Generic Haskell: applications

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

[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
johanj@cs.uu.nl
http://www.cs.uu.nl/~johanj/

[3] Open University, Heerlen, The Netherlands

Abstract. [1]
Generic Haskell is an extension of Haskell that supports the construction of generic programs. This article describes generic programming in practice. It discusses three advanced generic programming applications: generic dictionaries, compressing XML documents, and the zipper. When describing and implementing these examples, we will encounter some advanced features of Generic Haskell, such as type-indexed data types, dependencies between and generic abstractions of generic functions, adjusting a generic function using a default case, and generic functions with a special case for a particular constructor.

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 [50], such as *show*, *read*, and '=='. In [23] we have introduced type-indexed functions, and we have shown how to implement them in Generic Haskell [7]. For an older introduction to generic programming, see Backhouse et al [4].

Why is generic programming important? Generic programming makes programs easier to write:

- Programs that could only be written in an untyped style can now be written in a language with types.
- Some programs come for free.
- Some programs are simple adjustments of library functions, instead of complicated traversal functions.

[1] This is a preliminary version of the notes that will appear in the Lecture Notes of the Summer School on Generic Programming.

Of course not all programs become simpler when you write your programs in a generic programming language, but, on the other hand, no programs become more complicated. In this paper we will try to give you a feeling about where and when generic programs are useful.

This article describes three advanced generic programming applications: generic dictionaries, compressing XML documents (and XML tools in general), and the zipper. The example applications are described in more detail below. In the examples, we will encounter several new generic programming concepts:

- *Type-indexed data types.* A type-indexed data type is constructed in a generic way from an argument data type [24]. It is the equivalent of type-indexed functions on the level of data types.
- *Default cases.* To define a generic function that is the same as another function except for a few cases we use a default case [8]. If the new definition does not provide a certain case, then the default case applies and copies the case from another function.
- *Constructor cases.* A constructor case of a generic program deals with a constructor of a data type that requires special treatment [8]. Constructor cases are especially useful when dealing with data types with a large number of constructors, and only a small number of constructors need special treatment.
- *Dependencies and generic abstractions.* To write a generic function that uses another generic function we can use a dependency or a generic abstraction [8].

We will introduce these concepts where and when we need them.

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

$$\mathbf{data}\ \mathit{String} = \mathit{NilS} \mid \mathit{ConsS}\ \mathit{Char}\ \mathit{String},$$

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

$$\begin{aligned}\mathbf{data}\ \mathit{FMapString}\ v = {} & \mathit{NullString}\\ \mid {} & \mathit{TrieString}\ (\mathit{Maybe}\ v)\ (\mathit{FMapChar}\ (\mathit{FMapString}\ v))\end{aligned}$$

Such a trie for strings would typically be used for a concordance or another index on texts. The first component of the constructor *TrieString* contains the value associated with *NilS*. The second component of *TrieString* is derived from the constructor $\mathit{ConsS} :: \mathit{Char} \to \mathit{String} \to \mathit{String}$. 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. Given these prerequisites we can define a look-up function for strings as follows:

$$\begin{array}{l} lookupString :: String \to FMapString\ v \to Maybe\ v \\ lookupString\ NilS\ (TrieString\ tn\ tc) \qquad = tn \\ lookupString\ (ConsS\ c\ s)\ (TrieString\ tn\ tc) = \\ \quad (lookupChar\ c \diamond lookupString\ s)\ tc. \end{array}$$

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

$$\begin{array}{l} (\diamond) :: (a \to Maybe\ b) \to (b \to Maybe\ c) \to a \to Maybe\ c \\ (f \diamond g)\ a = \mathbf{case}\ f\ a\ \mathbf{of}\ \{Nothing \to Nothing; Just\ b \to g\ b\}. \end{array}$$

In the following section we will show how to define a trie and an associated look-up function for an arbitrary data type.

Example 2: Compressing XML documents. XML documents may become (very) large because of the markup that is added to the content. Because of the repetitive structure of many XML documents, these documents can be compressed by quite a large factor.

An XML document is usually structured according to a DTD (Document Type Definition), a specification that describes which tags may be used in the XML document, and in which positions and order they have to be. A DTD is, in a way, the *type* of an XML document. An XML document is called *valid* with respect to a certain DTD if it follows the structure that is specified by that DTD. An XML compressor can use information from the DTD to obtain better compression. For example, consider the following small XML file:

```
<book lang="English">
<title>  Dead Famous </title>
<author> Ben Elton   </author>
<date>   2001        </date>
</book>
```

This file may be compressed by separating the structure from the data, and compressing the two parts separately. For compressing the structure we can make good use of the DTD. If we know how many elements, say n, appear in the DTD (the DTD for the above document contains at least 4 elements), we can replace each occurrence of the markup of an element in an XML file which is valid with respect to the DTD by $log_2\ n$ bits. This simple idea is the main idea behind the tool described in Section 3, and has been described in the context of data conversion by Jansson and Jeuring [31, 35].

In Section 3 we use HaXml [58] to translate a DTD to a data type, and we construct generic functions for separating the contents (the strings) and the shape (the constructors) of a value of a data type, and for encoding the shape of a value of a data type using information about the (number of) constructors of the data type.

XML compressors ar just one class of XML tools that are easily implemented as generic programs. Other XML tools that can be implemented as generic programs are XML editors, XML databases, and XML version management tools.

