Generating Generic Functions

Johan Jeuring

Alexey Rodriguez

Gideon Smeding

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2006-039

www.cs.uu.nl
ISSN: 0924-3275

Generating Generic Functions

Johan Jeuring Alexey Rodriguez Gideon Smeding

Utrecht University, The Netherlands

{johanj,alexey,gjsmedin}@cs.uu.nl

Abstract

We present an approach to the generation of generic functions from user-provided specifications. The specifications consist of the type of a generic function, examples of instances that it should "match" when specialized, and properties that the generic function should satisfy. We use the type-based function generator Djinn to generate terms for specializations of the generic function types on the type indices of generic functions. Then we use QuickCheck to prune the generated terms by testing against properties, and by testing specialized candidate functions against the provided examples. Using this approach we have been able to generate generic equality, map, and zip functions, for example.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Generic Programming; I.2.2 [*Artificial Intelligence*]: Automatic Programming

General Terms Design, Experimentation

Keywords Generic programming, program synthesis, automated testing, generalized algebraic data types

1. Introduction

How do we introduce a datatype-generic function? Usually we do the following:

> Here is the instance of the function on the data type List, and here is the instance of the generic function on some kind of Tree. The pattern in these definitions should be clear. It follows that this is the type of the generic function, and the different lines in the definition of the generic function are as follows. This is the correct generic function, because if I instantiate it on the types List and Tree, I get functions with the same semantics as the functions I gave when I introduced the function.

Examples of this approach have been given by Hinze [8] for generalized folds, Gibbons [7] for accumulations, Backhouse et al. [4] for maps, and by many other authors introducing new generic functions. This approach has worked well, and both readers and students are usually convinced by the argument. However, it raises the question if there is any choice when we define a generic function after giving two or more examples on specific data types. Can we infer a generic function from examples? In general the answer to this question will be negative. Even the weaker problem of finding the type of the generic function that generalizes from the types of the examples does not have a unique solution [16]. The story doesn't end here, though. First, combining a desired type of a generic function with example instances might lead to fewer, and maybe even unique, solutions. And even though there might not be a unique solution that unifies example instances on particular data types, we might want to have a look at the different possibilities, and pick our favorite candidate from the set of all solutions. We want to *generate* all generic functions of a given type that are equal to example instances when specialized. Furthermore, we might even add properties to be satisfied by the desired generic function to further reduce the amount of possible solutions.

Why would we want to generate generic functions from examples?

- First, an algorithm that generates generic functions from examples (and possibly types and properties) is a convenient way to *specify* generic functions.
- Second, it offers (novice) users help when writing a generic function.
- Third, it formalizes the informal procedure with which we started this paper.

Recently there has been growing interest in the automatic generation of functions from user-provided specifications [14, 15, 2], mainly consisting of a type and a set of input-output examples. However, the function generation research above is not directly applicable to the generation of generic functions due to their different nature: the approach of Koopman and Plasmeijer [15] does not seem to be able to generate higher-order functions, which is essential for generating generic functions, while Djinn [2] and Katayama's MagicHaskeller [14] generate polymorphic functions but not generic functions.

In this paper, we propose a procedure for the generation of generic functions in Generic Haskell [18]. Generic Haskell is an extension of Haskell that supports the definition of datatype-generic functions.

Generic functions are generated from a user-provided specification consisting of the type of a generic function, examples of instances, and properties that the generic function should satisfy. The generation procedure proceeds as follows:

- The cases comprising a generic function are generated from the specialized generic-function type. In this paper we use the Djinn tool [2] for the type-based generation.
- The generated terms are pruned by testing each term against properties it should satisfy. We use the QuickCheck library [5] for testing.
- The set of candidate generic functions is constructed by taking the cross-product of the generated function cases.

WGP'06 September 16, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-492-6/06/0009…$5.00.

- This set is pruned by testing each candidate against examples instances. In this phase, candidate functions are instantiated using the specialization algorithm of Generic Haskell [11, 18, 16].

This paper is organized as follows. Section 2 introduces type-indexed and generic functions and briefly shows how Generic Haskell generates code for generic functions. Section 3 shows how a generic programmer typically writes a generic function in Generic Haskell. Section 4 presents the central ideas of this paper: how do we generate a generic function given its type, example instances, and properties it should satisfy. It introduces type-based function generation and shows how Djinn is used to generate generic functions from types. Section 5 explains the design choices we made for our tool and briefly explains its implementation. Section 6 reports the results of our research. Section 7 describes related work, future work, and concludes.

2. Generic functions 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 behavior that depends on this type argument. For example, suppose the unit type Unit, sum type $+$, and product type $\times$ are defined as follows:

$$\begin{aligned}
&\mathbf{data}\ \mathsf{Unit} &&= \mathit{Unit}\\
&\mathbf{data}\ \mathsf{a} + \mathsf{b} &&= \mathit{Inl}\ \mathsf{a} \mid \mathit{Inr}\ \mathsf{b}\\
&\mathbf{data}\ \mathsf{a} \times \mathsf{b} &&= \mathsf{a} \times \mathsf{b}.
\end{aligned}$$

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

$$\begin{array}{llll}
\mathit{equals}\{\!|\mathsf{Bool}|\!\} & n_1 & n_2 & = \mathit{equals}_{\mathsf{Bool}}\ n_1\ n_2\\
\mathit{equals}\{\!|\mathsf{Unit}|\!\} & \mathit{Unit} & \mathit{Unit} & = \mathit{True}\\
\mathit{equals}\{\!|\alpha + \beta|\!\} & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = \mathit{equals}\{\!|\alpha|\!\}\ x\ y\\
\mathit{equals}\{\!|\alpha + \beta|\!\} & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = \mathit{equals}\{\!|\beta|\!\}\ x\ y\\
\mathit{equals}\{\!|\alpha + \beta|\!\} & \_ & \_ & = \mathit{False}\\
\mathit{equals}\{\!|\alpha \times \beta|\!\} & (x_1 \times y_1) & (x_2 \times y_2) & = \mathit{equals}\{\!|\alpha|\!\}\ x_1\ x_2\\
 & & & \wedge\ \mathit{equals}\{\!|\beta|\!\}\ y_1\ y_2,
\end{array}$$

where $\mathit{equals}_{\mathsf{Bool}}$ is the standard equality function on booleans. The type signature of equals is as follows:

$$\mathit{equals}\{\!|\mathsf{a} :: *|\!\} :: (\mathit{equals}\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}.$$

The context $(\mathit{equals}\{\!|\mathsf{a}|\!\}) \Rightarrow$ in this signature says that equals has a *dependency* [17] on equals. 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 equals are dependency variables.

Generic functions

A type-indexed function such as equals does not only work on the types that appear as its type indices. To see why equals is in fact *generic* and works on arbitrary data types, we give a mapping from data types to view types such as units, sums, and products. It suffices to define a function on view types in order to obtain a function that can be applied to values of arbitrary data types. If there is no specific case for a type in the definition of a generic function, generic behavior is derived automatically by the compiler by exploiting the structural representation.

