Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Science of Computer Programming 51 (2004) 117–151
Science of Computer Programming
www.elsevier.com/locate/scico

Type-indexed data types

Ralf Hinze[a,*], Johan Jeuring[b,c], Andres Löh[b]

[a] *Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn, Germany*
[b] *Institute of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands*
[c] *Open University, Heerlen, The Netherlands*

Received 20 January 2003; received in revised form 18 July 2003; accepted 21 July 2003

Abstract

A polytypic function is a function that can be instantiated on many data types to obtain data type specific functionality. Examples of polytypic functions are the functions that can be derived in Haskell, such as *show*, *read*, and '=='. More advanced examples are functions for digital searching, pattern matching, unification, rewriting, and structure editing. For each of these problems, we not only have to define polytypic functionality, but also a *type-indexed data type*: a data type that is constructed in a generic way from an argument data type. For example, in the case of digital searching we have to define a search tree type by induction on the structure of the type of search keys. This paper shows how to define type-indexed data types, discusses several examples of type-indexed data types, and shows how to specialize type-indexed data types. The approach has been implemented in *Generic Haskell*, a generic programming extension of the functional language Haskell.

Keywords: Polytypic programming; Generic Haskell; Digital searching

1. Introduction

A polytypic (or generic, or type-indexed) function is a function that can be instantiated on many data types to obtain data type specific functionality. Examples of polytypic functions are the functions that can be derived in Haskell [2], such as *show*, *read*, and '=='. See [3] for an introduction to polytypic programming.

* Corresponding author.
E-mail addresses: ralf@informatik.uni-bonn.de (R. Hinze), johanj@cs.uu.nl (J. Jeuring), andres@cs.uu.nl (A. Löh).
URLs: http://www.informatik.uni-bonn.de/~ralf/, http://www.cs.uu.nl/andres, http://www.cs.uu.nl/johanj

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2003.07.001

More advanced examples of polytypic functions are functions for digital searching [15], pattern matching [29], unification [26,6], rewriting [27], and structure editing [13]. For each of these problems, we not only have to define polytypic functionality, but also a *type-indexed data type*: a data type that is constructed in a generic way from an argument data type. For instance, in the case of digital searching we have to define a search tree type by induction on the structure of the type of search keys. Since current strongly typed programming languages do not support type-indexed data types, the examples that appear in the literature are either implemented in an ad hoc fashion [26], or not implemented at all [15].

This paper shows how to define a type-indexed data type, discusses several examples of type-indexed data types, and shows how to specialize a type-indexed data type. The specialization is illustrated with example translations to Haskell. The approach has been implemented in *Generic Haskell*, a generic programming extension of the functional language Haskell. Generic Haskell can be obtained from `http://www.generic-haskell.org/`. This paper is a revised version of [21].

Example 1 (Digital searching). A digital search tree or trie is a search tree scheme that employs the structure of search keys to organize information. Searching is useful for various data types, so we would like to allow for keys and information of any data type. This means that we have to construct a new kind of trie for each key type. For example, consider the data type String defined by [1]

$$\textbf{data}\ \mathsf{String} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{Char}\ \mathsf{String}.$$

We can represent string-indexed tries with associated values of type v as follows:

$$\begin{aligned}\textbf{data}\ \mathsf{FMap\_String}\ \mathsf{v} = \mathit{Trie\_String}\ &(\mathsf{Maybe}\ \mathsf{v}) \\ &(\mathsf{FMapChar}\ (\mathsf{FMap\_String}\ \mathsf{v})),\end{aligned}$$

where FMap stands for 'finite map'. Such a trie for strings would typically be used for an index on texts. The first component of the constructor *Trie_String* contains the value associated with *Nil*. The second component of *Trie_String* is derived from the constructor $\mathit{Cons} :: \mathsf{Char} \to \mathsf{String} \to \mathsf{String}$. We assume that a suitable data structure, FMapChar, and an associated look-up function $\mathit{lookupChar} :: \forall \mathsf{v}\,.\,\mathsf{Char} \to \mathsf{FMapChar}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}$ for characters are predefined. We use the following naming convention: names such as FMap_String where an underscore separates the name of two types are used for instances of type-indexed entities. The goal of the paper is to describe how to generate such types automatically from a generic definition. Compound names (such as FMapChar) are used when we assume that a type or function is predefined or defined by the user.

[1] The examples are given in Haskell [2]. Deviating from Haskell, universal quantification of types is always made explicit by means of ∀'s in the type.

Given the definitions of String and FMap_String, we can define a look-up function for strings as follows:

$$
\begin{aligned}
&\mathit{lookup\_String} :: \forall \mathsf{v}\,.\,\mathsf{String} \to \mathsf{FMap\_String\ v} \to \mathsf{Maybe\ v}\\
&\mathit{lookup\_String}\ \mathit{Nil}\ \ \ \ \ \ \ \ (\mathit{Trie\_String}\ tn\ tc) = tn\\
&\mathit{lookup\_String}\ (\mathit{Cons}\ c\ s)\ (\mathit{Trie\_String}\ tn\ tc)\\
&\quad = (\mathit{lookupChar}\ c \diamond \mathit{lookup\_String}\ s)\ tc.
\end{aligned}
$$

To look up a non-empty string, *Cons c s*, we look up c in the FMapChar obtaining a trie, which is then recursively searched for s. Since the look-up functions have result type Maybe v, we use the reverse monadic composition of the Maybe monad, called '$\diamond$', to compose *lookup_String* and *lookup_Char*.

$$
\begin{aligned}
&(\diamond)\ \ \ \ \ \ :: \forall \mathsf{a\ b\ c}\,.\,(\mathsf{a} \to \mathsf{Maybe\ b}) \to (\mathsf{b} \to \mathsf{Maybe\ c}) \to \mathsf{a} \to \mathsf{Maybe\ c}\\
&(f \diamond g)\ a = \mathbf{case}\ f\ a\ \mathbf{of}\ \{\mathit{Nothing} \to \mathit{Nothing}; \mathit{Just}\ b \to g\ b\}
\end{aligned}
$$

Consider now the data type Bush of binary trees with characters in the leaves:

$$\mathbf{data}\ \mathsf{Bush} = \mathit{Leaf}\ \mathsf{Char} \mid \mathit{Fork}\ \mathsf{Bush\ Bush}.$$

Bush-indexed tries can be represented by the following data type:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{FMap\_Bush\ v} = \mathit{Trie\_Bush}\ (\mathsf{FMapChar\ v})\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{FMap\_Bush}\ (\mathsf{FMap\_Bush\ v})).
\end{aligned}
$$

Again, we have two components, one to store values constructed by *Leaf*, and one for values constructed by *Fork*. The corresponding look-up function is given by

$$
\begin{aligned}
&\mathit{lookup\_Bush} :: \forall \mathsf{v}\,.\,\mathsf{Bush} \to \mathsf{FMap\_Bush\ v} \to \mathsf{Maybe\ v}\\
&\mathit{lookup\_Bush}\ (\mathit{Leaf}\ c)\ \ \ \ (\mathit{Trie\_Bush}\ tl\ tf) = \mathit{lookupChar}\ c\ tl\\
&\mathit{lookup\_Bush}\ (\mathit{Fork}\ bl\ br)\ (\mathit{Trie\_Bush}\ tl\ tf)\\
&\quad = (\mathit{lookup\_Bush}\ bl \diamond \mathit{lookup\_Bush}\ br)\ tf.
\end{aligned}
$$

One can easily recognize that not only the look-up functions, but also the data types for the tries are instances of an underlying generic pattern. In the following section, we will show how to define a trie and associated functions generically for arbitrary data types. The material is taken from Hinze [15], and it is repeated here because it serves as a nice and simple example of a type-indexed data type.

Example 2 (Pattern matching). The polytypic functions for the maximum segment sum problem [4] and pattern matching [29] use labelled data types. These labelled data types, introduced in [4], can be used to store at each node the subtree rooted at that node, or a set of patterns (trees with variables) matching at a subtree, etc. For example, the data type of labelled bushes is defined by

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Lab\_Bush\ m} = \mathit{Label\_Leaf}\ \mathsf{Char\ m}\\
&\qquad\qquad\qquad\qquad \mid \mathit{Label\_Fork}\ (\mathsf{Lab\_Bush\ m})\ (\mathsf{Lab\_Bush\ m})\ \mathsf{m}.
\end{aligned}
$$

It can be constructed from the Bush data type by extending each constructor with an additional field to store the label. In the following section we show how to define such a labelled data type generically, and how this data type is used in a (specification of a) generic pattern matching program.

Example 3 (Zipper). The zipper [22,23] is a data structure that is used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up, or down the tree. For example, the zipper corresponding to the data type Bush, called Loc_Bush, is defined by

$$\begin{aligned}
&\textbf{type}\ \mathsf{Loc_Bush} &&= (\mathsf{Bush}, \mathsf{Context_Bush})\\
&\textbf{data}\ \mathsf{Context_Bush} &&= \mathit{Top}\\
&&&\mid \mathit{ForkL}\ \mathsf{Context_Bush}\ \mathsf{Bush}\\
&&&\mid \mathit{ForkR}\ \mathsf{Bush}\ \mathsf{Context_Bush}.
\end{aligned}$$

Using the type of locations we can efficiently navigate through a tree. For example:

$$\begin{aligned}
&\mathit{down_Bush} &&:: \mathsf{Loc_Bush} \to \mathsf{Loc_Bush}\\
&\mathit{down_Bush}\ (\mathit{Leaf}\ a, c) &&= (\mathit{Leaf}\ a, c)\\
&\mathit{down_Bush}\ (\mathit{Fork}\ tl\ tr, c) &&= (tl, \mathit{ForkL}\ c\ tr)\\
&\mathit{right_Bush} &&:: \mathsf{Loc_Bush} \to \mathsf{Loc_Bush}\\
&\mathit{right_Bush}\ (tl, \mathit{ForkL}\ c\ tr) &&= (tr, \mathit{ForkR}\ tl\ c)\\
&\mathit{right_Bush}\ m &&= m.
\end{aligned}$$

The navigation function *down_Bush* moves the focus of attention to the *leftmost* subtree of the current node; *right_Bush* moves the focus to its right sibling.

Huet [22] defines the zipper data structure for rose trees and for the data type Bush, and gives the generic construction in words. In Section 5 we describe the zipper in more detail and show how to define a zipper for an arbitrary data type.

Other examples. Besides these three examples, a number of other examples of type-indexed data types have appeared in the literature [5,12,14,40]. We expect that type-indexed data types will also be useful for generic DTD transformations [32]. Generally, we believe that type-indexed data types are almost as important as type-indexed functions.

Background and related work. There is little related work on type-indexed data types. Type-indexed functions [4,11,25,33,36] were introduced more than a decade ago. There are several other approaches to type-indexed functions, see [10,28,44], but none of them mentions user-defined type-indexed data types (Yang does mention value-indexed types, usually called dependent types).

Type-indexed data types, however, appear in the work on intensional type analysis [8,9,14,39,41]. Intensional type analysis is used in typed intermediate languages in compilers for polymorphic languages, among others to be able to optimize code for polymorphic functions. This work differs from our work in several aspects:

- typed intermediate languages are expressive, but rather complex languages not intended for programmers but for compiler writers;

- since Generic Haskell is built on top of Haskell, there is the problem of how to combine user-defined functions and data types with type-indexed functions and data types. This problem does not appear in typed intermediate languages;
- typed intermediate languages interpret (a representation of a) type argument at run-time, whereas the specialization technique described in this paper does not require passing around (representations of) type arguments;
- originally, typed intermediate languages were restricted to data types of kind ★. Building upon Hinze's work, Weirich recently generalized intensional type analysis to higher-order kinded types [42]. However, higher-order intensional type analysis does not support type-indexed data types.

Organization. The rest of this paper is organized as follows. We will show how to define type-indexed data types in Section 2 using Hinze's approach to polytypic programming [16,17]. Section 3 illustrates the process of specialization by means of example. Section 4 shows that type-indexed data types possess kind-indexed kinds, and provides theoretical background for the specialization of type-indexed data types and functions with arguments of type-indexed data types. Section 5 provides the details of the zipper example. Finally, Section 6 summarizes the main points and concludes.

2. Defining type-indexed data types

This section shows how to define type-indexed data types. Section 2.1 briefly reviews the concepts of polytypic programming necessary for defining type-indexed data types. The subsequent sections define type-indexed data types for the problems described in the introduction. We assume a basic familiarity with Haskell's type system and in particular with the concept of kinds [35]. For a more thorough treatment the reader is referred to Hinze's work [16,17].

2.1. Type-indexed definitions

The central idea of polytypic programming (also called type-indexed or generic programming) is to provide the programmer with the ability to define a function by induction on the structure of types. Since Haskell's type language is rather involved—we have mutually recursive types, parameterized types, nested types, and type constructors of higher-order kinds—this sounds like a hard nut to crack. Fortunately, one can show that a polytypic function is uniquely defined by giving cases for a very limited set of types and type constructors. For instance, to define a generic function on types of kind ★, we need cases for the unit type 1, the sum type constructor $+$, and the product type constructor $\times$. These three types are required for modelling Haskell's **data** construct that introduces a sum of products. We treat 1, $+$, and $\times$ as if they were given by the following **data** declarations:

$$\begin{aligned}
&\textbf{data}\ 1 &&= ()\\
&\textbf{data}\ a + b &&= \mathit{Inl}\ a \mid \mathit{Inr}\ b\\
&\textbf{data}\ a \times b &&= (a, b).
\end{aligned}$$

Additionally, if we want our generic functions to work on primitive types such as Char and Float, which are not defined by means of a Haskell **data** statement, we need to include additional cases for such types. For the purposes of the paper, we will assume Char to be the only primitive type.

Now, a polytypic function is given by a definition that is inductive on 1, Char, $+$, and $\times$. As an example, here is the polytypic equality function. For emphasis, the type index is enclosed in angle brackets.

$$
\begin{array}{lllll}
equal\langle \mathsf{t} :: \bigstar \rangle & & & :: \mathsf{t} \to \mathsf{t} \to \mathsf{Bool} \\
equal\langle 1 \rangle & () & () & = True \\
equal\langle \mathsf{Char} \rangle & c_1 & c_2 & = equalChar\ c_1\ c_2 \\
equal\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle & (Inl\ a_1) & (Inl\ a_2) & = equal\langle \mathsf{t}_1 \rangle\ a_1\ a_2 \\
equal\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle & (Inl\ a_1) & (Inr\ b_2) & = False \\
equal\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle & (Inr\ b_1) & (Inl\ a_2) & = False \\
equal\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle & (Inr\ b_1) & (Inr\ b_2) & = equal\langle \mathsf{t}_2 \rangle\ b_1\ b_2 \\
equal\langle \mathsf{t}_1 \times \mathsf{t}_2 \rangle & (a_1, b_1) & (a_2, b_2) & = equal\langle \mathsf{t}_1 \rangle\ a_1\ a_2 \wedge equal\langle \mathsf{t}_2 \rangle\ b_1\ b_2
\end{array}
$$

This simple definition contains all ingredients needed to specialize *equal* for arbitrary data types. Note that the definition does not mention type abstraction, type application, and fixed points. Instances of polytypic functions on types with these constructions can be generated automatically from just the cases given above. For example, if we used *equal* at the data type Bush, the generated specialization would behave exactly as the following hand-written code.

$$
\begin{array}{lll}
equal\_Bush & & :: \mathsf{Bush} \to \mathsf{Bush} \to \mathsf{Bool} \\
equal\_Bush\ (Leaf\ c_1) & (Leaf\ c_2) & = equalChar\ c_1\ c_2 \\
equal\_Bush\ (Fork\ m_1\ r_1) & (Fork\ m_2\ r_2) & = \\
\quad equal\_Bush\ m_1\ m_2 \wedge equal\_Bush\ r_1\ r_2 & & \\
equal\_Bush\ \_ & \_ & = False
\end{array}
$$

We will discuss the generation of specializations for generic functions in detail in Section 4.

