

# Generic Generation of Elements of Types

Pieter Koopman and Rinus Plasmeijer

Institute for Computing and Information Science,  
Radboud University Nijmegen, The Netherlands  
{pieter,rinus}@cs.ru.nl

## Abstract

For a model based automatic test system it is essential to generate elements of the used data types automatically. In this paper we introduce an elegant algorithm that is able to generate a list of all elements of arbitrary types using generic programming techniques. In order to allow exhaustive testing for finite types we need to be able to determine that all elements of type are generated. This is done by systematic generation of the elements of a type.

In order to improve the results of testing we also show a variant of the generation algorithm that yields the elements in a pseudo random order. Both algorithms are very efficient and lazy; only the elements actually needed are generated.

Using the interface functions of restricted data types, like search trees, and the algorithms introduced here it is also possible to generate instances of these types. The elements stored in an instance of the search tree are generated by the default generic algorithm.

## 1 INTRODUCTION

In this paper we describe an elegant generic algorithm that yields a list of all elements of a type. Such an algorithm was needed in our model based test tool GAST [7, 8, 16]. Also in other application areas it is handy or necessary to be able to generate one or more instances of an arbitrary type. For instance in Generic Editor Components [1], where the data generation is used to create an instance for all arguments of a chosen constructor of an algebraic data type.

Generic programming is a programming technique where manipulations of data types are specified on a general representation of the data types instead of the data types themselves. Usually the system takes care of the conversion between the actual types and their generic representation and vice versa. There are various variants of generic programming for functional programming described. Most implementations of generics in functional programming languages are based on the ideas of Ralf Hinze [5, 6]. There are various implementations in Haskell, like Generic Haskell [4] and the *scrap your boilerplate* approach [10, 11]. In this paper we will use generic programming in CLEAN [12] as introduced by [2].

Basically a test system represents a predicate of the form  $\forall x \in T. P(x)$  by a function  $P :: T \rightarrow \text{Bool}$ . Test systems like QuickCheck [3] and GAST evaluates this function for a fixed number  $N$  of elements of type  $T$ . In order to do this the test system has to be able to generate elements of this type. If the system generates

an element  $t_c$  of type  $T$  such that  $P(t_c)$  yields **False**, the test system has found a counterexample: the property does not hold. When the property holds for all generated test data the property passes the test. If the size of type  $T$  is less than  $N$ , the property can even be proven by exhaustive testing. In GAST this is achieved by generating each instance of the type exactly once.

A popular way to generate elements of a type is by some pseudo random algorithm. Experience shows that a well chosen pseudo random generation and sufficient number of tests most likely covers all interesting cases and hence discovers counterexamples if they exist. Tools like QuickCheck and Torx [13] use pseudo random generation of test data. In GAST we use a systematic generation of test data for logical properties. The main advantage of this approach is that it is easy to detect that all elements of the type are generated and hence that the property is proven by exhaustive testing. An additional advantage is that systematic test data generation is that it makes testing more effective. In pseudo random testing it cannot be avoided that tests are repeated (the same test value is generated again), but this repetition of the test does not give any additional information in a referential transparent context.

Experiments have not shown examples where fully random data generation is more efficient than systematic data generation. There are plenty examples where the systematic data generation that starts with common boundary values is more effective. Since the systematic data generation is also very efficient, this way of testing is usually more effective than testing using pseudo random data.

For the implementation of the generation algorithm in a functional programming language a class is the most obvious language construct. The user has to define an instance of this class for each new type used in the predicates. This is some work for the test engineer. QuickCheck uses this approach in the generation of test data. Experience indicates that it requires at least some practice to write an effective test data generation algorithm for new types in this way. Using a generic programming approach has an advantage that the instance of a new type can be derived instead of coded by hand for each new data type. Only for data types with more restrictions than imposed by the type system, like balanced search trees, a manual implementation needs to be provided. Even for those types the generic algorithm can be used to determine the values to be inserted in the instance of the restricted type.

The problem solved here is unlike the scrap your boilerplate approach, since there is not data type to be traversed. Instead the elements of data type have to be generated out of thin air. It should be fairly easy to express the described algorithm in Generic Haskell [4].

