Testing properties of generic functions

Patrik Jansson[1], Johan Jeuring[2], and students of the Utrecht University Generic Programming class[3]

[1] CSE, Chalmers University of Technology, Sweden, patrikj@chalmers.se
[2] ICS, Utrecht University, the Netherlands, johanj@cs.uu.nl
[3] L. Cabenda, G. Engels, J. Kleerekoper, S. Mak, M. Overeem, and K. Visser.

Abstract A datatype-generic function is a family of functions indexed by (the structure of) a type. Examples include equality tests, maps and pretty printers. Property based testing tools like QuickCheck and Gast support the definition of properties and test-data generators, and they check if a monomorphic property is satisfied by the test cases. Generic functions satisfy generic properties and this paper discusses specifying and testing such properties. It shows how generic properties and generators can be expressed, and explains three bugs we found and corrected in the Generic Haskell library.

1 Introduction

Software testing aims to find faults in software by comparing its behaviour with a specification. Testing comes in many flavours: validation testing, integration testing, system testing, unit testing, etc. We focus on property-based unit testing for datatype-generic functional programs.

In property-based testing, a specification is expressed in terms of executable properties. Together with a function a programmer writes one or more properties that should be satisfied by the function. Such properties can be used both as documentation (executable specifications) and as part of a test suite for regression testing. For example, consider the following excerpt from a Haskell module for manipulating bits.

```
data Bit = O | I deriving (Show, Eq)
bits2int      :: [Bit] → Int
bits2int bs   = bits2int' bs (length bs − 1)
  where bits2int' []       n = 0
        bits2int' (x : xs) n = bits2int' xs (n − 1) + bit2int x ∗ 2 ˆ n
int2bits    :: Int → [Bit]
int2bits n = if n ⩾ 0   then int2bits' n [] else []
  where int2bits' 0 bs = bs
        int2bits' n bs = int2bits' (n ‘div‘ 2) (int2bit (n ‘mod‘ 2) : bs)
bit2int b   = if b == O then 0 else 1
int2bit n   = if n == 0 then O else I
```

Functions *bits2int* and *int2bits* convert a list of bits to an integer and *vice versa*. To see if these functions are inverses we could check the following properties:

```
prop_int2bits_bits2int    :: [Bit] → Bool
prop_int2bits_bits2int bs = (int2bits . bits2int) bs == bs
prop_bits2int_int2bits    :: Int → Bool
prop_bits2int_int2bits n  = (bits2int . int2bits) n  == n
```

Checking with a property checker immediately reveals that they don't hold. A counterexample to the first property is $[\,O, I, I\,]$ (leading zeroes should be ignored in the first property), and to the second property is -3 (negative numbers are not properly encoded). Mistakes like these are common in specifications and programs, and ideas like Design-by-Contract [13] and Test-Driven Development [1] are now widely used in software development. For monomorphic programs and properties this is well understood, but for datatype-generic programs the testing area is largely unexplored.

A datatype-generic function is a family of functions indexed by a type. Examples of generic functions are equality, map, and pretty printers. A generic function can be seen as a template algorithm that can be instantiated with (the structure of) a data type. Similarly, a generic property can be seen as a template property that can be instantiated with a data type to obtain a simple property. A generic library is a highly reusable software component, and by stating and verifying properties for such a library, the effort spent on verification pays off over and over again.

QuickCheck [2] is one of the most advanced tools for testing properties of functional programs. It supports the definition of properties and random test-data generators in Haskell, and checks that a monomorphic property passes the test cases. Gast [8] is a tool similar to QuickCheck, but for property-based testing in Clean [15]. Gast comes with *generic* test-data generators which work for arbitrary data types. But Gast enumerates data which leads to bad coverage for infinite datatypes (we expand on this later).

This paper

- discusses specifying and testing properties of generic functions,
- shows how parts of the Generic Haskell [4] library can be specified and tested using QuickCheck (revealing three bugs), and
- defines generic QuickCheck generators using Generic Haskell. This means we get the best of both worlds — we combine the strengths of QuickCheck with generic support inspired by Gast.

The paper is organised as follows. Section 2 briefly introduces and compares a few property-based testing tools. Section 3 introduces generic programming in Generic Haskell. Section 4 shows how QuickCheck is used to check properties of generic functions. Section 5 discusses verification of the Generic Haskell library and explains the bugs found. Section 6 presents different ways of generating test cases for arbitrary data types. Section 7 concludes and discusses future work.

2 Property-based testing tools

This section introduces the testing tools QuickCheck, Gast and SmallCheck.

QuickCheck
QuickCheck is an automatic testing tool for Haskell programs. The programmer provides a specification of the program, in the form of executable properties that functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators defined in the QuickCheck library. The library provides combinators to define properties, observe the distribution of test data, and define test-case generators.

Many properties are simple Boolean functions, implicitly universally quantified over all arguments:

$$\begin{aligned}
&\mathit{prop\_PlusAssoc} && :: \mathsf{Float} \to \mathsf{Float} \to \mathsf{Float} \to \mathsf{Bool}\\
&\mathit{prop\_PlusAssoc}\ x\ y\ z && = (x + y) + z == x + (y + z)
\end{aligned}$$

To test a property it is passed to the function *test*:

```
Main> test prop_PlusAssoc
Falsifiable, after 8 tests:
-4.6
-4.0
3.6
```

Here QuickCheck finds a simple counterexample illustrating that finite precision Floats don't behave like ideal real numbers.