For example, the definition of the function equals that is generically derived for lists coincides with the following specific definition:

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

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

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

we obtain the following structural representations:

$$\begin{aligned}
&\mathbf{type}\ [\mathsf{a}]^{\circ} &&= \mathsf{Unit} + \mathsf{a} \times \mathsf{List\ a}\\
&\mathbf{type}\ \mathsf{Tree}^{\circ}\ \mathsf{a\ b} &&= \mathsf{a} + \mathsf{Tree\ a\ b} \times \mathsf{b} \times \mathsf{Tree\ a\ b},
\end{aligned}$$

where we assume that $\times$ 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 two types are isomorphic, a witness of the isomorphism, also called an embedding-projection pair, can be stored as a pair of functions converting back and forth:

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

A type T and its structural representation type T° are isomorphic, witnessed by a value $\mathit{conv}_{\mathsf{T}}\ ::\ \mathsf{EP\ T\ T}^{\circ}$. For example, for lists we have that $\mathit{conv}_{[\,]} = \mathit{Ep}\ \mathit{from}_{[\,]}\ \mathit{to}_{[\,]}$, where $\mathit{from}_{[\,]}$ and $\mathit{to}_{[\,]}$ are defined by:

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

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 structural 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°. For example, for equality we obtain a function of type $\mathsf{a}^{\circ} \to \mathsf{a}^{\circ} \to \mathsf{Bool}$. To convert this function back to a function of type $\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$ we use the function bimap [9]. Hence, if the generic function is defined for view types such as Unit, $+$, and $\times$, we do not need cases for specific data types such as List or Tree anymore. For primitive types such as Int, Float, IO or $\to$, no structure type is available. Therefore, for a generic function to work on these types, specific cases are necessary.

3. Writing a generic function

This section describes the steps a generic programmer may follow to arrive at the definition of a generic function.

Suppose that we want to construct the definition of a generic equality function. The first step is to write down examples of specific instances to understand the general pattern of equality. Consider, for example, equality for lists and trees.

$$\begin{aligned}
&equals_{[\mathsf{Int}]} :: [\mathsf{Int}] \to [\mathsf{Int}] \to \mathsf{Bool}\\
&equals_{[\mathsf{Int}]}\ [\,]\ \ [\,] = True\\
&equals_{[\mathsf{Int}]}\ (x : xs)\ (y : ys) = equals_{\mathsf{Int}}\ x\ y \wedge\\
&\qquad\qquad\qquad\qquad equals_{[\mathsf{Int}]}\ xs\ ys
\end{aligned}$$

$$\begin{aligned}
&equals_{\mathsf{Tree}} :: (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to\\
&\qquad (\mathsf{b} \to \mathsf{b} \to \mathsf{Bool}) \to\\
&\qquad \mathsf{Tree\ a\ b} \to \mathsf{Tree\ a\ b} \to \mathsf{Bool}\\
&equals_{\mathsf{Tree}}\ equals_{\mathsf{a}}\ equals_{\mathsf{b}}\ (Tip\ x)\ (Tip\ y)\\
&\quad = equals_{\mathsf{a}}\ x\ y\\
&equals_{\mathsf{Tree}}\ equals_{\mathsf{a}}\ equals_{\mathsf{b}}\ (Node\ x_1\ y_1\ z_1)\\
&\qquad\qquad\qquad\qquad (Node\ x_2\ y_2\ z_2)\\
&\quad = equals_{\mathsf{Tree}}\ x_1\ x_2\\
&\quad \wedge\ equals_{\mathsf{b}}\ \ \ y_1\ y_2\\
&\quad \wedge\ equals_{\mathsf{Tree}}\ z_1\ z_2,
\end{aligned}$$

where $equals_{\mathsf{Int}}$ is an equality function on integers. The data type Tree takes two type arguments, and therefore the equality function on Tree takes two functions as argument: the equality functions on the argument types. The types of these two examples are subsumed by the type of the generic equality function:

$$equals\{\mid \mathsf{a} :: *\mid\} :: (equals\{\mid\mathsf{a}\mid\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}.$$

It is obvious that this type is a generalization of the type of $equals_{[\mathsf{Int}]}$. In order to see that it is a generalization of the type of $equals_{\mathsf{Tree}}$ we have to instantiate this type on the data type Tree. Since $\mathsf{Tree} :: * \to * \to *$, and because $equals$ has a dependency on itself, the kind-indexed definition of type specialization [16] returns a type taking two arguments: for each argument kind the type of equality on a new type variable, here a and b, and the result type is the equality type on Tree a b.

These definitions suggest that, in general, two values are equal if their constructors are the same and if equality holds for every pair of corresponding constructor fields. It is natural to use this insight to write the definition of generic equality in Generic Haskell, taking into account the way that data types are represented by structural representations. For instance, the data type $[\,]$ is represented in Generic Haskell by the structural representation **type** $[\mathsf{a}]^\circ = \mathsf{Unit} + \mathsf{a} \times [\mathsf{a}]$. Sums represent choice between constructors, so we encode the fact that equality only holds for matching constructors as follows:

$$\begin{aligned}
&equals\{\mid\alpha + \beta\mid\}\ (Inl\ x)\ \ (Inl\ y)\ = equals\{\mid\alpha\mid\}\ x\ y\\
&equals\{\mid\alpha + \beta\mid\}\ (Inr\ x)\ (Inr\ y) = equals\{\mid\beta\mid\}\ x\ y\\
&equals\{\mid\alpha + \beta\mid\}\ \_\ \ \ \ \ \ \ \_\ \ \ \ \ \ \ = False.
\end{aligned}$$

If the values match the same alternative, the recursive call to $equals$ compares the remaining structure, namely the constructor fields or the remaining constructor choices.

As products represent the fields of a constructor, we compute the equality of the constructor fields by comparing the corresponding components of the two products:

$$\begin{aligned}
&equals\{\mid\alpha \times \beta\mid\}\ (x_1 \times y_1)\ (x_2 \times y_2) = equals\{\mid\alpha\mid\}\ x_1\ x_2\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ equals\{\mid\beta\mid\}\ y_1\ y_2.
\end{aligned}$$

The remaining case for the unit data type handles constructors with no fields and is trivial:

$$equals\{\mid\mathsf{Unit}\mid\}\ Unit\ Unit = True.$$

We emphasize that types play a prominent rôle in the definition of equality on sums and products. In both cases the arms do not have specific information about the components of sums and products. Therefore, the only possibility is to test for equality by recursively calling the generic equality function on the component types.

We summarize the informal procedure by which we obtain the generic equality function. First, we look at some instances of equality to gain insight into the general pattern of equality. Second, we obtain the type of the generic function that subsumes the types of the examples. Third, we write the definition taking into account the structure type encoding of data types and using the type information available in each case.

4. Generating generic functions

This section outlines our approach to generating generic functions, which is inspired by the informal procedure for writing generic functions described in the previous section.

