Generic Views on Data Types

Stefan Holdermans[1], Johan Jeuring[1], Andres Löh[2], and Alexey Rodriguez[1]

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

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 shows how generic views have been added 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 combine 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 [1–5] 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. The compiler generated functions, however, are also defined by induction on the structure of types.
- 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 chooses a different one.

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 [1] a data type is viewed as the fixed point of a pattern functor that has 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 [6], and functions that return the direct recursive children of a constructor [7]. A downside of this view on data types is that PolyP can only handle regular data types of kind $* \to *$.
- In Generic Haskell [2, 8, 9], a data type is described in terms of a top-level sums of products 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' [3, 10] 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, for example, 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, 9]. 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 [4], and in the work of De Moor and Hoogendijk [5].

Generic views on data types. An approach to generic programming essentially consists of two components: a facility to define recursive functions on a specific set of types, called view types, and a view on the inductive structure of data types, which maps data types onto view types. We call such a view on the structure of types a *generic view* (or just *view*) on data types. Wadler [12] also defines views on data types. The difference between Wadler's views and generic views is that the former constitute a method for viewing a single data type in different ways, whereas the latter 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 generic views have been added to Generic Haskell;
– provides a common framework which can be used to compare different approaches to generic programming.

Views add expressiveness to a generic programming language. Generic functions still work for arbitrary data types that can be expressed in the view, but the choice between different views allows us to define more generic functions.

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 formally defines a generic view. For some of the examples of Section 3, we give the formal definition. Section 5 discusses how views have been added to the Generic Haskell compiler. Section 6 gives related work and conclusions.

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 [13, 11, 9]. Generic Haskell has slightly changed in the last couple of years, and we will use the version described in Löh's thesis 'Exploring Generic Haskell' [9] (EGH) in this paper, which to a large extent has been implemented in the Coral release [8].

2.1 Type-indexed functions

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

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

We use infix types + and × and an infix value constructor × here to ease the presentation. The type-indexed function *collect* collects values from a data structure. We define function *collect* on the unit type, sums and products, integers, and characters as follows:

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

The type signature of *collect* is as follows:

$$collect\langle a :: * \mid c :: *\rangle :: (collect\langle a \mid c\rangle) \Rightarrow a \to [c].$$

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 parametrically polymorphic in the element type of its list result. It always returns the empty list, but we will show below 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 $(collect\langle a \mid c\rangle) \Rightarrow$ appears in the type because *collect* is called recursively on sums and products, which means that, for example, if we want an instance of *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.

The type signature of *collect* can be instantiated for specific cases, including cases omitted in the definition as we shall see later, by the Generic Haskell compiler, yielding, for example, the types

$$\begin{aligned} &collect\langle \text{Unit}\rangle :: \forall c \,.\, \text{Unit} \to [c] \\ &collect\langle [\alpha]\rangle \;\; :: \forall c \;a \,.\, (collect\langle \alpha\rangle :: a \to [c]) \Rightarrow [a] \to [c] \end{aligned}$$

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

Depending on the situation, the function $collect\langle\alpha\rangle$ 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, we write:

$$\begin{aligned} &\textbf{let } collect\langle\alpha\rangle\ x = \textbf{if } x > 0 \textbf{ then } [x] \textbf{ else } [\,] \\ &\textbf{in } \;\; collect\langle[\alpha]\rangle, \end{aligned}$$

which has type $Num\ a \Rightarrow [a] \to [a]$. Generally, we use a *local redefinition* to locally modify the behavior of a generic function. Some generic functions such as *collect* only reveal their full power in the context of local redefinitions.

2.2 Default cases

Suppose we wish to use function *collect* to collect the variables of the data type Term, which represents lambda terms:

$$\begin{aligned} &\textbf{data } \text{Term} \;\;\;\;\;\;\;\; = Var\ \text{Variable} \mid Abs\ \text{Variable Term} \mid App\ \text{Term Term} \\ &\textbf{newtype } \text{Variable} = V\ \text{String}. \end{aligned}$$

We cannot use local redefinitions as we did for the list case, because this would require that the data type Term is parameterized over the type of variables. Instead, we write function *varcollect* making use of *default cases*:

$$\begin{aligned}&\mathit{varcollect}\langle\text{Variable}\rangle\ v = [v]\\&\mathit{varcollect}\ \textbf{extends}\ \mathit{collect}\\&\quad\textbf{where}\ \mathit{collect}\ \textbf{as}\ \mathit{varcollect}.\end{aligned}$$

The first line defines the Variable case of *varcollect*. The next two lines copy the definition of *collect* and rename its dependency on *collect* to *varcollect*. The use of a default case is equivalent to manually copying the definition of *collect*, replacing the calls to *collect* with calls to *varcollect*, and adding the case for Variable. The more specific behavior of *varcollect* is reflected in its type signature:

$$\mathit{varcollect}\langle a :: * \rangle :: (\mathit{varcollect}\langle a \rangle) \Rightarrow a \to [\text{Variable}].$$

2.3 View types

A type-indexed function such as *collect* does not only work on the types that appear as its type indices. To see why *collect* is in fact *generic* and works on arbitrary data types, we give a mapping from data types to view types such as units, sums and products. It suffices to define a function on view types (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 behavior is derived automatically by the compiler by exploiting the structural representation.

