# Type-indexed data types

Ralf Hinze[1,2], Johan Jeuring[2], 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}

## 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 specialization is illustrated with example translations to Haskell. This specialization is also used in the extension of Generic Haskell with a construct for defining type-indexed data types.

## 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 [29], 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 [10], pattern matching [21], unification [18, 4], and rewriting [19]. 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. Since type-indexed types can not yet be implemented in the languages supporting generic programming, the type-indexed data types that appear in the literature are either implemented in an ad-hoc fashion [18], or not implemented at all [10].

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. This specialization is also used in the extension of Generic Haskell with a construct for defining type-indexed data types.

Note that there also exist many polytypic functions that do not use type-indexed data types: all basic polytypic functions from PolyLib [16] do not need type-indexed data types.

**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]

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

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

$$\begin{array}{l}\textbf{data}\ \mathit{FMapString}\ V \quad = \\ \quad \mathit{trieString}\ (\mathit{Maybe}\ V)\ (\mathit{FMapTChar}\ (\mathit{FMapString}\ V)).\end{array}$$

Our goal is to abstract from *String*, and to define a data type indexed by an arbitrary type $T$: $\mathit{FMap}\langle T\rangle$. The first component of the constructor *trieString* contains the value associated with *nil*. Its type is $\mathit{Maybe}\ V$ instead of $V$, since *nil* may not be in the domain of the finite map. The sec-

ond component of *trieString* is derived from the constructor $cons :: Char \to String \to String$. We assume that a suitable data structure, $FMap\,TChar$ (the $T$ that appears in the name avoids a name clash with a generated name), and an associated look-up function $lookup\,TChar :: \forall\, V\,.\, Char \to FMap\,TChar\ V \to Maybe\ V$ for characters are predefined.

[1] The examples are given in Haskell [29]. 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).

Such a trie for strings would typically be used for a concordance or another index on texts. Given these prerequisites we can define a look-up function for strings as follows.

$$\begin{array}{lcl} lookupString :: String \to FMapString\ V \to Maybe\ V & & \\ lookupString\ nil\ (trieString\ tn\ tc) & = & tn \\ lookupString\ (cons\ c\ s)\ (trieString\ tn\ tc) & = & \\ \quad (lookup\,TChar\ c \diamond lookupString\ s)\ tc. & & \end{array}$$

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

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

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 [10], and it is repeated here because it serves as a nice and simple example

of a type-indexed data type.

**Example 2: Zipper.** The zipper [14] 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 *LocBush*, are defined by

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Bush} & = & \mathit{leaf}\ \mathit{Char} \mid \mathit{fork}\ \mathit{Bush}\ \mathit{Bush} \\
\textbf{type}\ \mathit{LocBush} & = & (\mathit{Bush}, \mathit{ContextBush}) \\
\textbf{data}\ \mathit{ContextBush} & = & \mathit{top} \\
 & \mid & \mathit{forkL}\ \mathit{ContextBush}\ \mathit{Bush} \\
 & \mid & \mathit{forkR}\ \mathit{Bush}\ \mathit{ContextBush}.
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathit{downBush} & :: & \mathit{LocBush} \to \mathit{LocBush} \\
\mathit{downBush}\ (\mathit{leaf}\ a, c) & = & (\mathit{leaf}\ a, c) \\
\mathit{downBush}\ (\mathit{fork}\ tl\ tr, c) & = & (tl, \mathit{forkL}\ c\ tr) \\
\mathit{rightBush} & :: & \mathit{LocBush} \to \mathit{LocBush} \\
\mathit{rightBush}\ (tl, \mathit{forkL}\ c\ tr) & = & (tr, \mathit{forkR}\ tl\ c) \\
\mathit{rightBush}\ l & = & l.
\end{array}
$$

Note that we go down to the left subtree of a node. Huet [14] 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 show how to define a zipper for an arbitrary data type. This is a rather involved example of a type-indexed data type, together with some type-indexed functions that take an argument of a type-indexed data type.

**Example 3: Pattern matching.** The polytypic functions for the maximum segment sum problem [2] and pattern matching [21] use labelled data types. These labelled data types, introduced in [2], are used to store at each node the subtree rooted at that subtree, or a set of patterns (trees with variables) matching at a subtree, etc. For example, the data type of labelled bushes is defined by

$$\begin{array}{lcl} \textbf{data}\ \mathit{LabBush}\ L & = & \mathit{labelLeaf}\ \mathit{Char}\ L \\ & \mid & \mathit{labelFork}\ (\mathit{LabBush}\ L)\ (\mathit{LabBush}\ L)\ L. \end{array}$$

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



**Other examples.** Besides these three examples, a number of other examples of type-indexed data types have appeared in the literature [3, 9, 31]. Another field where we expect that type-indexed data types will be useful is generic DTD transformations [22]. We think 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 [23, 2, 26, 8, 15] were introduced back in the nineties. There exist other approaches to type-indexed functions, see Dubois et al [7], Jay et al [20] and Yang [33], but none of them mentions user-defined type-indexed data types.

