therefore, we restrict ourselves to QuickCheck – the most widely used test tool for Haskell – and

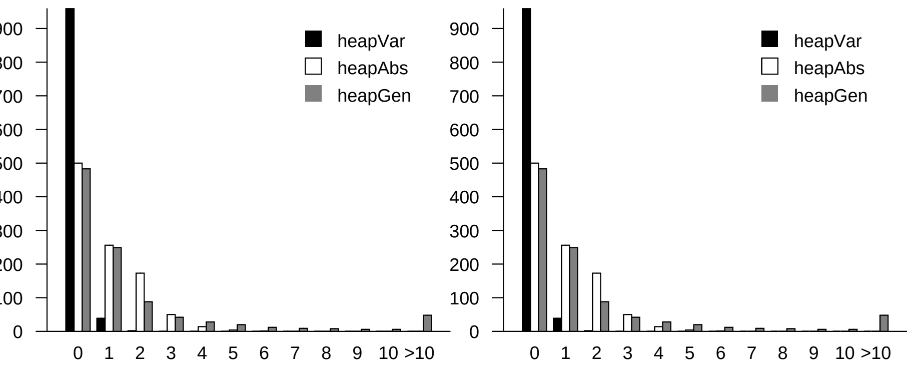

Fig. 5. Depth and size of generated `Heap` values

only describe data generators that resemble the ones presented before. QuickCheck provides additional combinators that allow to tune the distribution of test input. We want to point out, that the search presented in this paper relieves the programmer from the burden to manually adjust test-case distribution to some extent.

In QuickCheck we define data generators of type `Gen a` instead of non-deterministic values of type `a` by using `oneof :: [Gen a] -> Gen a`. It takes a list of generators and yields a generator where each of its arguments is chosen with equal probability. Figure 5 shows the distribution of the depth and size of the generated heaps for the three adapted generators. We have taken the average of 10 runs of QuickCheck.

The black bars represent the result for a generator that resembles a free variable. We just enumerate the constructors employing `oneof` and use the implication operator `(==>)` to reject invalid heaps. 960 out of 1,000 test cases are empty heaps. At first sight it seems easy to improve this. But if we use `(==>)` to prohibit the generation of empty heaps only around 300 out of the first 10,000 test cases are valid. Another way to improve this generator is by using the `frequency` operator. This operator is similar to `oneof` but each list element is assigned with a probability. Finding good probabilities is nontrivial. For example, we have faced the problem of undesirably large test input leading to a significant slowdown or even stack overflows. If we do not use `(==>)` only about 500 out of 1,000 heaps are valid and about 25 are valid and not empty. This is a well-known deficiency of QuickCheck also tackled by SparseCheck [10] with an approach similar to our restricted free variable. The distribution measured with the QuickCheck generators `heapAbs` and `heapGen` is acceptable. The number of generated heaps decreases with increasing depth and size but, nevertheless, larger heaps are generated.

The generation of test cases with QuickCheck is considerably faster than with EasyCheck. One reason is that a complete search is more time and space consuming than an incomplete random search. Also, the Curry system KiCS is a prototype and not as mature as modern Haskell systems – especially w.r.t. performance. Nevertheless, the run time of EasyCheck is acceptable, viz., a few seconds also for complex test-input.

6 Related Work

There are four implementations of automated test tools in functional languages, namely QuickCheck [6, 11], SmallCheck [12] and SparseCheck [10] in Haskell and G∀ST [7] in Clean [13]. Besides these, there are a couple of implementations of QuickCheck in other functional languages.

QuickCheck provides monadic combinators to define random test case generators in an elegant way. In order to test a function, the user has to define an instance of the type class `Arbitrary` for each type used as test input. Functional logic languages like Curry, already provide default generators for all datatypes, viz., free variables. Moreover, custom generators are defined by nondeterministic operations – no type class is necessary. With QuickCheck the user cannot ensure that all values of a datatype are used during a test. Furthermore, the same test data may be generated more than once. In EasyCheck, we employ a complete enumeration scheme for generating test data and ensure that every value is enumerated at most once. Moreover, every value would be eventually enumerated, if we would not abort the generation of test cases. In contrast to EasyCheck, QuickCheck can be used to generate higher order functions as test cases. The extension of EasyCheck to higher order values is future work.