For example, the definition of the function *collect* generically derived for lists coincides with the following specific definition:

$$\begin{aligned}&\mathit{collect}\langle[\alpha]\rangle\ [\,] &&= [\,]\\&\mathit{collect}\langle[\alpha]\rangle\ (x : xs) &&= \mathit{collect}\langle\alpha\rangle\ x +\!\!+ \mathit{collect}\langle[\alpha]\rangle\ xs.\end{aligned}$$

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

$$\begin{aligned}&\textbf{data}\ \text{List}\ a &&= \mathit{Nil} \mid \mathit{Cons}\ a\ (\text{List}\ a)\\&\textbf{data}\ \text{Tree}\ a\ b &&= \mathit{Tip}\ a \mid \mathit{Node}\ (\text{Tree}\ a\ b)\ b\ (\text{Tree}\ a\ b)\end{aligned}$$

we obtain the following structural representations:

$$\begin{aligned}&\textbf{type}\ \text{List}^{\circ}\ a &&= \text{Unit} + a \times \text{List}\ a\\&\textbf{type}\ \text{Tree}^{\circ}\ a\ b &&= a + \text{Tree}\ a\ b \times b \times \text{Tree}\ a\ b,\end{aligned}$$

where we assume that $\times$ binds stronger than $+$, and both type constructors associate to the right. Note that the representation of a recursive type is not

recursive, and refers to the recursive type itself. The representation of a type in Generic Haskell only represents the structure of the top level of the type.

If two types are isomorphic, the corresponding isomorphisms, also called embedding-projection pairs, can be stored as a pair of functions converting back and forth:

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

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 $conv_{\mathrm{List}} = EP\ from_{\mathrm{List}}\ to_{\mathrm{List}}$, where $from_{\mathrm{List}}$ and to_{List} are defined by

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

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

Using structural representation types and embedding-projection pairs, a call to a generic function on a data type T is reduced to a call on type T°. Hence, if the generic function is defined for view types such as Unit, $+$, and $\times$, we do not need cases for specific data types such as List or Tree anymore. For primitive types such as Int, Float, IO or $\to$, no structure is available. Therefore, for a generic function to work on these types, a specific case is necessary.

2.4 Specializing generic functions

In this section we sketch how Generic Haskell specializes a generic function. Assume that *collect*, the collect function from Section 2.1, is called on the type argument Bool. No case is given for Bool, so Generic Haskell considers the structural representation for Bool. The data type Bool and its structural representation are given by

$$\begin{array}{l}
\textbf{data}\ \mathrm{Bool}\ \ = False \mid True, \\
\textbf{type}\ \mathrm{Bool}^\circ = \mathrm{Unit} + \mathrm{Unit}.
\end{array}$$

We reduce a call of $collect\langle\mathrm{Bool}\rangle$ to a call $collect\langle\mathrm{Bool}^\circ\rangle$. The translation of the latter function to Haskell code, using the cases of *collect* for view types, is quite simple and described elsewhere (EGH,[2]). The call $collect\langle\mathrm{Bool}^\circ\rangle$ is of type $\mathrm{Bool}^\circ \to [c]$, whereas $collect\langle\mathrm{Bool}\rangle$ is of type $\mathrm{Bool} \to [c]$. So to express the call of $collect\langle\mathrm{Bool}\rangle$ in terms of the the call of $collect\langle\mathrm{Bool}^\circ\rangle$, we have to lift the isomorphism between Bool and its representation to the type of the generic function *collect*.

Given an embedding-projection pair 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 *collect* 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 → as an example (see, for example, EGH for a complete definition):

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

Using local redefinition, we can plug in an embedding-projection pair in *bimap* to lift the isomorphism between Bool and its representation to the type of the generic function *collect*.

$$collect\langle \mathrm{Bool}\rangle = \textbf{let}\ bimap\langle\alpha\rangle = ep_{\mathrm{Bool}}\ \textbf{in}\ to\ (bimap\langle\alpha \to [\,c]\rangle)\ collect\langle \mathrm{Bool}^\circ\rangle .$$

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 contain an isomorphism. If it does not, a value could change simply by the conversion functions that are applied, making it highly difficult to define, for example, the generic identity function.

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 improve the expressiveness of a generic programming system, because not every view is equally suitable for every generic function. In Section 4 we will give a formal definition of generic views.

3.1 Fixed points

Consider the data type Term, introduced in Section 2.2, and the function *subterms* that, given a term, produces the immediate subterms.

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

This function is an instance of a more general pattern. The function *subtrees*, for example, produces the immediate subtrees of an external binary search tree.

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

Given a recursive data type's value, both *subterms* and *subtrees* retrieve the immediate children corresponding to the recursion points in the data type's definition. 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 occurrences 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 [7]. Generic functions in PolyP, however, are limited in the sense that they can only be applied to regular[3] data types of kind $* \to *$. In particular, this precludes nested and mutually recursive data types.

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:

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

then the generic function *children* can be defined with a single case for Fix.

$$
\begin{aligned}
&\mathit{children}\langle a :: * \rangle :: (\forall c\,.\,\mathit{collect}\langle a \mid c \rangle) \Rightarrow a \to [a] \\
&\mathit{children}\langle \text{Fix}\ \varphi \rangle\ (\mathit{In}\ r) = \textbf{let}\ \mathit{collect}\langle \alpha \rangle\ x = [x]\ \textbf{in}\ \mathit{collect}\langle \varphi\ \alpha \rangle\ r
\end{aligned}
$$

[3] A data type is *regular* if it does not contain function spaces, and if the arguments of the type constructor on the left- and right-hand sides in its definition are the same. So the data type *Flip* defined by **data** *Flip a b* = *MkFlip a* (*Flip b a*) is not regular.