Type-indexed data types appear in the work on inten-

sional type analysis [9, 6, 5, 30, 32]. 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 that typed intermediate languages are expressive, but rather complex languages not intended for programmers but for compiler writers; typed intermediate languages are not built on top of an existing programming language, so there is no integration problem; and, most importantly, 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.

**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 [12, 13]. Section 3 gives some example specializations on concrete data types of type-indexed data types and functions defined on type-indexed data types in Haskell. This specialization is also used in the extension of Generic Haskell with a construct for defining type-indexed data types. 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 gives the details of the zipper example. 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 [25]. For a more thorough treatment the reader is referred to Hinze's work [13, 12].

### 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:

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

Note that the unit type, sum and product types are required for modeling Haskell's **data** construct that introduces a sum

of products. For example, for the type of naturals we have: $Nat = 1 + Nat$. We treat these type constructors as if they were given by the following **data** declarations:

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

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.

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

This simple definition contains all ingredients needed to specialize *equal* for arbitrary data types. Note that the 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 (type application is translated to value application, type abstraction to value abstraction, and type fixed points to value fixed points). 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 constructor names, for example in a polytypic show function, we add one extra case to the type language: $c$ **of** $A$, where $c$ is a value of type *String* or another appropriate abstract data type for constructors, and $A$ is a value of $T_\star$. For example, for the type of naturals we have $Nat = zero$ **of** $1 + succ$ **of** $Nat$. We will not always write the constructor names in the functors for the data types. If the $c$ **of** $A$ case is omitted in the definition of a polytypic function, we assume that it is equal to the $A$ case.

$T_\star$ only defines the type language on which we *define* polytypic functions. The process of specialization is described in detail in [12].

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:

$$\begin{array}{rcl} F_{\star\to\star} & ::= & Id \mid K\ 1 \mid K\ Char \mid F_{\star\to\star} + F_{\star\to\star} \mid \\ & & F_{\star\to\star} \times F_{\star\to\star} \mid c\ \mathbf{of}\ F_{\star\to\star}, \end{array}$$

where $Id$, $K\ T$ ($T = 1$ or $Char$), '+', '×', and **of** are given



by (note that we overload the symbols '+', '×', and **of**)

$$Id \quad = \quad \Lambda A\,.\,A$$

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

Here, $\Lambda A\,.\,T$ denotes abstraction on the type level. For example, the type of lists parameterized by some type are defined by $List = K\ 1 + Id \times List$. 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 $F\ A$.

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

Using *map* we can, for instance, define generic versions of cata- and anamorphisms [27]. 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.

$$\textbf{newtype}\ Fix\ F \quad = \quad in\{\, out :: F\ (Fix\ F)\}.$$

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

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

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

## 2.2 Tries

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

$$\begin{array}{rcl} 1 \to_{\mathrm{fin}} V & \cong & V \\ (T_1 + T_2) \to_{\mathrm{fin}} V & \cong & (T_1 \to_{\mathrm{fin}} V) \times (T_2 \to_{\mathrm{fin}} V) \\ (T_1 \times T_2) \to_{\mathrm{fin}} V & \cong & T_1 \to_{\mathrm{fin}} (T_2 \to_{\mathrm{fin}} V). \end{array}$$

where $T \to_{\mathrm{fin}} V$ denotes a finite map. As $FMap\langle T\rangle\ V$, the generalization of *FMapString* given in the Introduction section, represents the set of finite maps from $T$ to $V$, the isomorphisms above can be rewritten as defining equations for $FMap\langle T\rangle$.

$$FMap\langle T :: \star\rangle \qquad :: \qquad \star \to \star$$

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

Note that $FMap\langle 1\rangle$ is *Maybe* rather than *Id* since we use the *Maybe* monad for exception handling. We assume that a suitable data structure, $FMapTChar$, and an associated look-up function $lookupTChar :: \forall V\,.\,Char \to FMapTChar\ V \to Maybe\ V$ for characters are predefined.

### 2.2.1 Look-up function

The look-up function is given by the following generic definition.

$$
\begin{array}{lcl}
lookup\langle T :: \star\rangle :: \forall V\,.\,T \to FMap\langle T\rangle\ V \to Maybe\ V & & \\
lookup\langle 1\rangle\ ()\ t & = & t \\
lookup\langle Char\rangle\ c\ t & = & lookupTChar\ c\ t \\
lookup\langle T_1 + T_2\rangle\ (inl\ k_1)\ (t_1, t_2) & = & lookup\langle T_1\rangle\ k_1\ t_1 \\
lookup\langle T_1 + T_2\rangle\ (inr\ k_2)\ (t_1, t_2) & = & lookup\langle T_2\rangle\ k_2\ t_2 \\
lookup\langle T_1 \times T_2\rangle\ (k_1, k_2)\ t & = & \\
\quad (lookup\langle T_1\rangle\ k_1 \diamond lookup\langle K_2\rangle\ k_2)\ t. & &
\end{array}
$$

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

## 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: 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 pattern functor:

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

where 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.

