Chapter 2
Comparing Approaches to Generic Programming in Haskell

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 datatype-generic programming: PolyP, Functorial ML, 'Scrap Your Boilerplate', Generic Haskell, 'Generics for the Masses', and so on. 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 compare the various approaches to datatype-generic programming in Haskell. We introduce each approach by means of example, and we evaluate it along different dimensions (expressivity, ease of use, and so on).

1 Introduction

You just started implementing your third web shop in Haskell, and 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 at least eight approaches to solve your problem: DrIFT, PolyP, Generic Haskell, Derivable Type Classes, Template Haskell, Scrap Your Boilerplate, Generics for the Masses, Strafunski, and so on. How do you choose?

In these lecture notes we give arguments as to why you would choose a particular approach to generic programming in Haskell to solve your generic programming problem. We 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 compare the various approaches to generic programming we 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, and so on. Usually, the adjective 'generic' is used to indicate 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. [23]. Already in the 1970s this was an active area of research [89, 66, 20].

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 is the type of values to which the function is applied, or the type of the values returned by the function, or the type of values that are used internally in the function. Backhouse and Gibbons [24, 10, 25] call this kind of generic programming *datatype-generic* programming. 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 used the term *polytypic* [48] instead of 'generic', which is less confusing and describes the concept a bit more accurately. However, the term hasn't been picked up by other people working on conceptually the same topic, and maybe it sounds a bit off-putting.

The first programming languages with facilities for datatype-generic programming, beyond generating the definition of equality on user-defined data types, were Charity [18], and the lazy, higher-order, functional programming language Haskell [86]. Since then Haskell has been the most popular testbed for generic programming language extensions or libraries. Here is an incomplete list of approaches to generic programming in Haskell or based upon Haskell:

– Generic Haskell [31, 34, 71, 73].
– DrIFT [99].
– PolyP [48, 81].
– Derivable Type Classes [41].
– Lightweight Generics and Dynamics [15].
– Scrap Your Boilerplate [61, 64, 62, 44, 43].
– Generics for the Masses [35, 84].
– Clean [3, 2]. (Clean is not Haskell, but it is sufficiently close to be listed here.)
– Using Template Haskell for generic programming [82].
– Strafunski [65].
– Generic Programming, Now! [42]

Although Haskell has been the most popular testbed for generic programming extensions, many non-Haskell approaches to generic programming have been designed:

- Charity [18].
- ML [14, 22].
- Intensional type analysis [30, 19, 96].
- Extensional type analysis [21].
- Functorial ML [56, 78], the Constructor Calculus [53], the Pattern Calculus [54, 55], FISh [52].
- Dependently-typed generic programming [6, 11].
- Type-directed programming in Java [98].
- Adaptive Object-Oriented Programming [69].
- Maude [17].

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

In these lecture notes we compare most of the approaches to generic programming in Haskell or based upon Haskell. We do not include Strafunski and Generic Programming, Now! in our comparison. Strafunski is rather similar to Scrap Your Boilerplate, and Generic Programming, Now! is an advanced variant of the lightweight approaches we will discuss. Besides that, a paper [42] about the Generic Programming, Now! approach is included in these lecture notes, and itself contains a comparison to other approaches to generic programming. In future work we hope to also compare approaches to generic programming in other programming languages.

Types play a fundamental rôle in generic programming. In an untyped or dynamically typed language, it is possible to define functions that adapt to many data structures, and one could therefore argue that it is much easier to do generic programming in these languages. We strongly disagree: since generic programming is fundamentally about programming with types, simulating generic programming in an untyped language is difficult, since the concept of types and the accompanying checks and guidance are missing. Generic programs are often complex, and feedback from the type system is invaluable in their construction. This difficulty can also be observed in our treatment of DrIFT and Template Haskell, both approaches with only limited support from the type system.

We 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 generic functions defined in almost twenty papers introducing the various approaches to generic programming. Almost all of these papers contain at least one function from the following list:

- *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.
- *decode*, the inverse of *encode*, 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 parametrized 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 only useful when applied to type constructors, i.e., parametrized data types such as lists or trees. In particular, on types of kind $\star$ it is the identity function.
– *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 characteristic feature of this example is that *update* is only interested in values of a very small part of a possibly very large type. It is generic in the sense that it can be applied to a value of any data type, but it only updates salaries, and ignores all other information in a value of a data type.

The above functions all exhibit different characteristics, which we use to show differences between approaches to generic programming. We do not define all of these functions for each approach, in particular not for approaches that are very similar, but we use these examples to highlight salient points. We then investigate a number of properties for each approach. Examples of these properties are: whether it is possible to define a generic function on any data type that can be defined in the programming language (full reflexivity), whether the programming language is type safe, whether generic functions satisfy desirable properties, and so on. Sometimes we use examples beyond the above functions to better highlight specifics and peculiarities of a certain approach.

These notes are organized as follows. In Section 2 we discuss why generic programming matters by means of a couple of representative examples. We 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. 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, and pretty-printing a value. 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 [37], and type-safe XML data binding tools [7, 63];
– automatic testing [60];

– 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 [61, 71, 62];
– structure editors such as XML editors [29], and generic graphical user interfaces [1];
– typed middleware for distributed systems, such as CORBA [85];
– data-conversion tools [50] which for example store a data type value in a database [29], or output it as XML, or in a binary format [94].

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 [67, 68, 87].

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 to 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 [58].

In the rest of this section we show a number of examples of generic programs. We write the generic programs in Generic Haskell [31, 38, 70]. Generic Haskell is an extension of Haskell that supports generic programming. Any of the other approaches to generic programming could have been chosen for the following exposition. We choose Generic Haskell simply because we have to start somewhere, and because we are responsible for the development of Generic Haskell. We use the most recent version of Generic Haskell, known as *Dependency-style Generic Haskell* [71, 70]. Dependencies both simplify and increase the expressiveness of generic programming. In Section 4 we 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. We will say a bit more about kinds in Section 3.1. 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 expressions *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 like 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 like the List type constructor, and it follows that GRose has kind $(\star \to \star) \to \star \to \star$. We call such a kind that takes a kind constructor as argument a *higher-order* kind. The other kinds are called *first-order* kinds.

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 [12] 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 [32] – 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 view data types in a uniform manner: except for basic predefined types such as Float, IO, and →, 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 a
data Label a  = Label 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, marked by the type Con. While the representation types are generated, the compiler tags each occurrence of Con with an abstract value of type ConDescr describing the original constructor. The exact details of how constructors are represented are omitted [38, 70]. 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 structure-representation type represents the top-level structure of a data type. A type t and its structure-representation type t° are isomorphic. (Strictly speaking this is not true, because the two types may be distinguished using (partially) undefined values.) Here and in the rest of the paper 'isomorphism' should be read as isomorphic modulo undefined values. The isomorphism between a type and its structure-representation type is witnessed by a so-called *embedding-projection pair*: a value $conv_{\mathsf{t}} :: \mathsf{t} \leftrightarrow \mathsf{t}^{\circ}$ of the data type

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

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

$$\begin{array}{ll}
from_{\mathsf{List}} & :: \mathsf{List\ a} \to \mathsf{List}^{\circ}\ \mathsf{a} \\
from_{\mathsf{List}}\ Nil & = Inl\ (Con\ Unit) \\
from_{\mathsf{List}}\ (Cons\ a\ as) & = Inr\ (Con\ (a :\!*\!: as)) \\
to_{\mathsf{List}} & :: \mathsf{List}^{\circ}\ \mathsf{a} \to \mathsf{List\ a} \\
to_{\mathsf{List}}\ (Inl\ (Con\ \ Unit)) & = Nil \\
to_{\mathsf{List}}\ (Inr\ (Con\ (a :\!*\!: as))) & = Cons\ a\ as.
\end{array}$$

The Generic Haskell compiler generates the translation of a type to its structure-representation type, together with the corresponding embedding-projection pair. More details about the correspondence between these and Haskell types can be found elsewhere [34].

A generic program is defined by induction on the structure of structure-representation 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 structure-representation type. 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 consider first 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 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:

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

Here, the names O and I are used as constructors.

Function encode on CharList. To define *encode* on the data type CharList, we assume that there exists a function *encodeChar* :: Char → [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 code of the character. A value of type CharList is now encoded as follows:

```
encodeCharList              :: CharList → [Bit]
encodeCharList Nil          = [O]
encodeCharList (Cons c cs) = I : encodeChar c ++ encodeCharList cs.
```

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

```
ComparingGP⟩ encodeCharList bonn
[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, I, O, I, I, I, O, O].
```

Note that the type of the value that is encoded is not stored. This implies that when decoding, we have to know the type of the value being decoded.

Function encode on Tree. To define *encode* on the data type Tree, we assume there exists, besides a function *encodeChar*, a function *encodeInt* :: Int → [Bit], which takes an integer and returns a list of bits representing that integer. Function *encodeInt* should be defined such that the resulting list of bits can be unambiguously decoded back to an integer again. A value of type Tree can then be encoded as follows:

```
encodeTree              :: Tree → [Bit]
encodeTree Empty        = [O]
encodeTree (Leaf i)     = [I, O] ++ encodeInt i
encodeTree (Bin l c r) = [I, I ]
                         ++ encodeTree l
                         ++ encodeChar c
                         ++ encodeTree r.
```

The *Empty* constructor of the Tree data type is encoded with a single bit, and the other two constructors are encoded using a sequence of two bits.

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:

```
encodeList                      :: (a → [Bit]) → List a → [Bit]
encodeList encodeA Nil          = [O]
encodeList encodeA (Cons x xs) = I : encodeA x
                                 ++ encodeList encodeA xs.
```

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:

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

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

- The function $encode\{\!|\mathsf{a}|\!\}$ 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, $encode\{\!|\mathsf{CharList}|\!\}$ 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 $encode\{\!|\mathsf{a}|\!\}$ 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' (or branch) 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 $\star$ is derived automatically from this base type. In particular, $encode\{\!|\mathsf{List}|\!\}$ is translated to a value that has the type $(\mathsf{a} \to [\mathsf{Bit}]) \to (\mathsf{List\ a} \to [\mathsf{Bit}])$.
- The Generic Haskell code as given above is a bit prettier than the actual Generic Haskell code. In the actual Generic Haskell code we use the pre-

fix type constructor Sum instead of the infix type constructor · :+: ·, and similarly Prod instead of · :∗: ·.

- The constructor case Con has an extra argument c, which contains the constructor description of the current constructor. Similarly, the label case Label has an extra argument l that contains a description of the current label. This is a special type pattern also containing a value, namely a constructor (label) description. The constructor (label) description can only be accessed in the Con (Label) case.

