Comparing Approaches to Generic Programming in Haskell

Ralf Hinze

Johan Jeuring

Andres Löh

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2006-022

www.cs.uu.nl

ISSN: 0924-3275

Comparing Approaches to Generic Programming in Haskell

Draft lecture notes
for the
Spring School
on
Datatype-Generic Programming 2006

Ralf Hinze[1], Johan Jeuring[2], and Andres Löh[1]

[1] Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
{ralf,loeh}@informatik.uni-bonn.de
[2] Department of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
johanj@cs.uu.nl

Abstract. The last decade has seen a number of approaches to generic programming: PolyP, Functorial ML, 'Scrap Your Boilerplate', Generic Haskell, 'Generics for the Masses', etc. The approaches vary in sophistication and target audience: some propose full-blown programming languages, some suggest libraries, some can be seen as categorical programming methods. In these lecture notes, we shall compare the various approaches: we will introduce each method by means of example, and we will evaluate it along different dimensions (expressivity, ease of use, etc).

1 Introduction

You just started implementing your third web shop in Haskell, and you realize that a lot of the code you have to write is similar to the code for the previous web shops. Only the data types have changed. Unfortunately, this implies that all reporting, editing, storing and loading in the database functionality, and probably a lot more, has to be changed. You've heard about generic programming, a technique which can be used to automatically generate programs depending on types. But searching on the web gives you almost ten approaches to solve your problem: DrIFT, PolyP, Generic Haskell, Derivable Type Classes, Template Haskell, Scrap Your Boilerplate, Generics for the Masses, Strafunski, etc. How do you choose? And these are only the approaches to generic programming in Haskell. If you are also flexible in the programming language you use, there is a much larger variety of different approaches to generic programming to choose from.

In these lecture notes we give arguments why you would choose for a particular approach to generic programming to solve your generic programming problem. We will compare different approaches to generic programming along different lines, such as for example:

- Can you use generic programs on all types definable in the programming language?
- Are generic programs compiled or interpreted?
- Can you extend a generic program in a special way for a particular data type?

Before we will compare the various approaches to generic programming we will first discuss in detail the criteria on which the comparison is based.

'Generic' is an over-used adjective in computing science in general, and in programming languages in particular. Ada has generic packages, Java has generics, Eiffel has generic classes, etc. Usually, the adjective generic is used to describe that a concept allows abstractions over a larger class of entities than was previously possible. However, broadly speaking most uses of generic refer to some form of parametric polymorphism, ad-hoc polymorphism, and/or inheritance. For a nice

comparison of the different incarnations of generic concepts in different programming languages, see Garcia et al [20]. Already in the 1970s this was an active area of research [68, 51, 17].

In the context of these lecture notes, generic programming means a form of programming in which a function takes a type as argument, and its behavior depends upon the *structure* of this type. The type argument, which may be explicit or implicit, represents the type of values to which the function is applied, or which the function returns. A typical example is the equality function, where a type argument `t` dictates the form of the code that performs the equality test on two values of type `t`. In the past we have just the adjective *polytypic* instead of generic, which is less confusing and describes the concept a bit more accurate. However, the adjective hasn't been picked up by other people working on conceptually the same topic, and maybe it sounds a bit scary.

Types play a fundamental rôle in generic programming. Therefore, we will look in particular at programming languages with a static type system. It is possible to do generic programming in an untyped language or in a language with a dynamic type system, and some people think it is much easier to do generic programming in these languages. We disagree whole-heartedly: since generic programming is fundamentally about programming with types, simulating generic programming in an untyped language is difficult, since desirable concepts, checks, and abstractions are missing. To illustrate our argument we will include an untyped approach to generic programming in our comparison, namely DrIFT.

We will introduce each approach to generic programming by means of a number of, more or less, canonical examples. This set of examples has been obtained by collecting the functions defined in almost 20 papers introducing the various approaches to generic programming. Almost all of these papers contain at least one function from the top five of examples thus obtained. Furthermore, the top five examples exhibit different characteristics, which we use to show differences between approaches to generic programming. The functions we will implement for all approaches are:

- *encode*, a function that encodes a value of any type as a list of bits. The function encode is a simple recursive function which 'destructs' a value of a data type into a list of bits. The inverse of *encode*, called *decode*, is a function which *builds* a value of a data type from a list of bits.
- *eq*, a function that takes *two* values, and compares them for equality.
- *map*, a generalization of the standard *map* function on lists. On a parameterized data type, such as lists, function *map* takes a function argument and a value of the data type, and applies the function argument to all parametric values inside the value argument. The function *map* is particularly useful when applied to type constructors (of which the list type is an example), instead of types.
- *show*, a function that shows or pretty-prints a value of a data type.
- *update*, a function that takes a value of a data type representing the structure of a company, and updates the salaries that appear in this value. The characterizing feature of this example is that *update* is only interested in values of a very small part of a possibly very large type.

We will not define all of these functions for each approach, in particular not for approaches that are very similar, but we will use these examples to highlight salient points. We will then investigate a number of properties for each approach. Examples of these properties are: is it possible to define a generic function on any data type that can be defined in the programming language (full reflexivity), is the programming language type safe, do generic functions satisfy desirable properties, etc. Not all properties can be illustrated by means of these examples, so sometimes we will use other examples. In these draft lecture notes for the Spring School on Datatype-Generic Programming 2006 we will compare the approaches to generic programming in Haskell:

- Generic Haskell [24, 27, 55, 57].
- DrIFT [74].
- PolyP [37].
- Derivable Type Classes [32].
- Lightweight Generics and Dynamics [13].
- Scrap Your Boilerplate [48, 50, 49].

- Generics for the masses [28].
- Clean [3, 2].
- Strafunski [47].
- Using Template Haskell for generic programming [63].

The last two approaches will only be included in the final version of these lecture notes. They are rather similar to Scrap Your Boilerplate and DrIFT, respectively. In the final version of these lecture notes we also plan to consider the following non-Haskell approaches to generic programming:

- Charity [15].
- ML [12, 19].
- Intensional type analysis [22, 16, 72].
- Extensional type analysis [18].
- Functorial ML [43, 62], the Constructor Calculus [40], the Pattern Calculus [41, 42], FISh [39].
- Dependently typed generic programming [6, 10].
- Type-directed programming in Java [73].
- Maude [58].

We have tried to be as complete as possible, but certainly this list is not exhaustive.

These notes are organized as follows. In Section 2 we discuss why generic programming matters by means of a couple of representative examples. We will use these examples in Section 4 to compare the various approaches to generic programming by means of the criteria introduced and discussed in Section 3. Finally, Section 5 concludes.

2 Why generic programming matters

Software development often consists of designing a data type, to which functionality is added. Some functionality is data type specific, other functionality is defined on almost all data types, and only depends on the type structure of the data type. Examples of generic functionality defined on almost all data types are storing a value in a database, editing a value, comparing two values for equality, pretty-printing a value, etc. A function that works on many data types is called a generic function. Applications of generic programming can be found not just in the rather small programming examples mentioned, but also in

- XML tools such as XML compressors [29], and type-safe XML data binding tools [7];
- automatic testing [46];
- constructing 'boilerplate' code that traverses a value of a rich set of mutually recursive data types, applying real functionality at a small portion of the data type [48, 55, 49];
- structure editors such as XML editors [21], and generic graphical user interfaces [1];
- data conversion tools [38] which for example store a data type value in a database [21], or output it as XML, or in a binary format [70], or ...

Change is endemic to any large software system. Business, technology, and organization frequently change during the life cycle of a software system. However, changing a large software system is difficult: localizing the code that is responsible for a particular part of the functionality of a system, changing it, and ensuring that the change does not lead to inconsistencies in other parts of the system or in the architecture or documentation is usually a challenging task. *Software evolution* is a fact of life in the software development industry [52, 53, 66].

If a data type changes, or a new data type is added to a piece of software, a generic program automatically adapts to the changed or new data type. An example is a generic program for calculating the total amount of salaries paid by an organization. If the structure of the organization changes, for example by removing or adding an organizational layer, the generic program still calculates the total amount of salaries paid. Since a generic program automatically adapts changes of data types, a programmer only has to program 'the exception'. Generic programming has the potential to solve at least an important part of the software evolution problem [45].

In the rest of this section we will show a number of examples of generic programs. We will write the generic programs in Generic Haskell [24, 30, 54]. Generic Haskell is an extension of the lazy, higher-order, functional programming language Haskell [65] that supports generic programming. We could have chosen many of the approaches to generic programming for presenting the example generic programs: most approaches can express all the examples. The choice for Generic Haskell is rather random, and related to the fact that we are responsible for the development of Generic Haskell. We use the most recent version of Generic Haskell, known as *Dependency-style Generic Haskell* [55, 54]. Dependencies both simplify and increase the expressiveness of generic programming. In Section 4 we will show how these programs are written in other approaches to generic programming.

2.1 Data types in Haskell

The functional programming language Haskell 98 provides an elegant and compact notation for declaring data types. In general, a data type introduces a number of constructors, where each constructor takes a number of arguments. Here are two example data types:

```
data CharList = Nil | Cons Char CharList
data Tree     = Empty | Leaf Int | Bin Tree Char Tree.
```

A character list, a value of type CharList, is often called a *string*. It is either empty, denoted by the constructor *Nil*, or it is a character c followed by the remainder of the character list cs, denoted *Cons* c cs, where *Cons* is the constructor. A tree, a value of type Tree, is empty, a leaf containing an integer, or a binary node containing two subtrees and a character.

These example types are of kind $\star$, meaning that they do not take any type arguments. A kind can be seen as the 'type of a type'. The following type takes an argument; it is obtained by abstracting Char out of the CharList data type above:

```
data List a = Nil | Cons a (List a).
```

Here List is a type constructor, which, when given a type a, constructs the type List a. The type constructor List has the functional kind $\star \to \star$. The list data type is predefined in Haskell: the type List a is written [a], the constructors *Nil* and *Cons* x xs are written [] and $x : xs$, respectively. A type can take more than one argument. If we abstract from the types Char and Int in the type Tree, we obtain the type GTree defined by:

```
data GTree a b = GEmpty | GLeaf a | GBin (GTree a b) b (GTree a b).
```

The type constructor GTree takes two type arguments, both of kind $\star$, and hence has kind $\star \to \star \to \star$.

Arguments of type constructors need not be of kind $\star$. Consider the data type of Rose trees, defined by:

```
data Rose a = Node a [Rose a].
```

A Rose tree is a *Node* containing an element of type a, and a list of child trees. Just as List, Rose has kind $\star \to \star$. If we abstract from the list type in Rose, we obtain the data type GRose defined by:

```
data GRose f a = GNode a (f (GRose f a)).
```

Here the type argument f has kind $\star \to \star$, just as the List type constructor, and it follows that GRose has kind $(\star \to \star) \to \star \to \star$.

All the examples of data types we have given until now are examples of so-called *regular* data types: a recursive, parametrized type whose recursive definition does not involve a change of the type parameter(s). Non-regular or *nested* types [11] are practically important since they

can capture data-structural invariants in a way that regular data types cannot. For instance, the following data type declaration defines a nested data type: the type of perfectly balanced, binary leaf trees [25] — perfect trees for short.

```
data Perfect a = ZeroP a | SuccP (Perfect (Fork a))
data Fork a    = Fork a a
```

This equation can be seen as a bottom-up definition of perfect trees: a perfect tree is either a singleton tree or a perfect tree that contains pairs of elements. Here is a perfect tree of type Perfect Int:

```
SuccP (SuccP (SuccP (ZeroP (Fork (Fork (Fork 2 3)
                                       (Fork 5 7))
                                 (Fork (Fork 11 13)
                                       (Fork 17 19)))))).
```

Note that the height of the perfect tree is encoded in the prefix of *SuccP* and *ZeroP* constructors.

2.2 Structure representation types

To apply functions generically to all data types, we must view data types in a uniform manner: every Haskell data type can be viewed as a labeled sum of possibly labeled products. This encoding is based on the following data types:

```
data a :+: b  = Inl a | Inr b
data a :*: b  = a :*: b
data Unit     = Unit
data Con a    = Con ConDescr a
data Label a = Label LabelDescr a.
```

The choice between *Nil* and *Cons*, for example, is encoded as a sum using the type :+: (nested to the right if there are more than two constructors). The constructors of a data type are encoded as sum labels, represented by the type Con, which contains a description of a constructor in the form of a value of type ConDescr. The constructors *Nil* and *Cons* are represented by ConDescr values *nilDescr* and *consDescr*, respectively. The exact details of how constructors are represented are omitted. Record names are encoded as product labels, represented by a value of the type Label, which contains a value of type LabelDescr. Arguments such as the a and List a of the *Cons* are encoded as products using the type :*: (nested to the right if there are more than two arguments). In the case of *Nil*, an empty product, denoted by Unit, is used. The arguments of the constructors are not translated. Finally, abstract types and primitive types such as Char are not encoded, but left as they are.

Now we can encode CharList, Tree, and List as

```
type CharList° = Con Unit :+: Con (Char :*: CharList)
type Tree°     = Con Unit :+: Con Int :+: Con (Tree :*: (Char :*: Tree))
type List° a   = Con Unit :+: Con (a :*: (List a)).
```

These representations are called *structure representation types*. A type t and its structural representation type t° are isomorphic (ignoring undefined values). This isomorphism is witnessed by a so-called *embedding-projection pair*: a value $conv_{\mathsf{t}} :: \mathsf{EP\ t\ t^\circ}$ of the data type

```
data EP a b = EP (a → b) (b → a).
```

For example, for the List data type we have that $conv_{\mathsf{List}} = Ep\ from_{\mathsf{List}}\ to_{\mathsf{List}}$, where $from_{\mathsf{List}}$ and to_{List} are defined by

$$
\begin{aligned}
&from_{\mathsf{List}} && :: \forall \mathsf{a} \,.\, \mathsf{List\ a} \to \mathsf{List}^{\circ}\ \mathsf{a} \\
&from_{\mathsf{List}}\ Nil && = Inl\ (Con\ nilDescr\ Unit) \\
&from_{\mathsf{List}}\ (Cons\ a\ as) && = Inr\ (Con\ consDescr\ (a :*: as)) \\
&to_{\mathsf{List}} && :: \forall \mathsf{a} \,.\, \mathsf{List}^{\circ}\ \mathsf{a} \to \mathsf{List\ a} \\
&to_{\mathsf{List}}\ (Inl\ (Con\ \_\ Unit)) && = Nil \\
&to_{\mathsf{List}}\ (Inr\ (Con\ \_\ (a :*: as))) && = Cons\ a\ as.
\end{aligned}
$$

The Generic Haskell compiler generates the translation of a type to its structural representation, together with the corresponding embedding projection pair. More details about the correspondence between these and Haskell types can be found elsewhere [27].

A generic program is defined by induction on structure types. Whenever a generic program is applied to a user-defined data type, the Generic Haskell compiler takes care of the mapping between the user-defined data type and its corresponding structural representation. Furthermore, a generic program may also be defined directly on a user-defined data type, in which case this definition takes precedence over the automatically generated definitions. A definition of a generic function on a user-defined data type is called a *default case*. To develop a generic function, it is best to first consider a number of its instances for specific data types.

2.3 Encoding and decoding

A classic application area of generic programming is parsing and unparsing, i.e., reading values of different types from some universal representation, or writing values to that universal representation. The universal representation can be aimed at being human-readable (such as the result of Haskell's *show* function); or it can be intended for data exchange, such as XML. Other applications include encryption, transformation, or storage.

In this section we will treat a very simple case of compression, by defining functions that can write to and read from a sequence of bits. A bit is defined by the following data type declaration:

$$\textbf{data}\ \mathsf{Bit} = O \mid I$$

(the names O and I are used as constructors here).

Function encode on CharList. To define *encode* on the data type CharList, we assume that there exists a function $encodeChar :: \mathsf{Char} \to [\mathsf{Bit}]$, which takes a character and returns a list of bits representing that character. We assume that encodeChar returns a list of 8 bits, corresponding to the ASCII number of the character. A value of type CharList is now encoded as follows:

$$
\begin{aligned}
&encodeCharList && :: \mathsf{CharList} \to [\mathsf{Bit}] \\
&encodeCharList\ Nil && = [O] \\
&encodeCharList\ (Cons\ c\ cs) && = I : encodeChar\ c +\!\!+ encodeCharList\ cs.
\end{aligned}
$$

For example, applying *encodeCharList* to the string `"Bonn"` defined as a CharList by $bonn = Cons\ \texttt{'B'}\ (Cons\ \texttt{'o'}\ (Cons\ \texttt{'n'}\ (Cons\ \texttt{'n'}\ Nil)))$ gives

$$ComparingGP\rangle\ \ encodeCharList\ bonn$$

$$
\begin{aligned}
&[I, O, I, O, O, O, O, I, O, I, O, I, I, O, I, I, I, I, I \\
&, O, I, I, O, I, I, I, O, I, O, I, O, I, I, I, O, O].
\end{aligned}
$$

Function encode on Tree. To define *encode* on the data type Tree, we assume there exists, besides a function *encodeChar*, a function $encodeInt :: \mathsf{Int} \to [\mathsf{Bit}]$, which takes an integer and returns a list of bits representing that integer. A value of type Tree can then be encoded as follows:

$$
\begin{aligned}
&encodeTree && :: \mathsf{Tree} \to [\mathsf{Bit}] \\
&encodeTree\ Empty && = [O, O]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{encodeTree}\ (\mathit{Leaf}\ i) &&= [O, I] +\!\!+ \mathit{encodeInt}\ i\\
&\mathit{encodeTree}\ (\mathit{Bin}\ l\ c\ r) &&= [I, O]\\
&&&\quad +\!\!+ \mathit{encodeTree}\ l\\
&&&\quad +\!\!+ \mathit{encodeChar}\ c\\
&&&\quad +\!\!+ \mathit{encodeTree}\ r.
\end{aligned}
$$

Function encode on List a. The data type CharList is an instance of the data type List a, where a is Char. How do we define an encoding function on the data type List a? For character lists, we assumed the existence of an encoding function for characters. Here we take the same approach: to encode a value of type List a, we assume that we have a function for encoding values of type a. Abstracting from *encodeChar* in the definition of *encodeCharList* we obtain:

$$
\begin{aligned}
&\mathit{encodeList} &&:: (\mathsf{a} \to [\mathsf{Bit}]) \to \mathsf{List\ a} \to [\mathsf{Bit}]\\
&\mathit{encodeList}\ \mathit{encodeA}\ \mathit{Nil} &&= [O]\\
&\mathit{encodeList}\ \mathit{encodeA}\ (\mathit{Cons}\ x\ xs) &&= I : \mathit{encodeA}\ x +\!\!+ \mathit{encodeList}\ \mathit{encodeA}\ xs.
\end{aligned}
$$

Generic encode. The encoding functions on CharList, Tree and List a follow the same pattern: encode the choice made for the top level constructors, and concatenate the encoding of the children of the constructor. We can capture this common pattern in a single generic definition by defining the encoding function by induction on the structure of data types. This means that we define *encode* on sums (:+:), on products (:*:), and on base types such as Unit, Int and Char, as well as on the sum labels (Con) and the product labels (Label).