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

The type-indexed function *suffixes* labels a value of a data type with the subtree rooted at each node. It uses a function *add*, which adds a label to a value of type $F\ T$, giving a value of type $Label\langle F \rangle\ L\ T$.

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

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*.

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

A type-indexed data type can be viewed as an abstract data type. A value of a type-indexed data type can only be constructed by means of functions for this data type; there are no constructors for a type-indexed data type. Alternatively, the type-indexed functions that return values of type-indexed data types can be seen as constructors.



## 3 Examples of translations to Haskell

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

We illustrate the specialization of type-indexed data types by means of a translation of the digital search tries example to Haskell. This translation also shows how type-indexed data types are specialized in Generic Haskell: the translation given here will be generated by the compiler for Generic Haskell.

In the examples in the previous section we introduced three type-indexed programming concepts: type-indexed functions (*map*), type-indexed data types (*FMap*), and type-indexed functions that take arguments of type-indexed data types (*lookup*). The specialization of type-indexed functions (see Hinze [13]) is a special case of the specialization of type-indexed functions that take arguments of type-indexed data types given in this paper.

The example specializations are described in three subsections: a translation of data types, a translation of type-indexed data types, and a translation of type-indexed functions that take arguments of type-indexed data types.

### 3.1 Translating data types

A type-indexed function is translated to several functions: one for each user-defined data type on which it is used. These translated functions work on a slightly different, but isomorphic data type. This implies that values of user-defined data types have to be translated to these isomorphic data types too. For example, the type $Nat$ of natural numbers defined by

$$\textbf{data}\ Nat \quad = \quad zero \mid succ\ Nat,$$

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

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

This mapping avoids direct recursion by adding the extra layer of $Nat'$. Furthermore, it translates $n$-ary products and

sums to binary products and sums.

### 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 [13], 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. The type-indexed data type *FMap*, defined by

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

is translated to:

$$\begin{array}{l} \textbf{newtype}\ FMapUnit\ V\ = \\ \quad fMapUnit\ (Maybe\ V) \\ \textbf{newtype}\ FMapChar\ V\ = \\ \quad fMapChar\ (FMapTChar\ V) \\ \textbf{newtype}\ FMapEither\ FMA\ FMB\ V\ = \\ \quad fMapEither\ (FMA\ V, FMB\ V) \\ \textbf{newtype}\ FMapProduct\ FMA\ FMB\ V\ = \\ \quad fMapProduct\ (FMA\ (FMB\ V)). \end{array}$$

Furthermore, for each data type on which we want to use a trie we generate the trie type specialized on the data type. For example, for the type *Nat* of natural numbers we have

$$
\begin{array}{l}
\textbf{type}\ FMapNat'\ V \quad = \\
\quad FMapEither\ FMapUnit\ FMapNat\ V \\
\textbf{newtype}\ FMapNat\ V \quad = \\
\quad fMapNat\{\, unFMapNat :: FMapNat'\ V \,\}.
\end{array}
$$

Note that we use **newtype** for *FMapNat* because it is not possible to define recursive **type**s in Haskell.

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

The translation of a type-indexed function that takes an argument of a type-indexed data type is a generalization of the translation of other type-indexed functions. The translation consists of two parts: a translation of the type-indexed function, 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, for each line in its definition. For the type indices of kind $\star$ (i.e. 1, *Char*) we generate types that are instances of the type of the type-indexed function. The occurrences of the type index are replaced by the instance, and occurrences of type-indexed data types are replaced by the translation of the type-indexed data type on the type index. For example, for the type-indexed function *lookup* of type:

$$
lookup\langle T :: \star \rangle \quad :: \quad \forall V\ .\ T \to FMap\langle T \rangle\ V \to Maybe\ V,
$$

the instances are obtained by replacing $T$ by 1 or *Char*,

and by replacing $FMap\langle T\rangle$ by $FMapUnit$ or $FMapChar$, respectively. So for the function *lookup* we have that the lines

$$\begin{array}{lcl} lookup\langle 1\rangle\ ()\ t & = & t \\ lookup\langle Char\rangle\ c\ t & = & lookupTChar\ c\ t, \end{array}$$

are translated into

$$\begin{array}{l} lookupUnit :: \forall V\,.\,1 \to FMapUnit\ V \to Maybe\ V \\ lookupUnit\ ()\ (fMapUnit\ t) \quad = \quad t \end{array}$$

$$\begin{array}{l} lookupChar :: \forall V\,.\,Char \to FMapChar\ V \to Maybe\ V \\ lookupChar\ c\ (fMapChar\ t) \quad = \quad lookupTChar\ c\ t. \end{array}$$