In the next section we shortly review the original systematic data generation of GAST. In section 3 we introduce basic generic data generation algorithm. In section 4 we show how the order of test data generated can be changed in pseudo random way. Since the data generation algorithm uses only the type information, it cannot work correctly for restricted data types like search trees, balanced trees, AVL-trees et cetera. In section 5 we show how instances of these types can be

generated. Finally there are some conclusions. The reader is assumed to be familiar with generic programming in functional languages.

## 2 PREVIOUS WORK

One of the distinguishing features of GAST is that it is able to generate test data in a systematic way. This guarantees that test are never repeated, which is useless in a referential transparent language like CLEAN. For finite data types it is even possible to prove properties using a test system: a property is proven if it holds for all elements of the finite data type. GAST has used a systematic generic algorithm to generate test data from the very beginning. In this section we will review the original algorithm and the design decisions behind it.

In general it is impossible to test a property for all possible values. The number of values is simply too large (e.g. for the type `Int`), or even infinite (for every recursive data type). Boundary value analysis is a well-known technique in testing that tells that not all values are equally interesting for testing. The values where the specification or implementation indicates a bound and the values very close to such a bound are interesting test values. For numbers values like 0, 1 and  $-1$  are the most frequently occurring test values. For recursive types the non-recursive constructor (`[]` for lists and `Leaf` for trees) and small instances are the obvious boundary values. Therefore, these values have to be in the beginning of the list of data values generated. When specific boundary values for some situation are known, it is easy to include these in the tests, see [8] for details.

The initial algorithm [7] was rather simple, but also very crude and inefficient. The basic approach of the initial data generation algorithm was to use a tree to record the generic representations of the the values generated. For each new value to be generated the tree was extended according to the new value. In order to generate all small values early, the tree was extended in a breadth-first fashion. The definition of the tree used is:

```
:: Trace = Empty
      | Unit
      | Pair [(Trace,Trace)] [(Trace,Trace)]
      | Either Bool Trace Trace
      | Int [Int]
      | Done
```

The constructor `Empty` is used to indicate parts of the tree that are not yet visited. Initially, the entire tree is not yet visited. The constructor `Unit` indicates that a constructor is generated here. The two list of tuples of traces in a `Pair` together implement an efficient queue of new traces to be considered. The `Either` pairs the traces corresponding to the generic constructors `LEFT` and `RIGHT`. The `Boolean` indicates the direction where the first extension ought has to be sought. For basic types, like integers, special constructors, like `Int`, are used to record the values generated. When the generation algorithm discovers that some part of the tree

cannot be extended it is replaced by `Done`, in order to prevent fruitless traversals in future extensions of the tree. See [7] for more details.

### 3 GENERIC DATA GENERATION: BASIC APPROACH

The new generic data generation algorithm presented in this paper does not use a tree to record generated values. The use of the tree can be very time consuming. For instance the generation of all tuples of two characters takes nearly 20 minutes on a windows pc.

The generic function `gen` generates the lazy list of all values of a type by generating all relevant generic representations [2] of the members of that type.

```
generic gen a :: [a]
```

For the type `UNIT` there is only one possibility: the constructor `UNIT`.

```
gen {UNIT} = [UNIT]
```

For a *PAIR* that combines two kinds of values a naive definition using a list-comprehension would be `[PAIR a b \\a←f, b←g]`. However, we do not want the first element of `f` to be combined with all elements of `g` before we consider the second element of `f`, but some fair mixing of the values. This is also known as dovetailing. Suppose that `f` is the list `[a,b,c,..]` and `g` the list `[u,v,w,..]`. The desired order of pairs is *PAIR au*, *PAIR av*, *PAIR bu*, *PAIR aw*, *PAIR bv*, *PAIR cu*,.. rather than *PAIR au*, *PAIR av*, *PAIR aw*, .., *PAIR bu*, *PAIR bv*, *PAIR bw*, ... The *diagonalizing list comprehensions* from Miranda<sup>m</sup> [15] and the function `diag2` from the CLEAN standard environment exactly do this job. The function `diag2` is type `[a] [b] → [(a,b)]`, i.e. it generates a list of tuples with the elements of the argument lists in the desired order. Using a simple `map` function, or list comprehension the tuples are transformed to pairs.

```
gen {PAIR} f g = [ PAIR a b \\(a,b)←diag2 f g ]
```