The *children* function depends on the collect function *collect*[4] 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 *collect* 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 *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:

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

The definition of *children* is unchanged. For example, $children\langle[\mathrm{Int}]\rangle\ [1,2,3]$ yields $[[2,3]]$. 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.

Another well-known function that can be defined using the fixed-point view is the *catamorphism* [14]. In the definition of *cata* we use a type-indexed type AlgebraOf, which returns the algebra of a data type: a function from the pattern functor of the data type to the result type. The details of this definition can be found in EGH, and in the forthcoming release of Generic Haskell with 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 behavior of certain generic programs. The generic function *enc*, for instance, encodes values of data types as lists of bits.

$$\begin{aligned}
&\mathbf{data}\ \mathrm{Bit} = \mathit{Off} \mid \mathit{On}\\
&enc\langle a :: *\rangle :: (enc\langle a\rangle) \Rightarrow a \to [\mathrm{Bit}]\\
&enc\langle\mathrm{Unit}\rangle\ \ \mathit{Unit}\ \ \ \ = [\,]\\
&enc\langle\alpha+\beta\rangle\ (\mathit{Inl}\ a) = \mathit{Off} : enc\langle\alpha\rangle\ a
\end{aligned}$$

[4] One might be tempted to write $collect\langle a \mid a\rangle$ for the dependency, but this produces incorrect type signatures for some specializations of *children*. The reason is that the non-generic variable of *collect* must have kind $*$, which in general does not hold since variable a can have any arbitrary kind.

$$
\begin{aligned}
&enc\langle \alpha + \beta \rangle\ (Inr\ b) &&= On : enc\langle \beta \rangle\ b \\
&enc\langle \alpha \times \beta \rangle\ (a \times b) &&= enc\langle \alpha \rangle\ a +\!\!+ enc\langle \beta \rangle\ b \\
&enc\langle \mathrm{Int} \rangle\quad n &&= encInt\ n \\
&enc\langle \mathrm{Char} \rangle\ c &&= encChar\ c
\end{aligned}
$$

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.

Applying a nested right-deep view to the type *Compass* of directions

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

gives the structure

$$\textbf{type}\ Compass^{\circ}\ = \mathrm{Unit} + (\mathrm{Unit} + (\mathrm{Unit} + \mathrm{Unit})).$$

Using this structure, 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.,

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

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:

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

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 *showP* that shows only a part of a value. The part that is shown is determined by a path of type

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

Non-empty lists of type Path select a part of the value 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 has type

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

The motivation for *showP* comes from the Proxima editor [15], where there is a need to generically handle paths to selections in arbitrary documents. Let us look at the definition of *showP* on products:

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

If the first path element is 0, the left component 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.

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

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:

$$\begin{array}{l}\textbf{data}\ \text{Con012}\ a\ b\ \ = \mathit{Con0}\ \mid\ \mathit{Con1}\ a\ \mid\ \mathit{Con2}\ a\ b\\ \textbf{type}\ \text{Con012}^{\circ}\ a\ b = \text{Unit}\ \ + a\qquad\quad + a \times b.\end{array}$$

Using the standard view of Generic Haskell a product structure is only created if there are at least two fields. If there is only one field, such as for *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 *showP* to pass additional information around. However, consider a data type Rename and its structural representation:

$$\begin{array}{l}\textbf{data}\ \text{Rename}\ \ = R\ \mathit{Original}\\ \textbf{type}\ \text{Rename}^{\circ} = \mathit{Original}.\end{array}$$

The structural representation does not even contain a sum structure. Although it is possible to write *showP* 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).

$$\textbf{data}\ \text{Zero}$$

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:

type $\mathrm{Con012}^{\circ}_{L}\ a\ b = \mathrm{Unit} + a \times \mathrm{Unit} + a \times b \times \mathrm{Unit} + \mathrm{Zero}$.

Now, we can define *showP* easily:

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

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 [3, 10]. 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 based on this view. The emulation of the boilerplate approach in Generic Haskell is useful for comparing different approaches to generic programming, but it turns out to be less convenient to use than the original boilerplate library due to the lack of higher-order generic functions.

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 behavior 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.

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

type $\mathrm{List}^{\circ}_{BP}\ a \quad = \mathrm{Unit} + (\mathrm{Unit} \times a) \times \mathrm{List}\ a$
type $\mathrm{Tree}^{\circ}_{BP}\ a\ b = \mathrm{Unit} \times a + ((\mathrm{Unit} \times \mathrm{Tree}\ a\ b) \times b) \times \mathrm{Tree}\ a\ b$.

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. To emulate type extensions, we have to be able to distinguish types by name. Therefore we use a type-indexed

function equipped with cases for specific types. The remaining cases – that is, the extension – are provided in a definition on a view that does not operate on the structure of types. For this purpose, we use the identity view (Id), which merely wraps a data type in the Id data type:

$$\textbf{data}\ \mathrm{Id}\ a = Id\ a.$$

For example, consider the function *addPrefixVar* that adds prefixes to variables:

$$\begin{aligned} &addPrefixVar && :: \mathrm{Variable} \to \mathrm{Variable} \\ &addPrefixVar\ (V\ x) &&= V\ (\texttt{"gh\_"} +\!\!+ x), \end{aligned}$$

this function is extended as follows to work on any type:

$$\begin{aligned} &addPrefix\langle a :: * \ \textbf{viewed}\ \mathsf{Id}\rangle :: a \to a \\ &addPrefix\langle \mathrm{Variable}\rangle\ x &&= addPrefixVar\ x \\ &addPrefix\langle \mathrm{Id}\ \alpha\rangle\ (Id\ x) &&= Id\ x. \end{aligned}$$

The Generic Haskell specialization algorithm chooses the Variable arm when *addPrefix* is applied to that type. For all other types, the last arm is selected.