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. $+$, $\times$) we generate types that take two functions as arguments, corresponding to the instances of the type-indexed function on the arguments of $+$ and $\times$, and return a function of the



combined type, see Hinze [13]. For example, the following lines

$$\begin{array}{lcl} lookup\langle T_1 + T_2\rangle\ (inl\ k_1)\ (t_1, t_2) & = & lookup\langle T_1\rangle\ k_1\ t_1 \\ lookup\langle T_1 + T_2\rangle\ (inr\ k_2)\ (t_1, t_2) & = & lookup\langle T_2\rangle\ k_2\ t_2 \\ lookup\langle T_1 \times T_2\rangle\ (k_1, k_2)\ t & = & \\ \quad (lookup\langle T_1\rangle\ k_1 \diamond lookup\langle K_2\rangle\ k_2)\ t & & \end{array}$$

are translated into the following functions

$$
\begin{array}{l}
lookupEither :: \forall A\ FMA\ B\ FMB\ .\\
\qquad (\forall V\ .\ A \to FMA\ V \to Maybe\ V) \to\\
\qquad (\forall V\ .\ B \to FMB\ V \to Maybe\ V) \to\\
\qquad \forall V\ .\ A + B \to FMapEither\ FMA\ FMB\ V \to\\
\qquad\quad Maybe\ V\\
lookupEither\ lua\ lub\ (inl\ a)\ (fMapEither\ (fma, fmb)) \quad =\\
\quad lua\ a\ fma\\
lookupEither\ lua\ lub\ (inr\ b)\ (fMapEither\ (fma, fmb)) \quad =\\
\quad lub\ b\ fmb
\end{array}
$$

$$
\begin{array}{l}
lookupProduct :: \forall A\ FMA\ B\ FMB\ .\\
\qquad (\forall V\ .\ A \to FMA\ V \to Maybe\ V) \to\\
\qquad (\forall V\ .\ B \to FMB\ V \to Maybe\ V) \to\\
\qquad \forall V\ .\ A \times B \to FMapProduct\ FMA\ FMB\ V \to\\
\qquad\quad Maybe\ V\\
lookupProduct\ lua\ lub\ (a, b)\ (fMapProduct\ t) \qquad\qquad =\\
\quad (lua\ a \diamond lub\ b)\ t.
\end{array}
$$

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 on each data type on which it is used. For example, on $Nat$ we have

$$
\begin{array}{l}
lookupNat \quad :: \quad \forall V\ .\ Nat \to FMapNat\ V \to Maybe\ V\\
lookupNat \quad =\\
\quad convLookupNat\ (lookupEither\ lookupUnit\ lookupNat).
\end{array}
$$

The argument of function *convLookupNat* (defined below) is generated directly from the type $Nat'$. For each special-

ization we also have to generate a conversion function. 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 that appears as the type index. For example, function *convLookupNat* converts a *lookup* function on the internal isomorphic data type used for natural numbers to a *lookup* function on the type of natural numbers itself.

$$
\begin{aligned}
&\mathit{convLookupNat} :: (\mathit{Nat}' \to \mathit{FMapNat}'\ V \to \mathit{Maybe}\ V) \to \\
&\qquad\qquad (\mathit{Nat} \to \mathit{FMapNat}\ V \to \mathit{Maybe}\ V) \\
&\mathit{convLookupNat}\ \mathit{lu} = \\
&\quad \lambda t\ \mathit{fmt} \to \mathit{lu}\ (\mathit{fromNat}\ t)\ (\mathit{unFMapNat}\ \mathit{fmt}).
\end{aligned}
$$

Note that the functions *toNat* and *fMapNat* are not used on the right-hand side of the definition of function *convLookupNat*. This is because no values of type *Nat* or *FMapNat* are built for the result of the function. If the result of the type-indexed function consists of values of the type index or the type-indexed data type these functions will be applied at the appropriate position.

### 3.4 Implementing *FMap* in Haskell directly

Alternatively, we could also use multi-parameter type classes and functional dependencies to implement a type-indexed

$$
\begin{aligned}
&\textbf{class}\ \mathit{FMap}\ \mathit{fma}\ a \mid a \to \mathit{fma}\ \textbf{where} \\
&\quad \mathit{lookup} && :: && a \to \mathit{fma}\ v \to \mathit{Maybe}\ v \\
&\textbf{instance}\ \mathit{FMap}\ \mathit{Maybe}\ ()\ \textbf{where} \\
&\quad \mathit{lookup}\ ()\ \mathit{fm} && = && \mathit{fm} \\
&\textbf{data}\ \mathit{Pair}\ f\ g\ a && = && \mathit{Pair}\ (f\ a)\ (g\ a)
\end{aligned}
$$

```
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
```

Figure 1: Implementing *FMap* in Haskell directly

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.