To generate a generic function, a user specifies the type of the desired generic function, a set of instances for the desired function, and the properties that the function should satisfy. For example, the user might provide the type signature for generic equality given in the previous section, the instances $equals_{[\mathsf{Int}]}$ and $equals_{\mathsf{Tree}}$ of equality, and properties such as for example

$$\begin{aligned}
&\forall x\ \ \ \ \ \ .\ equals\{\mid\mathsf{t}\mid\}\ x\ x\\
&\forall x\ y\ \ \ .\ equals\{\mid\mathsf{t}\mid\}\ x\ y \Longrightarrow equals\{\mid\mathsf{t}\mid\}\ y\ x\\
&\forall x\ y\ z\ .\ equals\{\mid\mathsf{t}\mid\}\ x\ y \wedge equals\{\mid\mathsf{t}\mid\}\ y\ z \Longrightarrow\\
&\qquad\quad equals\{\mid\mathsf{t}\mid\}\ x\ z.
\end{aligned}$$

Our tool generates a set of generic functions, all of which have the specified type or a more general type. Furthermore, the instances of the generic functions on the types of the provided instances agree with the example instances on a number of test cases, and satisfy the specified properties on a number of test cases.

A generic function consists of a number of arms for type indices required by generic functions. Most generic functions in the Generic Haskell library have cases for base types such as Int, Float, Char, and Double, the sum type $+$, the product type $\times$, Unit, the types Con, and Label for representing constructors and record labels, respectively, and sometimes also $\to$, IO, etc. Not all of these cases are required: in principle, providing cases for $+$, $\times$, and Unit is sufficient if no base types are used in the data types on which a generic function is used. The Con and Label cases are generated by the Generic Haskell compiler if a user does not specify these cases.

For the purposes of this paper we will assume that the generated generic functions have arms for the types Bool (as a very simple representative of base types), $+$, $\times$, and Unit. In the final version of our tool we will generate more arms.

Since types play a prominent rôle in the definition of generic functions, types drive the generation of our functions. Our tool instantiates the type of the generic function on each of the required type indices. Then it uses Djinn to generate functions of the correct type for each of the type indices. It prunes the set of terms obtained by testing that the specified properties are satisfied by the terms for the arms generated by Djinn. The cross product of the lists of functions thus obtained gives a list of arms of generic functions.

Finally, the definitions of generic functions thus obtained are pruned by testing them against the instances provided by the user. The generic functions are specialized on the types of these instances and each of the specializations is tested for equality against the instances.

4.1 Djinn

In this paper we use Djinn for the type-based generation of terms. This tool implements a decision procedure for intuitionistic propositional calculus due to Dyckhoff [6], and, effectively exploiting the Curry-Howard correspondence, uses it to solve the type-inhabitation problem for the simply-typed lambda calculus.

To an end user, Djinn looks like the interactive environment of the Glasgow Haskell Compiler. A user can request the generation of a term by writing an identifier followed by a question mark and the type to be inhabited:

```
Djinn> id ? a -> a
id :: a -> a
id x1 = x1
Djinn> const ? a -> b -> a
const :: a -> b -> a
const x1 _ = x1
```

More complex queries can involve functional arguments to define, for example, the composition of functions, currying and uncurrying:

```
Djinn> o ? (b -> c) -> (a -> b) -> a -> c
o :: (b -> c) -> (a -> b) -> a -> c
o x1 x2 x3 = x1 (x2 x3)
Djinn> curry ? ((a, b) -> c) -> a -> b -> c
curry :: ((a, b) -> c) -> a -> b -> c
curry x1 c3 c4 = x1 (c3, c4)
Djinn> uncurry ? (a -> b -> c) -> (a, b) -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry x1 (v3, v4) = x1 v3 v4.
```

At any point the user can introduce new symbols in the environment by giving a data type declaration.

```
Djinn> data Maybe a = Nothing | Just a
Djinn> maybe ? b -> (a -> b) -> Maybe a -> b
maybe :: b -> (a -> b) -> Maybe a -> b
maybe x1 x2 x3 =
        case x3 of
        Nothing -> x1
        Just v6 -> x2 v6
```

So far we have seen only unique inhabitants of a specific given type. But this need not be the case:

```
Djinn> choose ? a -> a -> a
choose :: a -> a -> a
choose _ x2 = x2
-- or
choose x1 _ = x1.
```

Djinn can generate a number of terms for a type, but this does not mean that all possible terms are generated. We will discuss limitations of Djinn in a later subsection.

4.2 Type-based term generation

Given an environment Γ and a type t, a type-based term-generation tool such as Djinn generates a set of terms E, such that every term e in E has type t: $\Gamma \vdash e :: \mathsf{t}$. The size of this set is potentially infinite, thus tools that generate such definitions restrict either the maximal size of such a set or the maximal size of generated terms.

Using a type-based term-generation tool, we can exploit the fact that we possess the signature of a generic function to generate definitions for the different view types. As an example, we show various well-typed terms for the sum case of the generic equality function. The instance type is obtained by specializing the generic type signature. Due to the dependency of *equals* on *equals*, the instance type has functional arguments for computing the equality of the sum components:

$$
\begin{aligned}
\mathit{equals}\{\!|\alpha + \beta|\!\} :: \forall \mathsf{a}\ \mathsf{b}\,.\, & (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \\
& (\mathsf{b} \to \mathsf{b} \to \mathsf{Bool}) \to \\
& (\mathsf{a} + \mathsf{b}) \to (\mathsf{a} + \mathsf{b}) \to \mathsf{Bool}.
\end{aligned}
$$

There are infinitely many well-typed terms that have this type; we show a few of them:

$$
\begin{aligned}
&\lambda f\ g\ x\ y\ \ \to \mathit{True}\\
&\lambda f\ g\ x\ y\ \ \to \mathit{False}\\
&\lambda f\ g\ s_1\ s_2 \to \textbf{case}\ s_1\ \textbf{of}\\
&\qquad\qquad \mathit{Inl}\ x \to \textbf{case}\ s_2\ \textbf{of}\\
&\qquad\qquad\qquad \mathit{Inl}\ y \to f\ x\ y\\
&\qquad\qquad\qquad \mathit{Inr}\ y \to \mathit{False}\\
&\qquad\qquad \mathit{Inr}\ x \to \textbf{case}\ s_2\ \textbf{of}\\
&\qquad\qquad\qquad \mathit{Inl}\ y \to \mathit{False}\\
&\qquad\qquad\qquad \mathit{Inr}\ y \to g\ x\ y\\
&\lambda f\ g\ s_1\ s_2 \to \textbf{case}\ s_1\ \textbf{of}\\
&\qquad\qquad \mathit{Inl}\ x \to \textbf{case}\ s_2\ \textbf{of}\\
&\qquad\qquad\qquad \mathit{Inl}\ y \to \mathit{False}\\
&\qquad\qquad\qquad \mathit{Inr}\ y \to g\ y\ y\\
&\qquad\qquad \mathit{Inr}\ x \to \textbf{case}\ s_2\ \textbf{of}\\
&\qquad\qquad\qquad \mathit{Inl}\ y \to f\ y\ y\\
&\qquad\qquad\qquad \mathit{Inr}\ y \to \mathit{True}.
\end{aligned}
$$

Interestingly, the more generic the type of a (generic) function, the fewer terms are generated by a type-based term-generation tool. As an example, there exist more terms of type $\mathsf{a} \to \mathsf{a}$ than of type $\mathsf{a} \to \mathsf{b}$.