For the choice in the type `EITHER` we use an additional Boolean argument to merge the elements in a nice interleaved way. The definition of the function `Merge` is somewhat tricky in order to avoid that it becomes strict in one of its list arguments. If the function `Merge` becomes strict in one of its list arguments it generates all possible values before the current value is yielded. This causes a `Heap full` error.

```
gen {EITHER} f g = Merge True f g
```

**where**

```
Merge :: !Bool [a] [b] → [EITHER a b]
Merge left as bs
  | left
  = case as of
      [] = map RIGHT bs
      [a:as] = [LEFT a: Merge (not left) as bs]
  = case bs of
      [] = map LEFT as
      [b:bs] = [RIGHT b: Merge (not left) as bs]
```

In order to let this merge algorithm terminate for recursive data types we assume that the non recursive case (like `Nil` for lists, `Leaf` for trees) is listed first in the type definition. Using some insight knowledge of the generic representation of allow us to make the right initial choice in `gen {EITHER}`. In principle the generic representation contains sufficient information to find the terminating constructor dynamically, but this is more expensive and does not add any additional power. Since the order of constructors in a data type does not have any other significance in `CLEAN` the assumption on the order of constructors is not considered a serious one.

The actual implementation of generics in `CLEAN` uses some additional constructors in order to store additional information about constructors, fields in a record etcetera. The associated instances for the generic function `gen` are:

```
gen {CONS}    f = map CONS f
gen {OBJECT}  f = map OBJECT f
gen {FIELD}   f = map FIELD f
```

Finally we have to provide instances of `gen` for the basic types of `CLEAN`. Some examples are:

```
gen {Int}     = [0: [i \ \ n←[1..maxint], i←[n, -n]]]
gen {Bool}    = [False,True]
gen {Char}    = map toChar ([32..126] ++ [9,10,13]) // the printable characters
gen {String}  = map toString lists
where
  lists :: [[Char]]
  lists = gen{★}
```

After these preparations the generation of user defined type like

```
:: Color = Red | Yellow | Blue
:: Rec = { c :: Color, b :: Bool, i :: Int }
:: ThreeTree = ThreeLeaf | ThreeNode ThreeTree ThreeTree ThreeTree
:: Tree x = Leaf | Node (Tree x) x (Tree x)
```

and predefined types like two and three tuple can be derived by

```
derive gen Color, Rec, ThreeTree, Tree, (,), (,,)
```

Unfortunately the order of elements in the predefined type list does not obey the give assumption. The predefined `Cons` constructor is defined before the `Nil` constructor. This implies that `gen` would always chooses the `Cons` constructor if generic generation would be derived for lists.

Instead of changing the assumption, or the implementation of `CLEAN`, we supply a specific instance of `gen` for lists, instead of deriving one. A straight forward implementation is de direct translation of the general algorithm, where the order of constructors is reversed (first the empty list []).

```
gen {[]} f = [ [] : [ [h:t] \ \ (h,t)←diag2 f (gen {*→*} f) ] ]
```

Here the parameter `f` is the list of all elements that should be placed in the generated lists. The value of this list will be provided by the generic system. A somewhat

type	values
[Color]	[Red, Yellow, Blue]
[Int]	[0, 1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6, 7, -7, 8, -8, 9, ...]
[(Color, Color)]	[(Red, Red), (Yellow, Red), (Red, Yellow), (Blue, Red), (Yellow, Yellow), (Red, Blue), (Blue, Yellow), (Yellow, Blue), (Blue, Blue)]
[[Color]]	[[], [Red], [Yellow], [Red, Red], [Blue], [Yellow, Red], [Red, Yellow], [Blue, Red], [Yellow, Yellow], [Red, Red, Red], ...]
[[Int]]	[[], [0], [1], [0, 0], [-1], [1, 0], [0, 1], [2], [-1, 0], [1, 1], [0, 0, 0], [-2], [2, 0], [-1, 1], [1, 0, 0], ...]
[Rec]	[(Rec Red False 0), (Rec Yellow False 0), (Rec Red True 0), (Rec Blue False 0), (Rec Yellow True 0), (Rec Red False 1), (Rec Blue True 0), (Rec Yellow False 1), ...]
[Tree Color]	[Leaf, (Node Leaf Red Leaf), (Node (Node Leaf Red Leaf) Red Leaf), (Node Leaf Yellow Leaf), (Node(Node(Node Leaf Red Leaf) Red Leaf) Red Leaf) Red Leaf), (Node (Node Leaf Red Leaf) Yellow Leaf), (Node Leaf Red (Node Leaf Red Leaf)), ...]

