Generic Views on Data Types

Stefan Holdermans

Johan Jeuring

Andres Löh

institute of information and computing sciences, utrecht university

technical report UU-CS-2005-012

www.cs.uu.nl

Generic Views on Data Types

Stefan Holdermans Johan Jeuring Andres Löh

Institute of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht, the Netherlands
{stefan,johanj,andres}@cs.uu.nl

Abstract

A generic function is defined by induction on the structure of types. The structure of a data type can be defined in several ways. For example, in PolyP a pattern functor gives the structure of a data type viewed as a fixed point, and in Generic Haskell a structural representation type gives an isomorphic type view of a data type in terms of sums of products. Depending on this generic view on the structure of data types, some generic functions are easier, more difficult, or even impossible to define. Furthermore, the efficiency of some generic functions can be improved by choosing a different view. This paper introduces generic views on data types and shows why they are useful. Furthermore, it discusses how to add new generic views to Generic Haskell, an extension of the functional programming language Haskell that supports the construction of generic functions. The separation between inductive definitions on type structure and generic views allows us to view many approaches to generic programming in a single framework.

1. Introduction

A generic function is defined by induction on the structure of types. Several approaches to generic programming [15, 10, 19, 18, 14] have been developed in the last decade. These approaches have their commonalities and differences:

- All the approaches provide either a facility for defining a function by induction on the structure of types, or a set of basic, compiler generated, generic functions which are used as combinators in the construction of generic functions.
- All the approaches differ on how they *view* data types. There are various ways in which the inductive structure of data types can be defined, and each approach to generic programming takes a different one.

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

This paper introduces *generic views* on data types. Using generic views it is possible to define generic functions for different views on data types. Generic views provide a framework in which the different approaches to generic programming can be used and compared.

The inductive structure of types. Different approaches to generic programming view the structure of types differently:

- In PolyP [15] a data type is viewed as a fixed point of a pattern functor of kind $* \to * \to *$. Viewing a data type as a fixed point of a pattern functor allows us to define recursive combinators such as the catamorphism and anamorphism [24], and functions that return the direct recursive children of a constructor [17]. A downside of this view on data types is that PolyP can only handle regular data types of kind $* \to *$.
- In Generic Haskell [10, 23, 21], a data type is described in terms of a top-level structural representation type. Generic functions in Generic Haskell are defined on possibly nested data types of any kind. However, because the recursive structure of data types is invisible in Generic Haskell, it is hard to define the catamorphism and children functions in a natural way, for example.
- In the 'Scrap your boilerplate' [19, 20] approach the generic fold is the central steering concept. The generic fold views a value of a data type as either a constructor, or as an application of a (partially applied) constructor to a value. Using the generic fold it is easy to define traversal combinators on data types, which can easily be specialized to update small parts of a value of a large data structure. A disadvantage of the boilerplate approach is that some generic functions, such as the equality and zipping functions, are harder to define. Furthermore, the approach does not naturally generalize to type-indexed data types [11, 21]. We can translate the boilerplate approach to the level of data types by defining a particular generic view.

Other approaches to representing data types can be found in the Constructor Calculus [18], and in the work of Backhouse, De Moor, and Hoogendijk, where a data type is considered to be a relator with a membership relation [3, 14].

Generic views on data types. An approach to generic programming essentially consists of two components: a facility to define recursive functions on structure types, and a view on the inductive structure of data types, which maps data types onto structure types. We call such a view on the structure of types a *generic view* (or just *view*) on data types. Wadler [31] also defines views on data types. The difference between these views and generic views is that Wadler proposes a method for viewing a single data type in different ways, whereas a generic view describes how the structure of a large class of data types is viewed.

Each of the above generic views on data types has its advantages and disadvantages. Some views allow the definition of generic functions that are impossible or hard to define in other approaches, other views allow the definition of more efficient generic functions. This paper

- identifies the concept of generic views as an important building block of an implementation for generic programming;
- shows that different choices of generic views have significant influence on the class of generic functions that can be expressed;
- clearly defines what constitutes a generic view, and discusses how new views can be added in Generic Haskell by modifying the compiler;
- allows us to compare different approaches to generic programming by providing a common framework.

Organization. This paper is organized as follows. Section 2 briefly introduces generic programming in Generic Haskell. Section 3 shows by means of examples why generic views on data types are useful, and how they increase the expressiveness of a generic programming language. Section 4 defines formally what constitutes a generic view. For some of the examples of Section 3, we give the formal definition. In Section 5, we discuss how to add new views to the Generic Haskell compiler. Related work, future work, and conclusions are given in Section 6.

2. Introduction to generic programming in Generic Haskell

This section introduces generic programming in Generic Haskell. The introduction will be brief, for more information see [12, 11, 21]. Generic Haskell has slightly changed in the last couple of years, and we will use the version described in Löh's thesis [21] in this paper, which to a large extent has been implemented in the Coral release [23].

2.1 Type-indexed functions

A type-indexed function takes an explicit type argument, and can have behaviour that depends on the type argument. For example, suppose the unit type Unit, sum type $+$, and product type $\times$ are defined as follows,

$$\begin{array}{l}\mathbf{data}\ \mathrm{Unit} \;=\; \mathit{Unit}\\ \mathbf{data}\ a + b = \mathit{Inl}\ a \mid \mathit{Inr}\ b\\ \mathbf{data}\ a \times b = a \times b.\end{array}$$

We use infix types $+$ and $\times$ and an infix type constructor $\times$ here to ease the presentation.

The type-indexed function col is used to collect values from a data structure. We define function col on the unit type, sums and products, integers, characters, and lists as follows:

$$\begin{array}{lll}\mathit{col}\langle\mathrm{Unit}\rangle & \mathit{Unit} & = [\,]\\ \mathit{col}\langle\alpha+\beta\rangle & (\mathit{Inl}\ a) & = \mathit{col}\langle\alpha\rangle\ a\\ \mathit{col}\langle\alpha+\beta\rangle & (\mathit{Inr}\ b) & = \mathit{col}\langle\beta\rangle\ b\\ \mathit{col}\langle\alpha\times\beta\rangle & (a\times b) & = \mathit{col}\langle\alpha\rangle\ a +\!\!+ \mathit{col}\langle\beta\rangle\ b\\ \mathit{col}\langle\mathrm{Int}\rangle & n & = [\,]\\ \mathit{col}\langle\mathrm{Char}\rangle & c & = [\,]\\ \mathit{col}\langle[\alpha]\rangle & [\,] & = [\,]\\ \mathit{col}\langle[\alpha]\rangle & (a : as) & = \mathit{col}\langle\alpha\rangle\ a +\!\!+ \mathit{col}\langle[\alpha]\rangle\ as.\end{array}$$

The type signature of col is as follows:

$$\mathit{col}\langle a :: * \mid c :: *\rangle :: (\mathit{col}\langle a \mid c\rangle) \Rightarrow a \to [c].$$

The type of collect is parameterized over two type variables. The first type variable, a, appearing to the left of the vertical bar, is a generic type variable, and represents the type of the type argument of collect. Type variable c, appearing to the right of a vertical bar, is called a non-generic (or parametric) type variable. Such non-generic variables appear in type-indexed functions that are parametrically polymorphic with respect to some type variables. The collect function is, as defined, parametrically polymorphic in the element type of its list result. It always returns always the empty list, but we will show later how to adapt it so that it collects values from a data structure. Since it always returns the empty list there is no need, but also no desire, to fix the type of the list elements. The type context $(\mathit{col}\langle a \mid c\rangle) \Rightarrow$ appears in the type because collect is called recursively on sums and products, which means that, for example, if we want an instance for collect on the type $\alpha + \beta$, we need instances of

collect on the types α and β. Thus collect *depends on* itself. The theory of dependencies and type signatures of generic functions is an integral part of dependency-style Generic Haskell [22, 21].

The type signature of col can be instantiated for specific cases by the Generic Haskell compiler, yielding, for example, the types

$$col\langle \mathrm{Unit} \rangle :: \forall c \,.\, \mathrm{Unit} \to [c]$$
$$col\langle [\alpha] \rangle \;\; :: \forall c \; a \,.\, (col\langle \alpha \rangle :: a \to [c]) \Rightarrow [a] \to [c]$$

for the cases of the unit type and lists, respectively. The latter type can be read as "given a function $col\langle\alpha\rangle$ of type $a \to [c]$, the expression $col\langle[\alpha]\rangle$ is of type $[a] \to [c]$".

Depending on the situation, the function $col\langle\alpha\rangle$ can be automatically inferred by the compiler, or it can be user specified using *local redefinitions*. For example, if we only want to collect the positive numbers from a list using the function col, we can write:

$$\begin{array}{l} \mathbf{let}\ col\langle\alpha\rangle\ x \mid x > 0 \qquad = [x] \\ \qquad\qquad\quad \mid otherwise = [\,] \\ \mathbf{in}\ \ col\langle[\alpha]\rangle. \end{array}$$

Generally, we use a *local redefinition* of a generic function to locally modify the behaviour of a generic function. Some generic functions such as col only reveal their full power in the context of local redefinitions.

2.2 Structure types

Until now, it seems as if a type-indexed function is only defined on the types that appear as its type indices. In order to obtain a *generic* function that is defined on arbitrary data types, we give a mapping from data types to view types such as units, sums and products. It suffices to define a function on view types (and primitive or abstract types such as Int and Char) in order to obtain a function that can be applied to values of arbitrary data types. If there is no specific case for a type in the definition of a generic function, generic behaviour is derived automatically by the compiler by exploiting the structural representation.

