Type-indexed data types

Ralf Hinze[1,2], Johan Jeuring[2,3], and Andres Löh[2]

[1] Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
`ralf@informatik.uni-bonn.de`
`http://www.informatik.uni-bonn.de/~ralf/`
[2] Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{`ralf,johanj,andres`}`@cs.uu.nl`
`http://www.cs.uu.nl/~`{`ralf,johanj,andres`}`/`
[3] Open University, Heerlen, The Netherlands

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.

1 Introduction

A polytypic (or generic, 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 [32], such as *show*, *read*, and '`==`'. See Backhouse et al [1] for an introduction to polytypic programming.

More advanced examples of polytypic functions are functions for digital searching [12], pattern matching [23], unification [20, 4], and rewriting [21]. 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 programming languages do not support type-indexed data types, the examples that appear in the literature are either implemented in an ad-hoc fashion [20], or not implemented at all [12].

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 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. The current version of Generic Haskell can be obtained from `http://www.generic-haskell.org/`.

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[4]

$$\mathbf{data}\ \mathit{String} = \mathit{nil} \mid \mathit{cons}\ \mathit{Char}\ \mathit{String}.$$

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

$$\mathbf{data}\ \mathit{FMap\_String}\ V = \mathit{trie\_String}\ (\mathit{Maybe}\ V)\ (\mathit{FMap\_Char}\ (\mathit{FMap\_String}\ V)).$$

Such a trie for strings would typically be used for a concordance or another 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 $\mathit{cons} :: \mathit{Char} \to \mathit{String} \to \mathit{String}$. We assume that a suitable data structure, *FMap_Char*, and an associated look-up function $\mathit{lookup\_Char} :: \forall V\,.\,\mathit{Char} \to \mathit{FMap\_Char}\ V \to \mathit{Maybe}\ V$ for characters are predefined. Given these prerequisites we can define a look-up function for strings as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{lookup\_String} :: \mathit{String} \to \mathit{FMap\_String}\ V \to \mathit{Maybe}\ V} \\
\mathit{lookup\_String}\ \mathit{nil}\ (\mathit{trie\_String}\ \mathit{tn}\ \mathit{tc}) & = \mathit{tn} \\
\mathit{lookup\_String}\ (\mathit{cons}\ c\ s)\ (\mathit{trie\_String}\ \mathit{tn}\ \mathit{tc}) & = (\mathit{lookup\_Char}\ c \diamond \mathit{lookup\_String}\ s)\ \mathit{tc}.
\end{array}$$

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

$$\begin{array}{l}
(\diamond) :: (A \to \mathit{Maybe}\ B) \to (B \to \mathit{Maybe}\ C) \to A \to \mathit{Maybe}\ C \\
(f \diamond g)\ a = \mathbf{case}\ f\ a\ \mathbf{of}\ \{\mathit{nothing} \to \mathit{nothing}; \mathit{just}\ b \to g\ b\}.
\end{array}$$

In the following section we will show how to define a trie and an associated look-up function for an arbitrary data type. The material is taken from Hinze [12], and it is repeated here because it serves as a nice and simple example of a type-indexed data type.

[4] The examples are given in Haskell [32]. Deviating from Haskell we use identifiers starting with an upper case letter for types (this includes type variables), and identifiers starting with a lower case letter for values (this includes data constructors).

Example 2: Pattern matching. The polytypic functions for the maximum segment sum problem [2] and pattern matching [23] use labelled data types. These labelled data types, introduced in [2], are 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

$$\begin{aligned}\textbf{data}\ \mathit{Lab\_Bush}\ L = {} & \mathit{label\_Leaf}\ \mathit{Char}\ L \\ \mid {} & \mathit{label\_Fork}\ (\mathit{Lab\_Bush}\ L)\ (\mathit{Lab\_Bush}\ L)\ L.\end{aligned}$$

In the following section we show how to define such a labelled data type generically.

Example 3: Zipper. The zipper [17] 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 data type *Bush* and its corresponding zipper, called *Loc_Bush*, are defined by

$$\begin{aligned}\textbf{data}\ \mathit{Bush} &= \mathit{leaf}\ \mathit{Char} \mid \mathit{fork}\ \mathit{Bush}\ \mathit{Bush} \\ \textbf{type}\ \mathit{Loc\_Bush} &= (\mathit{Bush}, \mathit{Context\_Bush}) \\ \textbf{data}\ \mathit{Context\_Bush} &= \mathit{top} \\ &\mid \mathit{forkL}\ \mathit{Context\_Bush}\ \mathit{Bush} \\ &\mid \mathit{forkR}\ \mathit{Bush}\ \mathit{Context\_Bush}.\end{aligned}$$

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

$$\begin{aligned}&\mathit{down\_Bush} && :: \mathit{Loc\_Bush} \to \mathit{Loc\_Bush} \\ &\mathit{down\_Bush}\ (\mathit{leaf}\ a, c) && = (\mathit{leaf}\ a, c) \\ &\mathit{down\_Bush}\ (\mathit{fork}\ \mathit{tl}\ \mathit{tr}, c) && = (\mathit{tl}, \mathit{forkL}\ c\ \mathit{tr}) \\ &\mathit{right\_Bush} && :: \mathit{Loc\_Bush} \to \mathit{Loc\_Bush} \\ &\mathit{right\_Bush}\ (\mathit{tl}, \mathit{forkL}\ c\ \mathit{tr}) && = (\mathit{tr}, \mathit{forkR}\ \mathit{tl}\ c) \\ &\mathit{right\_Bush}\ l && = l.\end{aligned}$$

The navigator 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 [17] 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 [3, 11, 10, 34]. We expect that type-indexed data types will also be useful for generic DTD transformations [25]. Generally, we believe that type-indexed data types are just as important as type-indexed functions.

Background and related work. There is little related work on type-indexed data types. Type-indexed functions [26, 2, 29, 9, 18] were introduced more than a decade ago. There are several other approaches to type-indexed functions, see Dubois et al [8], Jay et al [22] and Yang [36], 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 [11, 6, 5, 33, 35]. 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;
– typed intermediate languages are restricted to data types of kind $\star$. There are many examples where we want to lift this restriction, and define type-indexed data types also on higher-order kinded 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 [14, 15]. Section 3 illustrates the process of specialization by means of example. Section 4 shows that type-indexed data types possess kind-indexed kinds, and gives a 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 [28]. For a more thorough treatment the reader is referred to Hinze's work [15, 14].

2.1 Type-indexed definitions

The central idea of polytypic programming (or type-indexed 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 primitive types and type constructors. For concreteness, let us assume that 1, *Char*, '+', and '×' are primitive, that is, the language of types of kind $\star$ is defined by the following grammar:

$$T_\star ::= \ 1 \mid \mathit{Char} \mid T_\star + T_\star \mid T_\star \times T_\star .$$

The unit type, sum and product types are required for modelling Haskell's **data** construct that introduces a sum of products. We treat these type constructors as if they were given by the following **data** declarations:

$$\begin{aligned}
&\textbf{data}\ 1 &&= () \\
&\textbf{data}\ A + B &&= \mathit{inl}\ A \mid \mathit{inr}\ B \\
&\textbf{data}\ A \times B &&= (A, B).
\end{aligned}$$

Now, a polytypic function is simply given by a definition that is inductive on the structure of $T_\star$. As an example, here is the polytypic equality function. For emphasis, the type index is enclosed in angle brackets.

$$\begin{aligned}
&\mathit{equal}\langle T :: \star\rangle && ::\ T \to T \to \mathit{Bool} \\
&\mathit{equal}\langle 1\rangle\ ()\ () && = \mathit{true} \\
&\mathit{equal}\langle \mathit{Char}\rangle\ c_1\ c_2 && = \mathit{equalChar}\ c_1\ c_2 \\
&\mathit{equal}\langle T_1 + T_2\rangle\ (\mathit{inl}\ a_1)\ (\mathit{inl}\ a_2) && = \mathit{equal}\langle T_1\rangle\ a_1\ a_2 \\
&\mathit{equal}\langle T_1 + T_2\rangle\ (\mathit{inl}\ a_1)\ (\mathit{inr}\ b_2) && = \mathit{false} \\
&\mathit{equal}\langle T_1 + T_2\rangle\ (\mathit{inr}\ b_1)\ (\mathit{inl}\ a_2) && = \mathit{false} \\
&\mathit{equal}\langle T_1 + T_2\rangle\ (\mathit{inr}\ b_1)\ (\mathit{inr}\ b_2) && = \mathit{equal}\langle T_2\rangle\ b_1\ b_2 \\
&\mathit{equal}\langle T_1 \times T_2\rangle\ (a_1, b_1)\ (a_2, b_2) && = \mathit{equal}\langle T_1\rangle\ a_1\ a_2 \wedge \mathit{equal}\langle T_2\rangle\ b_1\ b_2.
\end{aligned}$$

This simple definition contains all ingredients needed to specialize *equal* for arbitrary data types. Note that the type language $T_\star$ does not contain constructions for type abstraction, application, and fixed points. Instances of polytypic functions on types with these constructions are generated automatically, see Section 4.