The Con and the Label case are useful 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 arms. In this case, Generic Haskell allows to omit these arms 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 are recovered from an initial segment of the list of bits. We introduce a type Parser that is used as the type of function *decodes*. Furthermore, we assume we have a map function on this type. 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.

```
type Parser a = [Bit] → [(a, [Bit])]
mapP          :: (a → b) → Parser a → Parser b
decodes{|a :: ⋆|}     :: (decodes{|a|}) ⇒ Parser a
decodes{|Unit|}    xs = [(Unit, xs)]
decodes{|Int|}     xs = decodesInt xs
decodes{|Char|}    xs = decodesChar xs
decodes{|α :+: β|} xs = bitCase (mapP Inl (decodes{|α|}))
                                (mapP Inr (decodes{|β|}))
                                xs
decodes{|α :*: β|} xs = [(y1 :*: y2, r2) | (y1, r1) ← decodes{|α|} xs
                                         , (y2, r2) ← decodes{|β|} r1]
bitCase       :: Parser a → Parser a → Parser a
bitCase p q = λbits → case bits of
                         O : bs → p bs
                         I : bs → q bs
                         []     → []
```

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. We therefore use the standard list-of-successes technique [93], where the input list is transformed into a list of pairs, containing all possible parses with the associated unconsumed part of the input. Assuming that the decoding of primitive types such as Int and Char is unambiguous, 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 *decodesChar* is obtained from somewhere.

A value of type Unit is represented using no bits at all, hence it is 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*. Decoding fails if we run out of input while decoding a sum. The product case 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:

$$
\begin{aligned}
&\mathit{decode}\{\!|\mathsf{a} :: \star|\!\} \;\; :: \; (\mathit{decodes}\{\!|\mathsf{a}|\!\}) \Rightarrow [\mathsf{Bit}] \to \mathsf{a} \\
&\mathit{decode}\{\!|\mathsf{a}|\!\}\; \mathit{bits} = \textbf{case}\; \mathit{decodes}\{\!|\mathsf{a}|\!\}\; \mathit{bits}\; \textbf{of} \\
&\qquad\qquad\qquad\quad [(y, [\,])] \to y \\
&\qquad\qquad\qquad\quad \_ \qquad\;\; \to \mathit{error}\; \texttt{"decode: no parse"}.
\end{aligned}
$$

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. A generic abstraction only works on types that have the specified kind ($\star$ in the case of function *decode*).

For each type t in the domain of both *decode* and *encode*, we have that for any finite and total value x of type t,

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

2.4 Equality

The generic equality function takes two arguments instead of a single argument as *encode* does. 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.

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

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 are 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.

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

2.5 Map

In category theory, the functorial map is defined as the action of a functor on an arrow. There is no way to describe functors in Generic Haskell, and neither is it possible to distinguish argument types in structure-representation types. The approach we take to defining *map* in Generic Haskell 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:

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

Given a value, the *copy* function produces a copy of that value and is thus a generic version of the identity function. 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:

$$
\begin{array}{l}
\mathit{copy}\{\|\alpha :+: \beta\|\}\ x = x \\
\mathit{copy}\{\|\alpha :*: \beta\|\}\ x = x.
\end{array}
$$

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. Function *copy* has a dependency on itself. This implies that whenever *copy* is used on a type of a kind different from $\star$, extra components are needed. For example, applying *copy* to the type $[\mathsf{a}]$, where the type list has kind $\star \to \star$, requires a component of *copy* on the type a. The copy function on $[\mathsf{a}]$ takes a copy function on the type a as argument, and applies this copy function whenever it encounters an a-value. The standard behavior of generic functions with dependencies is that argument functions are constructed in exactly the same way as the instance of the generic function itself. So the copy function on $[\mathsf{Char}]$ would be the instance of the generic copy function on lists, taking the instance of the generic copy function on Char as argument. Local redefinition allows us to adapt the standard behavior. As an example, we can increase all elements of a list by one, using the function

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

Here we locally redefine *copy* to behave as the function $(+1)$ on values of type α that appear in a list of type $[\alpha]$. Obviously, this is only type correct if α equals Int (or, more generally, is an instance of the *Num* class). Note that *incBy1* 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

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

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

$$\begin{aligned} map' &:: (\mathsf{a} \to \mathsf{a}) \to [\mathsf{a}] \to [\mathsf{a}] \\ map &:: (\mathsf{a} \to \mathsf{b}) \to [\mathsf{a}] \to [\mathsf{b}]. \end{aligned}$$

The function passed to *map* may change the type of its argument; the function passed to map' preserves the argument type.

Inspired by this deficiency, we can ask ourselves if it is 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

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

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

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

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

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

The type of the *map* function is now parametrized over *two* type variables, and so is the dependency. The arms in the definition of *map* are still parametrized by a single type (Generic Haskell does not allow more than one type argument in definitions of generic functions). Function *map* is always called with a single type argument, which is the type argument that is used to induct over. When *map* is used at a constant type, both variables a and b are instantiated to the same constant type. Only when locally redefining the function for a dependency variable, the additional flexibility is available. Figure 1 shows some types (with explicit kind annotations for the type variables) for applications of *map* to specific type arguments.

$$
\begin{array}{l}
map\{\!|\mathsf{Tree} :: \star|\!\} :: \mathsf{Tree} \to \mathsf{Tree}\\
map\{\!|\mathsf{List}\ (\mathsf{a} :: \star) :: \star|\!\} ::\\
\quad \forall (\mathsf{a}_1 :: \star)\ (\mathsf{a}_2 :: \star)\,.\,(map\{\!|\mathsf{a}|\!\} :: \mathsf{a}_1 \to \mathsf{a}_2) \Rightarrow \mathsf{List}\ \mathsf{a}_1 \to \mathsf{List}\ \mathsf{a}_2\\
map\{\!|\mathsf{GTree}\ (\mathsf{a} :: \star)\ (\mathsf{b} :: \star) :: \star|\!\} ::\\
\quad \forall (\mathsf{a}_1 :: \star)\ (\mathsf{a}_2 :: \star)\ (\mathsf{b}_1 :: \star)\ (\mathsf{b}_2 :: \star)\,.\,(map\{\!|\mathsf{a}|\!\} :: \mathsf{a}_1 \to \mathsf{a}_2, map\{\!|\mathsf{b}|\!\} :: \mathsf{b}_1 \to \mathsf{b}_2) \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{GTree}\ \mathsf{a}_1\ \mathsf{a}_2 \to \mathsf{GTree}\ \mathsf{b}_1\ \mathsf{b}_2\\
map\{\!|\mathsf{GRose}\ (\mathsf{f} :: \star \to \star)\ (\mathsf{a} :: \star) :: \star|\!\} ::\\
\quad \forall (\mathsf{f}_1 :: \star \to \star)\ (\mathsf{f}_2 :: \star \to \star)\ (\mathsf{a}_1 :: \star)\ (\mathsf{a}_2 :: \star)\,.\\
\qquad (map\{\!|\mathsf{f}\ (\mathsf{c} :: \star)|\!\} :: \forall (\mathsf{c}_1 :: \star)\ (\mathsf{c}_2 :: \star)\,.\,(map\{\!|\mathsf{c}|\!\} :: \mathsf{c}_1 \to \mathsf{c}_2) \Rightarrow \mathsf{f}_1\ \mathsf{c}_1 \to \mathsf{f}_2\ \mathsf{c}_2\\
\qquad , map\{\!|\mathsf{a}|\!\} :: \mathsf{a}_1 \to \mathsf{a}_2\\
\qquad) \Rightarrow \mathsf{GRose}\ \mathsf{f}_1\ \mathsf{a}_1 \to \mathsf{GRose}\ \mathsf{f}_2\ \mathsf{a}_2.
\end{array}
$$

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

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

$$
\begin{array}{l}
\textbf{type}\ \mathsf{Pair}\ \mathsf{a}\ \mathsf{b} \quad = (\mathsf{a}, \mathsf{b})\\
\textbf{data}\ \mathsf{Either}\ \mathsf{a}\ \mathsf{b} = \mathit{Left}\ \mathsf{a} \mid \mathit{Right}\ \mathsf{b}.
\end{array}
$$

Then the expressions

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

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

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. We do not treat field labels, so our implementation is a simplification of Haskell's *show*; the complete definition of *show* can be found in Generic Haskell's library. 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 is 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 a space.

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, using the function *intersperse* from Haskell's List module.

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.

$$show\{|a :: \star|\} :: (showP\{|a|\}) \Rightarrow a \rightarrow String$$
$$show\{|a|\} = showP\{|a|\}\ id$$

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 [61], we use the following data types to represent the organizational structure of a company.

```
data Company  = C [Dept]
data Dept     = D Name Manager [SubUnit]
data SubUnit  = PU Employee | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Float
type Manager  = Employee
type Name     = String
type Address  = String
```

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 by Clarke and Löh [16].

```
update{|a :: ⋆|} :: (update{|a|}) ⇒ a → a
update extends map
update{|Salary|} (S s) = S (s ∗ (1 + 0.15))
```

Semantically, this is the same function as

```
update{|Unit|}  x  = x
update{|Int|}   x  = x
```

$$
\begin{aligned}
&update\{\!|\mathsf{Char}|\!\}\quad x &&= x\\
&update\{\!|\alpha :\!+\!: \beta|\!\}\ (Inl\ x) &&= Inl\ (update\{\!|\alpha|\!\}\ x)\\
&update\{\!|\alpha :\!+\!: \beta|\!\}\ (Inr\ x) &&= Inr\ (update\{\!|\beta|\!\}\ x)\\
&update\{\!|\alpha :\!*\!: \beta|\!\}\ (x :\!*\!: y) &&= update\{\!|\alpha|\!\}\ x :\!*\!: update\{\!|\beta|\!\}\ y\\
&update\{\!|\mathsf{Salary}|\!\}\ (S\ s) &&= S\ (s * (1 + 0.15)).
\end{aligned}
$$

The **extends** construct allows us to abbreviate such small variations of generic functions.

3 Criteria for comparison

This section discusses the criteria we use for comparing approaches to generic programming in Haskell. This is a subset of the criteria we would use for comparing approaches to generic programming in any programming language. Together, these criteria can be viewed as a characterization of generic programming. Adding generic programming capabilities to a programming language is a programming-language design problem. Many of the criteria we give are related to or derived from programming-language design concepts. 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 illustrate the criteria with an evaluation of Generic Haskell.

3.1 Structure in programming languages

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 some dependently-typed programming languages there is a possibly infinite hierarchy of levels, where level $n + 1$ imposes structure on elements of level n [90].

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 having 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

There even exist dependencies between non-adjacent levels: properties of generic functions are values that depend on kinds [33, 51]. However, we will not further discuss these non-adjacent dependencies in these notes.

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 current research [76, 9, 90, 100]. 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 a 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 $\forall a . \mathsf{List}\ a \rightarrow \mathsf{Int}$. By contrast, in a type-indexed function 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* [39], 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 [46]. 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 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 kind $\kappa \rightarrow \kappa'$, cannot be specified, but is automatically generated by the compiler (as it is determined by the way Generic Haskell specializes generic functions). Generic abstractions only generate code for functions on types of the kind specified in the type of the generic abstraction.

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 [39].

3.2 The Type Completeness Principle

The Type Completeness Principle [95] says that no programming-language operation should be arbitrarily restricted in the types of its operands, or, equivalently, all programming-language operations should be applicable to all operands for which they make sense. 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 (some Haskell compilers, such as GHC, do allow passing polymorphic arguments). 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 that are definable in Haskell 98, except for constraints in data-type definitions. So a data type of the form

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

is not dealt with correctly. However, constrained data types are a corner case in Haskell and can easily be simulated using other means. Furthermore, Nogueira [80] shows how to make Generic Haskell work for data types with constraints.