The idea behind SmallCheck is that counter examples often consist of a small number of constructors. Instead of testing randomly generated values, SmallCheck tests properties for all finitely many values up to some size. Size denotes the number of constructors of a value. The size of the test cases is increased in the testing process. That is, we get the same results as SmallCheck for EasyCheck by using an iterative deepening or breadth-first traversal for the search trees. This demonstrates the power of the separation of test case generation and enumeration.

The automatic test tool G∀ST uses generic programming to provide test data generators for all types. In contrast to QuickCheck, this relieves the user from defining instances of type classes. If the user wants to define a custom generator he has to employ the generic programming extension of Clean [14]. In our approach, no language extension is necessary from the point of view of a functional logic programmer. Of course, we heavily use logic programming extensions built into functional logic languages. An outstanding characteristics of G∀ST is that properties can be proven if there are only finitely many checks. However, G∀STs enumeration scheme is not complete because left recursive datatypes lead to an infinite loop. In EasyCheck, we can prove properties because of the complete enumeration of test data. Moreover, the algorithm that generates test data is independent from the algorithm that enumerates it. Therefore, we can apply flexible enumeration schemes.

The idea of SparseCheck is based on a work by Fredrik Lindblad [15]. He proposes a system that uses narrowing to generate test cases that fulfil additional requirements. The approach of QuickCheck, SmallCheck and G∀ST is to generate all values and discard the values that do not satisfy these requirements. If only a small percentage of test cases fulfils the requirements this strategy fails. In SparseCheck, values that fulfil the requirements are generated using narrowing implemented in a logic programming library for Haskell [16]. This library provides similar features like Curry but separates functional from logic definitions that use special purpose combinators and operate on values of a special term type.

Recently, an approach to glass-box testing of Curry programs was presented in [17]. Glass-box testing aims at a systematic coverage of tested code w.r.t. a coverage criterion. The main difference between a black-box tool like EasyCheck and a glass-box tool is that EasyCheck generates test input in advance and a glass-box tool narrows test input during the execution of the tested function. An advantage of the glass-box approach is that input that is not processed by the program does not need to be generated. However, a glass-box approach is not lightweight because it requires a program transformation or a modification of the run-time system in order to monitor code coverage. Another disadvantage is that nondeterminism introduced by the tested function cannot be distinguished from the nondeterminism introduced by the test input. For example, a glass-box test tool cannot detect test input that leads to a failure or multiple results of the tested function. EasyCheck has combinators to deal with nondeterminism and, therefore, also with failure. A glass-box tool is usually employed to generate unit tests for deterministic operations.

The Curry implementation PAKCS [18] comes with CurryTest – a unit-test tool for Curry. Unit tests specify test in- and output explicitly, while specification-based tests in QuickCheck, G∀ST and EasyCheck only specify properties and the tool generates test data automatically. Thanks to the simplified form of test-data generation, defining a unit test in EasyCheck is as elegant as in CurryTest.

Checking defined functions against a seperate specification by systematically enumerating the arguments of the function can be seen as (bounded) model checking. See [19] for recent work on this topic. Usually, model checking refers to specifications of concurrent systems in temporal logic. XMC [20] is a model checker based on logic programming.

7 Conclusions and Future Work

We have presented EasyCheck[3] – a lightweight tool for automated, specification-based testing of Curry programs. Compared to similar tools for purely functional languages, we provide additional combinators for testing nondeterministic operations (Section 3).

Functional logic languages already include the concept of test-data generation. Free variables provide default generators for free and the declaration of custom generators is integrated in the programming paradigm via nondeterminism. It does not require additional type classes nor language extensions like generic programming. Logic programming features allow for a simple and elegant declaration of test-data generators. In Section 5 we discussed different approaches to defining custom test-case generators and compared them w.r.t. test-data distribution using a non-trivial datatype representing heap trees.