The only place where there is a choice between different constructors is in the :+: type. Here, the value can be either an *Inl* or an *Inr*. If we have to encode a value of type Unit, it can only be *Unit*, so we need no bits to encode that knowledge. Similarly, for a product we know that the value is the first component followed by the second – we need no extra bits except the encodings of the components.

In Generic Haskell, the generic *encode* function is rendered as follows:

$$
\begin{aligned}
&\mathit{encode}\{\!|\mathsf{a} :: *|\!\}\ &&&&:: (\mathit{encode}\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to [\mathsf{Bit}]\\
&\mathit{encode}\{\!|\mathsf{Unit}|\!\}\ &&\mathit{Unit}\ &&= [\,]\\
&\mathit{encode}\{\!|\mathsf{Int}|\!\}\ &&i\ &&= \mathit{encodeInt}\ i\\
&\mathit{encode}\{\!|\mathsf{Char}|\!\}\ &&c\ &&= \mathit{encodeChar}\ c\\
&\mathit{encode}\{\!|\alpha :+: \beta|\!\}\ &&(\mathit{Inl}\ x)\ &&= O : \mathit{encode}\{\!|\alpha|\!\}\ x\\
&\mathit{encode}\{\!|\alpha :+: \beta|\!\}\ &&(\mathit{Inr}\ y)\ &&= I\ : \mathit{encode}\{\!|\beta|\!\}\ y\\
&\mathit{encode}\{\!|\alpha :*: \beta|\!\}\ &&(x_1 :*: x_2)\ &&= \mathit{encode}\{\!|\alpha|\!\}\ x_1 +\!\!+ \mathit{encode}\{\!|\beta|\!\}\ x_2\\
&\mathit{encode}\{\!|\mathsf{Label}\ l\ \alpha|\!\}\ &&(\mathit{Label}\ a)\ &&= \mathit{encode}\{\!|\alpha|\!\}\ a\\
&\mathit{encode}\{\!|\mathsf{Con}\ c\ \alpha|\!\}\ &&(\mathit{Con}\ a)\ &&= \mathit{encode}\{\!|\alpha|\!\}\ a.
\end{aligned}
$$

There are a couple of things to note about generic function definitions:

- The function $\mathit{encode}\{\!|\mathsf{a}|\!\}$ is a type-indexed function. The type argument appears in between special parentheses {|, |}. An instance of *encode* is obtained by applying *encode* to a type. For example, $\mathit{encode}\{\!|\mathsf{CharList}|\!\}$ is the instance of the generic function *encode* on the data type CharList. This instance is semantically the same as the definition of *encodeCharList*.
- The constraint $\mathit{encode}\{\!|\mathsf{a}|\!\}$ that appears in the type of *encode* says that *encode depends on* itself. A generic function f depends on a generic function g if there is an arm in the definition of f, for example the arm for $f\{\!|\alpha :+: \beta|\!\}$ that uses g on a variable in the type argument, for example $g\{\!|\alpha|\!\}$. If a generic function depends on itself it is defined by induction over the type structure.
- The type of *encode* is given for a type a of kind $\star$. This does not mean that *encode* can only be applied to types of kind $\star$; it only gives the type information for types of kind $\star$. The

type of function *encode* on types with kinds other than $\star$ can automatically be derived from this base type. In particular, $encode\{\!|\mathsf{List}|\!\}$ will be translated to a value that has the type $(\mathsf{a} \to [\mathsf{Bit}]) \to (\mathsf{List\ a} \to [\mathsf{Bit}])$.

The Con and the Label case are more interesting for generic functions that use the names of constructors and labels in some way, such as a generic *show* function. Most generic functions, however, essentially ignore these branches. In this case, we will omit these branches from the generic function definition.

Generic decode. The inverse of *encode* recovers a value from a list of bits. This inverse function is called *decode*, and is defined in terms of a function *decodes*, which takes a list of bits, and returns a list of values that can be recovered from an initial segment of the list of bits. The reason we define this example as well, is that we want to show how to generically build or construct a value of a data type.

$$
\begin{aligned}
&mapFst\ f\ (x, y) = (f\ x, y)\\
&decodes\{\!|\mathsf{a} :: *|\!\} &&:: (decodes\{\!|\mathsf{a}|\!\}) \Rightarrow [\mathsf{Bit}] \to [(\mathsf{a}, [\mathsf{Bit}])]\\
&decodes\{\!|\mathsf{Unit}|\!\}\ \ xs &&= [(Unit, xs\ \ \ \ )\ ]\\
&decodes\{\!|\mathsf{Int}|\!\}\ \ xs &&= decodesInt\ xs\\
&decodes\{\!|\mathsf{Char}|\!\}\ \ xs &&= decodesChar\ xs\\
&decodes\{\!|\alpha :\!+\!: \beta|\!\}\ (O : xs) &&= map\ (mapFst\ Inl)\ (decodes\{\!|\alpha|\!\}\ xs)\\
&decodes\{\!|\alpha :\!+\!: \beta|\!\}\ (I\ \ : xs) &&= map\ (mapFst\ Inr)\ (decodes\{\!|\beta|\!\}\ xs)\\
&decodes\{\!|\alpha :\!+\!: \beta|\!\}\ [\,] &&= [\,]\\
&decodes\{\!|\alpha :\!*\!: \beta|\!\}\ xs &&= [(y_1 :\!*\!: y_2, r_2) \mid (y_1, r_1) \leftarrow decodes\{\!|\alpha|\!\}\ xs\\
&&&\qquad\qquad , (y_2, r_2) \leftarrow decodes\{\!|\beta|\!\}\ r_1\,]
\end{aligned}
$$

The function is a bit more involved than *encode*, because it has to deal with incorrect input, and it has to return the unconsumed part of the input. This is solved using the standard list-of-successes technique, where the input list is transformed into a list of pairs, containing all possible parses with the associated unconsumed part of the input. The decoding process is not ambiguous, so only lists of zero (indicating failure) and one (indicating success) elements occur. As with *encodeChar*, we assume a function *decodeChar* is obtained from somewhere.

A value of type Unit is represented using no bits at all, therefore it can be decoded without consuming any input. Except for the primitive types such as Char and Int, the case for :+: is the only place where input is consumed (as it is the only case where output is produced in *encode*), and depending on the first bit of the input, we produce an *Inl* or an *Inr*. A third case lets the decoding process fail if we run out of input while decoding a sum. The product first decodes the left component, and then runs *decodes* for the right component on the rest of the input.

The inverse of *encode* is now defined by:

$$
\begin{aligned}
&decode\{\!|\mathsf{a} :: *|\!\}\ ::\ (decodes\{\!|\mathsf{a}|\!\}) \Rightarrow [\mathsf{Bit}] \to \mathsf{a}\\
&decode\{\!|\mathsf{a}|\!\}\ x\ \ = \mathbf{case}\ decodes\{\!|\mathsf{a}|\!\}\ x\ \mathbf{of}\\
&\qquad\qquad\qquad [(y, [\,])] \to y\\
&\qquad\qquad\qquad \_\qquad\ \to error\ \texttt{"decode: no parse"}.
\end{aligned}
$$

Note that although this is a generic function, it is not defined by induction on the structure of types. Instead, it is defined in terms of another generic function, *decodes*. A generic function f that is defined in terms of another generic function g is called a *generic abstraction*. Such a generic function does not depend on itself, but on g instead. Using a generic abstraction, we can thus define a function that depends on a type argument, but is not defined using cases on types.

For a value x of type t, we have

$$(decode\{\!|\mathsf{t}|\!\}\ .\ encode\{\!|\mathsf{t}|\!\})\ x == x,$$

provided there is no specialization error.

2.4 Equality

In this subsection we define the generic equality function, which takes *two* arguments instead of a single argument as *encode*. We define the equality function on two of the example data types given in Section 2.1. Two character lists are equal if both are empty, or if both are non-empty, the first elements are equal, and the tails of the lists are equal.

$$
\begin{array}{lllll}
\mathit{eqCharList} :: \mathsf{CharList} \to \mathsf{CharList} \to \mathsf{Bool} \\
\mathit{eqCharList} & \mathit{Nil} & \mathit{Nil} & = \mathit{True} \\
\mathit{eqCharList} & (\mathit{Cons}\ x\ xs) & (\mathit{Cons}\ y\ ys) & = \mathit{eqChar}\ x\ y \wedge \mathit{eqCharList}\ xs\ ys \\
\mathit{eqCharList} & \_ & \_ & = \mathit{False},
\end{array}
$$

where *eqChar* is the equality function on characters.

Two trees are equal if both are empty, both are a leaf containing the same integer, determined by means of function *eqInt*, or if both are nodes containing the same subtrees, in the same order, and the same characters.

$$
\begin{array}{lllll}
\mathit{eqTree} :: \mathsf{Tree} \to \mathsf{Tree} \to \mathsf{Bool} \\
\mathit{eqTree} & \mathit{Empty} & \mathit{Empty} & = \mathit{True} \\
\mathit{eqTree} & (\mathit{Leaf}\ i) & (\mathit{Leaf}\ j) & = \mathit{eqInt}\ i\ j \\
\mathit{eqTree} & (\mathit{Bin}\ l\ c\ r) & (\mathit{Bin}\ v\ d\ w) & = \mathit{eqTree}\ l\ v \wedge \mathit{eqChar}\ c\ d \wedge \mathit{eqTree}\ r\ w \\
\mathit{eqTree} & \_ & \_ & = \mathit{False}
\end{array}
$$

The equality functions on CharList and Tree follow the same pattern: compare the top level constructors, and, if they equal, pairwise compare their arguments. We can capture this common pattern in a single generic definition by defining the equality function by induction on the structure of data types.

$$
\begin{array}{lllll}
\mathit{eq}\{\!|\mathsf{a} :: *|\!\}\ ::\ (\mathit{eq}\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \\
\mathit{eq}\{\!|\mathsf{Unit}|\!\} & \_ & \_ & = \mathit{True} \\
\mathit{eq}\{\!|\mathsf{Int}|\!\} & i & j & = \mathit{eqInt}\ i\ j \\
\mathit{eq}\{\!|\mathsf{Char}|\!\} & c & d & = \mathit{eqChar}\ c\ d \\
\mathit{eq}\{\!|\alpha :+: \beta|\!\} & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = \mathit{eq}\{\!|\alpha|\!\}\ x\ y \\
\mathit{eq}\{\!|\alpha :+: \beta|\!\} & (\mathit{Inl}\ x) & (\mathit{Inr}\ y) & = \mathit{False} \\
\mathit{eq}\{\!|\alpha :+: \beta|\!\} & (\mathit{Inr}\ x) & (\mathit{Inl}\ y) & = \mathit{False} \\
\mathit{eq}\{\!|\alpha :+: \beta|\!\} & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = \mathit{eq}\{\!|\beta|\!\}\ x\ y \\
\mathit{eq}\{\!|\alpha :*: \beta|\!\} & (x :*: y) & (v :*: w) & = \mathit{eq}\{\!|\alpha|\!\}\ x\ v \wedge \mathit{eq}\{\!|\beta|\!\}\ y\ w
\end{array}
$$

2.5 Map

In this section we define the generic map function. This example illustrates the importance of kinds in generic programming. To understand the definition of the generic *map* function, it helps to first study the generic copy function. The generic *copy* function is defined as follows:

$$
\begin{array}{lll}
\mathit{copy}\{\!|\mathsf{a} :: *|\!\} & :: (\mathit{copy}\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \\
\mathit{copy}\{\!|\mathsf{Unit}|\!\}\ x & = x \\
\mathit{copy}\{\!|\mathsf{Int}|\!\}\ x & = x \\
\mathit{copy}\{\!|\mathsf{Char}|\!\}\ x & = x \\
\mathit{copy}\{\!|\alpha :+: \beta|\!\}\ (\mathit{Inl}\ x) & = \mathit{Inl}\ (\mathit{copy}\{\!|\alpha|\!\}\ x) \\
\mathit{copy}\{\!|\alpha :+: \beta|\!\}\ (\mathit{Inr}\ x) & = \mathit{Inr}\ (\mathit{copy}\{\!|\beta|\!\}\ x) \\
\mathit{copy}\{\!|\alpha :*: \beta|\!\}\ (x :*: y) & = \mathit{copy}\{\!|\alpha|\!\}\ x :*: \mathit{copy}\{\!|\beta|\!\}\ y.
\end{array}
$$

Note that we have made a choice in the code above: the definition is written recursively, applying the generic copy deeply to all parts of a value. We could have simplified the last three lines, removing the dependency of *copy* on itself:

$$copy\{\!|\alpha :+: \beta|\!\}\ x = x$$
$$copy\{\!|\alpha :*: \beta|\!\}\ x = x.$$

But retaining the dependency and applying the function recursively has an advantage: using a so-called *local redefinition* we can change the behavior of the function. As an example, we can increase all elements of a list by one, using the function

$$incBy1\ x = \mathbf{let}\ copy\{\!|\mathsf{a}|\!\} = (+1)\ \mathbf{in}\ copy\{\!|[\mathsf{a}]|\!\}\ x.$$

Here we locally redefine *copy* to behave as the function $(+1)$ on values of type a that appear in a list of type $[\mathsf{a}]$. Note that this is something that would normally be written as an application of *map*:

$$incBy1\ x = map\ (+1)\ x.$$

If we compare *map* with the locally redefined version of *copy*, then two differences spring to mind. First, the function *map* can only be used on lists, whereas *copy* can be used on other data types as well. Second, *map* has a more liberal type. If we define

$$map'\ f = \mathbf{let}\ copy\{\!|\mathsf{a}|\!\} = f\ \mathbf{in}\ copy\{\!|[\mathsf{a}]|\!\},$$

then we can observe that *map′*, compared to *map* has a more restricted type:

$$map' :: \forall \mathsf{a} :: \star\,.\qquad (\mathsf{a} \to \mathsf{a}) \to [\mathsf{a}] \to [\mathsf{a}]$$
$$map\ :: \forall (\mathsf{a} :: \star)\ (\mathsf{b} :: \star)\,.\,(\mathsf{a} \to \mathsf{b}) \to [\mathsf{a}] \to [\mathsf{b}].$$

The function passed to *map* may change the type of its argument; the function passed to *map′* must preserve the argument type.

Inspired by this deficiency, we can ask ourselves if it would not be possible to also pass a function of type $\mathsf{a} \to \mathsf{b}$ while locally redefining *copy*. The function $copy\{\!|[\mathsf{a}]|\!\}$ has the qualified type

$$copy\{\!|[\mathsf{a}]|\!\} :: \forall \mathsf{a} :: *\,.\,(copy\{\!|\mathsf{a}|\!\} :: \mathsf{a} \to \mathsf{a}) \Rightarrow [\mathsf{a}] \to [\mathsf{a}],$$

but we are now going to generalize this type to something like

$$map\{\!|[\mathsf{a}]|\!\} :: \forall (\mathsf{a} :: *)\ (\mathsf{b} :: *)\,.\,(map\{\!|\mathsf{a}|\!\} :: \mathsf{a} \to \mathsf{b}) \Rightarrow [\mathsf{a}] \to [\mathsf{b}],$$

thereby renaming function *copy* to *map* (but using exactly the same definition). For this to work, *map* needs a different type signature:

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

The function is now parametrized over *two* type variables, and so is the dependency. When used at a constant type, both variables a and b are instantiated to the same type – only when locally redefining the function for a dependency variable, the additional flexibility is available. Figure 1 shows some types for applications of *map* to specific type arguments.

For example, assume the (data) types Pair and Either are defined by:

$$\mathbf{type}\ \mathsf{Pair\ a\ b} \quad = (\mathsf{a}, \mathsf{b})$$
$$\mathbf{data}\ \mathsf{Either\ a\ b} = \mathit{Left}\ \mathsf{a} \mid \mathit{Right}\ \mathsf{b}.$$

Then the expressions

$$map\{\!|[\,]|\!\} \quad (+1) \quad [1,2,3,4,5]$$
$$map\{\!|\mathsf{Pair}|\!\} \quad (*2)\ (\texttt{"y"}+\!\!+)\ (21, \texttt{"es"})$$
$$map\{\!|\mathsf{Either}|\!\}\ \mathit{not}\ \ \mathit{id} \quad (\mathit{Left}\ \mathit{True})$$

evaluate to $[2,3,4,5,6]$, $(42, \texttt{"yes"})$, and $\mathit{Left}\ \mathit{False}$, respectively.

```
map{|Tree :: *|} :: Tree → Tree
map{|List (a :: *) :: *|} ::
  ∀(a1 :: *) (a2 :: *) . (map{|a|} :: a1 → a2) ⇒ List a1 → List a2
map{|GTree (a :: *) (b :: *) :: *|} ::
  ∀(a1 :: *) (a2 :: *) (b1 :: *) (b2 :: *) . (map{|a|} :: a1 → a2, map{|b|} :: b1 → b2) ⇒
                                              GTree a1 a2 → GTree b1 b2
map{|GRose (f :: * → *) (a :: *) :: *|} ::
  ∀(f1 :: * → *) (f2 :: * → *) (a1 :: *) (a2 :: *) .
    (map{|f c|} :: ∀(c1 :: *) (c2 :: *) . (map{|c|} :: c1 → c2) ⇒ f1 c1 → f2 c2
    , map{|a|} :: a1 → a2
    ) ⇒ GRose f1 a1 → GRose f2 a2.
```

Fig. 1. Example types for generic applications of *map* to type arguments of different forms.

2.6 Show

The function *show* shows a value of an arbitrary data type. In Haskell, the definition of *show* can be derived for most data types. In this subsection we explain how to define *show* as a generic function in Generic Haskell. The function *show* is an example of a function that uses the constructor descriptor in the Con case. We define *show* in terms of the function *showP*, a slightly generalized variant of Haskell's *show* that takes an additional argument of type String → String. This parameter is used internally to place parentheses around a fragment of the result when needed.

```
showP{|a :: *|} :: (showP{|a|}) ⇒ (String → String) → a → String
showP{|Unit|}       p Unit       = ""
showP{|α :+: β|}    p (Inl x)    = showP{|α|}  p x
showP{|α :+: β|}    p (Inr x)    = showP{|β|}  p x
showP{|α :*: β|}    p (x1 :*: x2) = showP{|α|} p x1 ++ "  " ++ showP{|β|} p x2
showP{|Con c α|}    p (Con x)    = let parens x = "(" ++ x ++ ")"
                                       body      = showP{|α|} parens x
                                   in  if null body
                                       then conName c
                                       else  p (conName c ++ "  " ++ body)
showP{|[α]|}        p xs         = let body = (concat
                                              . intersperse ",  "
                                              . map (showP{|α|} id)
                                              ) xs
                                   in  "[" ++ body ++ "]"
```

The type Unit represents a constructor with no fields. In such a situation, the constructor name alone is the representation, and it will be generated from the Con case, so we do not need to produce any output here. We just descend through the sum structure; again, no output is produced because the constructor names are produced in the Con case. A product concatenates fields of a single constructor; we therefore show both components, and separate them from each other by whitespace.