The QuickCheck library also provides conditional properties, where tests not satisfying the precondition are discarded:

$$\begin{aligned}
&\mathit{prop\_SmallPrime} && :: \mathsf{Integer} \to \mathsf{Property}\\
&\mathit{prop\_SmallPrime}\ x && = \mathit{prime}\ x \Longrightarrow x < 88
\end{aligned}$$

```
Main> test prop_SmallPrime
OK, passed 100 successful tests.
```

Here QuickCheck has generated a few hundred test cases (randomly chosen numbers x) out of which 100 were prime and all of those were unfortunately <88. In this case the brute force solution of asking QuickCheck to generate even more test cases works, but in general the coverage for the default generators is bad for "sparse" properties. Fortunately, it is also possible to define custom generators — here is an example using the infinite list of *primes*:

$$\begin{aligned}
&\mathit{primeNumbers} :: \mathsf{Gen\ Integer}\\
&\mathit{primeNumbers} = \mathbf{do}\ n \leftarrow \mathit{arbitrary}\\
&\qquad\qquad\qquad\qquad\ \ \mathit{return}\ (\mathit{primes}\ !!\ \mathit{abs}\ n)\\
&\mathit{prop\_SmallPrime2} :: \mathsf{Property}\\
&\mathit{prop\_SmallPrime2} = \mathit{forAll}\ \mathit{primeNumbers}\ (\lambda x \to x < 88)
\end{aligned}$$

```
Falsifiable, after 39 successful tests:
97
```

QuickCheck also supports a simple but powerful way of searching for small counter examples. When a test case fails, QuickCheck tries to shrink the test case until a "local minimum" is found. As an example, for the first property of the bits example in the introduction we get the following result:

```
Main> test prop_int2bits_bits2int
Falsifiable, after 2 successful tests
(shrunk failing case 3 times):
[0]
```

Gast
Gast (Generic Automated Software Testing) [8] is a property-based testing tool which can be seen as a QuickCheck for Clean. Gast is implemented in the non-strict functional language Clean [15], a close relative to Haskell. From the users perspective, Gast is very similar to QuickCheck — properties can be defined as normal Boolean functions and tests can be run by calling the function *test*:

```
listsAreShort    :: [Int] → Bool
listsAreShort xs = length xs < 5
Start = test listsAreShort
```

which in this case results in the answer

```
Passed after 500 tests.
```

This example is chosen to show that some care needs to be taken in interpreting the results from testing: Gast enumerates data in a breadth-first manner, only randomising the order "within each level". For recursive data types this is problematic, because of the exponential growth of the search space — as we can see, the first 500 test cases do not contain a single list with more than four elements. QuickCheck generates lists up to length around 200 in the same situation.

The enumeration approach used by Gast does have a few advantages: it avoids generating the same test case more than once and it makes it possible to actually prove properties over finite domains within the same framework (using exhaustive testing). Gast does not need to shrink failing test cases because they are generated and tested in order of increasing size. The Clean implementation of Gast is fast, but for recursive data types the exponential search space means that reaching reasonably sized test cases just takes too long.

SmallCheck
While finishing this paper we learnt about Runcimans recent work on SmallCheck — a combinator library for lightweight testing in Haskell closely based on QuickCheck. SmallCheck tests properties for all values up to some depth, progressively increasing the depth used. The SmallCheck library shares many of the strengths and weaknesses of Gast, but has no generic programming support. Both our generic generators and our methodology for testing generic properties would be useful in combination with SmallCheck, but that is left as future work.

3 Generic programming in Generic Haskell

In this section we introduce type-indexed functions by means of an example and we explain how type-indexed functions become generic in Generic Haskell.

Type-indexed functions
A type-indexed function takes an explicit type argument, and can have behaviour that depends on this type argument. For example, suppose the unit type Unit, sum type :+:, and product type :*: are defined as follows:

$$\begin{array}{ll}
\mathbf{data}\ \mathsf{Unit} & = U \\
\mathbf{data}\ \mathsf{a}\ {:}{+}{:}\ \mathsf{b} & = \mathit{Inl}\ \mathsf{a} \mid \mathit{Inr}\ \mathsf{b} \\
\mathbf{data}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{b} & = \mathsf{a}\ {:}{*}{:}\ \mathsf{b}.
\end{array}$$

We use infix type constructors :+: and :*: and an infix value constructor :*: to ease the presentation. The type-indexed function *eq* checks equality of two values. We define the function *eq* on booleans, the unit type, sums, and products as follows in Generic Haskell:

$$\begin{array}{llll}
\mathit{eq}\{\!|\mathsf{Bool}|\!\} & b_1 & b_2 & = \mathit{eqBool}\ b_1\ b_2 \\
\mathit{eq}\{\!|\mathsf{Unit}|\!\} & U & U & = \mathit{True} \\
\mathit{eq}\{\!|\alpha\ {:}{+}{:}\ \beta|\!\} & (\mathit{Inl}\ x_1) & (\mathit{Inl}\ x_2) & = \mathit{eq}\{\!|\alpha|\!\}\ x_1\ x_2 \\
\mathit{eq}\{\!|\alpha\ {:}{+}{:}\ \beta|\!\} & (\mathit{Inr}\ y_1) & (\mathit{Inr}\ y_2) & = \mathit{eq}\{\!|\beta|\!\}\ y_1\ y_2 \\
\mathit{eq}\{\!|\alpha\ {:}{+}{:}\ \beta|\!\} & \_ & \_ & = \mathit{False} \\
\mathit{eq}\{\!|\alpha\ {:}{*}{:}\ \beta|\!\} & (x_1\ {:}{*}{:}\ y_1) & (x_2\ {:}{*}{:}\ y_2) & = \mathit{eq}\{\!|\alpha|\!\}\ x_1\ x_2 \wedge \mathit{eq}\{\!|\beta|\!\}\ y_1\ y_2,
\end{array}$$

where *eqBool* is the standard equality function on Booleans. The *eq* type signature is $\mathit{eq}\{\!|\mathsf{a} :: *|\!\} :: (\mathit{eq}\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$. The context $(\mathit{eq}\{\!|\mathsf{a}|\!\}) \Rightarrow$ in this signature says that *eq* has a *dependency* [11] on *eq*. A type-indexed function f depends on another type-indexed function g if g is used on a type argument (a *dependency variable*) α in the definition of f. The occurrences of α and β in the definition of *eq* are dependency variables.

Generic functions
A type-indexed function such as *eq* does not only work on the types that appear as type indices in its definition. To see why *eq* is in fact *generic* and works on arbitrary data types, we give a mapping from data types to structure types such as units, sums, and products. If there is no specific case for a type in the definition of a generic function, generic behaviour is derived automatically by the compiler by exploiting the structural representation.

For example, the definition of the function *eq* that is generically derived for lists is equivalent to the following specific definition:

$$\begin{array}{llll}
\mathit{eq}\{\!|[\alpha]|\!\} & [\,] & [\,] & = \mathit{True} \\
\mathit{eq}\{\!|[\alpha]|\!\} & (x : \mathit{xs}) & (y : \mathit{ys}) & = \mathit{eq}\{\!|\alpha|\!\}\ x\ y \wedge \mathit{eq}\{\!|[\alpha]|\!\}\ \mathit{xs}\ \mathit{ys} \\
\mathit{eq}\{\!|[\alpha]|\!\} & \_ & \_ & = \mathit{False}
\end{array}$$

To obtain this instance, the compiler needs to know the structural representation of lists, and how to convert between lists and their structural representation. We will describe these components in the remainder of this section.