For example, the definition of the function col on lists is superfluous in the context of generic functions: the instance generically derived for lists coincides with the function obtained from the above definition of col on lists. In order to obtain this instance, the compiler needs to know the structural representation of lists, and how to convert between lists and their structural representation. We will describe these components in the remainder of this section.

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

$$\begin{array}{l} \mathbf{data}\ \mathrm{List}\ a \quad\ = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathrm{List}\ a) \\ \mathbf{data}\ \mathrm{Tree}\ a\ b = \mathit{Tip}\ a \\ \qquad\qquad\qquad\ \mid \mathit{Node}\ (\mathrm{Tree}\ a\ b)\ b\ (\mathrm{Tree}\ a\ b) \end{array}$$

we obtain the following structural representations:

$$\begin{array}{l} \mathbf{type}\ \mathrm{List}^\circ\ a \quad\ = \mathrm{Unit} + a \times \mathrm{List}\ a \\ \mathbf{type}\ \mathrm{Tree}^\circ\ a\ b = a + \mathrm{Tree}\ a\ b \times b \times \mathrm{Tree}\ a\ b, \end{array}$$

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

If two types are isomorphic, the corresponding isomorphisms can be stored as a pair of functions converting back and forth:

$$\mathbf{data}\ \mathrm{EP}\ a\ b = EP\{from :: (a \to b), to :: (b \to a)\}.$$

A type T and its structural representation type T° are isomorphic, witnessed by a value $conv_T :: \mathrm{EP}\ T\ T^\circ$. For example, for the list data type we have that $conv_{\mathrm{List}} = EP\ to_{\mathrm{List}}\ from_{\mathrm{List}}$, where to_{List} and $from_{\mathrm{List}}$ are defined by

$$\begin{array}{lll}
to_{\mathrm{List}} & :: \forall a\,.\,\mathrm{List}\ a \to \mathrm{List}^\circ\ a & \\
to_{\mathrm{List}} & Nil & = Inl\ Unit \\
to_{\mathrm{List}} & (Cons\ a\ as) & = Inr\ (a \times as) \\
from_{\mathrm{List}} & :: \forall a\,.\,\mathrm{List}^\circ\ a \to \mathrm{List}\ a & \\
from_{\mathrm{List}} & (Inl\ Unit) & = Nil \\
from_{\mathrm{List}} & (Inr\ (a \times as)) & = Cons\ a\ as.
\end{array}$$

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

With the structural representation types and the embedding-projection pairs in place, a call to a generic function on a data type T can always be reduced to a call on type T°. Hence, if the generic function is defined for the data types occurring in the structural representations, such as Unit, $+$, and $\times$, we do not need cases for specific datatypes such as List or Tree anymore.

For primitive types such as Int, Float, IO or $\to$, no structure is available. For a generic function to work on these types, a specific case is necessary. Other types are deliberately declared abstract to hide the structure of the type. Here, the situation is the same, and a specific case is necessary to extend a generic function to an abstract type. The types Unit, $+$, and $\times$ are abstract types as far as Generic Haskell is concerned, instead of being their own representation types. This choice implies that a missing case for one of these types is reported as a compile-time error rather than causing a run-time loop.

3. Views

We have explained how Generic Haskell defines a structural representation type plus an embedding-projection pair for any Haskell data type. A type-indexed function is generic because the embedding-projection pair is applied to the type arguments by the compiler as needed. Other approaches to generic programming use different, but still fixed representations of data types. In this section, we argue that different views can improve the expressiveness of a generic programming system, because not every view is equally suitable for every generic function. In the next section we will give a formal definition of generic views.

3.1 Fixed points

Consider the data type Term, representing lambda terms, and the function $subterms$ that, given a term, produces the immediate subterms.

$$\begin{array}{ll}
\mathbf{data}\ \mathrm{Term} & = Var\ \mathrm{Variable} \\
 & \mid Abs\ \mathrm{Variable\ Term} \\
 & \mid App\ \mathrm{Term\ Term} \\
\mathbf{type}\ \mathrm{Variable} & = \mathrm{String} \\
subterms & :: \mathrm{Term} \to [\mathrm{Term}] \\
subterms\ (Var\ x) & = [\,]
\end{array}$$

$$
\begin{aligned}
&\mathit{subterms}\ (\mathit{Abs}\ x\ t) &&= [t] \\
&\mathit{subterms}\ (\mathit{App}\ t\ u) &&= [t, u]
\end{aligned}
$$

This function is an instance of a more general pattern. The function $\mathit{subtrees}$, for example, produces the immediate subtrees of an external binary search tree.

$$
\begin{aligned}
&\mathit{subtrees} :: \forall a\ b\ .\ \mathrm{Tree}\ a\ b \to [\mathrm{Tree}\ a\ b] \\
&\mathit{subtrees}\ (\mathit{Tip}\ a) &&= [\,] \\
&\mathit{subtrees}\ (\mathit{Node}\ l\ b\ r) &&= [l, r]
\end{aligned}
$$

Both $\mathit{subterms}$ and $\mathit{subtrees}$ retrieve the immediate children of a recursive data type's value. Since the general pattern is clear, we would like to be able to express it as a generic function. However, Generic Haskell does not allow us to define such a function directly, due to the fact that the structure over which generic functions are inductively defined does not expose the recursive calls in a data type's definition.

Generic Haskell's precursor, PolyP, does give access to these recursive calls, enabling the definition of a generic function that collects the immediate recursive children of a value [17]. Generic functions in PolyP, however, are limited in the sense that they can only be applied to data types of kind $* \to *$.

Interestingly, it is possible to write a program in Generic Haskell that produces the immediate children of a value, but it requires some extra effort from the user of the program. If regular recursive data types are expressed using an explicit type-level fixed point operator:

$$
\begin{aligned}
&\mathbf{data}\ \mathrm{Fix}\ f &&= \mathit{In}\ (f\ (\mathrm{Fix}\ f)) \\
&\mathbf{data}\ \mathrm{TermBase}\ r &&= \mathit{VarBase}\ \mathrm{Variable} \\
&&&\mid \mathit{AbsBase}\ \mathrm{Variable}\ r \\
&&&\mid \mathit{AppBase}\ r\ r \\
&\mathbf{type}\ \mathrm{Term}' &&= \mathrm{Fix}\ \mathrm{TermBase} \\
&\mathbf{data}\ \mathrm{TreeBase}\ a\ b\ r &&= \mathit{TipBase}\ a \\
&&&\mid \mathit{NodeBase}\ r\ b\ r \\
&\mathbf{type}\ \mathrm{Tree}'\ a\ b &&= \mathrm{Fix}\ (\mathrm{TreeBase}\ a\ b),
\end{aligned}
$$

then the generic function $\mathit{children}$ can be defined with a single case for Fix.

$$
\begin{aligned}
&\mathit{children}\langle a :: * \rangle :: (\mathit{col}\langle a \mid a \rangle) \Rightarrow a \to [a] \\
&\mathit{children}\langle \mathrm{Fix}\ \varphi \rangle\ (\mathit{In}\ r) = \mathbf{let}\ &&\mathit{col}\langle \alpha \rangle\ x = [x] \\
&&\mathbf{in}\ \ &\mathit{col}\langle \varphi\ \alpha \rangle\ r
\end{aligned}
$$

The $\mathit{children}$ function depends on the collect function col defined in Section 2. The local redefinition fixes the type of the produced list and adapts the collect function to construct singleton lists from the recursive components in a fixed point's value. The function col ensures that these singletons are concatenated to produce the result list.

Although this approach works fine, there is an obvious downside. The programmer needs to redefine her recursive data types in terms of Fix. Whenever she wants to use $\mathit{children}$ to compute the recursive components of a value of any of the original recursive types, say Term or Tree, a user-defined bidirectional mapping from the original types to the fixed points, Term' and Tree', has to be applied.

With a *fixed-point view*, the compiler becomes capable of deriving the fixed point for any regular recursive data type and will generate and apply the required mappings automatically. The structure of a data type is then no longer perceived as a sum of products, but as the fixed point of a sum of

products. The only thing we have to change in the definition of $children$ to make use of the new view is to add the name of the view to the type signature:

$$children\langle a :: * \ \mathbf{viewed}\ \mathsf{Fix}\rangle :: (col\langle a \mid a\rangle) \Rightarrow a \to [a].$$

The definition of $children$ is unchanged. One can now, for example, apply $children\langle[\mathrm{Int}]\rangle\ [1,2,3]$ directly, yielding $[[2,3]]$. The user of the function does not have to worry about defining types in terms of Fix any longer: the translation happens behind the scenes.

The formal definition of the view Fix will be given in detail in Section 4.5. In the following, we will discuss examples of other views.

3.2 Balanced sums of products

Traditionally, Generic Haskell views the structure of data types using nested right-deep binary sums and products. The choice for such a view is rather arbitrary. A nested left-deep view or a balanced view may be just as suitable. However, the chosen view has some impact on the behaviour of certain generic programs.