Most of the work is done in the arm for Con. We show the body of the constructor, using parentheses where necessary. The body is empty if and only if there are no fields for this constructor. In this case, we only return the name of the constructor. Here we make use of the function *conName* on the constructor descriptor c to obtain that name. Otherwise, we connect the constructor name and the output of the body with a space, and possibly surround the result with parentheses.

The last case is for lists and implements Haskell's list syntax, with brackets and commas.

In addition to the cases above, we need cases for abstract primitive types such as Char, Int, or Float that implement the operation in some primitive way.

The function *show* is defined in terms of *showP* via generic abstraction, instantiating the first parameter to the identity function, because outer parentheses are not required.

$$
\begin{aligned}
&show\{\!|\mathsf{a} :: *|\!\} :: (showP\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{String} \\
&show\{\!|\mathsf{a}|\!\} \quad = showP\{\!|\mathsf{a}|\!\}\ id
\end{aligned}
$$

The definition of a generic *read* function that parses the generic string representation of a value is also possible using the Con case, and only slightly more involved because we have to consider partial consumption of the input string and possible failure.

2.7 Update salaries

Adapting from Lämmel and Peyton Jones [48], we use the following data types to represent the organizational structure of a company.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Company} = C\ [\mathsf{Dept}] \\
&\textbf{data}\ \mathsf{Dept} \quad = D\ \mathsf{Name}\ \mathsf{Manager}\ [\mathsf{SubUnit}] \\
&\textbf{data}\ \mathsf{SubUnit} \quad = PU\ \mathsf{Employee} \mid DU\ \mathsf{Dept} \\
&\textbf{data}\ \mathsf{Employee} = E\ \mathsf{Person}\ \mathsf{Salary} \\
&\textbf{data}\ \mathsf{Person} \quad = P\ \mathsf{Name}\ \mathsf{Address} \\
&\textbf{data}\ \mathsf{Salary} \quad = S\ \mathsf{Float} \\
&\textbf{type}\ \mathsf{Manager} = \mathsf{Employee} \\
&\textbf{type}\ \mathsf{Name} \quad = \mathsf{String} \\
&\textbf{type}\ \mathsf{Address} = \mathsf{String}
\end{aligned}
$$

We wish to update a Company value, which involves giving every Person a 15% pay rise. To do so requires visiting the entire tree and modifying every occurrence of Salary. The implementation requires pretty standard "boilerplate" code which traverses the data type, until it finds Salary, where it performs the appropriate update — itself one line of code — before reconstructing the result.

In Generic Haskell writing this function requires but a few lines. The code is based on the generic *map* function. The code to perform the updating is given by the following three lines, the first of which is the mandatory type signature, the second states that the function is based on *map*, and the third performs the update of the salary. The **extends** construct denotes that the cases of *map* are copied into *update*. These are the *default cases* described in Clarke and Löh [14].

$$
\begin{aligned}
&update\{\!|\mathsf{a} :: *|\!\} :: (update\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \\
&update\ \textbf{extends}\ map \\
&update\{\!|\mathsf{Salary}|\!\}\ (S\ s) = S\ (s * (1 + 0.15))
\end{aligned}
$$

3 Criteria for comparing approaches to generic programming

This section discusses the criteria we will use for comparing approaches to generic programming. Together, these criteria can be viewed as a characterization of generic programming. We don't think that all criteria are equally important: some criteria discuss whether or not some functions can be defined or used on particular data types, whereas other criteria discuss more cosmetic aspects. We will illustrate the criteria with an evaluation of Generic Haskell.

3.1 Structure in programming languages

Adding generic programming capabilities to a programming language is a programming language design problem. Many of the criteria we will give are related to or derived from programming language design concepts.

Ignoring modules, many modern programming languages have a two level structure. The bottom level, where the computations take place, consists of values. The top level imposes structure on the value level, and is inhabited by types. On top of this, Haskell adds a level that imposes structure on the type level, namely kinds. Finally, in a dependently typed programming language there is a possibly infinite hierarchy of levels, where level $n + 1$ imposes structure on elements of level n.

In ordinary programming we routinely define values that depend on values, that is, functions, and types that depend on types, that is, type constructors. However, we can also imagine to have dependencies between adjacent levels. For instance, a type might depend on a value or a type might depend on a kind. The following table lists the possible combinations:

kinds depending on kinds	parametric and kind-indexed kinds
kinds depending on types	dependent kinds
types depending on kinds	polymorphic and kind-indexed types
types depending on types	parametric and type-indexed types
types depending on values	dependent types
values depending on types	polymorphic and type-indexed functions
values depending on values	ordinary functions

If a higher level depends on a lower level we have so-called dependent types or dependent kinds. Programming languages with dependent types are the subject of intensive research [60, 9]. We will encounter some of these later in our comparison. Generic programming is concerned with the opposite direction, where a lower level depends on the same or a higher level. For instance, if a value depends on a type we either have a polymorphic or a type-indexed function. In both cases the function takes a type as an argument. What is the difference between the two? A polymorphic function is a function that happens to be insensitive to what type the values in some data type are. Take, for example, the length function that calculates the length of a list. Since it does not have to inspect the elements of an argument list, it has type $\forall \mathsf{a} \,.\, \mathsf{List\ a} \to \mathsf{Int}$. By contrast, a type-indexed function is defined by induction on the structure of its type argument. In some sense, the type argument guides the computation which is performed on the value arguments.

Not only values may depend on types, but also types. For example, the type constructor List depends on a type argument. We can make a similar distinction as on the value level. A parametric type, such as List, does not inspect its type argument. A *type-indexed type* [31], on the other hand, is defined by induction on the structure of its type argument. An example of a type-indexed data type is the zipper data type introduced by Huet [35]. Given a data type t, the zipper data type corresponding to t can be defined by induction on the data type t. Finally, we can play the same game on the level of kinds. The following table summarizes the interesting cases.

kinds defined by induction on the structure of kinds	kind-indexed kinds
kinds defined by induction on the structure of types	–
types defined by induction on the structure of kinds	kind-indexed types
types defined by induction on the structure of types	type-indexed types
types defined by induction on the structure of values	–
values defined by induction on the structure of types	type-indexed values
values defined by induction on the structure of values	–

For each of the approaches to generic programming we will discuss what can depend on what.

Structural dependencies. Which concepts may depend on which concepts?

Generic Haskell supports the definition of type-indexed values, as all the examples in the previous section show. Type arguments appear between special parentheses {|, |}. A type-indexed value has a kind-indexed type, of which the base case, the case for kind $\star$, has to be supplied by the programmer. The inductive case, the case for $k \to l$ but is automatically generated by the compiler (as it is determined by the way Generic Haskell specializes generic functions).

Generic Haskell also supports the definition of type-indexed types. A type-indexed type is defined in the same way as a type-indexed function, apart from the facts that every line in its definition starts with **type**, and its name starts with a capital. A type-indexed type has a kind-indexed kind [31].

3.2 The Type Completeness Principle

The Type Completeness Principle [71] says that no programming language operation should be arbitrarily restricted in the types of its operands. For example, in Haskell, a function can take an argument of any type, including a function type, and a tuple may contain a function. To a large extent, Haskell satisfies the type completeness principle on the value level. There are exceptions, however. For example, it is not possible to pass a polymorphic function as argument. Pascal does not satisfy the type completeness principle, since, for example, procedures cannot be part of composite values.

The type completeness principle leads to the following criteria.

Full reflexivity. A generic programming language is fully reflexive if a generic function can be used on *any* type that is definable in the language.

Generic Haskell is fully reflexive with respect to the types definable in Haskell 98, except for constraints in data type definitions. So a data type of the form

$$\textbf{data}\ \mathit{Eq}\ \mathsf{a} \Rightarrow \mathsf{Set\ a} = \mathit{NilSet} \mid \mathit{ConsSet}\ \mathsf{a}\ (\mathsf{Set\ a})$$

is not dealt with correctly. However, constrained data types are a corner case in Haskell and can easily be simulated using other means.

Generic functions cannot be used on existential data types, such as for example

$$\begin{aligned}&\textbf{data}\ \mathsf{Foo} = \forall \mathsf{a}\,.\,\mathit{MkFoo}\ \mathsf{a}\ (\mathsf{a} \to \mathsf{Bool})\\ &\qquad\quad \mid\ \mathsf{Foo}.\end{aligned}$$

Although these are not part of Haskell 98, they are supported by most compilers and interpreters for Haskell. Furthermore, generic functions cannot be applied to generalized algebraic data types (GADTs), a recent extension in GHC, of which the following type Term, representing typed terms, is an example:

$$\begin{aligned}&\textbf{data}\ \mathsf{Term} :: * \to * \ \textbf{where}\\ &\quad \mathit{Lit}\ \ :: \mathsf{Int} \to \mathsf{Term\ Int}\\ &\quad \mathit{Succ}\ :: \mathsf{Term\ Int} \to \mathsf{Term\ Int}\\ &\quad \mathit{IsZero} :: \mathsf{Term\ Int} \to \mathsf{Term\ Bool}\\ &\quad \mathit{If}\ \ :: \mathsf{Term\ Bool} \to \mathsf{Term\ a} \to \mathsf{Term\ a} \to \mathsf{Term\ a}\\ &\quad \mathit{Pair}\ :: \mathsf{Term\ a} \to \mathsf{Term\ b} \to \mathsf{Term\ (a, b)}.\end{aligned}$$

Generic Haskell is thus not fully reflexive with respect to modern extensions of Haskell.

First-class generic functions. Can a generic function take a generic function as an argument?

Generic Haskell does not have first-class generic functions. To a certain extent first-class generic functions can be mimicked by means of extending existing generic functions, but it is impossible to pass a generic function as an argument to another (generic) function. The reason for this is that generic functions in Generic Haskell are translated by means of specialization. Specialization eliminates the type arguments from the code, and specialized instances are used on the different types. Specialization has the advantage that types do not appear in the generated code, but the disadvantage that specializing higher-order generic programs becomes difficult: it is hard to determine which translated components are used where.

Multiple type arguments. Can a function be generic in more than one type argument? Induction over multiple types is for example useful when generically transforming values from one type structure into another type structure [8].

Generic functions in Generic Haskell can be defined by induction on a single type. It is impossible to induct over multiple types. If induction on multiple types is needed, the typical solution is to define two generic functions, with a universal data type in between them.

Transforming values from one type structure into another type structure is the only example we have encountered for which multiple type arguments would be useful. Hence we do not weigh this aspect heavily in our comparison.

3.3 Well-typed expressions do not go wrong

Well-typed expressions in the Hindley-Milner type system [61] do not go wrong. Does the same hold for generic functions?

Type system. Do generic functions have types?

In Generic Haskell, generic functions have explicit types. Type-correctness is only partially checked by the Generic Haskell compiler. Haskell type checks the generated code. Both Löh [54] and Hinze [26] describe a type system for Generic Haskell.

Type safety. Is the generic programming language type safe? By this we mean: is a type-correct generic function translated to a type-correct instance? And does a compiled program not crash because a non-existing instance of a generic function is called?

Generic Haskell is type safe in both aspects.

3.4 Information in types

How expressive is the type language? What does the type of a generic function reveal about the function? Can we infer a property of a generic function from its type? Since generic programming is about programming with types, questions about the type language are particularly interesting.

Type-language expressiveness. If a programming language has no types, it is impossible to define a function the behavior of which depends on a type, and hence it is impossible to define generic functions. But then, of course, there is no need for defining generic functions either. The type languages of programming languages with type systems vary widely. The less expressive a type language, the easier it becomes to write generic programs. And the more expressive a type language, the harder it becomes to write generic programs. The interesting question to ask here is: What kind of data types can be expressed in the type language?

Haskell (and hence Generic Haskell) has a very expressive type language, which can express regular data types, infinite data types, nested data types [11], data types that take type constructors as argument, etc.

The type of a generic function. Do types of generic functions in some way correspond to intuition? A generic function $f\{\!|\mathsf{a}|\!\}$ that has type $\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$ is probably a comparison function. But what does a function of type $\forall \mathsf{b} \,.\, (\forall \mathsf{a} \,.\, \mathsf{a} \to \mathsf{a}) \to \mathsf{b} \to \mathsf{b}$ do? This question is related to the possibility to infer useful properties, like free theorems [69], for a generic function from its type.

Generic Haskell's types of generic functions are relatively straightforward: a type like

$$eq\{\!|\mathsf{a} :: \star|\!\} :: (eq\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$$

is close to the type you would expect for the equality function, maybe apart from the dependency. The type for *map*:

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

is perhaps a little bit harder to understand, but playing with instances of the type of *map* for particular types, in particular for type constructors, probably helps understanding why this type is the one required by *map*.

Properties of generic functions. Is the approach based on a theory for generic functions? Do generic functions satisfy algebraic properties? How easy is it to reason about generic functions?

In his habilitations thesis [26], Hinze discusses generic programming and generic proofs in the context (of a 'core' version) of Generic Haskell. He shows a number of properties satisfied by generic functions, and he shows how to reason about generic functions.

3.5 Integration with the underlying programming language

How well does the generic programming language extension integrate with the underlying programming language? A type system can be *nominal* (based on the names of the types), *structural* (based on the structure of the types), or a mixture of the two. If a type system is nominal, it can distinguish types with exactly the same structure, but with different names. Generic functions are usually defined on a structural representation of types. Can I extend such a generic functions in a non-generic way, for example for a particular, named, data type? Or even for a particular constructor? The general question here is: how does generic programming interact with the typing system?

Using default cases, a generic function can be extended in a non-generic way in Generic Haskell. The *update* function defined in Section 2.7 provides an example. Generic functions can even be specialized for particular constructors.

3.6 Tools

Of course, a generic programming language extension is only useful if there exists an interpreter or compiler that understands the extension. Some 'light-weight' approaches to generic programming require no extra language support: the compiler for the underlying programming language is sufficient. However, most approaches require tools to be able to use them, and we can ask the following questions.

Specialization versus interpretation. Is a generic function interpreted at run-time on data types to which it is applied, or is it specialized at compile-time? The latter approach allows the optimization of generated code.

Generic Haskell specializes applications of generic functions at compile-time.

Code optimization. How efficient is the code generated for instances of generic functions?

Generic Haskell does not optimize away the extra marshaling that is introduced by the compiler for instances of generic functions. This might be an impediment for some applications.

Separate compilation. Can I use a generic function that is defined in one module on a data type defined in another module without having to recompile the module in which the generic function is defined?

Generic Haskell provides separate compilation.

Practical aspects. Does there exist an implementation? Is it maintained? On multiple platforms? Is it documented? What is the quality of the error messages given by the tool?

Generic Haskell is available on several platforms: Windows, Linux and MacOSX. The latest release is from January 14, 2005. A new release is expected in the first half of 2006. The distribution comes with a User Guide, which explains how to install Generic Haskell, how to use it, and introduces the functions that are in the library of Generic Haskell. The Generic Haskell compilers reports syntax errors. Type errors, however, are only reported when the file generated by Generic Haskell is compiled by a Haskell compiler. Type systems for Generic Haskell have been published [26, 55, 54], but not implemented.

4 Comparing approaches to generic programming

In this section we will describe seven different approaches to generic programming. We will give a brief introduction to each approach, and we will evaluate it using the criteria introduced in the previous section.

4.1 DrIFT

DrIFT [74] is a type sensitive preprocessor for Haskell. It extracts type declarations and directives from Haskell modules. The directives cause rules to be fired on the parsed type declarations, generating new code which is then appended to the bottom of the input file. An example of a directive is:

```
{-! for Foo derive : update, Show -}
```

Given such a directive in a module that defines the data type Foo, and rules for generating instances of the function *update* and the class *Show*, DrIFT generates a definition of the function *update* on the data type Foo, and an instance of *Show* for Foo. The rules are expressed as Haskell code, and a user can add new rules as required.

DrIFT comes with a number of predefined rules, for example for the classes derivable in Haskell and for several marshaling functions between Haskell data and, for example, XML, ATerm, and a binary data format.

A type is represented within DrIFT using the following data definition.

```
data Statement = DataStmt | NewTypeStmt
data Data = D{ name       :: Name           -- type name
             , constraints :: [(Class, Var)] -- constraints on type variables
             , vars       :: [Var]          -- parameters
             , body       :: [Body]         -- the constructors
             , derives    :: [Class]        -- derived classes
             , statement  :: Statement      -- data or newtype
             }
type Name      = String
type Var       = String
type Class     = String
```

A value of type Data represents one parsed data or newtype statement. These are held in a *D* constructor record. The body of a data type is represented by a value of type Body. It holds information about a single constructor.

```
data Body = Body{ constructor :: Constructor -- constructor name
                , labels      :: [Name]      -- label names
                , types       :: [Type]      -- type representations
                }
type Constructor = String
```

The definition of Type is as follows.

```
data Type = Arrow Type Type   -- function type
          | Apply Type Type   -- application
          | Var String        -- variable
          | Con String        -- constant
          | Tuple [Type]      -- tuple
          | List Type         -- list
            deriving (Eq, Show)
```

For example, the data type CharList is represented internally by:

```
reprCharList = D{     name        = "CharList"
                 ,    constraints = []
                 ,    vars        = []
                 ,    body        = [bodyNil, bodyCons]
                 ,    derives     = []
                 ,    statement   = DataStmt
                 }
bodyNil      = Body   {    constructor = "Nil"
                      ,    labels      = []
                      ,    types       = []
                      }
bodyCons     = Body   {    constructor = "Cons"
                      ,    labels      = []
                      ,    types       = [Con "Char"
                                         , Con "CharList"]
                      }
```

A rule consists of a name and a function that takes a Data and returns a document, a value of type Doc, containing the textual code of the rule for the Data value. The type Doc is defined in a module for pretty printing, and has several operators defined on it, for putting two documents beside each other (<+>) (list version *hsep*), above each other $$ (list version *vcat*), for printing texts (*text* and *texts*), etc [36]. Constructing output using pretty printing combinators is easier and more structured than manipulating strings.