The definitions of the traversal combinators rely on the list-like character of the Boilerplate view. For example, the *gmapT* combinator applies a transformation argument to the immediate children of a node, traversing it in a right to left fashion. The transformation argument is a type-extended function, which in our approach is modeled by a type-indexed function.

We implement type-extended arguments to combinators as type-indexed functions bound to the combinator's name followed by the *Par* suffix. For example, the *gmapT* combinator has the following definition (omitting the dependencies in the type, these can easily be inferred from the function definition):

$$\begin{aligned} &gmapT\langle a :: * \ \textbf{viewed}\ \mathsf{Boilerplate}\rangle :: a \to a \\ &gmapT\langle \mathrm{Unit}\rangle\ \ Unit &&= Unit \\ &gmapT\langle \alpha + \beta\rangle\ (Inl\ a) &&= Inl\ (gmapT\langle \alpha\rangle\ a) \\ &gmapT\langle \alpha + \beta\rangle\ (Inr\ b) &&= Inr\ (gmapT\langle \beta\rangle\ b) \\ &gmapT\langle \alpha \times \beta\rangle\ (a \times b) &&= gmapT\langle \alpha\rangle\ a \times gmapTPar\langle \beta\rangle\ b. \end{aligned}$$

The default definition of the transformation argument is the identity:

$$\begin{aligned} &gmapTPar\langle a :: * \ \textbf{viewed}\ \mathsf{Id}\rangle :: a \to a \\ &gmapTPar\langle \mathrm{Id}\ \alpha\rangle\ (Id\ x) &&= Id\ x. \end{aligned}$$

The *everywhere* combinator applies a transformation to all nodes in a tree, traversing it in a bottom-up fashion. It is defined in terms of the simple non-recursive one-layer traversal combinator *gmapT*, or rather in terms of *gmapTInst*, an instance of *gmapT* where the parameter *gmapTPar* is instantiated with *everywhere*.

$$\begin{aligned} &everywhere\langle a :: * \ \textbf{viewed}\ \mathsf{Id}\rangle :: a \to a \\ &everywhere\langle \mathrm{Id}\ \alpha\rangle\ (Id\ x) &&= Id\ (everywherePar\langle \alpha\rangle\ (gmapTInst\langle \alpha\rangle\ x)) \\ &everywherePar\langle a :: * \ \textbf{viewed}\ \mathsf{Id}\rangle :: a \to a \\ &everywherePar\langle \mathrm{Id}\ \alpha\rangle\ (Id\ x) &&= Id\ x \end{aligned}$$

The first function transforms the children by means of a call to *gmapTInst* and then applies the transformation argument *everywherePar* to the result. The generic function *gmapTInst* is the defunctionalized equivalent of the application of *gmapT* to *everywhere*:

```
gmapTInst⟨a :: ∗ viewed Boilerplate⟩ :: a → a
gmapTInst extends gmapT
  where gmapT      as gmapTInst
        gmapTPar as everywhere.
```

Function *gmapTInst* is defined by means of a *default case*: it behaves as *gmapT* except that the dependency on *gmapTPar* is changed to one on *everywhere*.

It is now trivial to write a function that adds prefixes generically by 'applying' *everywhere* to *addPrefix*.

```
genAddPrefix⟨a :: ∗⟩ :: a → a
genAddPrefix extends everywhere
  where everywhere      as genAddPrefix
        everywherePar as addPrefix.
```

Note that the *gmapT* 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 using type-indexed functions on the Id view. First class type-indexed functions are not supported in Generic Haskell. We emulate application of generic combinators to type-extension operators using defunctionalization techniques [16]. Defunctionalization is a standard technique to transform higher-order programs into first-order equivalents.

Because Generic Haskell lacks higher-order generic functions, these and other Scrap Your Boilerplate examples are better expressed using the standard view instead of the Boilerplate view. 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.

Hinze, Löh and Oliveira [17] define a generic boilerplate view using generalized algebraic data types. The view uses run-time representations of types and higher order functions, and is hence closer to the original Boilerplate approach. It follows that this view represents boilerplate functions more faithfully. Generic Haskell does not use run-time representations of types, so we cannot use the same approach.

Programs

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

Value declarations

$d ::= x = e$ function declaration

Patterns

$p ::= x$ variable pattern

$\mid (C\ \{p_i\}^i)$ constructor pattern

Expressions

$e ::= x$ variable

$\mid C$ data constructor

$\mid \lambda x\ .\ e$ abstraction

$\mid (e_1\ e_2)$ application

$\mid$ **case** e_0 **of** $\{p_i \to e_i\}_;^i$ case

$\mid$ (**fix** e) fixed point

$\mid$ **let** $\{d_i\}_;^i$ **in** e let

Fig. 1. Syntax of the expression language

4 Generic views, formally

The previous section shows why generic views are useful. This section formally defines generic views, and presents the formal definition of the standard view and the fixed-point view. The other views mentioned in the previous section can also be defined using the formalism introduced in this section.

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.