For instance, consider the generic map function *map* with signature

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

From the generic signature we generate the following type for the data type $+$

$$
\begin{aligned}
\mathit{gmap}\{\!|\alpha + \beta|\!\} :: \forall \mathsf{a}\ \mathsf{b}\ \mathsf{c}\ \mathsf{d}\,.\, & (\mathsf{a} \to \mathsf{c}) \to \\
& (\mathsf{b} \to \mathsf{d}) \to \\
& \mathsf{a} + \mathsf{b} \to \mathsf{c} + \mathsf{d}.
\end{aligned}
$$

Djinn generates the following term:

$$
\begin{aligned}
&\mathit{gmap}_{+}\ f\ g\ s = \textbf{case}\ s\ \textbf{of}\\
&\quad \mathit{Inl}\ x \to \mathit{Inl}\ (f\ x)\\
&\quad \mathit{Inr}\ x \to \mathit{Inr}\ (g\ x).
\end{aligned}
$$

Of course this is not the only term of this type: we can insert an arbitrary number of functions that have no effect on the type, such as for example the identity function, or sum swapping functions. Djinn does not generate such terms containing 'useless' occurrences of the identity or swapping functions. The smallest term is the term we expect. The same holds for the map term on the product type.

4.3 Limitations of Djinn

Recursive data types

In Section 4.1 we have shown that Djinn can also generate functions for types involving user-defined data types such as $\mathsf{Maybe\ a}$. However, Djinn does not support recursive data types such as lists or trees. While this restricts the family of functions that can be generated, this is not a big problem for our approach, since none of our view types is recursive. Of course, the generated generic functions can be specialized on a recursive data type without problem using the specialization algorithm of Generic Haskell. If our set of view types would include one or more recursive data types, which would be the case if we would use the view types for the fixed-point view on data types [12], which includes the data type Fix, this would be a problem. Furthermore, this Djinn restriction also rules out generating generic functions with recursive types in their signatures. For example, since function *enum* returns a *list* of values:

$$\mathit{enum}\{\!|\mathsf{a} :: *|\!\} :: (\mathit{enum}\{\!|\mathsf{a}|\!\}) \Rightarrow [\mathsf{a}],$$

we cannot use Djinn to generate it.

Finite number of terms

In order to ensure termination for every query, Djinn implements a sequent calculus that does not have a contraction rule. This design allows Djinn to answer the type-inhabitation problem in finite time. The price to be paid is that not all terms inhabiting a type are generated. Indeed, if a type is inhabited at all, only finitely many terms can be generated by the algorithm implemented in Djinn. For instance, the type below corresponds to Church numerals.

```
Djinn> num ? (a -> a) -> a -> a
num :: (a -> a) -> a -> a
num x1 = x1
-- or
num x1 x2 = x1 x2
-- or
num _ x2 = x2
```

While the set of Church numerals is infinite, Djinn only generates the terms corresponding to zero and one. For this reason, we have refrained from trying to generate the Int cases of generic functions.

4.4 Generalizing the type of a generic function

The set of terms generated by Djinn for an argument type may be very large. The size of this set is the limiting factor in our approach to generating generic functions. We apply two steps in our approach to restrain the size of the set of generated terms. The first step we apply is using the observation made in the previous subsection: the more generic the type of a function, the fewer terms are generated. It is easy to see why this is the case for the following example. Given an a it is easy to construct an a by means of the identity function, so asked for constructing a term of type $\mathsf{a} \to \mathsf{a}$, Djinn returns one term: the identity function. It is impossible to construct a value of type b out of a value a, so the set of values of type $\mathsf{a} \to \mathsf{b}$ returned by Djinn is empty. Generalizing a type where occurrences of the same variable are replaced by different variables leads to fewer terms generated by Djinn, since there are fewer input terms to choose from for the output.

Before we let Djinn generate terms for the types obtained by instantiating the generic type on the view types, we first generalize the type of the generic function. For example, while the signature we gave in Section 2 for equality is

$$\mathit{equals}\{\!|\mathsf{a} :: *|\!\} :: (\mathit{equals}\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool},$$

we replace it now by the more general

$$\mathit{equals}\{\!|\mathsf{a}, \mathsf{b} :: *|\!\} :: (\mathit{equals}\{\!|\mathsf{a}, \mathsf{b}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{b} \to \mathsf{Bool},$$

that is, we replace occurrences of the same variable in the type by different variables. Now, the type for the $+$ case of equality is

$$\begin{aligned}\mathit{equals}\{\!|\alpha + \beta|\!\} :: \forall \mathsf{a}\ \mathsf{b}\ \mathsf{c}\ \mathsf{d}\,.\, &(\mathsf{a} \to \mathsf{c} \to \mathsf{Bool}) \to \\ &(\mathsf{b} \to \mathsf{d} \to \mathsf{Bool}) \to \\ &(\mathsf{a} + \mathsf{b}) \to (\mathsf{c} + \mathsf{d}) \to \mathsf{Bool}.\end{aligned}$$

When we feed this type to Djinn, it generates a non-empty, and much smaller set of terms then for the original type of $\mathit{equals}\{\!|\alpha + \beta|\!\}$.

Generalizing the type of a generic function has the added advantage that the function becomes more flexible. For example, the generalized equality can also compare list of characters with list of integers (presumably representing entries in the ASCII table) by supplying the function ord. This type of generalization is very common in generic programming, for example, the type of `pequal` in PolyLib [13], the library of PolyP, is $(\mathsf{a} \to \mathsf{b} \to \mathsf{Bool}) \to \mathsf{d}\ \mathsf{a} \to \mathsf{d}\ \mathsf{b} \to \mathsf{Bool}$.

The type of a generic function can always safely be generalized: the generic function obtained from the code generated by Djinn with a more general type is always correct when typed with a less general type.

We leave the generalization step to the user of our tool. However, automating this step is very simple, so we might add the option to a future version of our tool.

4.5 Pruning generated terms by property

The second step in decreasing the number of terms generated by Djinn is to prune the set of terms obtained from Djinn by means of the properties specified for a generic function.

For example, we want the equality function to be reflexive, symmetric, and transitive. Since it is in general impossible to automatically *prove* properties of functions, we will *test* these properties instead. So for each of the four type indices, the functions generated by Djinn are tested whether or not they satisfy the instance of the properties of equality on the type. For the symmetry property on the Bool type we test the following property:

$$\forall x\ y\,.\ \mathit{equals}\{\!|\mathsf{Bool}|\!\}\ x\ y == \mathit{equals}\{\!|\mathsf{Bool}|\!\}\ y\ x.$$

This is a simple instance of the general symmetry property for equality. For the sum type, which has kind $* \to * \to *$, the instance of the property is a bit more involved:

$$\begin{aligned}&\forall \mathit{eqa}\ x\ y\,.\ \mathit{eqa}\ x\ y == \mathit{eqa}\ y\ x \Longrightarrow\\ &\quad\forall \mathit{eqb}\ v\ w\,.\ \mathit{eqb}\ v\ w == \mathit{eqb}\ w\ v \Longrightarrow\\ &\qquad\forall s\ t\,.\ \mathit{equals}\{\!|\mathsf{a} + \mathsf{b}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ s\ t ==\\ &\qquad\qquad \mathit{equals}\{\!|\mathsf{a} + \mathsf{b}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ t\ s,\end{aligned}$$

where the dependency of $\mathit{equals}\{\!|\mathsf{a} + \mathsf{b}|\!\}$ on equals has been made explicit.