Function encode. We will now explain the rules necessary for obtaining a definition of function *encode* on an arbitrary data type. For that purpose, we define the following class in our test file.

```
class Encode a where
  encode :: a → [Bit]
```

and ask DrIFT to generate instances of this class for all data types by means of the directive {-! **global** : *encode* -} . For example, for the type CharList it should generate:

```
instance Encode CharList where
  encode Nil          = [O]
  encode (Cons aa ab) = [I] ++ encode aa ++ encode ab
```

The rules for generating such instances have to be added to the file `UserRules.hs`.

```
encodefn    ::  Data → Doc
encodefn d =
  instanceSkeleton "Encode"
                   [(makeEncodefn (mkBits (body d)), empty)]
                   d
mkBits          :: [Body] → Constructor → String
mkBits bodies c = ( show
                  . intinrange2bits (length bodies)
                  . fromJust
                  . elemIndex c
                  . map constructor
                  ) bodies
```

The function *encodefn* generates an instance of the class *Encode* using the utility function *instanceSkeleton*. It applies *makeEncodefn* to each *Body* of a data type, and adds the *empty* document at the end of

the definition. The function *mkBits* takes a list of bodies, and returns a function that when given a constructor returns the list of bits for the constructor in its data type. For example, the list of bits for a data type with three constructors are $[[O,O],[O,I],[I,O]]$. The function *intinrange2bits*, which encodes a natural number in a given range as a list of bits, comes from a separate Haskell module for manipulating bits.

The function *makeEncodefn* takes an encoding function and a body, and returns a document containing the definition of function *encode* on the constructor represented by the body. If the constructor has no arguments, *encode* returns the list of bits for the constructor, obtained by means of the encoding function that is passed as an argument. If the constructor does have arguments, *encode* returns the list of bits for the constructor, followed by the encodings of the arguments of the constructor. For the argument of *encode* on the left-hand side of the definition we have to generate as many variables as there are arguments to the constructor. These variables are returned by the utility function *varNames*. Function *varNames* takes a list, and returns a list, the length of which is equal to the length of the argument list, of variable names. The constructor pattern is now obtained by prefixing the list generated by *varNames* with the constructor. This is *conPat* in the definition below. The encodings of the arguments of the constructor are obtained by prefixing the generated variables with the function *encode*, and separating the elements in the list with the list concatenation operator ++. Finally, *equals* is a utility function that returns the document containing an equality sign, '='.

```
makeEncodefn :: (Constructor → String) → (Body → Doc)
makeEncodefn enc (Body{ constructor = constructor, types = types }) =
  let bits = text (enc constructor)
      encodeText = text "encode"
      constrText = text constructor
  in let newVars = varNames types
         conPat  = parens . hsep $ constrText : newVars
         lhs     = encodeText <+> conPat
         rhs     = ( fsep
                   . sepWith (text "++")
                   . (bits:)
                   . map (λn → encodeText <+> n)
                   ) newVars
     in lhs <+> equals <+> rhs
```

Function decode. Decoding is a bit more complicated. First, we define the following class in our test file.

```
class Decode a where
  decodes    :: [Bit] → (a, [Bit])
  decode     :: [Bit] → a
  decode bits = let (a, rest) = decodes bits
                in  if null rest
                    then a
                    else error "decode: non-empty rest"
```

Then we ask DrIFT to generate instances of this class for all data types by means of the directive {-!**global**: *decode* -} . For example, for the type CharList it should generate:

```
instance Decode CharList where
  decodes (O : xs) = (Nil, xs)
  decodes (I : xs) = let (res1, xs1) = decodes xs
                     (res2, xs2) = decodes xs1
```

```
                  in  (Cons res1 res2, xs2)
  decodes []      = error "decodes"
```

The decode function generates an instance of the class *Decode*. At the end of each class instance it adds the declaration of *decodes* on the empty list.

```
decodefn    :: Data → Doc
decodefn d =
  instanceSkeleton "Decode"
                   [(mkDecodefn (mkBitsPattern (body d))
                   , text "decodes [] = error \"decodes\"")
                   ]
                   d
```

Here, function *mkBitsPattern* is almost the same as function *mkBits*, except for the way in which the list of bits is shown. We omit its definition.

The function *mkDecodefn* produces the cases for the different constructors. The left-hand side of these cases are obtained by constructing the appropriate bits pattern. The right-hand side is obtained by means of the function *decodechildren*, and returns a constructor (applied to its arguments). If a constructor has no arguments this is easy: return the constructor. If a constructor does have arguments, we first *decode* the arguments, and use the results of these decodings as arguments to the constructor.

```
mkDecodefn :: (Constructor → String) → (Body → Doc)
mkDecodefn enc body =
  let decodesText = text "decodes"
      decodechildren b =
        let nrOfArgs   = length (types b)
            argsList   = [1 .. nrOfArgs]
            listOfArgs = text "xs" : map ((text "xs"<>)  . int) argsList
            listOfRess =             map ((text "res"<>) . int) argsList
            listOfRests =            map ((text "xs"<>)  . int) argsList
            constrText = text (constructor b)
        in  if nrOfArgs == 0
            then parens (constrText <+> comma <+> text "xs")
            else text "let"
                 $$ vcat (map (nest 2)
                              (zipWith3 (λres rest arg →
                                           parens (res <+> comma <+> rest)
                                        <+> equals
                                        <+> decodesText
                                        <+> arg
                                        ) listOfRess listOfRests listOfArgs
                              )
                         )
                    $$ text "in"
                    <+> parens    (   constrText
                                  <+> hsep listOfRess
                                  <+> comma
                                  <+> text "xs"<>int nrOfArgs
                                    )
  in      decodesText
     <+> parens (text (enc (constructor body))<>text "xs")
     <+> equals
     <+> nest 6 (decodechildren body)
```

Instances of class Eq. The rules necessary for generating an instance of the class *Eq* for a data type are very similar to the rules for generating an instance of the class *Encode*. These rules are omitted, and can be found in the material accompanying these lecture notes, or in the file `StandardRules.hs` in the distribution of DrIFT.

Function map. The rules for generating instances of the *map* function on different data types differ from the rules given until now. The biggest difference is that we do not generate instances of a class. Any class definition is of the form **class** C `t` *wherer...*, in which the kind of the type `t` is fixed. So suppose we define the following class for *map*:

```
class Map t where
  map :: (a → b) → t a → t b.
```

The we can only instantiate this class with types of kind $\star \to \star$. Since the data type of generalized trees GTree has kind $\star \to \star \to \star$, we cannot represent the 'standard' *map* function on GTree by means of an instance of this class. Instead, we generate a separate *map* function on each data type. For example, on the type GTree we obtain:

```
mapGTree fa fb GEmpty       = GEmpty
mapGTree fa fb (GLeaf a)    = GLeaf (fa a)
mapGTree fa fb (GBin l v r) = GBin (mapGTree fa fb l)
                                   (fb v)
                                   (mapGTRee fa fb r)
```

The function *mapfn* generates a definition of *map* for each constructor using *mkMapfn*. The function *mkMapfn* takes as arguments the name of the data type (for generating the name of the *map* function on the data type) and the variables of the data type (for generating the names of the function arguments of *map*.

```
mapfn :: Data → Doc
mapfn (D{name = name, vars = vars, body = body}) =
  vcat (map (mkMapfn name vars) body)
```

Function *mkMapfn* creates the individual arms of the *map* function. For generating the right-hand side, it recurses over the type of the constructor in the declaration *rhsfn*.

```
mkMapfn name vars (Body{constructor = constructor, types = types}) =
  let mt name = text ("map" ++ name)
      mapArgs  = hsep (texts (map (λv → 'f' : v) vars))
      newVars  = varNames types
      conPat   = parens . hsep $ text constructor : newVars
      lhs      = mt name <+> mapArgs <+> conPat
      rhs      = hsep (text constructor
                       : map (parens . rhsfn) (zip newVars types)
                      )
      rhsfn    = λ(newVar, rhstype) →
                   case rhstype of
                     LApply t ts → hsep
                                     (mt (getName t)
                                      : hsep (map mkMapName ts)
                                      ++ [newVar]
                                     )
                     Var v       → text ('f' : v) <+> newVar
                     Con s       → mt s <+> newVar
                     List t      → text "map"
```

```
                                    <+> parens (mt (getName t)
                                                <+> mapArgs
                                               )
                                    <+> newVar
                          x       → newVar
  in lhs <+> equals <+> rhs
```

The utility functions *mkMapName* and *getName* return the name of the function to be applied to the arguments of a constructor, and the name of a type, respectively.

```
mkMapName (LApply s t) = parens (mkMapName s
                                 <+> hsep (map mkMapName t)
                                )
mkMapName (Var s)      = text ('f' : s)
mkMapName (Con s)      = text ("map" ++ s)
mkMapName (List t)     = text "map" <+> mkMapName t
mkMapName _            = error "mkMapName"

getName (LApply s t)   = getName s
getName (Var s)        = s
getName (Con s)        = s
getName (List t)       = getName t
getName _              = error "getName"
```

Evaluation

Structural dependencies. DrIFT supports the definition of functions that take the abstract syntax of a data type as an argument. From this definition it can generate any document, and it follows that it supports, in principle, the definition of type-indexed values, type-indexed types, type-indexed class instances, etc. There is no support for kind-indexed definitions, though: DrIFT does not perform any kind inference.

Full reflexivity. DrIFT is not fully reflexive with respect to the set of data types definable in Haskell 98: it cannot handle data types with type variables of higher-order kinds, such as GRose. Just as Generic Haskell, DrIFT cannot generate instances of functions on existential types or on GADTs.

We see, however, no principle reason why DrIFT cannot be fully reflexive with respect to the data types definable in Haskell 98.

First-class generic functions. Since rules are plain Haskell functions, they can take rules as arguments. First-class rules are inherited from Haskell. On the other hand, an instance of a class cannot be explicitly passed as an argument to a function or a class instance, so a rule that generates an instance of a class cannot be passed as argument to a rule that generates a function or a class instance.

Multiple type arguments. Rules cannot take multiple type arguments in DrIFT.

Type system. Rules for generic functions all have the same type in DrIFT: Data → Doc. There is no separate type system for rules; rules are ordinary Haskell functions.

Type safety. A type-correct rule does not guarantee that the generated code is type correct, as well. It is easy to define a type-correct rule that generates code that does not type-check in Haskell. DrIFT is not type safe.

Type-language expressiveness. The type language of DrIFT is Haskell's type language.

The type of a generic function. In DrIFT, every rule has type Data → Doc. Thus it is impossible to distinguish generic functions by type.

Properties of generic functions. Since rules generate pretty-printed documents (syntax), it is virtually impossible to specify properties of rules.

Integration with the underlying programming language. If a user wants to implement and use a new rule, DrIFT has to be recompiled. If a user wants to use a rule, adding a directive to a Haskell file suffices.

Specialization versus interpretation. DrIFT specializes rules on data types following directives.

Code optimization Code can be optimized by hand by specifying a more efficient rule. There need not be an efficiency penalty when using DrIFT.

Separate compilation. It is easy to use rules on data types that appear in a new module. Rules are separately compiled in DrIFT, and can then be used in any module.

Practical aspects. DrIFT is maintained. The last release is from September 2005. It runs on many platforms. The user guide explains how to use DrIFT.

No error messages are given for data types for which DrIFT cannot generate code.

4.2 PolyP

PolyP [37] is an extension of Haskell with a construct for defining so-called polytypic programs.

PolyP allows the definition of generic functions on *regular* data types of kind $\star \to \star$. A data type is regular if it does not contain function spaces, and if the arguments of the data type constructor on the left- and right-hand sides in its definition are the same. Examples of regular data types are List a, Rose a, and Fork a. The data types CharList, Tree, and GRose are regular, but have kind $\star$, $\star$, and $(\star \to \star) \to \star \to \star$, respectively. The data type Perfect a is not regular: in the right-hand side Perfect is applied to Fork a instead of a. Another example of a data type that is not regular is the data type Flip defined by **data** Flip a b = *MkFlip* a (Flip b a).

PolyP is rather similar to Generic Haskell in that it translates data types to structure representation types. The structure representation type of a data type d a is given by

Mu (FunctorOf d a),

where FunctorOf d is a type constructor of kind $\star \to \star \to \star$ representing the recursive structure of the data type d, and the data type Mu takes a type constructor and a type variable of kind $\star$, and returns the fixed point of the type constructor:

data Mu f a = *Inn* (f a (Mu f a)).

FunctorOf d is sometimes also called the *bifunctor* of d. The isomorphism between a data type and its structure representation type is witnessed by the functions *inn* and *out*.

```
inn          :: FunctorOf d a (d a) → d a
inn          = Inn
out          :: d a → FunctorOf d a (d a)
out (Inn x) = x
```

The restriction to regular data types imposed by PolyP is caused by the way the structure representation types are built up.

Here is the grammar for bifunctors:

```
f ::= g + h
    | g * h
    | Par
    | Rec
    | d@g
    | Const t
    | Empty.
```

Binary functors are sums (+, with constructors *InL* and *InR*) of products (∗, with constructor :∗:) of either the parameter type of kind ⋆ (represented by Par, with constructor *ParF* and destructor *unParF*), the data type itself (represented by Rec, with constructor *RecF* and destructor *unRecF*), compositions of data types and bifunctors (represented by @, with constructor *CompF* and destructor *unCompF*), or constant types (represented by Const t where t may be any of Float, Int, etc., with constructor *ConstF* and destructor *unConstF*). An empty product is represented by the unit type (represented by Empty). For example, for the data types List a, Rose a, and Fork a we have:

```
FunctorOf List  == Empty + Par * Rec
FunctorOf Rose == Par * List@Rec
FunctorOf Fork  == Par * Par.
```

This encoding of data types is similar to the encoding in Generic Haskell, but there is an important difference. In Generic Haskell the structure types only represent the top-level structure of a value, whereas in PolyP the encoding of values is *deep*: the original data type has disappeared in the encoded structure.

An important recursion combinator in PolyP is the *catamorphism* [59], which is defined in PolyLib, the library of PolyP. The catamorphism is a generalization of Haskell's *foldr* to an arbitrary data type. It takes an algebra as argument, and is defined in terms of a polytypic function *fmap2*, representing the action of the bifunctor of the data type on functions.

```
cata     :: Regular d ⇒ (FunctorOf d a b → b) → (d a → b)
cata alg = alg . fmap2 id (cata i) . out
```

Function *fmap2* is a polytypic function, the two-argument variant of *map*. It is defined by induction over the structure of bifunctors. It takes two functions p and r as arguments, and applies p to occurrences of the parameter, and r to occurrences of the recursive data type.

```
polytypic fmap2 :: (a → c) → (b → d) → f a b → f c d
  = λp r →
      case f of
        g + h   → (fmap2 p r) -+- (fmap2 p r)
        g * h   → (fmap2 p r) -*- (fmap2 p r)
        Empty   → λEmptyF → EmptyF
        Par     → ParF   . p .                  unParF
        Rec     → RecF   . r .                  unRecF
        d@g     → CompF . pmap (fmap2 p r) . unCompF
        Const t → ConstF .                      unConstF
```

Here -+- and -*- have the following types:

```
(-+-) :: (g a b → g c d) → (h a b → h c d) → ((g + h) a b → (g + h) c d)
(-*-) :: (g a b → g c d) → (h a b → h c d) → ((g * h) a b → (g * h) c d),
```

where + and ∗ are the internal sum and product types used by PolyP.

Function encode. Function *encode* takes an encoder for parameter values as argument, and recurses over its argument by means of a catamorphism. The algebra of the catamorphism is given by the polytypic function *fencode*. The choice between an O and an I is made, again, in the sum case. The encoder for parameter values is applied in the Par case. The other cases are standard.

```
encode        :: Regular d ⇒ (a → [Bit]) → d a → [Bit]
encode enca = cata (fencode enca)

polytypic fencode :: (a → [Bit]) → f a [Bit] → [Bit] =
  λenca →
    case f of
      g + h       → (λx → O : fencode enca x) `foldSum`
                    (λy → I : fencode enca y)
      g ∗ h       → λ(x :∗: y) → fencode enca x ++ fencode enca y
      Empty       → const []
      Par         → enca                      . unParF
      Rec         →                             unRecF
      d@g         → encode (fencode enca) . unCompF
      Const Int   → encodeInt                 . unConstF
      Const Char  → encodeChar                . unConstF

foldSum :: (g a b → c) → (h a b → c) → ((g + h) a b → c)
```

Function decode. Function *decode* is the inverse of function *encode*.

```
data Dec a = Dec ([Bit] → (a, [Bit]))

apply :: Dec a → [Bit] → (a, [Bit])
apply (Dec f) = f

instance Monad Dec where
  return x     = Dec (λbits → (x, bits))
  Dec f >>= g = Dec (λbits → let (result, rest) = f bits
                                 Dec g'         = g result
                             in  g' rest
                    )

decode       ::  Regular d ⇒ Dec a → Dec (d a)
decode deca =    liftM inn (fdecode deca (decode deca))

polytypic fdecode :: Dec a → Dec b → Dec (f a b) =
  λdeca decb →
    case f of
      g + h       →
        Dec (λbits → case bits of
                       O : bs → mapFst InL
                                  (apply (fdecode deca decb) bs)
                       I : bs → mapFst InR
                                  (apply (fdecode deca decb) bs)
            )
      g ∗ h       → fdecode deca decb >>= λx →
                    fdecode deca decb >>= λy →
                    return (x :∗: y)
      Empty       → return EmptyF
      Par         → deca                      >>= (return . ParF)
      Rec         → decb                      >>= (return . RecF)
      d@g         → decode (fdecode deca decb) >>= (return . CompF)
      Const Int   → decodeInt                 >>= (return . ConstF)
      Const Char  → decodeChar                >>= (return . ConstF)
```

where *liftM* :: *Monad* m ⇒ (a → b) → (m a → m b) lifts a function to the level of monads. Given the definition of function *encode*, the definition of function *decode* is rather standard.

The definition of the polytypic functions *eq* and *map* contain no surprises: both are similar to the definitions of function *fmap2* and *encode*, and can be found in PolyLib.

Function update. Function *update* is the same as the polytypic copy or identity function, except for a special case for the structure representation type Const Salary.

```
update :: Regular d ⇒ d a → d a
update = inn . fupdate id update . out
polytypic fupdate :: (a → b) → (c → d) → f a c → f b d =
  λp r →
    case f of
      g + h          → fupdate p r -+- fupdate p r
      g * h          → fupdate p r -*- fupdate p r
      Empty          → λEmptyF → EmptyF
      Par            → ParF    . p                          . unParF
      Rec            → RecF    . r                          . unRecF
      d@g            → CompF . update . pmap (fupdate p r) . unCompF
      Const Salary   → ConstF . updateSalary                 . unConstF
updateSalary (S f) = S (f * (1 + 0.15))
```

Evaluation

Structural dependencies. PolyP adds polytypic functions, which depend on types, to Haskell.