**TABLE 1. Examples of lists of values generated by gen**

more efficient implementation uses a cycle to use the generated lists as the tails of new lists.

```
gen { [] } f = list where list = [ [] : [ [h:t] \\ (h,t) ← diag2 f list ] ]
```

Here the function `diag2` is used again to get the desired mix extending existing lists and generating new lists with elements that are not used until now.

### 3.1 Examples

In order to illustrate the behavior of this algorithm we show (a part of) the list of values generated for some of the example types introduced above. The list of all values of type can be generated by an appropriate instance of `gen`. For instance the list of all elements of the type `Color` can be generated by:

```
list :: [Color]
list = gen { [*] }
```

The list of values generated by `list` with the indicated types are are show in table 1. For the types `Rec` and `Tree Color` the list of values is infinite, only an initial fragment of these lists is shown. Also for `Int` only an initial fragment of the list of values can be shown.

Note that the order of elements for parameterized types like `(Color,Color)` and `[Color]` reflects the dovetail behavior of the generation algorithm.

This algorithm is efficient. Generating  $10^6$  elements of a type takes typical 2 to 7 seconds on a basic windows PC, depending on the type of elements generated.

This algorithm generates all 9604 pairs of printable characters within 0.01 seconds, while the original algorithm outlined in section 2 needs 1136 seconds. This is five orders of magnitude faster.

#### 4 PSEUDO RANDOM DATA GENERATION

The actual algorithm used in GAST is slightly more complicated. It uses a stream of pseudo random numbers to make small perturbations to the order of elements generated. Basically the choice between `Left` and `Right` in `ggen {Either}` becomes a pseudo random one instead of strictly interleaved.

It is a widespread believe among testers that pseudo random generation of test values is needed in order to find issues<sup>1</sup> quickly. This seems somewhat in contradiction with rule that boundary values should be tested first. When we consider a predicate with multiple universal quantified variables of the same type, it can make sense to try the elements type in a somewhat different order for the various variables. We have encountered a number of examples where this indeed finds issues faster. On the other hand it is very easy to create examples where any perturbation of the order of test data delays the finding of counterexamples. In order to achieve the best of both worlds GAST uses a systematic generation of data values with a pseudo random perturbation of the order of elements discussed in section 3.

Is simple solution would be to randomize the generated list of elements based on a sequence of pseudo random numbers. This implies that test values will be generated (long) before they are actually used in the tests. This consumes just space and is considered undesirable in a lazy language like CLEAN.

As indicated above, the solution used in GAST is to replace the strict interleaved order of the choice in the instance of `gen` for `EITHER` by a pseudo random choice. The change of selecting `LEFT` of `Right` deserves some attention. At first sight a chance of 50% seems fine. This works also very well for nonrecursive type like `Color`, and recursive types like `list` and `Tree` from section 3.

For a type like `ThreeTree` this approach fails. If we chose the constructor `ThreeLeaf` with probability 50% than the change that all three arguments of the constructor `ThreeNode` terminate becomes too low. In practise such an algorithm generates too much huge or infinite data structures.

This problem can be solved by an elaborated analysis of the types involved in the data generation. Due to the possibility of nested and mutually recursive data types this analysis is far from simple. Fortunately, the is again a simple solution.

---

<sup>1</sup>A counterexample found by testing is called an *issue* until that it is clear that it is actually an error in the implementation. Other possible sources of counterexamples are for instance incorrect specifications and inaccuracies of the test system.

We still assume that nonrecursive constructor is the first constructor of a data type (if it exists). By increasing the probability of choosing the left branch in the recursive calls we can ensure that the small instance are near the beginning of the generated list of values.

In order to implement this we give the generic function `ggen` two arguments. The first is an integer indicating the recursion depth, the second one is a list of pseudo random numbers guiding the choice between left and right.