Example 3: Zipper. The zipper [27] 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 *Tree* and its corresponding zipper, called *Loc_Tree*, are defined by

$$\begin{aligned}
&\mathbf{data}\ \mathit{Tree} &&= \mathit{Leaf}\ \mathit{Char} \mid \mathit{Fork}\ \mathit{Tree}\ \mathit{Tree}\\
&\mathbf{type}\ \mathit{Loc_Tree} &&= (\mathit{Tree}, \mathit{Context_Tree})\\
&\mathbf{data}\ \mathit{Context_Tree} &&= \mathit{top}\\
& &&\mid \mathit{forkL}\ \mathit{Context_Tree}\ \mathit{Tree}\\
& &&\mid \mathit{forkR}\ \mathit{Tree}\ \mathit{Context_Tree}.
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{down_Tree} &&:: \mathit{Loc_Tree} \to \mathit{Loc_Tree}\\
&\mathit{down_Tree}\ (\mathit{Leaf}\ a, c) &&= (\mathit{Leaf}\ a, c)\\
&\mathit{down_Tree}\ (\mathit{Fork}\ \mathit{tl}\ \mathit{tr}, c) &&= (\mathit{tl}, \mathit{forkL}\ c\ \mathit{tr})\\
&\mathit{right_Tree} &&:: \mathit{Loc_Tree} \to \mathit{Loc_Tree}\\
&\mathit{right_Tree}\ (\mathit{tl}, \mathit{forkL}\ c\ \mathit{tr}) &&= (\mathit{tr}, \mathit{forkR}\ \mathit{tl}\ c)\\
&\mathit{right_Tree}\ l &&= l.
\end{aligned}$$

The navigator function *down_Tree* moves the focus of attention to the *leftmost* subtree of the current node; *right_Tree* moves the focus to its right sibling.

Huet [27] defines the zipper data structure for rose trees and for the data type *Tree*, and gives the generic construction in words. In Section 4 we describe the zipper in more detail and show how to define a zipper for an arbitrary data type.

Other applications of generic programming. Besides the applications mentioned in the examples above, there are several application areas in which generic programming can be used.

- *Haskell's* `deriving` *construct.* Haskell's `deriving` construct is used to generate code for for example the equality function, and for functions for reading and showing values of data types. Only the classes `Eq`, `Ord`, `Enum`, `Bounded`, `Show` and `Read` can be derived. The definitions of most of the derived functions can be found in the library of Generic Haskell.
- *Compiler functions.* Several functions that are used in compilers are generic functions: garbage collectors, tracers, debuggers, etc.
- *Typed term processing.* Functions like pattern matching, term rewriting and unification are generic functions, and have been implemented as generic functions in [36, 33, 34].

The form and functionality of these applications is exactly determined by the structure of the input data.

Maybe the most common applications of generic programming can be found in functions that traverse data built from rich mutually-recursive data types with many constructors, and which perform computations on a single (or a couple of) constructor(s). For example, consider a function which traverses an abstract syntax tree and returns the free variables in the tree. Only for the variable constructor something special has to be done, in all other cases the variables collected at the children have to be passed on to the parent. This function can be defined as an instance of a Generic Haskell library function *crush* [45], together with a special constructor case for variables [8].

Organization. The rest of this paper is organized as follows. Section 2 introduces generic dictionaries, and implements them in Generic Haskell. Section 3 describes how generic programming can be used to construct XML tools. In particular, it describes XComprez, a compressor for XML documents. Section 4 develops a generic zipper data structure. Finally, Section 5 summarizes the main points and concludes.

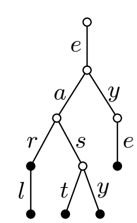
2 Generic dictionaries

A trie is a search tree scheme that employs the structure of search keys to organize information. Tries were originally devised as a means to represent a collection of records indexed by strings over a fixed alphabet. Based on work by Wadsworth and others, Connelly et al. [10] generalized the concept to permit indexing by elements built according to an arbitrary signature. In this section we go one step further and define tries and operations on tries generically for arbitrary data types of arbitrary kinds, including parameterized and nested data types.

The Generic Haskell code for this section can be downloaded from the applications page on `http://www.generic-haskell.org/`.

2.1 Introduction

The concept of a trie was introduced by Thue in 1912 as a means to represent a set of strings, see [39]. In its simplest form a trie is a multiway branching tree where each edge is labelled with a character. For example, the set of strings { *ear*, *earl*, *east*, *easy*, *eye* } is represented by the trie depicted on the right. Searching in a trie starts at the root and proceeds by traversing the edge that matches the first character, then traversing the edge that matches the second character, and so forth. The search key is a member of the represented set if the search stops in a node that is marked—marked nodes are drawn as filled circles on the right. Tries can also be used to represent finite maps. In this case marked



nodes additionally contain values associated with the strings. Interestingly, the move from sets to finite maps is not a mere variation of the scheme. As we shall see it is essential for the further development.

On a more abstract level a trie itself can be seen as a composition of finite maps. Each collection of edges descending from the same node constitutes a finite map sending a character to a trie. With this interpretation in mind it is relatively straightforward to devise an implementation of string-indexed tries. If strings are defined by the following data type:

$$\textbf{data}\ \mathit{String} = \mathit{NilS} \mid \mathit{ConsS}\ \mathit{Char}\ \mathit{String},$$

we can represent string-indexed tries with associated values of type v as follows.

$$\begin{aligned}\textbf{data}\ \mathit{FMapString}\ v &= \mathit{NullString}\\ &\mid\ \mathit{TrieString}\ (\mathit{Maybe}\ v)\ (\mathit{FMapChar}\ (\mathit{FMapString}\ v))\end{aligned}$$

Here, *NullString* represents the empty trie. The first component of the constructor *TrieString* contains the value associated with *NilS*. Its type is *Maybe* v instead of v since *NilS* may not be in the domain of the finite map represented by the trie. In this case the first component equals *Nothing*. The second component corresponds to the edge map. To keep the introductory example manageable we implement *FMapChar* using ordered association lists.