Full reflexivity. PolyP is not fully reflexive: polytypic functions can only be used on regular data types of kind ⋆ → ⋆. Important classes of data types for which polytypic functions do not work are mutually recursive data types and data types of kind ⋆.

First-class generic functions. Polytypic functions are not first class.

Multiple type arguments. Polytypic functions are defined by induction over a single bifunctor.

Type system. Polytypic functions are explicitly typed. The compiler checks type-correctness of polytypic functions.

Type safety. Type-correct polytypic functions are translated to type-correct Haskell functions. Forgetting an arm in the case expression of a polytypic function leads to an error when the generated Haskell is compiled or interpreted.

Type-language expressiveness. PolyP only works on regular data types of kind ⋆ → ⋆. Data types with different kinds should preferably not appear in files containing polytypic definitions. So the expressiveness of the type language is as in Haskell, but for polytypic functions, expressiveness is limited. Besides the obvious disadvantages, this has an advantage as well: since the structure of regular data types of kind ⋆ → ⋆ can be described by a bifunctor, we can define functions like the catamorphism on arbitrary data types in PolyP. The catamorphism cannot be defined in Generic Haskell. This is an instance of a general pattern: for restricted classes of data types, it might be possible to define particular generic functions that are only definable on this class.

The type of a generic function. Types of polytypic functions are direct abstractions of types on normal data types, and closely correspond to intuition.

Properties of generic functions. Jansson and Jeuring [44, 38] show how to reason about polytypic functions, and how to derive a property of a generic function from its type.

Integration with the underlying programming language. The integration of polytypic programming and Haskell is not completely seamless. PolyP does not know about classes, about types of kind other than $\star \to \star$, and lacks several syntactic constructions that are common in Haskell, such as **where** and operator sections. It is wise to separate the polytypic functions from other functions in a separate file, and only compile this file with PolyP.

Specialization versus interpretation. Polyp specializes applications of polytypic functions at compile-time.

Code optimization Like Generic Haskell, PolyP does not optimize away the extra marshaling that is introduced by the compiler for instances of generic functions. This might be an impediment for some applications.

Separate compilation. PolyP provides separate compilation.

Practical aspects. A compiler for PolyP can be downloaded. It is usable on the platforms on which GHC is available. It is not very actively maintained anymore: the latest release is from 2004. It is reasonably well documented, although not all limitations are mentioned in the documentation. PolyP's error messages can be improved.

4.3 Derivable Type Classes

Haskell's major innovation is its support for *overloading*, based on type classes. For example, the Haskell Prelude defines the class *Eq* (slightly simplified):

$$\begin{array}{l}\textbf{class } \mathit{Eq}\ \mathsf{a}\ \textbf{where}\\ \quad \mathit{eq} :: \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}\end{array}$$

This *class declaration* defines an overloaded top-level function, called *method*, whose type is

$$\mathit{eq} :: \forall \mathsf{a} \,.\, (\mathit{Eq}\ \mathsf{a}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}.$$

Before we can use *eq* on values of, say Int, we must explain how to take equality over Int values:

$$\begin{array}{l}\textbf{instance } \mathit{Eq}\ \mathsf{Int}\ \textbf{where}\\ \quad \mathit{eq} = \mathit{eqInt}.\end{array}$$

This *instance declaration* makes Int an element of the type class *Eq* and says 'the *eq* function at type Int is implemented by *eqInt*'. As a second example consider equality of lists. Two lists are equal if they have the same length and corresponding elements are equal. Hence, we require equality over the element type:

$$\begin{array}{l}\textbf{instance } (\mathit{Eq}\ \mathsf{a}) \Rightarrow \mathit{Eq}\ (\mathsf{List}\ \mathsf{a})\ \textbf{where}\\ \quad \begin{array}{ll}\mathit{eq}\ \mathit{Nil}\ \mathit{Nil} & = \mathit{True}\\ \mathit{eq}\ \mathit{Nil}\ (\mathit{Cons}\ a_2\ as_2) & = \mathit{False}\\ \mathit{eq}\ (\mathit{Cons}\ a_1\ as_1)\ \mathit{Nil} & = \mathit{False}\\ \mathit{eq}\ (\mathit{Cons}\ a_1\ as_1)\ (\mathit{Cons}\ a_2\ as_2) & = \mathit{eq}\ a_1\ a_2 \wedge \mathit{eq}\ as_1\ as_2.\end{array}\end{array}$$

This instance declaration says 'if a is an instance of *Eq*, then List a is an instance of *Eq*, as well'.

Though type classes bear a strong resemblance to generic definitions, they do not support generic programming. A type class declaration corresponds roughly to the type signature of a generic definition — or rather, to a collection of type signatures. Instance declarations are related to the type cases of a generic definition. The crucial difference is that a generic definition works for

all types, whereas instance declarations must be provided explicitly by the programmer for each newly defined data type. There is, however, one exception to this rule. For a handful of built-in classes Haskell provides special support, the so-called '**deriving**' mechanism. For instance, if you define

```
data List a = Nil | Cons a (List a) deriving (Eq)
```

then Haskell generates the 'obvious' code for equality. What 'obvious' means is specified informally in an Appendix of the language definition [65]. *Derivable type classes* (DTCs) [32] generalize this feature to arbitrary user-defined classes: generic definitions are used to specify *default methods* so that the programmer can define her own derivable classes.

Functions encode and decode. A type class usually gathers a couple of related methods. For that reason, we put *encode* and *decode* into a single class, called *Binary*.

```
class Binary a where
  encode  :: a → [Bit]
  decodes :: [Bit] → [(a, [Bit])]
```

Using two generic definitions we provide default methods for both *encode* and *decode*.

```
encode{|Unit|}    Unit      = []
encode{|b :+: c|} (Inl x)   = O : encode x
encode{|b :+: c|} (Inr y)   = I : encode y
encode{|b :*: c|} (x :*: y) = encode x ++ encode y
decodes{|Unit|}   bs        = [(Unit, bs)]
decodes{|b :+: c|} []       = []
decodes{|b :+: c|} (O : bs) = [(Inl x, cs) | (x, cs) ← decodes bs]
decodes{|b :+: c|} (I : bs) = [(Inr y, cs) | (y, cs) ← decodes bs]
decodes{|b :*: c|} bs       = [(x :*: y, ds) | (x, cs) ← decodes bs
                                             , (y, ds) ← decodes cs]
```

Incidentally, DTCs use the same structure representation types as Generic Haskell, so the corresponding definitions can be copied almost verbatim. There is one small difference though: explicit type arguments, written in curly braces, are only specified on the left-hand side of default method definitions. Elsewhere, Haskell's overloading resolution automatically determines the instance types, as for every other class method.

The function *decode* is defined in terms of *decodes*. We decided to turn the latter function into an overloaded function rather than a class method since its code is the same for all instances.

```
decode :: (Binary a) ⇒ [Bit] → a
decode bs = case decodes bs of
              [(x, [])] → x
              _         → error "decode: no parse"
```

Now, if we intend to use *encode* or *decode* on a particular type, we must first provide an instance declaration. However, by virtue of the default methods the instance declaration may be empty.

```
instance Binary CharList
instance Binary Tree
instance (Binary a) ⇒ Binary [a]
```

The compiler then automatically fills in the missing method definitions. However, if we say

```
instance (Compress a) ⇒ Compress [a] where
  encode xs   = encode (length xs) ++ concatMap encode xs
```

```
  decodes bs   = [(xs, ds) | (n, cs)  ← decodes bs
                           , (xs, ds) ← times n decodes cs]
times :: Int → ([Bit] → [(a, [Bit])]) → ([Bit] → [([a], [Bit])])
times 0       p bs = [([], bs)]
times (n + 1) p bs = [(x : xs, ds) | (x, cs) ← p bs, (xs, ds) ← times n p cs]
```

then this programmer-supplied code is used. Thus, the programmer can override the generic definition on a type-by-type basis. This ability is crucial to support *abstract types*. We can also use, indeed, we must use ordinary instance declarations to specify what a generic definition should do on primitive types such as Char or Int.

```
instance Binary Char where
  encode  = encodeChar
  decodes = decodesChar
instance Binary Int where
  encode  = encodeInt
  decodes = decodesInt
```

Function eq. The predefined *Eq* class can be thought of as a derivable type class.

```
class Eq a where
  eq, neq :: a → a → Bool
  eq{|Unit|}    Unit      Unit      = True
  eq{|b :+: c|} (Inl x)   (Inl v)   = eq x v
  eq{|b :+: c|} (Inl x)   (Inr w)   = False
  eq{|b :+: c|} (Inr y)   (Inl v)   = False
  eq{|b :+: c|} (Inr y)   (Inr w)   = eq y w
  eq{|b :*: c|} (x :*: y) (v :*: w) = eq x v ∧ eq y w
  neq x y = not (eq x y)
```

The class definition contains an ordinary default definition for inequality and a generic one for equality. Equality on characters and integers is specified using ordinary instance declarations.

```
instance Equal Char where
  eq = eqChar
instance Equal Int where
  eq = eqInt
```

Function map. Generic definitions for default methods may only be given for type classes whose type parameter ranges over types of kind $\star$. For that reason, we cannot specify a generic mapping function, There is, however, no principle hindrance in adding this feature.

Function show. The definition of *show* is similar to the one for Generic Haskell and hence omitted. Like equality, *show* is a class method of a built-in derivable type class.

Function update. We can define *update* as a variant of the generic identity, or copy function.

```
class Update a where
  update :: a → a
  update{|Unit|}    Unit      = Unit
  update{|b :+: c|} (Inl x)   = Inl (update x)
  update{|b :+: c|} (Inr y)   = Inr (update y)
```

$$update\{|\mathsf{b} :*: \mathsf{c}|\}\ (x :*: y) = update\ x :*: update\ y$$

Again, we have to provide instance declarations for all the types, on which we wish to use *update*.

```
instance Update Char where
  update = id
instance (Update a) ⇒ Update [a]
instance Update Company
instance Update Dept
instance Update SubUnit
instance Update Employee
instance Update Person
instance Update Salary where
  update (S s) = S (s ∗ (1 + 0.15))
```

All the instance declarations are trivial except the one for salary which specifies the salary increase.

Evaluation

Structural dependencies. DTCs allow the definition of $\star$-indexed functions in the style of Generic Haskell. There is no support for type-indexed data types.

Full reflexivity. DTCs share the limitations of class-based systems: higher-order kinded data types such as GRose cannot be turned into instance declarations as this requires so-called *higher-order contexts*. The original DTCs proposal recognizes this shortcoming and proposes a solutions in the form of higher-order contexts, but this extension has not been implemented yet.

First-class generic functions. Generic functions are tied to class methods. However, type classes are not first-class citizens. Consequently, generic functions are not first class either.

Multiple type arguments. Derivable type classes may only abstract over one type argument.

Type system. DTCs are fully integrated into Haskell.

Type safety. DTCs are fully type-safe. Since instance declaration must be explicitly provided, missing instances are detected at compile-time.

Type-language expressiveness. DTCs makes use of Haskell's expressive type language. However, not all types are equally usable in the context of generic functions (see 'full reflexivity').

The type of a generic function. The types are intuitive (and familiar to Haskell programmers).

Properties of generic functions. Properties of a generic function can be stated and proven as in Generic Haskell.

Integration with the underlying programming language. DTCs are fully integrated into Haskell. Only the module `Data.Generics` must be imported and the options `-fglasgow-exts`, `-fgenerics` and `-package lang` must be passed to the compiler, i.e., the Glasgow Haskell Compiler (GHC).

Specialization versus interpretation. The generic code is specialized for each instance.

Code optimization The overhead is similar to that of Generic Haskell.

Separate compilation. DTCs fully support separate compilation.

Practical aspects. The original DTCs proposal is partially implemented in GHC, the standard compiler for Haskell. A missing feature is the $c\,\mathbf{of}\,\mathsf{a}$ construct, with which one can access the names of constructors and labels. So, currently, one cannot define a generic version of *show* or *read*. The documentation is integrated into GHC's user guide (Section 7.11, "'Generic classes"'). Error messages are usually good.

4.4 Lightweight Implementation of Generics and Dynamics

Lightweight Implementation of Generics and Dynamics [13] (LIGD) is an approach to embedding generic functions and dynamic values into Haskell 98 augmented with existential types. For the purposes of these lecture notes we concentrate on the generics (which slightly simplifies the presentation). For the treatment of dynamics the interested reader is referred to the original paper [13] or to the companion lecture notes "Generic Programming, Now!", which elaborate on a closely related approach to generic programming.

A generic function in Generic Haskell is parameterized by type, essentially performing a dispatch on the type argument. The basic idea of the lightweight approach is to reflect the type argument onto the value level so that the type-case can be implemented by ordinary pattern matching. As a first try, we could, for instance, assign the generic *encode* function the type $\forall \mathsf{t} \,.\, \mathsf{Rep} \to \mathsf{t} \to [\mathsf{Bit}]$, where Rep is the type of type representations. A moment's reflection, however, reveals that this won't work. The parametricity theorem [69] implies that a function of this type must necessarily ignore its second argument. The trick is to use a parametric type for type representations: $\mathit{encode} :: \forall \mathsf{t} \,.\, \mathsf{Rep}\ \mathsf{t} \to \mathsf{t} \to [\mathsf{Bit}]$. Here $\mathsf{Rep}\ \mathsf{t}$ is the type representation of t. The one-million-dollar question is, of course, how can we define such a type?

Using a recent extension to Haskell, so-called *generalized algebraic data types*, Rep can be defined directly in Haskell, see also "Generic Programming, Now!".

```
data Rep :: ⋆ → ⋆ where
  Unit :: Rep Unit
  Int  :: Rep Int
  Sum  :: Rep a → Rep b → a :+: b
  Pair :: Rep a → Rep b → a :*: b
```

A type t is represented by a term of type $\mathsf{Rep}\ \mathsf{t}$. Note that the above declaration cannot be introduced by a Haskell 98 data declaration since none of the data constructors has result type $\mathsf{Rep}\ \mathsf{a}$.

If one wants to stick to Haskell 98 (or modest extensions thereof), one has to encode the representation type somehow. We discuss a direct encoding in the sequel and a more elaborate one in Section 4.6. The idea is to assign, for instance, *Int*, the representation of Int, the type $\mathsf{Rep}\ \mathsf{t}$ with the additional constraint that $\mathsf{t} = \mathsf{Int}$. The type equality is then encoded using the equivalence type $\mathsf{a} \leftrightarrow \mathsf{b}$ (the type is the same as the EP type of Section 2.2).

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

An element of $\mathsf{t} \leftrightarrow \mathsf{t}'$ can be seen as a 'proof' that the two types are equal. Of course, in Haskell, an equivalence pair only guarantees that t can be cast to t' and vice versa. This, however, turns out to be enough for our purposes. Figure 2 displays the full-fledged version of Rep that uses equivalence types. The constructors *Unit*, *Int*, *Char*, *Sum*, *Pair* and *Con* correspond to the type patterns Unit, Int, Char, $\cdot :+: \cdot$, $\cdot :*: \cdot$ and Con in Generic Haskell. The constructor *Type* is used for representing user-defined data types, see below.

In general, approaches to generics contain three components: code for generic values, per data type code, and shared library code. In Generic Haskell and other approaches the per data type code is not burdened upon the programmer but is generated automatically. Here the programmer is responsible for supplying the required definitions. (Of course, she or he may use tools such as DrIFT to generate the code automatically.) To see what is involved, re-consider the List data type

```
data Rep t =       Unit                  (t ↔ Unit)
           |       Int                   (t ↔ Int)
           |       Char                  (t ↔ Char)
           | ∀a b . Sum (Rep a) (Rep b)   (t ↔ (a :+: b))
           | ∀a b . Pair (Rep a) (Rep b)  (t ↔ (a :*: b))
           | ∀a .   Type        (Rep a)   (t ↔ a)
           |       Con String (Rep t)
```

Fig. 2. A type representation type.

```
data List a = Nil | Cons a (List a)
```

and recall that the structure type of `List a` is `Unit :+: (a :*: (List a))`. To turn `List a` into a *representable type*, a type on which a generic function can be used, we define

```
list   :: ∀a . Rep a → Rep (List a)
list a = Type ((Con "Nil" unit) + (Con "Cons" (a ∗ (list a))))
              (EP fromList toList)
```

where *unit*, $\cdot + \cdot$ and $\cdot * \cdot$ are smart versions of the respective constructors (defined in the LIGD library) and *fromList* and *toList* convert between the type `List` and its structure type.

```
fromList                :: ∀a . List a → Unit :+: (a :*: (List a))
fromList Nil            = Inl Unit
fromList (Cons a as)    = Inr (a :*: as)
toList                  :: ∀a . Unit :+: (a :*: (List a)) → List a
toList (Inl Unit)       = Nil
toList (Inr (a :*: as)) = Cons a as
```

Note that the representation of the structure type records the name of the constructors.

So, whenever we define a new data type and we intend to use a generic function on that type, we have to do a little bit of extra work. However, this has to be done only once.

Function encode. The definition of *encode* is very similar to the Generic Haskell definition.

```
encode :: ∀t . Rep t → t → [Bit]
encode (Unit      ep) t = case from ep t of
                            Unit → []
encode (Char      ep) t = encodeChar (from ep t)
encode (Int       ep) t = encodeInt (from ep t)
encode (Sum a b ep) t = case from ep t of
                            Inl x → O : encode a x
                            Inr y → I : encode b y
encode (Pair a b ep) t = case from ep t of
                            x :*: y → encode a x ++ encode b y
encode (Type a    ep) t = encode a (from ep t)
encode (Con s a)      t = encode a t
```

The main difference is that we have to use an explicit cast, *from ep*, to turn the second argument of type `t` into a character, an integer, and so forth. In Generic Haskell this cast is automatically inserted by the compiler.