**generic** `ggen a :: Int [Int] → [a]`

The instance of this generation function for `EITHER` is the only one that changes significantly.

```
ggen {EITHER} f g n rnd = Merge n 1 (f n r3) (g (n+1) r4)
```

**where**

```
(r1,r2) = split rnd
(r3,r4) = split r2
```

```
Merge :: Int RandomStream [a] [b] → [EITHER a b]
```

```
Merge n [i:r] as bs
```

```
| (i rem n) ≠ 0
```

```
  = case as of
```

```
    [] = map RIGHT bs
```

```
    [a:as] = [LEFT a: Merge n r as bs]
```

```
  = case bs of
```

```
    [] = map LEFT as
```

```
    [b:bs] = [RIGHT b: Merge n r as bs]
```

The functions `split` splits a random stream into two independent random streams.

Also the order of elements in the predefined data types is changed in a pseudo random way. For enumeration types like `Bool` and `Char` the given order of elements is randomized.

```
ggen {Bool} n rnd = randomize [False,True] rnd 2 (λ_.[])
```

```
ggen {Char} n rnd
```

```
  = randomize (map toChar [32..126]++[9,10,13]) rnd 98 (λ_.[])
```

```
randomize :: [a] [Int] Int ([Int] → [a]) → [a]
```

```
randomize list rnd n c = rand list rnd n []
```

**where**

```
rand [] rnd n [] = c rnd
```

```
rand [] rnd n [x] = [x:c rnd]
```

```
rand [] rnd n l = rand l rnd n []
```

```
rand [a:x] [i:rnd] n l
```

```
  | n==0 || (i rem n) == 0
```

```
    = [a:rand x rnd (n-1) l]
```

```
    = rand x rnd n [a:l]
```

For integers and reals we even generate pseudo random values after the common boundary values. This introduces the possibility that tests are repeated, but for

these types it is usually less work than preventing duplicates. Due to the size of these types proofs are not feasible anyway.

```
ggen {Int} n rnd = randomize [0,1,-1,maxint,minint] rnd 5 id
```

This algorithm appears to be very effective in practise. It works also for some types that does not obey the rule that the nonrecursive constructor is the first one. Termination depends on the ratio between the number of points of recursion in the type and the number of constructors. One of the examples is the type list. This implies that this generation for lists can be derived in the from the general generic algorithm. In contrast to the previous algorithm, no hand coded definition is needed here.

```
derive ggen []
```

GAST uses the Marsenne Twister algorithm [14] from the CLEAN libraries for the generation of pseudo random numbers.

#### 4.1 Examples

The exact effect of the pseudo random data generation depends on the pseudo random numbers supplied as argument. By default the random numbers are generated by the function `genRandInt` from the CLEAN library `MersenneTwister`. The seed can be fixed to obtain repeatable tests, or for instance be obtained from the clock to obtain different test values for each run. The list of all pairs of colors with 42 as seed for the random number generation is generated by:

```
list :: [(Color,Color)]
list = ggen{[*]} 2 (genRandInt 42)
```

In table 2 we show the effects using the default random stream of GAST.

Note that the generated lists of values contain the same elements as the lists generated by the algorithm `gen` in section 3.1. The property that all instances of a type occur exactly once is preserved<sup>2</sup>. This algorithm needs about 40% more time to generate the same number of elements for a type compared to the function `gen`, but it is still very efficient. The test system GAST spends it time on evaluating predicates and the administration of the test results, but not on generating test data.

## 5 RESTRICTED DATA TYPES

Types like search trees, balanced trees, AVL-trees, ordered lists have more restrictions than the type system imposes. Since the generic algorithm does not know

---

<sup>2</sup>The shown instance for integers is the only exception. It generates pseudo random numbers with period  $2^{19937} - 1$ , and the 623-dimensional equidistribution property is assured. The type is so large that a proof by exhaustive testing is not feasible, preventing duplicated integers is more expensive than repeating the test for the duplicates if they might be generated. It is easy to change this definition, if that would be desired.