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

Generic programming extensions. To facilitate generic programming, the core language should be extended with parameterized type patterns and type-indexed functions with dependencies, and adapted with the facility to specify a view in the signature of a generic function.

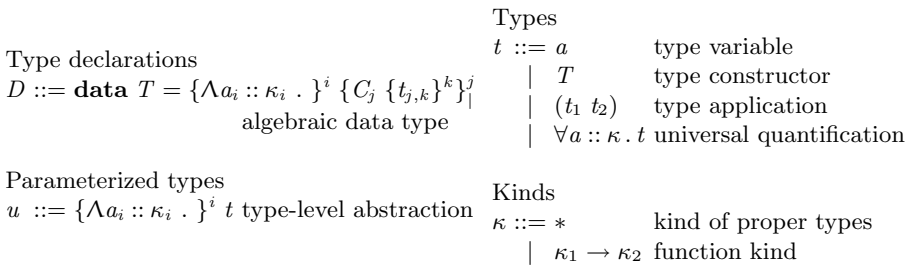

Fig. 2. Syntax of types and kinds

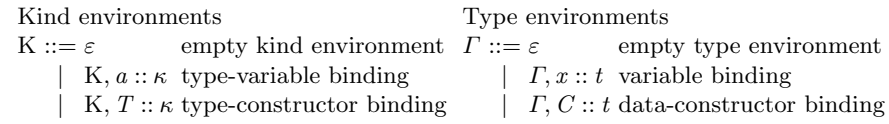

Fig. 3. Syntax of environments

Structure types in Haskell are declared as **type** synonyms. Type synonyms are not supported in the core language. Therefore, to describe structure types, the language contains *parameterized types*, which are essentially a nesting of type-level lambda abstractions around a type of kind $*$. Parameterized types are only used in view definitions, they cannot appear in a core-language program.

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 (EGH,[18]).

4.3 Definitions

Using the notion of parameterized types, we can formalize the observation that a view comprises 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 3.

Definition 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,

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

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, we require it to have three essential properties. First, the mapping from types to structure types should preserve kinds.

Definition 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 such that $\mathrm{K} \vdash T :: \kappa$, for which a structure type u can be derived,*

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

it follows that under kind environment K'

$$\mathrm{K}' \equiv \mathrm{K}, \mathrm{K}_{\mathcal{V}}\ \{, T_i :: \kappa_i\}^{i \in 1..n},$$

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 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 $\{\kappa_i \to\}^{i \in 1..\ell} *$, for which a structure type t can be derived,*

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

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

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

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

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

under environments K' as in Definition 2 and Γ' containing the view bindings $\Gamma_{\mathcal{V}}$ and the types of the constructors from D_0 and all D_i.

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

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

Fig. 4. Representation of data types in the standard view

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

Definition 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,*

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

such that e_{to} is the left inverse of e_{from} with respect to function composition:

$e_{\mathrm{to}}\ (e_{\mathrm{from}}\ v)$ *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.) Why do we want a generic view to have well-behaved conversions? Assume function *gid* is a generic identity function that is defined as a recursive function that traverses and copies the structure. To prove that this function is an identity, we have to ensure that the conversions that are applied during the traversal are well-behaved and do not modify the value.

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

Definition 5 (Validity). *A generic view is* valid *if it is kind preserving and generates well-typed, well-behaved conversions.*

We claim the validity of the standard, fixed point, balanced, list-like and boilerplate views.

The validity of a view has two important consequences. First, well-behaved conversions allow us to prove properties like the property for the generic identity function given after Definition 4. Second, let us recall from Section 5.2 that a generic function call using a new data type can be reduced to a call using the structural representation of the data type. This reduction is achieved by means of a wrapper that uses the structural representation and embedding-projection specified in the view. The theorem states that the generated wrapper is type-correct:

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

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

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

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

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

Fig. 5. Representation of constructors in the standard view

Theorem 1. *Let $\mathcal{V}$ be a view with*

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

For a type-indexed function x of arity $\langle r \mid s\rangle$, where all types γ_j in non-generic positions of x are of kind $*$, the declaration

let $\{bimap\langle\beta_i\rangle = EP\ e_{\mathrm{from}}\ e_{\mathrm{to}}\}_;^{i\in 1..r}$
$\quad \{bimap\langle\gamma_j\rangle = EP\ id\ id\}_;^{j\in 1..s}$
in *to* $bimap\langle \mathsf{base}\ (x\langle\{\beta_i\}_;^{i\in 1..r} \mid \{\gamma_j\}_;^{j\in 1..s}\rangle)\rangle\ x\langle u\rangle$

has the same type as $x\langle T\ \{\alpha_j\}^{j\in 1..n}\rangle$. Here base *$(f)$ returns the* base type *(see EGH) of f, i.e., the type specified for the generic function.*

We will use the above declaration as the translation (or specialization) of

$x\langle T\ \{\alpha_j\}^{j\in 1..n}$ **viewed** $\mathcal{V}\rangle$.

The proof of this theorem (which is very similar to the proof of Theorem 11.1 in EGH) uses the facts that a valid view preserves kinds, and has well-typed conversions. Well-behavedness of the conversions is not necessary for proving the theorem.

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

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

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

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

Fig. 6. Conversions for data types in the standard view

View types. The view types of the standard view are given by the 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

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

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 4 and 5.