The generic function enc, for instance, encodes values of data types as lists of bits [16, 9].

$$\begin{aligned}
&\mathbf{data}\ \mathrm{Bit} = \mathit{Off} \mid \mathit{On}\\
&enc\langle a :: *\rangle :: (enc\langle a\rangle) \Rightarrow a \to [\mathrm{Bit}]\\
&enc\langle\mathrm{Unit}\rangle\ \mathit{Unit} = [\,]\\
&enc\langle\alpha+\beta\rangle\ (\mathit{Inl}\ a) = \mathit{Off} : enc\langle\alpha\rangle\ a\\
&enc\langle\alpha+\beta\rangle\ (\mathit{Inr}\ b) = \mathit{On} : enc\langle\beta\rangle\ b\\
&enc\langle\alpha\times\beta\rangle\ (a\times b) = enc\langle\alpha\rangle\ a +\!\!+ enc\langle\beta\rangle\ b\\
&enc\langle\mathrm{Int}\rangle\ n = \mathit{encInt}\ n\\
&enc\langle\mathrm{Char}\rangle\ c = \mathit{encChar}\ c
\end{aligned}$$

Here, $encInt$ and $encChar$ denote primitive encoders for integers and characters, respectively. The interesting cases are the ones for sums where a bit is emitted for every choice that is made between a pair of constructors. In the case for products the encodings of the constituent parts are concatenated.

When we apply a nested right-deep view to the type $Compass$ of directions

$$\mathbf{data}\ \mathit{Compass} = \mathit{North} \mid \mathit{East} \mid \mathit{South} \mid \mathit{West},$$

yielding the structure

$$\mathbf{type}\ \mathit{Compass}^{\circ} = \mathrm{Unit} + (\mathrm{Unit} + (\mathrm{Unit} + \mathrm{Unit})),$$

encoding a value with enc takes one bit at best ($North$) and three bits at worst ($West$). In contrast, a balanced view Bal on the structure, i.e.,

$$\mathbf{type}\ \mathit{Compass}^{\circ}_{B} = (\mathrm{Unit} + \mathrm{Unit}) + (\mathrm{Unit} + \mathrm{Unit}),$$

requires only two bits for any value of $Compass$.

In general, encoding requires $O(n)$ bits on average when a nested structure representation is applied, and $O(\log n)$ bits when a balanced representation is used. All we have to do (next to implementing a balanced view Bal) is to change the type signature of enc accordingly:

$$enc\langle a :: * \ \mathbf{viewed}\ \mathsf{Bal}\rangle :: (enc\langle a\rangle) \Rightarrow a \to [\mathrm{Bit}].$$

3.3 List-like sums and products

Suppose we have a generic function $show$ which is of type

$show\langle a :: * \rangle :: (show\langle a \rangle) \Rightarrow a \to \mathrm{String}$

and produces a human-readable string representation of a value. We want to write a function that shows only a part of a value. The part that is shown is determined by a path, of type

$\mathbf{type}\ \mathrm{Path} = [\mathrm{Int}].$

Non-empty lists of type Path select a part of the data structure to print. For instance, $[1]$ selects the second field of the top-level constructor, and $[1, 0]$ selects the first field of the top-level constructor thereof. The function is called sP and of type

$sP\langle a :: * \rangle :: (show\langle a \rangle, sP\langle a \rangle) \Rightarrow \mathrm{Path} \to a \to \mathrm{String}.$

Let us look at the definition of sP on products:

$sP\langle \alpha \times \beta \rangle\ (0 : p)\ (a \times b) =$
$\quad \mathbf{if}\ null\ p\ \mathbf{then}\ show\langle \alpha \rangle\ a\ \mathbf{else}\ sP\langle \alpha \rangle\ p\ a.$

If the first path element is 0, we know that the leftmost field is selected. The encoding in binary products is such that the left component is always a field of the constructor, and not an encoding of multiple fields. We can therefore test if the remainder of the path is empty: if this is the case, we show the complete field using $show$; otherwise, we show the part of the field that is selected by the tail of the current path.

$sP\langle \alpha \times \beta \rangle\ (n : p)\ (a \times b) = sP\langle \beta \rangle\ (n - 1 : p)\ b$

If the first path element is not 0, we can decrease it by one and show a part of the right component, containing the remaining fields.

There are several problems with this approach. Consider the following data type and its structural representation:

$\mathbf{data}\ \mathrm{Con012}\ a\ b\ \ = \mathit{Con0}\ \mid\ \mathit{Con1}\ a\ \mid\ \mathit{Con2}\ a\ b$
$\mathbf{type}\ \mathrm{Con012}^{\circ}\ a\ b = \mathrm{Unit}\ \ + a\qquad\quad + a \times b.$

A product structure is only created if there are at least two fields. If there is only one, such as for $\mathit{Con1}$, the single field (here a) is the representation. Obviously, we then cannot use the product case of the generic function to make sure that 0 is the topmost element of the path.

We could add a check to the sum case of the function, detecting the size of the underlying product structure by calling another generic function, or by modifying the type of sP to pass additional information around. However, consider a datatype Rename and its structural representation:

$\mathbf{data}\ \mathrm{Rename}\ \ = R\ \mathit{Original}$
$\mathbf{type}\ \mathrm{Rename}^{\circ} = \mathit{Original}.$

The structural representation does not even contain a sum structure. Although it is possible to write sP in the standard view, it is extremely tiresome to do so. The same functionality has to be distributed over a multitude of different cases, simply because the structural encoding is so irregular, and we cannot rely on sum and product structure to be present in any case.

A list-like view List on data types can help. For this purpose we introduce a data type without constructors and without values (except bottom).

$\mathbf{data}\ \mathrm{Zero}$

The type Zero plays the role of a neutral element for sums in the same way as Unit does for products. The above definition is not Haskell 98, but is supported by GHC and can be simulated in Haskell 98.

In our list-like view, the left component of a sum always encodes a single constructor, and the right component of a sum is either another sum or Zero. For products, the left component is always a single field, and the right component is either another product or Unit. In particular, there is always a sum and a product structure. The data type Con012 is encoded as follows:

$$\begin{aligned}&\textbf{type}\ \mathrm{Con012}^{\circ}_{L}\ a\ b =\\ &\quad \mathrm{Unit} + a \times \mathrm{Unit} + a \times b \times \mathrm{Unit} + \mathrm{Zero}.\end{aligned}$$

Now, we can define sP easily:

$$\begin{aligned}&sP\langle a :: * \ \textbf{viewed}\ \mathsf{List}\rangle ::\\ &\quad (show\langle a\rangle, sP\langle a\rangle) \Rightarrow \mathrm{Path} \to a \to \mathrm{String}\\ &sP\langle \mathrm{Unit}\rangle\ \_\ \ \ Unit\ \ \ = error\ \texttt{"illegal path"}\\ &sP\langle \alpha \times \beta\rangle\ (0 : p)\ (a \times b) = sP\langle \alpha\rangle\ \ \ p\ \ a\\ &sP\langle \alpha \times \beta\rangle\ (n : p)\ (a \times b) = sP\langle \beta\rangle\ (n-1 : p)\ b\\ &sP\langle \mathrm{Zero}\rangle\ \_\ \ \ \_\ \ \ = error\ \texttt{"cannot happen"}\\ &sP\langle \alpha + \beta\rangle\ [\,]\ \ \ x\ \ \ = show\langle \alpha + \beta\rangle\ x\\ &sP\langle \alpha + \beta\rangle\ p\ \ \ (Inl\ a) = sP\langle \alpha\rangle\ p\ a\\ &sP\langle \alpha + \beta\rangle\ p\ \ \ (Inr\ b) = sP\langle \beta\rangle\ p\ b.\end{aligned}$$

We have moved the check for the empty path to the sum case. We can do this because we know that every data type has a sum structure in the list-like view.

3.4 Boilerplate approach

In the 'Scrap Your Boilerplate' approach, Lämmel and Peyton Jones present a design pattern for writing programs that traverse data structures [19, 20]. These traversals are defined using a relatively small library that comprises two types of generic combinators: recursive traversals and type extensions. Generic functions are defined in terms of these library functions, and not by induction on the structure of types. The library functions, however, do use a particular view on data types. This section discusses this view, dubbed Boilerplate, and shows how to implement a traversal function on this view. The emulation of the boilerplate approach in Generic Haskell extended with generic views is useful for comparing different approaches to generic programming, but it turns out to be less convenient to use than the original boilerplate library.

In the boilerplate approach all traversals are instances of a general scheme imposed by a left-associative generic fold over constructor applications. So a type is viewed as a sum of products, where a product is either a nullary constructor, or the application of a constructor(-application) to a type. To emulate the behaviour of the generic fold, the product constructor $\times$ in the Boilerplate view is left associative as well. The right component of a product is always a single field, and the left component is either another product or Unit, similar to the List view from Section 3.3.