Sometimes we want to be able to refer to the name of a constructor. To this end, we add one more special type constructor for which a case can be defined in a generic function: c **of** t, where c is a value, and t is a type of kind $\bigstar$. The value c represents the name of a constructor. If the 'c **of** t' case is omitted in the definition of a polytypic function *poly*, as in function *equal*, we assume that $poly\langle c\ \mathbf{of}\ \mathsf{t} \rangle = poly\langle \mathsf{t} \rangle$. For the purposes of this paper, we assume that c is of type String.

As an example for the use of constructor names in a generic function, we give a very simple variant of the polytypic *show* function, that computes a textual representation of any value:

$$
\begin{array}{lll}
show\langle \mathsf{t} :: \bigstar \rangle & & :: \mathsf{t} \to \mathsf{String} \\
show\langle 1 \rangle & () & = "" \\
show\langle \mathsf{Char} \rangle & c & = showChar\ c
\end{array}
$$

$$
\begin{aligned}
&\mathit{show}\langle \mathsf{t}_1 + \mathsf{t}_2\rangle\ (\mathit{Inl}\ a) = \mathit{show}\langle \mathsf{t}_1\rangle\ a\\
&\mathit{show}\langle \mathsf{t}_1 + \mathsf{t}_2\rangle\ (\mathit{Inr}\ b) = \mathit{show}\langle \mathsf{t}_2\rangle\ b\\
&\mathit{show}\langle \mathsf{t}_1 \times \mathsf{t}_2\rangle\ (a,b) \quad = \mathit{show}\langle \mathsf{t}_1\rangle\ a +\!\!+ \texttt{" "} +\!\!+ \mathit{show}\langle \mathsf{t}_2\rangle\ b\\
&\mathit{show}\langle c\ \mathbf{of}\ \mathsf{t}\rangle\ \ t \qquad\quad = \texttt{"("} +\!\!+ c +\!\!+ \texttt{" "} +\!\!+ \mathit{show}\langle \mathsf{t}\rangle\ t +\!\!+ \texttt{")"}.
\end{aligned}
$$

Here, $(+\!\!+) :: \mathsf{String} \to \mathsf{String} \to \mathsf{String}$ stands for string concatenation.

Whenever we make use of a polytypic function for a specific data type, we implicitly view this data type as if it were constructed by the unit type, sum, product, and the marker for constructors. For example, the Haskell data type of natural numbers

$$\mathbf{data}\ \mathsf{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathsf{Nat}$$

is represented by

$$\mathsf{Nat} = \texttt{"Zero"}\ \mathbf{of}\ 1 + \texttt{"Succ"}\ \mathbf{of}\ \mathsf{Nat},$$

and the data type of bushes

$$\mathbf{data}\ \mathsf{Bush} = \mathit{Leaf}\ \mathsf{Char} \mid \mathit{Fork}\ \mathsf{Bush}\ \mathsf{Bush}$$

is viewed as

$$\mathsf{Bush} = \texttt{"Leaf"}\ \mathbf{of}\ \mathsf{Char} + \texttt{"Fork"}\ \mathbf{of}\ \mathsf{Bush} \times \mathsf{Bush}.$$

The details of the type representation are given in Section 3.

The functions *equal* and *show* are indexed by a type of kind $\star$. A polytypic function may also be indexed by type constructors of kind $\star \to \star$ (and, of course, by type constructors of other kinds, but these are not needed in the sequel). We need slightly different base cases for generic functions operating on types of kind $\star \to \star$:

$$
\begin{aligned}
&\mathsf{Id} &&= \Lambda \mathsf{a}\,.\,\mathsf{a}\\
&\mathsf{K\ t} &&= \Lambda \mathsf{a}\,.\,\mathsf{t}\\
&\mathsf{f}_1 + \mathsf{f}_2 &&= \Lambda \mathsf{a}\,.\,\mathsf{f}_1\ \mathsf{a} + \mathsf{f}_2\ \mathsf{a}\\
&\mathsf{f}_1 \times \mathsf{f}_2 &&= \Lambda \mathsf{a}\,.\,\mathsf{f}_1\ \mathsf{a} \times \mathsf{f}_2\ \mathsf{a}\\
&c\ \mathbf{of}\ \mathsf{f} &&= \Lambda \mathsf{a}\,.\,c\ \mathbf{of}\ \mathsf{f\ a}.
\end{aligned}
$$

Here, $\Lambda \mathsf{a}\,.\,\mathsf{t}$ denotes abstraction on the type level. We have the constant functor K, which lifts a type of kind $\star$ to kind $\star \to \star$. We will need $\mathsf{K\ 1}$ as well as $\mathsf{K\ Char}$ (or more general, $\mathsf{K\ t}$ for all primitive types). We overload $+$, $\times$ and c **of** to be the lifted versions of their previously defined counterparts. The only new type index in this set of indices of kind $\star \to \star$ is the identity functor Id. Hinze [16] shows that these types are the normal forms of types of kind $\star \to \star$.

A well-known example of a $(\star \to \star)$-indexed function is the mapping function, which applies a given function to each element of type a in a given structure of type $\mathsf{f\ a}$:

$$
\begin{aligned}
&\mathit{map}\langle \mathsf{f} :: \star \to \star\rangle &&:: \forall \mathsf{a\ b}\,.\,(\mathsf{a} \to \mathsf{b}) \to (\mathsf{f\ a} \to \mathsf{f\ b})\\
&\mathit{map}\langle \mathsf{Id}\rangle\ \ m\ a &&= m\ a
\end{aligned}
$$

$$
\begin{aligned}
&map\langle \mathsf{K}\ 1\rangle \quad m\ c &&= c\\
&map\langle \mathsf{K}\ \mathsf{Char}\rangle\ m\ c &&= c\\
&map\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ m\ (Inl\ f) &&= Inl\ (map\langle \mathsf{f}_1\rangle\ m\ f)\\
&map\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ m\ (Inr\ g) &&= Inr\ (map\langle \mathsf{f}_2\rangle\ m\ g)\\
&map\langle \mathsf{f}_1 \times \mathsf{f}_2\rangle\ m\ (f,g) &&= (map\langle \mathsf{f}_1\rangle\ m\ f, map\langle \mathsf{f}_2\rangle\ m\ g).
\end{aligned}
$$

Using *map* we can, for instance, define generic versions of cata- and anamorphisms [37]. To this end we assume that data types are given as fixed points of so-called pattern functors. In Haskell the fixed point combinator can be defined as follows:

$$\textbf{newtype}\ \mathsf{Fix}\ \mathsf{f} = In\{out :: \mathsf{f}\ (\mathsf{Fix}\ \mathsf{f})\}.$$

It follows that the constructor *In* and the 'destructor' *out* have the following types:

$$
\begin{aligned}
&In &&:: \forall \mathsf{f}.\mathsf{f}\ (\mathsf{Fix}\ \mathsf{f}) \to \mathsf{Fix}\ \mathsf{f}\\
&out &&:: \forall \mathsf{f}.\mathsf{Fix}\ \mathsf{f} \to \mathsf{f}\ (\mathsf{Fix}\ \mathsf{f}).
\end{aligned}
$$

For example, we could have defined the type of bushes by $\mathsf{Bush} = \mathsf{Fix}\ \mathsf{BushF}$, where

$$\textbf{data}\ \mathsf{BushF}\ \mathsf{r} = LeafF\ \mathsf{Char} \mid ForkF\ \mathsf{r}\ \mathsf{r}.$$

It is easy to convert between this data type defined as a fixed point and the original type definition of bushes.

Cata- and anamorphisms are now given by

$$
\begin{aligned}
&cata\langle \mathsf{f} :: \bigstar \to \bigstar\rangle &&:: \forall \mathsf{a}.(\mathsf{f}\ \mathsf{a} \to \mathsf{a}) \to (\mathsf{Fix}\ \mathsf{f} \to \mathsf{a})\\
&cata\langle \mathsf{f}\rangle\ \varphi &&= \varphi \cdot map\langle \mathsf{f}\rangle\ (cata\langle \mathsf{f}\rangle\ \varphi) \cdot out\\
&ana\langle \mathsf{f} :: \bigstar \to \bigstar\rangle &&:: \forall \mathsf{a}.(\mathsf{a} \to \mathsf{f}\ \mathsf{a}) \to (\mathsf{a} \to \mathsf{Fix}\ \mathsf{f})\\
&ana\langle \mathsf{f}\rangle\ \psi &&= In \cdot map\langle \mathsf{f}\rangle\ (ana\langle \mathsf{f}\rangle\ \psi) \cdot \psi.
\end{aligned}
$$

Note that both functions are parameterized by the pattern functor f rather than by the fixed point Fix f. For example, the catamorphism on the functor of bushes, BushF, would be defined by

$$
\begin{aligned}
&cata\langle \mathsf{BushF}\rangle &&:: \forall \mathsf{a}.(\mathsf{BushF}\ \mathsf{a} \to \mathsf{a}) \to (\mathsf{Fix}\ \mathsf{BushF} \to \mathsf{a})\\
&cata\langle \mathsf{BushF}\rangle\ \varphi &&= \varphi \cdot map\langle \mathsf{BushF}\rangle\ (cata\langle \mathsf{BushF}\rangle\ \varphi) \cdot out,
\end{aligned}
$$

where $map\langle \mathsf{BushF}\rangle$ is an instance of the generic function *map*, defined above, that is equivalent to

$$
\begin{aligned}
&map\_BushF &&:: \forall \mathsf{a}\ \mathsf{b}.(\mathsf{a} \to \mathsf{b}) \to (\mathsf{BushF}\ \mathsf{a} \to \mathsf{BushF}\ \mathsf{b})\\
&map\_BushF\ m\ (LeafF\ c) &&= LeafF\ c\\
&map\_BushF\ m\ (ForkF\ bl\ br) &&= ForkF\ (m\ bl)\ (m\ br).
\end{aligned}
$$

Both *cata* and *ana* are so-called *generic abstractions*, i.e. generic functions that are not defined by induction on base types, but in terms of other generic functions. Generic Haskell supports generic abstractions [7].

2.2. Tries

Tries are based on the following isomorphisms, also known as the laws of exponentials:

$$\begin{array}{rcl} 1 \to_{\mathrm{fin}} \mathsf{v} & \cong & \mathsf{v} \\ (\mathsf{t}_1 + \mathsf{t}_2) \to_{\mathrm{fin}} \mathsf{v} & \cong & (\mathsf{t}_1 \to_{\mathrm{fin}} \mathsf{v}) \times (\mathsf{t}_2 \to_{\mathrm{fin}} \mathsf{v}) \\ (\mathsf{t}_1 \times \mathsf{t}_2) \to_{\mathrm{fin}} \mathsf{v} & \cong & \mathsf{t}_1 \to_{\mathrm{fin}} (\mathsf{t}_2 \to_{\mathrm{fin}} \mathsf{v}). \end{array}$$

There are more laws for exponentials, but these are the ones we need in our definition of tries. Here, $\mathsf{t} \to_{\mathrm{fin}} \mathsf{v}$ denotes the type of finite maps from t to v. Using the isomorphisms above as defining equations, we can give a type-indexed definition for the data type $\mathsf{FMap}\langle\mathsf{t}\rangle\ \mathsf{v}$ of finite maps from t to v, which generalizes FMap_String from the introduction to arbitrary data types. This is our first example of a type-indexed data type.

$$\begin{array}{lll} \mathsf{FMap}\langle \mathsf{t} :: \bigstar \rangle & :: \bigstar \to \bigstar & \\ \mathsf{FMap}\langle 1 \rangle & \mathsf{v} = \mathsf{Maybe}\ \mathsf{v} & \\ \mathsf{FMap}\langle \mathsf{Char} \rangle & \mathsf{v} = \mathsf{FMapChar}\ \mathsf{v} & \\ \mathsf{FMap}\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle & \mathsf{v} = \mathsf{FMap}\langle \mathsf{t}_1 \rangle\ \mathsf{v} \times \mathsf{FMap}\langle \mathsf{t}_2 \rangle\ \mathsf{v} & \\ \mathsf{FMap}\langle \mathsf{t}_1 \times \mathsf{t}_2 \rangle & \mathsf{v} = \mathsf{FMap}\langle \mathsf{t}_1 \rangle\ (\mathsf{FMap}\langle \mathsf{t}_2 \rangle\ \mathsf{v}) & \end{array}$$

The definition of a type-indexed data type is very similar to the definition of a type-indexed function as seen in the previous subsection. Note that a name of a type-indexed data type starts with a capital letter. We give cases for 1, Char, $+$, and $\times$, and with these cases we have sufficient information to subsequently use FMap at any data type of kind $\bigstar$. Note that $\mathsf{FMap}\langle 1 \rangle$ is Maybe rather than Id since we use the Maybe monad for exception handling in the case of a partially defined finite map.

We assume that a suitable data structure, FMapChar, and an associated look-up function $\mathit{lookupChar} :: \forall \mathsf{v} . \mathsf{Char} \to \mathsf{FMapChar}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}$ for characters are predefined. The generic look-up function is then given by the following definition:

$$\begin{array}{lllll} \mathit{lookup}\langle \mathsf{t} :: \bigstar \rangle & & & :: \forall \mathsf{v} . \mathsf{t} \to \mathsf{FMap}\langle \mathsf{t} \rangle\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v} \\ \mathit{lookup}\langle 1 \rangle & () & t & = t \\ \mathit{lookup}\langle \mathsf{Char} \rangle & c & t & = \mathit{lookupChar}\ c\ t \\ \mathit{lookup}\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle & (\mathit{Inl}\ k_1) & (t_1, t_2) & = \mathit{lookup}\langle \mathsf{t}_1 \rangle\ k_1\ t_1 \\ \mathit{lookup}\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle & (\mathit{Inr}\ k_2) & (t_1, t_2) & = \mathit{lookup}\langle \mathsf{t}_2 \rangle\ k_2\ t_2 \\ \mathit{lookup}\langle \mathsf{t}_1 \times \mathsf{t}_2 \rangle & (k_1, k_2) & t & = (\mathit{lookup}\langle \mathsf{t}_1 \rangle\ k_1 \diamond \mathit{lookup}\langle \mathsf{t}_2 \rangle\ k_2)\ t. \end{array}$$

On sums the look-up function selects the appropriate map; on products it 'composes' the look-up functions for the component keys. The second argument to the look-up function is an element of the type-indexed type that we have defined before. Note how the definition of *lookup* relies on the fact that the second argument is a pair in the $+$-case and a nested finite map in the $\times$-case. This generic look-up function is a generalization of the type-specific look-up functions on strings and bushes that we have seen in the introduction.

Another generic function can be used to produce the empty trie for any data type:

$$
\begin{aligned}
&\mathit{empty}\langle \mathsf{t} :: \bigstar \rangle &&:: \forall \mathsf{v} \,.\, \mathsf{FMap}\langle \mathsf{t} \rangle\ \mathsf{v} \\
&\mathit{empty}\langle 1 \rangle &&= \mathit{Nothing} \\
&\mathit{empty}\langle \mathsf{Char} \rangle &&= \mathit{emptyChar} \\
&\mathit{empty}\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle &&= (\mathit{empty}\langle \mathsf{t}_1 \rangle, \mathit{empty}\langle \mathsf{t}_2 \rangle) \\
&\mathit{empty}\langle \mathsf{t}_1 \times \mathsf{t}_2 \rangle &&= \mathit{empty}\langle \mathsf{t}_1 \rangle,
\end{aligned}
$$

where *emptyChar* is the empty value of type FMapChar. The *empty* function serves as a simple example for a function that constructs values in a generic way.