As it happens, the four equality functions on the sum type generated by Djinn given in Section 4.2 satisfy the above property. A generated term that does not satisfy it is:

$$\begin{array}{l}\lambda x\ y\ z \to \mathbf{case}\ z\ \mathbf{of}\\ \quad \mathit{Inl}\ v\ \ \to \lambda s \to \mathbf{case}\ s\ \mathbf{of}\\ \qquad\qquad\qquad\qquad \mathit{Inl}\ m \to x\ v\ m\\ \qquad\qquad\qquad\qquad \mathit{Inr}\ \_\ \to \mathit{False}\\ \quad \mathit{Inr}\ w \to \lambda t \to \mathbf{case}\ t\ \mathbf{of}\\ \qquad\qquad\qquad\qquad \mathit{Inl}\ \_\ \to \mathit{True}\\ \qquad\qquad\qquad\qquad \mathit{Inr}\ n \to y\ w\ n.\end{array}$$

Just as with types of type-indexed functions, properties on type-indexed functions depend on the kind of the type arguments [9]. It suffices to specify the property on types of kind $*$, such as the property for $\mathit{equals}\{\!|\mathsf{Bool}|\!\}$ above; the properties for types with higher-order kinds can then automatically be generated.

At the moment the instances of properties on particular types have to be written by hand; we expect that in the near future we will add kind-indexed property generation to our tool.

4.6 QuickCheck

We use the QuickCheck library [5] for testing whether or not properties hold, and for testing equality of example instances provided by the user and specialized instances obtained from the generated generic functions. QuickCheck tries to falsify properties specified by the user. It typically generates 100 random test cases, which are used to test a property. One of the properties we define is the following:

$$\begin{aligned}&\mathit{prop\_EqFuns}\ \mathit{specializedFun}\ \mathit{exampleFun}\\ &\quad = \lambda x \to \mathit{specializedFun}\ x == \mathit{exampleFun}\ x.\end{aligned}$$

This property is parametrized with the specialized generic function and the user-specified example function. It tests the equality of the two functions for an arbitrary argument. Note that this property applies to functions of one argument; functions with more arguments would take additional x's. A property like this, properly instantiated with the example and the generic function, is passed to the *run* function in QuickCheck, and QuickCheck then randomly generates several values of x to test the validity of the property.

Not all properties are easy to check using QuickCheck. Since QuickCheck randomly generates values, it is rather unlikely that it generates equal values if we test on a data type that contains many values. So the transitivity property of equality, which takes three random values x, y, and, z, and checks that x and z are equal whenever x and y are equal and y and z are equal, is unlikely to ever check that x and z are equal. Special measures have to be taken in such cases, such as for example restricting the type of values to small types, or to write special generators.

5. Implementation

This section discusses an implementation of our tool, and the design choices we made to implement it.

As outlined in Section 4, our approach consists of a generation phase, implemented by Djinn, and a couple of pruning phases. The terms generated for the type indices of generic functions are pruned by checking their validity against properties. The set of generic functions obtained by taking the cross product of the type-indexed terms, is pruned by checking the equivalence of specializations with respect to user-provided instances. Since, in general, there is no algorithm that can prove the validity of a term with respect to a property or an instance, we turn to type-based automatic testing to test properties and the equivalence with user-provided examples.

Besides the functionality offered by Djinn, our approach requires specialization and testing. A very straightforward approach would be to pretty print the terms generated by Djinn to Generic Haskell source files, compile these files using the Generic Haskell compiler, and then run the Glasgow Haskell Compiler to produce a binary that tests and reports which properties pass or fail for which generated functions. The advantage of this approach is that we can use the specialization algorithm from Generic Haskell for specializing generic functions and signatures, and the automated testing library QuickCheck. This approach, however, is likely to incur significant overhead due to the generation and compilation of code for *every* candidate function. This problem becomes more serious if we take into account that we may generate and prune thousands of expressions for a given generic-function signature.

An alternative approach is to write an expression interpreter integrated with Djinn. We then would have to write specialization functionality and QuickCheck functionality for this expression language. Although the interpreter approach does not suffer from compiler overheads, it is a considerable task to implement (subsets of) the languages and libraries of Generic Haskell, Haskell, and QuickCheck.

In this paper, we use a more sophisticated variant of the interpreter approach. Instead of directly interpreting the terms generated by Djinn, we first transform terms into well-typed terms. In this way it becomes possible to implement an efficient interpreter that does away with the tagging and untagging that an untyped interpreter needs. Generalized algebraic data types [21, 20] (GADTs) play an essential rôle here. Moreover, since the interpreted values are Haskell values, it is possible to use the QuickCheck library. In the rest of this section we describe our typed interpreter and how it integrates with automated testing using QuickCheck and with the specialization of generic functions.

5.1 Expressions and types

Although the generation algorithm of Djinn is type based, which guarantees that generated terms are well-typed expressions, the type information is not preserved in the terms. So the terms we obtain from Djinn are untyped. To use these terms in Haskell, we have to evaluate them into untagged Haskell values. These Haskell values can then directly be tested against properties and instances using QuickCheck.

As usual, expressions consist of variables, lambda abstractions, applications and constants.

```
data Expr = EVar String
          | ELam String Expr
          | EApp Expr Expr
          | ECon String
```

As an example, here is how we write the *swap* function as a value of type Expr:

```
swap = ELam "x"
         (EApp (EApp (ECon "(,)")
                     (EApp (ECon "snd")
                           (EVar "x"))
               (ECon "fst" `EApp` EVar "x")).
```

Types are either variables, constants, or the application of a type to a type.

```
data Type = TVar String
          | TCon String
          | TApp Type Type
```

We do not extend the syntax of expressions to cover generic functions as well, but instead represent generic functions by labeled records that contain the arms for the different type indices Bool, Unit, $\times$, and $+$ for generic functions.

```
data GenDefinition =
       GenDefinition{ bool :: Expr
                    , unit :: Expr
                    , prod :: Expr
                    , sum  :: Expr
                    }
```

A signature of a generic function consists of a name, a list of generic type variables (before type generalization the equality function has one generic type variable, *gmap* has two), and a type.

```
data GenSignature =
       GenSignature{ genName :: String
                   , genVars :: [String]
                   , genType :: Type
                   }
```

We assume generic functions depend on themselves and not on other generic functions. The first assumption does not restrict the domain: very few generic functions are not self dependent [16], and dependencies do not have to be used. The second assumption does restrict the applicability of the tool. However, allowing dependencies on other generic functions is not difficult.