The type language $T_\star$ does not contain a construction for referring to constructor names either. Since we sometimes want to be able to refer to the name of a constructor, for example in a polytypic show function, we add one extra case to the type language: c **of** T, where c is a value of type *String* or another appropriate abstract data type for constructors, and T is a value of $T_\star$. For example, the Haskell data type of natural numbers

$$\textbf{data}\ \mathit{Nat} = \mathit{zero} \mid \mathit{succ}\ \mathit{Nat}$$

is represented in $T_\star$ by

$$\mathit{Nat} = \mathit{zero}\ \textbf{of}\ 1 + \mathit{succ}\ \textbf{of}\ \mathit{Nat}.$$

We adopt the convention that if the 'c **of** T' case is omitted in the definition of a polytypic function *poly*, we assume that $\mathit{poly}\langle c\ \mathbf{of}\ T\rangle = \mathit{poly}\langle T\rangle$.

The function *equal* is indexed by types 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). The language of types of kind $\star \to \star$ is characterized by the following grammar:

$$F_{\star\to\star} ::= \mathit{Id} \mid K\ 1 \mid K\ \mathit{Char} \mid F_{\star\to\star} + F_{\star\to\star} \mid F_{\star\to\star} \times F_{\star\to\star} \mid c\ \mathbf{of}\ F_{\star\to\star},$$

where Id, $K\ T$ ($T = 1$ or Char), '+', '$\times$', and **of** are given by (note that we overload the symbols '+', '$\times$', and **of**)

$$\begin{aligned}
\mathit{Id} &= \Lambda A\,.\,A\\
K\ T &= \Lambda A\,.\,T\\
F_1 + F_2 &= \Lambda A\,.\,F_1\ A + F_2\ A\\
F_1 \times F_2 &= \Lambda A\,.\,F_1\ A \times F_2\ A\\
c\ \mathbf{of}\ F &= \Lambda A\,.\,c\ \mathbf{of}\ F\ A.
\end{aligned}$$

Here, $\Lambda A\,.\,T$ denotes abstraction on the type level. For example, the type of lists parameterized by some type is defined by $\mathit{List} = K\ 1 + \mathit{Id} \times \mathit{List}$. Again, $F_{\star\to\star}$ is used to describe the language on which we define polytypic functions by induction, it is not a complete description of all 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 $F\ A$.

$$\begin{array}{lcl}
\mathit{map}\langle F :: \star \to \star\rangle & :: & \forall A\ B\,.\,(A \to B) \to (F\ A \to F\ B)\\
\mathit{map}\langle \mathit{Id}\rangle\ m\ a & = & m\ a\\
\mathit{map}\langle K\ 1\rangle\ m\ c & = & c\\
\mathit{map}\langle K\ \mathit{Char}\rangle\ m\ c & = & c\\
\mathit{map}\langle F_1 + F_2\rangle\ m\ (\mathit{inl}\ f) & = & \mathit{inl}\ (\mathit{map}\langle F_1\rangle\ m\ f)\\
\mathit{map}\langle F_1 + F_2\rangle\ m\ (\mathit{inr}\ g) & = & \mathit{inr}\ (\mathit{map}\langle F_2\rangle\ m\ g)\\
\mathit{map}\langle F_1 \times F_2\rangle\ m\ (f, g) & = & (\mathit{map}\langle F_1\rangle\ m\ f, \mathit{map}\langle F_2\rangle\ m\ g).
\end{array}$$

Using *map* we can, for instance, define generic versions of cata- and anamorphisms [30]. 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.

$$\mathbf{newtype}\ \mathit{Fix}\ F = \ \mathit{in}\{\mathit{out} :: F\ (\mathit{Fix}\ F)\}.$$

For example, the type of naturals might have been defined by $\mathit{Nat} = \mathit{Fix}\ (K\ 1 + \mathit{Id})$. Cata- and anamorphisms are then given by

$$\begin{array}{lcl}
\mathit{cata}\langle F :: \star \to \star\rangle & :: & \forall A\,.\,(F\ A \to A) \to (\mathit{Fix}\ F \to A)\\
\mathit{cata}\langle F\rangle\ \varphi & = & \varphi \cdot \mathit{map}\langle F\rangle\ (\mathit{cata}\langle F\rangle\ \varphi) \cdot \mathit{out}\\
\mathit{ana}\langle F :: \star \to \star\rangle & :: & \forall A\,.\,(A \to F\ A) \to (A \to \mathit{Fix}\ F)\\
\mathit{ana}\langle F\rangle\ \psi & = & \mathit{in} \cdot \mathit{map}\langle F\rangle\ (\mathit{ana}\langle F\rangle\ \psi) \cdot \psi.
\end{array}$$

Note that both functions are parameterized by the type functor F rather than by the fixed point $\mathit{Fix}\ F$.

2.2 Tries

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

$$\begin{aligned} 1 \rightarrow_{\text{fin}} V &\cong V \\ (T_1 + T_2) \rightarrow_{\text{fin}} V &\cong (T_1 \rightarrow_{\text{fin}} V) \times (T_2 \rightarrow_{\text{fin}} V) \\ (T_1 \times T_2) \rightarrow_{\text{fin}} V &\cong T_1 \rightarrow_{\text{fin}} (T_2 \rightarrow_{\text{fin}} V) \end{aligned}$$

Here, $T \rightarrow_{\text{fin}} V$ denotes a finite map. As $\mathit{FMap}\langle T\rangle\ V$, the generalization of *FMap_String* given in the introduction, represents the set of finite maps from T to V, the isomorphisms above can be rewritten as defining equations for $\mathit{FMap}\langle T\rangle$.

$$\begin{aligned} &\mathit{FMap}\langle T :: \star\rangle && :: \ \star \to \star \\ &\mathit{FMap}\langle 1\rangle && = \ \Lambda V \,.\, \mathit{Maybe}\ V \\ &\mathit{FMap}\langle \mathit{Char}\rangle && = \ \Lambda V \,.\, \mathit{FMapChar}\ V \\ &\mathit{FMap}\langle T_1 + T_2\rangle && = \ \Lambda V \,.\, \mathit{FMap}\langle T_1\rangle\ V \times \mathit{FMap}\langle T_2\rangle\ V \\ &\mathit{FMap}\langle T_1 \times T_2\rangle && = \ \Lambda V \,.\, \mathit{FMap}\langle T_1\rangle\ (\mathit{FMap}\langle T_2\rangle\ V) \end{aligned}$$

Note that $\mathit{FMap}\langle 1\rangle$ is *Maybe* rather than *Id* since we use the *Maybe* monad for exception handling. We assume that a suitable data structure, *FMapChar*, and an associated look-up function $\mathit{lookupChar} :: \forall V \,.\, \mathit{Char} \to \mathit{FMapChar}\ V \to \mathit{Maybe}\ V$ for characters are predefined. The generic look-up function is then given by the following definition.

$$\begin{aligned} &\mathit{lookup}\langle T :: \star\rangle && :: \ \forall V \,.\, T \to \mathit{FMap}\langle T\rangle\ V \to \mathit{Maybe}\ V \\ &\mathit{lookup}\langle 1\rangle\ ()\ t && = \ t \\ &\mathit{lookup}\langle \mathit{Char}\rangle\ c\ t && = \ \mathit{lookupChar}\ c\ t \\ &\mathit{lookup}\langle T_1 + T_2\rangle\ (\mathit{inl}\ k_1)\ (t_1, t_2) && = \ \mathit{lookup}\langle T_1\rangle\ k_1\ t_1 \\ &\mathit{lookup}\langle T_1 + T_2\rangle\ (\mathit{inr}\ k_2)\ (t_1, t_2) && = \ \mathit{lookup}\langle T_2\rangle\ k_2\ t_2 \\ &\mathit{lookup}\langle T_1 \times T_2\rangle\ (k_1, k_2)\ t && = \ (\mathit{lookup}\langle T_1\rangle\ k_1 \diamond \mathit{lookup}\langle T_2\rangle\ k_2)\ t. \end{aligned}$$

On sums the look-up function selects the appropriate map; on products it 'composes' the look-up functions for the component keys.