2.3. Generic pattern matching

The pattern matching problem (for exact patterns) can be informally specified as follows: given a pattern and a text, find all occurrences of the pattern in the text. The pattern and the text may both be lists, or they may both be trees, etc. This section specifies a generic pattern-matching program for data types specified as fixed points of pattern functors. The specification is a rather inefficient program, but it can be transformed into an efficient program [29]. The efficient program is a generalization of the Knuth, Morris, and Pratt algorithm on lists [31] to arbitrary data types.

A pattern is a value of a type extended with variables. For example, the data type Bush is extended with a constructor for variables as follows:

```
data Var_Bush = Var       Int
              | Var_Leaf  Char
              | Var_Fork  Var_Bush Var_Bush.
```

In general, we want to extend a data type given as the fixed point of a functor Fix f with a case for variables. We can perform the extension on the functor directly, and we can parametrize over the functor f in question:

```
data VarF f r = Var Int
              | Val (f r).
```

With this definition, Fix (VarF f) is the extension of Fix f with variable case that we are interested in. In particular, one can easily define isomorphisms to confirm that Fix (VarF BushF) is equivalent to the previously defined type Var_Bush.

To establish that we want to store patterns in the extended data types, we define the abbreviation

```
type Pattern f = Fix (VarF f).
```

We construct a specification for pattern matching in three steps: firstly, we define a generic function *match* that matches a pattern against a complete value. In particular, it does not look for occurrences of the pattern in substructures of the value; secondly, we can systematically compute substructures of a value using a generic function *suffixes*. Finally, both *match* and *suffixes* are then combined into the function *pattern_match*,

that looks for a pattern all over a value. It turns out that we need a type-indexed type to store the results of *suffixes* and *pattern_match*.

We start with function *match* that matches a pattern against a value. A pattern matches a value if it is a variable, or if it has the same top-level constructor as the value, and all children match pairwise. On Bush, for example:

```
match_Bush                                  :: Bush → Var_Bush → Bool
match_Bush t              (Var i)           = True
match_Bush (Leaf c)       (Var_Leaf c')     = equalChar c c'
match_Bush (Fork l r)     (Var_Fork l' r') =
  match_Bush l l' ∧ match_Bush r r'
match_Bush _              _                 = False.
```

For the general case we use the function *zipWith* to match all children of a constructor pairwise:

```
match⟨f :: ★ → ★⟩                :: Fix f → Pattern f → Bool
match⟨f⟩ t          (In (Var x)) = True
match⟨f⟩ (In x) (In (Val y)) = case zipWith⟨f⟩ (match⟨f⟩) x y of
                                 {Nothing → False;
                                  Just t  → and⟨f⟩ t},
```

where *zipWith* and *and* are the generalizations of the list-processing functions defined in the Haskell prelude. On BushF, function *zipWith* is defined as follows:

```
zipWith_BushF :: ∀a b c.(a → b → c)
                 → BushF a → BushF b → Maybe (BushF c)
zipWith_BushF f (LeafF c1)     (LeafF c2)     =
  if equalChar c1 c2 then Just (LeafF c1) else Nothing
zipWith_BushF f (ForkF a1 b1) (ForkF a2 b2) =
  Just (ForkF (f a1 a2) (f b1 b2))
zipWith_BushF f _              _              = Nothing.
```

Of course, *zipWith* can also be defined generically for types of kind $\star \to \star$:

```
zipWith⟨f :: ★ → ★⟩                 :: ∀a b c.(a → b → c)
                                       → f a → f b → Maybe (f c)
zipWith⟨Id⟩       f a       b       = Just (f a b)
zipWith⟨K 1⟩      f ()      ()      = Just ()
zipWith⟨K Char⟩   f c1      c2      = if equalChar c1 c2
                                      then Just c1
                                      else Nothing
zipWith⟨f1 + f2⟩  f (Inl a1) (Inl a2) = do {x ← zipWith⟨f1⟩ f a1 a2;
                                            return (Inl x)}
zipWith⟨f1 + f2⟩  f (Inl a1) (Inr b2) = Nothing
zipWith⟨f1 + f2⟩  f (Inr b1) (Inl a2) = Nothing
```

$$
\begin{aligned}
&zipWith\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ f\ (Inr\ b_1)\ (Inr\ b_2) = \mathbf{do}\ \{y \leftarrow zipWith\langle \mathsf{f}_2\rangle\ f\ b_1\ b_2;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (Inr\ y)\}\\
&zipWith\langle \mathsf{f}_1 \times \mathsf{f}_2\rangle\ f\ (a_1,b_1)\ (a_2,b_2)\ = \mathbf{do}\ \{x \leftarrow zipWith\langle \mathsf{f}_1\rangle\ f\ a_1\ a_2;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad y \leftarrow zipWith\langle \mathsf{f}_2\rangle\ f\ b_1\ b_2;\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (x,y)\}.
\end{aligned}
$$

On BushF, function *and* is defined as follows:

$$
\begin{aligned}
&and\_BushF &&:: \mathsf{BushF\ Bool} \to \mathsf{Bool}\\
&and\_BushF\ (LeafF\ c) &&= True\\
&and\_BushF\ (ForkF\ a\ b) &&= a \wedge b.
\end{aligned}
$$

The generic definition of *and* is:

$$
\begin{aligned}
&and\langle \mathsf{f} :: \bigstar \to \bigstar\rangle &&:: \mathsf{f\ Bool} \to \mathsf{Bool}\\
&and\langle \mathsf{Id}\rangle\ b &&= b\\
&and\langle \mathsf{K\ 1}\rangle\ u &&= True\\
&and\langle \mathsf{K\ Char}\rangle\ c &&= True\\
&and\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ (Inl\ a) &&= and\langle \mathsf{f}_1\rangle\ a\\
&and\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ (Inr\ b) &&= and\langle \mathsf{f}_2\rangle\ b\\
&and\langle \mathsf{f}_1 \times \mathsf{f}_2\rangle\ (a,b) &&= and\langle \mathsf{f}_1\rangle\ a \wedge and\langle \mathsf{f}_2\rangle\ b.
\end{aligned}
$$

Having defined *match*, we need to define *suffixes* that computes the suffixes of a data structure generically. For a list, a suffix is a tail of the list. For example, the string *per* is a suffix of the string *paper*. We will now generalize the concept of suffixes in the following way: given a set of patterns, the generic pattern-matching problem will require finding for each suffix the subset of patterns matching (in the sense of *match*) the suffix. How do we compute all suffixes of a value of a data type? On lists, the suffixes of a list can be represented as a list of tails, computed by *tails*, a standard Haskell function:

$$
\begin{aligned}
&tails &&:: \forall \mathsf{a}\,.\,[\mathsf{a}] \to [[\mathsf{a}]]\\
&tails\ [\,] &&= [[\,]]\\
&tails\ t@(\_ : xs) &&= t : tails\ xs.
\end{aligned}
$$

For a value of an arbitrary data type we construct a value of a new data type, a *labelled* data type, that can be used to store all suffixes. A labelled data type is an extension of a data type that is used to store information at the internal nodes of a value.

The data type Labelled labels a data type given by a pattern functor:

$$
\begin{aligned}
&\mathsf{Labelled}\langle \mathsf{f} :: \bigstar \to \bigstar\rangle :: \bigstar \to \bigstar\\
&\mathsf{Labelled}\langle \mathsf{f}\rangle\ \mathsf{m} = \mathsf{Fix\ (Label}\langle \mathsf{f}\rangle\ \mathsf{m}).
\end{aligned}
$$

Here we use a generic abstraction, see Section 2.1, on the type level. The idea is the same as generic abstractions on functions. The type-indexed data type Label adds a label type to each constructor of a data type. In its definition, we make use of the fact that a Haskell data type is viewed as a sum of constructor applications, where the

fields of a constructor form a product. In Label, we traverse the sum structure, and add the label type once we reach a constructor. There are no recursive calls in the constructor case, therefore the product of fields is never traversed, and no $\times$-case is needed. We want to label the whole data type, but Label does not work recursively. Therefore, we compute a fixed point using Label in Labelled.

$$\begin{aligned}
&\mathsf{Label}\langle \mathsf{f} :: \star \to \star \rangle :: \star \to \star \to \star \\
&\mathsf{Label}\langle \mathsf{f}_1 + \mathsf{f}_2 \rangle\ \mathsf{m}\ \mathsf{r} = \mathsf{Label}\langle \mathsf{f}_1 \rangle\ \mathsf{m}\ \mathsf{r} + \mathsf{Label}\langle \mathsf{f}_2 \rangle\ \mathsf{m}\ \mathsf{r} \\
&\mathsf{Label}\langle c\ \mathbf{of}\ \mathsf{f} \rangle\ \ \mathsf{m}\ \mathsf{r} = \mathsf{f}\ \mathsf{r} \times \mathsf{m}
\end{aligned}$$

The type-indexed function *suffixes*, defined below, labels a value of a data type with the subtree rooted at each node. It uses a helper function *add*, which adds a label to a value of type f t, returning a value of type $\mathsf{Label}\langle \mathsf{f} \rangle\ \mathsf{m}\ \mathsf{t}$. As for the type-indexed type Label, we omit the $\times$-case for *add*: the function only inspects the sum structure and the constructors of a data type.

$$\begin{aligned}
&\mathit{add}\langle \mathsf{f} :: \star \to \star \rangle && :: \forall \mathsf{m}\ \mathsf{t}.\ \mathsf{m} \to \mathsf{f}\ \mathsf{t} \to \mathsf{Label}\langle \mathsf{f} \rangle\ \mathsf{m}\ \mathsf{t} \\
&\mathit{add}\langle \mathsf{f}_1 + \mathsf{f}_2 \rangle\ m\ (\mathit{Inl}\ x) &&= \mathit{Inl}\ (\mathit{add}\langle \mathsf{f}_1 \rangle\ m\ x) \\
&\mathit{add}\langle \mathsf{f}_1 + \mathsf{f}_2 \rangle\ m\ (\mathit{Inr}\ y) &&= \mathit{Inr}\ (\mathit{add}\langle \mathsf{f}_2 \rangle\ m\ y) \\
&\mathit{add}\langle c\ \mathbf{of}\ \mathsf{f} \rangle\ \ m\ x &&= (x, m)
\end{aligned}$$

The function *suffixes* is then defined as a recursive function that adds the subtrees rooted at each level to the tree. It adds the argument tree to the top level, and applies *suffixes* to the children by means of function *map*. It is the generalization of function *tails* to arbitrary data types.

$$\begin{aligned}
&\mathit{suffixes}\langle \mathsf{f} :: \star \to \star \rangle :: \mathsf{Fix}\ \mathsf{f} \to \mathsf{Labelled}\langle \mathsf{f} \rangle\ (\mathsf{Fix}\ \mathsf{f}) \\
&\mathit{suffixes}\langle \mathsf{f} \rangle\ m@(\mathit{In}\ t) = \mathit{In}\ (\mathit{add}\langle \mathsf{f} \rangle\ m\ (\mathit{map}\langle \mathsf{f} \rangle\ (\mathit{suffixes}\langle \mathsf{f} \rangle)\ t)).
\end{aligned}$$

Finally, we can specify a generic pattern-matching program. For each suffix, we compute the set of patterns that matches that suffix:

$$\begin{aligned}
&\mathit{pattern_match}\langle \mathsf{f} :: \star \to \star \rangle :: [\mathsf{Pattern}\ \mathsf{f}] \to \mathsf{Fix}\ \mathsf{f} \\
&\qquad\qquad \to \mathsf{Labelled}\langle \mathsf{f} \rangle\ [\mathsf{Pattern}\ \mathsf{f}] \\
&\mathit{pattern_match}\langle \mathsf{f} \rangle\ \mathit{pats} = \mathit{map}\langle \mathsf{Labelled}\langle \mathsf{f} \rangle \rangle \\
&\qquad\qquad (\lambda t \to \mathit{filter}\ (\mathit{match}\langle \mathsf{f} \rangle\ t)\ \mathit{pats}) \\
&\qquad\qquad \cdot\ \mathit{suffixes}\langle \mathsf{f} \rangle.
\end{aligned}$$

The data type Labelled that has been introduced in this section has other applications: for instance, it can also be used in the generic maximum segment sum problem [11], which requires finding a subtree of a tree with maximum sum.

3. Examples of translations to Haskell

The semantics of type-indexed data types will be given by means of specialization. This section gives some examples as an introduction to the formal rules provided in the following section.

We illustrate the main ideas by translating the digital search tree example to Haskell. This translation shows in particular how type-indexed data types are specialized in Generic Haskell: the Haskell code given here will be automatically generated by the Generic Haskell compiler. The example is structured into three sections: a translation of data types, a translation of type-indexed data types, and a translation of type-indexed functions that operate on type-indexed data types.

3.1. Translating data types

In general, a type-indexed function is translated to several functions: one for each user-defined data type on which it is used. These instances work on a slightly different, but isomorphic data type, that makes use of the types 1, $+$, and $\times$. We call this isomorphic type the *generic representation type* of a data type. By applying such a transformation, concepts that are hardwired in Haskell's **data** statement, such as a data type having multiple constructors, with a variable number of fields per constructor, are replaced by just type abstraction, type application and some basic type constructors. This implies, of course, that values of user-defined data types have to be translated to generic representation types. For example, the type Nat of natural numbers defined by

$$\textbf{data}\ \mathsf{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathsf{Nat}$$

is translated to the following type (in which Nat itself still appears), together with two conversion functions:

$$\begin{array}{ll}
\textbf{type}\ \mathsf{Nat}' & = 1 + \mathsf{Nat} \\
\mathit{from_Nat} & :: \mathsf{Nat} \to \mathsf{Nat}' \\
\mathit{from_Nat}\ \mathit{Zero} & = \mathit{Inl}\ () \\
\mathit{from_Nat}\ (\mathit{Succ}\ x) & = \mathit{Inr}\ x \\
\mathit{to_Nat} & :: \mathsf{Nat}' \to \mathsf{Nat} \\
\mathit{to_Nat}\ (\mathit{Inl}\ ()) & = \mathit{Zero} \\
\mathit{to_Nat}\ (\mathit{Inr}\ x) & = \mathit{Succ}\ x
\end{array}$$

The conversion functions *from_Nat* and *to_Nat* transform the top-level structure of a natural number; they are not recursive.

Furthermore, the mapping between data types and generic representation types translates n-ary products and n-ary sums to binary products and binary sums. This is revealed by looking at a more complex data type, for instance

$$\textbf{data}\ \mathsf{Tree\ a} = \mathit{Empty} \mid \mathit{Node}\ (\mathsf{Tree\ a})\ \mathsf{a}\ (\mathsf{Tree\ a}),$$

where the constructor *Node* takes three arguments. The generic representation type for Tree is

$$\textbf{type}\ \mathsf{Tree}'\ \mathsf{a} = 1 + \mathsf{Tree\ a} \times (\mathsf{a} \times \mathsf{Tree\ a}),$$

the conversion functions are

$$
\begin{array}{ll}
\mathit{from\_Tree} & :: \forall \mathsf{a} \,.\, \mathsf{Tree\ a} \to \mathsf{Tree'\ a} \\
\mathit{from\_Tree\ Empty} & = \mathit{Inl}\ () \\
\mathit{from\_Tree}\ (\mathit{Node}\ l\ v\ r) & = \mathit{Inr}\ (l, (v, r)) \\
\mathit{to\_Tree} & :: \forall \mathsf{a} \,.\, \mathsf{Tree'\ a} \to \mathsf{Tree\ a} \\
\mathit{to\_Tree}\ (\mathit{Inl}\ ()) & = \mathit{Empty} \\
\mathit{to\_Tree}\ (\mathit{Inr}\ (l, (v, r))) & = \mathit{Node}\ l\ v\ r.
\end{array}
$$

For convenience, we pair the two conversion functions:

$$
\begin{array}{ll}
\textbf{data}\ \mathsf{Iso\ a\ b} = \mathit{Iso}\{\mathit{from} :: \mathsf{a} \to \mathsf{b}, \mathit{to} :: \mathsf{b} \to \mathsf{a}\} \\
\mathit{iso\_Nat} & :: \mathsf{Iso\ Nat\ Nat'} \\
\mathit{iso\_Nat} & = \mathit{Iso\ from\_Nat\ to\_Nat} \\
\mathit{iso\_Tree} & :: \mathsf{Iso\ Tree\ Tree'} \\
\mathit{iso\_Tree} & = \mathit{Iso\ from\_Tree\ to\_Tree}.
\end{array}
$$

The conversion functions only affect the top-level structure of a data type. For recursive data types, the generic representation type still contains the original data type. The isomorphisms will be used in the translation of type-indexed data types and type-indexed functions to move between the structural view and the original data type as needed. If the function is recursive and operates on a recursive data type, then the conversion functions will be applied recursively, as well.