A generic function can be specialized on a type. We have functions for specializing generic signatures and functions:

```
specGenSignature  :: GenSignature →
                     Type → Type
specGenDefinition :: GenSignature →
                     GenDefinition →
                     Type → Expr.
```

The definitions of these functions are based upon the standard specialization algorithms for Generic Haskell, which have been described in several places [10, 11, 16], and are therefore omitted from this paper.

5.2 Typing untyped terms

To turn Djinn-generated terms into untagged Haskell values we first have to type them. This section briefly explains how we type the untyped terms generated by Djinn. Our approach is reminiscent of the dynamic typing solution given by Baars and Swierstra [3] for a similar problem. The main difference is that we use De Bruijn indices for variables, making it impossible to fail when looking up a variable.

Typed terms are defined as a GADT to enforce the constraints that make terms well-typed.

```
data Zero
data Succ a

data Lookup :: ∗ → ∗ → ∗ → ∗ where
  ZL :: Lookup Zero (a, env) a
  SL :: Lookup i env a → Lookup (Succ i) (x, env) a

data TExpr :: ∗ → ∗ → ∗ where
  TApp     :: TExpr env (a → b) →
              TExpr env a →
              TExpr env b
  TVar     :: Lookup i env a → TExpr env a
  TLambda :: TExpr (a, env) b → TExpr env (a → b)

evaluate :: TExpr () a → a
```

This presentation is heavily inspired by the representation of typed terms in dependently typed programming languages [19]. The TExpr data type represents a valid typing judgement that follows the structure of a typed term. The first argument is the environment under which the judgement holds and the second argument is the type assigned by the judgement to the term. Variables are represented using De Bruijn indices. The *TVar* constructor selects a type in the environment using the data type Lookup. If a type expression has an empty environment we can obtain the represented type by means of the function *evaluate*.

The function *typeInfer* recovers the type information for a term generated by Djinn.

```
typeInfer :: Expr → Maybe Typed

data Typed :: ∗ where
  Typed :: TExpr () a → Rep a → Typed

data Rep :: ∗ → ∗ where
  RU     :: Rep ()
  RBool :: Rep Bool
  RProd :: Rep a → Rep b → Rep (a, b)
  RSum  :: Rep a → Rep b → Rep (Either a b)
  RArr  :: Rep a → Rep b → Rep (a → b)
```

If function *typeInfer* manages to infer a type, it returns a value of type Typed. A Typed value contains an existentially quantified type and its representation. It is possible to recover this type by analyzing the type-representation witness. Since the type of the expression is existential, it is sometimes necessary to compare it with another type representation to obtain a type-equality proof:

```
unify :: Rep a → Rep b → Maybe (TEq a b)

data TEq :: ∗ → ∗ → ∗ where
  TEq :: TEq a a.
```

We omit the definitions of *evaluate*, *unify* and *typeInfer*. The interested reader can find the ideas behind these definitions in the literature [3, 21, 20].

5.3 Term generation

The generation function has the following, simple, type:

```
generate :: Type → [Expr].
```

Our implementation of *generate* uses Djinn, but we could have used any of the approaches described in [14, 15].

Type-based generation yields a set of functions for each of the type indices that make up a generic function. Hence, to obtain the generic function we combine all generated terms in a cross product, using a list comprehension.

```
generateGenDefinition :: GenSignature →
                          [GenDefinition]
generateGenDefinition sig =
  [GenDefinition    {bool = boolDef
                    , unit = unitDef
                    , prod = prodDef
                    , sum  = sumDef
                    }
  | boolDef ← boolTerms
  , unitDef ← unitTerms
  , prodDef ← prodTerms
  , sumDef ← sumTerms
  ]
  where
    boolTerms = genTypeCase tBool
    unitTerms = genTypeCase tUnit
    prodTerms = genTypeCase tProd
    sumTerms = genTypeCase tSum
    genTypeCase = generate . specGenSignature sig,
```

where *tBool*, *tUnit*, *tProd*, and *tSum* are values of type Type representing the types Bool, Unit, $\times$, and $+$, respectively.

Since Djinn sometimes generates many terms, it is important to first prune the terms generated by *genTypeCase* by property. The implementation of pruning is discussed in the following sections.

Note that if the generator returns many terms for the type indices, huge structures are built here. The cross-product function is the bottleneck of our approach.

5.4 Testing properties of functions

To prune the set of candidate functions, we test generated terms against user-provided properties, and specializations of generated generic functions against user-provided example instances. We use QuickCheck for testing, so we have to construct QuickCheck properties for pruning.

```
data Prop = ∀a b . Testable b ⇒ Prop (Rep a) (a → b)
```

A property consists of a representation of the type a, and a function that takes a value of type a and returns a value which, when tested with QuickCheck, determines whether or not the original value of type a is valid. In general, a property takes the form of a function that applies the value to be checked to test cases generated by QuickCheck. The type of the property (b) is an instance of the type class Testable to enable the automatic generation of test cases. As an example, consider the reflexivity property for the equality function on booleans and products of booleans:

```
reflEqBool, reflEqProdBool :: Prop
reflEqBool         = reflEq RBool
reflEqProdBool = reflEq (RProd RBool RBool)

reflEq    :: (Testable (a → Bool)) ⇒ Rep a → Prop
reflEq r = Prop (sigEq r) (λf x → f x x == True)

sigEq    :: Rep a → Rep (a → a → Bool)
sigEq t = t `RArr` (t `RArr` RBool).
```

The function that represents the reflexivity property, applied to equality on booleans, can be tested with QuickCheck because its type, [Bool] → Bool, is an instance of type class Testable.

The testing function tests an untyped expression against a user-specified property:

$$testProp :: \mathsf{Prop} \to \mathsf{Expr} \to \mathsf{Bool}.$$

This function converts an expression to a Haskell value (Section 5.2) and, using QuickCheck, tests its validity with respect to the property. For example, the set of terms for booleans in the cross product is pruned as follows:

```
boolTerms = filter (testProp reflEqBool)
                   (genTypeCase tBool).
```

For type constructors the situation is a bit different. As explained in Section 4.5, a property for sums or products takes properties for its dependencies as argument. As a pragmatic solution we let the property depend on the first term for booleans that satisfies the property.

```
prodTerms   = filter pruneProd (genTypeCase tProd)
pruneProd e = testProp reflEqProdBool (appDep e)
appDep e    = e `EApp` dep `EApp` dep
dep         = head boolTerms
```

Note that the arms of the generic function are pruned using properties *before* they are combined in the cross product. As a result, many unnecessary combinations that would otherwise be produced are no longer considered.

5.5 Pruning by example

Pruning by example compares the specialization of a candidate generic function on a data type for which we also have an example instance. For example, we might have equality on lists of integers as an example instance. If a counterexample, i.e. an argument for which the two functions return different results, is found, the candidate generic-function definition is discarded.

Each example contains a type index and a property that tests the function with an example:

```
data Example = Example Type Prop.
```

The type index requests a specialization of the generic function to be compared against the example function.