2.3 Labelling

A labelled data type is used to store information at the nodes of a tree. The kind of information that is stored varies from application to application: in the case of the maximum segment sum it is the subtree rooted at that node, in the case of pattern matching it is the set of patterns matching at that node. We will show how to define such labelled data types in this section. The data type *Labelled* labels a data type given by a so-called pattern functor:

$$\begin{aligned} &\mathit{Labelled}\langle F :: \star \to \star\rangle && :: \ \star \to \star \\ &\mathit{Labelled}\langle F\rangle && = \ \Lambda L \,.\, \mathit{Fix}\ (\Lambda R \,.\, \mathit{Label}\langle F\rangle\ L\ R). \end{aligned}$$

The type-indexed data type *Label* distributes the label type over the sum, and adds a label type L to each other construct. Since each construct is guarded by a constructor (c **of** F), it suffices to add labels to constructors.

$$\begin{array}{lcl} \mathit{Label}\langle F :: \star \to \star\rangle & :: & \star \to \star \to \star \\ \mathit{Label}\langle F_1 + F_2\rangle & = & \Lambda L\ R\,.\,\mathit{Label}\langle F_1\rangle\ L\ R + \mathit{Label}\langle F_2\rangle\ L\ R \\ \mathit{Label}\langle c\ \mathbf{of}\ F\rangle & = & \Lambda L\ R\,.\,F\ R \times L. \end{array}$$

The type-indexed function *suffixes* 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 $\mathit{Label}\langle F\rangle\ L\ T$.

$$\begin{array}{lcl} \mathit{add}\langle F :: \star \to \star\rangle & :: & \forall L\ T\,.\,L \to F\ T \to \mathit{Label}\langle F\rangle\ L\ T \\ \mathit{add}\langle F_1 + F_2\rangle\ l\ (\mathit{inl}\ x) & = & \mathit{inl}\ (\mathit{add}\langle F_1\rangle\ l\ x) \\ \mathit{add}\langle F_1 + F_2\rangle\ l\ (\mathit{inr}\ y) & = & \mathit{inr}\ (\mathit{add}\langle F_2\rangle\ l\ y) \\ \mathit{add}\langle c\ \mathbf{of}\ F\rangle\ l\ x & = & (x, l). \end{array}$$

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*.

$$\begin{array}{l} \mathit{suffixes}\langle F :: \star \to \star\rangle\ :: \mathit{Fix}\ F \to \mathit{Labelled}\langle F\rangle\ (\mathit{Fix}\ F) \\ \mathit{suffixes}\langle F\rangle\ l@(\mathit{in}\ t) = \mathit{in}\ (\mathit{add}\langle F\rangle\ l\ (\mathit{map}\langle F\rangle\ (\mathit{suffixes}\langle F\rangle)\ t)). \end{array}$$

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 take type-indexed data types as arguments.

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 is close to the type language $T_\star$ and reveals the structure of the Haskell data type. This implies, of course, that values of user-defined data types have to be translated to these isomorphic data types. For example, the type *Nat* of natural numbers defined by

$$\mathbf{data}\ \mathit{Nat} = \mathit{zero} \mid \mathit{succ}\ \mathit{Nat},$$

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

$$\begin{array}{ll}
\textbf{type}\ Nat' & = 1 + Nat \\
from\_Nat & :: Nat \to Nat' \\
from\_Nat\ zero & = inl\ () \\
from\_Nat\ (succ\ x) & = inr\ x \\
to\_Nat & :: Nat' \to Nat \\
to\_Nat\ (inl\ ()) & = zero \\
to\_Nat\ (inr\ x) & = succ\ x.
\end{array}$$

This mapping avoids direct recursion by adding the extra layer of Nat'. [5]

For convenience, we collect the conversion functions together in an embedding-projection pair:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{data}\ EP\ a\ b = EP\{from :: a \to b, to :: b \to a\}} \\
ep\_Nat & :: EP\ Nat\ Nat' \\
ep\_Nat & = EP\ from\_Nat\ to\_Nat.
\end{array}$$

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 Hinze [15], but now for types instead of values. For example, the product case $T_1 \times T_2$ takes two argument types for T_1 and T_2, and returns the type for the product. Recall the type-indexed data type $FMap$ defined by

$$\begin{array}{ll}
FMap\langle 1\rangle & = \Lambda V\,.\,Maybe\ V \\
FMap\langle Char\rangle & = \Lambda V\,.\,FMapChar\ V \\
FMap\langle T_1 + T_2\rangle & = \Lambda V\,.\,FMap\langle T_1\rangle\ V \times FMap\langle T_2\rangle\ V \\
FMap\langle T_1 \times T_2\rangle & = \Lambda V\,.\,FMap\langle T_1\rangle\ (FMap\langle T_2\rangle\ V).
\end{array}$$

These equations are translated to:

$$\begin{array}{lcl}
\textbf{newtype}\ FMap\_Unit\ V & = & fMap\_Unit\ (Maybe\ V) \\
\textbf{newtype}\ FMap\_Char\ V & = & fMap\_Char\ (FMapChar\ V) \\
\textbf{newtype}\ FMap\_Either\ FMA\ FMB\ V & = & fMap\_Either\ (FMA\ V, FMB\ V) \\
\textbf{newtype}\ FMap\_Product\ FMA\ FMB\ V & = & fMap\_Product\ (FMA\ (FMB\ V)).
\end{array}$$

[5] Furthermore, the mapping translates n-ary products and sums to binary products and sums. This is revealed by looking at a more complex data type, for instance

$$\textbf{data}\ Tree\ A = \ empty \mid node\ (Tree\ A)\ A\ (Tree\ A)$$

where the constructor $node$ takes three arguments. The isomorphic type generated for $Tree$ is

$$\textbf{data}\ Tree'\ a = \ 1 + Tree\ A \times (A \times Tree\ A).$$

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

$$\begin{aligned}
&\textbf{type}\ \mathit{FMap\_Nat}'\ V &&= \mathit{fMap\_Either}\ \mathit{FMap\_Unit}\ \mathit{FMap\_Nat}\ V\\
&\textbf{newtype}\ \mathit{FMap\_Nat}\ V &&= \mathit{fMap\_Nat}\{\mathit{unFMap\_Nat} :: \mathit{FMap\_Nat}'\ V\}.
\end{aligned}$$

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 embedding-projection pair:

$$\begin{aligned}
&\mathit{ep\_FMap\_Nat} :: &&\mathit{EP}\ (\mathit{FMap\_Nat}\ V)\ (\mathit{FMap\_Nat}'\ V)\\
&\mathit{ep\_FMap\_Nat} = &&\mathit{EP}\ \mathit{unFMap\_Nat}\ \mathit{fMap\_Nat}.
\end{aligned}$$

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 line of its definition. For the type indices of kind $\star$ (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:

$$\mathit{lookup}\langle T :: \star\rangle :: \ \forall V\,.\ T \to \mathit{FMap}\langle T\rangle\ V \to \mathit{Maybe}\ V,$$

the instances are obtained by replacing T by 1 or *Char*, and by replacing $\mathit{FMap}\langle T\rangle$ by *FMap_Unit* or *FMap_Char*, respectively. So, for the function *lookup* we have that the user-supplied equations

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

are translated into

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

Note that we add the constructors for the tries to the trie arguments of the function.