Structure types
The structural representation (or structure type) of types is expressed in terms of units, sums, products, and base types such as integers, characters, etc. For example, for the list and tree data types defined by

$$\begin{array}{lll}
\textbf{data}\ [\mathsf{a}] & = [\,] & \mid \mathsf{a} : [\mathsf{a}] \\
\textbf{data}\ \mathsf{Tree\ a\ b} & = \mathit{Tip}\ \mathsf{a} & \mid \mathit{Node}\ (\mathsf{Tree\ a\ b})\ \mathsf{b}\ (\mathsf{Tree\ a\ b}),
\end{array}$$

we obtain the following structural representations:

$$\begin{array}{lll}
\textbf{type}\ [\mathsf{a}]^{\circ} & = \mathsf{Unit} & \mathsf{:+:}\ \mathsf{a}\ \mathsf{:*:}\ [\mathsf{a}] \\
\textbf{type}\ \mathsf{Tree}^{\circ}\ \mathsf{a\ b} & = \mathsf{a} & \mathsf{:+:}\ \mathsf{Tree\ a\ b}\ \mathsf{:*:}\ \mathsf{b}\ \mathsf{:*:}\ \mathsf{Tree\ a\ b},
\end{array}$$

where we assume that :*: binds stronger than :+: and both type constructors associate to the right. Note that the representation of a recursive type is not recursive, and refers to the recursive type itself: the representation of a type in Generic Haskell only represents the structure of the top level of the type.

Embedding-projection pairs
If a type a can be embedded in, or represented by, another type b, a witness of this property can be stored as a pair of functions converting back and forth (an embedding-projection pair):

$$\textbf{data}\ \mathsf{EP\ a\ b} = \mathit{Ep}\{\mathit{from} :: \mathsf{a} \rightarrow \mathsf{b}, \mathit{to} :: \mathsf{b} \rightarrow \mathsf{a}\}.$$

A type T can be embedded in its structure-representation type T°, witnessed by a value $\mathit{conv}_{\mathsf{T}} :: \mathsf{EP\ T\ T}^{\circ}$. For example we get $\mathit{conv}_{[\,]} = \mathit{Ep}\ \mathit{from}_{[\,]}\ \mathit{to}_{[\,]}$:

$$\begin{array}{ll}
\mathit{from}_{[\,]} & :: [\mathsf{a}] \rightarrow [\mathsf{a}]^{\circ} \\
\mathit{from}_{[\,]}\ [\,] & = \mathit{Inl}\ U \\
\mathit{from}_{[\,]}\ (x : \mathit{xs}) & = \mathit{Inr}\ (x\ \mathsf{:*:}\ \mathit{xs}) \\
\mathit{to}_{[\,]} & :: [\mathsf{a}]^{\circ} \rightarrow [\mathsf{a}] \\
\mathit{to}_{[\,]}\ (\mathit{Inl}\ U) & = [\,] \\
\mathit{to}_{[\,]}\ (\mathit{Inr}\ (x\ \mathsf{:*:}\ \mathit{xs})) & = x : \mathit{xs}.
\end{array}$$

The definitions of such embedding-projection pairs are automatically generated by the Generic Haskell compiler for all data types that appear in a program.

Tying the knot
Using structure-representation types and embedding-projection pairs, a call to a generic function on a data type T is reduced to a call on type T°. The inductive definition of a generic function is used to generate an instance on the structure type T°. For example, for equality we obtain a function of type $\mathsf{T}^{\circ} \rightarrow \mathsf{T}^{\circ} \rightarrow \mathsf{Bool}$. To convert this function back to a function of type $\mathsf{T} \rightarrow \mathsf{T} \rightarrow \mathsf{Bool}$ we use the function *bimap* [3]. Function *bimap* is a bi-directional generic variant of the well-known map function, of the following type:

$$bimap\{\!|\mathsf{a} :: *, \mathsf{b} :: *|\!\} :: (bimap\{\!|\mathsf{a}, \mathsf{b}|\!\}) \Rightarrow \mathsf{EP\ a\ b}.$$

When using *bimap*, it is only applied to one type argument which is used both for a and b. So $bimap\{\!|\mathsf{a}|\!\}$ is an embedding-projection pair of type $\mathsf{EP\ a\ a}$. The type index can have higher kind, and the fully generic type for *bimap* is actually kind-indexed. For example, the instance of *bimap* on the type constructor Tree has the following type:

$$bimap\{\!|\mathsf{Tree}|\!\} :: \mathsf{EP\ a\ c} \to \mathsf{EP\ b\ d} \to \mathsf{EP\ (Tree\ a\ b)\ (Tree\ c\ d)}$$

Kind-indexed types can be defined in GH but are not used in this paper.

To turn a function of type $\mathsf{T}^\circ \to \mathsf{T}^\circ \to \mathsf{Bool}$ into a function of type $\mathsf{T} \to \mathsf{T} \to \mathsf{Bool}$, we call $bimap\{\!|\mathsf{T} \to \mathsf{T} \to \mathsf{Bool}|\!\}$ in which we use $conv_{\mathsf{T}}$ for the T-values. Thus we obtain a function of type $\mathsf{EP}\ (\mathsf{T}^\circ \to \mathsf{T}^\circ \to \mathsf{Bool})\ (\mathsf{T} \to \mathsf{T} \to \mathsf{Bool})$. The *from*-component of this embedding-projection pair is the function that converts the implementation of the generic function on structure types back to a function that works on the original data type values. Hence, if the generic function is defined for structure types such as Unit, :+:, and :*:, we do not need cases for specific data types such as List or Tree anymore. For primitive types such as Int, Float, IO or →, no structure type is available. Therefore, for a generic function to work on these types, specific cases are necessary.