In EasyCheck, we separate test-case generation and enumeration, i.e., test-data generators can be written without committing to a specific enumeration scheme. Therefore, better enumeration schemes will improve test data distribution for existing generators. We present a new search tree traversal, viz., *combined randomized level diagonalization* (Section 4), and show that it is better suited for generating test cases than other traversals provided by Curry implementations.

Although this traversal turns out to be quite useful already, we plan to investigate new traversals to improve the diversity of large test cases. Another direction for future work is to examine time and space requirements of randomized level diagonalization. Furthermore, we would like to investigate the distribution of values generated by this traversal and to develop traversals with a similar distribution that do not rely on randomization.

[3] available at http://www-ps.informatik.uni-kiel.de/currywiki/tools/easycheck

References

1. Antoy, S., Hanus, M.: Overlapping rules and logic variables in functional logic programs. In: Proceedings of the International Conference on Logic Programming (ICLP 2006), Springer LNCS 4079 (2006) 87–101
2. Braßel, B., Hanus, M., Huch, F.: Encapsulating non-determinism in functional logic computations. Volume 6., EAPLS (2004)
3. Peyton Jones, S.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
4. Braßel, B., Huch, F.: Translating Curry to Haskell. In: Proc. of the ACM SIGPLAN Workshop on Curry and Functional Logic Programming, ACM Press (2005) 60–65
5. Braßel, B., Huch, F.: The Kiel Curry System KiCS. In Seipel, D., Hanus, M., eds.: Preproceedings of the 21st Workshop on (Constraint) Logic Programming. (2007) 215–223 Technical Report 434.
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices **35**(9) (2000) 268–279
7. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic automated software testing. In Peña, R., ed.: The 14th International workshop on the Implementation of Functional Languages, Selected Papers. Volume 2670 of LNCS., Madrid, Spain, Springer (2002) 84–100
8. Turner, D.A.: Miranda: a non-strict functional language with polymorphic types. In: Proc. of a conference on Functional programming languages and computer architecture, New York, NY, USA, Springer-Verlag New York, Inc. (1985) 1–16
9. Brassel, B., Fischer, S., Huch, F.: Declaring numbers. (2007) to be published.
10. Naylor, M.: A logic programming library for test-data generation. Available at `http://www-users.cs.york.ac.uk/~mfn/sparsecheck/` (2007)
11. Claessen, K., Hughes, J.: Quickcheck: Automatic specification-based testing. Available at `http://www.cs.chalmers.se/~rjmh/QuickCheck/` (2002)
12. Runciman, C.: Smallcheck: another lightweight testing library. Available at `http://www.cs.york.ac.uk/fp/darcs/smallcheck/` (2006)
13. Plasmeijer, R., van Eekelen, M.: Concurrent Clean language report (version 2.0). See also `http://www.cs.ru.nl/~clean` (2001)
14. Alimarine, A., Plasmeijer, M.J.: A generic programming extension for Clean. In: The 13th International workshop on the Implementation of Functional Languages, Selected Papers. Lecture Notes in Computer Science (2002) 168–185
15. Lindblad, F.: Property directed generation of first-order test data. In Morazan, M.T., Nilsson, H., eds.: Draft Proceedings of the Eighth Symposium on Trends in Functional Programming. (2007)
16. Naylor, M., Axelsson, E., Runciman, C.: A functional-logic library for wired˙ In: Proceedings of the ACM SIGPLAN workshop on Haskell. (2007)
17. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: Proc. of the 9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, ACM Press (2007)
18. Hanus, M., et al.: PAKCS: The Portland Aachen Kiel Curry System (version 1.8.1). Available at `http://www.informatik.uni-kiel.de/~pakcs/` (2007)
19. Cheney, J., Momigliano, A.: Mechanized metatheory model-checking. In: PPDP '07: Proc. of the 9th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming, New York, NY, USA, ACM (2007) 75–86
20. Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.N.: XMC: A logic-programming-based verification toolset. In: Computer Aided Verification. (2000) 576–580