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{type}\ \mathit{FMapChar}\ v = [(\mathit{Char}, v)]}\\
\mathit{lookupChar} & :: \forall v\,.\, \mathit{Char} \to \mathit{FMapChar}\ v \to \mathit{Maybe}\ v\\
\mathit{lookupChar}\ c\ [\,] & = \mathit{Nothing}\\
\multicolumn{2}{l}{\mathit{lookupChar}\ c\ ((c', v) : x)}\\
\quad \mid c < c' & = \mathit{Nothing}\\
\quad \mid c == c' & = \mathit{Just}\ v\\
\quad \mid c > c' & = \mathit{lookupChar}\ c\ x
\end{array}$$

Note that *lookupChar* has result type *Maybe* v. If the key is not in the domain of the finite map, *Nothing* is returned.

Building upon *lookupChar* we can define a look-up function for strings. To look up the empty string we access the first component of the trie. To look up a non-empty string, say, *ConsS* c s we look up c in the edge map obtaining a trie, which is then recursively searched for s.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{lookupString} :: \forall v\,.\, \mathit{String} \to \mathit{FMapString}\ v \to \mathit{Maybe}\ v}\\
\mathit{lookupString}\ s\ \mathit{NullString} & = \mathit{Nothing}\\
\mathit{lookupString}\ \mathit{NilS}\ (\mathit{TrieString}\ tn\ tc) & = tn\\
\mathit{lookupString}\ (\mathit{ConsS}\ c\ s)\ (\mathit{TrieString}\ tn\ tc) & =\\
\quad (\mathit{lookupChar}\ c \diamond \mathit{lookupString}\ s)\ tc &
\end{array}$$

In the last equation we use monadic composition to take care of the error signal *Nothing*.

Based on work by Wadsworth and others, Connelly et al. [10] have generalized the concept of a trie to permit indexing by elements built according to an arbitrary signature, that is, by elements of an arbitrary non-parameterized data type. The definition of *lookupString* already gives a clue what a suitable generalization might look like: the trie *TrieString tn tc* contains a finite map for each constructor of the data type *String*; to look up *ConsS c s* the look-up functions for the components, c and s, are composed. Generally, if we have a data type with k constructors, the corresponding trie has k components. To look up a constructor with n fields, we must select the corresponding finite map and compose n look-up functions of the appropriate types. If a constructor has no fields (such as *NilS*), we extract the associated value.

As a second example, consider the data type of external search trees:

$$\textbf{data}\ \mathit{Dict} = \mathit{Leaf}\ \mathit{String} \mid \mathit{Node}\ \mathit{Dict}\ \mathit{String}\ \mathit{Dict}.$$

A trie for external search trees represents a finite map from *Dict* to some value type v. It is an element of *FMapDict* v given by

$$\begin{aligned}\textbf{data}\ \mathit{FMapDict}\ v = {} & \mathit{NullDict} \\ \mid {} & \mathit{TrieDict}\ (\mathit{FMapString}\ v) \\ & \qquad (\mathit{FMapDict}\ (\mathit{FMapString}\ (\mathit{FMapDict}\ v))).\end{aligned}$$

Note that *FMapDict* is a nested data type, since the recursive call on the right hand side, *FMapDict* (*FMapString* (*FMapDict* v)), is a substitution instance of the left hand side. Consequently, the look-up function on external search trees requires polymorphic recursion.

$$\begin{aligned}&\mathit{lookupDict} && :: \forall v\,.\, \mathit{Dict} \to \mathit{FMapDict}\ v \to \mathit{Maybe}\ v \\ &\mathit{lookupDict}\ d\ \mathit{NullDict} && = \mathit{Nothing} \\ &\mathit{lookupDict}\ (\mathit{Leaf}\ s)\ (\mathit{TrieDict}\ tl\ tn) && = \mathit{lookupString}\ s\ tl \\ &\mathit{lookupDict}\ (\mathit{Node}\ l\ s\ r)\ (\mathit{TrieDict}\ tl\ tn) && = \\ &\quad (\mathit{lookupDict}\ l \diamond \mathit{lookupString}\ s \diamond \mathit{lookupDict}\ r)\ tn &&\end{aligned}$$

Looking up a node involves two recursive calls. The first, *lookupDict* l, is of type $\mathit{Dict} \to \mathit{FMapDict}\ X \to \mathit{Maybe}\ X$ where $X = \mathit{FMapString}\ (\mathit{FMapDict}\ v)$, which is a substitution instance of the declared type.

Note that it is absolutely necessary that *FMapDict* and *lookupDict* are parametric with respect to the codomain of the finite maps. Had we restricted the type of *lookupDict* to $\mathit{Dict} \to \mathit{FMapDict}\ T \to T$ for some fixed type T, the definition would have no longer type-checked. This also explains why the construction does not work for the finite set abstraction.

Generalized tries make a particularly interesting application of generic programming. The central insight is that a trie can be considered as a *type-indexed data type*. This makes it possible to define tries and operations on tries generically for arbitrary data types. We already have the necessary prerequisites at hand: we know how to define tries for sums and for products. A trie for a sum is essentially a product of tries and a trie for a product is a composition of tries.

The extension to arbitrary data types is then uniquely defined. Mathematically speaking, generalized tries are based on the following isomorphisms.

$$\begin{aligned} 1 \rightarrow_{\text{fin}} v &\cong v \\ (t_1 + t_2) \rightarrow_{\text{fin}} v &\cong (t_1 \rightarrow_{\text{fin}} v) \times (t_2 \rightarrow_{\text{fin}} v) \\ (t_1 \times t_2) \rightarrow_{\text{fin}} v &\cong t_1 \rightarrow_{\text{fin}} (t_2 \rightarrow_{\text{fin}} v) \end{aligned}$$

Here, $t \rightarrow_{\text{fin}} v$ denotes the set of all finite maps from t to v. Note that $t \rightarrow_{\text{fin}} v$ is sometimes written $v^{[t]}$, which explains why these equations are also known as the 'laws of exponentials'.