Generic abstractions, local redefinitions, and default cases

Generic Haskell supports a number of extensions that simplify defining and using generic functions. First, using a *generic abstraction*, we can define a generic function in terms of another generic function instead of by induction on the structure types. For example, we can test pointwise equality of functions by means of the following generic function:

$$\begin{aligned}
&feq\{\!|\mathsf{b} :: *|\!\} \ :: \ (eq\{\!|\mathsf{b}|\!\}) \Rightarrow (\mathsf{a} \to \mathsf{b}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{Bool}\\
&feq\{\!|\mathsf{b}|\!\}\ f\ g = \lambda x \to eq\{\!|\mathsf{b}|\!\}\ (f\ x)\ (g\ x)
\end{aligned}$$

which is a generic abstraction that is defined in terms of, and depends on, the generic equality function. Note that each generic abstraction (including *feq*) works for types of of *fixed kind*. This is in contrast to generic functions defined by induction on the type structure which work for types of arbitrary kinds.

Generic functions may have dependencies. We can use *local redefinition* to redefine the dependencies of generic functions. For example, if we want equality on lists of characters to be case insensitive, we can write

$$\begin{aligned}
&equalCaseInsensitive \qquad :: \mathsf{Char} \to \mathsf{Char} \to \mathsf{Bool}\\
&equalCaseInsensitive\ x\ y = toUpper\ x == toUpper\ y\\
&\mathbf{let}\ eq\{\!|\alpha|\!\} = equalCaseInsensitive\\
&\mathbf{in}\ eq\{\!|[\alpha]|\!\}\ \texttt{"Generic Programming"}\ \texttt{"GENERIC programming"}
\end{aligned}$$

Another way in which we may obtain this behaviour is via a so-called *default case*, which allows us to extend an existing generic function by adding new cases or overriding existing ones.

```
cieq{|a :: *|} :: (cieq{|a|}) ⇒ a → a → Bool
cieq extends eq      -- default for cieq is eq
cieq{|Char|} x y = toUpper x == toUpper y
```

Many more examples of these extensions, and a discussion about the merits and disadvantages of these constructs can be found in Löh's thesis [10].

4 QuickCheck for generic functions

This section explains how we use QuickCheck for testing properties of generic functions. The biggest challenge here is to *formulate* generic properties. We start this section with a number of generic properties, and then discuss how we can use QuickCheck to test them.

Minimal and maximal values
Haskell's prelude contains a class *Bounded* defined by

```
class Bounded a where minBound, maxBound :: a
```

The methods *minBound* and *maxBound* should satisfy

```
prop_minBound x = compare minBound x ≢ GT
prop_maxBound x = compare maxBound x ≢ LT
```

that is, *minBound* is smaller than or equal to any other value, and *maxBound* is larger than or equal to any other value. The method *compare*::a → a → Ordering, in the class *Ord* (used for totally ordered data types) allows a single comparison to determine the precise ordering of two elements:

```
data Ordering = LT | EQ | GT
```

Haskell allows to derive the bounds automatically for some user-defined data types (enumeration types and single-constructor data types whose constituent types are in *Bounded*). Generic Haskell's library contains definitions of the generic values *gminBound* and *gmaxBound* for all algebraic types (not only for those types for which Haskell supports deriving). To formulate generalisations of the properties above, we also need the generic compare function *gcompare* from Generic Haskell's library. The desired properties now read as follows:

```
prop_gminBound{|t :: *|}  :: (gcompare{|t|}, gminBound{|t|}) ⇒ t → Bool
prop_gminBound{|t|} x     = gcompare{|t|} (gminBound{|t|}) x ≢ GT
prop_gmaxBound{|t :: *|}  :: (gcompare{|t|}, gmaxBound{|t|}) ⇒ t → Bool
prop_gmaxBound{|t|} x     = gcompare{|t|} (gmaxBound{|t|}) x ≢ LT
```

Note that the properties are formulated as generic abstractions, thus restricting t to types of kind $*$. Later we will see an example of using local redefinition as a work-around.

Properties of gmap
The generic equivalent *gmap* of the well-known *map* function applies zero or more functions (depending on the kind of its data-type argument) to the appropriate elements in a value of the data type.

$$gmap\{\!|\mathsf{a} :: *, \mathsf{b} :: *|\!\} :: (gmap\{\!|\mathsf{a}, \mathsf{b}|\!\}) \Rightarrow \mathsf{a} \rightarrow \mathsf{b}$$

Function *gmap* is defined as the deep identity function, and local redefinition can be used to obtain *map*-like behaviour. For $tree :: \mathsf{Tree\ Int\ Char}$ we can write

$$\begin{aligned}
&\mathbf{let}\ gmap\{\!|\alpha|\!\} = toEnum\\
&\phantom{\mathbf{let}}\ gmap\{\!|\beta|\!\} = fromEnum\\
&\mathbf{in}\ \ gmap\{\!|\mathsf{Tree}\ \alpha\ \beta|\!\}\ tree
\end{aligned}$$

to convert the integers to characters, and the characters to integers.

Properties of *gmap* can be derived from properties of *map*. Function *map* on lists is a part of a functor, and satisfies the functor laws: it preserves the identity, and distributes over composition:

$$\begin{aligned}
&map\ id && {=}{=}{=}\ id\\
&map\ (f\ .\ g) && {=}{=}{=}\ map\ f\ .\ map\ g
\end{aligned}$$

Here $({=}{=}{=})$ is pointwise equality of functions on lists, implemented by $feq\{\!|[\alpha]|\!\}$, see Section 3. Generalised versions of these properties should hold for the generic map function *gmap*. We take the composition law as an example.

For a type constructor $\mathsf{c} :: * \rightarrow *$ we have two function arguments (the f and g in the above property), and for a type constructor $\mathsf{d} :: * \rightarrow * \rightarrow *$ we have four function arguments (two functions per type argument):

$$\begin{aligned}
&prop\_gmap\_comp1\,\{\!|\mathsf{c}|\!\}\ f\ g && = gmap\{\!|\mathsf{c}|\!\}\ (f\ .\ g)\ {=}{=}{=}\ (gmap\{\!|\mathsf{c}|\!\}\ f\ .\ gmap\{\!|\mathsf{c}|\!\}\ g)\\
&prop\_gmap\_comp2\,\{\!|\mathsf{d}|\!\}\ f\ g\ h\ j && = gmap\{\!|\mathsf{d}|\!\}\ (f\ .\ g)\ (h\ .\ j)\ {=}{=}{=}\\
& && \quad\ (gmap\{\!|\mathsf{d}|\!\}\ f\ h\ .\ gmap\{\!|\mathsf{d}|\!\}\ g\ j)
\end{aligned}$$

Hinze [3] shows how to generalise this property to types of arbitrary kinds. The resulting, fully generic property is *kind-indexed*, but cannot be expressed in GH.