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

$$\textbf{data}\ \mathsf{Foo} = \forall \mathsf{a}\,.\ \mathit{MkFoo}\ \mathsf{a}\ (\mathsf{a} \to \mathsf{Bool}).$$

Although such types 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 the Glasgow Haskell Compiler (GHC), of which the following type Term, representing typed terms, is an example:

```
data Term :: * → * where
  Lit    :: Int → Term Int
  Succ   :: Term Int → Term Int
  IsZero :: Term Int → Term Bool
```

$$
\begin{array}{ll}
\mathit{If} & :: \mathsf{Term\ Bool} \to \mathsf{Term\ a} \to \mathsf{Term\ a} \to \mathsf{Term\ a} \\
\mathit{Pair} & :: \mathsf{Term\ a} \to \mathsf{Term\ b} \to \mathsf{Term\ (a, b)}.
\end{array}
$$

Note that the result types of the constructors are restricted for Terms, so that if we pattern match on a Term Bool, for example, we already know that it cannot be constructed by means of *Lit*, *Succ* or *Pair*. The structural representation using sums of products that Generic Haskell uses to process data types uniformly is not directly applicable to data types containing existential components or to GADTs. Generic Haskell is thus not fully reflexive with respect to modern extensions of Haskell.

Type universes. Some generic functions only make sense on a particular set of data types, or on a subset of all data types. For example, Malcolm [75] defines the *catamorphism* only for regular data types of kind $\star \to \star$. Bird and Paterson [13] have shown how to define catamorphisms on nested data types, and using tupling it is possible to define catamorphisms on mutually recursive types, but we are not aware of a single definition of a catamorphism that combines these definitions. Many generic functions, such as *show* and equality, cannot sensibly be defined on the type of functions. Is it possible to define generic functions on a particular set of data types, or on a subset of data types? Can we describe *type universes* [11]?

Generic Haskell has some facilities to support defining generic functions on a particular set of data types. If we only want to use a generic function on a particular set of data types, we can define it for just those data types. This is roughly equivalent to defining a class and providing instances of the class for the given data types.

Furthermore, by not giving a case for the function space (or other basic types for which we do not want to define the generic function), a generic function is not defined for data types containing function spaces, and it is a static error for a generic function to be used on a data type containing function spaces.

Finally, Generic Haskell supports so-called *generic views* [45] on data types, by means of which we can view the structure of data types in different ways. Using generic views, we can for example view (a subset of) data types as fixed points of regular functors, which enables the definition of the catamorphism.

First-class generic functions. Can a generic function take a generic function as argument? We will also use the term higher-order generic functions for first-class generic functions. An example where a higher-order generic function might be useful is in a generic *show* function that only prints part of its input, depending on whether or not some property holds of the input.

Generic Haskell does not have first-class generic functions. To a certain extent first-class generic functions can be mimicked by means of dependencies and 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. Note that the *type* of a generic function may take multiple type arguments (such as the type of *map*).

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. Usually, transforming one type structure into another can be achieved by combining two generic functions – one that maps a value into a universal structure, and another that recovers a value from the universal structure. Instances of these functions on for example the data type lists can be implemented by means of a *fold* (mapping into a universal structure) and an *unfold* (parsing from a universal structure). Compositions of unfolds with folds are so-called *metamorphisms* [26]. Since we are not aware of generic metamorphisms, 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 [77] 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. A type system for Generic Haskell has been given by Hinze [33] and Löh [70] (an extension of Hinze's system with several extra features).

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 is a compiled program prevented from crashing because a non-existing instance of a generic function is called?

Generic Haskell is type safe in both aspects.

3.4 Information in types

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.

The type of a generic function. Do types of generic functions in some way correspond to intuition? A generic function $f\{\!|a|\!\}$ that has type $a \to a \to \mathsf{Bool}$ is probably a comparison function. But what does a function of type $(\forall a\ b\,.\, \mathit{Data}\ a \Rightarrow f\ (a \to b) \to a \to f\ b) \to (\forall a\,.\, a \to f\ a) \to a \to f\ a$ do (this is a rather powerful combinator, which we will encounter again in one of the approaches)? This question is related to the possibility to infer useful properties, like free theorems [92], for a generic function from its type [57, 28].

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

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

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

$$map\{\!|a :: \star, b :: \star|\!\} :: (map\{\!|a, b|\!\}) \Rightarrow a \to b$$

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 habilitation thesis [33], 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, in our case Haskell?

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 such a generic function be extended 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?

A generic program can be used on many data types. But how much work needs to be done to use a generic function on a new data type? Is it simply a matter of writing **deriving** ... in a data-type declaration, or do we also have to implement the embedding-projection pair for the data type, for example?

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. Generic functions can be used on data types with no extra work. Generic Haskell generates the necessary machinery such as structure-representation types and embedding-projection pairs behind the scenes.

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 'lightweight' approaches to generic programming require no additional 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? What is the speed of the generated code? Ideally generic programming does not lead to a performance penalty. For example, in the STL community, this is a requirement for a generic function [79] (not to be confused with a datatype-generic function).

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. There exists a prototype implementation of Generic Haskell in which the extra marshaling is fused away [91], but the techniques have not been added to the Generic Haskell compiler releases. The fusion optimization techniques in the underlying programming language Haskell are not strong enough to optimize generated Generic Haskell code.

Separate compilation. Can a generic function that is defined in one module be used 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, and it should be possible to build Generic Haskell anywhere where GHC is installed. The latest release is from October, 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 compiler 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 [33, 71, 70], but only partially implemented.

3.7 Other criteria

This section lists some of the criteria that do not fall in the categories discussed in the previous subsections, or that are irrelevant for comparing the generic programming approaches in Haskell, but might be relevant for comparing approaches to generic programming in different programming languages.

Type-language expressiveness of the underlying programming language. If all values in a programming language have the same type, 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. Flexible and powerful type languages are desirable, but the more expressive a type language, the harder it becomes to write generic programs. What kind of data types can be expressed in the type language?

Haskell has a very powerful, flexible, and expressive type language. This make generic programming in Haskell particularly challenging.

Size matters. The size of a program matters – some people are even paid per line of code –, and the same holds for a generic program. It is usually easier to read and maintain a single page of code than many pages of code, although sometimes extra information, such as type information, properties satisfied by a program, or test cases for a program, are useful to have. So code size matters, but not always. Except for some obvious cases, we will not say much about code size in our comparisons.

Ease of learning. Some programming approaches are easier to learn than others. Since there are so many factors to the question how easy it is to learn a programming language, and since it is hard to quantify, we refrain from making statements about this question, other than whether or not the approach to generic programming is documented. However, it is an important question.

4 Comparing approaches to generic programming

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

We can distinguish three groups of approaches with similar characteristics among the approaches to generic programming in Haskell.

- Generic Haskell and Clean are programming-language extensions based on Hinze's theory of type-indexed functions with kind-indexed types [34].
- DrIFT and implementations of generic programming using Template Haskell are based on a kind of reflection mechanism.
- Derivable Type Classes, Lightweight Generics and Dynamics, Generics for the Masses, and PolyP2 ([81], the latest version of PolyP [48]) are lightweight approaches that do not require reflection or programming-language extensions.

PolyP (in its original version) and Scrap Your Boilerplate are sufficiently different to not be placed in one of these groups. We evaluate the approaches in the groups together, since most aspects of the evaluation are the same. Of course, we already evaluated Generic Haskell in the previous section, so Clean is evaluated separately.

4.1 Clean

Clean's generic programming extension [3, 2] is based on Hinze's work on type-indexed functions with kind-indexed types [34], just like Generic Haskell.

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 structure-representation types in terms of 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*.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{generic}\ \mathit{encode}\ \mathsf{a} :: \mathsf{a} \to [\mathsf{Bit}]} \\
\mathit{encode}\{\!|\mathsf{UNIT}|\!\} & & \mathit{UNIT} & = [\,] \\
\mathit{encode}\{\!|\mathsf{Int}|\!\} & & i & = \mathit{encodeInt}\ i \\
\mathit{encode}\{\!|\mathsf{Char}|\!\} & & c & = \mathit{encodeChar}\ c \\
\mathit{encode}\{\!|\mathsf{EITHER}|\!\} & \mathit{enc}_a\ \mathit{enc}_b & (\mathit{LEFT}\ x) & = [\,O : \mathit{enc}_a\ x\,] \\
\mathit{encode}\{\!|\mathsf{EITHER}|\!\} & \mathit{enc}_a\ \mathit{enc}_b & (\mathit{RIGHT}\ y) & = [\,I : \mathit{enc}_b\ y\,] \\
\mathit{encode}\{\!|\mathsf{PAIR}|\!\} & \mathit{enc}_a\ \mathit{enc}_b & (\mathit{PAIR}\ x\ y) & = \mathit{enc}_a\ x +\!\!+\ \mathit{enc}_b\ y \\
\mathit{encode}\{\!|\mathsf{CONS}|\!\} & \mathit{enc}_a & (\mathit{CONS}\ x) & = \mathit{enc}_a\ x \\
\mathit{encode}\{\!|\mathsf{FIELD}|\!\} & \mathit{enc}_a & (\mathit{FIELD}\ x) & = \mathit{enc}_a\ x \\
\mathit{encode}\{\!|\mathsf{OBJECT}|\!\} & \mathit{enc}_a & (\mathit{OBJECT}\ x) & = \mathit{enc}_a\ x \\
\multicolumn{4}{l}{\textbf{derive}\ \mathit{encode}\ \mathsf{Tree}}
\end{array}
$$

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 $\mathsf{EITHER} :: \star \to \star \to \star$ or $\mathsf{CONS} :: \star \to \star$, additional arguments are passed to the generic function representing the recursive calls. This is very close to Hinze's theory [34] which states that the type of *encode* is based on the kind of the type argument as follows:

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

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

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

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 $t :: \mathsf{Tree}$ by calling $\mathit{encode}\{\!|\star|\!\}\ t$.

If $\mathit{encode}\{\!|\star|\!\}\ x$ is used in another function on a value $x :: \mathsf{a}$, then a class constraint of the form $\mathit{encode}\{\!|\star|\!\}\ \mathsf{a}$ 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, maybe because uniqueness annotations for higher-order kinded (higher-kinded) types are not supported.