Function decode. For *decode* we have to cast an integer etc into an element of the result type `t` using *to ep*.

```
decodes :: ∀t . Rep t → [Bit] → [(t, [Bit])]
decodes (Unit     ep) bs       = [(to ep Unit, bs)]
decodes (Char     ep) bs       = map (mapFst (to ep)) (decodesChar bs)
decodes (Int      ep) bs       = map (mapFst (to ep)) (decodesInt bs)
decodes (Sum a b  ep) []       = []
decodes (Sum a b  ep) (O : bs) = map (mapFst (to ep . Inl)) (decodes a bs)
decodes (Sum a b  ep) (I : bs) = map (mapFst (to ep . Inr)) decodes b bs)
decodes (Pair a b ep) bs       = [ (to ep (x :*: y), ds)
                                   | (x, cs) ← decodes a bs
                                   , (y, ds) ← decodes b cs]
decodes (Type a   ep) bs       = map (mapFst (to ep)) (decodes a bs)
decodes (Con s a)     bs       = decodes a bs
```

A big plus of the lightweight approach is that *encode* and *decode* are ordinary Haskell functions. We can, for instance, pass them to other functions or we can define other functions in terms of them.

```
decode :: Rep a → [Bit] → a
decode a bs = case decodes a bs of
                [(x, [])] → x
                _         → error "decode: no parse"
```

Function eq. The equality function is again very similar to the version in Generic Haskell.

```
eq :: ∀t . Rep t → t → t → Bool
eq (Int      ep) t1 t2 = from ep t1 == from ep t2
eq (Char     ep) t1 t2 = from ep t1 == from ep t2
eq (Unit     ep) t1 t2 = case (from ep t1, from ep t2) of
                           (Unit, Unit) → True
eq (Sum a b  ep) t1 t2 = case (from ep t1, from ep t2) of
                           (Inl a1, Inl a2) → eq a a1 a2
                           (Inr b1, Inr b2) → eq b b1 b2
                           _                → False
eq (Pair a b ep) t1 t2 = case (from ep t1, from ep t2) of
                           (a1 :*: b1, a2 :*: b2) → eq a a1 a2 ∧ eq b b1 b2
eq (Type a   ep) t1 t2 = eq a (from ep t1) (from ep t2)
eq (Con s a) t1 t2     = eq a t1 t2
```

Function map. The function *map* abstracts over a type constructor of kind $\star \to \star$, or is indexed by kind as in Generic Haskell. Defining such a version of *map* requires a different type representation. A discussion of the design space can be found in the companion lecture notes "Generic Programming, Now!".

Function show. The implementation of *show* is again straightforward. The constructor names can be accessed using the *Con* pattern (an analogous approach can be used for record labels).

```
shows :: ∀t . Rep t → t → ShowS
shows (Int      ep) t      = showsInt (from ep t)
shows (Char     ep) t      = showsChar (from ep t)
shows (Unit     ep) t      = showString ""
shows (Sum a b  ep) t      = case from ep t of
                               Inl a1 → shows a a1
                               Inr b1 → shows b b1
```

```
shows (Pair a b ep) t       = case from ep t of
                                (a1 :*: b1) →  shows a a1
                                               · showString " "
                                               · shows b b1
shows (Type a   ep) t       = shows a (from ep t)
shows (Con s (Unit ep)) t = showString s
shows (Con s a) t           =   showChar '('
                              · showString s
                              · showChar ' '
                              · shows a t
                              · showChar ')'
```

Since types are reflected onto the value level, we can use the full convenience of Haskell pattern matching. For instance, in the definition of *shows* we treat nullary constructors in a special way (omitting parentheses) through the use of the pattern *Con s* (*Unit ep*).

Function update. An implementation of *update* requires an extension of the Rep data type, which means that one has to modify the source of the library. Alternatively, one could turn Rep into a so-called *open data type* [56]. The code for *update* is then entirely straightforward and omitted for reasons of space.

Evaluation

Structural dependencies. LIGD supports the definition of $\star$-indexed functions in the style of Generic Haskell. Using a different representation type we can also define generic functions that are indexed by first- or higher-order kinds (this is not detailed in the original paper). Type-indexed data types are out of reach.

Full reflexivity. LIGD is in principle fully reflexive. However, to support types of arbitrary ranks, so-called *rank-n types* are required (a function has rank 2 if it takes a *polymorphic function* as an argument). Most Haskell implementations support rank-n types.

First-class generic functions. A generic function is an ordinary polymorphic Haskell function of type $\forall$t . Rep t $\rightarrow$ Poly t. As such it is first-class, assuming that rank-n functions are supported.

Multiple type arguments. Since types are reflected onto the value level, a generic function may have multiple type arguments.

Type system. LIGD is fully integrated into Haskell's type system.

Type safety. LIGD is fully type-safe. A missing type-case, however, only generates a warning at compile-time. Depending on the complexity of the 'type' patterns it may not be detected at all (in particular, if patterns are used in conjunction with guards). In this case, we get a pattern-matching failure at run-time.

Type-language expressiveness. LIGD makes full use of Haskell's expressive type language.

The type of a generic function. The types are intuitive; we only have to prefix a 'Rep t $\rightarrow$' type.

Properties of generic functions. Properties of a generic function can be stated and proven as in Generic Haskell.

Integration with the underlying programming language. LIGD is fully integrated into Haskell.

Specialization versus interpretation. Representation of types are passed and analyzed at run-time. A generic function can be seen as an interpreter.

Code optimization The run-time passing of type representations incurs a small overhead compared to Generic Haskell.

Separate compilation. LIGD supports separate compilation.

Practical aspects. The implementation of LIGD consists of a few dozen lines of code (see Appendix A of the original paper). So it can be easily integrated into one's programs and also be adapted to one's needs (for instance, if additional type cases are required).

4.5 Scrap Your Boilerplate

Scrap Your Boilerplate (SYB) [48, 50] is a library that provides combinators to build traversals and queries in Haskell. A traversal processes and selectively modifies a possibly complex data structure, whereas a query collects specific information from a data structure. Using SYB one can extend basic traversals and queries with type-specific information, thereby writing generic functions.

Generic functions in SYB are applicable to all data types of the type class *Data*. This class provides fundamental operations to consume or build values of a data type, as well as general information about the structure of a data type. All other functions are built on top of methods of the class *Data*.

A partial definition of the class *Data* is shown in Figure 3.

```
import (Typeable a) ⇒ Data a where
  toConstr    :: a → Constr
  dataTypeOf  :: a → DataType
  gfoldl      :: ∀a f .
                 (∀a b . Data a ⇒ f (a → b) → a → f b)
                 → (∀a . a → f a)
                 → a → f a
```

Fig. 3. Partial definition of the type class *Data*

The functions *toConstr* yields information about the data constructor that has constructed the given value. The data type Constr is abstract and can be queried for information such as the name of the constructor, or the data type it belongs to.

Similarly, *dataTypeOf* returns information about the data type of a value, again encapsulated in an abstract data type DataType.

The function *gfoldl* is a very general function that allows the destruction of a single input value – the third argument – of type a into a result of type f a. Almost any Haskell value is an application of a data constructor to other values. This is the structure that *gfoldl* works on. If a value v is of the form

$$C\ v_1\ v_2\ ...\ v_n$$

then *gfoldl* $(\diamond)$ c v is

$$(\cdots\ ((c\ C \diamond v_1) \diamond v_2) \diamond \cdots \diamond v_n).$$

The second argument c is applied to the data constructor C, and each application is replaced by the first argument $(\diamond)$. In particular,

$$unId \,.\, gfoldl\ (\lambda x\ y \to Id\ (unId\ x\ y))\ Id$$

is the identity on types of class *Data*. Here, the auxiliary type

$$\textbf{newtype}\ \mathsf{Id\ a} = \mathsf{Id}\{\, unId :: \mathsf{a}\}$$

is used, because the result type of $\mathsf{f\ a}$ of *gfoldl* can be instantiated to $\mathsf{Id\ a}$, but not directly to a in Haskell. If we could, then

$$gfoldl\ (\$)\ id$$

would be the identity, making the role of *gfoldl* more obvious.

With the help of *gfoldl*, a basic query combinator can be defined, which also forms part of the SYB library:

$$gmapQ :: \forall \mathsf{a} \,.\, Data\ \mathsf{a} \Rightarrow (\forall \mathsf{b} \,.\, Data\ \mathsf{b} \Rightarrow \mathsf{b} \to \mathsf{c}) \to \mathsf{a} \to [\mathsf{c}].$$

A call $gmapQ\ q\ x$ takes a query q (of type $\forall \mathsf{b} \,.\, Data\ \mathsf{b} \Rightarrow \mathsf{b} \to \mathsf{c}$) and applies it to the immediate subterms of q, collecting the results in a list.

Function encode. A good example of a function using *gmapQ* is the function *encode*, which can be written using the SYB library as follows:

$$\begin{aligned} &encode :: Data\ \mathsf{a} \Rightarrow \mathsf{a} \to [\mathsf{Bit}] \\ &encode\ x = concat\ (encodeConstr\ (toConstr\ x) : gmapQ\ encode\ x). \end{aligned}$$

The function *encodeConstr* takes the current constructor and encodes it as a list of bits:

$$\begin{aligned} &encodeConstr :: \mathsf{Constr} \to [\mathsf{Bit}] \\ &encodeConstr\ c = intinrange2bits\ (maxConstrIndex\ (constrType\ c)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad (constrIndex\ c - 1). \end{aligned}$$

As before, we use the utility function *intinrange2bits* to encode a natural number in a given range. In *encode*, the constructor for the current value x is encoded, and we use *gmapQ* to recursively encode the subterms of x.

With *encode*, we can for instance encode booleans, lists, trees etc: we have a generic function. However, the default behavior is unsuitable for handling base types such as Int and Char. If we want to use type-specific behavior such as *encodeInt* and *encodeChar*, the SYB library allows us to extend a query with a type-specific case, using *extQ*:

$$extQ :: \forall \mathsf{a\ b\ c} \,.\, (Typeable\ \mathsf{a}, Typeable\ \mathsf{b}) \Rightarrow (\mathsf{a} \to \mathsf{c}) \to (\mathsf{b} \to \mathsf{c}) \to (\mathsf{a} \to \mathsf{c}).$$

This function makes use of run-time type information which is encapsulated in the type class *Typeable* and available for all types in *Data*, as *Typeable* is a superclass of *Data*. It is essentially a one-branch *type-case*. Using *extQ*, we can write *encode* with type-specific behavior for Ints and Chars:

$$\begin{aligned} &encode :: Data\ \mathsf{a} \Rightarrow \mathsf{a} \to [\mathsf{Bit}] \\ &encode = (\lambda x \to concat\ (encodeConstr\ (toConstr\ x) : gmapQ\ encode\ x)) \\ &\qquad `extQ`\ encodeInt \\ &\qquad `extQ`\ encodeChar. \end{aligned}$$

Note that we cannot reuse the previously defined version of *encode* in this new definition, because the recursive call to *encode* that appears as an argument to *gmapQ* must point to the extended function.

Function decode. The *gfoldl* combinator is only suitable for *processing* values. In order to write a generic *producer* such as *decode*, a different combinator is required. The *Data* class provides one, called *gunfold*:

```
gunfold :: ∀a f .
             (∀a b . Data a ⇒ f (a → b) → f b)
           → (∀a . a → f a)
           → Constr → f a.
```

If $d :: \mathsf{Constr}$ is the constructor information for the data constructor C, which takes n arguments, then *gunfold app c d* is

$$app\ (\cdots\ (app\ (c\ C))\ \cdots),$$

thus *app* applied n times to $c\ C$. As with *gfoldl*, SYB provides several combinators built on top of *gunfold*, the most useful being *fromConstrM*, that monadically constructs a value of a certain constructor:

```
fromConstrM :: ∀a f . (Data a, Monad f) ⇒ (∀b . Data b ⇒ f b) → Constr → f a
fromConstrM p = gunfold (`ap`p) return
```

Here, $ap :: \forall a\ b\ f\ .\ Monad\ f \Rightarrow f\ (a \to b) \to f\ a \to f\ b$ is lifted function application.

Using *fromConstrM*, we can define *decodes*, a monadic version of *decode* that keeps the "current" list of bits as state in a state monad:

```
decodes :: Data a ⇒ State [Bit] a
decodes    = decodes' ⊥
               `extR` decodesInt
               `extR` decodesChar
  where
    decodes'          :: Data a ⇒ a → State [Bit] a
    decodes' dummy =
      do let d    = dataTypeOf dummy
             l    = length (int2bits (length (dataTypeConstrs d) − 1))
         c ← consume l
         let con = decodeConstr c d
         fromConstrM decodes con.
```

A few remarks are in order. The function *decodes* calls *decodes′* with $\bot$. This is a convenient way to obtain a value of the result type a, so that we can apply *dataTypeOf* to it. The function *decodes′* reads in l bits from the input via *consume*, interprets these bits as a constructor *con* using *decodeConstr*, and finally employs *fromConstrM* to decode the children of the constructor recursively. Both *consume* and *decodeConstr* are easy to define. Type-specific behavior for integers and characters is added to the function using the SYB extension operator *extR*, which plays a role analogous to *extQ*, in the context of monadic generic producers:

```
extR :: ∀a b f . (Monad f, Typeable a, Typeable b) ⇒ f a → f b → f a.
```

From *decodes*, we get *decode* in the obvious way:

```
decode :: Data a ⇒ [Bit] → a
decode bs = case runState decodes bs of
              (r, []) → r
              _       → error "decode: no parse".
```

Function eq. The definition of generic equality in SYB is simple, but requires yet another combinator:

```
eq :: Data a ⇒ a → a → Bool
eq = eq'
eq' :: (Data a, Data b) ⇒ a → b → Bool
eq' x y = toConstr x == toConstr y ∧
          and (gzipWithQ eq' x y).
```

The function *eq* is a type-restricted variant of *eq′*, which accepts two arguments of potentially different types. The constructors of the two values are compared, and *gzipWithQ* is used to pairwise compare the subterms of the two values recursively.

The combinator *gzipWithQ* is a two-argument variant of *mapQ*. It is a bit tricky to define, but it can be defined in terms of *gfoldl*.

Note that *eq′* requires the relaxed type, because the subterms of x and y only have compatible types if they really are of the same data constructor. If we compare unequal values, we are likely to get incompatible types sooner or later.

If the trick to relax the type of the function is not available for a two-argument generic function , an alternative solution is to use the dynamically available type information from class *Typeable* to define a unification function

```
unify :: (Typeable a, Typeable b) ⇒ Maybe (a → b).
```

Function map. A generic function such as *map* that abstracts over a type constructor cannot be defined using SYB, because the *Data* class contains only types of kind $*$. It is possible to define variants of *map*, such as traversals that increase all integers in a complex data structure, but it isn't possible to define a function of type

```
∀a b f . (a → b) → f a → f b,
```

where the arguments of the container type f are modified, and the function is parametrically polymorphic in a and b (cf. the section on "SYB Revolutions" below)

Function show. We define *show* in two steps, as we have done in the Generic Haskell case. The function *showP* takes an additional string transformer that encodes whether to place surrounding parentheses on non-atomic expressions or not.

We have already seen how constructor information can be accessed in the definition of *encode*. Therefore, the definition of *showP* does not come as a surprise:

```
showP :: Data a ⇒ (String → String) → a → String
showP p = (λx → showApp (showConstr (toConstr x))
                        (gmapQ ((++) " " . showP parens) x))
          `ext1Q` showList
          `extQ`  (Prelude.show :: String → String)
  where
    parens x = "(" ++ x ++ ")"
    showApp :: String → [String] → String
    showApp x [] = x
    showApp x xs = p (concat (x : xs))
    showList :: Data a ⇒ [a] → String
    showList xs =
      "[" ++ concat (intersperse "," (map (showP id) xs)) ++ "]"
```

We feed each constructor application to *showApp*. On atomic subexpressions, *showApp* never produces parentheses, otherwise it consults p.

The most interesting part is how to define type-specific behavior for lists and strings. Placing strings between double quotes is achieved by the standard Haskell *show* function using the *extQ* extension operator. However, The more general syntactic sugar for lists (placed between square brackets, elements separated by commas) is not achieved so easily, because *showList* is a polymorphic function, and *extQ* only works if the second argument is of monomorphic type. SYB therefore provides a special, polymorphic, extension operator

$$\begin{aligned} &\mathit{ext1Q} :: \forall \mathsf{a}\ \mathsf{c}\,.\,(\mathit{Typeable1}\ \mathsf{f}, \mathit{Data}\ \mathsf{a}) \Rightarrow \\ &\qquad (\mathsf{a} \to \mathsf{c}) \to (\forall \mathsf{b}\,.\,\mathit{Data}\ \mathsf{b} \Rightarrow \mathsf{f}\ \mathsf{b} \to \mathsf{c}) \to (\mathsf{a} \to \mathsf{c}) \end{aligned}$$

Note that polymorphic extension requires a separate operator for each kind, and also a separate variant of the cast operation: the run-time type information of the type constructor f of kind $* \to *$ is made available using the type class *Typeable1* rather than *Typeable*.

Function update. Traversals that update a large heterogeneous data structure in selective places were one of the main motivations for designing SYB, therefore it isn't surprising that defining such a traversal is extremely simple:

$$\begin{aligned} &\mathit{update} :: \mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{a} \\ &\mathit{update} = \mathit{everywhere}\ (\mathit{id}\ \text{`}\mathit{extT}\text{`}\ (\lambda(S\ s) \to S\ (s * (1 + 0.15)))). \end{aligned}$$

The argument to *everywhere* is the identity function, extended with a type-specific case for the type Salary. The function *everywhere* is a SYB combinator that applies a function at any point in a data structure. It is defined in terms of

$$\mathit{gmapT} :: \forall \mathsf{a}\,.\,\mathit{Data}\ \mathsf{a} \Rightarrow (\forall \mathsf{b}\,.\,\mathit{Data}\ \mathsf{b} \Rightarrow \mathsf{b} \to \mathsf{b}) \to (\mathsf{a} \to \mathsf{a}),$$

a variant of *gmapQ* that applies a given generic function to the immediate subterms of a value. The *gmapT* in turn can again be defined using *gfoldl*.

Derived work: SYB with Class. Lämmel and Peyton Jones have shown [49] that using type classes rather than run-time type casts can make generic programming using SYB more flexible. Their work aims at replacing SYB extension operators such as *extQ* and *extR*: each generic function is then defined as a class with a default behavior, and type-specific behavior can be added by defining specific instances of the class.

This approach is a bit more verbose, but has a significant advantage: instances of classes can be added in a modular way, also at a later stage. There is always the possibility to extend an already existing generic function with new behavior, without modification of already written code.

Derived work: SYB Reloaded and Revolutions. In their SYB Reloaded and Revolutions papers, Hinze, Löh and Oliveira [34, 33] demonstrate that SYB's *gfoldl* function is in essence a catamorphism on the Spine data type, which can be defined as follows:

$$\begin{aligned} &\textbf{data}\ \mathsf{Spine}\ a\ \textbf{where} \\ &\quad \mathit{Constr} :: \mathsf{Constr} \to \mathsf{a} \to \mathsf{Spine\ a} \\ &\quad (\diamond) \qquad :: \mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{Spine}\ (\mathsf{a} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{Spine\ b}. \end{aligned}$$

Furthermore, a "type spine" type is given as a replacement for *gunfold*, and a "lifted spine" type for generic functions that are parameterized over type constructors. For example, using the lifted spine type, *map* can be defined.

Evaluation

Structural dependencies. SYB allows the definition of generic functions. There is no support for defining type-indexed data types.

Full reflexivity. The SYB approach is not fully reflexive. Generic functions are only applicable to data types for which a *Typeable* instance can be specified .

Type-specific behavior is only possible for types of kind $*$.