Testing generic properties
As the examples of generic properties for *gmap* show, a generic property may involve kinds, type constructors, polymorphic types, higher-order functions, and plain values. To test a property, we have to supply values for each of the above components. QuickCheck can generate values of monomorphic types and functions, but generating type constructors, let alone kinds, is out of reach. This implies, amongst others, that we have to instantiate the properties on fixed monomorphic types

Happily, generating type constructors and kinds is not necessary. To *prove* a generic property, it suffices to prove instances of the property on the structure types [3]. Similarly, to *test* the validity of a generic property, it suffices to test the validity of a property on the structure types. To test the validity of a property on all structure types, we would have to write a separate instance of the property for each structure type. Take the property *prop_gminBound* as an example. The simplest structure type is Unit. For this case, the following expression would be tested:

```
gcompare{|Unit|} (gminBound{|Unit|}) U ≢ GT
```

By definition of *gminBound* and *gcompare*, this test, and the equivalent tests for Int and Char trivially pass. For the sum type case QuickCheck would need to test something like

```
prop_gminBound_Sum cmpa cmpb mba mbb x =
  (∀a . cmpa mba a ≢ GT) ⟹
  (∀b . cmpb mbb b ≢ GT) ⟹
  (gminBound_Sum cmpa cmpb mba mbb x ≢ GT)
```

Since *gminBound* depends on *gcompare* and on itself, *prop_gminBound_Sum* takes five arguments. The last argument is a value of type a :+: b, and the other arguments are instances of *gcompare* and *gminBound* on the types a and b, respectively.

In general, implications $P \Longrightarrow Q$ may be hard to test in QuickCheck. In particular when the condition P is often *False*, Q is only tested for a few of the generated test cases. For many of the properties this turns out to be a problem — for example, for most properties of equality the condition requires independently generated values to be equal. For *prop_gminBound_Sum* the problem is even worse, because the left-hand side of the implication includes a local universal quantification which is not implementable with QuickCheck properties. We can solve this problem by supplying generators: instead of testing $\lambda x \rightarrow P\ x \Longrightarrow Q\ x$ we test $\mathit{forAll}\ \mathit{genP}\ (\lambda x \rightarrow Q\ x)$. In general it is hard or impossible to convert a property to a generator, but to obtain testable properties we need at least a good approximation of *genP*.

To avoid some of the problems with implications and local quantification, we define a data type which combines the structure types in a single data type, and use that data type for testing generic functions. The following data type combines the most important structure types, and is easily extended with more cases for basic structure types. (In the code we have also used an infix constructor for *STProd*.)

```
data StructureTypes a = STUnit
                      | STInt Int
                      | STChar Char
                      | STProd (StructureTypes a) (StructureTypes a)
                      | STLabel{ anA :: a}
```

The data type StructureTypes contains cases for units, integers, characters, products, and labels. The cases for sums and constructors are implicit, but appear since there is a choice between constructors in the data type, and there are constructor names in the data type. The type is parameterised to make it possible to test *gmap* — in all other tests we instantiate the type parameter (to Int).

To test the validity of the property *prop_gminBound* with this approach we use the QuickCheck function *test* on the data type $\mathsf{StructureTypes\ Int}$:

$$\mathit{test}\ (\mathit{prop\_gminBound}\{\!|\mathsf{StructureTypes\ Int}|\!\})$$

QuickCheck generates test cases from the data type $\mathsf{StructureTypes\ Int}$ if we provide a generator (an element of $\mathsf{Gen}\ (\mathsf{StructureTypes\ Int})$). We have used an instance of the generic generator *arb3* (defined later in Section 6).

5 Properties of the Generic Haskell Library

The Generic Haskell library consists of a number of basic generic functions that are used often in generic programs. Many functions of the Generic Haskell library are generic versions of Haskell's prelude [14] functions. This includes functions that implement the methods that are derivable in Haskell, and generalisations of list functions such as *map*, *sum*, *prod*, *and*, etc. Another source of inspiration for the Generic Haskell library is PolyLib [7], the library of PolyP, which contains many basic generic functions and some properties.

Since generic functions from the library will often be used as basic building blocks in generic-programming applications, it is important that they are correct. Therefore, the generic functions in the Generic Haskell library are natural candidates for applying our approach to testing generic functions.

The Generic Haskell library consists of twelve modules, of which we will consider the following six: `Eq`, `Compare`, `Enum`, `Bounds`, and `ReadShow`, corresponding to the derivable Haskell classes *Eq*, *Ord*, *Enum*, *Bounded*, *Read*, and *Show*, and the module `Map`, which implements the generic map function *gmap*. We will introduce the generic functions used in this section briefly, often referring to their non-generic Haskell equivalents. More information about the functions in the Generic Haskell library can be found in the user's guide [12].

Properties of gread and gshow
Functions *gread* and *gshow* implement the derivable *read* and *show* functions from Haskell. Just as in Haskell, they are defined in terms of helper functions *gshowsPrec* and *greadsPrec*. Reading a value after showing it should be the identity. Showing after reading need not be the identity: parsing may fail or the original value might contain concrete syntax (spaces, newlines) that is not generated by the *show* function (like the leading zeros in the bits example from the introduction). We have tested the following property:

$$\begin{aligned}
&\mathit{prop\_gread\_gshow}\{\!|\mathsf{t} :: *|\!\} &&:: (\mathit{eq}\{\!|\mathsf{t}|\!\}, \mathit{greadsPrec}\{\!|\mathsf{t}|\!\}, \mathit{gshowsPrec}\{\!|\mathsf{t}|\!\}) \Rightarrow \\
& && \quad \mathsf{t} \to \mathsf{Bool} \\
&\mathit{prop\_gread\_gshow}\{\!|\mathsf{t}|\!\} &&= \mathit{feq}\{\!|\mathsf{t}|\!\}\ (\mathit{gread}\{\!|\mathsf{t}|\!\} \,.\, \mathit{gshow}\{\!|\mathsf{t}|\!\})\ \mathit{id}
\end{aligned}$$

where *feq* is pointwise equality of functions, see Section 3.