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

$$\begin{aligned}
&\textbf{generic}\ \mathit{map}\ \mathsf{a}\ \mathsf{b} :: \mathsf{a} \to \mathsf{b} \\
&\mathit{map}\{\!|\mathsf{UNIT}|\!\} && x && = x \\
&\mathit{map}\{\!|\mathsf{Int}|\!\} && i && = i \\
&\mathit{map}\{\!|\mathsf{Char}|\!\} && c && = c \\
&\mathit{map}\{\!|\mathsf{EITHER}|\!\}\ \mathit{map}_a\ \mathit{map}_b && (\mathit{LEFT}\ x) && = \mathit{LEFT}\ (\mathit{map}_a\ x) \\
&\mathit{map}\{\!|\mathsf{EITHER}|\!\}\ \mathit{map}_a\ \mathit{map}_b && (\mathit{RIGHT}\ y) && = \mathit{RIGHT}\ (\mathit{map}_b\ y) \\
&\mathit{map}\{\!|\mathsf{PAIR}|\!\}\ \mathit{map}_a\ \mathit{map}_b && (\mathit{PAIR}\ x_1\ x_2) && = \mathit{PAIR}\ (\mathit{map}_a\ x_1)\ (\mathit{map}_b\ x_2) \\
&\mathit{map}\{\!|\mathsf{CONS}|\!\}\ \mathit{map}_a && (\mathit{CONS}\ x) && = \mathit{CONS}\ (\mathit{map}_a\ x) \\
&\mathit{map}\{\!|\mathsf{FIELD}|\!\}\ \mathit{map}_a && (\mathit{FIELD}\ x) && = \mathit{FIELD}\ (\mathit{map}_a\ x) \\
&\mathit{map}\{\!|\mathsf{OBJECT}|\!\}\ \mathit{map}_a && (\mathit{OBJECT}\ x) && = \mathit{OBJECT}\ (\mathit{map}_a\ x)
\end{aligned}$$

The type of *map* makes use of two type variables and is equivalent to the Generic Haskell type $(\mathit{map}\{\!|\mathsf{a}, \mathsf{b}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{b}$ or the kind-indexed type signature

$$map\{\!|\mathsf{a} :: \kappa|\!\} :: Map\{\![\kappa]\!\}\ \mathsf{a}\ \mathsf{a}$$
$$Map\{\![\star]\!\}\qquad \mathsf{a}\ \mathsf{b} = \mathsf{a} \rightarrow \mathsf{b}$$
$$Map\{\![\kappa \rightarrow \kappa']\!\}\ \mathsf{f}\ \mathsf{g} = \forall \mathsf{a} :: \kappa\ (\mathsf{b} :: \kappa)\ .\ Map\{\![\kappa]\!\}\ \mathsf{a}\ \mathsf{b} \rightarrow Map\{\![\kappa']\!\}\ (\mathsf{f}\ \mathsf{a})\ (\mathsf{g}\ \mathsf{b}).$$

As before, Clean leaves the dependency of *map* on itself implicit, but otherwise uses type signatures similar to Generic Haskell.

Function update. Reusing the definition of *map* to define *update* is not possible in Clean, as it supports neither default cases nor higher-order generic functions. To define *update*, we have to reimplement the *map* function plus the special case for Salary.

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. Generic functions in Clean do not work for types with higher-order kinds, so the generic programming extension of Clean is not fully reflexive.

Type universes. Clean can define generic functions on subsets of data types in the same way as Generic Haskell, but it does not support default cases or generic views.

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.

The type of a generic function. The type of a generic function is declared using the **generic** construct. 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. To use a generic function g on a data type `t` we write **derive** g `t`; no type-specific code is needed.

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

Code optimization. Because Clean uses essentially the same implementation technique as Generic Haskell, there is a risk that specialized code is inefficient. There is extensive work on optimizing specialized code for generic functions generated by Clean [4, 5], and the resulting code is almost as efficient as hand-written specialized code. Not all optimization algorithms have been included in the compiler yet.

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 [60] as well as a GUI library [1].

4.2 PolyP

PolyP [48] is an extension of Haskell with a construct for defining so-called polytypic programs. There are two versions of PolyP: the original version [48], called PolyP1 from now on, is an extension of Haskell that requires a compiler to compile the special constructs for generic programming. The second version, PolyP2 [81], is a lightweight approach, with an optional compiler for generating the necessary code for a data type. In this section we will mainly describe PolyP1, but we will sometimes use PolyP2 to explain special constructs. If the distinction is not important, we will use PolyP.

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).

PolyP1 is rather similar to Generic Haskell in that it translates data types to structure-representation types. The structure-representation types are then used together with polytypic definitions to generate Haskell code for (applications of) generic functions. 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.

Structure-representation types are expressed in terms of bifunctors. In PolyP2, bifunctors are defined by:

```
data (g + h) a b     = InL (g a b) | InR (h a b)
data (g ∗ h) a b     = g a b :*: h a b
newtype Par a b      = ParF   {unParF   :: a}
newtype Rec a b      = RecF   {unRecF   :: b}
newtype (d@g) a b    = CompF {unCompF :: d (g a b)}
newtype Const t      = ConstF{unConstF :: t}
data Empty           = EmptyF.
```

Binary functors are sums (+, with constructors *InL* and *InR*) of products (∗, with constructor :*:) of either the parameter type of kind $\star$ (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, and so on, 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, PolyP uses the following internal representations:

$$\begin{aligned}
&\mathsf{FunctorOf\ List} \ == \mathsf{Empty} + \mathsf{Par} * \mathsf{Rec}\\
&\mathsf{FunctorOf\ Rose} == \mathsf{Par} * \mathsf{List@Rec}\\
&\mathsf{FunctorOf\ Fork} == \mathsf{Par} * \mathsf{Par}.
\end{aligned}$$

There is an important difference between this encoding of data types and the encoding of data types in Generic Haskell. 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.

In PolyP1, bifunctors are only used internally to construct structure-representation types. Furthermore, Empty is called (), and Const is called Con. Bifunctors can only appear in the type cases of a generic (called polytypic in PolyP) function. Furthermore, the constructors and destructors are added automatically.

An important recursion combinator in PolyP is the *catamorphism* [75], which is defined in PolyLib, the library of PolyP [49]. 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. The catamorphism is intimately tied to the representation of data types as fixed points of bifunctors; it is impossible to define the catamorphism if this fixed point is not explicitly available (as in Generic Haskell).

```
cata     :: Regular d ⇒ (FunctorOf d a b → b) → (d a → b)
cata alg = alg . fmap2 id (cata alg) . 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   → id
        Par     → p
        Rec     → r
        d@g     → pmap (fmap2 p r)
        Const t → id
```

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
      Rec        → id
      d@g        → encode (fencode enca)
      Const Int  → encodeInt
      Const Char → encodeChar
foldSum :: (g a b → c) → (h a b → c) → ((g + h) a b → c)
```

Function decode. Function *decode* is the inverse of function *encode*. It is defined in terms of function *decodes*:

```
decodes        :: Regular d ⇒ Parser a → Parser (d a)
decodes deca = mapP inn (fdecodes deca (decodes deca))
polytypic fdecodes :: Parser a → Parser b → Parser (f a b) =
  λdeca decb →
    case f of
      g + h      → bitCase (mapP Left   (fdecodes deca decb))
                           (mapP Right (fdecodes deca decb))
      g ∗ h      → λbits →  [ ((x, y), r2)
                            | (x, r1) ← fdecodes deca decb bits
                            , (y, r2) ← fdecodes deca decb r1]
      Empty      → λbits → [((), bits)]
      Par        → deca
      Rec        → decb
      d@g        → decodes (fdecodes deca decb)
      Const Int  → decodesInt
      Const Char → decodesChar.
```

Given the definition of function *encode*, the definition of functions *decode* (omitted) and *decodes* is rather standard. We have used a list comprehension in the product case of function *fdecodes* to stay as close as possible to the implementation of *decodes* in Generic Haskell. List comprehensions are not supported by

PolyP, so to compile the program, this piece of code should be replaced by its equivalent not using list comprehensions.

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 [49].

Function update. It is impossible to define a generic function in PolyP that can be used to update the salaries in a Company value. First, the data type Company does not have kind $\star \to \star$. But even if we add a superfluous type variable to the data type Company, PolyP does not 'look into' the constituent Dept values, and hence never changes a Salary. The only way to update a salary in a company structure is by defining Company as one big recursive data type, 'inlining' the definitions of most of the constituent data types, and by adding a superfluous type variable.

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 $\star \to \star$. Important classes of data types for which polytypic functions do not work are mutually-recursive data types and data types of kind $\star$.

Type universes. PolyP only works on regular data types of kind $\star \to \star$. Besides the obvious disadvantages, this has an advantage as well: since the structure of regular data types of kind $\star \to \star$ 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 without the concept of generic views [45]. PolyP supports defining generic functions on particular data types using the Const case.

First-class generic functions. Polytypic functions are not first class in PolyP1. In the lightweight approach PolyP2 polytypic functions are 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.

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 [57, 50] show how to reason about polytypic functions, and how to derive a property of a polytypic function from its type.

Integration with the underlying programming language. The integration of polytypic programming and Haskell is not completely seamless. PolyP1 and the optional compiler of PolyP2 do not know about classes, or types of kind other than $\star \rightarrow \star$, and lack several syntactic constructions that are common in Haskell, such as **where** clauses and operator sections. It is wise to separate the polytypic functions from other functions in a separate file, and only compile this file with PolyP1 or PolyP2. The Haskell library part of PolyP2 integrates seamlessly with Haskell.

Polytypic functions can be used on values of data types without any extra work. It is not necessary to specify a type argument: PolyP1 infers the data types on which a polytypic function is called, and uses this information to specialize a polytypic function for a particular data type.

Specialization versus interpretation. PolyP1 and the optional PolyP2 compiler specialize applications of polytypic functions at compile-time. The PolyP2 Haskell library interprets bifunctors at run time.

Code optimization. Like Generic Haskell, PolyP1 does not optimize away the extra marshaling that is introduced by the compiler for instances of polytypic 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 could be improved.

4.3 Scrap Your Boilerplate

Scrap Your Boilerplate (SYB) [61, 64] 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

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

Fig. 2. Partial definition of the type class *Data*

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 2.

The function *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

$$C\ v_1\ v_2 \ldots v_n$$

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 (λx y → Id (unId x y)) Id
```

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

```
newtype Id a = Id{ unId :: a }
```

is used, because the result type of f a of *gfoldl* can be instantiated to 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 x, 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:

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

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. 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, and trees: 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*:

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

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-arm *type-case* [83]. Using *extQ*, we can write *encode* with type-specific behavior for Ints and Chars:

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

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 (this is solved by the modified approach discussed in the section on “SYB with Class”).

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*:

$$
\begin{aligned}
gunfold &:: \forall \mathsf{a}\ \mathsf{f}\,.\\
&\quad (\forall \mathsf{a}\ \mathsf{b}\,.\, Data\ \mathsf{a} \Rightarrow \mathsf{f}\ (\mathsf{a} \rightarrow \mathsf{b}) \rightarrow \mathsf{f}\ \mathsf{b})\\
&\rightarrow (\forall \mathsf{a}\,.\, \mathsf{a} \rightarrow \mathsf{f}\ \mathsf{a})\\
&\rightarrow \mathsf{Constr} \rightarrow \mathsf{f}\ \mathsf{a}.
\end{aligned}
$$

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 $fromConstrM$, which monadically constructs a value of a certain constructor:

$$
\begin{aligned}
&fromConstrM :: \forall \mathsf{a}\ \mathsf{f}\,.\,(Data\ \mathsf{a}, Monad\ \mathsf{f}) \Rightarrow (\forall \mathsf{b}\,.\, Data\ \mathsf{b} \Rightarrow \mathsf{f}\ \mathsf{b}) \rightarrow\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Constr} \rightarrow \mathsf{f}\ \mathsf{a}\\
&fromConstrM\ p = gunfold\ (`ap`p)\ return.
\end{aligned}
$$

Here, $ap :: \forall \mathsf{a}\ \mathsf{b}\ \mathsf{f}\,.\, Monad\ \mathsf{f} \Rightarrow \mathsf{f}\ (\mathsf{a} \rightarrow \mathsf{b}) \rightarrow \mathsf{f}\ \mathsf{a} \rightarrow \mathsf{f}\ \mathsf{b}$ is lifted function application.