Besides generic traversals such as the generic fold, the Boilerplate view makes use of type extensions. A type extension extends the type of a function such that it works on many types instead of a single type. Since Generic Haskell operates on the structure of types, isomorphic types are treated as if they were identical. To emulate type extensions, we have to be able to distinguish types by name. Therefore a data type is represented by a tagged sum of list-like products in the Boilerplate view. Each sum is tagged with the name of the corresponding type:

$$\textbf{data}\ \mathrm{Tagged}\ a = \mathit{Tagged}\ \mathrm{String}\ a.$$

For example, the Boilerplate view representations of the types of lists and trees are given by:

$$\begin{aligned}
&\mathbf{type}\ \mathrm{List}^{\circ}_{BP}\ a = \\
&\quad \mathrm{Tagged}\ (\mathrm{Unit} + (\mathrm{Unit} \times a) \times \mathrm{List}\ a) \\
&\mathbf{type}\ \mathrm{Tree}^{\circ}_{BP}\ a\ b = \\
&\quad \mathrm{Tagged}\ (\mathrm{Unit} \times a \\
&\quad + ((\mathrm{Unit} \times \mathrm{Tree}\ a\ b) \times b) \times \mathrm{Tree}\ a\ b).
\end{aligned}$$

The definitions of the traversal combinators rely on the list-like character of the Boilerplate view. For example, the *everywhere* combinator applies a transformation to all nodes in a tree, traversing it in a top-down fashion. It can be defined in terms of a simple non-recursive one-layer traversal combinator $gmapT$ as follows:

$$\begin{aligned}
&everywhere\langle a :: * \ \mathbf{viewed}\ \mathsf{Boilerplate}\rangle :: \\
&\quad (typeof\langle a\rangle) \Rightarrow \\
&\quad (\forall x\,.\,((x \to x) \to \mathrm{Type}) \to x \to x) \to a \to a \\
&everywhere\ \mathbf{extends}\ gmapT \\
&everywhere\langle \alpha \times \beta\rangle\ f\ (a \times b) = \\
&\qquad everywhere\langle \alpha\rangle\ f\ a \\
&\quad \times f\ typeof\langle \beta \to \beta\rangle\ (everywhere\langle \beta\rangle\ f\ b).
\end{aligned}$$

Function *everywhere* is defined by means of a ***default case*** [8, 21]: it behaves as $gmapT$ on all types except for the product type, for which specific behaviour is defined. The transformation that is an argument to everywhere is supposed to perform different actions depending on the type of node it is currently processing. Unfortunately, Generic Haskell currently lacks support for higher-order generic functions; we thus use *typeof* as a workaround. The call $typeof\langle T\rangle$ produces a universal representation of type Type from a value of type T. Its implementation is beyond the scope of this paper.

We yet have to define $gmapT$, which is a non-recursive one-layer traversal combinator, applying the transformation argument to the immediate children of a node.

$$\begin{aligned}
&gmapT\langle a :: * \ \mathbf{viewed}\ \mathsf{Boilerplate}\rangle :: \\
&\quad (typeof\langle a\rangle) \Rightarrow \\
&\quad (\forall x\,.\,((x \to x) \to \mathrm{Type}) \to x \to x) \to a \to a \\
&gmapT\langle \mathrm{Tagged}\ \alpha\rangle\ f\ (Tagged\ t\ a) = \\
&\quad Tagged\ t\ (gmapT\langle \alpha\rangle\ f\ a) \\
&gmapT\langle \mathrm{Unit}\rangle\ \ f\ Unit\ \ = Unit \\
&gmapT\langle \alpha + \beta\rangle\ f\ (Inl\ a) = Inl\ (gmapT\langle \alpha\rangle\ f\ a) \\
&gmapT\langle \alpha + \beta\rangle\ f\ (Inr\ b) = Inr\ (gmapT\langle \beta\rangle\ f\ b) \\
&gmapT\langle \alpha \times \beta\rangle\ f\ (a \times b) = \\
&\quad gmapT\langle \alpha\rangle\ f\ a \times f\ typeof\langle \beta \to \beta\rangle\ b \\
&gmapT\langle \mathrm{Int}\rangle\ \ f\ n\ \ \ = n \\
&gmapT\langle \mathrm{Char}\rangle\ f\ c\ \ \ = c.
\end{aligned}$$

Note that the case for products only recurses on the left component of a product. Since the Boilerplate view guarantees that all fields of a constructor are encoded as right components of products, it is easy to verify that $gmapT$ does indeed define a non-recursive traversal. This simple non-recursive scheme allows us to derive several rich recursive traversal strategies from a single base combinator. These strategies are written using default cases.

The type-extension operators used in the Boilerplate approach can be defined by making use of the type tags that the Boilerplate view embeds in structural representations. The definitions of these operators are omitted.

In summary, it is clear that generic traversals should either be programmed using the original Boilerplate approach or using the standard view in Generic Haskell plus default cases. We believe, however, that an encoding of the Boilerplate approach within the view formalism can help to better compare it with other approaches, and improve the overall understanding of different generic programming techniques.

3.5 Other views

Many other views, besides the views we have listed so far, are useful.

In the Generic Haskell compiler [23], for example, we do not really use the standard view as presented here, but additionally use representation data types Con and Lab to encode information about constructors and record field labels in the data type. The presence of these data types makes it possible to write functions such as $show$ and $read$ that produce or consume a representation of a value and therefore rely on the names of constructors and labels.

The example views we have given share the property that they are applicable to a relative large class of data types. For instance, the standard, balanced, and list-like views are applicable to all Haskell data types, and the fixed-point view works for regular data types.

If one further restricts the class of data types a view should apply to, a multitude of new possibilities arises. If values from specific domains such as SQL or XML are modelled in the Haskell type system, the resulting data types have a special structure that can be exploited by a view. For instance, XML Schema [30] has an element 'all' that allows certain elements to appear in any order, whereas in the element 'sequence' the order is fixed. Furthermore, XML Schema allows for mixed content where elements occur interleaved with strings.

Finite types or renamed types (data types in the form of a Haskell $\mathbf{newtype}$ declaration) are other subclasses of Haskell data types that allow special treatment.

A corner case is given by an encoding of the "lightweight" approach to generic programming [7] as a generic view: each datatype T is paired with a run-time representation $Rep\ T$ of the type. Type-level pattern matching becomes trivial – there is only one case for every generic function. The real distinction is performed at run-time, using value-level pattern matching on the type representation.

Another extreme case is a view that only works for a single datatype, on which it performs an isomorphic transformation. Generic views thus subsume Wadler's views.

4. Generic views, formally

The previous section shows why generic views are useful. This section formally defines generic views, and shows the formal definition of the standard view and the fixed-point view.

4.1 Notation

Throughout this section, we often use the following notation to denote repetition:

$$\{X_i\}^{i\in 1..n} \equiv X_1 \ldots X_n$$
$$\{X_i\}_{;}^{i\in 1..n} \equiv X_1; \ldots; X_n$$

If not explicitly mentioned otherwise, such repetitions can be empty, i.e., n can be 0. We sometimes omit the range of the variable if it is irrelevant or clear from the context.

Programs

P	$::=$	$\{D_i;\}^i\ e$	type declarations and main expression

Value declarations

d	$::=$	$x = e$	function declaration

Expressions

e	$::=$	x	variable
	$\mid$	C	data constructor
	$\mid$	$\lambda x\ .\ e$	abstraction
	$\mid$	$(e_1\ e_2)$	application
	$\mid$	$\mathbf{case}\ e_0\ \mathbf{of}\ \{p_i \to e_i\}_{;}^{i}$	case
	$\mid$	$(\mathbf{fix}\ e)$	fixed point
	$\mid$	$\mathbf{let}\ \{d_i\}_{;}^{i}\ \mathbf{in}\ e$	let

Patterns

p	$::=$	x	variable pattern
	$\mid$	$(C\ \{p_i\}^i)$	constructor pattern

Figure 1. Syntax of the expression language

Type declarations

D	$::=$	$\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}_{\mid}^{j}$	algebraic data type

Parameterized types

u	$::=$	$\{\Lambda a_i :: \kappa_i\ .\ \}^i\ t$	type-level abstraction

Types

t	$::=$	a	type variable
	$\mid$	T	type constructor
	$\mid$	$(t_1\ t_2)$	type application
	$\mid$	$\forall a :: \kappa\ .\ t$	universal quantification

Figure 2. Syntax of the type language

4.2 Syntax

Programs. Figure 1 shows the syntax of programs in the core language. This language is a rather standard functional language. A program consists of zero or more type declarations and a single expression: the main function.

Types and kinds. The syntax of the type language is shown in Figure 2. New types are introduced by means of **data** declarations. Such a declaration associates a, possibly parameterized, type constructor with zero or more data constructors, each of which has zero or more fields. The parameterized types are explained below. The syntax of the kind language is shown in Figure 3.

Kinds

κ	$::=$	$*$	kind of proper types
	$\mid$	$\kappa_1 \to \kappa_2$	function kind

Figure 3. Syntax of the kind language

Kind environments

K	$::=$	ε	empty kind environment
	$\mid$	K, $a :: \kappa$	type-variable binding
	$\mid$	K, $T :: \kappa$	type-constructor binding

Type environments

Γ	$::=$	ε	empty type environment
	$\mid$	$\Gamma, x :: t$	variable binding
	$\mid$	$\Gamma, C :: t$	data-constructor binding

Figure 4. Syntax of environments

Generic programming extensions. To facilitate generic programming, the core language has to be extended with parameterized type patterns and type-indexed functions with dependencies [21]. Apart from a small detail, this extension is not relevant for generic views. The detail that has to be added is the facility to specify a view in the signature of a generic function.

As shown in Section 2, structure types in Haskell are declared by means of the **type** construct. This construct is used to define type synonyms, rather than new algebraic data types. In particular, a **type** declaration does not introduce any new data constructors.

Type synonyms are not supported in the core language. Therefore, to be able to describe structure types, there are *parameterized types* in the language, which are essentially a nesting of type-level lambda abstractions around a type. Parameterized types cannot appear in a core-language program; they are only used in view definitions.

Rules. The well-formedness rules for programs, types and kinds, the kinding rules for types and the typing rules for expressions are standard. The operational semantics of the core language is omitted. More information about the core language can be found elsewhere [21, 13].

4.3 Definitions

Using the notion of parameterized types, we can formalize the observation that a view is constituted by a collection of view types and algorithms for the generation of structure types and conversion functions. In the following definitions we will use kind environments and type environments; their syntax is defined in Figure 4.

DEFINITION 4.1 (Generic View). *A* generic view $\mathcal{V}$ *consists of a collection of bindings for view types,*

$$\mathsf{viewtypes}_{\mathcal{V}} \equiv \mathrm{K}; \Gamma,$$

a partial mapping from types to structure types,

$$\mathcal{V}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i \in 1..n},$$

and, for each type in the domain of this mapping, conversions between values and structure values,

$\mathcal{V}[\![D_0]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}}$.