For the type indices of kind $\star \to \star \to \star$ (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 Hinze [15]. For example, the following lines

$$\begin{aligned}
&\mathit{lookup}\langle T_1 + T_2\rangle\ (\mathit{inl}\ k_1)\ (t_1, t_2) &&= \mathit{lookup}\langle T_1\rangle\ k_1\ t_1\\
&\mathit{lookup}\langle T_1 + T_2\rangle\ (\mathit{inr}\ k_2)\ (t_1, t_2) &&= \mathit{lookup}\langle T_2\rangle\ k_2\ t_2\\
&\mathit{lookup}\langle T_1 \times T_2\rangle\ (k_1, k_2)\ t &&= (\mathit{lookup}\langle T_1\rangle\ k_1 \diamond \mathit{lookup}\langle T_2\rangle\ k_2)\ t
\end{aligned}$$

are translated into the following functions

$$\begin{aligned}
&\mathit{lookup\_Either} :: \forall A\ \mathit{FMA}\,.\,\forall B\ \mathit{FMB}\,.\\
&\qquad (\forall V\,.\,A \to \mathit{FMA}\ V \to \mathit{Maybe}\ V)\\
&\qquad \to (\forall V\,.\,B \to \mathit{FMB}\ V \to \mathit{Maybe}\ V)\\
&\qquad \to (\forall V\,.\,A + B \to \mathit{FMap\_Either}\ \mathit{FMA}\ \mathit{FMB}\ V \to \mathit{Maybe}\ V)\\
&\mathit{lookup\_Either}\ \mathit{lua}\ \mathit{lub}\ (\mathit{inl}\ a)\ (\mathit{fMap\_Either}\ (\mathit{fma}, \mathit{fmb})) = \mathit{lua}\ a\ \mathit{fma}\\
&\mathit{lookup\_Either}\ \mathit{lua}\ \mathit{lub}\ (\mathit{inr}\ b)\ (\mathit{fMap\_Either}\ (\mathit{fma}, \mathit{fmb})) = \mathit{lub}\ b\ \mathit{fmb}
\end{aligned}$$

$$\begin{aligned}
&\mathit{lookup\_Product} :: \forall A\ \mathit{FMA}\,.\,\forall B\ \mathit{FMB}\,.\\
&\qquad (\forall V\,.\,A \to \mathit{FMA}\ V \to \mathit{Maybe}\ V)\\
&\qquad \to (\forall V\,.\,B \to \mathit{FMB}\ V \to \mathit{Maybe}\ V\quad) \to\\
&\qquad \to (\forall V\,.\,A \times B \to \mathit{FMap\_Product}\ \mathit{FMA}\ \mathit{FMB}\ V \to \mathit{Maybe}\ V)\\
&\mathit{lookup\_Product}\ \mathit{lua}\ \mathit{lub}\ (a, b)\ (\mathit{fMap\_Product}\ t) \qquad = (\mathit{lua}\ a \diamond \mathit{lub}\ b)\ t.
\end{aligned}$$

These functions are obtained from the definition of *lookup* by replacing the occurrences of the *lookup* function in the right-hand sides by their corresponding arguments.

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

$$\begin{aligned}
&\mathit{lookup\_Nat} :: \forall V\,.\,\mathit{Nat} \to \mathit{FMap\_Nat}\ V \to \mathit{Maybe}\ V\\
&\mathit{lookup\_Nat} = \mathit{conv\_Lookup\_Nat}\ (\mathit{lookup\_Either}\ \mathit{lookup\_Unit}\ \mathit{lookup\_Nat}).
\end{aligned}$$

The argument of function *conv_Lookup_Nat* (defined below) is generated directly from the type Nat'. Finally, for each instance we have to generate a conversion function like *conv_Lookup_Nat*. In general, the conversion function converts a type-indexed function that works on the translated isomorphic data type to a function that works on the original data type. As an example, the function *conv_Lookup_Nat* converts a *lookup* function on the internal data type Nat' to a *lookup* function on the type of natural numbers itself.

$$\begin{aligned}
&\mathit{conv\_Lookup\_Nat} &&:: (\forall V\,.\,\mathit{Nat}' \to \mathit{FMap\_Nat}'\ V \to \mathit{Maybe}\ V)\\
& &&\to (\forall V\,.\,\mathit{Nat} \to \mathit{FMap\_Nat}\ V \to \mathit{Maybe}\ V)\\
&\mathit{conv\_Lookup\_Nat}\ \mathit{lu} &&= \lambda t\ \mathit{fmt} \to \mathit{lu}\ (\mathit{from\_Nat}\ t)\ (\mathit{unFMap\_Nat}\ \mathit{fmt})
\end{aligned}$$

Note that the functions *to_Nat* and *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 the result of the type-indexed function consisted of values of the type index or of the type-indexed data type, these functions would be applied at the appropriate positions.

3.4 Implementing *FMap* in Haskell directly

Alternatively, we can use multi-parameter type classes and functional dependencies [24] to implement a type-indexed data type such as *FMap* in Haskell. An example is given in Figure 1. However, to use this implementation we would have to marshal and unmarshal user-defined data types and values of user-defined data types by hand. Furthermore, this approach does not work for all types of all kinds.

```
class FMap fma a | a → fma where
  lookup                          :: a → fma v → Maybe v
instance FMap Maybe () where
  lookup () fm                    = fm
data Pair f g a                   = Pair (f a) (g a)
instance (FMap fma a, FMap fmb b) ⇒ FMap (Pair fma fmb) (Either a b) where
  lookup (Left a) (Pair fma fmb)  = lookup a fma
  lookup (Right b) (Pair fma fmb) = lookup b fmb
data Comp f g a                   = Comp (f (g a))
instance (FMap fma a, FMap fmb b) ⇒ FMap (Comp fma fmb) (a, b) where
  lookup (a, b) (Comp fma)        = (lookup a ◇ lookup b) fma
```

Fig. 1. Implementing *FMap* in Haskell directly.

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 Hinze [13] to the level of data types.

Type-indexed data types and type-indexed functions take types as arguments, and return types and functions. 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 in 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 take type-indexed data types as arguments. Section 4.5 describes how the gap between the formal type language and Haskell's data types can be bridged.

4.1 The polymorphic lambda calculus

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

Kind terms are formed by:

$$\begin{array}{rlll} \mathfrak{T}, \mathfrak{U} \in \mathit{Kind} ::= & \star & & \text{kind of types} \\ \mid & (\mathfrak{T} \rightarrow \mathfrak{U}) & & \text{function kind.} \end{array}$$

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.

$$\begin{array}{rll} T, U \in \mathit{Type} ::= & C & \text{type constant} \\ \mid & A & \text{type variable} \\ \mid & (\Lambda A :: \mathfrak{U} \,.\, T) & \text{type abstraction} \\ \mid & (T\ U) & \text{type application} \end{array}$$

For typographic simplicity, we will often omit the kind annotation in $\Lambda A :: \mathfrak{U} \,.\, T$ (especially if $\mathfrak{U} = \star$) and we abbreviate nested abstractions $\Lambda A_1 \,.\, \ldots \Lambda A_m \,.\, T$ by $\Lambda A_1\ \ldots\ A_m \,.\, T$.

In order to be able to model Haskell's data types the set of type constants should include at least the types 1, *Char*, '+', '×', and 'c **of**' for all known constructors in the program. Furthermore, it should include a family of fixed point operators indexed by kind: $\mathit{Fix}_{\mathfrak{T}} :: (\mathfrak{T} \rightarrow \mathfrak{T}) \rightarrow \mathfrak{T}$. In the examples, we will often omit the kind annotation $\mathfrak{T}$ in $\mathit{Fix}_{\mathfrak{T}}$. We may additionally add the function space constructor '→' or universal quantifiers $\forall_{\mathfrak{U}} :: (\mathfrak{U} \rightarrow \star) \rightarrow \star$ to the set of type constants (see Section 4.5 for an example).

Note that both type languages we have introduced in Section 2.1, $T_\star$ and $F_{\star\rightarrow\star}$, are subsumed by this type language.