Using $fromConstrM$, we can define $decodes$, but since $fromConstrM$ requires a monad, we have to turn our parser type into a monad. Recall that

$$\mathbf{type}\ \mathsf{Parser}\ \mathsf{a} = [\mathsf{Bit}] \rightarrow [(\mathsf{a}, [\mathsf{Bit}])].$$

We turn Parser into a state monad by wrapping it into a **newtype** construct and defining appropriate class instances:

```
newtype ParserM a = M{ runM :: Parser a }
instance Monad ParserM where
  return x = M (λs → [(x, s)])
  f >>= g  = M (λs → [r | (x, s′) ← runM f s, r ← runM (g x) s′])
instance MonadState [Bit] ParserM where
  get   = M (λs → [(s, s)])
  put s = M (λ_ → [((), s)]).
```

The code for $decodes$ is then defined as follows:

```
decodes :: Data a ⇒ Parser a
decodes     = decodes′ ⊥
                `extR` decodesInt
                `extR` decodesChar
  where
    decodes′          :: Data a ⇒ a → Parser a
    decodes′ dummy = runM $
      do let d   = dataTypeOf dummy
             l   = length (int2bits (length (dataTypeConstrs d) − 1))
         c ← consume l
```

$$\textbf{let } con = decodeConstr\ c\ d$$
$$fromConstrM\ (M\ decodes)\ con.$$

A few remarks are in order. The function *decodes* calls *decodes′* with ⊥. 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. In addition, *decodes′* performs the necessary conversions between Parser and *ParserM*.

The functions *consume* and *decodeConstr* are both easy to define. Type-specific behavior for integers and characters is added to *decodes* using the SYB extension operator *extR*, which plays a role analogous to *extQ*, in the context of monadic generic producers:

$$extR :: \forall \mathsf{a}\ \mathsf{b}\ \mathsf{f}\,.\,(Monad\ \mathsf{f}, Typeable\ \mathsf{a}, Typeable\ \mathsf{b}) \Rightarrow \mathsf{f}\ \mathsf{a} \to \mathsf{f}\ \mathsf{b} \to \mathsf{f}\ \mathsf{a}.$$

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

```
decode :: Data a ⇒ [Bit] → a
decode bs = case 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.

The trick to relax the type of a generic function is not always applicable. For example, if we also want to extend equality on an abstract type for which we only have a normal equality function (one that expects two arguments of the same type), we have to make sure that both arguments are indeed of the same type. In this case, we can use the dynamically available type information from class *Typeable* to define a unification function

$$unify :: (Typeable\ \mathsf{a}, Typeable\ \mathsf{b}) \Rightarrow \mathsf{Maybe}\ (\mathsf{a} \to \mathsf{b})$$

and then call *unify* to coerce the types where necessary.

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 $\star$. 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

$$\forall \mathsf{a}\ \mathsf{b}\ \mathsf{f}\,.\,(\mathsf{a} \to \mathsf{b}) \to \mathsf{f}\ \mathsf{a} \to \mathsf{f}\ \mathsf{b},$$

where the arguments of the container type f are modified, and the function is parametrically polymorphic in a and b (see also 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

$$ext1Q :: \forall \mathsf{a}\ \mathsf{c}\,.\,(Typeable1\ \mathsf{f}, Data\ \mathsf{a}) \Rightarrow$$
$$(\mathsf{a} \to \mathsf{c}) \to (\forall \mathsf{b}\,.\,Data\ \mathsf{b} \Rightarrow \mathsf{f}\ \mathsf{b} \to \mathsf{c}) \to (\mathsf{a} \to \mathsf{c}).$$

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 $\star \to \star$ 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:

$$update :: Data\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{a}$$
$$update = everywhere\ (id\ \text{‘}extT\text{‘}\ (\lambda(S\ s) \to S\ (s * (1 + 0.15)))).$$

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 (constructor) in a data structure. It is defined in terms of

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

a variant of *gmapQ* that applies a given generic function to the immediate subterms of a value. The *gmapT* can again be defined using *gfoldl*. Note that all these functions similar to, but different from the generic *map* function, which applies an argument function to all occurrences of values of a parameter type in a data type of a higher kind.

Derived work: SYB with Class. Lämmel and Peyton Jones have shown [62] 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.

To achieve this added flexibility, some alterations to the class *Data* are required. The class must be parametrized over a *context* parameter:

```
class (Typeable a, Sat c a) ⇒ Data c a where
  toConstr    :: Proxy c → a → Constr
  dataTypeOf  :: a → DataType
  gfoldl      :: ∀f . Proxy c
              → (∀a b . Data c a ⇒ f (a → b) → a → f b)
              → (∀a . a → f a)
              → a → f a.
```

The context parameter c together with the class constraint on *Sat* c a simulates abstracting over a superclass: recursive generic functions are defined as a class.

Because the class methods make use of the generic combinators such as *gfoldl* or derived combinators such as *gmapQ*, *Data* must be a superclass of the class of the function. But because the *Data* constraint occurs inside the type of the generic combinators such as *gfoldl*, the class of the function must also be a superclass of *Data*. This is not directly possible, hence the encoding via the context parameter.

The presence of this encoding leads to a number of encumbrances and subtleties in the "SYB with Class" approach. Sometimes, Haskell is not clever enough to figure out the correct instantiation of the context parameter itself. Therefore, the class methods of *Data* all take an additional parameter of type Proxy c, with the sole purpose to make the instantiation of c explicit. Furthermore, the possible instantiations of c are dictionary types that have to be defined for each generic function (or group of mutually recursive generic functions).

As an example, let us look at *encode* again. In the class-based approach, we define *encode* simply as follows:

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

However, to turn it into a generic definition, we must now define a suitable context to use in the argument of *Data*. This requires the following definitions:

```
data Encode a = Encode{ encodeD :: a → [Bit] }
encodeProxy :: Proxy Encode
encodeProxy = ⊥
instance Encode a ⇒ Sat Encode a where
  dict = Encode{ encodeD = encode }.
```

The class *Sat* need only be defined once and is given simply as

```
class Sat c a where
  dict :: c a.
```

We are now in a position to give the generic definition of *encode*:

```
instance (Data Encode a) ⇒ Encode a where
  encode x = concat (encodeConstr (toConstr encodeProxy x) :
                     gmapQ encodeProxy (encodeD dict) x).
```

If we compare this definition with the definition of *encode* in original SYB style on page 110, then there are only few differences: first, the type-specific cases are missing (they can be added later using specific class instances); second, the proxy arguments are passed (also *gmapQ* takes a proxy argument now) to help the type checker along; third, the recursive call of *encode* is replaced by *encodeD dict*. The latter is because the argument to *gmapQ* must actually have type ∀a . *Data* Encode a ⇒ a → [Bit] in this case, and the direct use of *encode* would introduce an illegal constraint on *Encode* a.

Type-specific cases can now be defined separately (and later) as additional instances of *Encode*:

$$
\begin{aligned}
&\textbf{instance } \mathit{Encode}\ \mathsf{Int}\ \textbf{where} \\
&\quad \mathit{encode} = \mathit{encodeInt} \\
&\textbf{instance } \mathit{Encode}\ \mathsf{Char}\ \textbf{where} \\
&\quad \mathit{encode} = \mathit{encodeChar}.
\end{aligned}
$$

As we can see from this example, there is a significant advantage to using SYB with classes, but there are disadvantages as well: the user has additional work, because for each generic function, an additional context type, a proxy, and an embedding instance for *Sat* must be defined. The use of *dict* rather than direct recursive calls, and the passing of proxy arguments is quite subtle. Furthermore, the class structure used here requires the GHC extensions of overlapping and undecidable instances.

Derived work: SYB Reloaded and Revolutions. In their SYB Reloaded and Revolutions papers, Hinze, Löh and Oliveira [44, 43] 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 parametrized 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. This implies, amongst others, that higher-kinded data types such as GRose cannot be turned into instance declarations as this requires so-called *higher-order contexts*. The original proposal for Derivable Type Classes (discussed in Section 4.5) recognizes this shortcoming and proposes a solution in the form of higher-order contexts, but this extension has never been implemented.

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

Type universes. There is no support for type universes in SYB. All generic functions are supposed to work on all types in the *Typeable* class.

First-class generic functions. In SYB, generic functions are normal polymorphic Haskell functions, and as such are first-class. However, 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.

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

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

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.

The type of a generic function. Types of generic functions have one or more constraints for the *Data* class. The types are intuitive, maybe except for the generic combinators such as *ext1Q* and *gunfold*.

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 [88], but he translates SYB combinators into Generic Haskell to do so.

Integration with the underlying programming language. SYB is fully integrated into GHC. Making SYB available for Hugs or another Haskell compiler would be a major effort. 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. No extra work is needed to use a generic function on a data type other than writing **deriving** (*Data*, *Typeable*) after the data-type declaration.

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 passing of class dictionaries, the type casts, and the use of many higher-order functions might sometimes lead to a considerable overhead.

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.

4.4 Approaches based on reflection

Both DrIFT [99] and generic programming approaches using Template Haskell [82] use a kind of reflection mechanism to generate instances of generic functions for a data type. Generic functions are defined on an abstract syntax for data types. This section introduces and evaluates these two approaches.

DrIFT

DrIFT 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 vars
             , 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 example for putting two documents beside each other (<+>) (list version *hsep*), above each other $$ (list version *vcat*), and for printing texts (*text* and *texts*) [47]. Constructing output using pretty-printing combinators is easier and more structured than manipulating strings.

Function encode. We 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 generates:

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

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 $[[O,O],[O,I],[I,O]]$. As before, we use the utility function *intinrange2bits* to encode a natural number in a given range.

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 of variable names, the length of which is equal to the length of the argument list. 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     :: Parser a
```

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

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) = [(Cons res1 res2, xs2) | (res1, xs1) ← decodes xs
                                              , (res2, xs2) ← decodes xs1]
  decodes []       = error "decodes".
```

The decode function generates an instance of the class *Decode*. It adds the declaration of *decodes* on the empty list as the last line in each class instance.

```
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. The implementation of function *mkDecodefn* is almost a page of Haskell code, and can be found in the accompanying technical report [40].

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 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 **where** ..., 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.
```

Then 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).
```

It is impossible to define a generic map that works on types of different kinds for many of the other approaches to generic programming. DrIFT allows us to do anything we want, which we illustrate by defining map in an alternative fashion.