It turned out that *gread* could not cope with named fields in data types. The $\mathsf{StructureTypes\ a}$ data type contains the constructor $\mathit{STLabel}\{\,\mathit{anA} :: \mathsf{a}\,\}$. The *anA* field triggered a runtime error (pattern match failure) in *gread*. QuickCheck does not trap exceptions, so when a property fails, QuickCheck fails instead of just counting this as a failed test case. Fortunately, the Haskell compiler ghc includes (unsafe) functions to catch exceptions in pure code, so by wrapping the property in an exception handler returning *False* for all exceptions, we have used QuickCheck to find the bug.

```
Main> test (protect prop_gread_gshow_STInt)
Falsifiable, after 3 successful tests
(shrunk failing case 3 times):
STLabel {anA =-2}
```

The problem was actually not in *gread*, but in *gshow*. There was no space character after the equality sign, so when a negative integer was shown, the two characters "=-" were later parsed by *gread* as one token. A one-character change to the source code fixed this problem, but revealed another bug, this time in *gread*. Function *gread* did not allow parentheses around $\mathit{STLabel}\{\,\mathit{anA} = 2\,\}$, while *gshow* (and the derived *show* in Haskell) printed parentheses. After this second fix, all tests passed. (Adding infix constructors to $\mathsf{StructureTypes}\ a$ we revealed yet another bug, but constructor fixity problems was already noted in the Generic Haskell release notes so we already knew that.)

Properties of gmap
Function *gmap* preserves the identity:

$$\begin{array}{l}
\mathit{prop\_gmap\_id}\{\!|\mathsf{t}|\!\} :: (\mathit{eq}\{\!|\mathsf{t}|\!\}, \mathit{gmap}\{\!|\mathsf{t},\mathsf{t}|\!\}) \Rightarrow \mathsf{t} \to \mathsf{Bool} \\
\mathit{prop\_gmap\_id}\{\!|\mathsf{t}|\!\} = \mathit{feq}\{\!|\mathsf{t}|\!\}\ (\mathit{gmap}\{\!|\mathsf{t}|\!\})\ \mathit{id}
\end{array}$$

To test this function, we instantiate it on the type $\mathsf{StructureTypes\ a}$.

$$\begin{array}{lll}
\mathit{prop\_gmap\_id\_ST} :: (\mathit{Eq}\ \mathsf{a}) \Rightarrow \mathsf{StructureTypes\ a} \to \mathsf{Bool} & & \\
\mathit{prop\_gmap\_id\_ST} = & \mathbf{let} & \mathit{eq}\{\!|\mathsf{a}|\!\} \quad\ = (==) \\
 & & \mathit{gmap}\{\!|\mathsf{a}|\!\} = \mathit{id} \\
 & \mathbf{in} & \mathit{prop\_gmap\_id}\{\!|\mathsf{StructureTypes\ a}|\!\}
\end{array}$$

Function *gmap* distributes over composition. We formulate the distributivity property by means of three copies of *gmap*, of which we only define *gmap1* here.

$$\begin{array}{l}
\mathit{gmap1}\{\!|\mathsf{a} :: *, \mathsf{b} :: *|\!\} :: (\mathit{gmap1}\{\!|\mathsf{a},\mathsf{b}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{b} \\
\mathit{gmap1}\ \mathbf{extends}\ \mathit{gmap} \\
\mathit{prop\_gmap\_comp}\{\!|\mathsf{a} :: *, \mathsf{b} :: *, \mathsf{c} :: *|\!\} :: \\
\quad (\mathit{eq}\{\!|\mathsf{c}|\!\}, \mathit{gmap1}\{\!|\mathsf{b},\mathsf{c}|\!\}, \mathit{gmap2}\{\!|\mathsf{a},\mathsf{b}|\!\}, \mathit{gmap3}\{\!|\mathsf{a},\mathsf{c}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{Bool} \\
\mathit{prop\_gmap\_comp}\{\!|\mathsf{t}|\!\} = \mathit{feq}\{\!|\mathsf{t}|\!\}\ (\mathit{gmap1}\{\!|\mathsf{t}|\!\} \,.\, \mathit{gmap2}\{\!|\mathsf{t}|\!\})\ (\mathit{gmap3}\{\!|\mathsf{t}|\!\})
\end{array}$$

To instantiate this property on the data type $\mathsf{StructureTypes\ a}$, we locally redefine the *gmap* copies.

```
prop_gmap_comp_ST op f g =
  let eq{|a|}        = op
        gmap1{|a|} = f;   gmap2{|a|} = g;   gmap3{|a|} = f . g
  in prop_gmap_comp{|StructureTypes a|}
```

We have also tested *gmap* on the structure types (:+:), (:*:), etc.

Properties of enum

Function *enum* exhaustively enumerates all possible instances of a particular data type.

```
enum{|t :: *|} :: (enum{|t|}) ⇒ [t]
```

For example, *enum*{|Int|} yields the list of all possible (machine-) integers. A property that should hold for this function is the following:

```
prop_enum{|t|}          :: t → Bool
prop_enum{|t|} value = value ∈ enum{|t|}
```

This property says that any value of type t should be in the enumeration of that type. Interestingly, checking this property is not really an option — at least for most real-life data types. Recursive data types often have infinitely many values, so using QuickCheck to test whether or not a value appears in the enumeration may take infinitely long. When testing the property instantiated with the StructureTypes Int data type QuickCheck just looped, and at first we thought this was just to be expected. But a more careful examination revealed that the property looped already for the first test case, which should have been small enough to be found early in the enumeration list. It turned out to be a subtle bug in the definition of the generic *enum* function. The enumeration used a version of Cantor diagonalisation which was "non-productive" in the case of infinite lists. By replacing just the diagonalisation function, the generic *enum* implementation worked as expected. Still, the property remains effectively untestable — already some trees built from just seven constructors are more than 10000 elements down the list.

The problem is just another instance of the problem Gast has with coverage for recursive data types (remember that Gast also uses (randomised) enumeration): While every element is *somewhere* in the enumeration list, and will *eventually* be generated by Gast, only small elements are reachable (will be tested by Gast) within reasonable time. Testing the enumeration property with Gast (instead of QuickCheck) is possible but not very useful — it is not very surprising that values (test cases) generated from an enumeration list actually are elements of a very similar enumeration list.