## 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 specialization have been given in the previous section. The specialization on concrete data type instances given in this section removes the type arguments of type-indexed data types and functions. Thus all type arguments are removed at compile-time, and type-indexed data types and functions can be used at no run-time cost. 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 [11] to the level of data types.

An alternative approach to giving a semantics of type-indexed data types is obtained by translating type-indexed data types to the intensional type analysis extension of

ML [9], in which types can be analyzed at run-time.

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 by means of specialization 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 on the specialization of type-indexed data types, and Section 4.4 on the specialization of type-indexed functions that take arguments of type-indexed data types.

## 4.1 The polymorphic lambda calculus

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

**Kind terms** Kind terms are formed by:

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



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

| | | | |
|---|---|---|---|
| $T, U \in \mathit{Type}$ | $::=$ | $C$ | type constant |
| | $\mid$ | $A$ | type variable |
| | $\mid$ | $(\Lambda A :: \mathfrak{U} \,.\, T)$ | type abstraction |
| | $\mid$ | $(T\ U)$ | type application. |

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 $\Lambda A_1 \ \ldots \ A_m \,.\, T$.

The set of type constants includes a family of fixed point operators indexed by kind: $\mathit{Fix}_{\mathfrak{T}} :: (\mathfrak{T} \to \mathfrak{T}) \to \mathfrak{T}$. In the examples, we will often omit the kind annotation $\mathfrak{T}$ in $\mathit{Fix}_{\mathfrak{T}}$.

**Type schemes** Type schemes are formed by:

| | | | |
|---|---|---|---|
| $R, S \in \mathit{Scheme}$ | $::=$ | $T$ | type term |
| | $\mid$ | $(R \to S)$ | functional type |
| | $\mid$ | $(\forall A :: \mathfrak{U} \,.\, S)$ | polymorphic type. |

**Terms** Terms are formed by:

| | | | |
|---|---|---|---|
| $t, u \in \mathit{Term}$ | $::=$ | $c$ | constant |
| | $\mid$ | $a$ | variable |
| | $\mid$ | $(\lambda a :: S \,.\, t)$ | abstraction |
| | $\mid$ | $(t\ u)$ | application |
| | $\mid$ | $(\lambda A :: \mathfrak{U} \,.\, t)$ | universal abstraction |
| | $\mid$ | $(t\ R)$ | universal application. |

Here, $\lambda A :: \mathfrak{U} \,.\, t$ denotes universal abstraction (forming a polymorphic value) and $t\ R$ denotes universal application

(instantiating a polymorphic value). Note that 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

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

and suitable functions for each of the other type constants $C$ (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:

$$\begin{array}{lcl} \mathit{cata}\langle 1 \rangle & = & \mathit{cata}_1 \\ \mathit{cata}\langle \mathit{Char} \rangle & = & \mathit{cata}_{\mathit{Char}} \\ \mathit{cata}\langle T_1 + T_2 \rangle & = & \mathit{cata}_+ \ (\mathit{cata}\langle T_1 \rangle) \ (\mathit{cata}\langle T_2 \rangle) \\ \mathit{cata}\langle T_1 \times T_2 \rangle & = & \mathit{cata}_\times \ (\mathit{cata}\langle T_1 \rangle) \ (\mathit{cata}\langle T_2 \rangle). \end{array}$$

These equations implicitly define the (family of) functions

$cata_1$, $cata_{Char}$, $cata_+$, and $cata_\times$. In this section we will assume that a type-indexed function and data type are explicitly defined as a catamorphism. For example, for digital search tries we have

$$\begin{array}{lcl} FMap_1 & = & \Lambda V\,.\,Maybe\ V \\ FMap_{Char} & = & \Lambda V\,.\,FMapTChar\ V \\ FMap_+ & = & \Lambda FMap_A\ FMap_B\ V\,.\,FMap_A\ V \times FMap_B\ V \\ FMap_\times & = & \Lambda FMap_A\ FMap_B\ V\,.\,FMap_A\ (FMap_B\ V), \end{array}$$

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 [26], and are characterized by:

$$\begin{array}{lcl} para\langle 1\rangle & = & para_1 \\ para\langle Char\rangle & = & para_{Char} \\ para\langle T_1 + T_2\rangle & = & para_+\ T_1\ T_2\ (para\langle T_1\rangle)\ (para\langle T_2\rangle) \\ para\langle T_1 \times T_2\rangle & = & para_\times\ T_1\ T_2\ (para\langle T_1\rangle)\ (para\langle T_2\rangle). \end{array}$$

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 $C$ of type constants: 1, *Char*, $+$, and $\times$. However, we have also used the type indices *Id*, $K\ 1$, $K\ Char$, and lifted versions of $+$ and $\times$. So 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 Section 4.3. The process described in that section has to be lifted to higher-order type indices. For the details of of this lifting process, see Hinze [11], where it is also shown that this approach is limited to types of second-order kinds.