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{ constr = constructor, types = types }) =
  let mt name = text ("map" ++ name)
      mapArgs  = hsep (texts (map (λv → 'f' : v) vars))
      newVars  = varNames types
      conPat   = parens . hsep $ text constr : newVars
      lhs      = mt name <+> mapArgs <+> conPat
      rhs      = hsep (text constr
                        : 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"
```

Template Haskell

Template Haskell is a language extension that allows meta-programming within the Haskell language. Template Haskell consists of multiple components.

A library (exported by *Language.Haskell.TH*) provides access the the abstract syntax of the Haskell language. This makes it possible to analyze and construct Haskell programs within Haskell. A monad Q is provided to generate fresh names on demand.

Haskell expressions can be *quoted* to easily construct terms in the abstract syntax. For example,

$$[\![2 + 2]\!] :: \mathsf{Q\ Exp}.$$

Template Haskell supports reflection (*reification*), so that it is possible to analyze the structure of an already defined value or data type:

$reify$ :: Name → Q Info.

The Info data type has multiple constructors corresponding to different kinds of declarations, but in particular, there is a constructor for data types:

data Info = ...
 | *TyConI* Dec
data Dec = ...
 | *DataD* Cxt Name [Name] [Con] [Name]
data Con = *NormalC* Name [StrictType]
 | *RecC* Name [VarStrictType]
 | *InfixC* StrictType Name StrictType
 | *ForallC* [Name] Cxt Con.

Each data type comprises a context (possible class constraints), a name, a list of parameters, a list of constructors, and a list of classes it automatically derives. Constructors can either be normal constructors, records, infix constructors, or constructors with quantification. A StrictType is a type with a possible strictness annotation, a VarStrictType additionally contains a record label.

Finally, in Template Haskell we can *splice* values constructed in the abstract syntax into Haskell programs, making it possible to run programs that are generated by meta-programs. Splicing is dual to quoting, so that

$\$([\![2 + 2]\!])$ :: Int

results in 4.

By its very nature, Template Haskell can be used to write programs that cannot be expressed, or are at least difficult to express, in the Haskell language, such as generic programs. With Template Haskell, we can analyze data-type definitions, and depending on their structure, generate specialized code.

It is important to realize that Template Haskell itself is not an approach to generic programming, but more like an implementation technique. Template Haskell gives the programmer a lot of power and freedom, but does not provide any guidance or even a framework for generic programming.

While DrIFT's main focus is to generate type-class instances, we can use Template Haskell much more flexibly:

– we can generate the structure-representation type (like in Generic Haskell) for a given data type, plus the embedding-projection pairs;
– for a generic function, we can construct a recipe that uses the abstract syntax of a data type to construct the abstract syntax of a specialized instance of the generic function;
– we can generate instances of a type class, both for a generic function directly (like in DrIFT, Derivable Type Classes, or Generics for the Masses), or for a powerful combinator like *gfold* in Scrap Your Boilerplate.

In principle, Template Haskell can be used to simulate or support any approach to generic programming in Haskell. However, we also run into many of the problems that we encountered in DrIFT:

- everything happens at the syntactic level, not the semantic level. While constructing generic functions, we have to pay attention to low-level concepts such as free and bound variables;
- the analysis of data types is also purely syntactic. We do not have access to kind information, or recursion on the type level, directly, but have to infer that from the definitions; writing generic functions for mutually recursive data types or higher-kinded data types is difficult, because we have to implement parts of a compiler;
- there is no guarantee that the meta-programs produce correct code under all circumstances. The generated code is type-checked, so we are safe from errors in the end, but this is a much weaker guarantee than we get from other approaches such as Generic Haskell, where we know that the type-correctness of a generic definition implies the type-correctness of all instances.

Because of the above-mentioned freedom, it is difficult to implement the canonical examples for generic programming using Template Haskell: there is no single idiomatic version of a generic function, but there are many different possibilities. We therefore don't include specific examples in this document.

We are aware of one attempt to provide a serious framework for generic programming within Template Haskell: Norell and Jansson [82] present a very sophisticated embedding of both PolyP and Generic Haskell into Template Haskell. Among other things, they describe how to define generic *map* in the two different encodings.

Evaluation

Structural dependencies. DrIFT and Template Haskell support the definition of functions that take the abstract syntax of a data type as an argument, and return executable Haskell code. In principle, both DrIFT and Template Haskell can generate any document, even type-indexed data types. Especially for DrIFT, generating anything other than class instances amounts to writing part of a compiler for generic programming. In Template Haskell, it is feasible to design at least a reusable framework for such advanced tasks. Both systems provide no way to access type or kind information of the analyzed code. In particular, the lack of kind inference for data types makes the creation of generic programs on complex data types tedious.

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 higher-kinded type variables, such as GRose. Just like Generic Haskell, DrIFT cannot generate instances of functions on existential types or on GADTs.

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

Template Haskell's abstract syntax handles all of Haskell 98 and beyond. It does not yet support GADTs, but there is no reason why it could not be extended in that way. Full reflexivity therefore depends on the generic programming approach one tries to simulate within Template Haskell.

Type universes. There is no support for type universes in DrIFT. Neither does Template Haskell have any direct support for this concept.

First-class generic functions. DrIFT rules are plain Haskell functions, they can take rules as arguments. First-class rules are inherited from Haskell. But it needs a lot of imagination to see rules as generic programs. And 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 (the only supported kind of definition in DrIFT) cannot be passed as argument to a rule that generates a function or a class instance.

Similarly, we have all the abstraction possibilities of Haskell for generic programs within Template Haskell. We can write generic meta-programs that are parametrized over other generic meta-programs.

However, both DrIFT and Template Haskell are two-level approaches. DrIFT always needs to be invoked before compilation of a Haskell module to fill in the missing code. Template Haskell requires splicing of the generated code. Splicing is a syntactic construct which is foreign to the Haskell language and furthermore underlies certain restrictions (sometimes, code that contributes to Template Haskell programs must reside in several modules). Therefore, DrIFT and Template Haskell cannot provide generic functions that are truly first-class.

Multiple type arguments. Rules cannot take multiple type arguments in DrIFT. In Template Haskell, there are no theoretical limits.

Type system. Rules for generic functions all have the same type in DrIFT: $\mathsf{Data} \to \mathsf{Doc}$. There is no separate type system for rules; rules are ordinary Haskell functions. In Template Haskell, the situation is similar. All Haskell expressions, for instance, are of type Exp in the abstract syntax of expressions, but no further type information about the actual constructed expression is maintained. In particular, it is possible to construct type-incorrect expressions, causing type errors only when spliced.

Note that in addition to type errors, it is easy to generate lexer and parser errors in DrIFT.

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. The same holds for Template Haskell, where the type correctness of a meta-program does not imply that the use of that meta-program produces type-correct code.

The type of a generic function. In DrIFT, every rule has type $\mathsf{Data} \to \mathsf{Doc}$. Thus it is impossible to distinguish generic functions by type. For Template Haskell, the type of generic functions depends completely on the approach that is simulated. Generally, however, not much of a generic function's type is reflected in the type of the meta-program: as in DrIFT, generic functions in Template Haskell typically map the abstract syntax of one or more data types to a number of Haskell declarations. Lynagh [74] shows how to give more informative types to Template Haskell programs.

Properties of generic functions. Since rules generate pretty-printed documents (syntax), it is virtually impossible to specify properties of rules. For Template Haskell, it is similarly impossible to specify properties. However, libraries for generic programming defined in Template Haskell may allow to state and prove properties.

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. Template Haskell is superior here, because Template Haskell code can almost freely be mixed with normal Haskell code. Sometimes, code has to be divided in separate modules.

Specialization versus interpretation. DrIFT specializes rules on data types following directives. Template Haskell also generates the programs in advance, but a hybrid approach is conceivable: in the simulation of a lightweight approach, some code would be generated for each data type, but a generic function would be interpreted.

Code optimization. Code can be optimized by hand by specifying a more sophisticated rule or meta-program. There need not be a run-time efficiency penalty when using DrIFT or Template Haskell.

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. Separate compilation in Template Haskell is possible because of its integration with Haskell.

Practical aspects. DrIFT is actively maintained. The last release is from April 2006. It runs on many platforms. The user guide explains how to use DrIFT. Template Haskell is actively maintained as part of GHC; the flag `-fth` must be passed to GHC to be able to use it. Template Haskell is, however, still in development, with new GHC releases regularly changing the interface in an incompatible way. Documentation for the current state of affairs is difficult to come by, but this situation is likely to improve when the speed of development slows down.

No error messages are given for data types for which DrIFT cannot generate code. Error messages provided by Template Haskell are often in terms of the generated code and difficult to interpret for the user of a generic programming library.

4.5 Lightweight approaches to generic programming

Due to Haskell's advanced type language and type classes it is possible to write generic programs in Haskell itself, without extending the language. An approach in which generic programs are plain Haskell programs is called a lightweight approach. Lightweight approaches to generic programming in Haskell have become popular in the last couple of years. In this section we discuss three relatively lightweight approaches to generic programming: "A Lightweight Implementation of Generics and Dynamics", "Generics for the Masses", and "Derivable Type Classes". The last approach is actually a language extension, but since it shares many characteristics with the other two approaches, it is listed here.

We do not include a comparison of some very recent lightweight approaches to generic programming such as Replib [97], Smash your boiler-plate without class and Typeable [59], and TypeCase [83]. Neither do we discuss PolyP2 here: the subsection on PolyP discusses the main ideas behind PolyP.

Lightweight Implementation of Generics and Dynamics

Lightweight Implementation of Generics and Dynamics [15] (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 [15] or to the companion lecture notes on Generic Programming, Now!, which elaborate on a closely related approach to generic programming.

A generic function in Generic Haskell is parametrized by types, 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 typecase can be implemented by ordinary pattern matching. As a first try, we could, for instance, assign the generic *encode* function the type Rep → t → [Bit], where Rep is the type of type representations. A moment's reflection, however, reveals that this won't work. The parametricity theorem [92] implies that a function of this type necessarily ignores its second argument. The trick is to use a parametric type for type representations: *encode* :: Rep t → t → [Bit]. Here Rep t is the type representation of t. In this section we will show a number of ways in which such a type can be defined.

Using a recent extension to Haskell, so-called generalized algebraic data types, Rep can be defined directly in Haskell; see also Generic Programming, Now! (Section 3.1 in [42], where Rep is called Type).

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

A type t is represented by a term of type Rep t. Note that the above declaration cannot be introduced by a Haskell 98 data declaration since none of the data constructors has result type Rep 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 4.5. The idea is to assign, for instance, *Int*, the representation of Int, the type Rep t with the additional constraint that t = Int. The type equality is then encoded using the equivalence type $a \leftrightarrow b$ introduced in Section 2.2. An element of $t \leftrightarrow t'$ 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 3 displays the fully-fledged version of Rep that uses equivalence types. The constructors *Unit*, *Int*, *Char*, *Sum*, *Pair* and

```
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. 3. A type-representation type.

Con correspond to the type patterns Unit, Int, Char, :+:, :*: 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 a burden 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 or Template Haskell to generate the code automatically.) To see what is involved, re-consider the List data 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   :: Rep a → Rep (List a)
list a = Type ((Con "Nil" unit) + (Con "Cons" (a ∗ (list a))))
              (EP fromList toList),
```

where *unit*, + and ∗ 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                :: List a → Unit :+: (a :*: (List a))
fromList Nil            = Inl Unit
fromList (Cons a as)    = Inr (a :*: as)
toList                  :: 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 per data type.

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

```
encode                  :: 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 and values of other types into an element of the result type t using *to ep*.

```
decodes                  :: Rep t → Parser t
decodes (Unit      ep) bs = [(to ep Unit, bs)]
decodes (Char      ep) bs = mapP (to ep) decodesChar bs
decodes (Int       ep) bs = mapP (to ep) decodesInt bs
decodes (Sum a b   ep) bs = bitCase (mapP (to ep . Inl) (decodes a))
                                    (mapP (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]
```

$$
\begin{array}{ll}
\textit{decodes}\ (\textit{Type}\ a\ \ \ \textit{ep})\ \textit{bs} = \textit{mapP}\ (\textit{to}\ \textit{ep})\ (\textit{decodes}\ a)\ \textit{bs} \\
\textit{decodes}\ (\textit{Con}\ s\ a)\ \ \ \ \ \textit{bs} = \textit{decodes}\ a\ \textit{bs}
\end{array}
$$

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                       :: 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 on 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                   :: 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* [72]. The code for *update* is then entirely straightforward and omitted for reasons of space.