Type declarations are handled by the rule in Figure 4. 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 6 and 7 and are of the forms

$$\begin{array}{ll}\mathcal{S}[\![D_0]\!]^{\mathrm{conv}} & \equiv e_{\mathrm{from}}; e_{\mathrm{to}} \\ \mathcal{S}[\![\{C_j\ \{t_{j,k}\}^k\}_{|}^j]\!]^{\mathrm{conv}} & \equiv \{p_{\mathrm{from},j}\}_{|}^j; \{p_{\mathrm{to},j}\}_{|}^j,\end{array}$$

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

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

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

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

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

Fig. 7. Conversions for constructors in the standard view

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 6 constructs a 'from' function that matches values of the original type against a list of patterns. If a value matches a certain pattern, a structure value is produced by using a complementary pattern; 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 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*.

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 8, which takes the form

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

The metafunction ptr produces 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.

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

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

Fig. 8. Pattern functors

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

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

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

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

is given in Figure 9. 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.

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

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

Fig. 9. Representation of data types in the fixed-point view

Generating conversions. Generating conversion functions for $\mathcal{F}$ is more involved. The algorithm is presented in Figures 10 and 11. It consists of judgements of the forms

$$\begin{array}{ll} \mathcal{F}[\![D_0]\!]^{\mathrm{conv}} & \equiv e_{\mathrm{from}};\, e_{\mathrm{to}} \\ \mathcal{F}[\![t]\!]^{\mathrm{conv}}_{T\ \{a_i\}^i;ep} & \equiv ep'. \end{array}$$

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 embedding projection expressions that convert constructor fields to or from the representation type. The generation of these embedding projections is driven by the original type $T\ \{a_i\}^i$, the type of the field t and the embedding-projection ep for the data type, with type EP $(T\ \{a_i\}^i)$ (Fix $(\mathsf{ptr}(T)\ \{a_i\}^i)$).

Each conversion function converts the fields using the appropriate components of the field embedding-projections. Because the embedding-projection may be recursively defined, Figure 10 uses the core language's recursive **let** construct.

In Figure 11, the conversion functions for a field are given by the application of *bimap* to the field type with occurrences of $T\ \{a_i\}^i$ abstracted. This application lifts the given embedding-projection to the abstracted field type in the fashion of Section 2.4. For instance, the embedding-projection of the data type Rose

data Rose a = *Rose* a [Rose a],

with type EP (Rose a) (Fix (ptr(Rose) a)), is lifted to types EP a a and EP [Rose a] [Fix (ptr(Rose) a)] for each of the two constructor fields respectively.

Note that the conversion functions are not directly defined by means of *bimap*. Rule conv-fix uses a special bracket notation $[\![\cdot]\!]$ to denote the translation of the *bimap* expression to the core language. In other words, the conversion functions do not include calls to the generic function *bimap*, but rather, the core language equivalents of those calls.

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

$$\frac{\begin{array}{c} \{\{\mathcal{F}[\![t_{j,k}]\!]^{\mathrm{conv}}_{T\ \{a_i\}^i;ep} \equiv e_{j,k}\}^k\}^j \\ e_{\mathrm{from}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{C_j\ \{x_{j,k}\}^k \rightarrow \mathit{In}\ (\mathsf{ptr}(C_j)\ \{\mathit{from}\ e_{j,k}\ x_{j,k}\}^k)\}^j \\ e_{\mathrm{to}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{In}\ (\mathsf{ptr}(C_j)\ \{x_{j,k}\}^k) \rightarrow C_j\ \{\mathit{to}\ e_{j,k}\ x_{j,k}\}^k\}^j \\ ep' \equiv \mathbf{let}\ ep = \mathrm{EP}\ e_{\mathrm{from}}\ e_{\mathrm{to}}\ \mathbf{in}\ ep \end{array}}{\mathcal{F}[\![\mathbf{data}\ T = \{\Lambda a_i :: *\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}^j_{\mid}]\!]^{\mathrm{conv}} \equiv \mathit{from}\ ep';\, \mathit{to}\ ep'}$$

Fig. 10. Conversions for data types in the fixed-point view

$$\boxed{\mathcal{F}[\![t]\!]^{\mathrm{conv}}_{T\ \{a_i\}^i;ep} \equiv ep'}$$

$$\frac{t' \equiv [\alpha \,/\, T\ \{a_i\}^i]\ t}{\mathcal{F}[\![t]\!]^{\mathrm{conv}}_{T\ \{a_i\}^i;ep} \equiv [\![\ \mathbf{let}\ bimap\langle\alpha\rangle = ep\ \mathbf{in}\ bimap\langle t'\rangle\ ]\!]} \quad \text{(conv-fix)}$$

Fig. 11. Conversions for fields in the fixed-point view

Mutual recursion, nested data types and higher-order kinded types. In PolyP, generic functions can only be applied to regular data types. This restriction excludes mutually recursive data types, nested data types and higher-order kinded data types from the class of data types to which a generic function can be applied. The fixed-point view has exactly the same restrictions, and is hence a faithful implementation of the view on data types used in PolyP. It is possible to adapt the fixed-point view such that higher-order kinded types can be handled, but in order to stay as close as possible to the view of data types of PolyP, we refrain from doing so.