type	values
[Color]	[Red,Blue,Yellow]
[Int]	[0,-2147483648,2147483647,-1,1,684985474,862966190,-1707763078,-930341561,-1734306050,-114325444,-1262033632,-702429463,-913904323, ...]
[(Color,Color)]	[(Red,Red),(Yellow,Red),(Red,Blue),(Blue,Red),(Yellow,Blue),(Red,Yellow),(Blue,Blue),(Yellow,Yellow),(Blue,Yellow)]
[[Color]]	[Red],[],[Yellow],[Red,Red],[Blue],[Yellow,Red],[Red,Yellow],[Blue,Red],[Yellow,Yellow],[Red,Red,Red,Red,Red],[Blue,Yellow],...
[[Int]]	[[1],[],[-2147483648],[1,1],[-1],[-2147483648,1],[1,-1],[0],[-1,1],[-2147483648,-1],[1,1,2147483647,-1,0],[1,-1],[0],[-1,1], ...]
[Rec]	[(Rec Red False 2147483647),(Rec Yellow False 2147483647),(Rec Red True 2147483647),(Rec Blue False 2147483647), ...]
[Tree Color]	[Leaf,(Node (Node Leaf Red (Node (Node Leaf Yellow (Node Leaf Red Leaf)) Red Leaf)) Yellow (Node Leaf Red (Node (Node Leaf Red Leaf) Red Leaf))) ,(Node Leaf Yellow (Node Leaf Red (Node (Node Leaf Red Leaf) Red Leaf))), ...]

**TABLE 2. Examples of lists of values generated by ggen**

these restrictions, it cannot cope with them. The generic algorithm will generate instance that are type correct, but may or may not obey the additional constraints.

The interface of such a restricted type will contain functions to create an initial instance of the type, e.g. an empty tree, and to add elements to a valid instance of the restricted type. Using these constructor functions and the generic generation of elements to be included in the instance of the restricted type, we can easily generate instances of the generic type.

As example we will consider a search tree of integers. A typical interface to this abstract data type is:

```

:: SearchTree

empty :: SearchTree
ins   :: Int SearchTree → SearchTree
delete :: Int SearchTree → SearchTree
occurs :: Int SearchTree → Bool

```

Using the functions `ins` and `empty` appropriate trees can be constructed. This can

be used in the instance of `gen` or `ggen` by inserting lists of integers in the empty tree. These lists of integers are generated by the ordinary generic algorithm.

```
gen {SearchTree} = map (foldr ins empty) gen{[*]}
```

The initial part of the list of values is (using `E` for the empty tree and `N` as constructor for binary nodes):

```
[E, N E 0 E, N E 1 E, N E 0 E, N E -1 E, N E 0 (N E 1 E), N (N E 0 E) 1 E,
,N E 2 E, N (N E -1 E) 0 E, N E 1 E, N E 0 E, N E -2 E, N E 0 (N E 2 E),
,N (N E -1 E) 1 E, N E 0 (N E 1 E), N E -1 (N E 0 E), N E 3 E, ...
```

For the algorithm with pseudo random changes in the order, only the additional arguments of the function `ggen` have to be passed around.

This approach is applicable to every ordinary restricted data type, since they all have an initial value and an insert operator. Depending on the restricted type it is possible that duplicated values are generated by a naive implementation following this scheme.

## 6 RELATED WORK

Any test tool that wants to do more than only executing predefined test scripts, needs to generate these scripts. For any specification that contains variables, it is necessary to generate values for these variables. To the best of our knowledge this is the first approach to generate these values based on the type definition only.

The other tool that is able to test properties over types in a functional programming language is QuickCheck [3]. Its data generation is based on an ordinary class instead of on generic programming. This implies that the user has to define an instance of the generation class for each type used as in an universal quantification. Moreover, the generation algorithm uses pseudo random data generation without omitting duplicated elements. As a consequence Quickcheck is not able to determine that all elements of a type are used in a test. Hence, Quickcheck cannot stop at that point, nor conclude that it has achieved a proof by exhaustive testing.

In [9] we show how functions can be generated based on the grammar of the body of the functions. This grammar is a recursive algebraic data type. Hence the grammars used to describe the instances of the functions can be generated by the algorithm described in this paper.

## 7 DISCUSSION

This paper introduces an efficient and elegant generic algorithm to generate the members of arbitrary data types. The elements are generated from small to large as required for effective testing based on boundary values. We show also a variant of this algorithm that imposes a pseudo random perturbation of the order, but maintains the basic small to large order and avoids omissions or duplicates. This is believed to make finding counterexamples on average faster.