3.2. Translating type-indexed data types

A type-indexed data type is translated to several **newtype**s in Haskell: one for each type case in its definition. The translation proceeds in a similar fashion as in [17], but now for types instead of values. For example, the product case $\mathsf{t_1 \times t_2}$ takes two argument types for $\mathsf{t_1}$ and $\mathsf{t_2}$, and returns the type for the product. Recall the type-indexed data type FMap defined by

$$
\begin{array}{ll}
\mathsf{FMap\langle 1\rangle} & \mathsf{v} = \mathsf{Maybe\ v} \\
\mathsf{FMap\langle Char\rangle} & \mathsf{v} = \mathsf{FMapChar\ v} \\
\mathsf{FMap\langle t_1 + t_2\rangle} & \mathsf{v} = \mathsf{FMap\langle t_1\rangle\ v} \times \mathsf{FMap\langle t_2\rangle\ v} \\
\mathsf{FMap\langle t_1 \times t_2\rangle} & \mathsf{v} = \mathsf{FMap\langle t_1\rangle\ (FMap\langle t_2\rangle\ v)}.
\end{array}
$$

These equations are translated to:

$$
\begin{array}{lll}
\textbf{newtype}\ \mathsf{FMap\_Unit} & \mathsf{v} = \mathit{FMap\_Unit} & \mathsf{(Maybe\ v)} \\
\textbf{newtype}\ \mathsf{FMap\_Char} & \mathsf{v} = \mathit{FMap\_Char} & \mathsf{(FMapChar\ v)} \\
\textbf{newtype}\ \mathsf{FMap\_Either} & \mathsf{fma\ fmb\ v} = \mathit{FMap\_Either} & \mathsf{(fma\ v, fmb\ v)} \\
\textbf{newtype}\ \mathsf{FMap\_Product} & \mathsf{fma\ fmb\ v} = \mathit{FMap\_Product} & \mathsf{(fma\ (fmb\ v))}.
\end{array}
$$

The constructor names are generated automatically. This implies that a value of a type-indexed data type can only be constructed by means of a generic function. Thus, a type-indexed data type can be viewed as an abstract type.

Finally, for each data type t on which we want to use a trie we generate a suitable instance FMap_t.

```
type    FMap_Nat' v = FMap_Either FMap_Unit FMap_Nat v
newtype FMap_Nat  v = FMap_Nat{unFMap_Nat :: FMap_Nat' v}
```

Note that we use **newtype** for FMap_Nat because it is not possible to define recursive **type**s in Haskell. The types FMap_Nat and FMap_Nat′ can easily be converted into each other by means of the following pair of isomorphisms:

```
iso_FMap_Nat :: ∀v . Iso (FMap_Nat v) (FMap_Nat' v)
iso_FMap_Nat = Iso unFMap_Nat FMap_Nat.
```

3.3. Translating type-indexed functions on type-indexed data types

The translation of a type-indexed function that takes a type-indexed data type as an argument is a generalization of the translation of 'ordinary' type-indexed functions. The translation consists of two parts: a translation of the type-indexed function itself, and a specialization on each data type on which the type-indexed function is used, together with a conversion function.

A type-indexed function is translated by generating a function, together with its type signature, for each case of its definition. For the type indices of kind ★ (i.e. 1 and Char) we generate types that are instances of the type of the generic function. The occurrences of the type index are replaced by the instance type, and occurrences of type-indexed data types are replaced by the translation of the type-indexed data type on the type index. As an example, for the generic function *lookup* of type:

$$lookup\langle t :: \bigstar \rangle :: \forall v\,.\,t \to \mathsf{FMap}\langle t \rangle\ v \to \mathsf{Maybe}\ v,$$

the instances are obtained by replacing t by 1 or Char, and by replacing FMap⟨t⟩ by FMap_Unit or FMap_Char, respectively. So, for the function *lookup* we have that the user-supplied equations

$$\begin{aligned} &lookup\langle 1 \rangle \quad ()\ t = t \\ &lookup\langle \mathsf{Char} \rangle\ c\ \ t = lookupChar\ c\ t, \end{aligned}$$

are translated into

$$\begin{aligned} &lookup\_Unit && :: \forall v\,.\,1 \to \mathsf{FMap\_Unit}\ v \to \mathsf{Maybe}\ v \\ &lookup\_Unit\ ()\ (FMap\_Unit\ t) && = t \\ &lookup\_Char && :: \forall v\,.\,\mathsf{Char} \to \mathsf{FMap\_Char}\ v \to \mathsf{Maybe}\ v \\ &lookup\_Char\ c\ \ (FMap\_Char\ t) && = lookupChar\ c\ t. \end{aligned}$$

Note that we have to wrap the trie constructors around the second argument of the function.

For the type indices of kind $\bigstar \to \bigstar \to \bigstar$ (i.e. '+' and '×') we generate types that take two functions as arguments, corresponding to the instances of the generic function

on the arguments of '+' and '×', and return a function of the combined type, see [17]. For example, the following lines

$$\begin{aligned}
&\mathit{lookup}\langle \mathsf{t}_1 + \mathsf{t}_2\rangle\ (\mathit{Inl}\ k_1)\ (t_1, t_2) = \mathit{lookup}\langle \mathsf{t}_1\rangle\ k_1\ t_1\\
&\mathit{lookup}\langle \mathsf{t}_1 + \mathsf{t}_2\rangle\ (\mathit{Inr}\ k_2)\ (t_1, t_2) = \mathit{lookup}\langle \mathsf{t}_2\rangle\ k_2\ t_2\\
&\mathit{lookup}\langle \mathsf{t}_1 \times \mathsf{t}_2\rangle\ (k_1, k_2)\ \ t \qquad = (\mathit{lookup}\langle \mathsf{t}_1\rangle\ k_1 \diamond \mathit{lookup}\langle \mathsf{t}_2\rangle\ k_2)\ t
\end{aligned}$$

are translated into the following functions:

$$\begin{aligned}
&\mathit{lookup_Either} :: \forall \mathsf{a}\ \mathsf{fma}\,.\,\forall \mathsf{b}\ \mathsf{fmb}\,.\\
&\qquad (\forall \mathsf{v}\,.\,\mathsf{a} \to \mathsf{fma}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\qquad \to (\forall \mathsf{v}\,.\,\mathsf{b} \to \mathsf{fmb}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\qquad \to (\forall \mathsf{v}\,.\,\mathsf{a} + \mathsf{b} \to \mathsf{FMap_Either}\ \mathsf{fma}\ \mathsf{fmb}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\mathit{lookup_Either}\quad \mathit{lua}\ \mathit{lub}\ (\mathit{Inl}\ a)\ (\mathit{FMap_Either}\ (\mathit{fma}, \mathit{fmb})) = \mathit{lua}\ a\ \mathit{fma}\\
&\mathit{lookup_Either}\quad \mathit{lua}\ \mathit{lub}\ (\mathit{Inr}\ b)\ (\mathit{FMap_Either}\ (\mathit{fma}, \mathit{fmb})) = \mathit{lub}\ b\ \mathit{fmb}\\
&\mathit{lookup_Product} :: \forall \mathsf{a}\ \mathsf{fma}\,.\,\forall \mathsf{b}\ \mathsf{fmb}\,.\\
&\qquad (\forall \mathsf{v}\,.\,\mathsf{a} \to \mathsf{fma}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\qquad \to (\forall \mathsf{v}\,.\,\mathsf{b} \to \mathsf{fmb}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\qquad \to (\forall \mathsf{v}\,.\,\mathsf{a} \times \mathsf{b} \to \mathsf{FMap_Product}\ \mathsf{fma}\ \mathsf{fmb}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\mathit{lookup_Product}\ \mathit{lua}\ \mathit{lub}\ (a, b)\quad (\mathit{FMap_Product}\ t) = (\mathit{lua}\ a \diamond \mathit{lub}\ b)\ t.
\end{aligned}$$

The translation involves replacing the recursive invocations $\mathit{lookup}\langle \mathsf{t}_1\rangle$ and $\mathit{lookup}\langle \mathsf{t}_2\rangle$ by the function arguments *lua* and *lub*.

Now we generate a specialization of the type-indexed function for each data type on which it is used. For example, on Nat we have

$$\begin{aligned}
&\mathit{lookup_Nat} :: \forall \mathsf{v}\,.\,\mathsf{Nat} \to \mathsf{FMap_Nat}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}\\
&\mathit{lookup_Nat} = \mathit{conv_lookup_Nat}\ (\mathit{lookup_Either}\ \mathit{lookup_Unit}\ \mathit{lookup_Nat}).
\end{aligned}$$

The expression (*lookup_Either lookup_Unit lookup_Nat*) is generated directly from the type Nat', which is defined as $1 + \mathsf{Nat}$: each of the type constants has been replaced by the corresponding case or specialization of the *lookup* function, and type application is translated into value application. Unfortunately, this expression does not have the type we require for *lookup_Nat*—the type given in the type signature—but rather the type

$$\forall \mathsf{v}\,.\,\mathsf{Nat}' \to \mathsf{FMap\_Nat}'\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}.$$

However, this type is isomorphic to the type we need, because Nat' is isomorphic to Nat, and $\mathsf{FMap\_Nat}'$ is isomorphic to FMap_Nat. The conversion function *conv_lookup_Nat* witnesses this isomorphism:

$$\begin{aligned}
&\mathit{conv_lookup_Nat} :: (\forall \mathsf{v}\,.\,\mathsf{Nat}' \to \mathsf{FMap_Nat}'\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\qquad \to (\forall \mathsf{v}\,.\,\mathsf{Nat} \to \mathsf{FMap_Nat}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\mathit{conv_lookup_Nat}\ \mathit{lu}\\
&\quad = \lambda t\ \mathit{fmt} \to \mathit{lu}\ (\mathit{from}\ \mathit{iso_Nat}\ t)\ (\mathit{from}\ \mathit{iso_FMap_Nat}\ \mathit{fmt}).
\end{aligned}$$

```
class FMap fma a | a → fma where
  lookup                                  :: ∀v . a → fma v → Maybe v
instance FMap Maybe () where
  lookup ()       fm                      = fm
data FMap_Either fma fmb v                = FMap_Either (fma v, fmb v)
instance (FMap fma a, FMap fmb b)
      ⇒ FMap (FMap_Either fma fmb) (a + b) where
  lookup (Inl a) (FMap_Either fma fmb) = lookup a fma
  lookup (Inr b) (FMap_Either fma fmb) = lookup b fmb
data FMap_Product fma fmb v = FMap_Product (fma (fmb v))
instance (FMap fma a, FMap fmb b)
      ⇒ FMap (FMap_Product fma fmb) (a × b) where
  lookup (a, b)   (FMap_Product fma)      = (lookup a ◇ lookup b) fma
```

Fig. 1. Implementing FMap in Haskell directly.

Note that the functions *to iso_Nat* and *to iso_FMap_Nat* are not used on the right-hand side of the definition of *conv_lookup_Nat*. This is because no values of type Nat or FMap_Nat are built for the result of the function. If we look at the instance of *empty* for Nat, we are in a different situation. Here we have

```
empty_Nat :: ∀v . FMap_Nat v
empty_Nat = conv_Empty_Nat (empty_Either empty_Unit empty_Nat)
```

where

```
conv_empty_Nat    :: (∀v . FMap_Nat′ v) → (∀v . FMap_Nat v)
conv_empty_Nat e = to iso_FMap_Nat e.
```

Generally, for each specialization of a generic function a conversion function is generated that uses the relevant isomorphism pairs at the appropriate positions, dictated by the generic representation type of the type on which we want to obtain an instance. Section 4.5 shows that this can be done in a systematic way.

3.4. *Implementing* FMap *using type classes*

Alternatively, we can use multi-parameter type classes and functional dependencies [30] to implement a type-indexed data type such as FMap in Haskell. An example is given in Fig. 1. With type classes, the recursive invocations of the generic functions are not passed as explicit type arguments (*lua* and *lub* in the definition of *lookup_Either* and *lookup_Product*). They remain implicit in the class context, and it is the task of the Haskell compiler to pass these implicit contexts around and to use them as necessary.

We will use the explicit style introduced in Section 3.3 throughout the rest of the paper.

4. Specializing type-indexed types and values

This section gives a formal semantics of type-indexed data types by means of specialization. Examples of this translation have been given in the previous section. The specialization to concrete data type instances removes the type arguments of type-indexed data types and functions. In other words, type-indexed data types and functions can be used at no run-time cost, since all type arguments are removed at compile-time. The specialization can be seen as partial evaluation of type-indexed functions where the type index is the static argument. The specialization is obtained by lifting the semantic description of type-indexed functions given in [18] to the level of data types.

Type-indexed data types and type-indexed functions take types as arguments, and return types and functions, respectively. For the formal description of type-indexed data types and functions and for their semantics we use an extension of the polymorphic lambda calculus, described in Section 4.1. Section 4.2 briefly discusses the form of type-indexed definitions. The description of the specialization is divided into two parts: Section 4.3 deals with the specialization of type-indexed data types, and Section 4.4 deals with the specialization of type-indexed functions that involve type-indexed data types. Section 4.5 shows how the gap between the formal type language and Haskell's data types can be bridged, and Section 4.6 summarizes.

4.1. The polymorphic lambda calculus

This section briefly introduces kinds, types, type schemes, and terms.

Kind terms are formed by

$$\begin{array}{lll} \mathfrak{T},\mathfrak{U} \in \mathit{Kind} ::= \bigstar & & \text{kind of types} \\ \quad\mid\ (\mathfrak{T} \to \mathfrak{U}) & & \text{function kind.} \end{array}$$

We distinguish between type terms and type schemes: the language of type terms comprises the types that may appear as type indices; the language of type schemes comprises the constructs that are required for the translation of generic definitions (such as polymorphic types).

Type terms are built from type constants and type variables using type application and type abstraction:

$$\begin{array}{lll} \mathsf{t},\mathsf{u} \in \mathit{Type} ::= \mathsf{C} & & \text{type constant} \\ \quad\mid\ \mathsf{a} & & \text{type variable} \\ \quad\mid\ (\Lambda \mathsf{a} :: \mathfrak{U}\,.\,\mathsf{t}) & & \text{type abstraction} \\ \quad\mid\ (\mathsf{t}\ \mathsf{u}) & & \text{type application.} \end{array}$$

For typographic simplicity, we will often omit the kind annotation in $\Lambda \mathsf{a} :: \mathfrak{U}\,.\,\mathsf{t}$ (especially if $\mathfrak{U} = \bigstar$) and we abbreviate nested abstractions $\Lambda \mathsf{a}_1 \ldots \Lambda \mathsf{a}_m\,.\,\mathsf{t}$ by $\Lambda \mathsf{a}_1\ \ldots\ \mathsf{a}_m\,.\,\mathsf{t}$.