Another property of *enum* relates *enum* to the generic function *empty* that returns the 'least' value of a type. For example, for the List type *empty* would return the empty list.

```
prop_enum_empty{|t|} :: Bool
prop_enum_empty{|t|} = empty{|t|} ∈ enum{|t|}
```

As the type signature reveals, this is more a unit test than a QuickCheck property. No random value is generated, so QuickCheck tests the same thing in each test. It would be more interesting to range over different types for t, but this does not fit the (current, non-generic) QuickCheck framework.

Properties of gcompare
Function *gcompare* generalises the derivable *compare* function from Haskell. We have tested what corresponds to reflexivity, anti-symmetry and transitivity for *gcompare*. Transitivity can be expressed as a QuickCheck property by:

$$\begin{aligned}
&\mathit{prop\_gcompare\_trans}\{\!|\mathsf{t} :: *|\!\} &&:: (\mathit{gcompare}\{\!|\mathsf{t}|\!\}) \Rightarrow \mathsf{t} \to \mathsf{t} \to \mathsf{t} \to \mathsf{Property}\\
&\mathit{prop\_gcompare\_trans}\{\!|\mathsf{t}|\!\}\ x\ y\ z &&= \mathit{gcompare}\{\!|\mathsf{t}|\!\}\ x\ y == \mathit{gcompare}\{\!|\mathsf{t}|\!\}\ y\ z \Longrightarrow\\
& && \quad\ \mathit{gcompare}\{\!|\mathsf{t}|\!\}\ x\ y == \mathit{gcompare}\{\!|\mathsf{t}|\!\}\ x\ z
\end{aligned}$$

This captures transitivity for $(<)$, $(==)$ and $(>)$ when $\mathit{gcompare}\{\!|\mathsf{t}|\!\}\ x\ y$ has values LT, EQ and GT. We use the QuickCheck conditional operator $\Longrightarrow$ to rule out non-interesting test cases. Reflexivity and anti-symmetry are implemented in a similar fashion.

Another property relates function *gcompare* with the generic equality function *eq*. Function *gcompare* returns *EQ* iff function *eq* returns *True*.

$$\begin{aligned}
&\mathit{prop\_gcompare\_eq}\{\!|\mathsf{t} :: *|\!\} &&:: (\mathit{gcompare}\{\!|\mathsf{t}|\!\}, \mathit{eq}\{\!|\mathsf{t}|\!\}) \Rightarrow \mathsf{t} \to \mathsf{t} \to \mathsf{Bool}\\
&\mathit{prop\_gcompare\_eq}\{\!|\mathsf{t}|\!\}\ x\ y &&= (\mathit{gcompare}\{\!|\mathsf{t}|\!\}\ x\ y == EQ) == \mathit{eq}\{\!|\mathsf{t}|\!\}\ x\ y
\end{aligned}$$

This concludes the section on properties for generic functions in the Generic Haskell library. Formulating and testing these properties has been useful: we have discovered three bugs in the library.

6 Generic generators

Normally, QuickCheck requires a user to write a test-case generator for a user-defined data type on which QuickCheck is used. Generic programming allows us to automatically generate test cases for any given data type. This makes testing properties of (generic) functions easier. This section shows the implementation of generic generators in Generic Haskell. We could have chosen any of the approaches to generic programming to implement generic generators. The expressivity and type safety of Generic Haskell, and the recently added generic views feature, are the most important reasons why we use Generic Haskell. A detailed comparison of the different approaches to generic programming in Haskell can be found elsewhere [5].

Porting the Gast generator to Generic Haskell
For Clean a generic approach to generating test cases is already available: Gast (Generic Automated Software Testing) [8]. We have translated their implementation of pseudo random data generation [9] into Generic Haskell.

$$\mathit{generate}\{\!|\mathsf{g} :: *|\!\} :: \mathsf{Int} \to \mathsf{StdGen} \to [\mathsf{g}]$$

To make this a generator we can use the same technique as in the *primeNumbers* example — let *gast* be the (often infinite) list from *generate* and pick the value at a random index n. We just have to be careful not to index outside the list in case it turns out to be finite.

Thus we obtain a QuickCheck generator, written in Generic Haskell, which works for all Haskell data types. But, unfortunately, it has the same weakness for recursive types as the Gast generator in that it takes very long before any reasonably sized elements are generated. Worse, where Gast can use the systematic generation of test data for exhaustive checking for finite types, QuickCheck cannot guarantee to generate all elements (incompleteness). Still, it is convenient to have a fully generic generator around, and it can be modified with default cases and local redefinitions to customise its behaviour for selected constructors or types.

Non-terminating generators
Instead of first enumerating and then selecting it should be possible to define a generic generator directly. As a first try we can define the following generic generator:

```
arb1{|a :: ∗|}     :: (arb1{|a|}) ⇒ Gen a
arb1{|Unit|}       = return U
arb1{|Int|}        = arbitrary
arb1{|Char|}       = arbitrary
arb1{|α :+: β|} = arb_Sum (arb1{|α|}) (arb1{|β|})
arb1{|α :*: β|} = liftM2 (:*:) (arb1{|α|}) (arb1{|β|})


arb_Sum        :: Gen a → Gen b → Gen (a :+: b)
arb_Sum ga gb = oneof [liftM Inl ga, liftM Inr gb]
```

This generator is very simple, works for all data types and does generate reasonably sized values, but it has at least two drawbacks: a skewed distribution and possible non-termination.

The first problem is because Generic Haskell encodes multiple-constructor data types with nested binary sums, which means that *arb1* will give a very skewed distribution of the constructors. If p_i denotes the probability of constructor C_i we get $p_i = 1/2^i$ for $i \in \{1..n-1\}$. Here a balanced encoding would help and the next Generic Haskell release will support this as described in the Generic Views [6] paper. It is possible to work around this problem already in the current version of Generic Haskell by first analysing the data type, but we have not done so.

The second problem is more subtle, but it was noted already in the first QuickCheck paper (for a specific Tree data type). For recursive data types that branch into more than one subtree, it is fairly easy to accidentally define a generator that often fails to terminate (or, actually, terminates but with an infinite tree as the result). The problem is that if a branching constructor is often generated, the final tree is only finite if all the subtrees are finite and after a few branches the number of subtrees is high. The skewed distribution offers some degree of protection against these infinite trees, but this Bin data type is an example of the problem:

```
data Bin = B1 Bin Bin | B2 Bin Bin | L.
```

Here the probability to generate L is 1/4 and the probability for a finite tree is only 1/3.