Here are some typical examples for the generic equality function:

```
[Example tBool
          (Prop (sigEq RBool)
                (λf x y → f x y == (x == y)))
,Example (tProd tBool tBool)
          (Prop (sigEq (RProd RBool RBool))
                (λf x y → f x y == (x == y)))
].
```

Given a generic signature, a generic function and an example we can determine the equivalence of the specialization of the generic function and the example by means of testing:

```
testGenDefinition ::
  GenSignature → [Example] → GenDefinition → Bool
testGenDefinition genSig examples genDef =
  and (map check examples)
  where check (Example tindex prop) =
          testProp prop (specGenDefinition
                           genSig
                           genDef
                           tyindex).
```

The implementation of pruning for a list of generic functions involves filtering the list using *testGenDefinition* as predicate, just as for properties. Pruning by example, as the name suggests, prunes away the generic functions that are not extensionally equal to all the examples provided by the user.

```
pruneByExamples :: GenSignature →
                   [Example] →
                   [GenDefinition] →
                   [GenDefinition]
pruneByExamples genSig examples
  = filter (testGenDefinition genSig examples)
```

5.6 Termination of testing

In our approach we test terms generated by Djinn against properties, and we test specialized generic functions against provided examples. If such a test does not terminate, neither will our tool.

Testing terms generated by Djinn is not a problem, since Djinn only generates non-recursive functions, and termination is guaranteed.

Specializations generated for recursive types are recursive functions themselves, so here there is a possibility that a non-terminating specialization is generated, and that a test for such a function might not terminate.

Our tool does terminate for the subclass of generic functions consisting of so-called consuming (*equals*) and transforming (*gmap*, *zipWith*) generic functions. This subclass of generic functions is terminating because the recursive calls in the product and sum cases use the self-dependency functions, which can only take a component of the original product or sum as argument. So $equals\{\!|\alpha + \beta|\!\}$ can only be defined in terms of $equals\{\!|\alpha|\!\}$ and $equals\{\!|\beta|\!\}$. This ensures that at every recursive call the type argument becomes smaller until a base case is reached (Unit or Bool) and the function terminates. Generating producing generic functions such as *empty* and *enum* [16] may result in non-terminating generic functions. In future work we intend to generate this subclass of generic functions by imposing a timeout when performing tests.

6. Results

Experimentation with our tool has produced encouraging results with the generation of generic functions. We have generated generic equality, generic map, and generic zip. In this section we give an account of the attempts for each function.

To handle the generation of the generic equality function, we have generalized its signature to

$$equals\{\!|\mathsf{a},\mathsf{b} :: *|\!\} :: (equals\{\!|\mathsf{a},\mathsf{b}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{b} \to \mathsf{Bool},$$

as explained in Section 4.4. Furthermore, we supply the properties that specify that equality is reflexive, symmetric, and transitive, and we provide the instances of equality on lists of booleans and trees. Our tool generates the standard implementation of generic equality (a number of times in alpha-equivalent versions).

We have experimented a bit with taking subsets of the specification, all of which include the generalized type of equality.

First, only providing the examples is not always enough. For example, if we supply as example only equality on lists of booleans we also get a generic function with incorrect behavior for the sum case. The function returns *True* for two left components independent of whatever is in those components. This is because the left component of the sum in the structure type for lists is Unit, and the equality function for Unit always returns *True*. We had to wait for quite a while to obtain this definition: specifying properties not only helps in getting the right functions, we also get them (much!) faster.

Second, providing no examples, but only the properties, is not enough either. One of the functions generated for products only compares the first components of the products, and ignores the second components. However, the more properties are specified, the fewer solutions we get.

By far the easiest function to generate has been the generic map function. The highly polymorphic type of this function gives very little freedom to Djinn to generate terms. In fact, each of the following type queries in Djinn produces a unique correct answer:

$$\begin{array}{ll} gmap_{+} & ?\ (\mathsf{a} \to \mathsf{c}) \to (\mathsf{b} \to \mathsf{d}) \to \mathsf{a} + \mathsf{b} \to \mathsf{c} + \mathsf{d} \\ gmap_{\times} & ?\ (\mathsf{a} \to \mathsf{c}) \to (\mathsf{b} \to \mathsf{d}) \to \mathsf{a} \times \mathsf{b} \to \mathsf{c} \times \mathsf{d} \\ gmap_{\mathsf{Unit}} & ?\ \mathsf{Unit} \to \mathsf{Unit} \\ gmap_{\mathsf{Bool}} & ?\ \mathsf{Bool} \to \mathsf{Bool}. \end{array}$$

The only exception to this rule is the case for booleans, whose type leaves some freedom for term generation. The first solution returned by Djinn is the expected identity function. We enforce this choice by providing the property that map is the identity function on types of kind $*$ (such as Bool).

When trying to generate the generic *zipWith* function, a generalization of the standard *zipWith* function with signature

$$zipWith\{\!|\mathsf{a},\mathsf{b},\mathsf{c} :: *|\!\} :: (zipWith\{\!|\mathsf{a},\mathsf{b},\mathsf{c}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{b} \to \mathsf{c},$$

we encountered a problem with the sum case. Consider the sum case definition of *zipWith*, as given in Löh's thesis [16]

$$\begin{array}{l} zipWith\{\!|\alpha + \beta|\!\}\ (Inl\ x)\ \ (Inl\ y)\ = \\ \quad Inl\ (zipWith\{\!|\alpha|\!\}\ x\ y) \\ zipWith\{\!|\alpha + \beta|\!\}\ (Inr\ x)\ (Inr\ y) = \\ \quad Inr\ (zipWith\{\!|\beta|\!\}\ x\ y) \\ zipWith\{\!|\alpha + \beta|\!\}\ \_ \qquad\qquad\quad = undefined. \end{array}$$

Using the analogy with lists, we can say that different alternatives correspond to lists of different lengths. This implementation, unlike the Haskell ones which returns an empty list, fails when structures of different shapes are zipped. To generate the sum case, Djinn needs to find a term of the following type:

$$\begin{array}{l} (\mathsf{a} \to \mathsf{c} \to \mathsf{e}) \to \\ (\mathsf{b} \to \mathsf{d} \to \mathsf{f}) \to \\ \mathsf{a} + \mathsf{b} \to \mathsf{c} + \mathsf{d} \to \mathsf{e} + \mathsf{f}. \end{array}$$

Djinn cannot populate this type because it cannot find a proof of the corresponding logical expression. Which is not surprising, because there does not exist such a proof. Djinn only generates total functions and the *zipWith* case for sums cannot be total. We could get around this limitation by adopting Haskell's solution and using the *empty* function [16] to generate a value corresponding to the empty list. However, we prefer to follow a simpler approach and change the type of *zipWith* so that it can cope with failure without depending on other functions.

$$\begin{array}{l} zipWith\{\!|\mathsf{a},\mathsf{b},\mathsf{c} :: *|\!\} :: (zipWith\{\!|\mathsf{a},\mathsf{b},\mathsf{c}|\!\}) \Rightarrow \\ \qquad\qquad\qquad\qquad\quad \mathsf{a} \to \mathsf{b} \to \mathsf{Maybe}\ \mathsf{c} \end{array}$$

This is a variant of the type of *zipWith* used in PolyLib. With this type, and the examples of *zipWith* on lists and booleans, Djinn generates the correct definition of *zipWith*. In particular, it is not hard to find a term of type:

$$\begin{array}{l} (\mathsf{a} \to \mathsf{c} \to \mathsf{Maybe}\ \mathsf{e}) \to \\ (\mathsf{b} \to \mathsf{d} \to \mathsf{Maybe}\ \mathsf{f}) \to \\ \mathsf{a} + \mathsf{b} \to \mathsf{c} + \mathsf{d} \to \mathsf{Maybe}\ (\mathsf{e} + \mathsf{f}). \end{array}$$

Alternatively, the property

$$\begin{array}{l} x \equiv y \wedge zipWith\ x\ y \equiv Just\ x\ \vee \\ x \not\equiv y \wedge zipWith\ x\ y \equiv Nothing \end{array}$$

suffices to find this definition of *zipWith*.