In order to be able to model Haskell's data types the set of type constants should include at least the types $1, \mathsf{Char}$, '$+$', '$\times$', and 'c **of**' for all known constructors in the program. Furthermore, it should include a family of fixed point operators indexed by kind: $\mathsf{Fix}_{\mathfrak{T}} :: (\mathfrak{T} \to \mathfrak{T}) \to \mathfrak{T}$. In the examples, we will often omit the kind annotation

$\mathfrak{T}$ in $\mathsf{Fix}_{\mathfrak{T}}$. We may additionally add the function space constructor '$\rightarrow$' or universal quantifiers $\forall_{\mathfrak{U}} :: (\mathfrak{U} \rightarrow \bigstar) \rightarrow \bigstar$ to the set of type constants (see Section 4.5 for an example).

Type schemes are formed by

$$\begin{array}{lll} \mathsf{r},\mathsf{s} \in \mathit{Scheme} ::= & \mathsf{t} & \text{type term} \\ \mid & (\mathsf{r} \rightarrow \mathsf{s}) & \text{functional type} \\ \mid & (\forall \mathsf{a} :: \mathfrak{U} . \mathsf{s}) & \text{polymorphic type.} \end{array}$$

Terms are formed by

$$\begin{array}{lll} t,u \in \mathit{Term} ::= & c & \text{constant} \\ \mid & a & \text{variable} \\ \mid & (\lambda a :: \mathsf{s} . t) & \text{abstraction} \\ \mid & (t\ u) & \text{application} \\ \mid & (\lambda \mathsf{a} :: \mathfrak{U} . t) & \text{universal abstraction} \\ \mid & (t\ \mathsf{r}) & \text{universal application.} \end{array}$$

Here, $\lambda \mathsf{a} :: \mathfrak{U} . t$ denotes universal abstraction (forming a polymorphic value) and $t\ \mathsf{r}$ denotes universal application (instantiating a polymorphic value). We use the same syntax for value abstraction $\lambda a :: \mathsf{s} . t$ (here a is a value variable) and universal abstraction $\lambda \mathsf{a} :: \mathfrak{U} . t$ (here a is a type variable). We assume that the set of value constants includes at least the polymorphic fixed point operator

$$\mathit{fix} :: \forall \mathsf{a} . (\mathsf{a} \rightarrow \mathsf{a}) \rightarrow \mathsf{a}$$

and suitable functions for each of the other type constants (such as () for '1', *Inl*, *Inr*, and **case** for '+', and *outl*, *outr*, and (,) for '$\times$'). To improve readability we will usually omit the type argument of *fix*.

We omit the standard typing rules for the polymorphic lambda calculus.

4.2. On the form of type-indexed definitions

The type-indexed definitions given in Section 2 implicitly define a catamorphism on the language of types. For the specialization we have to make these catamorphisms explicit. This section describes the different views on type-indexed definitions.

Almost all inductive definitions of type-indexed functions and data types given in Section 2 take the form of a catamorphism:

$$\begin{array}{ll} \mathit{cata}\langle 1 \rangle & = \mathit{cata}_1 \\ \mathit{cata}\langle \mathsf{Char} \rangle & = \mathit{cata}_{\mathsf{Char}} \\ \mathit{cata}\langle \mathsf{t}_1 + \mathsf{t}_2 \rangle & = \mathit{cata}_+\ (\mathit{cata}\langle \mathsf{t}_1 \rangle)\ (\mathit{cata}\langle \mathsf{t}_2 \rangle) \\ \mathit{cata}\langle \mathsf{t}_1 \times \mathsf{t}_2 \rangle & = \mathit{cata}_\times\ (\mathit{cata}\langle \mathsf{t}_1 \rangle)\ (\mathit{cata}\langle \mathsf{t}_2 \rangle) \\ \mathit{cata}\langle c\ \mathbf{of}\ \mathsf{t}_1 \rangle & = \mathit{cata}_{c\ \mathbf{of}}\ \ (\mathit{cata}\langle \mathsf{t}_1 \rangle). \end{array}$$

These equations implicitly define the family of functions cata_1, $\mathit{cata}_{\mathsf{Char}}$, cata_+, $\mathit{cata}_\times$, and $\mathit{cata}_{c\ \mathbf{of}}$. In the sequel, we will assume that type-indexed functions and data types

are explicitly defined as a catamorphism. For example, for digital search trees we have

$$\begin{aligned}
\mathsf{FMap}_1 &= \Lambda \mathsf{v}\,.\,\mathsf{Maybe}\ \mathsf{v}\\
\mathsf{FMap}_{\mathsf{Char}} &= \Lambda \mathsf{v}\,.\,\mathsf{FMapChar}\ \mathsf{v}\\
\mathsf{FMap}_{+} &= \Lambda \mathsf{fMap_a}\ \mathsf{fMap_b}\,.\,\Lambda \mathsf{v}\,.\,\mathsf{fMap_a}\ \mathsf{v} \times \mathsf{fMap_b}\ \mathsf{v}\\
\mathsf{FMap}_{\times} &= \Lambda \mathsf{fMap_a}\ \mathsf{fMap_b}\,.\,\Lambda \mathsf{v}\,.\,\mathsf{fMap_a}\ (\mathsf{fMap_b}\ \mathsf{v})\\
\mathsf{FMap}_{c\ \mathbf{of}} &= \Lambda \mathsf{fMap_a}\,.\,\Lambda \mathsf{v}\,.\,\mathsf{fMap_a}\ \mathsf{v}.
\end{aligned}$$

Some inductive definitions, such as the definition of Label, also use the argument types themselves in their right-hand sides. Such functions are called paramorphisms [36], and are characterized by

$$\begin{aligned}
para\langle 1\rangle &= para_1\\
para\langle \mathsf{Char}\rangle &= para_{\mathsf{Char}}\\
para\langle \mathsf{t}_1 + \mathsf{t}_2\rangle &= para_{+}\ \mathsf{t}_1\ \mathsf{t}_2\ (para\langle \mathsf{t}_1\rangle)\ (para\langle \mathsf{t}_2\rangle)\\
para\langle \mathsf{t}_1 \times \mathsf{t}_2\rangle &= para_{\times}\ \mathsf{t}_1\ \mathsf{t}_2\ (para\langle \mathsf{t}_1\rangle)\ (para\langle \mathsf{t}_2\rangle)\\
para\langle c\ \mathbf{of}\ \mathsf{t}_1\rangle &= para_{c\ \mathbf{of}}\ \mathsf{t}_1\ (para\langle \mathsf{t}_1\rangle).
\end{aligned}$$

Fortunately, every paramorphism can be transformed into a catamorphism by tupling it with the identity. Likewise, mutually recursive definitions can be transformed into simple catamorphisms using tupling.

Section 4.3 describes how to specialize type-indexed data types with type indices that appear in the set of type constants: 1, Char, '+', '×', and 'c **of**'. However, we have also used the type indices Id., K 1, K Char, and lifted versions of '+' and '×'. How are type-indexed data types with these type indices specialized? The specialization of type-indexed data types with higher-order type indices proceeds in much the same fashion as in the following section. Essentially, the process only has to be lifted to higher-order type indices. For the details of this lifting process see [18, Section 3.2].

4.3. Specializing type-indexed data types

Rather amazingly, the process of specialization of type-indexed functions and type-indexed data types can be phrased as an interpretation of the simply typed lambda calculus. The interpretation of the constants (1, Char '+', '×', and 'c **of**') is obtained from the definition of the type-indexed data type as a catamorphism. The remaining constructs are interpreted generically: type application is interpreted as type application (albeit in a different domain), abstraction as abstraction, and fixed points as fixed points.

The first thing we have to do is to generalize the 'type' of a type-indexed data type. In the previous sections, the type-indexed data types had a fixed kind, for example, $\mathsf{FMap}_{\mathsf{t}::\star} :: \star \to \star$. However, when type application is interpreted as application, we have that $\mathsf{FMap}_{\mathsf{List\ a}} = \mathsf{FMap}_{\mathsf{List}}\mathsf{FMap}_{\mathsf{a}}$. Since List is of kind $\star \to \star$, we have to extend the domain of FMap$_{_}$ by giving it a kind-indexed kind, in such a way that $\mathsf{FMap}_{\mathsf{List}} :: (\star \to \star) \to (\star \to \star)$.

Generalizing the above example, we have that a type-indexed data type possesses a kind-indexed kind:

$$\mathsf{Data}_{\mathsf{t}::\mathfrak{T}} :: \mathfrak{Data}_{\mathfrak{T}},$$

where $\mathfrak{Data}_{\mathfrak{T}}$ has the following form:

$$\begin{aligned}
\mathfrak{Data}_{\mathfrak{T}::\square} &:: \square \\
\mathfrak{Data}_{\star} &= \boxed{} \\
\mathfrak{Data}_{\mathfrak{A}\to\mathfrak{B}} &= \mathfrak{Data}_{\mathfrak{A}} \to \mathfrak{Data}_{\mathfrak{B}}.
\end{aligned}$$

Here, '$\square$' is the superkind: the type of kinds. Note that only the definition of $\mathfrak{Data}_{\star}$, as indicated by the box, has to be given to complete the definition of the kind-indexed kind. The definition of $\mathfrak{Data}$ on functional kinds is dictated by the specialization process. Since type application is interpreted by type application, the kind of a type with a functional kind is functional.

For example, the kind of the type-indexed data type $\mathsf{FMap}_{\mathsf{t}}$, where t is a type of kind $\star$ is

$$\mathfrak{FMap}_{\star} = \star \to \star.$$

As noted above, the process of specialization is phrased as an interpretation of the simply typed lambda calculus. The interpretation of the constants (1, Char, '$+$', '$\times$', and 'c **of**') is obtained from the definition of the type-indexed data type as a catamorphism, and the interpretation of application, abstraction, and fixed points is given via an environment model [38] for the type-indexed data type.

An environment model is an applicative structure $(\mathbf{M}, \mathbf{app}, \mathbf{const})$, where $\mathbf{M}$ is the domain of the structure, **app** is a mapping that interprets functions, and where **const** maps constants to the domain of the structure. In order to qualify as an environment model, an applicative structure has to be extensional and must satisfy the so-called combinatory model condition. The precise definitions of these concepts can be found in [38]. For an arbitrary type-indexed data type $\mathsf{Data}_{\mathsf{t}::\mathfrak{T}} :: \mathfrak{Data}_{\mathfrak{T}}$ we use the following applicative structure:

$$\begin{aligned}
\mathbf{M}^{\mathfrak{T}} &= \mathit{Type}^{\mathfrak{Data}_{\mathfrak{T}}} / \mathscr{E} \\
\mathbf{app}_{\mathfrak{T},\mathfrak{U}}\,[\mathsf{t}]\,[\mathsf{u}] &= [\mathsf{t}\ \mathsf{u}] \\
\mathbf{const}(\mathsf{C}) &= [\mathsf{Data}_{\mathsf{C}}].
\end{aligned}$$

The domain of the applicative structure for a kind $\mathfrak{T}$ is the equivalence class of the set of types of kind $\mathfrak{Data}_{\mathfrak{T}}$, under an appropriate set of equations $\mathscr{E}$ between type terms, that is, β- and η-equality and $\mathsf{f}\ (\mathsf{Fix}_{\mathfrak{T}}\ \mathsf{f}) = \mathsf{Fix}_{\mathfrak{T}}\ \mathsf{f}$ for all kinds $\mathfrak{T}$ and type constructors f of kind $\mathfrak{T}\to\mathfrak{T}$. The application of two equivalence classes of types (denoted by $[\mathsf{t}]$ and $[\mathsf{u}]$) is the equivalence class of the application of the types. The definition of the constants is obtained from the definition as a catamorphism. It can be verified that the applicative structure thus defined is an environment model.

It remains to specify the interpretation of the fixed point operators, which is the same for all type-indexed data types:

$$\mathbf{const}(\mathsf{Fix}_{\mathfrak{T}}) = [\mathsf{Fix}_{\mathfrak{Data}_{\mathfrak{T}}}].$$

4.4. Specializing type-indexed values

A type-indexed value possesses a kind-indexed type [17],

$$poly_{\mathsf{t}::\mathfrak{T}} :: \mathsf{Poly}_{\mathfrak{T}}\ \mathsf{Data}_{\mathsf{t}}^{1}\ \ldots\ \mathsf{Data}_{\mathsf{t}}^{n}$$

in which $\mathsf{Poly}_{\mathfrak{T}}$ has the following general form

$$\begin{aligned}
\mathsf{Poly}_{\mathfrak{T}::\square} &:: \mathfrak{Data}_{\mathfrak{T}}^{1} \to \cdots \to \mathfrak{Data}_{\mathfrak{T}}^{n} \to \bigstar \\
\mathsf{Poly}_{\bigstar} &= \Lambda \mathsf{x}_1 :: \mathfrak{Data}_{\bigstar}^{1} . \ldots . \Lambda \mathsf{x}_n :: \mathfrak{Data}_{\bigstar}^{n} . \boxed{} \\
\mathsf{Poly}_{\mathfrak{A}\to\mathfrak{B}} &= \Lambda \mathsf{x}_1 :: \mathfrak{Data}_{\mathfrak{A}\to\mathfrak{B}}^{1} . \ldots . \Lambda \mathsf{x}_n :: \mathfrak{Data}_{\mathfrak{A}\to\mathfrak{B}}^{n} . \\
&\qquad \forall \mathsf{a}_1 :: \mathfrak{Data}_{\mathfrak{A}}^{1} . \ldots . \forall \mathsf{a}_n :: \mathfrak{Data}_{\mathfrak{A}}^{n} . \\
&\qquad\quad \mathsf{Poly}_{\mathfrak{A}}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_n \to \mathsf{Poly}_{\mathfrak{B}}\ (\mathsf{x}_1\ \mathsf{a}_1)\ \ldots\ (\mathsf{x}_n\ \mathsf{a}_n).
\end{aligned}$$