2.2 Signature

To put the above idea in concrete terms we will define a type-indexed data type *FMap*, which has the following type for types t of kind $\star$.

$$\mathit{FMap}\langle t :: \star\rangle :: \star \rightarrow \star,$$

So *FMap* assigns a type constructor of kind $\star \rightarrow \star$ to each key type t of kind $\star$.

We will implement the following operations on tries.

$$\begin{aligned} &\mathit{empty}\langle t\rangle &&:: \forall v\,.\,\mathit{FMap}\langle t\rangle\ v \\ &\mathit{isempty}\langle t\rangle &&:: \forall v\,.\,\mathit{FMap}\langle t\rangle\ v \rightarrow \mathit{Bool} \\ &\mathit{single}\langle t\rangle &&:: \forall v\,.\,t \times v \rightarrow \mathit{FMap}\langle t\rangle\ v \\ &\mathit{lookup}\langle t\rangle &&:: \forall v\,.\,t \rightarrow \mathit{FMap}\langle t\rangle\ v \rightarrow \mathit{Maybe}\ v \\ &\mathit{insert}\langle t\rangle &&:: \forall v\,.\,(v \rightarrow v \rightarrow v) \rightarrow t \times v \rightarrow (\mathit{FMap}\langle t\rangle\ v \rightarrow \mathit{FMap}\langle t\rangle\ v) \\ &\mathit{merge}\langle t\rangle &&:: \forall v\,.\,(v \rightarrow v \rightarrow v) \rightarrow (\mathit{FMap}\langle t\rangle\ v \rightarrow \mathit{FMap}\langle t\rangle\ v \rightarrow \mathit{FMap}\langle t\rangle\ v) \\ &\mathit{delete}\langle t\rangle &&:: \forall v\,.\,t \rightarrow (\mathit{FMap}\langle t\rangle\ v \rightarrow \mathit{FMap}\langle t\rangle\ v) \end{aligned}$$

The value $\mathit{empty}\langle t\rangle$ is the empty trie. Function $\mathit{isempty}\langle t\rangle$ takes a trie and determines whether or not it is empty. Function $\mathit{single}\langle t\rangle\ (t, v)$ constructs a trie that contains the binding (t, v) as its only element. The function $\mathit{lookup}\langle t\rangle$ takes a key and a trie and looks up the value associated with the key. The function $\mathit{insert}\langle t\rangle$ inserts a new binding into a trie and $\mathit{merge}\langle t\rangle$ combines two tries. Function $\mathit{delete}\langle t\rangle$ takes a key and a trie, and removes the binding for the key from the trie. The two functions $\mathit{insert}\langle t\rangle$ and $\mathit{merge}\langle t\rangle$ take as a first argument a so-called *combining function*, which is applied whenever two bindings have the same key. For instance, $\lambda \mathit{new}\ \mathit{old} \rightarrow \mathit{new}$ is used as the combining function for $\mathit{insert}\langle t\rangle$ if the new binding is to override an old binding with the same key. For finite maps of type $\mathit{FMap}\langle t\rangle\ \mathit{Int}$ addition may also be a sensible choice. Interestingly, we will see that the combining function is not only a convenient feature for the user; it is also necessary for defining $\mathit{insert}\langle t\rangle$ and $\mathit{merge}\langle t\rangle$ generically for all types!

2.3 Type-indexed tries

We have already noted that generalized tries are based on the laws of exponentials.

$$\begin{aligned} 1 \to_{\text{fin}} v &\cong v \\ (t_1 + t_2) \to_{\text{fin}} v &\cong (t_1 \to_{\text{fin}} v) \times (t_2 \to_{\text{fin}} v) \\ (t_1 \times t_2) \to_{\text{fin}} v &\cong t_1 \to_{\text{fin}} (t_2 \to_{\text{fin}} v) \end{aligned}$$

In order to define the notion of finite map it is customary to assume that each value type v contains a distinguished element or *base point* $\perp_v$, see [10]. A finite map is then a function whose value is $\perp_v$ for all but finitely many arguments. For the implementation of tries it is, however, inconvenient to make such a strong assumption (though one could use type classes for this purpose). Instead, we explicitly add a base point when necessary motivating the following definition of *FMap*:

$$\begin{aligned} &\textbf{type}\ \mathit{FMap}\langle \mathit{Unit}\rangle\ v &&= \mathit{FMUnit}\ (\mathit{Maybe}\ v) \\ &\textbf{type}\ \mathit{FMap}\langle \mathit{Char}\rangle\ v &&= \mathit{FMChar}\ (\mathit{FMapChar}\ v) \\ &\textbf{type}\ \mathit{FMap}\langle \mathit{Int}\rangle\ v &&= \mathit{FMInt}\ (\mathit{Patricia.Dict}\ v) \\ &\textbf{type}\ \mathit{FMap}\langle \texttt{:+:}\rangle\ \mathit{fma}\ \mathit{fmb}\ v &&= \mathit{FMEither}\ (\mathit{fma}\ v \times_{\bullet} \mathit{fmb}\ v) \\ &\textbf{type}\ \mathit{FMap}\langle \texttt{:*:}\rangle\ \mathit{fma}\ \mathit{fmb}\ v &&= \mathit{FMProd}\ (\mathit{fma}\ (\mathit{fmb}\ v)) \\ &\textbf{type}\ \mathit{FMap}\langle \mathit{Con}\rangle\ \mathit{fma}\ v &&= \mathit{FMCon}\ (\mathit{fma}\ v) \\ &\textbf{type}\ \mathit{FMap}\langle \mathit{Label}\rangle\ \mathit{fma}\ v &&= \mathit{FMLabel}\ (\mathit{fma}\ v) \end{aligned}$$