Alternative solution. To circumvent the fixed-point view problems with higher-order kinded types, we consider an alternative view in which recursive calls in data types are modeled 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,

$$children\langle \mathrm{Rec}\ \varphi\ \rho\rangle\ (InR\ r\ \_) = \mathbf{let}\ collect\langle\alpha\rangle\ a = [\,a\,]\ \mathbf{in}\ collect\langle\varphi\ \alpha\rangle\ r$$
-- type incorrect,

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

$$\begin{aligned} &\mathit{children}\langle \mathrm{Rec}\ \varphi\ \rho\rangle\ (\mathit{InR}\ r\ \mathit{ep}) = \mathbf{let}\ \mathit{collect}\langle\alpha\rangle\ a = [\mathit{from}\ \mathit{ep}\ a] \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathit{collect}\langle\varphi\ \alpha\rangle\ r. \end{aligned}$$

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

$$\begin{aligned} &\mathbf{data}\ \mathrm{GRoseBase}\ f\ a\ r = \mathit{GBranchBase}\ a\ (f\ r) \\ &\mathbf{type}\ \mathrm{GRose}^{\circ}_{R}\ f\ a \qquad = \mathrm{Rec}\ (\mathrm{GRoseBase}\ f\ a)\ (\mathrm{GRose}\ f\ a) \\ &\mathit{conv}_{\mathrm{GRose},R} \quad :: \forall f\ a\,.\,\mathrm{EP}\ (\mathrm{GRose}\ f\ a)\ (\mathrm{GRose}^{\circ}_{R}\ f\ a) \\ &\mathit{conv}_{\mathrm{GRose},R} \quad = \mathit{EP}\ \mathit{from}_{\mathrm{GRose},R}\ \mathit{to}_{\mathrm{GRose},R}. \\ &\mathit{from}_{\mathrm{GRose},R}\ (\mathit{GBranch}\ a\ \mathbf{as}) \\ &\qquad\qquad = \mathit{InR}\ (\mathit{GBranchBase}\ a\ \mathbf{as})\ \mathit{conv}_{\mathrm{GRose},R} \\ &\mathit{to}_{\mathrm{GRose},R} \quad (\mathit{InR}\ (\mathit{GBranchBase}\ a\ \mathbf{as})\ \mathit{ep}) \\ &\qquad\qquad = \mathit{GBranch}\ a\ \mathbf{as} \end{aligned}$$

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.

5 Generic views in the Generic Haskell compiler

The latest version of the Generic Haskell compiler that implements views can be downloaded via svn: `https://svn.cs.uu.nl:12443/repos/Generic-Haskell/branches/GenericViews`. We have implemented (an extension of) the standard view, the fixed-point view and the list-like sums and products view. The next release of Generic Haskell will come with all the views mentioned in this paper: in addition to the views already implemented, the balanced sums and products view, and the identity and boilerplate views. In the previous version of the Generic Haskell compiler [8] 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.

Since there is no reason to assume that the six views given in this paper are the only useful views, we have considered developing a special-purpose language for specifying views in user programs. We have decided not to do this for three reasons:

- We expect that these views suffice for most purposes and users.
- A generic view consists of a set of view types, a function that generates structure types, and a function that generates conversion functions, and it

follows that such a special-purpose language for specifying views would be a complete programming language in itself.
– To add a new view to Generic Haskell, the compiler has to be modified. Although this might sound scary, in practice it is rather simple.

The next section describes how a view is added to the Generic Haskell compiler.

5.1 Adding a view to the Generic Haskell compiler

A new view is added to the Generic Haskell compiler by implementing a module that contains a view declaration with the following type:

$$(\text{Name}, \text{TDecl} \to \text{Maybe}\ (\text{LamType}, [\text{TDecl}], \text{Expr}, \text{Expr}, [\text{TDecl}])).$$

A view consists of a name, and a function that can be called on the abstract representation of a type synonym or a data type (a TDecl) to produce a parameterized structure representation type (a LamType), supporting type declarations (first [TDecl]) and an embedding-projection pair (two Expr's). Views that apply to a subset of the Haskell data types can be implemented by returning *Nothing* on data type definitions that are outside of the view domain. Note that the result of the view-generating function directly corresponds to the maps $\mathcal{V}[\![\cdot]\!]^{\text{str}}$ and $\mathcal{V}[\![\cdot]\!]^{\text{conv}}$. 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 need for the second list of type declarations is better explained with an example. Consider the application of the *children* function (Section 3) to the Tree data type. This function definition uses the fixed-point view structure type of Tree. That is, the type Fix applied to the supporting type TreeBase, which is the base functor of Tree. This structure type is not yet enough. The definition of *children* applies *collect* to the base functor of the data type. Because *collect* is defined on the standard view, we need to generate a standard structure type for TreeBase, a pair of standard embedding projections and supporting declarations. In short, if we need a fixed-point view on Tree, we also need a standard view on TreeBase. This is achieved by returning TreeBase in the second declaration list when generating the view components for Tree. This list, which is a subset of the list of supporting declarations, is recursively processed by view-generating functions. The implementation keeps track of additional information to determine which view-generating functions should be called on these declarations, and to avoid non-termination in certain cases.

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.

A view implementor has to deal with some additional implementation details that slightly complicate views, but that are not of direct concern for this paper. The interested reader can find more implementation information in the source distribution of Generic Haskell extended with views, in particular in the file `/src/views/README` and the view modules in the same directory.

5.2 Specialization

The specialization mechanism is independent of the actual view, see Theorem 1. 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 has changed in the implementation of specialization within the Generic Haskell compiler is that all the references to structural representation types and embedding-projection pairs point to the view that is specified for the function in question.