This algorithm is an essential component of the test tool GAST. The property that test data are not duplicated makes testing more efficient, evaluating a property two times for the same value will always yield an identical result in a functional context. The avoidance of omissions and duplicates makes it possible to proof properties for finite types by exhaustive testing.

The presented algorithms are efficient. Each of the algorithms is able to generate hundreds of thousands elements of a type within one second on a fairly basic Windows PC.

Apart from a very useful algorithm in the context of an automatic test system, it is also an elegant application of generic programming. The test system GAST follows the trend towards constructing general usable algorithms by generic programming techniques also for more traditional applications as comparing and printing values.

## REFERENCES

- [1] Peter Achten, Marko van Eekelen, Rinus Plasmeijer. Compositional Model-Views with Generic Graphical User Interfaces. In Jayaraman, ed. Proceedings Practical Aspects of Declarative Programming, PADL04, 2004. LNCS 3057.
- [2] Artem Alimarine and Rinus Plasmeijer. A Generic Programming Extension for Clean. In: Arts, Th., Mohnen M.: IFL 2001, LNCS 2312, pp 168–185, 2002.
- [3] Koen Claessen, John Hughes. QuickCheck: A lightweight Tool for Random Testing of Haskell Programs. ICFP, ACM, pp 268–279, 2000. See also [www.cs.chalmers.se/~rjmh/QuickCheck](http://www.cs.chalmers.se/~rjmh/QuickCheck).
- [4] Ralf Hinze and Johan Jeuring. *Generic Haskell: Practice and Theory*, Summer School on Generic Programming, 2002. See also <http://www.generic-haskell.org/>.
- [5] R. Hinze, and S. Peyton Jones *Derivable Type Classes*, Proceedings of the Fourth Haskell Workshop, Montreal Canada, 2000.
- [6] Ralf Hinze. *Polytypic values possess polykinded types*, Fifth International Conference on Mathematics of Program Construction, LNCS 1837, pp 2–27, 2000.
- [7] Pieter Koopman, Artem Alimarine, Jan Tretmans and Rinus Plasmeijer. GAST: Generic Automated Software Testing. In R. Peña, *IFL 2002*, LNCS 2670, pp 84–100, 2002.
- [8] Pieter Koopman and Rinus Plasmeijer. *Testing reactive systems with GAST*. In S. Gilmore, *Trends in Functional Programming 4*, pp 111–129, 2004.
- [9] Pieter Koopman and Rinus Plasmeijer. *Testing Higher Order Functions*. Draft proceedings 17th International Workshop on Implementation and Application of Functional Languages, IFL05, 2005. See also <https://www.cs.tcd.ie/ifl05/>.
- [10] Ralf Lämmel and Simon Peyton Jones *Scrap your boilerplate: a practical design pattern for generic programming*, ACM SIGPLAN Notices, **38**, 3, pp 26–37, mar, 2003, Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

- [11] Ralf Lämmel and Simon Peyton Jones, *Scrap more boilerplate: reflection, zips, and generalised casts*, Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004), ACM Press, 2004, pp 244–25.
- [12] Rinus Plasmeijer, Marko van Eekelen. *Clean language report version 2.1*. [www.cs.ru.nl/~clean](http://www.cs.ru.nl/~clean), 2005.
- [13] Jan Tretmans. *Testing Concurrent Systems: A Formal Approach*. In J. Baeten and S. Mauw, editors, *CONCUR'99 – 10<sup>th</sup>*, LNCS 1664, pp 46–65, 1999.
- [14] M. Matsumoto and T. Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Trans. on Modeling and Computer Simulation, Vol. 8, No. 1, January pp 3–30 , 1998.
- [15] Miranda is a trademark of Research Software Limited of Europe. David Turner *Miranda: a non-strict functional language with polymorphic types*, Proceedings FPLCA, LNCS 201, pp 1–16, 1985.
- [16] Arjen van Weelden, Martijn Oostdijk, Lars Frantzen, Pieter Koopman, Jan Tretmans: *On-the-Fly Formal Testing of a Smart Card Applet*, In Ryoichi Sasaki and Sihan Qing and Eiji Okamoto and Hiroshi Yoshiura: *Proceedings of the 20th IFIP TC11 International Information Security Conference SEC 2005* <http://www.sec2005.org/>, pp 564–576, Springer 0-387-25658-X, 2005.