Here, $(\times_{\bullet})$ is the type of optional pairs.

$$\textbf{data}\ a \times_{\bullet} b = \mathit{Null} \mid \mathit{Pair}\ a\ b$$

Instead of optional pairs we can also use ordinary pairs in the definition of *FMap*:

$$\textbf{type}\ \mathit{FMap}\langle \texttt{:+:}\rangle\ \mathit{fma}\ \mathit{fmb}\ v = \mathit{FMEither}\ (\mathit{fma}\ v \times \mathit{fmb}\ v).$$

This representation has, however, two major drawbacks: (i) it relies in an essential way on lazy evaluation and (ii) it is inefficient, see [21].

We assume there exists a suitable library implementing finite maps with integer keys. Such a library could be based, for instance, on a data structure known as a *Patricia tree* [49]. This data structure fits particularly well in the current setting since Patricia trees are a variety of tries. For clarity, we will use qualified names when referring to entities defined in the hypothetical module *Patricia*.

FMap is a type-indexed data type. We introduce type-indexed data types by example, for more background and theory, see [24]. Note that in each line of the definition of *FMap* we define a constructor name such as for example *FMUnit* in the *Unit* case, which can be used to construct elements of the type-indexed data type.

Furthermore, in contrast with type-indexed functions, the constructor index *Con* doesn't mention a constructor description anymore. This is because a type cannot depend on the value, so the constructor description can never be used in the definition of a type-indexed data type.

Type-indexed data types should also be defined on *Label*, which is used to represent a record in a data type. Since the definition of a type-indexed data type on *Label* is almost always exactly the same as the definition of the type-indexed data type on *Con*, we will almost always omit the definition of type-indexed data types (and functions) on *Label*.

Since the trie for the unit type is given by *Maybe* v rather than v itself, tries for isomorphic types are, in general, not isomorphic. We have, for instance, $\mathit{Unit} \cong \mathit{Unit} :\!*\!: \mathit{Unit}$ (ignoring laziness) but $\mathit{FMap}\langle \mathit{Unit}\rangle\ v = \mathit{Maybe}\ v \ncong \mathit{Maybe}\ (\mathit{Maybe}\ v) = \mathit{FMap}\langle \mathit{Unit} :\!*\!: \mathit{Unit}\rangle\ v$. The trie type *Maybe* (*Maybe* v) has two different representations of the empty trie: *Nothing* and *Just Nothing*. However, only the first one will be used in our implementation. Similarly, $\mathit{Maybe}\ v \times_\bullet \mathit{Maybe}\ v$ has two elements, *Null* and *Pair Nothing Nothing*, that represent the empty trie. Again, only the first one will be used.

As mentioned in Section 2.2, the type of *FMap* for types of kind $\star$ is $\star \to \star$. For type constructors with higher-order kinds, the type of *FMap* looks surprisingly similar to the type of type-indexed functions for higher-order kinds. A trie on the type *List* a is a trie for the type *List*, applied to a trie for the type a:

$$\mathit{FMap}\langle f :: \star \to \star\rangle :: (\star \to \star) \to (\star \to \star)$$

The 'type' of a type-indexed type is a *kind-indexed kind*. In general, we have:

$$\begin{aligned}
&\mathit{FMap}\langle f :: \kappa\rangle && :: \mathit{FMAP}\langle\kappa\rangle\ f\\
&\mathit{FMAP}\langle \kappa :: \Box\rangle && :: \Box\\
&\mathit{FMAP}\langle\star\rangle && = \star \to \star\\
&\mathit{FMAP}\langle\kappa \to \nu\rangle && = \mathit{FMAP}\langle\kappa\rangle \to \mathit{FMAP}\langle\nu\rangle
\end{aligned}$$

Example 1. Let us specialize *FMap* to the following data types.

$$\begin{aligned}
&\mathbf{data}\ \mathit{List}\ a && = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)\\
&\mathbf{data}\ \mathit{Tree}\ a\ b && = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a\ b)\ b\ (\mathit{Tree}\ a\ b)\\
&\mathbf{data}\ \mathit{Fork}\ a && = \mathit{ForkF}\ a\ a\\
&\mathbf{data}\ \mathit{Sequ}\ a && = \mathit{EndS} \mid \mathit{ZeroS}\ (\mathit{Sequ}\ (\mathit{Fork}\ a)) \mid \mathit{OneS}\ a\ (\mathit{Sequ}\ (\mathit{Fork}\ a))
\end{aligned}$$

Recall that these types are represented by

$$\begin{aligned}
&\mathit{List} && = \mathit{Fix}\ (\Lambda \mathit{List}\,.\,\Lambda a\,.\,\mathit{Unit} :\!+\!: a :\!*\!: \mathit{List}\ a)\\
&\mathit{Tree} && = \mathit{Fix}\ (\Lambda \mathit{Tree}\,.\,\Lambda a\ b\,.\,a :\!+\!: \mathit{Tree}\ a\ b :\!*\!: b :\!*\!: \mathit{Tree}\ a\ b)\\
&\mathit{Fork} && = \Lambda a\,.\,a :\!*\!: a\\
&\mathit{Sequ} && = \mathit{Fix}\ (\Lambda \mathit{Sequ}\,.\,\Lambda a\,.\,\mathit{Unit} :\!+\!: \mathit{Sequ}\ (\mathit{Fork}\ a) :\!+\!: a :\!*\!: \mathit{Sequ}\ (\mathit{Fork}\ a)).
\end{aligned}$$