Notice that we allow the mapping from types to structure types to generate zero or more additional declarations for supporting data types. The types introduced by these declarations can be used for the generation of structure types. This is used in the fixed-point view, for example.

For a view to be useful for generic programming with structural polymorphism, we require it to have three essential properties. First, the mapping from types to structure types should preserve kinds.

DEFINITION 4.2 (Kind Preservation). *A generic view $\mathcal{V}$ with*

$$\mathsf{viewtypes}_{\mathcal{V}} \equiv \mathrm{K}_{\mathcal{V}}; \Gamma_{\mathcal{V}}$$

is kind preserving *if for each well-formed declaration D_0 of a type constructor T with kind κ under a suitable kind environment* K *for which a structure type u can be derived,*

$$\mathcal{V}[\![D_0]\!]^{\text{str}} \equiv u; \{D_i\}_{,}^{i\in 1..n},$$

it follows that under kind environment K′ *containing* K, $\mathrm{K}_{\mathcal{V}}$*, and all the kinds of the D_i declarations, the supporting type declarations D_i are well-formed and the structure type u has the same kind κ as the original type T,*

$$\mathrm{K}' \vdash u :: \kappa.$$

Furthermore, the conversion functions derived from a type declaration should be well-typed and indeed convert between values of the original type and values of the structure type, which is captured by the following definition.

DEFINITION 4.3 (Well-typed Conversion). *A view $\mathcal{V}$ with*

$$\mathsf{viewtypes}_{\mathcal{V}} \equiv \mathrm{K}_{\mathcal{V}}; \Gamma_{\mathcal{V}}$$

generates well-typed conversions *if, for each well-formed declaration D_0 of a type constructor T of kind $\{\kappa_i \rightarrow\}^{i\in 1..\ell}*$, for which a structure type t can be derived,*

$$\mathcal{V}[\![D_0]\!]^{\text{str}} \equiv \{\Lambda a_i :: \kappa_i \,.\,\}^{i\in 1..\ell}\ t; \{D_i\}^{i\in 1..n},$$

it follows that the corresponding conversion functions e_{from} and e_{to},

$$\mathcal{V}[\![D_0]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}},$$

take values of the original data type T to values of the structure type t and vice versa,

$$\mathrm{K}'; \Gamma' \vdash e_{\text{from}} :: \{\forall a_i :: \kappa_i \,.\,\}^{i\in 1..\ell}\ T\ \{a_i\}^{i\in 1..\ell} \rightarrow t$$
$$\mathrm{K}'; \Gamma' \vdash e_{\text{to}} \quad :: \{\forall a_i :: \kappa_i \,.\,\}^{i\in 1..\ell}\ t \rightarrow T\ \{a_i\}^{i\in 1..\ell}$$

under environments K′ *containing both the view bindings $\mathrm{K}_{\mathcal{V}}$ and the kind of T and* Γ' *containing the view bindings $\Gamma_{\mathcal{V}}$ and the types of the constructors from D_0.*

Finally, the conversion functions from structure values to values should form the inverses of the corresponding functions in the opposite direction:

DEFINITION 4.4 (Well-behaved Conversion). *A generic view $\mathcal{V}$ produces* well-behaved conversions *if, for each well-formed declaration D of a type constructor T, conversion functions e_{from} and e_{to} are generated,*

$\mathcal{V}[\![D]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}},$

such that e_{to} *is the left inverse of* e_{from} *with respect to function composition—i.e.,*

$e_{\text{to}}\ (e_{\text{from}}\ v)$ *evaluates to* v

for each value v *of type* T.

(Note that, for a well-behaved conversion pair, the function that takes values to structure values is injective; thus, a structure type should have at least as many elements as the corresponding original type.)

Only views that possess all three of the discussed properties are considered valid:

DEFINITION 4.5 (Validity). *A generic view is* valid *if it is kind preserving and generates well-typed, well-behaved conversions.*

4.4 The standard view

We describe the three components of a generic view for the standard Generic Haskell view $\mathcal{S}$ of data types

View types. The view types of the standard view are given by the following declarations:

$$\begin{array}{l}\mathbf{data}\ \mathrm{Zero}\ = \\ \mathbf{data}\ \mathrm{Unit}\ = \mathit{Unit} \\ \mathbf{data}\ \mathrm{Sum}\ = \Lambda a :: * \,.\, \Lambda b :: * \,.\, \mathit{Inl}\ a \mid \mathit{Inr}\ b \\ \mathbf{data}\ \mathrm{Prod} = \Lambda a :: * \,.\, \Lambda b :: * \,.\, a \times b.\end{array}$$

These types represent nullary sums, nullary products, binary sums, and binary products, respectively. It is easy to convert these definitions into bindings in the environments Γ and K.

Generating structure types. The algorithm that generates structural representations for data types is expressed by judgements of the forms

$$\begin{array}{ll}\mathcal{S}[\![D_0]\!]^{\text{str}} & \equiv u; \{D_i\}_{,}^{i \in 1..n} \\ \mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\text{str}} & \equiv t.\end{array}$$

The former express how type declarations are mapped to parameterized types and lists of supporting declarations; the latter express how a type is derived from a list of constructors. The rules are shown in Figures 5 and 6.

Type declarations are handled by the rule in Figure 5. The type parameters of a declared type constructor are directly copied to the resulting structure type. Notice that the standard view does not need auxiliary declarations.

For constructors, we distinguish five cases. The first rule, (str-std-1), represents empty constructor lists with Zero. The next three cases handle singleton lists of constructors. Fieldless constructors are, by rule (str-std-2), represented by nullary products. Rule (str-std-3) represents a unary constructors by the type of its field. If a constructor has two or more fields, rule (str-std-4) generates a product type and recurses. Finally, lists that contain two or more constructors are represented by a recursively built sum (str-std-5).

Generating conversions. The rules for generating conversion functions are shown in Figures 7 and 8 and are of the forms

$\mathcal{S}[\![D_0]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}}$

$$\boxed{\mathcal{S}[\![D_0]\!]^{\text{str}} \equiv u; \{D_i\}_{,}^{i}}$$

$$\frac{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^k\}_{|}^j]\!]^{\text{str}} \equiv t}{\mathcal{S}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}_{|}^j]\!]^{\text{str}} \equiv \{\Lambda a_i :: \kappa_i\ .\ \}^i\ t; \varepsilon}$$

Figure 5. Representation of data types in the standard view

$$\boxed{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^k\}_{|}^j]\!]^{\text{str}} \equiv t}$$

$$\frac{}{\mathcal{S}[\![\varepsilon]\!]^{\text{str}} \equiv \text{Zero}}\ \text{(str-std-1)} \qquad \frac{}{\mathcal{S}[\![C]\!]^{\text{str}} \equiv \text{Unit}}\ \text{(str-std-2)}$$

$$\frac{}{\mathcal{S}[\![C\ t]\!]^{\text{str}} \equiv t}\ \text{(str-std-3)}$$

$$\frac{n \in 2\,..\qquad \mathcal{S}[\![C\ \{t_k\}^{k \in 2..n}]\!]^{\text{str}} \equiv t_2'}{\mathcal{S}[\![C\ \{t_k\}^{k \in 1..n}]\!]^{\text{str}} \equiv \text{Prod}\ t_1\ t_2'}\ \text{(str-std-4)}$$

$$\frac{\begin{array}{c} m \in 2\,.. \\ \mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 2..m}]\!]^{\text{str}} \equiv t_2 \\ \mathcal{S}[\![C_1\ \{t_{1,k}\}^{k \in 1..n_1}]\!]^{\text{str}} \equiv t_1 \end{array}}{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\text{str}} \equiv \text{Sum}\ t_1\ t_2}\ \text{(str-std-5)}$$

Figure 6. Representation of constructors in the standard view

$$\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^k\}_{|}^j]\!]^{\text{conv}} \equiv \{p_{\text{from},j}\}_{|}^j; \{p_{\text{to},j}\}_{|}^j,$$

i.e., type declarations give rise to pairs of conversion functions, while lists of data constructors give rise to pairs of patterns.

The rule in Figure 7 constructs a 'from' function that matches values of the original type against a list of patterns. If a value is successfully matched against a certain pattern, a structure value is produced by using a complementary pattern; hence, here, we make use of the fact that the pattern language is just a subset of the expression language. A 'to' function is created by inverting the patterns.

The pairs of pattern lists are generated using the rules for constructor lists. These rules are analogous to the rules for generating structure types from constructor lists.