First-class generic functions. In SYB, generic functions are normal polymorphic Haskell functions, and as such are first-class under the precondition of arbitrary-rank polymorphism.

Multiple type arguments. There is no restriction on the number of type arguments that a generic function can have in SYB, although the basic combinators are tailored for functions of the form

$$\mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow \ldots$$

that consume a single value.

Type system. SYB is completely integrated in Haskell's type system.

Type safety. SYB is type-safe, but type-specific extensions of generic functions rely on run-time type casting via the *Typeable* class. It is possible for a user to break type safety by defining bogus instances for the *Typeable* class. The implementation could be made more robust if user-defined instances of class *Typeable* would not be allowed, and all *Typeable* instances would be derived automatically by the compiler.

Type-language expressiveness. SYB makes use of Haskell's expressive type language. However, not all types are equally usable in the context of generic functions (see "full reflexivity").

The type of a generic function. Types of generic functions have one or more constraints for the *Data* class. The types are intuitive.

Properties of generic functions. The use of type classes *Data* and *Typeable* at the basis of SYB makes proving properties relatively difficult. Instances for these classes can be generated automatically, but automatic generation is only described informally. User-defined instances of these classes can cause unintended behavior. There is no small set of fundamental data types (such as Generic Haskell's unit, binary sum, and binary pair types) to which Haskell data types are reduced. Lämmel and Peyton Jones state a few properties of basic SYB combinators in the original paper, but provide no proof. The only work we are aware of trying to prove properties about SYB is of Reig [67], but he translates SYB combinators into Generic Haskell in order to do so.

Integration with the underlying programming language. SYB is fully integrated into Haskell. The module `Data.Generics` contains all SYB combinators. The options `-fglasgow-exts` is required for GHC to support the higher-ranked types of some of the SYB combinators.

Specialization versus interpretation. The SYB approach makes use of run-time type information. Generic functions have *Data* class constraints. Most Haskell compilers implement type classes using *dictionary passing*: for each *Data* constraint, a record containing the appropriate class methods is passed along at run-time. The *Data* is a subclass of *Typeable*, which provides the actual structure of the type at run-time. This information is used to provide run-time type casts to enable type-specific behavior.

Code optimization As SYB is a Haskell library, the code is not optimized in any special way. The implementation of generic functions is relatively direct, the only overhead is due to the passing of class dictionaries and the use of many higher-order functions.

Separate compilation. Generic functions are normal Haskell functions, and can be placed in different modules and compiled separately. Generic functions themselves are not extensible, however. If new specific cases must be added to a generic function, the whole definition has to be repeated. This restriction is lifted by "SYB with Class".

Practical aspects. SYB is shipped as a library with current releases of GHC and supported. It is planned to provide the functionality of "SYB with Class" in future releases of GHC. The Spine data type from "SYB Reloaded" is not yet used in the official release, but might be integrated in the future.

4.6 Generics for the masses

Generics for the masses [28, 23] (GM) is similar in spirit to LIGD. The approach shows that one can program generically within Haskell 98 obviating to some extent the need for fancy type systems or separate tools. Like LIGD, generics for the masses builds upon an encoding of the type representation type *Rep*, this time a class-based one. The details of the encoding are not relevant here; the interested reader is referred to the journal paper [23].

Function encode. To define a generic function the generic programmer has to provide a signature and an implementation. Rather unusually, the type of a generic function is specified using a **newtype** declaration.

```
newtype Encode a = Encode{ applyEncode :: a → [Bit] }
```

We already know that the generic function *encode* cannot be a genuine polymorphic function of type $\mathsf{a} \to [\mathsf{Bit}]$. Data compression does not work for arbitrary types, but only for types that are *representable*. Representable means that the *type* can be represented by a certain *value*. Here a type representation is simply an overloaded value called *rep*. The first part of the generic compression function is then given by the following definition.

```
encode :: (Rep a) ⇒ a → [Bit]
encode = applyEncode rep
```

Loosely speaking, we apply the generic function to the type representation *rep*. Of course, this is not the whole story. The code above defines only a convenient shortcut. The actual definition of *encode* is provided by an instance declaration, but you should read it instead as just a generic definition.

```
instance Generic Encode where
  unit   = Encode (λx → [])
  plus   = Encode (λx → case x of   Inl l → O : encode l
                                    Inr r → I : encode r)
  pair   = Encode (λx → encode (outl x) ++ encode (outr x))
  datatype descr iso
         = Encode (λx → encode (fromData iso x))
  char   = Encode (λx → encodeChar x)
  int    = Encode (λx → encodeInt  x)
```

Most of the cases are familiar — just read the method definitions as type cases. To encode an element of an arbitrary data type, we first convert the element into a sum of products, which is then encoded. That said is becomes clear that GM uses the same structure types as Generic Haskell. The function *fromData* is an accessor of the data type Iso (which is the same as the EP type of Section 2.2).

```
data Iso a b = Iso{fromData :: b → a, toData :: a → b}
```

That's it, at least, as far as the generic function is concerned. Before we can actually compress data to strings of bits, we first have to turn the types of the to-be-compressed values into representable types. Consider as an example the type of binary leaf trees.

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

We have to show that this type is representable. To this end we exhibit an isomorphic type built from representable type constructors. This is the familiar *structure type* of Tree, denoted Tree′.

```
type Tree′ a = (Constr a) :+: (Constr ((Tree a) :*: (Tree a)))
```

The main work goes into defining two mappings, *fromTree* and *toTree*, which certify that Tree a and its structure type Tree′ a are indeed isomorphic.

```
fromTree                        :: Tree a → Tree′ a
fromTree (Leaf x)               = Inl (Con x)
fromTree (Fork l r)             = Inr (Con (l :*: r))
toTree                          :: Tree′ a → Tree a
toTree (Inl (Con x))            = Leaf x
toTree (Inr (Con (l :*: r)))    = Fork l r
```

The *Con* constructor just marks the position of the original data constructors *Leaf* and *Fork*. The isomorphism is then used to turn Tree into a representable type.

```
instance (Rep a) ⇒ Rep (Tree a) where
  rep = datatype ("Leaf" ./ 1 .| "Fork" ./ 2)   -- syntax
                 (Iso fromTree toTree)          -- semantics
```

The declaration specifies the syntax — name and arity of the constructors — and the semantics — the structure — of the tree data type. Such a declaration has to be provided once per data type and is used for all the generic functions.

For reference, Figure 4 lists the definition of the class *Generic* (g is the type of a generic function).

```
class Generic g where
  unit     ::                                          g Unit
  plus     ::  (Rep a, Rep b) ⇒                        g (a :+: b)
  pair     ::  (Rep a, Rep b) ⇒                        g (a :*: b)
  datatype ::  (Rep a) ⇒ DataDescr → Iso a b → g b
  char     ::                                          g Char
  int      ::                                          g Int
  list     ::  (Rep a) ⇒                               g [a]
  constr   ::  (Rep a) ⇒                               g (Constr a)
  list     =   datatype ("[]" ./ 0 .| ":" ./ 2) (Iso fromList toList)
  constr   =   datatype ("Con" ./ 1)            (Iso arg Con)
data DataDescr =  NoData
               |  DataDescr{name :: String,      arity :: Int}
               |      Alt   {getl  :: DataDescr, getr  :: DataDescr}
infix  2 ./
infixr 1 .|
f ./ n   = DataDescr{name = f,  arity = n}
d1 .| d2 = Alt       {getl  = d1, getr  = d2}
newtype Constr a     =     Con{arg :: a}
```

Fig. 4. The class *Generic*.

Function decode. The definition of *decodes* follows exactly the same scheme.

```
newtype Decodes a = Decodes{ applyDecodes :: [Bit] → [(a, [Bit])] }
decodes :: (Rep a) ⇒ [Bit] → [(a, [Bit])]
decodes = applyDecodes rep
instance Generic Decodes where
  unit  = Decodes (λbs → [(Unit, bs)])
  plus  = Decodes (λbs → case bs of [] → []
                                    O : bs → map (mapFst Inl) (decodes bs)
                                    I : bs → map (mapFst Inr) (decodes bs))
  pair  = Decodes (λbs → [(x :*: y, ds) | (x, cs) ← decodes bs
                                        , (y, ds) ← decodes cs])
  datatype descr iso
        = Decodes (λbs → map (mapFst (toData iso)) (decodes bs))
  char  = Decodes (λbs → decodesChar bs)
  int   = Decodes (λbs → decodesInt  bs)
```

It is worth noting that Haskell's overloading resolution automatically determines the instance types: we just call *decodes* rather than *decodes*{|t|}.

The function *decode* can easily be defined in terms of *decodes*.

```
decode :: (Rep a) ⇒ [Bit] → a
decode a bs = case decodes a bs of
                [(x, [])] → x
                _         → error "decode: no parse"
```

Note that the class context only records that *decode* depends on some generic function. This is in sharp contrast to DTC where the context precisely records, on which overloaded function(s) *decode* depends: (*Binary* a) ⇒ [Bit] → a.

Function eq. The definition of *eq* is straightforward.

```
newtype Equal a = Equal{ applyEqual :: a → a → Bool }
eq :: (Rep a) ⇒ a → a → Bool
eq = applyEqual rep
instance Generic Equal where
  unit  = Equal (λx1 x2 → True)
  plus  = Equal (λx1 x2 → case (x1, x2) of
                            (Inl a1, Inl a2) → eq a1 a2
                            (Inr b1, Inr b2) → eq b1 b2
                            _ → False)
  pair  = Equal (λx1 x2 → eq (outl x1) (outl x2) ∧ eq (outr x1) (outr x2))
  datatype descr iso
        = Equal (λx1 x2 → eq (fromData iso x1) (fromData iso x2))
  char  = Equal (λx1 x2 → x1 == x2)
  int   = Equal (λx1 x2 → x1 == x2)
```

Function map. The function *map* cannot be defined using the *Generic* class that we have employed for *encode* and *decode*. Rather, we need a new tailor-made class that allows us to define generic functions whose type is parameterized by *two* type arguments (see Section 2.5). The definition is then very similar to what we have seen before.

```
newtype Map a1 a2 = Map{ applyMap :: a1 → a2 }
instance Generic Map where
  unit  = Map (λx → x)
```

```
plus a b = Map (λx → case x of Inl l  → Inl (applyMap a l)
                               Inr r → Inr (applyMap b r))
pair a b = Map (λx → applyMap a (outl x) :*: applyMap b (outr x))
datatype iso1 iso2 a
         = Map (λx → toData iso2 (applyMap a (fromData iso1 x)))
char     = Map (λx → x)
int      = Map (λx → x)
```

Using *frep*, the representation of types of kind $\star \to \star$, we can define a generic version of Haskell's *fmap*.

```
fmap   :: (FRep f) ⇒ (a1 → a2) → (f a1 → f a2)
fmap f = applyMap (frep (Map f))
```

Function show. To implement *show* we have to access the syntax of data constructors. To this end, we extends *shows′* by an additional argument of type DataDescr that provides information about the syntax of the to-be-printed value. This argument is initialized to *NoData*, because initially we have no information.

```
shows :: (Rep a) ⇒ a → ShowS
shows = shows′ NoData
```

In the *datatype* case, which signals that the current argument is an element of some data type, we use the first argument of *datatype* as the new syntax description.

```
newtype Shows′ a = Shows′{ applyShows′ :: DataDescr → a → ShowS }
shows′   :: (Rep a) ⇒ DataDescr → a → ShowS
shows′   = applyShows′ rep
instance Generic Shows′ where
  unit   = Shows′ (λd x → showString "")
  plus   = Shows′ (λd x → case x of  Inl l  → shows′ (getl  d) l
                                     Inr r → shows′ (getr d) r)
  pair   = Shows′ (λd x → shows (outl x) · showChar ' ' · shows (outr x))
  char   = Shows′ (λd x → showsChar x)
  int    = Shows′ (λd x → showsInt  x)
  list   = Shows′ (λd x → showsl shows x)
  datatype descr iso
         = Shows′ (λd x → shows′ descr (fromData iso x))
  constr = Shows′ (λd x → if arity d == 0 then
                            showString (name d)
                          else
                            showChar '(' · showString (name d) · showChar ' '
                              · shows (arg x) · showChar ')')
```

The implementation of *shows′* has a special case for lists which are converted to Haskell list syntax, with brackets and commas. The helper function *showsl* does the main work.

```
showsl :: (a → ShowS) → ([a] → ShowS)
showsl p []       = showString "[]"
showsl p (a : as) = showChar '[' · p a · rest as
  where rest []       = showChar ']'
        rest (x : xs) = showChar ',' · p x · rest xs
```

Function update. An implementation of *update* requires an extension of the class *Generic*, which means that one has to modify the source of the library. An alternative approach based on subclasses is described in a recent paper [64].

Evaluation

Structural dependencies. GM supports the definition of generic functions on types and type constructors. For each brand of generic functions a tailor-made *Generic* class must be used. Because of the class-based encoding the code looks somewhat different to that of Generic Haskell. The difference is, however, only superficial.

Full reflexivity. GM is in principle fully reflexive. Rank-n types are required in order to support types of higher kinds. Furthermore, if one wants to use the convenience of the *Rep* class, one additionally needs higher-order contexts, see the evaluation of DTCs.

First-class generic functions. A generic function is an ordinary polymorphic Haskell function of type $\forall \mathsf{t} \,.\, (\mathit{Rep}\ \mathsf{t}) \Rightarrow \mathsf{Poly\ t}$. In a language with rank-n types, generic functions are consequently first-class citizens.

Multiple type arguments. GM also supports multiple type arguments (through a nested type case).

Type system. GM is fully integrated into Haskell's type system.

Type safety. GM is fully type-safe. A missing case branch issues a warning at compile-time (about a missing method).

Type-language expressiveness. GM makes full use of Haskell's expressive type language. The caveats of DTCs also apply here.

The type of a generic function. The types are intuitive; we only have to prefix a '$(\mathit{Rep}\ \mathsf{t}) \Rightarrow$' context.

Properties of generic functions. Properties of a generic function can be stated and proven as in Generic Haskell.

Integration with the underlying programming language. GM is fully integrated into Haskell.

Specialization versus interpretation. Instances of generic functions are assembled at compile-time.

Code optimization The overhead is similar to that of Generic Haskell. The code quality possibly depends a bit more on GHC's optimizer.

Separate compilation. GM supports separate compilation.

Practical aspects. GM comprises three major implementations of generics and a few variations. The approach is extremely lightweight; each implementation consists of roughly two dozen lines of Haskell code. It is less suited as a library (unless one makes do with the predefined types cases), but it can easily be adopted to one's needs.

4.7 Clean

Clean's generic programming extension [3, 2] is just as Generic Haskell based on Hinze's work on type-indexed functions with kind-indexed types [27].

The language of data types in Clean is very similar to that of Haskell, and the description from Section 2.1 on how to convert between data types and their structural representations as binary sums of binary products applies to Clean as well, only that the unit type is called UNIT, the sum type EITHER, and the product type PAIR. There are special structural markers for constructors and record field names called CONS and FIELD, and one for objects called OBJECT.

Clean's generic functions are integrated with its type class system. Each generic function defines a kind-indexed family of type classes, the generic function itself being the sole method of these classes. Let us look at an example.

Function encode. Here is the code for the generic function *encode*.

```
generic encode a :: a → [Bit]
encode{|UNIT|}                 UNIT          = []
encode{|Int|}                  i             = encodeInt i
encode{|Char|}                 c             = encodeChar c
encode{|EITHER|}  enc_a enc_b (LEFT x)       = [O : enc_a x]
encode{|EITHER|}  enc_a enc_b (RIGHT y)      = [I : enc_b y]
encode{|PAIR|}    enc_a enc_b (PAIR x y)     = enc_a x ++ enc_b y
encode{|CONS|}    enc_a       (CONS x)       = enc_a x
encode{|FIELD|}   enc_a       (FIELD x)      = enc_a x
encode{|OBJECT|}  enc_a       (OBJECT x)     = enc_a x
derive encode Tree
```

The keyword **generic** introduces the type signature of a generic function, which takes the same form as a type signature in Generic Haskell, but without dependencies. Each generic function automatically depends on itself in Clean, and in the cases for types of higher kinds such as EITHER:: $* \to * \to *$ or CONS :: $* \to *$, additional arguments are passed to the generic function representing the recursive calls. This is very close to Hinze's theory [27] which states that the type of *encode* is based on the kind of the type argument as follows:

$$
\begin{aligned}
&\mathit{encode}\{\!|\mathsf{a} :: \kappa|\!\} :: \mathit{Encode}\{\!|\kappa|\!\}\ \mathsf{a}\\
&\mathit{Encode}\{\!|*|\!\} \qquad \mathsf{a} = \mathsf{a} \to [\mathsf{Bit}]\\
&\mathit{Encode}\{\!|\kappa \to \kappa'|\!\}\ \mathsf{a} = \forall \mathsf{b} :: \kappa\,.\,\mathit{Encode}\{\!|\kappa|\!\}\ \mathsf{b} \to \mathit{Encode}\{\!|\kappa'|\!\}\ (\mathsf{a}\ \mathsf{b}).
\end{aligned}
$$

In particular, if we instantiate this type to the kinds $*$, $* \to *$, and $* \to * \to *$, we get the types of the UNIT, EITHER, CONS cases of the definition of *encode*, respectively:

$$
\begin{aligned}
&\mathit{encode}\{\!|\mathsf{a} :: *|\!\} && :: \mathsf{a} \to [\mathsf{Bit}]\\
&\mathit{encode}\{\!|\mathsf{f} :: * \to *|\!\} && :: (\mathsf{a} \to [\mathsf{Bit}]) \to (\mathsf{f}\ \mathsf{a} \to [\mathsf{Bit}])\\
&\mathit{encode}\{\!|\mathsf{f} :: * \to * \to *|\!\} && :: (\mathsf{a} \to [\mathsf{Bit}]) \to (\mathsf{b} \to [\mathsf{Bit}]) \to (\mathsf{f}\ \mathsf{a}\ \mathsf{b} \to [\mathsf{Bit}]).
\end{aligned}
$$

The **derive** statement is an example of how generic behavior must be explicitly derived for additional data types. If Tree is a type that we want to encode, we have to request this using a **derive** statement.

Because generic functions automatically define type classes in Clean, the type arguments (but not the kind arguments) can usually be inferred automatically. The function *encode* can thus be invoked on a tree $t ::$ Tree by calling $\mathit{encode}\{\!|*|\!\}\ t$.

If $\mathit{encode}\{\!|*|\!\}\ x$ is used in another function on a value $x ::$ a, then a class constraint of the form $\mathit{encode}\{\!|*|\!\}\ \mathsf{a}$ arises and is propagated as usual. Other first-order kinds can be passed to *encode*, but Clean does not currently support generic functions on higher-order kinds.