Again, note that only an equation for $\mathsf{Poly}_{\bigstar}$ has to be given to complete the definition of the kind-indexed type. The definition of Poly on functional kinds is dictated by the specialization process. The presence of type-indexed data types slightly complicates the type of a type-indexed value. In [17] $\mathsf{Poly}_{\mathfrak{T}}$ takes n arguments of kind $\mathfrak{T}$. Here $\mathsf{Poly}_{\mathfrak{T}}$ takes n possibly different type arguments obtained from the type-indexed data type arguments. For example, for the type of the look-up function we have

$$\begin{aligned}
\mathsf{Lookup}_{\mathfrak{T}::\square} &:: \mathfrak{Id}_{\mathfrak{T}} \to \mathfrak{FMap}_{\mathfrak{T}} \to \bigstar \\
\mathsf{Lookup}_{\bigstar} &= \Lambda \mathsf{k} . \Lambda \mathsf{fmk} . \forall \mathsf{v} . \mathsf{k} \to \mathsf{fmk}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v},
\end{aligned}$$

where $\mathfrak{Id}$ is the identity function on kinds. From the definition of the generic look-up function we obtain the following equations:

$$\begin{aligned}
lookup_{\mathsf{t}::\mathfrak{T}} &:: \mathsf{Lookup}_{\mathfrak{T}}\ \mathsf{Id}_{\mathsf{t}}\ \mathsf{FMap}_{\mathsf{t}} \\
lookup_{1} &= \lambda \mathsf{v}\ k\ fmk\ .fmk \\
lookup_{\mathsf{Char}} &= lookupChar \\
lookup_{+} &= \lambda \mathsf{a}\ \mathsf{fma}\ lookup_{\mathsf{a}} . \lambda \mathsf{b}\ \mathsf{fmb}\ lookup_{\mathsf{b}} . \\
&\quad \lambda \mathsf{v}\ k\ (fmkl, fmkr)\ .\ \mathbf{case}\ k\ \mathbf{of}\ \{ Inl\ a \to lookup_{\mathsf{a}}\ \mathsf{v}\ a\ fmkl; \\
&\qquad\qquad\qquad\qquad\qquad\qquad Inr\ b \to lookup_{\mathsf{b}}\ \mathsf{v}\ b\ fmkr \} \\
lookup_{\times} &= \lambda \mathsf{a}\ \mathsf{fma}\ lookup_{\mathsf{a}} . \lambda \mathsf{b}\ \mathsf{fmb}\ lookup_{\mathsf{b}} . \\
&\quad \lambda \mathsf{v}\ (kl, kr)\ fmk\ .\ \mathbf{case}\ lookup_{\mathsf{a}}\ (\mathsf{fmb}\ \mathsf{v})\ kl\ fmk\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \{ Nothing \to Nothing; \\
&\qquad\qquad\qquad\qquad\ \ Just\ fmk' \to lookup_{\mathsf{b}}\ \mathsf{v}\ kr\ fmk' \} \\
lookup_{c\ \mathbf{of}} &= \lambda \mathsf{a}\ \mathsf{fma}\ lookup_{\mathsf{a}} . \lambda \mathsf{v}\ k\ fmk\ .\ lookup_{\mathsf{a}}\ \mathsf{v}\ k\ fmk.
\end{aligned}$$

Just as with type-indexed data types, type-indexed values on type-indexed data types are specialized by means of an interpretation of the simply typed lambda calculus. The

environment model used for the specialization is somewhat more involved than the one given in Section 4.3. The domain of the environment model is now a dependent product: the type of the last component (the equivalence class of the terms of type $\mathsf{Poly}_{\mathfrak{T}}\ \mathsf{d}_1\ \ldots\ \mathsf{d}_n$) depends on the first n components (the equivalence classes of the type schemes $\mathsf{d}_1\ \ldots\ \mathsf{d}_n$ of kind $\mathfrak{T}$). Note that the application operator applies the term component of its first argument to both the type and the term components of the second argument:

$$\begin{aligned}
\mathbf{M}^{\mathfrak{T}} &= ([\mathsf{d}_1] \in \mathit{Scheme}^{\mathfrak{Data}^1_{\mathfrak{T}}} / \mathscr{E}, \ldots, [\mathsf{d}_n] \in \mathit{Scheme}^{\mathfrak{Data}^n_{\mathfrak{T}}} / \mathscr{E};\\
&\qquad \mathit{Term}^{\mathsf{Poly}_{\mathfrak{T}}\ \mathsf{d}_1\ \ldots\ \mathsf{d}_n} / \mathscr{E})\\
\mathbf{app}_{\mathfrak{T},\mathfrak{U}}\ &([\mathsf{r}_1], \ldots, [\mathsf{r}_n]; [t])\ ([\mathsf{s}_1], \ldots, [\mathsf{s}_n]; [u])\\
&= ([\mathsf{r}_1\ \mathsf{s}_1], \ldots, [\mathsf{r}_n\ \mathsf{s}_n]; [t\ \mathsf{s}_1\ \ldots\ \mathsf{s}_n\ u])\\
\mathbf{const}(\mathsf{C}) &= ([\mathsf{Data}^1_{\mathsf{C}}], \ldots, [\mathsf{Data}^n_{\mathsf{C}}]; [\mathit{poly}_{\mathsf{C}}]).
\end{aligned}$$

Again, the interpretation of fixed points is the same for different type-indexed values:

$$\mathbf{const}(\mathsf{Fix}_{\mathfrak{T}}) = ([\mathsf{Fix}_{\mathfrak{Data}^1_{\mathfrak{T}}}], \ldots, [\mathsf{Fix}_{\mathfrak{Data}^n_{\mathfrak{T}}}]; [\mathit{poly}_{\mathsf{Fix}_{\mathfrak{T}}}]),$$

where $\mathit{poly}_{\mathsf{Fix}_{\mathfrak{T}}}$ is given by

$$\begin{aligned}
\mathit{poly}_{\mathsf{Fix}_{\mathfrak{T}}} &= \lambda \mathsf{f}_1\ \ldots\ \mathsf{f}_n\,.\,\lambda \mathit{poly}_{\mathsf{f}} :: \mathsf{Poly}_{\mathfrak{T}\to\mathfrak{T}}\ \mathsf{f}_1\ \ldots\ \mathsf{f}_n\,.\\
&\qquad \mathit{fix}\ \mathit{poly}_{\mathsf{f}}\ (\mathsf{Fix}_{\mathfrak{Data}^1_{\mathfrak{T}}}\ \mathsf{f}_1)\ \ldots\ (\mathsf{Fix}_{\mathfrak{Data}^n_{\mathfrak{T}}}\ \mathsf{f}_n).
\end{aligned}$$

4.5. Conversion functions

As can be seen in the example of Section 3, we do not interpret type-indexed functions and data types on Haskell data types directly, but rather on slightly different, yet isomorphic types. Furthermore, since Haskell does not allow recursive type synonyms, we must introduce a **newtype** for each specialization of a type-indexed data type, thereby again creating a different, but isomorphic type from the one we are interested in. As a consequence, we have to generate conversion functions that mediate between these isomorphic types.

These conversion functions are easily generated, both for type-indexed values and data types, and can be stored in pairs, as values of type *Iso*. The only difficult task is to plug them in at the right positions. This problem is solved by lifting the conversion functions to the type of the specialized generic function. This again is a generic program [18, Section 6.1.3], which makes use of the *bimap* function displayed in Fig. 2 (we omit the type arguments for function composition and identity functions).

Consider the generic function

$$\mathit{poly}_{\mathsf{t}::\mathfrak{T}} :: \mathsf{Poly}_{\mathfrak{T}}\ \mathsf{Data}^1_{\mathsf{t}}\ \ldots\ \mathsf{Data}^n_{\mathsf{t}}.$$

Let $\mathit{iso}_{\mathsf{Data}_{\mathsf{t}}}$ denote $\mathit{iso\_tT}$ if $\mathsf{Data}_{\mathsf{t}} = \mathsf{Id}_{\mathsf{t}}$, and $\mathit{iso\_Data\_tT}$ otherwise. The conversion function can now be derived as

$$\mathit{conv\_poly\_t} = \mathit{to}\ (\mathit{bimap}_{\mathsf{Poly}_\star}\ \mathit{iso}_{\mathsf{Data}^1_{\mathsf{t}}}\ \ldots\ \mathit{iso}_{\mathsf{Data}^n_{\mathsf{t}}}).$$

$$
\begin{array}{ll}
\mathsf{Bimap}_{\mathfrak{T}::\square} & :: \mathfrak{Id}_{\mathfrak{T}} \to \mathfrak{Id}_{\mathfrak{T}} \to \star \\
\mathsf{Bimap}_{\star} & = \Lambda \mathsf{t}_1 \,.\, \Lambda \mathsf{t}_2 \,.\, \mathsf{Iso}\ \mathsf{t}_1\ \mathsf{t}_2 \\
bimap_{\mathsf{t}::\mathfrak{T}} & :: \mathsf{Bimap}_{\mathfrak{T}}\ \mathsf{Id}_{\mathsf{t}}\ \mathsf{Id}_{\mathsf{t}} \\
bimap_{1} & = Iso\ id\ id \\
bimap_{\mathsf{Char}} & = Iso\ id\ id \\
bimap_{+} & = \lambda \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}} \,.\, \lambda \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}} \,. \\
 & \quad Iso\ (\lambda ab \to \mathbf{case}\ ab\ \mathbf{of} \\
 & \qquad \{ Inl\ \ a \to (Inl \cdot from\ \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}})\ a; \\
 & \qquad\ \ Inr\ b \to (Inr \cdot from\ \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}})\ b \}) \\
 & \qquad (\lambda ab \to \mathbf{case}\ ab\ \mathbf{of} \\
 & \qquad \{ Inl\ \ a \to (Inl \cdot to\ \ \ \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}})\ a; \\
 & \qquad\ \ Inr\ b \to (Inr \cdot to\ \ \ \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}})\ b \}) \\
bimap_{\times} & = \lambda \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}} \,.\, \lambda \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}} \,. \\
 & \quad Iso\ (\lambda(a,b) \to (from\ \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}}\ a, from\ \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}}\ b)) \\
 & \qquad (\lambda(a,b) \to (to\ \ \ \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}}\ a, to\ \ \ \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}}\ b)) \\
bimap_{\to} & = \lambda \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}} \,.\, \lambda \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}} \,. \\
 & \quad Iso\ (\lambda ab \to from\ \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}} \cdot ab \cdot to\ \ \ \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}}) \\
 & \qquad (\lambda ab \to to\ \ \ \mathsf{b}_1\ \mathsf{b}_2\ bimap_{\mathsf{b}} \cdot ab \cdot from\ \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}}) \\
bimap_{\forall_\star} & = \lambda \mathsf{f}_1\ \mathsf{f}_2\ bimap_{\mathsf{f}} \,. \\
 & \quad Iso\ (\lambda f\ \mathsf{v} \,.\, from\ (\mathsf{f}_1\ \mathsf{v})\ (\mathsf{f}_2\ \mathsf{v}) \\
 & \qquad\qquad (bimap_{\mathsf{f}}\ \mathsf{v}\ \mathsf{v}\ (Iso\ id\ id))\ (f\ \mathsf{v})) \\
 & \qquad (\lambda f\ \mathsf{v} \,.\, to\ \ \ \ (\mathsf{f}_1\ \mathsf{v})\ (\mathsf{f}_2\ \mathsf{v}) \\
 & \qquad\qquad (bimap_{\mathsf{f}}\ \mathsf{v}\ \mathsf{v}\ (Iso\ id\ id))\ (f\ \mathsf{v})) \\
bimap_{c\ \mathbf{of}} & = \lambda \mathsf{a}_1\ \mathsf{a}_2\ bimap_{\mathsf{a}} \,.\, bimap_{\mathsf{a}}
\end{array}
$$

Fig. 2. Lifting isomorphisms with a generic function.

For example, the conversion function for the specialization of *lookup* to Nat is given by

$$conv\_lookup\_Nat = to\ (bimap_{\mathsf{Lookup}_\star}\ iso\_Nat\ iso\_FMap\_Nat),$$

which is extensionally the same as the function given in Section 3.

Note that the definition of *bimap* must include a case for the quantifier $\forall_\star :: (\star \to \star) \to \star$ since $\mathsf{Lookup}_\star$ is a polymorphic type. In this specific case, however, polymorphic type indices can be easily handled, see Fig. 2. The further details are exactly the same as for type-indexed values [18,20], and are omitted here.

4.6. Summary

For a Generic Haskell program including type-indexed types, the Generic Haskell compiler does the following:

- For each data type, the corresponding generic representation type is generated, together with a pair of isomorphisms.

- Each type-indexed type is translated into a series of **newtype** statements, one for each case.
- Analogously, each case of each type-indexed function is translated into one ordinary function definition.
- Finally, each call to a generic function is replaced by a call to the appropriate specialization.

It is sufficient to specialize generic functions to type constants only. Because we assign semantics of generic functions via an interpretation of the simply typed lambda calculus, the calls to generic functions where the type argument is a more complex type term can be simplified. For instance, the expression

$$lookup\langle \mathsf{List\ Char} \rangle$$

can be simplified to

$$lookup\langle \mathsf{List} \rangle\ lookup\langle \mathsf{Char} \rangle,$$

hence only the specializations of *lookup* to List and Char are required. If generic functions involve type-indexed types, then specializations for those are needed as well. The same observation holds for type-indexed types, though: specializations to type constants suffice.

It is thus obvious that the additional code size of the translated program is in the order the number of generic functions times the number of data types in the program. Careful analysis of which calls actually appear in a program can be used to reduce the number of specializations that are generated.

5. An advanced example: the Zipper

This section shows how to define a so-called zipper for an arbitrary data type. This is a more complex example demonstrating the full power of a type-indexed data structure together with a number of type-indexed functions working on it.

The zipper is a data structure that is used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down in the tree. The zipper is used in tools where a user interactively manipulates trees, for instance, in editors for structured documents such as proofs or programs. For the following it is important to note that the focus of the zipper may only move to recursive components. Consider as an example the data type Tree:

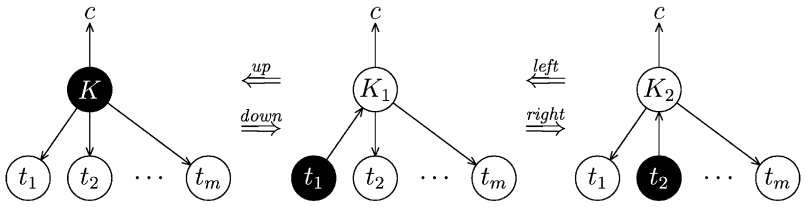
$$\textbf{data}\ \mathsf{Tree\ a} = \mathit{Empty} \mid \mathit{Node}\ (\mathsf{Tree\ a})\ \mathsf{a}\ (\mathsf{Tree\ a}).$$

If the left subtree of a *Node* constructor is the current focus, moving right means moving to the right tree, not to the a-label. This implies that recursive positions in trees play an important role in the definition of a generic zipper data structure. To obtain access to these recursive positions, we have to be explicit about the fixed points in data type definitions. The zipper data structure is then defined by induction on the so-called pattern functor of a data type.

The tools in which the zipper is used, allow the user to repeatedly apply navigation or edit commands, and to update the focus accordingly. In this section we define a type-indexed data type for locations, which consist of a subtree (the focus) together with a context, and we define several navigation functions on locations.