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):

```
class Eq a where
  eq :: a → a → Bool.
```

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

```
eq :: (Eq a) ⇒ a → a → Bool.
```

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

```
instance Eq Int where
  eq = eqInt.
```

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:

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

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

$$\textbf{data}\ \mathsf{List\ a} = Nil \mid Cons\ \mathsf{a}\ (\mathsf{List\ a})\ \textbf{deriving}\ (Eq),$$

then Haskell generates the 'obvious' code for equality. What 'obvious' means is specified informally in an Appendix of the language definition [86]. *Derivable type classes* (DTCs) [41] 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*.

$$
\begin{array}{l}
\textbf{class } Binary\ \mathsf{a}\ \textbf{where} \\
\quad encode :: \mathsf{a} \to [\mathsf{Bit}] \\
\quad decodes :: \mathsf{Parser\ a}
\end{array}
$$

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

$$
\begin{array}{lll}
encode\{\!|\mathsf{Unit}|\!\}\ Unit & = [\,] \\
encode\{\!|\mathsf{b :+: c}|\!\}\ (Inl\ x) & = O : encode\ x \\
encode\{\!|\mathsf{b :+: c}|\!\}\ (Inr\ y) & = I : encode\ y \\
encode\{\!|\mathsf{b :*: c}|\!\}\ (x :\!*\!: y) & = encode\ x +\!\!+\ encode\ y \\
decodes\{\!|\mathsf{Unit}|\!\}\ bs & = [(Unit, bs)] \\
decodes\{\!|\mathsf{b :+: c}|\!\}\ bs & = bitCase\ (mapP\ Inl\ decodes) \\
 & \qquad (mapP\ Inr\ decodes) \\
 & \qquad bs \\
decodes\{\!|\mathsf{b :*: c}|\!\}\ bs & = [(x :\!*\!: y, ds) \mid (x, cs) \leftarrow decodes\ bs \\
 & \qquad\qquad\qquad , (y, ds) \leftarrow decodes\ cs]
\end{array}
$$

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 (Binary a) ⇒ Binary [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 → Parser a → Parser [a]
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 — 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 Eq Char where
  eq = eqChar
instance Eq 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. A missing feature of DTCs is a *c* **of** 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*.

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{|b :*: 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.

Generics for the Masses

Generics for the Masses [35, 36] (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 [36].

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 a → [Bit]. Data compression does not work for arbitrary types, but only for types that are *representable*, that is, where 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 one 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)
```

$$
\begin{aligned}
&\mathit{pair} = \mathit{Encode}\ (\lambda x \to \mathit{encode}\ (\mathit{outl}\ x) +\!\!+\ \mathit{encode}\ (\mathit{outr}\ x))\\
&\mathit{datatype}\ \mathit{descr}\ \mathit{iso}\\
&\qquad = \mathit{Encode}\ (\lambda x \to \mathit{encode}\ (\mathit{from}\ \mathit{iso}\ x))\\
&\mathit{char} = \mathit{Encode}\ (\lambda x \to \mathit{encodeChar}\ x)\\
&\mathit{int}\ \ = \mathit{Encode}\ (\lambda x \to \mathit{encodeInt}\ \ \ x)
\end{aligned}
$$

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 it becomes clear that GM uses the same structure types as Generic Haskell. The function *from* is the record selector *from* of the data type $\cdot \leftrightarrow \cdot$ introduced in Section 2.2.

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 BinTree a = *BTLeaf* a | *BTBin* (BinTree a) (BinTree 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 BinTree, denoted BinTree°.

type BinTree° a = (Constr a) :+: (Constr ((BinTree a) :*: (BinTree a)))

The main work goes into defining two mappings, *fromBinTree* and *toBinTree*, which certify that BinTree a and its structure type BinTree° a are indeed isomorphic.

```
fromBinTree                        :: BinTree a → BinTree° a
fromBinTree (BTLeaf x)             = Inl (Con x)
fromBinTree (BTBin l r)            = Inr (Con (l :*: r))
toBinTree                          :: BinTree° a → BinTree a
toBinTree (Inl (Con x))            = BTLeaf x
toBinTree (Inr (Con (l :*: r)))    = BTBin l r
```

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

```
instance (Rep a) ⇒ Rep (BinTree a) where
  rep = datatype ("BTLeaf" ./ 1 .| "BTBin" ./ 2)    -- syntax
                 (EP fromBinTree toBinTree)         -- semantics
```

The operator ./ turns a constructor name and an arity into a constructor description, and the operator .| combines two alternatives into a data description, see Figure 4. The declaration *rep* 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 instances of generic functions on that data type.

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 → 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) (EP fromList toList)
  constr   = datatype ("Con" ./ 1)            (EP arg Con)
data DataDescr = NoData
               | ConDescr{name :: String,      arity :: Int}
               | Alt     {getl  :: DataDescr, getr  :: DataDescr}
infix  2 ./
infixr 1 .|
f ./ n   = ConDescr{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 :: Parser a}
decodes :: (Rep a) ⇒ Parser a
decodes = applyDecodes rep
instance Generic Decodes where
  unit = Decodes (λbs → [(Unit, bs)])
  plus = Decodes (λbs → bitCase (mapP Inl decodes)
                                (mapP Inr decodes)
                                bs)
  pair = Decodes (λbs → [(x :*: y, ds) | (x, cs) ← decodes bs
                                       , (y, ds) ← decodes cs])
  datatype descr iso
       = Decodes (λbs → mapP (to 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 (from iso x1) (from 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 *Generic2* that allows us to define generic functions whose type is parametrized 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 Generic2 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 → to iso2 (applyMap a (from 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 extend *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 (from 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 "[]"
```

$$
\begin{aligned}
&\mathit{showsl}\ p\ (a : as) &&= \mathit{showChar}\ \texttt{'['} \cdot p\ a \cdot \mathit{rest}\ as\\
&\quad \textbf{where}\ \mathit{rest}\ [\,] &&= \mathit{showChar}\ \texttt{']'}\\
&\qquad\quad \mathit{rest}\ (x : xs) &&= \mathit{showChar}\ \texttt{','} \cdot p\ x \cdot \mathit{rest}\ xs
\end{aligned}
$$

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 [84].

Evaluation

Structural dependencies. All lightweight approaches support the definition of functions in the style of Generic Haskell. Type-indexed data types are out of reach.

Using a different representation type in LIGD we can also define generic functions that are indexed by first- or higher-order kinds (this is not detailed in the original paper).

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. LIGD is in principle fully reflexive. However, to support types of arbitrary ranks, rank-n types are required.

GM is not fully reflexive: for different kinds we need different type representations. But it is possible to construct a family of incompatible GM implementations. 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 SYB.

DTCs also share the limitations of class-based systems: higher-order contexts are needed to apply generic functions to higher-kinded data types such as GRose.

Type universes. By changing the classes for type representations used in LIGD and GM other type universes can be introduced and used. Since type representations are given by the user, they are very flexible.

DTCs support default cases, but otherwise the type universe is fixed.

First-class generic functions. In LIGD, a generic function is an ordinary polymorphic Haskell function of type $\mathit{Rep}\ \mathtt{t} \to \mathit{Poly}\ \mathtt{t}$. As such it is first-class, assuming that rank-n functions are supported.

Similarly, in GM a generic function is an ordinary polymorphic Haskell function of type $(\mathit{Rep}\ \mathtt{t}) \Rightarrow \mathit{Poly}\ \mathtt{t}$. Again, in a language with rank-n types, generic functions are first-class citizens.

In DTCs, 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. In both LIGD and GM a generic function may have multiple type arguments. Derivable type classes may only abstract over a single type argument.

Type system. All approaches are fully integrated into Haskell's type system.

Type safety. All approaches are fully type-safe. A missing type-case in LIGD, 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. In GM a missing case branch issues a warning at compile-time (about a missing method). Since instance declaration must be explicitly provided, missing instances in DTCs are detected at compile-time.

The type of a generic function. The types are intuitive; we only have to prefix a 'Rep `t` $\rightarrow$' argument or a '(Rep `t`) $\Rightarrow$' context for LIGD and GM, respectively. The types of member functions of DTCs are familiar to Haskell programmers.

Properties of generic functions. For all approaches, properties of a generic function can be stated and proven as in Generic Haskell.

Integration with the underlying programming language. All approaches are fully integrated into Haskell. For DTCs, only the module `Data.Generics` need be imported and the options `-fglasgow-exts` and `-fgenerics` must be passed to the GHC.

In LIGD and GM the user has to specify the structure representation type and the embedding-projection pair between the data type and the structure representation type for every data type on which generic functions are used.

In DTCs, a generic function g, implemented in the class G, can be used on a data type `t` by writing **instance** G `t`. No other data-type-specific code is needed.

Specialization versus interpretation. Representations of types are passed and analyzed at run-time in LIGD. A generic function can be seen as an interpreter. In GM, instances of generic functions are assembled at compile-time. In DTCs, generic code is specialized for each instance.

Code optimization. In LIGD, the run-time passing of type representations incurs a small overhead compared to Generic Haskell. For GM and DTCs the overhead is similar to that of Generic Haskell. The code quality possibly depends a bit more on GHC's optimizer.

Separate compilation. All approaches support 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).

GM comprises three major implementations of generics and a few variations. The approach is extremely light weight; 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 type cases), but it can easily be adapted to one's needs.

The original DTCs proposal is partially implemented in GHC, the most popular compiler for Haskell. Names of constructors and labels cannot be accessed in DTCs, so 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.

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 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 (almost) 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 of our evaluations into points on this five-point scale.

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

Fig. 5. Evaluation results for approaches to generic programming

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 and Template Haskell can generate anything, they 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.

The type completeness principle. No approach truly satisfies the type completeness principle.

SYB, GM, and DTCs suffer from the fact that higher-order contexts (not implemented in Haskell) are needed to generate instances of generic functions on higher-kinded data types. On the other hand, both SYB and GM allow higher-order generic functions. Just as with classes, DTCs cannot represent higher-order generic functions. Furthermore, DTCs cannot access constructor names, which limits their usability a bit. LIGD allows higher-order generic functions and generic functions on almost all data types definable in Haskell. However, it is impossible to define the generic map function in LIGD and SYB. GM allows higher-order generic functions, and the definition of generic map, but needs different classes for different brands of generic functions.

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 orders do not really play a rôle in DrIFT and Template Haskell, and DrIFT cannot handle higher-kinded data types. PolyP does not allow higher-order generic functions either and only works for regular data types of kind $\star \to \star$.

Generic views in Generic Haskell allow defining generic functions for different views on data types, which can be used to specify different type universes. LIGD and GM allow very flexible sets of types on which generic functions can be defined, and it is possible to define many type universes. Clean, PolyP, SYB, and DTCs have a fixed type universe. DrIFT and Template Haskell offer no support for type universes.