Type schemes are formed by:

$$\begin{array}{rll} R, S \in \mathit{Scheme} ::= & T & \text{type term} \\ \mid & (R \rightarrow S) & \text{functional type} \\ \mid & (\forall A :: \mathfrak{U} \,.\, S) & \text{polymorphic type.} \end{array}$$

Terms are formed by:

$$\begin{array}{rll} t, u \in \mathit{Term} ::= & c & \text{constant} \\ \mid & a & \text{variable} \\ \mid & (\lambda a :: S \,.\, t) & \text{abstraction} \\ \mid & (t\ u) & \text{application} \\ \mid & (\lambda A :: \mathfrak{U} \,.\, t) & \text{universal abstraction} \\ \mid & (t\ R) & \text{universal application.} \end{array}$$

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

$$\mathit{fix} :: \forall A \,.\, (A \to A) \to A$$

and suitable functions for each of the other type constants (such as () for '1', *inl*, *inr*, and **case** for '+', and *outl*, *outr*, and (,) for '×'). 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:

$$\begin{aligned}
\mathit{cata}\langle 1\rangle &= \mathit{cata}_1\\
\mathit{cata}\langle \mathit{Char}\rangle &= \mathit{cata}_{\mathit{Char}}\\
\mathit{cata}\langle T_1 + T_2\rangle &= \mathit{cata}_+\ (\mathit{cata}\langle T_1\rangle)\ (\mathit{cata}\langle T_2\rangle)\\
\mathit{cata}\langle T_1 \times T_2\rangle &= \mathit{cata}_\times\ (\mathit{cata}\langle T_1\rangle)\ (\mathit{cata}\langle T_2\rangle)\\
\mathit{cata}\langle c\ \mathbf{of}\ T_1\rangle &= \mathit{cata}_{c\ \mathbf{of}}\ (\mathit{cata}\langle T_1\rangle).
\end{aligned}$$

These equations implicitly define the family of functions cata_1, $\mathit{cata}_{\mathit{Char}}$, 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

$$\begin{aligned}
\mathit{FMap}_1 &= \Lambda V \,.\, \mathit{Maybe}\ V\\
\mathit{FMap}_{\mathit{Char}} &= \Lambda V \,.\, \mathit{FMapChar}\ V\\
\mathit{FMap}_+ &= \Lambda \mathit{FMap}_A\ \mathit{FMap}_B\ V \,.\, \mathit{FMap}_A\ V \times \mathit{FMap}_B\ V\\
\mathit{FMap}_\times &= \Lambda \mathit{FMap}_A\ \mathit{FMap}_B\ V \,.\, \mathit{FMap}_A\ (\mathit{FMap}_B\ V)\\
\mathit{FMap}_{c\ \mathbf{of}} &= \Lambda \mathit{FMap}_A\ V \,.\, \mathit{FMap}_A\ V.
\end{aligned}$$

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 [29], and are characterized by:

$$\begin{aligned}
\mathit{para}\langle 1\rangle &= \mathit{para}_1\\
\mathit{para}\langle \mathit{Char}\rangle &= \mathit{para}_{\mathit{Char}}\\
\mathit{para}\langle T_1 + T_2\rangle &= \mathit{para}_+\ T_1\ T_2\ (\mathit{para}\langle T_1\rangle)\ (\mathit{para}\langle T_2\rangle)\\
\mathit{para}\langle T_1 \times T_2\rangle &= \mathit{para}_\times\ T_1\ T_2\ (\mathit{para}\langle T_1\rangle)\ (\mathit{para}\langle T_2\rangle)\\
\mathit{para}\langle c\ \mathbf{of}\ T_1\rangle &= \mathit{para}_{c\ \mathbf{of}}\ T_1\ (\mathit{para}\langle T_1\rangle).
\end{aligned}$$

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 below describes how to specialize type-indexed data types with type indices that appear in the set C 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 of this lifting process see Hinze [13].

4.3 Specializing type-indexed data types

Rather amazingly, the process of specialization 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, $FMap_{T::\star} :: \star \to \star$. However, when type application is interpreted as application, we have that $FMap_{List\ A} = FMap_{List}\ FMap_A$. 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 $FMap_{List} :: (\star \to \star) \to (\star \to \star)$.

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

$$Data_{T::\mathfrak{T}} :: \mathfrak{Data}_{\mathfrak{T}},$$

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

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

Here, '$\Box$' 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 $FMap_T$, 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 [31] for the type-indexed data type.

An environment model is an applicative structure ($\mathbf{\mathsf{M}}, \mathbf{\mathsf{app}}, \mathbf{\mathsf{const}}$), where $\mathbf{\mathsf{M}}$ is the domain of the structure, $\mathbf{\mathsf{app}}$ a mapping that interprets functions, and $\mathbf{\mathsf{const}}$ maps constants to the domain of the structure. Furthermore, 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 Mitchell [31]. For an arbitrary type-indexed data type $Data_{T::\mathfrak{T}} :: \mathfrak{Data}_{\mathfrak{T}}$ we use the following applicative structure:

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

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 $\mathcal{E}$ between type terms (e.g. $F\ (Fix_{\mathfrak{T}}\ F) = Fix_{\mathfrak{T}}\ F$ 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 $[\,T\,]$ and $[\,U\,]$) 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 defined thus 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:

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

4.4 Specializing type-indexed values

A type-indexed value possesses a kind-indexed type [15],

$$poly_{T::\mathfrak{T}} :: Poly_{\mathfrak{T}}\ Data^1_T\ \ldots\ Data^n_T$$

in which $Poly_{\mathfrak{T}}$ has the following general form

$$\begin{aligned}
Poly_{\mathfrak{T}::\square} &:: \mathfrak{Data}^1_{\mathfrak{T}} \to \cdots \to \mathfrak{Data}^n_{\mathfrak{T}} \to \star\\
Poly_{\star} &= \Lambda X_1 :: \mathfrak{Data}^1_{\star} \,.\, \ldots \,.\, \Lambda X_n :: \mathfrak{Data}^n_{\star} \,.\, \boxed{}\\
Poly_{\mathfrak{A}\to\mathfrak{B}} &= \Lambda X_1 :: \mathfrak{Data}^1_{\mathfrak{A}\to\mathfrak{B}} \,.\, \ldots \,.\, \Lambda X_n :: \mathfrak{Data}^n_{\mathfrak{A}\to\mathfrak{B}} \,.\\
&\qquad \forall A_1 :: \mathfrak{Data}^1_{\mathfrak{A}} \,.\, \ldots \,.\, \forall A_n :: \mathfrak{Data}^n_{\mathfrak{A}} \,.\\
&\qquad\quad Poly_{\mathfrak{A}}\ A_1\ \ldots\ A_n \to Poly_{\mathfrak{B}}\ (X_1\ A_1)\ \ldots\ (X_n\ A_n).
\end{aligned}$$

Again, note that only an equation for $Poly_{\star}$ 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 Hinze [15] $Poly_{\mathfrak{T}}$ takes

n arguments of type $\mathfrak{T}$. Here $Poly_{\mathfrak{T}}$ takes n possibly different type arguments obtained from the type-indexed data type arguments. For example, for the look-up function we have:

$$\begin{aligned} &Lookup_{\mathfrak{T}::\Box} &&:: \mathfrak{Id}_{\mathfrak{T}} \to \mathfrak{FMap}_{\mathfrak{T}} \to \star \\ &Lookup_{\star} &&= \Lambda K \,.\, \Lambda FMK \,.\, \forall V \,.\, K \to FMK\ V \to Maybe\ V, \end{aligned}$$

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

$$\begin{aligned} &lookup_{T::\mathfrak{T}} &&:: Lookup_{\mathfrak{T}}\ Id_T\ FMap_T \\ &lookup_{1} &&= \lambda V\ k\ fmk \,.\, fmk \\ &lookup_{Char} &&= lookupChar \\ &lookup_{+} &&= \lambda A\ FMA\ lookup_A \,.\, \lambda B\ FMB\ lookup_B \,. \\ & &&\quad \lambda V\ k\ (fmkl, fmkr) \,.\, \mathbf{case}\ k\ \mathbf{of}\ \{ inl\ a \to lookup_A\ V\ a\ fmkl; \\ & &&\qquad\qquad\qquad\qquad\qquad\qquad inr\ b \to lookup_B\ V\ b\ fmkr \} \\ &lookup_{\times} &&= \lambda A\ FMA\ lookup_A \,.\, \lambda B\ FMB\ lookup_B \,. \\ & &&\quad \lambda V\ (kl, kr)\ fmk \,.\, (lookup_A\ V\ kl \diamond lookup_B\ V\ kr)\ fmk \\ &lookup_{c\ \mathbf{of}} &&= \lambda A\ FMA\ lookup_A \,.\, \lambda V\ k\ fmk \,.\, lookup_A\ V\ k\ fmk. \end{aligned}$$