Note that `:*:` binds stronger than `:+:`. Consequently, the corresponding trie types are

$$\begin{aligned}
FMapList &= Fix\ (\Lambda FMapList\,.\,\Lambda fa\,.\,Maybe \times_\bullet fa \cdot FMapList\ fa)\\
FMapTree &= Fix\ (\Lambda FMapTree\,.\,\Lambda fa\ fb\,.\\
&\qquad fa \times_\bullet\\
&\qquad FMapTree\ fa\ fb \cdot fb \cdot FMapTree\ fa\ fb)\\
FMapFork &= \Lambda fa\,.\,fa \cdot fa\\
FMapSequ &= Fix\ (\Lambda FMapSequ\,.\,\Lambda fa\,.\\
&\qquad Maybe \times_\bullet\\
&\qquad FMapSequ\ (FMapFork\ fa) \times_\bullet\\
&\qquad fa \cdot FMapSequ\ (FMapFork\ fa)).
\end{aligned}$$

As an aside, note that we interpret $a_1 \times_\bullet a_2 \times_\bullet a_3$ as the type of optional triples and not as nested optional pairs:

$$\mathbf{data}\ a_1 \times_\bullet a_2 \times_\bullet a_3 = Null \mid Triple\ a_1\ a_2\ a_3.$$

Now, since Haskell permits the definition of higher-order kinded data types, the second-order type constructors above can be directly coded as data types. All we have to do is to bring the equations into an applicative form.

$$\begin{aligned}
&\mathbf{data}\ FMapList\ fa\ v &&= NullList\\
& &&\mid\ TrieList\ (Maybe\ v)\\
& &&\qquad\qquad (fa\ (FMapList\ fa\ v))\\
&\mathbf{data}\ FMapTree\ fa\ fb\ v &&= NullTree\\
& &&\mid\ TrieTree\ (fa\ v)\\
& &&\qquad\qquad (FMapTree\ fa\ fb\\
& &&\qquad\qquad\quad (fb\ (FMapTree\ fa\ fb\ v)))
\end{aligned}$$

These types are the parametric variants of *FMapString* and *FMapDict* defined in Section 2.1: we have $FMapString \approx FMapList\ FMapChar$ (corresponding to $String \approx List\ Char$) and $FMapDict \approx FMapTree\ FMapString\ FMapString$ (corresponding to $Dict \approx Tree\ String\ String$). Things become interesting if we consider nested data types.

$$\begin{aligned}
&\mathbf{data}\ FMapFork\ fa\ v = TrieFork\ (fa\ (fa\ v))\\
&\mathbf{data}\ FMapSequ\ fa\ v = NullSequ\\
&\qquad\qquad\qquad\qquad \mid\ TrieSequ\ (Maybe\ v)\\
&\qquad\qquad\qquad\qquad\qquad (FMapSequ\ (FMapFork\ fa)\ v)\\
&\qquad\qquad\qquad\qquad\qquad (fa\ (FMapSequ\ (FMapFork\ fa)\ v))
\end{aligned}$$

The generalized trie of a nested data type is a second-order nested data type! A nest is termed second-order, if a parameter that is instantiated in a recursive call ranges over type constructors of first-order kind. The trie *FMapSequ* is a second-order nest since the parameter *fa* of kind $\star \to \star$ is changed in the recursive calls. By contrast, *FMapTree* is a first-order nest since its instantiated parameter v

has kind $\star$. It is quite easy to produce generalized tries that are both first- and second-order nests. If we swap the components of *Sequ*'s third constructor—*OneS a* (*Sequ* (*Fork a*)) becomes *OneS* (*Sequ* (*Fork a*)) *a*—then the third component of *FMapSequ* has type *FMapSequ* (*FMapFork fa*) (*fa v*) and since both *fa* and *v* are instantiated, *FMapSequ* is consequently both a first- and a second-order nest.

2.4 Empty tries

The empty trie is defined as follows.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Empty}\langle\!\langle\star\rangle\!\rangle\ t && = \forall v\,.\,\mathit{FMap}\langle t\rangle\ v\\
&\textbf{type}\ \mathit{Empty}\langle\!\langle\kappa\to\nu\rangle\!\rangle\ t && = \forall a\,.\,\mathit{Empty}\langle\!\langle\kappa\rangle\!\rangle\ a \to \mathit{Empty}\langle\!\langle\nu\rangle\!\rangle\ (t\ a)\\
&\mathit{empty}\langle t :: \kappa\rangle && ::\ \mathit{Empty}\langle\!\langle\kappa\rangle\!\rangle\ t\\
&\mathit{empty}\langle \mathit{Unit}\rangle && =\ \mathit{FMUnit}\ \mathit{Nothing}\\
&\mathit{empty}\langle \mathit{Char}\rangle && =\ \mathit{FMChar}\ [\,]\\
&\mathit{empty}\langle \mathit{Int}\rangle && =\ \mathit{FMInt}\ \mathit{Patricia.empty}\\
&\mathit{empty}\langle \texttt{:+:}\rangle\ \mathit{ea}\ \mathit{eb} && =\ \mathit{FMEither}\ \mathit{Null}\\
&\mathit{empty}\langle \texttt{:*:}\rangle\ \mathit{ea}\ \mathit{eb} && =\ \mathit{FMProd}\ \mathit{ea}\\
&\mathit{empty}\langle \mathit{Con}\ c\rangle\ \mathit{ea} && =\ \mathit{FMCon}\ \mathit{ea}
\end{aligned}
$$

The definition already illustrates several interesting aspects of programming with generalized tries. First, the explicit polymorphic type of *empty* is necessary to make the definition work. Consider the line $\mathit{empty}\langle \texttt{:*:}\rangle\ \mathit{ea}\ \mathit{eb}$, which is of type $\forall v\,.\,\mathit{FMap}\langle t_1\rangle\ (\mathit{FMap}\langle t_2\rangle\ v)$ for some t_1 and t_2. It is defined in terms of *ea*, which is of type $\forall v\,.\,\mathit{FMap}\langle t_1\rangle\ v$. That means that *ea* is used polymorphically. In other words, *empty* makes use of polymorphic recursion!