Functions decode, eq, map and show. Apart from the already mentioned differences and a few syntactic differences between the Clean and Haskell languages, many of the other example functions can be implemented exactly as in Generic Haskell. We therefore present only *map* as another example:

generic *map* a b :: a → b

$$
\begin{array}{llll}
map\{|\mathsf{UNIT}|\} & & x & = x \\
map\{|\mathsf{Int}|\} & & i & = i \\
map\{|\mathsf{Char}|\} & & c & = c \\
map\{|\mathsf{EITHER}|\} & map_a\ map_b & (LEFT\ x) & = LEFT\ (map_a\ x) \\
map\{|\mathsf{EITHER}|\} & map_a\ map_b & (RIGHT\ y) & = RIGHT\ (map_b\ y) \\
map\{|\mathsf{PAIR}|\} & map_a\ map_b & (\mathsf{PAIR}\ x_1\ x_2) & = \mathsf{PAIR}\ (map_a\ x_1)\ (map_b\ x_2) \\
map\{|\mathsf{OBJECT}|\} & map_a & (\mathsf{OBJECT}\ x) & = \mathsf{OBJECT}\ (map_a\ x) \\
map\{|\mathsf{FIELD}|\} & map_a & (\mathsf{FIELD}\ x) & = \mathsf{FIELD}\ (map_a\ x) \\
map\{|\mathsf{CONS}|\} & map_a & (\mathsf{CONS}\ x) & = \mathsf{CONS}\ (map_a\ x)
\end{array}
$$

Function update. A function such as *update* cannot currently be defined in Clean, because there is no support for higher-order generic functions, nor are there default cases as in Generic Haskell.

Evaluation

Structural dependencies. Clean supports the definition of generic functions in the style of Generic Haskell. It does not support type-indexed data types.

Full reflexivity. We couldn't get generic functions in Clean to work for types with higher-order kinds, so the generic programming extension of Clean doesn't seem to be fully reflexive.

First-class generic functions. Generic functions are treated as kind-indexed families of type classes. Type classes are not first-class, so generic functions are not first-class either.

Multiple type arguments. Clean allows the definition of classes with multiple type arguments. All type arguments, however, must be instantiated to the same type at the call site. Therefore, true multi-argument generic functions are not supported.

Type system. Generic functions are fully integrated into Clean's type system, by mapping each generic function to a family of type classes. The compiler ensures type-correctness.

Type safety. Clean's generic programming extension is fully type safe.

Type-language expressiveness. The type language of Clean is comparable in expressiveness to that of Haskell. Clean additionally supports uniqueness annotations on types. We haven't investigated the interaction between generics and uniqueness types.

The type of a generic function. The type of a generic function is declared using the **generic** construct. For instance, the type of the generic equality function looks as follows:

generic *eq* a :: a a → Bool

The types are very similar in nature to those of Generic Haskell. They lack dependencies, which makes them a bit less expressive, but in turn a bit easier to understand.

Properties of generic functions. Again, Hinze's theory is the basis of Clean's generic programming extension. Therefore it is possible to state and prove theorems following his formalism.

Integration with the underlying programming language. Generic programming is fully integrated with the Clean language. Only the module `StdGeneric` must be imported in order to define new generic functions.

Specialization versus interpretation. Clean uses specialization to compile generic functions. Specialization is explicit, using the **derive** construct.

Code optimization Because Clean makes use of the same theory as Generic Haskell, specialized code is inherently inefficient: data is converted between its original form and a structural view several times during the call of a generic functions. There is extensive work on optimizing specialized code for generic functions generated by Clean [4, 5].

Separate compilation. Generic programming is integrated into Clean, and Clean supports separate compilation.

Practical aspects. Clean is maintained and runs on several platforms. However, the documentation of generic programming in Clean is lacking. The chapter in the Clean documentation is missing, and there's a gap between the syntax used in papers and the implementation. Furthermore, the error messages of the Clean compiler with respect to generic functions are not very good Nevertheless, generic programming in Clean seems very usable and has been used, for example, to implement a library for generating test data [46] as well as a GUI library [1].

5 Conclusions and future work

In this section we draw conclusions from the evaluations in the previous section. Using these conclusions, we try to answer the question we posed in the introduction of these lecture notes: 'How do you choose between the different approaches to generic programming in Haskell?' This question is a bit similar to the question how you choose a programming language for solving a programming problem. Answers to this question usually contain 'religious' aspects. We try to avoid religion as much as possible, and answer the question in two ways. First, we summarize the evaluations of the previous section, and draw conclusions about the suitability of the different approaches for different generic programming concepts. Second, to end on a positive note, for each approach we try to give arguments why you would use it. Furthermore, we describe future work.

5.1 Conclusions

5.2 Suitability for generic programming concepts.

Figure 5 shows the results of our evaluations of the different approaches to generic programming in Haskell. Such a presentation does not offer the possibility to make subtle distinctions, but it does give an overview of the evaluation results. We use the following categories in this table:

- `++`: satisfies all requirements.
- `+`: satisfies the requirements except for some small details.
- `o`: satisfies a number of requirements.
- `-`: satisfies just a few of the requirements.
- `--`: does not satisfy the requirements.

The results are obtained by an informal translation our evaluations into points on this five-point scale.

Structure in programming languages. Generic Haskell allows the definition of type-indexed functions with kind-indexed types, and type-indexed data type with kind-indexed kinds. Since DrIFT can generate anything, it can also be used to generate type-indexed types. There is no support (library, predefined constructs) for doing so, however. The other approaches only allow the definition of type-indexed functions.

	Structure	Completeness	Safe	Info	Integration	Tools
GH	++	+	++	++	++	+
DrIFT	+	o	−−	−	+	+
PolyP	o	−	+	+	+	o
DTCs	o	o	++	++	++	+
LIGD	o	+	++	++	++	++
SYB	o	+	++	+	++	+
GM	o	+	++	++	++	+
Clean	o	+	++	++	++	+

Fig. 5. Evaluation results for approaches to generic programming

The type completeness principle . No approach truly satisfies the type completeness principle. Both SYB and LIGD allow higher-order generic functions, and generic functions on almost all data types definable in Haskell. On the other hand, it is impossible to define the generic map function in these approaches. GM allows higher-order generic functions, and the definition of generic map, but needs different classes for different brands of generic functions. Furthermore, GM cannot handle higher-order kinded data types. Generic Haskell and Clean do not offer higher-order generic functions, but generic functions work on almost any data type definable in the underlying programming language, and defining the generic map function is no problem. Higher-orderness does not really play a rôle in DrIFT, and DrIFT cannot handle higher-order kinded data types. DTCs cannot handle higher-order kinded data types, and, just as classes, cannot represent higher-order generic functions. PolyP does not allow higher-order generic functions either and only works for regular data types of kind $\star \to \star$.

Well-typed expressions do not go wrong . Clean, Generic Haskell, DTCs, LIGD, GM, and SYB are type safe. DrIFT offers no safety at all: a generated document can represent a completely bogus program. PolyP does not complain about undefined arms, but otherwise type checks generic functions.

Information in types . Clean, LIGD, and Generic Haskell allow the definition of generic functions on almost any data type. Types of generic functions generally correspond to intuition, and there exists a theory of generic functions by means of which properties for generic functions can be proved. SYB defines generic functions on more or less the same class of data types. Proving properties of generic functions in SYB is hard because they rely on properties of, possibly user-defined, instances of the classes *Data* and *Typeable*. DTCs, GM, and SYB suffer from the fact that higher-order contexts (not implemented in Haskell) are needed to generate instances of generic functions on higher-order kinded data types. In DrIFT all rules have the same type, namely $\mathsf{Data} \to \mathsf{Doc}$, and it is virtually impossible to prove anything about the functions represented by the documents. Proving properties about generic functions in PolyP is relatively easy. However, the type language allowed in PolyP is restricted.

Integration with the underlying programming language . Generic Haskell, Clean, DTCs, LIGD, GM, and SYB are fully integrated with the underlying programming language, where Clean, DTCs, LIGD, GM, and SYB don't even need a separate compiler. PolyP can only deal with a subset of Haskell. DrIFT has to be recompiled if a new generic function is added to the rules.

Tools . SYB is shipped as a library of GHC, and is fully supported. The latest versions of SYB have not been included yet in GHC, which means that the current version still suffers from some of the limitations of previous versions of SYB, in particular the limitation that generic functions cannot be extended. Generic Haskell, LIGD, GM, and DTCs do not do any optimization on the generic code, but otherwise provide good error messages. DTCs cannot access constructor names, which limits their usability a bit. Clean does optimize the generated code, but provides no error

messages. PolyP is not very actively maintained anymore. DrIFT is maintained, but also provides no error messages.

5.3 Why would I use this approach?

- Use Generic Haskell if you want to experiment with type-indexed functions with kind-indexed types and/or type-indexed data types. Generic Haskell is probably the most expressive generic programming extension of Haskell. A disadvantage of using Generic Haskell might be that the generated code contains quite a number of mappings from data types to structure types and back again, and hence not as efficient as hand-written code might be.
- Use Clean if you want to use an approach to generic programming that is similar to Generic Haskell, is fully integrated in its underlying programming language, and generates nearly optimal code for generic functions. Clean does not support the advanced features of Generic Haskell such as dependencies, type-indexed data types, and default cases.
- Use DrIFT if you want a lot of flexibility in the way you generate code, or if you want to format the code you generate in a particular way. Make sure you don't generate code on data types that use higher-order kinds.
- Use PolyP if you want to define generic functions that use the recursive structure of data types, such as a generalization of the *foldr* function on lists, the catamorphism. Remember that PolyP only generates code for data types of kind $\star \rightarrow \star$.
- Use Derivable Type Classes if you want (limited) Generic Haskell like generic programming functionality fully integrated in the underlying programming language. DTCs don't support type-indexed data types, or types with higher-order kinds.
- Use the Lightweight approach if you want to use a simple but expressive library for generic programming, and your generic functions don't have to work on many different data types.
- Use Scrap Your Boilerplate if you want to manipulate a value of a large abstract syntax at a particular place in the abstract syntax, and if you want to have an approach to generic programming that is fully integrated in the underlying programming language.
- Use Generics for the Masses if you want a fully Haskell 98 compatible library that supports generic programming.

Between the eight approaches to generic programming in Haskell described in these lecture notes, we distinguish two related groups:

- Generic Haskell and Clean.
- Lightweight approaches: Derivable Type Classes, Lightweight Generics and Dynamics, and Generics for the Masses.

The difference between Generic Haskell and Clean is that Generic Haskell is more expressive and provides more features, whereas Clean produces better code. The various lightweight approaches can be compared as follows. DTCs and GM use classes for defining generic functions, so higher-order kinded data types are out of reach for these approaches. DTCs automatically generate the conversion functions for instances of generic functions, something that has to be done by hand for LIGD and GM.

5.4 Future work

These lecture notes only compare approaches to generic programming in Haskell. The only approaches to generic programming we have not addressed are Strafunski, and the approach that uses Template Haskell for generic programming. Strafunski is rather similar to SYB. Using Template Haskell for generic programming is rather similar to DrIFT, but since types play a larger rôle in Template Haskell, it is sufficiently different to warrant a separate discussion. The final version of these lecture notes will contain a description of Strafunski and Template Haskell as well.

We have yet to perform the same exercise for approaches to generic programming in different programming languages. In the final version of these lecture notes we hope to include all approaches to generic programming we know of.

Acknowledgements. We thank the participants of the 61st IFIP WG 2.1 meeting for their comments on a presentation about this work.

References

1. Peter Achten, Marko van Eekelen, and Rinus Plasmeijer. Generic Graphical User Interfaces. In *The 15th International Workshop on the Implementation of Functional Languages, IFL 2003, Selected Papers*, volume 3145 of *LNCS*, pages 152–167. Springer-Verlag, 2004.
2. Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, University of Nijmegen, The Netherlands, 2005.
3. Artem Alimarine and Rinus Plasmijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, 2002.
4. Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, volume 3125 of *LNCS*, pages 16–31. Springer-Verlag, 2004.
5. Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In *Proceedings of Seventh International Symposium on Practical Aspects of Declarative Languages, PADL 2005*, volume 3350 of *LNCS*, pages 203–218. Springer-Verlag, 2005.
6. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, pages 1–20. Kluwer, 2003.
7. Frank Atanassow, Dave Clarke, and Johan Jeuring. Scripting XML with Generic Haskell. In *Proceedings of the 7th Brazilian Symposium on Programming Languages, SBLP 2003*, 2003. An extended version of this paper appears as ICS, Utrecht University, technical report UU-CS-2003-023.
8. Frank Atanassow and Johan Jeuring. Inferring type isomorphisms generically. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, volume 3125 of *LNCS*, pages 32–53. Springer-Verlag, 2004.
9. L. Augustsson. Cayenne — a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, 1998. Presented at ICFP'98.
10. Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, 2003.
11. Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.
12. Juan Chen and Andrew W. Appel. Dictionary passing for polytypic polymorphism. Technical Report TR-635-01, Princeton University, 2001.
13. James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press, 2002.
14. Dave Clarke and Andres Löh. Generic Haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming*, volume 243 of *IFIP*, pages 21–48. Kluwer Academic Publishers, 2003.
15. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.
16. Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 1998*, pages 301–312. ACM Press, 1998.
17. Alan Demers, James Donahue, and Glenn Skinner. Data types as values: polymorphism, type-checking, encapsulation. In *Conference Record of POPL '78: The 5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 23–30. ACM Press, 1978.
18. C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *22nd Symposium on Principles of Programming Languages, POPL '95*, pages 118–129, 1995.
19. Jun Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, January 2001.
20. Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134. ACM Press, 2003.
21. Paul Hagg. A framework for developing generic XML Tools. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.

22. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages, POPL '95*, pages 130–141, 1995.
23. Ralf Hinze. Generics for the masses. *Journal of Functional Programming*. to appear.
24. Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, Technical report of Utrecht University, UU-CS-1999-28, 1999.
25. Ralf Hinze. Functional pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, 2000.
26. Ralf Hinze. *Generic Programs and Proofs*. 2000. Habilitationsschrift, Bonn University.
27. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.
28. Ralf Hinze. Generics for the masses. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 236–243. ACM Press, 2004.
29. Ralf Hinze and Johan Jeuring. Generic Haskell: applications. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 57–97. Springer-Verlag, 2003.
30. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.
31. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51((1-2)):117–151, 2004.
32. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the Fourth Haskell Workshop*, 2000.
33. Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In Tarmo Uustalu, editor, *Proceedings of the Eighth International Conference on Mathematics of Program Construction, MPC 2006*, LNCS. Springer-Verlag, 2006. To appear.
34. Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In Philip Wadler and Masimi Hagiya, editors, *Proceedings of the Eighth International Symposium on Functional and Logic Programming, FLOPS 2006*, LNCS. Springer-Verlag, 2006.
35. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
36. J. Hughes. The design of a pretty-printing library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.
37. Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
38. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
39. C. Barry Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2:307–315, 1999.
40. C. Barry Jay. Distinguishing data structures and functions: the constructor calculus and functorial types. In S. Abramsky, editor, *Typed Lambda Calculi and Applications: 5th International Conference TLCA 2001, Kraków, Poland, May 2001 Proceedings*, volume 2044 of *LNCS*, pages 217–239. Springer-Verlag, 2001.
41. C. Barry Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6):911–937, 2004.
42. C. Barry Jay and Delia Kesner. Pure pattern calculus. In Peter Sestoft, editor, *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *LNCS*. Springer-Verlag, 2006.
43. C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.
44. J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming '96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.
45. Johan Jeuring and Rinus Plasmeijer. Generic programming for software evolution. In *Informal proceedings of the ERCIM workshop on Software Evolution*, 2006.
46. Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic Automated Software Testing. In Ricardo Peña, editor, *The 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Selected Papers*, volume 2670 of *LNCS*. Springer-Verlag, 2003.
47. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proceedings Practical Aspects of Declarative Programming, PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, 2002.
48. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. TLDI'03.

49. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, 40(9):204–215, 2005.
50. Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 244–255. ACM Press, 2004.
51. Bernard Lang. Threshold evaluation and the semantics of call by value, assignment and generic procedures. In *Conference Record of POPL '77: The 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–237. ACM Press, 1977.
52. M.M. Lehman. Programs, life cycles and the laws of software evolution. *Proc. IEEE*, 68(9):1060–1078, 1980.
53. M.M. Lehman and L.A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, London, 1985.
54. Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.
55. Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Olin Shivers, editor, *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2003*, pages 141–152. ACM Press, August 2003.
56. Andres Löh and Ralf Hinze. Open data types. Submitted for publication, 2006.
57. 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.
58. N. Marti-Oliet M. Clavel, F. Duran. Polytypic programming in maude. In *WRLA 2000*, 2000.
59. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
60. Conor McBride. Epigram: practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 130–170. Springer-Verlag, 2005.
61. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.
62. E Moggi, Bellè, and C.B. Jay. Monads, shapely functors and traversals. In M. Hoffman, Pavlović, and P. Rosolini, editors, *Proceedings of the Eighth Conference on Category Theory and Computer Science (CTCS'99)*, volume 24 of *Electronic Lecture Notes in Computer Science*, pages 265–286. Elsevier, 1999.
63. Ulf Norell and Patrik Jansson. Prototyping generic programming in Template Haskell. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, volume 3125 of *LNCS*, pages 314–333. Springer-Verlag, 2004.
64. Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Generics as a library. In Henrik Nilsson, editor, *Proceedings of the Seventh Symposium on Trends in Functional Programming, April 19–21, 2006, Nottingham, UK*, 2006.
65. Simon Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of the Journal of Functional Programming.
66. A.L. Powell. A literature review on the quantification of software change. Technical Report YCS 305, Computer Science, University of York, 1998.
67. Fermín Reig. Generic proofs for combinator-based generic programs. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5. Intellect, 2006.
68. Stephen A. Schuman. On generic functions. In Stephen A. Schuman, editor, *First IFIP WG 2.1 Working Conference on New Directions in Algorithmic Languages 1975*, pages 169–192. IRIA, 1975.
69. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89*, pages 347–359. ACM Press, 1989.
70. M. Wallace and C. Runciman. Heap compression and binary I/O in Haskell. In *2nd ACM Haskell Workshop*, 1997.
71. David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, Ltd, 2004.
72. Stephanie Weirich. Higher-order intensional type analysis. In D. Le Métayer, editor, *Proceedings of the 11th European Symposium on Programming, ESOP 2002*, volume 2305 of *LNCS*, pages 98–114. Springer-Verlag, 2002.
73. Stephanie Weirich and Liang Huang. A design for type-directed programming in Java. In *Workshop on Object-Oriented Developments, WOOD 2004*, 2004.
74. Noel Winstanley and John Meacham. The DrIFT manual. http://repetae.net/ john/computer/haskell/DrIFT/, 1997–2005.