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 a dependent product: the type of the last component (the equivalence class of the terms of type $Poly_{\mathfrak{T}}\ D_1\ \ldots\ D_n$) depends on the first n components (the equivalence classes of the type schemes $D_1\ \ldots\ D_n$ 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.

$$\begin{aligned} &\mathsf{M}^{\mathfrak{T}} &&= ([D_1] \in Scheme^{\mathfrak{Data}^1_{\mathfrak{T}}} / \mathcal{E}, \ldots, [D_n] \in Scheme^{\mathfrak{Data}^n_{\mathfrak{T}}} / \mathcal{E}; \\ & &&\quad Term^{Poly_{\mathfrak{T}}\ D_1\ \ldots\ D_n} / \mathcal{E}) \\ &\mathsf{app}_{\mathfrak{T},\mathfrak{U}}\ ([R_1], \ldots, [R_n]; [t])\ ([S_1], \ldots, [S_n]; [u]) \\ & &&= ([R_1\ S_1], \ldots, [R_n\ S_n]; [t\ S_1\ \ldots\ S_n\ u]) \\ &\mathsf{const}(C) = ([Data^1_C], \ldots, [Data^n_C]; [poly_C]). \end{aligned}$$

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

$$\mathsf{const}(Fix_{\mathfrak{T}}) = ([Fix_{\mathfrak{Data}^1_{\mathfrak{T}}}], \ldots, [Fix_{\mathfrak{Data}^n_{\mathfrak{T}}}]; [poly_{Fix_{\mathfrak{T}}}]),$$

where $poly_{Fix_{\mathfrak{T}}}$ is given by

$$\begin{aligned} poly_{Fix_{\mathfrak{T}}} = {}& \lambda F_1\ \ldots\ F_n \,.\, \lambda poly_F :: Poly_{\mathfrak{T}\to\mathfrak{T}}\ F_1\ \ldots\ F_n \,. \\ & fix\ poly_F\ (Fix_{\mathfrak{Data}^1_{\mathfrak{T}}}\ F_1)\ \ldots\ (Fix_{\mathfrak{Data}^n_{\mathfrak{T}}}\ F_n). \end{aligned}$$

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 specialisation 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 between these isomorphic types.

These conversion functions are easily generated, both for type-indexed values and data types, and can be stored in embedding-projection pairs. 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 [13], which makes use of the *bimap* function displayed in Figure 2.

$$
\begin{aligned}
&\mathit{Bimap}_{\mathfrak{T}::\Box} && :: \mathfrak{ID}_{\mathfrak{T}} \to \mathfrak{ID}_{\mathfrak{T}} \to \star \\
&\mathit{Bimap}_{\star} && = \Lambda T_1 \,.\, \Lambda T_2 \,.\, \mathit{EP}\ T_1\ T_2 \\
&\mathit{bimap}_{T::\mathfrak{T}} && :: \mathit{Bimap}_{\mathfrak{T}}\ \mathit{Id}_T\ \mathit{Id}_T \\
&\mathit{bimap}_{1} && = \mathit{EP}\ \mathit{id}\ \mathit{id} \\
&\mathit{bimap}_{\mathit{Char}} && = \mathit{EP}\ \mathit{id}\ \mathit{id} \\
&\mathit{bimap}_{+} && = \lambda A_1\ A_2\ \mathit{bimap}_A \,.\, \lambda B_1\ B_2\ \mathit{bimap}_B \,. \\
& && \quad \mathit{EP}\ (\lambda ab \to \mathbf{case}\ ab\ \mathbf{of}\ \{\mathit{inl}\ a \to (\mathit{inl} \,.\, \mathit{from}\ \mathit{bimap}_A)\ a; \\
& && \qquad\qquad\qquad\qquad\qquad\quad \mathit{inr}\ b \to (\mathit{inr} \,.\, \mathit{from}\ \mathit{bimap}_B)\ b\}) \\
& && \qquad\quad (\lambda ab \to \mathbf{case}\ ab\ \mathbf{of}\ \{\mathit{inl}\ a \to (\mathit{inl} \,.\, \mathit{to}\ \mathit{bimap}_A)\ a; \\
& && \qquad\qquad\qquad\qquad\qquad\quad \mathit{inr}\ b \to (\mathit{inr} \,.\, \mathit{to}\ \mathit{bimap}_B)\ b\}) \\
&\mathit{bimap}_{\times} && = \lambda A_1\ A_2\ \mathit{bimap}_A \,.\, \lambda B_1\ B_2\ \mathit{bimap}_B \,. \\
& && \quad \mathit{EP}\ (\lambda(a,b) \to (\mathit{from}\ \mathit{bimap}_A\ a, \mathit{from}\ \mathit{bimap}_B\ b)) \\
& && \qquad\quad (\lambda(a,b) \to (\mathit{to}\ \mathit{bimap}_A\ a, \mathit{to}\ \mathit{bimap}_B\ b)) \\
&\mathit{bimap}_{\to} && = \lambda A_1\ A_2\ \mathit{bimap}_A \,.\, \lambda B_1\ B_2\ \mathit{bimap}_B \,. \\
& && \quad \mathit{EP}\ (\lambda ab \to \mathit{from}\ \mathit{bimap}_B \,.\, ab \,.\, \mathit{to}\ \mathit{bimap}_A) \\
& && \qquad\quad (\lambda ab \to \mathit{to}\ \mathit{bimap}_B \,.\, ab \,.\, \mathit{from}\ \mathit{bimap}_A) \\
&\mathit{bimap}_{\forall_\star} && = \lambda F_1\ F_2\ \mathit{bimap}_F \,. \\
& && \quad \mathit{EP}\ (\lambda f\ V \,.\, \mathit{from}\ (\mathit{bimap}_F\ V\ V\ (\mathit{EP}\ \mathit{id}\ \mathit{id}))\ (f\ V)) \\
& && \qquad\quad (\lambda f\ V \,.\, \mathit{to}\ (\mathit{bimap}_F\ V\ V\ (\mathit{EP}\ \mathit{id}\ \mathit{id}))\ (f\ V)) \\
&\mathit{bimap}_{c\ \mathbf{of}} && = \lambda A_1\ A_2\ \mathit{bimap}_A \,.\, \mathit{bimap}_A
\end{aligned}
$$

Fig. 2. Lifting isomorphisms with a generic function.

Consider the generic function

$$\mathit{poly}_{T::\mathfrak{T}} :: \mathit{Poly}_{\mathfrak{T}}\ \mathit{Data}^1_T\ \ldots\ \mathit{Data}^n_T.$$

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

$$\mathit{conv\_poly\_T} = \mathit{to}\ (\mathit{bimap}_{\mathit{Poly}_\star}\ \mathit{ep}_{\mathit{Data}^1_T}\ \ldots\ \mathit{ep}_{\mathit{Data}^n_T}).$$

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

$$conv\_lookup\_Nat = to\ (bimap_{Lookup_\star}\ ep\_Nat\ ep\_FMap\_Nat),$$

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 $Lookup_\star$ is a polymorphic type. In this specific case, however, polymorphic type indices can be easily handled, see Figure 2. The further details are exactly the same as for type-indexed values [16, 13], and are omitted here.