Example 2. Let us specialize *empty* to lists and binary random-access lists.

$$
\begin{aligned}
&\mathit{emptyList} && ::\ \forall \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w) \to (\forall v\,.\,\mathit{FMapList}\ \mathit{fa}\ v)\\
&\mathit{emptyList}\ \mathit{ea} && =\ \mathit{NullList}\\
&\mathit{emptyFork} && ::\ \forall \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w) \to (\forall v\,.\,\mathit{FMapFork}\ \mathit{fa}\ v)\\
&\mathit{emptyFork}\ \mathit{ea} && =\ \mathit{TrieFork}\ \mathit{ea}\\
&\mathit{emptySequ} && ::\ \forall \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w) \to (\forall v\,.\,\mathit{FMapSequ}\ \mathit{fa}\ v)\\
&\mathit{emptySequ}\ \mathit{ea} && =\ \mathit{NullSequ}
\end{aligned}
$$

The second function, *emptyFork*, illustrates the polymorphic use of the parameter: *ea* has type $\forall w\,.\,\mathit{fa}\ w$ but is used as an element of $\mathit{fa}\ (\mathit{fa}\ w)$. The functions *emptyList* and *emptySequ* show that the 'mechanically' generated definitions can sometimes be slightly improved: the argument *ea* is not needed.

Function $\mathit{isempty}\langle t\rangle$ takes a trie and determines whether or not it is empty.

$$
\begin{aligned}
&\textbf{type}\ \mathit{IsEmpty}\langle\!\langle\star\rangle\!\rangle\ t && = \forall v\,.\,\mathit{FMap}\langle t\rangle\ v \to \mathit{Bool}\\
&\textbf{type}\ \mathit{IsEmpty}\langle\!\langle\kappa\to\nu\rangle\!\rangle\ t && = \forall a\,.\,\mathit{IsEmpty}\langle\!\langle\kappa\rangle\!\rangle\ a \to \mathit{IsEmpty}\langle\!\langle\nu\rangle\!\rangle\ (t\ a)
\end{aligned}
$$

$$
\begin{array}{ll}
\mathit{isempty}\langle t :: \kappa\rangle & :: \mathit{IsEmpty}\langle\!\langle \kappa \rangle\!\rangle\ t \\
\mathit{isempty}\langle \mathit{Unit}\rangle\ (\mathit{FMUnit}\ v) & = \mathit{isNothing}\ v \\
\mathit{isempty}\langle \mathit{Char}\rangle\ (\mathit{FMChar}\ l) & = \mathit{null}\ l \\
\mathit{isempty}\langle \mathit{Int}\rangle\ (\mathit{FMInt}\ l) & = \mathit{Patricia.isempty}\ l \\
\mathit{isempty}\langle \texttt{:+:}\rangle\ \mathit{iea}\ \mathit{ieb}\ (\mathit{FMEither}\ \mathit{Null}) & = \mathit{True} \\
\mathit{isempty}\langle \texttt{:+:}\rangle\ \mathit{iea}\ \mathit{ieb}\ (\mathit{FMEither}\ d) & = \mathit{False} \\
\mathit{isempty}\langle \texttt{:*:}\rangle\ \mathit{iea}\ \mathit{ieb}\ (\mathit{FMProd}\ d) & = \mathit{iea}\ d \\
\mathit{isempty}\langle \mathit{Con}\ c\rangle\ \mathit{iea}\ (\mathit{FMCon}\ d) & = \mathit{iea}\ d
\end{array}
$$

Example 3. Let us specialize *isempty* to lists and binary random-access lists.

$$
\begin{array}{ll}
\mathit{isemptyList} & :: \forall \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w \to \mathit{Bool}) \to \\
 & \qquad (\forall v\,.\,\mathit{FMapList}\ \mathit{fa}\ v \to \mathit{Bool}) \\
\mathit{isemptyList}\ \mathit{iea}\ \mathit{NullList} & = \mathit{True} \\
\mathit{isemptyList}\ \mathit{iea}\ (\mathit{TrieList}\ \mathit{tn}\ \mathit{tc}) & = \mathit{False} \\
\mathit{isemptyFork} & :: \forall \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w \to \mathit{Bool}) \to \\
 & \qquad (\forall v\,.\,\mathit{FMapFork}\ \mathit{fa}\ v \to \mathit{Bool}) \\
\mathit{isemptyFork}\ \mathit{iea}\ (\mathit{TrieFork}\ \mathit{tf}) & = \mathit{iea}\ \mathit{tf} \\
\mathit{isemptySequ} & :: \forall \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w \to \mathit{Bool}) \to \\
 & \qquad (\forall v\,.\,\mathit{FMapSequ}\ \mathit{fa}\ v \to \mathit{Bool}) \\
\mathit{isemptySequ}\ \mathit{iea}\ \mathit{NullSequ} & = \mathit{True} \\
\mathit{isemptySequ}\ \mathit{iea}\ (\mathit{TrieSequ}\ \mathit{tv}\ \mathit{tf}\ \mathit{ts}) & = \mathit{False}
\end{array}
$$

2.5 Singleton tries

Function $\mathit{single}\langle t\rangle\ (t, v)$ constructs a trie that contains the binding (t, v) as its only element. To construct a trie in the sum case, we have to return a *Pair*, of which only one component is inhabited. The other component is the empty trie. This implies that *single depends on empty*, which in Generic Haskell is denoted by

$$\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty}$$

This line says that the generic function *single* depends on both *empty* and itself. The right-hand side of the arrow $\leftarrow$ enumerates the functions on which the left-hand side argument depends. The effect of this dependency is that we can use both the empty trie and the single trie of the children in the sum, product, and constructor cases of function *single*. On higher-order kinds, the dependency on function *empty* is reflected in the type of function *single*.