If there are no constructors, there are no patterns either (conv-std-1). Rule (conv-std-2) associates a single constructor with the value $Unit$. Rule (conv-std-3) associates unary constructors with variables that correspond to their field values. If a constructor has two or more fields, rule (conv-std-4) associates

$$\boxed{\mathcal{S}[\![D]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}}}$$

$$\frac{\begin{array}{c}\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^k\}_|^j]\!]^{\text{conv}} \equiv \{p_{\text{from},j}\}_|^j; \{p_{\text{to},j}\}_|^j \\ e_{\text{from}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{p_{\text{from},j} \to p_{\text{to},j}\}_;^j \\ e_{\text{to}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{p_{\text{to},j} \to p_{\text{from},j}\}_;^j\end{array}}{\mathcal{S}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}_|^j]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}}}$$

Figure 7. Conversions for data types in the standard view

$$\boxed{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^k\}_|^j]\!]^{\text{conv}} \equiv \{p_{\text{from},j}\}_|^j; \{p_{\text{to},j}\}_|^j}$$

$$\frac{}{\mathcal{S}[\![\varepsilon]\!]^{\text{conv}} \equiv \varepsilon; \varepsilon}\ \text{(conv-std-1)} \qquad \frac{}{\mathcal{S}[\![C]\!]^{\text{conv}} \equiv C; \mathit{Unit}}\ \text{(conv-std-2)} \qquad \frac{}{\mathcal{S}[\![C\ t]\!]^{\text{conv}} \equiv C\ x; x}\ \text{(conv-std-3)}$$

$$\frac{\begin{array}{c}n \in 2\,.\,. \qquad \{x_1 \not\equiv x_k\}^{k \in 2..n} \\ \mathcal{S}[\![C\ \{t_k\}^{k \in 2..n}]\!]^{\text{conv}} \equiv C\ \{x_k\}^{k \in 2..n}; p_{\text{to}}\end{array}}{\mathcal{S}[\![C\ \{t_k\}^{k \in 1..n}]\!]^{\text{conv}} \equiv C\ \{x_k\}^{k \in 1..n}; x_1 \times p_{\text{to}}}\ \text{(conv-std-4)}$$

$$\frac{\begin{array}{c}m \in 2\,.\,. \\ \mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_|^{j \in 2..m}]\!]^{\text{conv}} \equiv \{p_{\text{from},j}\}_|^{j \in 2..m}; \{p_{\text{to},j}\}_|^{j \in 2..m} \\ \mathcal{S}[\![C_1\ \{t_{1,k}\}^{k \in 1..n_1}]\!]^{\text{conv}} \equiv p_{\text{from},1}; p_{\text{to},1}\end{array}}{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_|^{j \in 1..m}]\!]^{\text{conv}} \equiv \{p_{\text{from},j}\}_|^{j \in 1..m}; \mathit{Inl}\ p_{\text{to},1}\ \{|\ \mathit{Inr}\ p_{\text{to},j}\}^{j \in 2..m}}\ \text{(conv-std-5)}$$

Figure 8. Conversions for constructors in the standard view

the corresponding variables to product patterns. Finally, if the list of constructors has two or more elements, rule (conv-std-5) applies; it prefixes the patterns with the injection constructors Inl and Inr.

THEOREM 4.6. *The standard view is valid.*

4.5 The fixed-point view

An essential aspect of the fixed-point view is the automatic derivation of pattern functors.

Given a declaration D_1 of type T, a declaration D_2 for the pattern functor $\mathsf{ptr}(T)$ is generated by the rule in Figure 9, which takes the form

$$[\![D_1]\!]^{\text{ptr}} \equiv D_2.$$

The metafunction ptr is assumed to produce a unique name for the functor. The definition of $\mathsf{ptr}(T)$ follows the structure of T, replacing all recursive calls by an extra type argument.

View types. The sole view type of the fixed-point view is Fix:

$$\boxed{[\![D_1]\!]^{\text{ptr}} \equiv D_2}$$

$$\frac{\{a_{\ell+1} \not\equiv a_i\}^i \qquad \{\{t'_{j,k} \equiv [\, a_{\ell+1} \,/\, T\ \{a_i\}^i\,]\ t_{j,k}\}^k\}^j_{|}}{[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}^j_{|}]\!]^{\text{ptr}} \equiv \mathbf{data}\ \mathsf{ptr}(T) = \{\Lambda a_i :: \kappa_i\ .\ \}^i\ \Lambda a_{\ell+1} :: *\ .\ \{\mathsf{ptr}(C_j)\ \{t'_{j,k}\}^k\}^j_{|}}$$

Figure 9. Pattern functors

$$\boxed{\mathcal{F}[\![D_0]\!]^{\text{str}} \equiv u; \{D_i\}^i_{,}}$$

$$\frac{D \equiv \mathbf{data}\ T = \{\Lambda a_i :: *\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}^j_{|}}{\mathcal{F}[\![D]\!]^{\text{str}} \equiv \{\Lambda a_i :: *\ .\ \}^i\ \text{Fix}\ (\mathsf{ptr}(T)\ \{a_i\}^i); [\![D]\!]^{\text{ptr}}}$$

Figure 10. Representation of data types in the fixed-point view

$\mathbf{data}\ \text{Fix} = \Lambda\varphi :: * \to *\ .\ In\ (\varphi\ (\text{Fix}\ \varphi))$.

Generating structure types. The rule for generating structure types for $\mathcal{F}$, which has the form

$$\mathcal{F}[\![D_0]\!]^{\text{str}} \equiv u; \{D_i\}^{i \in 1..n}_{,}.$$

is given in Figure 10. This rather straightforward rule states that a structure type is derived by applying the type Fix to a partially applied pattern functor. The declaration of the pattern functor is emitted as a supporting declaration. Note that the parameters of the original data type are restricted to kind $*$, excluding higher-order kinded types from the view domain. The need for this restriction will be explained later.

Generating conversions. Generating conversion functions for $\mathcal{F}$ is more involved. The algorithm is presented in Figures 11 and 12. It consists of judgements of the forms

$$\begin{array}{ll} \mathcal{F}[\![D_0]\!]^{\text{conv}} & \equiv e_{\text{from}}; e_{\text{to}} \\ \mathcal{F}[\![\{t_k\}^{k \in 1..n}]\!]^{\text{conv}}_{t_0;e,e'} & \equiv \{p_k\}^{k \in 1..n}; \{e_k\}^{k \in 1..n}. \end{array}$$

The first form indicates that conversion functions e_{from} and e_{to} are derived based on the structure of a type declaration D_0. The second form expresses the generation of pattern-expression pairs $\{p_k\}^{k \in 1..n}; \{e_k\}^{k \in 1..n}$ for a list of constructor fields $\{t_k\}^{k \in 1..n}$; the generation of these pairs is driven by the original type t_0 and the conversion functions e and e'.

Because the conversion functions may be mutually recursive, the rule in Figure 11 makes use of the core language's recursive **let** construct. To this end, the rule introduces fresh variables x_{from} and x_{to} that refer to the conversions. It relies on the rules for constructor fields to issue the recursive calls to x_{from} and x_{to} in the appropriate positions.

For the generation of patterns and expressions from lists of fields, we distinguish three cases. If the field list is empty, rule (conv-fix-1) applies and no patterns or expressions are generated. If the field list is non-empty, we recursively generate patterns and expressions for its tail, while focussing on the head element. If this head element equals the original type, rule (conv-fix-2) makes sure that the

$$\boxed{\mathcal{F}[\![D]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}}}$$

$$\frac{\begin{array}{c} t_0 \equiv T\ \{a_i\}^i \\ \{\mathcal{F}[\![\{t_{j,k}\}^k]\!]^{\text{conv}}_{t_0;x_{\text{from}};x_{\text{to}}} \equiv \{p_{j,k}\}^k; \{e_{j,k}\}^k\}^j \qquad \{\mathcal{F}[\![\{t_{j,k}\}^k]\!]^{\text{conv}}_{t_0;x_{\text{to}};x_{\text{from}}} \equiv \{p'_{j,k}\}^k; \{e'_{j,k}\}^k\}^j \\ x_{\text{from}} \notin \{\{\mathsf{pv}(p_{j,k}), \mathsf{pv}(p'_{j,k})\}^k_,\}^j_, \qquad x_{\text{to}} \notin \{\{\mathsf{pv}(p_{j,k}), \mathsf{pv}(p'_{j,k})\}^k_,\}^j_,\ x_{\text{from}} \\ d_{\text{from}} \equiv x_{\text{from}} = \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{C_j\ \{p_{j,k}\}^k \to In\ (\mathsf{ptr}(C_j)\ \{e_{j,k}\}^k)\}^j \\ d_{\text{to}} \equiv x_{\text{to}} = \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{In\ (\mathsf{ptr}(C_j)\ \{p'_{j,k}\}^k) \to C_j\ \{e'_{j,k}\}^k\}^j \\ e_{\text{from}} \equiv \mathbf{let}\ d_{\text{from}}; d_{\text{to}}\ \mathbf{in}\ x_{\text{from}} \qquad e_{\text{to}} \equiv \mathbf{let}\ d_{\text{from}}; d_{\text{to}}\ \mathbf{in}\ x_{\text{to}} \end{array}}{\mathcal{F}[\![\mathbf{data}\ T = \{\Lambda a_i :: *\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}^j_|]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}}}$$

Figure 11. Conversions for data types in the fixed-point view

$$\boxed{\mathcal{F}[\![\{t_k\}^k]\!]^{\text{conv}}_{t_0;e;e'} \equiv \{p_k\}^k; \{e_k\}^{k\in 1..n}}$$

$$\frac{}{\mathcal{F}[\![\varepsilon]\!]^{\text{conv}}_{t_0;e_{\text{conv}};e'_{\text{conv}}} \equiv \varepsilon;\varepsilon} \quad \text{(conv-fix-1)}$$

$$\frac{n \in 1\,..\qquad t_1 \equiv t_0 \qquad \{x_1 \not\equiv x_k\}^{k\in 2..n} \qquad \mathcal{F}[\![\{t_k\}^{k\in 2..n}]\!]^{\text{conv}}_{t_0;e_{\text{conv}};e'_{\text{conv}}} \equiv \{x_k\}^{k\in 2..n}; \{e_k\}^{k\in 2..n}}{\mathcal{F}[\![\{t_k\}^{k\in 1..n}]\!]^{\text{conv}}_{t_0;e_{\text{conv}};e'_{\text{conv}}} \equiv \{x_k\}^{k\in 1..n}; (e_{\text{conv}}\ x_1)\ \{e_k\}^{k\in 2..n}} \quad \text{(conv-fix-2)}$$

$$\frac{n \in 1\,..\qquad t_1 \not\equiv t_0 \qquad \{x_1 \not\equiv x_k\}^{k\in 2..n} \qquad \mathcal{F}[\![\{t_k\}^{k\in 2..n}]\!]^{\text{conv}}_{t_0;e_{\text{conv}};e'_{\text{conv}}} \equiv \{x_k\}^{k\in 2..n}; \{e_k\}^{k\in 2..n}}{\mathcal{F}[\![\{t_k\}^{k\in 1..n}]\!]^{\text{conv}}_{t_0;e_{\text{conv}};e'_{\text{conv}}} \equiv \{x_k\}^{k\in 1..n}; \mathsf{map}(x_1, t_1, t_0, e_{\text{conv}}, e'_{\text{conv}})\ \{e_k\}^{k\in 2..n}} \quad \text{(conv-fix-3)}$$

Figure 12. Conversions for fields in the fixed-point view

conversion function is applied. Rule (conv-fix-3) deals with the situation in which the head element does not equal the original type. In that case it may be necessary to map the conversion functions over a fixed data structure; here, we assume that we have a metafunction map that takes care of the details.