5 An advanced example: the Zipper

This section shows how to define a 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 and 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*:

$$\textbf{data}\ Tree\ A = empty \mid node\ (Tree\ A)\ A\ (Tree\ A).$$

If the left subtree of a *node* constructor is selected, moving right means moving to the right tree, not to the *A*-label. This implies that recursive positions in trees play an important rôle 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 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.

$$\begin{aligned}
&Loc\langle F :: \star \to \star\rangle && :: \star\\
&Loc\langle F\rangle && = (Fix\ F, Context\langle F\rangle\ (Fix\ F))\\
&Context\langle F :: \star \to \star\rangle && :: \star \to \star\\
&Context\langle F\rangle && = \Lambda R\,.\,Fix\ (\Lambda C\,.\,1 + Ctx\langle F\rangle\ C\ R).
\end{aligned}$$

The type *Loc* is defined in terms *Context*, which constructs the context parameterized by the original tree type. The *Context* of a value is either empty (represented by 1 in the pattern functor for *Context*), or it is a path from the root down into the tree. Such a path is constructed by means of the second component of the pattern functor for *Context*: 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.

$$\begin{array}{ll}
\mathit{Ctx}\langle F :: \star \to \star\rangle & :: \star \to \star \to \star \\
\mathit{Ctx}\langle \mathit{Id}\rangle & = \Lambda C\ R\,.\,C \\
\mathit{Ctx}\langle K\ 1\rangle & = \Lambda C\ R\,.\,0 \\
\mathit{Ctx}\langle K\ \mathit{Char}\rangle & = \Lambda C\ R\,.\,0 \\
\mathit{Ctx}\langle F_1 + F_2\rangle & = \Lambda C\ R\,.\,\mathit{Ctx}\langle F_1\rangle\ C\ R + \mathit{Ctx}\langle F_2\rangle\ C\ R \\
\mathit{Ctx}\langle F_1 \times F_2\rangle & = \Lambda C\ R\,.\,(\mathit{Ctx}\langle F_1\rangle\ C\ R \times F_2\ R) + (F_1\ R \times \mathit{Ctx}\langle F_2\rangle\ C\ R)
\end{array}$$

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' 0. Note that although 0 does not appear in the grammars for types introduced in Section 2.1, it may appear as the result of a type-indexed data type. 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 $K\ 1 + \mathit{Id}$ or, equivalently, $\Lambda N\,.\,1 + N$, and for trees of type *Bush* whose pattern functor is $K\ \mathit{Char} + \mathit{Id} \times \mathit{Id}$ or, equivalently, $\Lambda T\,.\,\mathit{Char} + (T \times T)$ we obtain

$$\begin{array}{ll}
\mathit{Context}\langle K\ 1 + \mathit{Id}\rangle & = \Lambda R\,.\,\mathit{Fix}\ (\Lambda C\,.\,1 + (0 + C)) \\
\mathit{Context}\langle K\ \mathit{Char} + \mathit{Id} \times \mathit{Id}\rangle & = \Lambda R\,.\,\mathit{Fix}\ (\Lambda C\,.\,1 + (0 + (C \times R + R \times C))),
\end{array}$$

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 *Ctx_Bush* introduced in Section 1.

We recently found that McBride [27] 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. Interestingly, he observes that the *Context* of a data type is its derivative (as in calculus).

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

$$\forall l\,.\,down\langle F\rangle\ l \neq l \implies (up\langle F\rangle \cdot down\langle F\rangle)\ l = l,$$

where 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 $down\langle F\rangle \cdot up\langle F\rangle = id$ also does not hold in general. However, there does exist a natural number n such that

$$\forall l\,.\,up\langle F\rangle\ l \neq l \implies (right\langle F\rangle^n \cdot down\langle F\rangle \cdot up\langle F\rangle)\ l = l.$$

The function *down* is defined as follows.

$$\begin{array}{ll} down\langle F :: \star \to \star\rangle & ::\ Loc\langle F\rangle \to Loc\langle F\rangle \\ down\langle F\rangle\ (t, c) & = \mathbf{case}\ first\langle F\rangle\ (out\ t)\ c\ \mathbf{of} \\ & \qquad just\ (t', c') \to (t', in\ (inr\ c')) \\ & \qquad nothing \to (t, c). \end{array}$$

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 $Ctx\langle F\rangle\ C\ T$) 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 *inr*, and applying the fixed point constructor *in* to it.

$$\begin{array}{ll} first\langle F :: \star \to \star\rangle & ::\ \forall C\ T\,.\,F\ T \to C \to Maybe\ (T, Ctx\langle F\rangle\ C\ T) \\ first\langle Id\rangle\ t\ c & = return\ (t, c) \\ first\langle K\ 1\rangle\ t\ c & = fail \\ first\langle K\ Char\rangle\ t\ c & = fail \\ first\langle F_1 + F_2\rangle\ (inl\ x)\ c & = \mathbf{do}\ \{(t, cx) \leftarrow first\langle F_1\rangle\ x\ c; return\ (t, inl\ cx)\} \\ first\langle F_1 + F_2\rangle\ (inr\ y)\ c & = \mathbf{do}\ \{(t, cy) \leftarrow first\langle F_2\rangle\ y\ c; return\ (t, inr\ cy)\} \\ first\langle F_1 \times F_2\rangle\ (x, y)\ c & = \mathbf{do}\ \{(t, cx) \leftarrow first\langle F_1\rangle\ x\ c; return\ (t, inl\ (cx, y))\} \\ & \quad +\!\!+ \mathbf{do}\ \{(t, cy) \leftarrow first\langle F_2\rangle\ y\ c; return\ (t, inr\ (x, cy))\} \end{array}$$

Here, $(+\!\!+)$ is the standard monadic plus, called *mplus* in Haskell, given by

$$\begin{array}{ll} (+\!\!+) & ::\ Maybe\ A \to Maybe\ A \to Maybe\ A \\ nothing +\!\!+ m & = m \\ just\ a +\!\!+ m & = just\ a. \end{array}$$

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* to the reader.

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

$$\begin{aligned}
&\mathit{up}\langle F :: \star \to \star\rangle &&:: \mathit{Loc}\langle F\rangle \to \mathit{Loc}\langle F\rangle\\
&\mathit{up}\langle F\rangle\ (t, c) &&= \mathbf{case}\ \mathit{out}\ c\ \mathbf{of}\\
& && \qquad \mathit{inl}\ () \to (t, c)\\
& && \qquad \mathit{inr}\ c' \to \mathbf{do}\ \{\mathit{ft} \leftarrow \mathit{insert}\langle F\rangle\ c'\ t;\\
& && \qquad\qquad\qquad\quad c'' \leftarrow \mathit{extract}\langle F\rangle\ c';\\
& && \qquad\qquad\qquad\quad \mathit{return}\ (\mathit{in}\ \mathit{ft}, c'')\}.
\end{aligned}$$

Remember that *inl* () 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 $\mathit{Ctx}\langle F\rangle\ C\ T$ has at most one C component (by an easy inductive argument), which marks the context of the parent of the current node. The polytypic function *extract* extracts this context.

$$\begin{aligned}
&\mathit{extract}\langle F :: \star \to \star\rangle &&:: \forall C\ T\,.\,\mathit{Ctx}\langle F\rangle\ C\ T \to \mathit{Maybe}\ C\\
&\mathit{extract}\langle \mathit{Id}\rangle\ c &&= \mathit{return}\ c\\
&\mathit{extract}\langle K\ 1\rangle\ c &&= \mathit{fail}\\
&\mathit{extract}\langle K\ \mathit{Char}\rangle\ c &&= \mathit{fail}\\
&\mathit{extract}\langle F_1 + F_2\rangle\ (\mathit{inl}\ \mathit{cx}) &&= \mathit{extract}\langle F_1\rangle\ \mathit{cx}\\
&\mathit{extract}\langle F_1 + F_2\rangle\ (\mathit{inr}\ \mathit{cy}) &&= \mathit{extract}\langle F_2\rangle\ \mathit{cy}\\
&\mathit{extract}\langle F_1 \times F_2\rangle\ (\mathit{inl}\ (\mathit{cx}, y)) &&= \mathit{extract}\langle F_1\rangle\ \mathit{cx}\\
&\mathit{extract}\langle F_1 \times F_2\rangle\ (\mathit{inr}\ (x, \mathit{cy})) &&= \mathit{extract}\langle F_2\rangle\ \mathit{cy}
\end{aligned}$$