### 4.3 Specializing type-indexed data types

The process of specialization of type-indexed data types on concrete data type instances is phrased as an interpretation of the simply typed lambda calculus. The interpretation of the constants (1, *Char*, + and ×) is obtained from the definition of the type-indexed data type as a catamorphism. Type application is interpreted as type application (in a different domain), abstraction as abstraction, and fixed points as fixed points.

In the previous sections, the 'types' of type-indexed data types are of 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 $\star \to \star$, it is not in the domain of *FMap*. We 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:

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

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

$$\begin{array}{lcl} \mathfrak{Data}_{\mathfrak{T}::\Box} & :: & \Box \\ \mathfrak{Data}_{\star} & = & \ldots \\ \mathfrak{Data}_{\mathfrak{A}\to\mathfrak{B}} & = & \mathfrak{Data}_{\mathfrak{A}} \to \mathfrak{Data}_{\mathfrak{B}}. \end{array}$$

Here $\Box$ is the superkind: the type of kinds. Note that only the definition of $\mathfrak{Data}_{\star}$, indicated by $\ldots$, 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:

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

As described above, the process of specialization of type-indexed data types on concrete data type instances is phrased as an interpretation of the simply typed lambda calculus. The interpretation of the constants ($1, Char$, $+$ and $\times$) is obtained from the definition of the type-indexed data type as a catamorphism, and the interpretation of application, abstraction, and fixed points is obtained from the definition of an environment model [28] for the type-indexed data type.

An environment model is an applicative structure $(\mathsf{M}, \mathbf{app}, \mathbf{const})$, where $\mathsf{M}$ is the domain of the structure, **app** a mapping that interprets functions, and **const** maps con-

stants to the domain of the structure. Furthermore, in order to qualify as an environment model, an applicative structure has to be extensional and to satisfy the combinatory model condition. The precise definitions of these concepts can be found in Mitchell [28], where environment models are used to give semantics to the simply typed lambda calculus. For an arbitrary type-indexed data type $Data_{T::\mathfrak{T}} :: \mathfrak{Data}_{\mathfrak{T}}$ we use the following applicative structure:

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

The domain of the applicative structure for a kind $\mathfrak{T}$ is the equivalence class, under an appropriate set of equations $\mathcal{E}$ between type terms, of the set of types of kind $\mathfrak{Data}_{\mathfrak{T}}$. 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 of the type-indexed data type.

It can be verified that the applicative structure defined thus is an environment model.

The family of functions obtained from the formulation as a catamorphism of the type-indexed data type does not contain a component for the fixed point. The interpretation of fixed points is the same for different type-indexed data types:

$$\mathbf{const}(Fix_{\mathfrak{T}}) \quad = \quad [\,Data_{Fix_{\mathfrak{T}}}\,],$$

where $Data_{Fix_{\mathfrak{T}}}$ is defined by

$$Data_{Fix_{\mathfrak{T}}} \quad = \quad \lambda D :: \mathfrak{Data}_{\mathfrak{T}\to\mathfrak{T}} \,.\, Fix_{\mathfrak{Data}_{\mathfrak{T}}}\ D.$$

### 4.4 Specializing type-indexed values

A type-indexed value possesses a kind-indexed type [13],

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

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

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

Again, note that only the definition of $Poly_{\star}$ 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 [13] $Poly_{\mathfrak{T}}$ 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:

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

where $\mathfrak{Id}$ is the identity function on kinds. From the definition of the look-up function in Section 2.2.1 we obtain the following equations:

$$
\begin{array}{lcl}
lookup_{T::\mathfrak{T}} & :: & Lookup_{\mathfrak{T}}\ Id_T\ FMap_T \\
lookup_1 & = & \lambda V\ k\ fmk\,.\,fmk \\
lookup_{Char} & = & lookupTChar \\
lookup_+ & = & \lambda A\ FMA\ lookup_A\ B\ FMB\ lookup_B\,. \\
 & & \lambda V\ k\ fmk\,. \\
 & & \mathbf{case}\ k\ \mathbf{of}\ \{inl\ a \to lookup_A\ V\ a\ (outl\ fmk) \\
 & & \quad ; inr\ b \to lookup_B\ V\ b\ (outr\ fmk) \\
 & & \quad \} \\
lookup_\times & = & \lambda A\ FMA\ lookup_A\ B\ FMB\ lookup_B\,. \\
 & & \lambda V\ k\ fmk\,. \\
 & & (lookup_A\ V\ (outl\ k) \diamond lookup_B\ V\ (outr\ k)) \\
 & & \quad fmk.
\end{array}
$$

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 environment model used for the specialization of type-indexed data types. 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.