We think that *zipWith* in the library of Generic Haskell should be replaced by this version.

7. Conclusions

We have presented an approach to the automatic generation of generic functions. Our approach uses Djinn to generate arms of a generic function for instances of the user-specified generic signature of the desired function on the type indices (or view types) of generic functions. A safe and useful step a user almost always should take here is generalizing the type of the desired generic function: the more general a type, the fewer terms are generated by Djinn. We prune the set of terms returned by Djinn by testing them against user-specified properties using QuickCheck. We further reduce the set of generic functions by testing specializations of generated generic functions against user-provided examples. We have shown that type-based generation of a number of generic functions is realistic. Our approach has a number of limitations, which are listed in the future work section below.

7.1 Related work

Koopman and Plasmeijer [15] use a type-based systematic enumeration of terms for the problem of function generation. Function terms are represented in a system of data types such that all the values that are generated are well typed and follow syntactic restrictions that prevent the generation of non-terminating functions. Candidate functions are tested against input-output pairs until a function satisfying those tests is found. Enforcing well-typedness in the syntax of expressions gives a simple and efficient approach for function generation. However, it is not clear how to extend this approach to generate functions that take functions as arguments, such as the sum and product cases of generic functions. We believe that our typed term representation makes it possible to use the enumeration approach with higher-order functions.

Katayama [14] provides another approach to the systematic generation of lambda expressions. The enumeration takes into account type information when doing a breadth-first search of expressions. Recursion is only possible by means of paramorphism operators on lists and natural numbers. Because this is a general approach, we believe that the performance improvement techniques can also be applied to our tool, should we choose to pursue the systematic enumeration direction.

7.2 Future work

The current tool is a proof of concept, with which we have shown that it is possible to automatically generate generic functions from their type and example instances. We simplified our domain in a number of places: generic functions are self dependent, and cannot have dependencies on other generic functions. These restrictions are easy to lift. The fact that Djinn generates a finite number of terms for a type implies that it will be hard to generate Int or String cases using Djinn. Furthermore, since Djinn cannot handle recursive types, we cannot handle generic functions with recursive types in their signature. Lifting these restrictions requires adapting Djinn in a fundamental way. We will investigate to what extent this is possible. Since QuickCheck can only check monomorphic properties, example instances of generic functions have to provided for types of kind $*$. The type-specialization and function-specialization algorithms in our tool have only been implemented for types of kind $*$.

We intend to investigate using the systematic enumeration of typed terms as in the work of Koopman and others [15], instead of, or besides, Djinn.

Our approach uses type-based generation, and example-based testing. It would be interesting to see if example-based generation with type-based testing would work equally well. We have experimented with example-based generation, in which we try to generate arms in the definition of a generic function from example instances. Our *ad-hoc* attempts were not very successful. We want to investigate whether or not it is possible to "invert" the fusion algorithm from Alimarine et al. [1], which fuses embedding-projection pairs with type-indexed functions to obtain code that is almost equal to hand-written example instances of generic functions, to obtain arms in the definition of a generic function.

To get access to constructor names, structure types in Generic Haskell contain occurrences of the Con type:

$$\mathbf{data}\ \mathsf{Con}\ \mathsf{a} = \mathit{Con}\ \mathsf{a}$$

The compiler tags occurrences of the Con type with information about the constructor, such as its name and arity. In a generic function we get access to this information in the Con case, for example (from the library of Generic Haskell):

$$\mathit{constructorOf}\,\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ a) = c$$

Of course, Djinn cannot generate functions based on constructor information. So generic functions that depend on constructor information are out of reach for the approach described in this paper. The example-based generation type-based testing approach might not suffer from this problem.

Acknowledgements
Andres Löh discussed this question with us years ago. Kasper Brink and Remco Burema contributed to our discussions when we started the research reported in this paper. Remco Burema and Stefan Holdermans commented on previous versions of this paper. Anonymous referees provided useful feedback. Thanks are due to Lennart Augustsson for implementing Djinn, on which our work is fundamentally based.

References

[1] Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC'04*, volume 3125 of *LNCS*, pages 16–31. Springer-Verlag, 2004.

[2] Lennart Augustsson. Announcing Djinn, version 2004-12-11, a coding wizard. Available from `http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747`, 2005.

[3] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the 7th ACM SIGPLAN international conference on Functional programming, ICFP'02*, pages 157–166. ACM Press, 2002.

[4] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.

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

[6] R Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3):795–807, 1992.

[7] Jeremy Gibbons. Polytypic downwards accumulations. In Johan Jeuring, editor, *Proceedings of the 4th International Conference on Mathematics of Program Construction, MPC'98*, volume 1422 of *LNCS*, pages 207–233. Springer-Verlag, 1998.

[8] Ralf Hinze. Efficient generalized folds. In Johan Jeuring, editor, *Proceedings Workshop on Generic Programming, WGP 2000*, pages 1–16, 2000. Utrecht Technical Report UU-CS-2000-19.

[9] Ralf Hinze. *Generic Programs and Proofs*. 2000. Habilitationsschrift, Bonn University.

[10] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and José Nuno Fonseca de Oliveira, editors, *Proceedings of the 5th International Conference on Mathematics of Program Construction, MPC'00*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.

[11] Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.

[12] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *Proceedings 8th International Conference on Mathematics of Program Construction, MPC'06*, volume 4014 of *LNCS*. Springer-Verlag, 2006.

[13] P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, 1998.

[14] Susumu Katayama. Systematic search for lambda expressions. In M. van Eekelen, editor, *6th Symposium on Trends in Functional Programming, TFP 2005*. Institute of Cybernetics, Tallinn, 2005.

[15] Pieter Koopman and Rinus Plasmeijer. Systematic synthesis of functions. In Henrik Nilsson, editor, *7th Symposium on Trends in Functional Programming, TFP 2006*, 2006.

[16] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, September 2004.

[17] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Olin Shivers, editor, *Proceedings of the International Conference on Functional Programming, ICFP'03*, pages 141–152. ACM Press, August 2003.

[18] Andres Löh and Johan Jeuring (editors). The Generic Haskell user's guide, Version 1.42 - Coral release. Technical Report UU-CS-2005-004, Utrecht University, 2005.

[19] Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003.

[20] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. To appear in *Proceedings International Conference on Functional Programming, ICFP'06*, 2006.

[21] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL'03*, pages 224–235, New Orleans, January 2003.