THEOREM 4.7. *The fixed point view is valid.*

Higher-order kinded types. Unfortunately there is a class of data types that troubles the implementation of the fixed-point view: higher-order kinded types, i.e., types that have one or more parameters that range over parametric types rather than proper types; for instance

$$\mathbf{data}\ \mathrm{GRose}\ f\ a = GBranch\ a\ (f\ (\mathrm{GRose}\ f\ a)).$$

For the fixed-point view, these types might require embedding-projection functions to map over arbitrary structures. We have considered some ad-hoc solutions to this problem, but they all require significant changes to the underlying specialization algorithm (see Section 5.2). Here, we have adopted

the most pragmatic approach: we have excluded types with a higher-order kind from the domain of the fixed-point view. In the sequel, we present an alternative solution that involves a modification of the view.

Alternative solution. To circumvent the problems with higher-order kinded types, we consider a view in which recursive calls in data types are modelled by a type similar to Fix, but which also maintains an embedding-projection pair between the original data type and its representation as a fixed point.

$$\mathbf{data}\ \mathrm{Rec}\ f\ r = InR\ (f\ r)\ (\mathrm{EP}\ r\ (\mathrm{Rec}\ f\ r))$$

Like Fix, Rec takes a type argument of kind $* \to *$, which will be used to pass in the base functor. Additionally, Rec takes an argument of kind $*$ that will represent the data type itself. The structural representation for a type $T :: \{* \to\}^i *$ is now given by

$$\mathbf{type}\ T^{\circ}_R\ \{a\}^i = \mathrm{Rec}\ (\mathsf{ptr}(T)\ \{a\}^i)\ (T\ \{a\}^i).$$

As defined, a value of type $T^{\circ}_R\ \{a\}^i$ consists of two parts: a value of type $\mathsf{ptr}(T)\ \{a\}^i$ and an embedding-projection pair witnessing the isomorphism between T and T°_R.

The need for explicitly encoding the isomorphism into the structure type becomes clear when we consider the Rec case for the generic function $children$. Instantiating the type of $children$, given in Section 3, to Rec yields (dependencies omitted):

$$children\langle \mathrm{Rec}\ \varphi\ \rho\rangle :: \forall f\ r\,.\,(\ldots) \Rightarrow \mathrm{Rec}\ f\ r \to [\mathrm{Rec}\ f\ r].$$

The 'natural' definition of the case for Rec does not adhere to this type though,

$$\begin{array}{l} children\langle \mathrm{Rec}\ \varphi\ \rho\rangle\ (InR\ r\ \_) = \\ \quad \mathbf{let}\ col\langle\alpha\rangle\ a = [a] \\ \quad \mathbf{in}\ \ col\langle\varphi\ \alpha\rangle\ r \quad \text{-- type incorrect,} \end{array}$$

because it produces a list of which the elements are of type r rather than type $\mathrm{Rec}\ f\ r$. This can be fixed using the embedding-projection pair that is contained within the Rec value:

$$\begin{array}{l} children\langle \mathrm{Rec}\ \varphi\ \rho\rangle\ (InR\ r\ ep) = \\ \quad \mathbf{let}\ col\langle\alpha\rangle\ a = [from\ ep\ a] \\ \quad \mathbf{in}\ \ col\langle\varphi\ \alpha\rangle\ r. \end{array}$$

The compiler-derived embedding-projection maps for the Rec view are included in the generated structure-type values:

$$\begin{array}{l} \mathbf{data}\ \mathrm{GRoseBase}\ f\ a = GBranchBase\ a\ (f\ r) \\ \mathbf{type}\ \mathrm{GRose}^{\circ}_R\ f\ a \quad = \\ \quad \mathrm{Rec}\ (\mathrm{GRoseBase}\ f\ a)\ (\mathrm{GRose}\ f\ a) \\ from_{\mathrm{GRose},R} :: \forall f\ a\,.\,\mathrm{GRose}\ f\ a \to \mathrm{GRose}^{\circ}_R\ f\ a \\ from_{\mathrm{GRose},R}\ (GBranch\ a\ as) = \\ \quad InR\ (GBranchBase\ a\ as)\ conv_{\mathrm{GRose},R} \\ to_{\mathrm{GRose},R} \quad :: \forall f\ a\,.\,\mathrm{GRose}^{\circ}_R\ f\ a \to \mathrm{GRose}\ f\ a \\ from_{\mathrm{GRose},R}\ (InR\ (GBranchBase\ a\ as)\ ep) = \\ \quad GBranch\ a\ as \\ conv_{\mathrm{GRose},R} :: \forall f\ a\,.\,\mathrm{EP}\ (\mathrm{GRose}\ f\ a)\ (\mathrm{GRose}^{\circ}_R\ f\ a) \\ conv_{\mathrm{GRose},R} = EP\ from_{\mathrm{GRose},R}\ to_{\mathrm{GRose},R}. \end{array}$$

Instead of applying the conversion functions recursively, they are embedded in the structure-type value. Hence, we do not encounter problems with higher-order kinded types, as we do for representations involving Fix.

Using the modified structural representation and embedding-projection pairs yields a variant of Fix which we call Rec.

5. Generic views in the Generic Haskell compiler

At the moment, the Generic Haskell compiler itself has to be modified in order to add a new view. It is not yet possible to specify new views in a Generic Haskell program.

In this section we describe how to modify the Generic Haskell compiler to add new views. This amounts to identifying the part that currently implements the standard view, and abstracting from the information that constitutes the standard view. New views can then be added to the compiler by implementing a number of Haskell functions that directly correspond to the definition of a generic view in Section 4. Adding a new view could be made more comfortable by using Template Haskell [28] or a plug-in architecture [27].

Ultimately, we would like a special-purpose language for defining own views in user programs. Since a generic view consists of a set of view types, a function that generates structure types, and a function that generates conversion functions, the special-purpose language for specifying views is a complete programming language in itself. A compiler for a language in which generic views can be specified would contain two phases, in which the first phase compiles the views, and the second phase uses the compiled views to generate code for functions using those views. The special-purpose language for specifying generic views remains future work.

5.1 Abstracting from the standard view

In the Generic Haskell compiler, a view would have the following type:

$$
\begin{array}{l}
\mathbf{data}\ \mathrm{View} = \\
\quad \mathit{View}\ \mathrm{Name} \\
\qquad (\mathrm{TDecl} \to \\
\qquad\quad \mathrm{Maybe}\ (\mathrm{LamType}, [\mathrm{TDecl}], \mathrm{Expr}, \mathrm{Expr})).
\end{array}
$$

A value of type View consists of a name, and a function that can be called on the declaration of a type synonym or a datatype (a TDecl) to produce a parameterized structural representation type (a LamType) and an embedding-projection pair (two *expr*'s). Views that apply to a subset of the Haskell datatypes can be implemented by returning *Nothing* on datatype definitions that are outside of the view's domain.

Note that the result of the view-generating function in type View directly corresponds to the maps $\mathcal{V}[\![\cdot]\!]^{\mathrm{str}}$ and $\mathcal{V}[\![\cdot]\!]^{\mathrm{conv}}$. The collection $\mathsf{viewtypes}_{\mathcal{V}}$ of bindings that are required by the view must be added to the Generic Haskell Prelude, i.e., they must be available for the Generic Haskell compiler to parse.

The functionality that makes up the standard view is currently distributed over multiple source files in the Generic Haskell compiler, but it can easily be extracted into a value of type View. For other views such as the fixed-point view, other values of type View can be defined.

The whole compiler can then be parameterized over a list of views [View].

The validity of a view can only be checked to a certain extent. The compiler can verify the kind preservation and well-typed conversion properties of the view: for each structural representation and embedding-projection pair generated, kind and type checking is performed. The well-behavedness of the conversion cannot be verified by the compiler, since verifying that the composition of two arbitrary

functions is the identity is an undecidable problem. This property remains a proof obligation for the implementor of the view.