5.1. The basic idea

The zipper is based on pointer reversal. If we follow a pointer to a subterm, the pointer is reversed to point from the subterm to its parent so that we can go up again later. A location is a pair (t, c) consisting of the current subterm t and a pointer c to its parent. The upward pointer corresponds to the *context* of the subterm. It can be represented as follows. For each constructor K that has m recursive subcomponents we introduce m context constructors $K_1, \ldots, K_m$. Now, consider the location $(K\ t_1\ t_2\ \ldots\ t_m, c)$. If we go down to t_1, we are left with the context $K\ \bullet\ t_2\ \ldots\ t_m$ and the old context c. To represent the combined context, we simply plug c into the hole to obtain $K_1\ c\ t_2\ \ldots\ t_m$. Thus, the new location is $(t_1, K_1\ c\ t_2\ \ldots\ t_m)$. The following picture illustrates the idea (the filled circle marks the current cursor position):



5.2. Locations

A location is a subtree, together with a context, which encodes the path from the top of the original tree to the selected subtree. The type-indexed data type Loc returns a type for locations given an argument pattern functor.

```
Loc⟨f::★ → ★⟩        :: ★
Loc⟨f⟩               = (Fix f, Context⟨f⟩ (Fix f))
Context⟨f::★ → ★⟩ :: ★ → ★
Context⟨f⟩ r         = Fix (LMaybe (Ctx⟨f⟩ r))
data LMaybe f a      = LNothing | LJust (f a),
```

where LMaybe is the lifted version of Maybe. The type Loc is defined in terms of Context, which constructs the context parameterized by the original tree type. The Context of a value is either empty (represented by *LNothing* in the LMaybe type), or it is a path from the root down into the tree. Such a path is constructed by means of the argument type of LMaybe: the type-indexed data type Ctx. The type-indexed data type Ctx is defined by induction on the pattern functor of the original data type. It can be seen as the *derivative* (as in calculus) of the pattern functor f [1,34]. If the

derivative of f is denoted by f′, we have

$$\begin{aligned}
\mathsf{const}' &= \mathsf{Void} \\
(\mathsf{f}+\mathsf{g})' &= \mathsf{f}'+\mathsf{g}' \\
(\mathsf{f}\times\mathsf{g})' &= \mathsf{f}'\times\mathsf{g}+\mathsf{f}\times\mathsf{g}'.
\end{aligned}$$

It follows that in the definition of Ctx we will also need access to the type arguments themselves on the right-hand side of the definition:

$$\begin{aligned}
&\mathsf{Ctx}\langle \mathsf{f}::\bigstar\to\bigstar\rangle \;::\; \bigstar\to\bigstar\to\bigstar \\
&\mathsf{Ctx}\langle \mathsf{Id}\rangle \quad\; \mathsf{r}\;\mathsf{c} = \mathsf{c} \\
&\mathsf{Ctx}\langle \mathsf{K}\;1\rangle \quad\; \mathsf{r}\;\mathsf{c} = \mathsf{Void} \\
&\mathsf{Ctx}\langle \mathsf{K}\;\mathsf{Char}\rangle\; \mathsf{r}\;\mathsf{c} = \mathsf{Void} \\
&\mathsf{Ctx}\langle \mathsf{f}_1+\mathsf{f}_2\rangle\; \mathsf{r}\;\mathsf{c} = \mathsf{Ctx}\langle \mathsf{f}_1\rangle\;\mathsf{r}\;\mathsf{c}+\mathsf{Ctx}\langle \mathsf{f}_2\rangle\;\mathsf{r}\;\mathsf{c} \\
&\mathsf{Ctx}\langle \mathsf{f}_1\times\mathsf{f}_2\rangle\; \mathsf{r}\;\mathsf{c} = (\mathsf{Ctx}\langle \mathsf{f}_1\rangle\;\mathsf{r}\;\mathsf{c}\times\mathsf{f}_2\;\mathsf{r})+(\mathsf{f}_1\;\mathsf{r}\times\mathsf{Ctx}\langle \mathsf{f}_2\rangle\;\mathsf{r}\;\mathsf{c}).
\end{aligned}$$

This definition can be understood as follows. Since it is not possible to descend into a constant, the constant cases do not contribute to the result type, which is denoted by the 'empty type' Void, a type without values. The Id case denotes a recursive component, in which it is possible to descend. Hence it may occur in a context. Descending in a value of a sum type follows the structure of the input value. Finally, there are two ways to descend in a product: descending left, adding the contents to the right of the node to the context, or descending right, adding the contents to the left of the node to the context.

For example, for natural numbers with pattern functor $\mathsf{K}\;1+\mathsf{Id}$, and for trees of type Bush with pattern functor BushF, which can be represented by $\mathsf{K}\;\mathsf{Char}+(\mathsf{Id}\times\mathsf{Id})$ we obtain

$$\begin{aligned}
&\mathsf{Context}\langle \mathsf{K}\;1+\mathsf{Id}\rangle \quad\; \mathsf{r} = \mathsf{Fix}\;(\mathsf{LMaybe}\;(\mathsf{NatC}\;\;\mathsf{r})) \\
&\mathsf{Context}\langle \mathsf{K}\;\mathsf{Char}+\mathsf{Id}\times\mathsf{Id}\rangle\; \mathsf{r} = \mathsf{Fix}\;(\mathsf{LMaybe}\;(\mathsf{BushC}\;\mathsf{r})) \\
&\textbf{data}\;\mathsf{NatC}\;\;\mathsf{r}\;\mathsf{c} \quad\quad\quad\quad = \mathit{ZeroC}\;\mathsf{Void}\mid \mathit{SuccC}\;\mathsf{c} \\
&\textbf{data}\;\mathsf{BushC}\;\mathsf{r}\;\mathsf{c} \quad\quad\quad\quad = \mathit{LeafC}\;\mathsf{Void}\mid \mathit{ForkCL}\;(\mathsf{c},\mathsf{r})\mid \mathit{ForkCR}\;(\mathsf{r},\mathsf{c}).
\end{aligned}$$

Note that the context of a natural number is isomorphic to a natural number (the context of m in n is $n-m$), and the context of a Bush applied to the data type Bush itself is isomorphic to the type Context_Bush introduced in Section 1.

McBride [1,34] also defines a type-indexed zipper data type. His zipper slightly deviates from Huet's and our zipper: the navigation functions on McBride's zipper are not constant time anymore. The observation that the Context of a data type is its derivative (as in calculus) is due to McBride.

5.3. Navigation functions

We define type-indexed functions on the type-indexed data types Loc, Context, and Ctx for navigating through a tree. All of these functions act on locations. These are the basic functions for the zipper.

Function *down*. The function *down* is a type-indexed function that moves down to the leftmost recursive child of the current node, if such a child exists. Otherwise, if the current node is a leaf node, then *down* returns the location unchanged.

$$down\langle \mathsf{f}::\bigstar \to \bigstar\rangle::\mathsf{Loc}\langle \mathsf{f}\rangle \to \mathsf{Loc}\langle \mathsf{f}\rangle$$

The instantiation of *down* to the data type Bush has been given in Section 1. The function *down* satisfies the following property:

$$\forall m\,.\,down\langle \mathsf{f}\rangle\ m \neq m \implies (up\langle \mathsf{f}\rangle \cdot down\langle \mathsf{f}\rangle)\ m = m,$$

where the function *up* goes up in a tree. So first going down the tree and then up again is the identity function on locations in which it is possible to go down.

Since *down* moves down to the leftmost recursive child of the current node, the inverse equality $down\langle \mathsf{f}\rangle \cdot up\langle \mathsf{f}\rangle = id$ does not hold in general. However, there does exist a natural number n such that

$$\forall m\,.\,up\langle \mathsf{f}\rangle\ m \neq m \implies (right\langle \mathsf{f}\rangle^n \cdot down\langle \mathsf{f}\rangle \cdot up\langle \mathsf{f}\rangle)\ m = m,$$

where the function *right* goes right in a tree. These properties do not completely specify function *down*. The other properties it should satisfy are that the selected subtree of $down\langle \mathsf{f}\rangle\ m$ is the leftmost tree-child of the selected subtree of m, and the context of $down\langle \mathsf{f}\rangle\ m$ is the context of m extended with all but the leftmost tree-child of m.

The function *down* is defined as follows:

$$\begin{aligned} down\langle \mathsf{f}\rangle\ (t,c) = \mathbf{case}\ first\langle \mathsf{f}\rangle\ (out\ t)\ c\ \mathbf{of} \\ \{\ Just\ (t',c') &\to (t', In\ (LJust\ c'));\\ Nothing &\to (t,c)\ \}. \end{aligned}$$

To find the leftmost recursive child, we have to pattern match on the pattern functor f, and find the first occurrence of Id. The helper function *first* is a type-indexed function that possibly returns the leftmost recursive child of a node, together with the context (a value of type $\mathsf{Ctx}\langle \mathsf{f}\rangle\ \mathsf{c}\ \mathsf{t}$) of the selected child. The function *down* then turns this context into a value of type Context by inserting it in the right ('non-top') component of a sum by means of *LJust*, and applying the fixed point constructor *In* to it.

$$\begin{aligned} &first\langle \mathsf{f}::\bigstar \to \bigstar\rangle && :: \forall \mathsf{c}\ \mathsf{t}\,.\,\mathsf{f}\ \mathsf{t} \to \mathsf{c} \to \mathsf{Maybe}\ (\mathsf{t}, \mathsf{Ctx}\langle \mathsf{f}\rangle\ \mathsf{c}\ \mathsf{t})\\ &first\langle \mathsf{Id}\rangle\ t\ c &&= return\ (t,c)\\ &first\langle \mathsf{K}\ 1\rangle\ t\ c &&= Nothing\\ &first\langle \mathsf{K}\ \mathsf{Char}\rangle\ t\ c &&= Nothing\\ &first\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ (Inl\ x)\ c &&= \mathbf{do}\ \{(t,cx) \leftarrow first\langle \mathsf{f}_1\rangle\ x\ c; return\ (t, Inl\ cx)\}\\ &first\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ (Inr\ y)\ c &&= \mathbf{do}\ \{(t,cy) \leftarrow first\langle \mathsf{f}_2\rangle\ y\ c; return\ (t, Inr\ cy)\}\\ &first\langle \mathsf{f}_1 \times \mathsf{f}_2\rangle\ (x,y)\ c &&= \mathbf{do}\ \{(t,cx) \leftarrow first\langle \mathsf{f}_1\rangle\ x\ c;\\ &&&\qquad return\ (t, Inl\ (cx,y))\}\\ &&&+\!\!+\ \mathbf{do}\ \{(t,cy) \leftarrow first\langle \mathsf{f}_2\rangle\ y\ c;\\ &&&\qquad return\ (t, Inr\ (x,cy))\}. \end{aligned}$$

Here, *return* is obtained from the Maybe monad, and the operator $(+\!\!+)$ is the standard monadic plus, called *mplus* in Haskell, given by

$$
\begin{array}{ll}
(+\!\!+) & :: \forall \mathsf{a}\,.\,\mathsf{Maybe\ a} \to \mathsf{Maybe\ a} \to \mathsf{Maybe\ a} \\
\mathit{Nothing} +\!\!+ m & = m \\
\mathit{Just}\ a \quad +\!\!+ m & = \mathit{Just}\ a.
\end{array}
$$

The function *first* returns the value and the context at the leftmost Id position. So in the product case, it first tries the left component, and only if it fails, it tries the right component.

The definitions of functions *up*, *right* and *left* are not as simple as the definition of *down*, since they are defined by pattern matching on the context instead of on the tree itself. We will just define functions *up* and *right*, and leave function *left* as an exercise.

Function *up*. The function *up* moves up to the parent of the current node, if the current node is not the top node:

$$
\begin{array}{lll}
\mathit{up}\langle \mathsf{f}::\star \to \star \rangle & :: \mathsf{Loc}\langle \mathsf{f} \rangle \to \mathsf{Loc}\langle \mathsf{f} \rangle & \\
\mathit{up}\langle \mathsf{f} \rangle\ (t, c) & = \mathbf{case}\ \mathit{out}\ c\ \mathbf{of} & \\
& \quad \{\mathit{LNothing} \to (t, c); & \\
& \quad\ \mathit{LJust}\ c' \ \to \mathbf{do}\ \{\mathit{ft} \leftarrow \mathit{insert}\langle \mathsf{f} \rangle\ c'\ t; & \\
& \qquad\qquad\qquad\quad c'' \leftarrow \mathit{extract}\langle \mathsf{f} \rangle\ c'; & \\
& \qquad\qquad\qquad\quad \mathit{return}\ (\mathit{In}\ \mathit{ft}, c'')\}\}. &
\end{array}
$$

Remember that *LNothing* denotes the empty top context. The navigation function *up* uses two helper functions: *insert* and *extract*. The latter returns the context of the parent of the current node. Note that each element of type $\mathsf{Ctx}\langle \mathsf{f} \rangle\ \mathsf{c}\ \mathsf{t}$ has at most one c component (by an easy inductive argument), which marks the context of the parent of the current node. The generic function *extract* extracts this context:

$$
\begin{array}{lll}
\mathit{extract}\langle \mathsf{f}::\star \to \star \rangle & :: \forall \mathsf{c}\ \mathsf{t}\,.\,\mathsf{Ctx}\langle \mathsf{f} \rangle\ \mathsf{c}\ \mathsf{t} \to \mathsf{Maybe\ c} \\
\mathit{extract}\langle \mathsf{Id} \rangle\ c & = \mathit{return}\ c \\
\mathit{extract}\langle \mathsf{K\ 1} \rangle\ c & = \mathit{Nothing} \\
\mathit{extract}\langle \mathsf{K\ Char} \rangle\ c & = \mathit{Nothing} \\
\mathit{extract}\langle \mathsf{f}_1 + \mathsf{f}_2 \rangle\ (\mathit{Inl}\ cx) & = \mathit{extract}\langle \mathsf{f}_1 \rangle\ cx \\
\mathit{extract}\langle \mathsf{f}_1 + \mathsf{f}_2 \rangle\ (\mathit{Inr}\ cy) & = \mathit{extract}\langle \mathsf{f}_2 \rangle\ cy \\
\mathit{extract}\langle \mathsf{f}_1 \times \mathsf{f}_2 \rangle\ (\mathit{Inl}\ (cx, y)) & = \mathit{extract}\langle \mathsf{f}_1 \rangle\ cx \\
\mathit{extract}\langle \mathsf{f}_1 \times \mathsf{f}_2 \rangle\ (\mathit{Inr}\ (x, cy)) & = \mathit{extract}\langle \mathsf{f}_2 \rangle\ cy.
\end{array}
$$

Note that *extract* is polymorphic in c and in t.

Function *insert* takes a context and a tree, and inserts the tree in the current focus of the context, effectively turning a context into a tree:

$$
\begin{array}{lll}
\mathit{insert}\langle \mathsf{f}::\star \to \star \rangle & :: \forall \mathsf{c}\ \mathsf{t}\,.\,\mathsf{Ctx}\langle \mathsf{f} \rangle\ \mathsf{c}\ \mathsf{t} \to \mathsf{t} \to \mathsf{Maybe\ (f\ t)} \\
\mathit{insert}\langle \mathsf{Id} \rangle\ c & t = \mathit{return}\ t \\
\mathit{insert}\langle \mathsf{K\ 1} \rangle\ c & t = \mathit{Nothing} \\
\mathit{insert}\langle \mathsf{K\ Char} \rangle\ c & t = \mathit{Nothing}
\end{array}
$$

$$
\begin{array}{lll}
insert\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ (Inl\ cx) & t = \mathbf{do}\ \{x \leftarrow insert\langle \mathsf{f}_1\rangle\ cx\ t; return\ (Inl\ x)\} \\
insert\langle \mathsf{f}_1 + \mathsf{f}_2\rangle\ (Inr\ cy) & t = \mathbf{do}\ \{y \leftarrow insert\langle \mathsf{f}_2\rangle\ cy\ t; return\ (Inr\ y)\} \\
insert\langle \mathsf{f}_1 \times \mathsf{f}_2\rangle\ (Inl\ (cx, y)) & t = \mathbf{do}\ \{x \leftarrow insert\langle \mathsf{f}_1\rangle\ cx\ t; return\ (x, y)\} \\
insert\langle \mathsf{f}_1 \times \mathsf{f}_2\rangle\ (Inr\ (x, cy)) & t = \mathbf{do}\ \{y \leftarrow insert\langle \mathsf{f}_2\rangle\ cy\ t; return\ (x, y)\}.
\end{array}
$$

Note that the extraction and insertion is happening in the identity case Id; the other cases only pass on the results.

Since $up\langle \mathsf{f}\rangle \cdot down\langle \mathsf{f}\rangle = id$ on locations in which it is possible to go down, we expect similar equalities for the functions *first*, *extract*, and *insert*. We have that the following computation

$$
\begin{array}{l}
\mathbf{do}\ \{(t, c') \leftarrow first\langle \mathsf{f}\rangle\ ft\ c; \\
\quad c'' \leftarrow extract\langle \mathsf{f}\rangle\ c'; \\
\quad ft' \leftarrow insert\langle \mathsf{f}\rangle\ c'\ t; \\
\quad return\ (c == c'' \wedge ft == ft')\}
\end{array}
$$

returns *True* on locations in which it is possible to go down.