6 Conclusions, and related work

We have shown that generic views on data types can make generic functions both easier to write and more efficient. Generic views add expressiveness to a generic programming language. 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. Using our approach to views we can express many different approaches to generic programming in a single framework. Our framework does have a limitation though: a structure type is created from a single data type declaration at a time. This poses no problems for views that transform only the top level representation of a data type, consider for instance type List° in Section 2.3. Views that deeply transform representations, however, face limitations: the fixed-point view, for example, transforms the recursive occurrences of a single recursive data type, and hence it cannot handle mutually recursive ones. This is not a fundamental problem: we could have adapted the framework to allow for this. However, since we know of no existing approach to generic programming that would need this extra complexity we have refrained from doing so.

The name "generic view" is derived from Wadler's proposal to introduce views in (a predecessor of) Haskell [12]. 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. Moreover, the representation types need

not be new data types, but can be built from existing data types. Wadler's views have the 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 to be added to a generic programming system, such as the Generic Haskell compiler, following the guidelines described in the previous sections. However, we expect that the views we describe in this paper are sufficient for most purposes and users, and we do not assume a user will frequently want to add a new view to the Generic Haskell compiler.

Both views and generic views require 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 the original type and the view type (structure type in our framework), 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 [19–21]. 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 (structure type). Views have also been proposed in the context of XML and databases [22, 23]. Generic views as proposed in [24] 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 [25, 26]. 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 a number of anonymous referees and Daan Leijen for several helpful comments. This research has partly been funded by the Netherlands Organization for Scientific Research (NWO) (project Generic Haskell, nr. 612.069.000).

References

1. Jansson, P., Jeuring, J.: PolyP — a polytypic programming language extension. In: Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (1997) 470–482
2. Hinze, R.: Polytypic values possess polykinded types. Science of Computer Programming **43**(2-3) (2002) 129–159
3. Lämmel, R., Jones, S.P.: Scrap your boilerplate: a practical approach to generic programming. ACM SIGPLAN Notices **38**(3) (2003) 26–37 Proceedings ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

4. Jay, C.B.: Distinguishing data structures and functions: the constructor calculus and functorial types. In Abramsky, S., ed.: Typed Lambda Calculi and Applications: 5th International Conference TLCA 2001. Volume 2044 of LNCS., Springer-Verlag (2001) 217–239
5. Hoogendijk, P., de Moor, O.: Container types categorically. Journal of Functional Programming **10**(2) (2000) 191–225
6. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes, and barbed wire. In Hughes, J., ed.: Functional Programming Languages and Computer Architecture, FPCA 1991. Volume 523 of LNCS., Springer-Verlag (1991) 124–144
7. Jansson, P., Jeuring, J.: A framework for polytypic programming on terms, with an application to rewriting. In Jeuring, J., ed.: Workshop on Generic Programming 2000, Ponte de Lima, Portugal, July 2000. (2000) 33–45 Utrecht Technical Report UU-CS-2000-19.
8. Löh, A., Jeuring, J., Clarke, D., Hinze, R., Rodriguez, A., de Wit, J.: 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)
9. Löh, A.: Exploring Generic Haskell. PhD thesis, Utrecht University (September 2004)
10. Lämmel, R., Peyton Jones, S.: Scrap more boilerplate: reflection, zips, and generalised casts. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, ACM Press (2004) 244–255
11. Hinze, R., Jeuring, J.: Generic Haskell: applications. In: Generic Programming, Advanced Lectures. Volume 2793 of LNCS., Springer-Verlag (2003) 57–97
12. Wadler, P.: 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)
13. Hinze, R., Jeuring, J.: Generic Haskell: practice and theory. In: Generic Programming, Advanced Lectures. Volume 2793 of LNCS., Springer-Verlag (2003) 1–56
14. Malcolm, G.: Data structures and program transformation. Science of Computer Programming **14** (1990) 255–279
15. Schrage, M.: Proxima, a presentation-oriented editor for structured documents. PhD thesis, Utrecht University (October 2004)
16. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: ACM '72: Proceedings of the ACM annual conference, New York, NY, USA, ACM Press (1972) 717–740
17. Hinze, R., Löh, A., Oliveira, B.C.d.S.: "Scrap Your Boilerplate" Reloaded. In: Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006). Volume 3945 of LNCS., Springer-Verlag (2006)
18. Holdermans, S.: Generic views. Master's thesis, Institute of Information and Computing Sciences, Utrecht University (2005)
19. Burton, F.W., Cameron, R.D.: Pattern matching with abstract data types. Journal of Functional Programming **3**(2) (1993) 117–190
20. Okasaki, C.: Views for Standard ML. In: SIGPLAN Workshop on ML. (1998) 14–23
21. Burton, F.W., Meijer, E., Sansom, P., Thompson, S., Wadler, P.: Views: an extension to Haskell pattern matching. Available from `http://www.haskell.org/development/views.html` (1996)
22. Abiteboul, S.: On views and XML. In: Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM Press (1999) 1–9

23. Ohori, A., Tajima, K.: A polymorphic calculus for views and object sharing. In: Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. (1994) 255–266
24. Souza dos Santos, C., Abiteboul, S., Delobel, C.: Virtual schemas and bases. In: Proceedings of the International Conference on Extensive Data Base Technology (EDBT'94), Cambridge, UK, Springer-Verlag (1994) 81–94
25. Altenkirch, T., McBride, C.: Generic programming within dependently typed programming. In Gibbons, J., Jeuring, J., eds.: 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, Kluwer Academic Publishers (2003) 1–20
26. Benke, M., Dybjer, P., Jansson, P.: Universes for generic programs and proofs in dependent type theory. Nordic Journal of Computing **10**(4) (2003) 265–289