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

Information in types. In Generic Haskell, Clean, PolyP, and LIGD 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. 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*.

In DrIFT all rules have the same type, namely $\mathsf{Data} \rightarrow \mathsf{Doc}$, and it is virtually impossible to prove anything about the functions represented by the documents. The same holds for Template Haskell, although libraries for generic programming defined in Template Haskell may allow to state and prove properties.

Integration with the underlying programming language. Generic Haskell, Clean, SYB, Template Haskell, LIGD, GM, and DTCs are fully integrated with the underlying programming language, where Clean, SYB, Template Haskell, LIGD, GM, and DTCs 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.

To use a generic function on a new data type, almost no work is required in Generic Haskell, Clean, PolyP, SYB, DrIFT, Template Haskell, and DTCs. In the lightweight approaches LIGD and GM the structure representation type and the embedding-projection pair between the structure representation type and the original data type have to be supplied.

Tools. Generic Haskell, LIGD, GM, and DTCs do not do any optimization on the generic code, but otherwise provide good error messages. Clean does optimize the generated code, but provides no error messages. PolyP is not very actively maintained anymore. 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. DrIFT is maintained, but also provides no error messages. Template Haskell is maintained, but the documentation is outdated, and error messages are not always very helpful.

5.2 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, in particular if you

want to play with higher-kinded and/or nested data types. Generic Haskell is probably the most expressive generic programming extension of Haskell. A disadvantage of using Generic Haskell is 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 into 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 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 $\star \to \star$.
- 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 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 higher-kinded data types.
- Use Template Haskell if you want to experiment with different implementations of generic programming styles.
- Use the LIGD 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 Generics for the Masses if you want a fully Haskell 98 compatible library that supports generic programming.
- Use Derivable Type Classes if you want (limited) Generic Haskell like generic programming functionality fully integrated into the underlying programming language. DTCs don't support type-indexed data types, or higher-kinded data types.

We distinguished three related groups between the nine approaches to generic programming in Haskell described in these lecture notes:

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

PolyP and SYB form their own subcategories (but we might have placed PolyP2 in the lightweight approaches). 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. GM and DTCs use classes for defining generic functions, so higher-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. Also, DTCs allow to extend generic functions with new, type-specific cases without modifying already existing code.

5.3 Future work

These lecture notes only compare approaches to generic programming in Haskell. The only approaches to generic programming in Haskell we have not addressed are Strafunski, Generic Programming, Now!, and several other new lightweight approaches which have appeared only very recently (after the first drafts of these lecture notes were written). Strafunski is rather similar to SYB, but has a more combinator-like, point-free flavor. Generic Programming, Now! is described at length, including a comparison to other approaches, in this volume.

We have yet to perform the same exercise for approaches to generic programming in different programming languages.

Acknowledgements. We thank the participants of the 61st IFIP WG 2.1 meeting for their comments on a presentation about this work. The participants of the Spring School on Datatype-Generic Programming, Nottingham, April 2006 also provided a number of useful suggestions. Jeremy Gibbons, Patrik Jansson, and Ralf Lämmel carefully read a previous version of this paper, and suggested many improvements.

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'04*, volume 3125 of *LNCS*, pages 16–31. Springer-Verlag, 2004.
5. Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In *Proceedings of 7th 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 Gibbons and Jeuring [27], pages 1–20.
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. Customizing an XML-Haskell data binding with type isomorphism inference in Generic Haskell. *Science of Computer Programming*, 65(2):72–107, 2007.
9. Lennart Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 1998*, pages 239–250. ACM Press, 1998.
10. Roland Backhouse and Jeremy Gibbons. The EPSRC project on Datatype-Generic Programming. `http://web.comlab.ox.ac.uk/oucl/research/pdt/ap/dgp/`, 2003–2006.
11. Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.
12. Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Proceedings of the 4th International Conference on Mathematics of Program Construction, MPC'98*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.
13. Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.
14. Juan Chen and Andrew W. Appel. Dictionary passing for polytypic polymorphism. Technical Report TR-635-01, Princeton University, 2001.
15. James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press, 2002.
16. Dave Clarke and Andres Löh. Generic Haskell, specifically. In Gibbons and Jeuring [27], pages 21–48.
17. M. Clavel, F. Duran, and N. Marti-Oliet. Polytypic programming in Maude. In *Workshop on Rewriting Logic and its Applications 2000*, 2000.
18. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.
19. 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.
20. 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.
21. C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Conference Record of POPL '95: The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 118–129, 1995.
22. Jun Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, January 2001.
23. 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.

24. Jeremy Gibbons. Patterns in datatype-generic programming. In Jörg Striegnitz and Kei Davis, editors, *Multiparadigm Programming*, volume 27, pages 277–289. John von Neumann Institute for Computing (NIC), 2003. First International Workshop on Declarative Programming in the Context of Object-Oriented Languages (DPCOOL).
25. Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Generic Programming, Advanced Lectures*, LNCS. Springer-Verlag, 2006. This volume.
26. Jeremy Gibbons. Metamorphisms: Streaming representation-changers. *Science of Computer Programming*, 65(2):108–139, 2007.
27. Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming*, volume 243 of *IFIP*. Kluwer Academic Publishers, 2003.
28. Jeremy Gibbons and Ross Paterson. Parametric datatype-genericity. Unpublished manuscript, 2006.
29. Paul Hagg. A framework for developing generic XML Tools. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.
30. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, 1995.
31. 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.
32. Ralf Hinze. Functional pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, 2000.
33. Ralf Hinze. *Generic Programs and Proofs*. 2000. Habilitationsschrift, Bonn University.
34. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.
35. 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.
36. Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16:451–482, 2006.
37. Ralf Hinze and Johan Jeuring. Generic Haskell: applications. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 57–97. Springer-Verlag, 2003.
38. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.
39. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51(1–2):117–151, 2004.
40. Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing Approaches to Generic Programming in Haskell. Technical Report UU-CS-2006-022, Utrecht University, 2006.
41. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 4th Haskell Workshop*, 2000.

42. Ralf Hinze and Andres Löh. Generic programming, now! In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming, Advanced Lectures*, LNCS. Springer-Verlag, 2006. This volume.
43. Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In Tarmo Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC'06*, volume 4014 of *LNCS*, pages 180–208. Springer-Verlag, 2006.
44. Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In Philip Wadler and Masimi Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS 2006*, volume 3945 of *LNCS*. Springer-Verlag, 2006.
45. Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC'06*, volume 4014 of *LNCS*, pages 209–234. Springer-Verlag, 2006.
46. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
47. John Hughes. The design of a pretty-printing library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 53–96. Springer-Verlag, 1995.
48. 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.
49. Patrik Jansson and Johan Jeuring. PolyLib – a polytypic function library. In *Workshop on Generic Programming, Marstrand*, June 1998.
50. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
51. Patrik Jansson, Johan Jeuring, and students of the Utrecht University Generic Programming class. Testing properties of generic functions. In Zoltán Horváth, editor, *Proceedings 18th International Symposium on Implementation and Application of Functional Languages, IFL'06*, volume 4449 of *LNCS*. Springer-Verlag, 2007.
52. C. Barry Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2:307–315, 1999.
53. 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.
54. C. Barry Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6):911–937, 2004.
55. 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.
56. C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.
57. Johan Jeuring and Patrik Jansson. Polytypic programming. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming '96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.
58. Johan Jeuring and Rinus Plasmeijer. Generic programming for software evolution. In *Informal proceedings of the ERCIM workshop on Software Evolution*, 2006.

59. Oleg Kiselyov. Smash your boiler-plate without class and Typeable, 2006. Published on the Haskell mailing list.
60. 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.
61. 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.
62. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, pages 204–215. ACM Press, 2005.
63. Ralf Lämmel and Erik Meijer. Revealing the X/O impedance mismatch. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming, Advanced Lectures*, LNCS. Springer-Verlag, 2006. This volume.
64. 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.
65. Ralf Lämmel and Joost Visser. Typed Combinators for Generic Traversal. In *Proceedings of 4th International Symposium on Practical Aspects of Declarative Languages, PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, 2002.
66. 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.
67. M.M. Lehman. Programs, life cycles and the laws of software evolution. *Proc. IEEE*, 68(9):1060–1078, 1980.
68. M.M. Lehman and L.A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, London, 1985.
69. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
70. Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.
71. 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.
72. Andres Löh and Ralf Hinze. Open data types. In Michael Maher, editor, *Proceedings of the 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming, PPDP'06*, 2006.
73. 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.
74. Ian Lynagh. Typing Template Haskell: Soft Types. URL: `http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/Typing_Template_Haskell:_Soft_Types.ps`, August 2004.
75. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

76. 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.
77. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.
78. E Moggi, Bellè, and C.B. Jay. Monads, shapely functors and traversals. In M. Hoffman, Pavlović, and P. Rosolini, editors, *Proceedings of the 8th Conference on Category Theory and Computer Science, CTCS'99*, volume 24 of *Electronic Lecture Notes in Computer Science*, pages 265–286. Elsevier, 1999.
79. David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library.* Addison-Wesley, 2001.
80. Pablo Nogueira. Context-parametric polykinded types. In Ralf Hinze, editor, *Proceedings of the of the ACM SIGPLAN Workshop on Generic Programming 2006*, pages 45–54. ACM Press, 2006.
81. Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In *Implementation of Functional Languages*, volume 3145 of *LNCS*, pages 168–184, 2004.
82. 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'04*, volume 3125 of *LNCS*, pages 314–333. Springer-Verlag, 2004.
83. Bruno C. d. S. Oliveira and Jeremy Gibbons. TypeCase: A design pattern for type-indexed functions. In Andres Löh, editor, *Proceedings Haskell Workshop.* ACM Press, 2005.
84. Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Generics as a library. In Henrik Nilsson, editor, *Proceedings of the 7th Symposium on Trends in Functional Programming, April 19–21, 2006, Nottingham, UK*, 2006.
85. OMG. Corba. `http://www.omg.org/corba/`.
86. 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.
87. A.L. Powell. A literature review on the quantification of software change. Technical Report YCS 305, Computer Science, University of York, 1998.
88. Fermín Reig. Generic proofs for combinator-based generic programs. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5. Intellect, 2006.
89. 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.
90. Tim Sheard. Generic programming in Ωmega. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming, Advanced Lectures*, LNCS. Springer-Verlag, 2006. This volume.
91. Martijn de Vries. Specializing type-indexed values by partial evaluation. Master's thesis, Rijksuniversiteit Groningen, 2004.
92. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89*, pages 347–359. ACM Press, 1989.
93. Philip Wadler. How to replace failure by a list of successes. In *Proceedings conference on Functional programming languages and computer architecture, FPCA '85*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
94. M. Wallace and C. Runciman. Heap compression and binary I/O in Haskell. In *2nd ACM Haskell Workshop*, 1997.

95. David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, Ltd, 2004.
96. 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.
97. Stephanie Weirich. Replib: a library for derivable type classes. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 1–12. ACM Press, 2006.
98. Stephanie Weirich and Liang Huang. A design for type-directed programming in Java. In *Workshop on Object-Oriented Developments, WOOD 2004*, 2004.
99. Noel Winstanley and John Meacham. The DrIFT manual. `http://repetae.net/~john/computer/haskell/DrIFT/`, 1997–2005.
100. Hongwei Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, 1998.