Function *right*. The function *right* moves the focus to the next (right) sibling in a tree, if it exists. The context is moved accordingly. The instance of *right* on the data type Bush has been given in Section 1. The function *right* satisfies the following property:

$$\forall m.\, right\langle \mathsf{f}\rangle\ m \neq m \implies (left\langle \mathsf{f}\rangle \cdot right\langle \mathsf{f}\rangle)\ m = m,$$

that is, first going right in the tree and then left again is the identity function on locations in which it is possible to go to the right. Of course, the dual equality holds on locations in which it is possible to go to the left. Furthermore, the selected subtree of $right\langle \mathsf{f}\rangle\ m$ is the sibling to the right of the selected subtree of m, and the context of $right\langle \mathsf{f}\rangle\ m$ is the context of m in which the context is replaced by the selected subtree of m, and the first subtree to the right of the context of m is replaced by the context of m.

Function *right* is defined by pattern matching on the context. It is impossible to go to the right at the top of a tree. Otherwise, we try to find the right sibling of the current focus.

$$
\begin{array}{l}
right\langle \mathsf{f}::\bigstar \to \bigstar\rangle :: \mathsf{Loc}\langle \mathsf{f}\rangle \to \mathsf{Loc}\langle \mathsf{f}\rangle \\
right\langle \mathsf{f}\rangle\ (t, c) = \mathbf{case}\ out\ c\ \mathbf{of} \\
\qquad \{LNothing \to (t, c); \\
\qquad\ LJust\ c' \to \mathbf{case}\ next\langle \mathsf{f}\rangle\ t\ c'\ \mathbf{of} \\
\qquad\qquad \{Just\ (t', c'') \to (t', In\ (LJust\ c'')); \\
\qquad\qquad\ Nothing \to (t, c)\}\}
\end{array}
$$

The helper function *next* is a type-indexed function that returns the first location that has the recursive value to the right of the selected value as its focus. Just as there exists a function *left* such that $left\langle \mathsf{f}\rangle \cdot right\langle \mathsf{f}\rangle = id$ (on locations in which it is possible to go to the right), there exists a function *previous*, such that

$$
\begin{array}{l}
\mathbf{do}\ \{(t', c') \leftarrow next\langle \mathsf{f}\rangle\ t\ c;
\end{array}
$$

$$(t'', c'') \leftarrow previous\langle \mathsf{f} \rangle\ t'\ c';$$
$$return\ (c == c'' \wedge t == t'')\}$$

returns *True* (on locations in which it is possible to go to the right). We will define function *next*, and omit the definition of function *previous*:

$$\begin{aligned}
&next\langle \mathsf{f}::\star \to \star \rangle::\forall \mathsf{c\ t}.\mathsf{t} \to \mathsf{Ctx}\langle \mathsf{f} \rangle\ \mathsf{c\ t} \to \mathsf{Maybe}\ (\mathsf{t}, \mathsf{Ctx}\langle \mathsf{f} \rangle\ \mathsf{c\ t})\\
&next\langle \mathsf{Id} \rangle\ \ t\ c = Nothing\\
&next\langle \mathsf{K\ 1} \rangle\ \ t\ c = Nothing\\
&next\langle \mathsf{K\ Char} \rangle\ t\ c = Nothing\\
&next\langle \mathsf{f}_1 + \mathsf{f}_2 \rangle\ \ t\ (Inl\ cx)\\
&\quad = \mathbf{do}\ \{(t', cx') \leftarrow next\langle \mathsf{f}_1 \rangle\ t\ cx; return\ (t', Inl\ cx')\}\\
&next\langle \mathsf{f}_1 + \mathsf{f}_2 \rangle\ \ t\ (Inr\ cy)\\
&\quad = \mathbf{do}\ \{(t', cy') \leftarrow next\langle \mathsf{f}_2 \rangle\ t\ cy; return\ (t', Inr\ cy')\}\\
&next\langle \mathsf{f}_1 \times \mathsf{f}_2 \rangle\ t\ (Inl\ (cx, y))\\
&\quad = \mathbf{do}\ \{(t', cx') \leftarrow next\langle \mathsf{f}_1 \rangle\ t\ cx; return\ (t', Inl\ (cx', y))\}\\
&\quad +\!\!+ \mathbf{do}\ \{c \leftarrow extract\langle \mathsf{f}_1 \rangle\ cx;\\
&\qquad\quad x \leftarrow insert\langle \mathsf{f}_1 \rangle\ cx\ t;\\
&\qquad\quad (t', cy) \leftarrow first\langle \mathsf{f}_2 \rangle\ y\ c;\\
&\qquad\quad return\ (t', Inr\ (x, cy))\}\\
&next\langle \mathsf{f}_1 \times \mathsf{f}_2 \rangle\ \ t\ (Inr\ (x, cy))\\
&\quad = \mathbf{do}\ \{(t', cy') \leftarrow next\langle \mathsf{f}_2 \rangle\ t\ cy; return\ (t', Inr\ (x, cy'))\}.
\end{aligned}$$

The first three lines in this definition show that it is impossible to go to the right in an identity or constant context. If the context argument is a value of a sum, we select the next element in the appropriate component of the sum. The product case is the most interesting one. If the context is in the right component of a pair, *next* returns the next value of that context, properly combined with the left component of the tuple. On the other hand, if the context is in the left component of a pair, the next value may be either in that left component (the context), or it may be in the right component (the value). If the next value is in the left component, it is returned by the first line in the definition of the product case. If it is not, *next* extracts the context c (the context of the parent) from the left context cx, it inserts the given value in the context cx giving a 'tree' value x, and selects the first component in the right component of the pair, using the extracted context c for the new context. The new context that is thus obtained is combined with x into a context for the selected tree.

6. Conclusion

We have shown how to define type-indexed data types, and we have given several examples of type-indexed data types: digital search trees, generic pattern-matching using a labelled data type, and the zipper. Furthermore, we have shown how to specialize type-indexed data types and type-indexed functions that take values of type-indexed data types as arguments. The treatment generalizes the specialization of type-

indexed functions given in Hinze [17], and used in the implementation of Generic Haskell, a generic programming extension of the functional language Haskell, see http://www.generic-haskell.org/. A technical overview of the compiler can be found in De Wit's thesis [43]. The current release of Generic Haskell contains an experimental implementation of type-indexed data types. The syntax for type-indexed types used in the Generic Haskell compiler differs from the syntax used in this paper in a few places. There is a tutorial by Hinze and Jeuring [19] that explains the syntax used in the implementation.

A type-indexed data type is defined in a similar way as a type-indexed function. The only difference is that the 'type' of a type-indexed data type is a kind instead of a type. Note that a type-indexed data type may also be a type constructor, it need not necessarily be a type of kind ★. For instance, Label is indexed by types of kind $\star \to \star$ and yields types of kind $\star \to \star \to \star$.

The approach taken in this paper is powerful enough to be used for sets of mutually recursive type-indexed data types. Hagg [13] uses mutually recursive type-indexed data types to specify data types with holes, for use in a generic editor.

Acknowledgements

Thanks are due to Dave Clarke, Ralf Lämmel, Doaitse Swierstra, and the anonymous referees for comments on previous versions of the paper. Jan de Wit suggested an improvement in the labelling functions.

References

[1] M. Abott, T. Altenkirch, N. Ghani, C. McBride, Derivatives of containers, in: Typed Lambda Calculi and Applications, TLCA 2003, Lecture Notes in Computer Science, vol. 2701, Springer, Berlin, 2003, pp. 16–30.

[2] L. Augustsson, D. Barton, B. Boutel, W. Burton, S. Fraser, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, P. Wadler, in: S.P. Jones J. Hughes (Eds.), Haskell 98—A Non-Strict, Purely Functional Language (February 1999), available from http://www.haskell.org/definition/.

[3] R. Backhouse, P. Jansson, J. Jeuring, L. Meertens, Generic programming: an introduction, in: S.D. Swierstra, P.R. Henriques, J.N. Oliveira (Eds.), Advanced Functional Programming, Lecture Notes in Computer Science, vol. 1608, Springer, Berlin, 1999, pp. 28–115.

[4] R. Bird, O. de Moor, P. Hoogendijk, Generic functional programming with types and relations, J. Funct. Program. 6 (1) (1996) 1–28.

[5] M.M.T. Chakravarty, G. Keller, More types for nested data parallel programming, in: Proc. ICFP 2000: Internat. Conf. on Functional Programming, ACM Press, New York, 2000, pp. 94–105.

[6] K. Claessen, P. Ljunglöf, Typed logical variables in Haskell, in: Proc. Haskell Workshop 2000, 2000.

[7] D. Clarke, A. Löh, Generic Haskell, specifically, in: J. Gibbons, J. Jeuring (Eds.), Generic Programming, IFIP, vol. 243, Kluwer Academic Publishers, Dordrecht, 2003, pp. 21–48.

[8] K. Crary, S. Weirich, Flexible type analysis, in: Proc. ICFP 1999: Internat. Conf. on Functional Programming, ACM Press, New York, 1999, pp. 233–248, URL citeseer.nj.nec.com/crary99flexible.html

[9] K. Crary, S. Weirich, J.G. Morrisett, Intensional polymorphism in type-erasure semantics, in: Proc. ICFP 1998: Internat. Conf. on Functional Programming, ACM Press, New York, 1998, pp. 301–312.

[10] C. Dubois, F. Rouaix, P. Weis, Extensional polymorphism, in: 22nd Symp. on Principles of Programming Languages, POPL '95, 1995, pp. 118–129.
[11] M. Fokkinga, Law and order in algorithmics, Ph.D. Thesis, Dept. INF, University of Twente, Enschede, The Netherlands, 1992.
[12] J. Gibbons, Polytypic downwards accumulations, in: Proc. of Mathematics of Program Construction, Lecture Notes in Computer Science, vol. 1422, Springer, Berlin, 1998, pp. 207–233.
[13] P. Hagg, A framework for developing generic XML Tools, Master's Thesis, Department of Information and Computing Sciences, Utrecht University, 2002.
[14] R. Harper, G. Morrisett, Compiling polymorphism using intensional type analysis, in: 22nd Symp. Principles of Programming Languages, POPL '95, 1995, pp. 130–141.
[15] R. Hinze, Generalizing generalized tries, J. Funct. Program. 10(4) (2000) 327–351, URL citeseer.nj.nec.com/hinze99generalizing.html
[16] R. Hinze, A new approach to generic functional programming, in: Conf. Record of POPL '00: The 27th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, ACM Press, New York, 2000, pp. 119–132, URL citeseer.nj.nec.com/hinze99new.html
[17] R. Hinze, Polytypic values possess polykinded types, in: R. Backhouse, J.N. Oliveira (Eds.), Mathematics of Program Construction, Lecture Notes in Computer Science, vol. 1837, Springer, Berlin, 2000, pp. 2–27.
[18] R. Hinze, Generic Programs and Proofs, Habilitationsschrift, Bonn University, 2000.
[19] R. Hinze, J. Jeuring, Generic Haskell: Applications, in: R. Backhouse, J. Gibbons (Eds.), Generic Programming, Lecture Notes in Computer Science, vol. 2793, Springer, Berlin, 2003, pp. 57–97.
[20] R. Hinze, S. P. Jones, Derivable type classes, in: G. Hutton (Ed.), Proc. 2000 ACM SIGPLAN Haskell Workshop, Electronic Notes in Theoretical Computer Science, vol. 41.1, Elsevier, Amsterdam, 2001, the preliminary proc. appeared as a University of Nottingham technical report.
[21] R. Hinze, J. Jeuring, A. Löh, Type-indexed data types, in: Proc. 6th Mathematics of Program Construction Conf. MPC'02, Lecture Notes in Computer Science, vol. 2386, 2002, pp. 148–174.
[22] G. Huet, The zipper, J. Funct. Program. 7 (5) (1997) 549–554.
[23] G. Huet, Linear contexts and the sharing functor: techniques for symbolic computation, in: F. Kamareddine (Ed.), Thirty Five Years of Automating Mathematics, Kluwer, Dordrecht, 2003.
[24] P. Jansson, The WWW home page for polytypic programming, available from http://www.cs.chalmers.se/~patrikj/poly/ (2001).
[25] P. Jansson, J. Jeuring, PolyP—a polytypic programming language extension, in: Conf. Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, ACM Press, New York, 1997, pp. 470–482.
[26] P. Jansson, J. Jeuring, Functional pearl: polytypic unification, J. Funct. Program. 8 (5) (1998) 527–536.
[27] P. Jansson, J. Jeuring, A framework for polytypic programming on terms, with an application to rewriting, in: J. Jeuring (Ed.), Workshop on Generic Programming 2000, Ponte de Lima, Portugal, July 2000, 2000, pp. 33–45; Utrecht Technical Report UU-CS-2000-19.
[28] C. Jay, G. Bellè, E. Moggi, Functorial ML, J. Funct. Program. 8 (6) (1998) 573–619.
[29] J. Jeuring, Polytypic pattern matching, in: Conference Record of FPCA'95, SIGPLAN-SIGARCH-WG2.8 Conf. on Functional Programming Languages and Computer Architecture, ACM Press, New York, 1995, pp. 238–248.
[30] M.P. Jones, Type classes with functional dependencies, in: G. Smolka (Ed.), Proc. 9th European Symp. on Programming, ESOP 2000, Berlin, Germany, Lecture Notes in Computer Science, vol. 1782, Springer, Berlin, 2000, pp. 230–244.
[31] D. Knuth, J. Morris, V. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1978) 323–350.
[32] R. Lämmel, W. Lohmann, Format Evolution, in: J. Kouloumdjian, H. Mayr, A. Erkollar (Eds.), Proc. 7th Internat. Conf. on Reverse Engineering for Information Systems (RETIS 2001), books@ocg.at, vol. 155, OCG, 2001, pp. 113–134.
[33] G. Malcolm, Data structures and program transformation, Sci. Comput. Program. 14 (1990) 255–279.
[34] C. McBride, The derivative of a regular type is its type of one-hole contexts, 2001, unpublished manuscript.
[35] N.J. McCracken, An investigation of a programming language with a polymorphic type structure, Ph.D. Thesis, Syracuse University, 1979.

[36] L. Meertens, Paramorphisms, Formal Aspects Comput. 4 (5) (1992) 413–425.
[37] E. Meijer, M. Fokkinga, R. Paterson, Functional programming with bananas, lenses, envelopes, and barbed wire, in: J. Hughes (Ed.), FPCA'91: Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, vol. 523, Springer, Berlin, 1991, pp. 124–144.
[38] J.C. Mitchell, Foundations for Programming Languages, The MIT Press, Cambridge, MA, 1996.
[39] V. Trifonov, B. Saha, Z. Shao, Fully reflexive intensional type analysis, in: Proc. ICFP 2000: Internat. Conf. on Functional Programming, ACM Press, New York, 2000, pp. 82–93, URL citeseer.nj.nec.com/saha00fully.html
[40] M. Vestin, Genetic algorithms in Haskell with polytypic programming, Examensarbeten 1997:36, Göteborg University, Gothenburg, Sweden, available from the Polytypic programming WWW page [23] (1997).
[41] S. Weirich, Encoding intensional type analysis, in: European Symp. on Programming, Lecture Notes in Computer Science, vol. 2028, Springer, Berlin, 2001, pp. 92–106, URL http://link.springer.de/link/service/series/0558/tocs/t%2028.htm
[42] S. Weirich, Higher-order intensional type analysis, in: D. Le Métayer (Ed.), Proc. 11th European Symp. on Programming, ESOP 2002, Lecture Notes in Computer Science, vol. 2305, 2002, pp. 98–114.
[43] J. de Wit, A technical overview of Generic Haskell, Master's Thesis, Department of Information and Computing Sciences, Utrecht University, 2002.
[44] Z. Yang, Encoding types in ML-like languages, in: Proc. ICFP 1998: Internat. Conf. on Functional Programming, ACM Press, New York, 1998, pp. 289–300, URLciteseer.nj.nec.com/zhe99encoding.html