Here, *return* is obtained from the *Maybe* monad and *fail* is shorthand for *nothing*. 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.

$$\begin{aligned}
&\mathit{insert}\langle F :: \star \to \star\rangle &&:: \forall C\ T\,.\,\mathit{Ctx}\langle F\rangle\ C\ T \to T \to \mathit{Maybe}\ (F\ T)\\
&\mathit{insert}\langle \mathit{Id}\rangle\ c\ t &&= \mathit{return}\ t\\
&\mathit{insert}\langle K\ 1\rangle\ c\ t &&= \mathit{fail}\\
&\mathit{insert}\langle K\ \mathit{Char}\rangle\ c\ t &&= \mathit{fail}\\
&\mathit{insert}\langle F_1 + F_2\rangle\ (\mathit{inl}\ \mathit{cx})\ t &&= \mathbf{do}\ \{x \leftarrow \mathit{insert}\langle F_1\rangle\ \mathit{cx}\ t; \mathit{return}\ (\mathit{inl}\ x)\}\\
&\mathit{insert}\langle F_1 + F_2\rangle\ (\mathit{inr}\ \mathit{cy})\ t &&= \mathbf{do}\ \{y \leftarrow \mathit{insert}\langle F_2\rangle\ \mathit{cy}\ t; \mathit{return}\ (\mathit{inr}\ y)\}\\
&\mathit{insert}\langle F_1 \times F_2\rangle\ (\mathit{inl}\ (\mathit{cx}, y))\ t &&= \mathbf{do}\ \{x \leftarrow \mathit{insert}\langle F_1\rangle\ \mathit{cx}\ t; \mathit{return}\ (x, y)\}\\
&\mathit{insert}\langle F_1 \times F_2\rangle\ (\mathit{inr}\ (x, \mathit{cy}))\ t &&= \mathbf{do}\ \{y \leftarrow \mathit{insert}\langle F_2\rangle\ \mathit{cy}\ t; \mathit{return}\ (x, y)\}.
\end{aligned}$$

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

Since $up\langle F\rangle \cdot down\langle F\rangle = id$ 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

```
**do** { $(t, c') \leftarrow first\langle F\rangle\ ft\ c$;
       $c'' \leftarrow extract\langle F\rangle\ c'$;
       $ft' \leftarrow insert\langle F\rangle\ c'\ t$;
       $return\ (c == c'' \wedge ft == ft')$ }
```

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

Function right. The function *right* moves the focus to the next sibling to the right 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:

$$\forall l\,.\,right\langle F\rangle\ l \neq l \implies (left\langle F\rangle \cdot right\langle F\rangle)\ l = l,$$

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.

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

```
right⟨F :: ⋆ → ⋆⟩ :: Loc⟨F⟩ → Loc⟨F⟩
right⟨F⟩ (t, c)   =  case out c of
                       inl () → (t, c)
                       inr c' → case next⟨F⟩ t c' of
                                  just (t', c'') → (t', in (inr c''))
                                  nothing → (t, c).
```

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 $left\langle F\rangle \cdot right\langle F\rangle = id$ (on locations in which it is possible to go to the right), there exists a function *previous*, such that

```
do { (t', c') ← next⟨F⟩ t c;
     (t'', c'') ← previous⟨F⟩ t' c';
     return (c == c'' ∧ 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*.

```
next⟨F :: ⋆ → ⋆⟩ :: ∀C T . T → Ctx⟨F⟩ C T → Maybe (T, Ctx⟨F⟩ C T)
next⟨Id⟩ t c            = fail
next⟨K 1⟩ t c           = fail
next⟨K Char⟩ t c        = fail
next⟨F1 + F2⟩ t (inl cx) = do {(t′, cx′) ← next⟨F1⟩ t cx; return (t′, inl cx′)}
next⟨F1 + F2⟩ t (inr cy) = do {(t′, cy′) ← next⟨F2⟩ t cy; return (t′, inr cy′)}
next⟨F1 × F2⟩ t (inl (cx, y))
  = do {(t′, cx′) ← next⟨F1⟩ t cx; return (t′, inl (cx′, y))}
    ++ do {c ← extract⟨F1⟩ cx;
           x ← insert⟨F1⟩ cx t;
           (t′, cy) ← first⟨F2⟩ y c;
           return (t′, inr (x, cy))}
next⟨F1 × F2⟩ t (inr (x, cy))
  = do {(t′, cy′) ← next⟨F2⟩ t cy; return (t′, inr (x, cy′))}.
```

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, the zipper, and labelling a data type. 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 [15], and is used in the implementation of Generic Haskell, a generic programming extension of the functional language Haskell. The first release of Generic Haskell was published on 1st November 2001, see `http://www.generic-haskell.org/`. A technical overview of the compiler can be found in De Wit's thesis [7]. The next release will contain an experimental implementation of type-indexed data types. Check the webpage or contact the authors for a preliminary version.

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 $\star$. For instance, *Label* is indexed by types of kind $\star \to \star$ and yields types of kind $\star \to \star \to \star$.

There are several things that remain to be done. We want to test our framework on the type-indexed data types appearing in the literature [3, 10, 34], and we want to create a library of recurring examples. Furthermore, we have to investigate how we can deal with sets of mutually recursive type-indexed data types (this extension requires tuples on the kind level).

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. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.
2. Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.
3. Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *Proceedings ICFP 2000: International Conference on Functional Programming*, pages 94–105. ACM Press, 2000.
4. Koen Claessen and Peter Ljunglöf. Typed logical variables in Haskell. In *Proceedings Haskell Workshop 2000*, 2000.
5. Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings ICFP 1999: International Conference on Functional Programming*, pages 233–248. ACM Press, 1999.
6. Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings ICFP 1998: International Conference on Functional Programming*, pages 301–312. ACM Press, 1998.
7. Jan de Wit. A technical overview of Generic Haskell. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.
8. C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *22nd Symposium on Principles of Programming Languages, POPL '95*, pages 118–129, 1995.
9. M.M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.
10. Jeremy Gibbons. Polytypic downwards accumulations. In *Proceedings of Mathematics of Program Construction*, volume 1422 of *LNCS*. Springer-Verlag, June 1998.
11. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages, POPL '95*, pages 130–141, 1995.

12. Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.
13. Ralf Hinze. *Generic Programs and Proofs.* 2000. Habilitationsschrift, Bonn University.
14. Ralf Hinze. A new approach to generic functional programming. In *Conference Record of POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 2000.
15. Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.
16. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, August 2001. The preliminary proceedings appeared as a University of Nottingham technical report.
17. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
18. P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
19. Patrik Jansson. The WWW home page for polytypic programming. Available from `http://www.cs.chalmers.se/~patrikj/poly/`, 2001.
20. Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.
21. Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In J. Jeuring, editor, *Workshop on Generic Programming 2000, Ponte de Lima, Portugal, July 2000*, pages 33–45, 2000. Utrecht Technical Report UU-CS-2000-19.
22. C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.
23. J. Jeuring. Polytypic pattern matching. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248. ACM Press, 1995.
24. Mark P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany*, volume 1782 of *LNCS*, pages 230–244. Springer-Verlag, March 2000.
25. Ralf Lämmel and Wolfgang Lohmann. Format Evolution. In J. Kouloumdjian, H.C. Mayr, and A. Erkollar, editors, *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.
26. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
27. Connor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.
28. Nancy J. McCracken. *An investigation of a programming language with a polymorphic type structure.* PhD thesis, Syracuse University, June 1979.
29. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.
30. E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *FPCA'91: Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.

31. John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.
32. Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.
33. Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Proceedings ICFP 2000: International Conference on Functional Programming*, pages 82–93. ACM Press, 2000.
34. Måns Vestin. Genetic algorithms in Haskell with polytypic programming. Examensarbeten 1997:36, Göteborg University, Gothenburg, Sweden, 1997. Available from the Polytypic programming WWW page [19].
35. Stephanie Weirich. Encoding intensional type analysis. In *European Symposium on Programming*, volume 2028 of *LNCS*, pages 92–106. Springer-Verlag, 2001.
36. Zhe Yang. Encoding types in ML-like languages. In *Proceedings ICFP 1998: International Conference on Functional Programming*, pages 289–300. ACM Press, 1998.