5.2 Specialization

In this section we sketch how a generic view is actually put to use by the compiler. Assume that enc, the encode function from Section 3.2, is called on the type argument Bool. No case is given for Bool, so the standard view is applied. The data type Bool is defined as

$$\mathbf{data}\ \mathrm{Bool} = \mathit{False} \mid \mathit{True},$$

and the top-level structural representation type of Bool is given by

$$\mathbf{type}\ \mathrm{Bool}^\circ = \mathrm{Unit} + \mathrm{Unit}.$$

We reduce a call of $enc\langle\mathrm{Bool}\rangle$ to a call $enc\langle\mathrm{Bool}^\circ\rangle$. The representation Bool° that is actually used now depends on the view specified for the enc function! The translation of the latter function to Haskell code is described elsewhere [10, 21]. The call $enc\langle\mathrm{Bool}^\circ\rangle$ is of type $\mathrm{Bool}^\circ \to [\mathrm{Bit}]$, whereas $enc\langle\mathrm{Bool}\rangle$ is of type $\mathrm{Bool} \to [\mathrm{Bit}]$. So to express the call of $enc\langle\mathrm{Bool}\rangle$ in terms of the the call of $enc\langle\mathrm{Bool}^\circ\rangle$, we have to lift the isomorphism between Bool and its representation to the type of the generic function enc.

Given an embedding-projection map between a type D and its structure type D°, we can use the generic function $bimap$ to lift the isomorphism to arbitrarily complex types. Recall that enc, the collection function in Section 2.1, is defined in such a way that it returns the empty list for every data type, and only becomes useful when locally redefined. Similarly, $bimap$ defines the identity embedding-projection pair for each data type generically. A remarkable fact is that $bimap$ can be defined on function types. We give the cases for Unit, $+$, and $\to$ as an example (see, for example, [21] for a complete definition):

$$\begin{array}{l}
bimap\langle a_1 :: *, a_2 :: *\rangle :: (bimap\langle a_1, a_2\rangle) \Rightarrow \mathrm{EP}\ a_1\ a_2 \\
bimap\langle\mathrm{Unit}\rangle \quad = \mathit{EP}\ \mathit{id}\ \mathit{id} \\
bimap\langle\alpha + \beta\rangle\ = \\
\quad \mathbf{let}\ \mathit{from}_+\ (\mathit{Inl}\ a) \quad = \mathit{Inl}\ (\mathit{from}\ bimap\langle\alpha\rangle\ a) \\
\qquad \mathit{from}_+\ (\mathit{Inr}\ b) \quad = \mathit{Inr}\ (\mathit{from}\ bimap\langle\beta\rangle\ b) \\
\qquad \mathit{to}_+ \quad (\mathit{Inl}\ a) \quad = \mathit{Inl}\ (\mathit{to} \quad bimap\langle\alpha\rangle\ a) \\
\qquad \mathit{to}_+ \quad (\mathit{Inr}\ b) \quad = \mathit{Inr}\ (\mathit{to} \quad bimap\langle\beta\rangle\ b) \\
\quad \mathbf{in}\ \ \mathit{EP}\ \mathit{from}_+\ \mathit{to}_+ \\
bimap\langle\alpha \to \beta\rangle = \\
\quad \mathbf{let}\ \mathit{from}_\to\ c = \mathit{from}\ bimap\langle\beta\rangle \cdot c \cdot \mathit{to} \quad bimap\langle\alpha\rangle \\
\qquad \mathit{to}_\to \quad c = \mathit{to} \quad bimap\langle\beta\rangle \cdot c \cdot \mathit{from}\ bimap\langle\alpha\rangle \\
\quad \mathbf{in}\ \ \mathit{EP}\ \mathit{from}_\to\ \mathit{to}_\to.
\end{array}$$

Using local redefinition, we can plug in an embedding-projection pair provided by the generic view:

$$\begin{array}{l}
enc\langle\mathrm{Bool}\rangle = \mathbf{let}\ bimap\langle\alpha\rangle = ep_{\mathrm{Bool}} \\
\qquad\qquad\qquad\ \mathbf{in}\ \ \mathit{to}\ (bimap\langle\alpha \to [\mathrm{Bit}]\rangle)\ enc\langle\mathrm{Bool}^\circ\rangle.
\end{array}$$

The details of why this works are omitted here. It is, however, important to realize that for generic functions that both consume and produce values of the type argument's type, both components of the embedding projection pair will be applied: a value of the original type D is transformed into D° to be in suitable form to be passed to the function that works on the structural representation. Because the

function also returns something containing values of type D°, these values are then converted back to type D. This is the reason why the embedding-projection pair should really contain an isomorphism. If it does not, a value could change simply by the conversion functions that are applied, leading to unexpected results for the user.

The specialization mechanism is independent of the actual view. For other views than the standard view, different structural representations and embedding-projection pairs are used, but the specialization procedure remains exactly the same. The only thing that must be changed in the implementation of specialization within the Generic Haskell compiler is that all the references to structural representation types and embedding-projection pairs should point to the view that is specified for the function in question.

6. Conclusions, related and future work

We have shown that generic views on data types can make generic functions both easier to write and more efficient. Furthermore, generic views allow us to use different generic programming styles in a single framework.

Although there are a multitude of approaches for generic programming, the idea to use multiple views on data types in a single approach is, to the best of our knowledge, original.

The name "generic view" is derived from Wadler's proposal to introduce views in (a predecessor of) Haskell [31]. Using one of these views, a single Haskell data type can be analyzed in a different way, by introducing additional constructors by which a value can be constructed, and on which pattern matching can be performed. A view is essentially like the introduction of an additional data type, together with the definition of conversion functions between values from the original type and values of the view type. These conversions are then transparently applied by the compiler where necessary.

Generic views are different in that they define a representation and conversions for many types at the same time. Furthermore, the representation types need not be new data types, but can be built from existing data types. Wadler's views have the immense advantage that they can be added to the Haskell programming language relatively easily, allowing every programmer to add her own views. On the other hand, generic views have, for now, to be added to a generic programming system, such as the Generic Haskell compiler, following the guidelines described in the previous sections. Designing a language extension of Generic Haskell that allows user-defined views is on the top of the list of future work.

Both views and generic views have in common that the definition of a new view goes along with a proof obligation for the programmer that cannot easily be captured in a language like Haskell. The conversion between original type and view type, be it a single pair of functions such as in Wadler's proposal, or a type-indexed family of functions such as for generic views, must really witness isomorphisms, otherwise unexpected results may occur.

Since Wadler's views proposal, several variations of views have been given [5, 26, 6]. Our approach is closest to Wadler's proposal in that we also require the existence of an isomorphism between the original type and the view type.

Views have also been proposed in the context of XML and databases [1, 25]. Generic views as proposed in [29] are used to automatically convert between two given views. The generic view concept as introduced in this paper does not seem to have been investigated in this field.

The idea of using different sets of data types for inductive definitions of type-indexed functions is common in the world of dependent types [2, 4]. This corresponds to the idea of having views that work on different subsets of the Haskell data types. However, in the approaches we have seen there is no automatic conversion between syntactically definable data types as offered by the dependently typed programming language into representations as defined by the view or universe.

Acknowledgements. Our thanks go to three anonymous referees and Daan Leijen for several helpful comments.

References

[1] S. Abiteboul. On views and XML. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–9. ACM Press, 1999.

[2] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11–12, 2002, Dagstuhl, Germany*, number 115 in International Federation for Information Processing, pages 1–20. Kluwer Academic Publishers, 2003.

[3] Roland Backhouse and Paul Hoogendijk. Generic properties of data types. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 97–132. Springer-Verlag, 2003.

[4] 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.

[5] F. W. Burton and R. D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):117–190, 1993.

[6] F. Warren Burton, Erik Meijer, Patrick Sansom, Simon Thompson, and Philip Wadler. Views: an extension to Haskell pattern matching. Available from `http://www.haskell.org/development/views.html`, 1996.

[7] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press, 2002.

[8] Dave Clarke and Andres Löh. Generic Haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming July 11–12, 2002, Dagstuhl, Germany*, number 115 in International Federation for Information Processing, pages 21–47. Kluwer Academic Publishers, 2003.

[9] 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.

[10] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.

[11] Ralf Hinze and Johan Jeuring. Generic Haskell: applications. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 57–97. Springer-Verlag, 2003.

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

[13] Stefan Holdermans. Generic views. Master's thesis, Institute of Information and Computing Sciences, Utrecht University, 2005.

[14] Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

[15] 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.

[16] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In Doaitse Swierstra, editor, *Proceedings of the 8th European Symposium on Programming, ESOP'99*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.

[17] Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In Johan Jeuring, editor, *Workshop on Generic Programming 2000, Ponte de Lima, Portugal, July 2000*, pages 33–45, 2000. Utrecht Technical Report UU-CS-2000-19.

[18] 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*, volume 2044 of *LNCS*, pages 217–239. Springer-Verlag, 2001.

[19] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. Proceedings ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[20] 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.

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

[22] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 141–152. ACM Press, 2003.

[23] Andres Löh, Johan Jeuring, Dave Clarke, Ralf Hinze, Alexey Rodriguez, and Jan de Wit. The Generic Haskell user's guide, version 1.42 (Coral). Technical Report UU-CS-2005-004, Institute of Information and Computing Sciences, Utrecht University, 2005.

[24] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, FPCA 1991*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

[25] A. Ohori and K. Tajima. A polymorphic calculus for views and object sharing. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–266, 1994.

[26] Chris Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML*, pages 14–23, 1998.

[27] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell In, 2004.

[28] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop, HW 2002*, pages 1–16. ACM Press, 2002.

[29] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proceedings of the International Conference on Extensive Data Base Technology (EDBT'94), Cambridge, UK*, pages 81–94. Springer-Verlag, 1994.

[30] W3C. XML Schema: Formal description, 2001.

[31] Phil Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Conference Record of POPL '87: The 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1987.