$$
\begin{array}{lcl}
\mathbf{M}^{\mathfrak{T}} & = & ([D_1] \in Scheme^{\mathfrak{Data}^1_{\mathfrak{T}}} \ / \ \mathcal{E} \\
 & & , \ldots \\
 & & , [D_n] \in Scheme^{\mathfrak{Data}^n_{\mathfrak{T}}} \ / \ \mathcal{E} \\
 & & ; Term^{Poly_{\mathfrak{T}}\ D_1\ \ldots\ D_n} \ / \ \mathcal{E} \\
 & & )
\end{array}
$$

$$
\begin{array}{l}
\mathbf{app}_{\mathfrak{T},\mathfrak{U}}\ ([R_1], \ldots, [R_n]; [t])\ ([S_1], \ldots, [S_n]; [u]) \\
\qquad = \quad ([R_1\ S_1], \ldots, [R_n\ S_n]; [t\ S_1\ \ldots\ S_n\ u])
\end{array}
$$

$$
\mathbf{const}(C) \quad = \quad ([Data^1_C], \ldots, [Data^n_C]; [poly_C]).
$$

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

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

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

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

## 5 An advanced example: the Zipper

This section shows how to define a zipper for an arbitrary data type. This is a rather involved example of a type-indexed data type, together with some type-indexed functions that take an argument of a type-indexed data type.

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 in which a user interactively manipulates trees, such as editors for structured documents such as proofs and programs. The focus of the zipper may only move to (sub-) trees, so, for example in the data type *Tree*:

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

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 the data type definitions. The zipper data structure is then defined by induction on the 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 or tree 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. We will not define edit functions, and we will not further specify the edit loop of the tools in which the zipper is used.

## 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.

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

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 is it 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.

$$
\begin{array}{lcl}
Ctx\langle F :: \star \to \star \rangle & :: & \star \to \star \to \star \\
Ctx\langle Id \rangle & = & \Lambda C\ R\,.\,C \\
Ctx\langle K\ 1 \rangle & = & \Lambda C\ R\,.\,0 \\
Ctx\langle K\ Char \rangle & = & \Lambda C\ R\,.\,0 \\
Ctx\langle F_1 + F_2 \rangle & = & \Lambda C\ R\,.\,Ctx\langle F_1 \rangle\ C\ R + Ctx\langle F_2 \rangle\ C\ R \\
Ctx\langle F_1 \times F_2 \rangle & = & \\
\multicolumn{3}{c}{\Lambda C\ R\,.\,(Ctx\langle F_1 \rangle\ C\ R \times F_2\ R) + (F_1\ R \times Ctx\langle F_2 \rangle\ C\ R).}
\end{array}
$$

This definition can be understood as follows. There are three simple cases: the sum case, and the constant cases $K\ 1$ and $K\ Char$. 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, we do allow it to 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. 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 $K\ 1 + Id$ (or, equivalently, $\Lambda N\,.\,1 + N$), and for trees of type *Bush* whose pattern functor is $K\ Char + Id \times Id$ (or, equivalently, $\Lambda T\,.\,Char + (T \times T)$) we obtain

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

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 *CtxBush* introduced in Section 1.

We recently found that McBride [24] also defines a type-indexed zipper data type. His zipper slightly deviates from Huets 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. If the current node is a leaf node such a child does not exist, and then function *down* returns the location unchanged. The instantiation of function *down* on the data type *Bush* has been given in Section 1. Function *down* satisfies the following property:

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



that is, first going down the tree and then up again is the identity function on locations in which it is possible to go down. Since function *down* moves down to the leftmost recursive child of the current node, the equality $down\langle F\rangle \cdot up\langle F\rangle = id$ does not hold in general. However, there does exist a natural number $n$ such that

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

Function *down* is defined as follows.

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

To find the leftmost recursive child, we have to pattern match on the pattern functor $F$, and find the first occurrence of *Id*. 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. 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.

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

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

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

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.

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

Remember that *inl* () denotes the empty top context. Function *up* uses two helper functions: *insert* and *extract*. Function *extract* returns the context of the parent of the current node. Note that each element of type $Ctx\langle F \rangle\ C\ T$ 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.

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

where *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.

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

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

Since $\mathit{up}\langle F \rangle \cdot \mathit{down}\langle F \rangle = \mathit{id}$ 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

$$
\begin{array}{lll}
\mathbf{do} & \{ & (t, c') \leftarrow \mathit{first}\langle F \rangle\ \mathit{ft}\ c \\
 & ; & c'' \leftarrow \mathit{extract}\langle F \rangle\ c' \\
 & ; & \mathit{ft}' \leftarrow \mathit{insert}\langle F \rangle\ c'\ t \\
 & ; & \mathit{return}\ c == c'' \wedge \mathit{ft} == \mathit{ft}'
\end{array}
$$