A terminating generic generator

The solution to the termination problem is to use *sized* generators — we use a parameter n to limit the size of the generated trees. For a generic function it is not obvious to define what "size" should measure, but one simple choice is the number of constructors in the tree. Using a sized generator, we generate trees of size at most n. The first few cases in the definition are simple generalisations of *arb1*:

```
arb2{|a :: ∗|}       :: (arb2{|a|}, empty{|a|}) ⇒ Int → Gen a
arb2{|Unit|}       n = return U
arb2{|Int|}        n = arbitrary
arb2{|Char|}       n = arbitrary
arb2{|α :+: β|}    n = arb_Sum (arb2{|α|} n) (arb2{|β|} n)
```

Our size measure tells us that we should reduce the size when passing through a constructor and distribute the size over the two subtrees in the product. In the product case it is tempting to just use

```
arb2{|α :*: β|} n = liftM2 (:*:) (arb2{|α|} (n / 2)) (arb2{|β|} (n / 2))
```

but that would tend to generate almost balanced trees. Instead we divide the size randomly over the two subtree:

```
arb2{|Con c α|}      n  = liftM Con (arb2{|α|} (n − 1))
arb2{|α :*: β|}      n
   | n > 1 = do m ← choose (1, n − 1)
                x  ← arb2{|α|} m
                y  ← arb2{|β|} (n − m)
                return (x :*: y)
   | n ⩽ 1 = return (empty{|α|} :*: empty{|β|})
```

This generator works for all data types, it always terminates and generates finite trees (if there are any). It still has the skewed constructor distribution and it has a similar problem with a skewed size distribution for nested products. Both these problems can be avoided with a balanced view or with an analysis of the data type. Initial experiments are promising, but messy, so we leave that for future work.

Better distribution for regular data types
A problem with all the "fully generic" generators is that they cannot treat the recursive case differently from other cases. As an example, the *arb2* generator for a normal list will distribute the size parameter evenly between the element and the tail. This makes long lists very unusual and the sizes of the elements will decrease exponentially along the list. For lists we can include a special case in the definition, but similar problems occur also for other data types. Generic Haskell has been extended with some Generic Views [6], and using the Fix view it is possible to detect the recursive case, at least for regular data types.

Using the latest version of Generic Haskell (1.61) we have implemented yet another (sized) generic generator:

$$arb3\{\!|\mathsf{a} :: *|\!\} :: \mathsf{Int} \to \mathsf{Gen\ a}$$

This generator produces finite elements and has an even distribution of constructor probabilities and subtree sizes. The limitation is that it only works for regular data types (no mutual recursion and recursive occurrences must have the same parameters). The code depends on the generic function

$$children\{\!|\mathsf{a} :: *\ \textit{viewed}\ \mathsf{Fix}|\!\} :: \mathsf{a} \to [\mathsf{a}]$$

which is the classical example of what could be done in PolyP but cannot be done in the "old" Generic Haskell implementation.

7 Conclusions and future work

We have shown how we can formulate and test properties of generic functions, we have used QuickCheck to test the Generic Haskell libraries and we have defined a few generic QuickCheck generators.

Since an inductive proof of a property of a generic function only requires cases for the structure types used to represent data types, it suffices to test properties of generic functions on these structure types. We go one step further and collect the structure types into one representative type, StructureTypes a, which we use to instantiate the generic functions before testing them.

We have implemented a number of properties for generic functions in the Generic Haskell library. Formulating and testing these properties has revealed three bugs in the library. We have not yet completed the description of the properties of the functions in the library, so we expect (but do not hope) to find more bugs.

The generic QuickCheck test-case generators produce test data with a much better spread than the Gast generator. We have explored several variants with different random distributions and we have identified the Generic Views extension of GH as an important step towards better generic generators.

While implementing the different tests using QuickCheck we encountered a few problems, in particular with exception handling and a better control of the size of generated test cases. It turned out that the latest version of QuickCheck (obtained from CVS) solves most of these problems.

Future work consists of finishing formulating properties for the functions in the Generic Haskell library, further fine-tuning the generic QuickCheck test-data generators and adding tests of (non-)strictness. Another idea we would like to investigate is to generate random *types* as well as random values, and use these randomly generated types for testing, instead of the StructureTypes a type. It would also be natural to add generic support to SmallCheck.

Acknowledgements. A. Rodriguez, N.A. Danielsson and anonymous referees commented on previous versions of this paper.

References

1. K. Beck. *Test-Driven Development by Example.* Addison Wesley, 2003.
2. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP'00*, pages 268–279. ACM Press, 2000.
3. R. Hinze. *Generic Programs and Proofs.* Bonn University, 2000. Habilitation.
4. R. Hinze and J. Jeuring. Generic Haskell: practice and theory. In R. Backhouse and J. Gibbons, editors, *Generic Programming*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.
5. R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. Technical Report UU-CS-2006-022, ICS, Utrecht University, 2006. To appear in Datatype-Generic Programming, LNCS, Springer, 2007.
6. S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In T. Uustalu, editor, *MPC'06*, volume 4014 of *LNCS*. Springer-Verlag, 2006.
7. P. Jansson and J. Jeuring. PolyLib – a polytypic function library. In *Workshop on Generic Programming, Marstrand*, June 1998.
8. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *IFL'02*, pages 84–100, 2002.
9. P. Koopman and R. Plasmeijer. Generic generation of elements of types. In *TFP'05*, pages 167–179. Tallinn, 2005.
10. A. Löh. *Exploring Generic Haskell.* PhD thesis, Utrecht University, 2004.
11. A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In O. Shivers, editor, *ICFP'03*, pages 141–152. ACM Press, August 2003.
12. A. Löh, J. Jeuring, and A. Rodriguez (editors) et al. The Generic Haskell user's guide, Version 1.60 - Diamond release. Technical Report UU-CS-2006-049, ICS, Utrecht University, 2006.
13. R. Mitchelland and J. McKim. *Design by Contract: by example.* Addison-Wesley, 2002.
14. S. Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report.* Cambridge University Press, 2003.
15. R. Plasmeijer and M. van Eekelen. *Clean Language Report version 2.1*, 2005.