}

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

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

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

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).
```

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*,

$$\begin{array}{l} next\langle F :: \star \to \star\rangle, previous\langle F :: \star \to \star\rangle :: \\ \quad T \to Ctx\langle F\rangle\ C\ T \to Maybe\ (T, Ctx\langle F\rangle\ C\ T) \end{array}$$

such that

$$\begin{array}{lll} \textbf{do} & \{ & (t', c') \leftarrow next\langle F\rangle\ t\ c \\ & ; & (t'', c'') \leftarrow previous\langle F\rangle\ t'\ c' \\ & ; & return\ c == c'' \wedge t == t'' \\ & \} & \end{array}$$

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*.

$$\begin{array}{l} next\langle F :: \star \to \star\rangle :: T \to Ctx\langle F\rangle\ C\ T \to Maybe\ (T, Ctx\langle F\rangle\ C\ T) \\ next\langle Id\rangle\ t\ c = fail \\ next\langle K\ 1\rangle\ t\ c = fail \\ next\langle K\ Char\rangle\ t\ c = fail \\ next\langle F_1 + F_2\rangle\ t\ (inl\ cx) = \\ \quad \textbf{do}\ \{(t', cx') \leftarrow next\langle F_1\rangle\ t\ cx; return\ (t', inl\ cx')\} \\ next\langle F_1 + F_2\rangle\ t\ (inr\ cy) = \\ \quad \textbf{do}\ \{(t', cy') \leftarrow next\langle F_2\rangle\ t\ cy; return\ (t', inr\ cy')\} \\ next\langle F_1 \times F_2\rangle\ t\ (inl\ (cx, y)) = \\ \quad \textbf{do}\ \{(t', cx') \leftarrow next\langle F_1\rangle\ t\ cx; return\ (t', inl\ (cx', y))\} \\ \quad +\!\!+ \textbf{do}\ \{c \leftarrow extract\langle F_1\rangle\ cx \end{array}$$

$$
\begin{array}{l}
\quad ; x \leftarrow \mathit{insert}\langle F_1\rangle\ cx\ t \\
\quad ; (t', cy) \leftarrow \mathit{first}\langle F_2\rangle\ y\ c \\
\quad ; \mathit{return}\ (t', \mathit{inr}\ (x, cy)) \\
\quad \} \\
\mathit{next}\langle F_1 \times F_2\rangle\ t\ (\mathit{inr}\ (x, cy)) \quad = \\
\quad \mathbf{do}\ \{(t', cy') \leftarrow \mathit{next}\langle F_2\rangle\ t\ cy; \mathit{return}\ (t', \mathit{inr}\ (x, cy'))\}.
\end{array}
$$

The first three lines in this definition show that it is impossible to go 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 interesting case of function *next*. 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, for digital search tries, 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 specialization is a generalization of the specialization of type-indexed functions given in Hinze [13], and is used in the extension of Generic Haskell with a construct for defining type-indexed data types. Please contact the authors for the current, experimental, implementation of Generic Haskell with type-indexed data types. The first release of Generic Haskell is planned for November 1, 2001, see also `http://www.generic-haskell.org/`.

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 some things that remain to be done. We want to test our framework on the type-indexed data types appearing in the literature [3, 31], and we want to create a library of type-indexed data type examples. Furthermore, we have to investigate how we can deal with sets of possibly mutually recursive type-indexed data types.

**Acknowledgements.** Thanks to Dave Clarke, Ralf Lämmel, and Doaitse Swierstra for comments on a previous version 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] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *22nd Symposium on Principles of Programming Languages, POPL '95*, pages 118–129, 1995.

[8] M.M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.

[9] 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.

[10] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.

[11] Ralf Hinze. *Generic Programs and Proofs.* 2000. Habilitationsschrift, Bonn University.

[12] 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.

[13] 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.

[14] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[15] 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.

[16] P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998. Available from the Polytypic programming WWW page [17].

[17] Patrik Jansson. The WWW home page for polytypic programming. Available from `http://www.cs.chalmers.se/~patrikj/poly/`, 2001.

[18] Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.

[19] 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.

[20] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[21] 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.

[22] Ralf Lämmel and Wolfgang Lohmann. Format evolution. In *RETIS'01*, 2001.

[23] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[24] Connor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.

[25] Nancy J. McCracken. *An investigation of a programming language with a polymorphic type structure.* PhD thesis, Syracuse University, June 1979.

[26] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[27] 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.

[28] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.

[29] 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.

[30] 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.

[31] Måns Vestin. Genetic algorithms in Haskell with polytypic programming. Examensarbeten 1997:36, Göteborg University, Gothenburg, Sweden, 1997. Available from the Polytypic programming WWW page [17].

[32] Stephanie Weirich. Encoding intensional type analysis. In *European Symposium on Programming*, volume 2028 of *LNCS*, pages 92–106. Springer-Verlag, 2001.

[33] Zhe Yang. Encoding types in ML-like languages. In *Proceedings ICFP 1998: International Conference on Functional Programming*, pages 289–300. ACM Press, 1998.