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Single}\langle\!\langle \star \rangle\!\rangle\ t & = \forall v\,.\,(t, v) \to \mathit{FMap}\langle t\rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \to \nu \rangle\!\rangle\ t & = \forall a\,.\,\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ a \to \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ a \to \mathit{Single}\langle\!\langle \nu \rangle\!\rangle\ (t\ a)
\end{array}
$$

Plain generic functions can be seen as catamorphisms [42, 46] over the structure of data types. With dependencies, we also get the power of paramorphisms [44] (or, more general, even zygomorphisms [41]).

$$
\begin{array}{ll}
\mathit{single}\langle t :: \kappa\rangle & :: \mathit{Single}\langle\!\langle\kappa\rangle\!\rangle\ t \\
\mathit{single}\langle \mathit{Unit}\rangle\ (\mathit{Unit}, v) & = \mathit{FMUnit}\ (\mathit{Just}\ v) \\
\mathit{single}\langle \mathit{Char}\rangle\ (c, v) & = \mathit{FMChar}\ [(c, v)] \\
\mathit{single}\langle \mathit{Int}\rangle\ (i, v) & = \mathit{FMInt}\ (\mathit{Patricia.single}\ (i, v)) \\
\mathit{single}\langle \texttt{:+:}\rangle\ \mathit{sa}\ \mathit{ea}\ \mathit{sb}\ \mathit{eb}\ (\mathit{Inl}\ a, v) & = \mathit{FMEither}\ (\mathit{Pair}\ (\mathit{sa}\ (a, v))\ \mathit{eb}) \\
\mathit{single}\langle \texttt{:+:}\rangle\ \mathit{sa}\ \mathit{ea}\ \mathit{sb}\ \mathit{eb}\ (\mathit{Inr}\ b, v) & = \mathit{FMEither}\ (\mathit{Pair}\ \mathit{ea}\ (\mathit{sb}\ (b, v))) \\
\mathit{single}\langle \texttt{:*:}\rangle\ \mathit{sa}\ \mathit{ea}\ \mathit{sb}\ \mathit{eb}\ (a\ \texttt{:*:}\ b, v) & = \mathit{FMProd}\ (\mathit{sa}\ (a, \mathit{sb}\ (b, v))) \\
\mathit{single}\langle \mathit{Con}\ d\rangle\ \mathit{sa}\ \mathit{ea}\ (\mathit{Con}\ b, v) & = \mathit{FMCon}\ (\mathit{sa}\ (b, v))
\end{array}
$$

Example 4. Let us again specialize the generic function to lists and binary random-access lists.

$$
\begin{array}{ll}
\mathit{singleList} & :: \forall kT\ \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w) \to (\forall w\,.\,kT \times w \to \mathit{fa}\ w) \\
& \quad \to (\forall v\,.\,(\mathit{List}\ kT \times v \to \mathit{FMapList}\ \mathit{fa}\ v)) \\
\mathit{singleList}\ \mathit{ea}\ \mathit{sa}\ (\mathit{Nil}, v) & = \mathit{TrieList}\ (\mathit{Just}\ v)\ \mathit{ea} \\
\mathit{singleList}\ \mathit{ea}\ \mathit{sa}\ (\mathit{Cons}\ k\ \mathit{ks}, v) & = \mathit{TrieList}\ \mathit{Nothing}\ (\mathit{sa}\ (k, \mathit{singleList}\ \mathit{ea}\ \mathit{sa}\ (\mathit{ks}, v))) \\
\mathit{singleFork} & :: \forall kT\ \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w) \to (\forall w\,.\,kT \times w \to \mathit{fa}\ w) \\
& \quad \to (\forall v\,.\,(\mathit{Fork}\ kT \times v \to \mathit{FMapFork}\ \mathit{fa}\ v)) \\
\mathit{singleFork}\ \mathit{ea}\ \mathit{sa}\ (\mathit{ForkF}\ k_1\ k_2, v) & = \mathit{TrieFork}\ (\mathit{sa}\ (k_1, \mathit{sa}\ (k_2, v))) \\
\mathit{singleSequ} & :: \forall kT\ \mathit{fa}\,.\,(\forall w\,.\,\mathit{fa}\ w) \to (\forall w\,.\,kT \times w \to \mathit{fa}\ w) \\
& \quad \to (\forall v\,.\,(\mathit{Sequ}\ kT \times v \to \mathit{FMapSequ}\ \mathit{fa}\ v)) \\
\mathit{singleSequ}\ \mathit{ea}\ \mathit{sa}\ (\mathit{EndS}, v) & = \mathit{TrieSequ}\ (\mathit{Just}\ v)\ \mathit{NullSequ}\ \mathit{ea} \\
\mathit{singleSequ}\ \mathit{ea}\ \mathit{sa}\ (\mathit{ZeroS}\ s, v) & \\
\quad = \ \mathit{TrieSequ}\ \mathit{Nothing}\ (\mathit{singleSequ}\ (\mathit{emptyFork}\ \mathit{ea})\ (\mathit{singleFork}\ \mathit{ea}\ \mathit{sa})\ (s, v))\ \mathit{ea} & \\
\mathit{singleSequ}\ \mathit{ea}\ \mathit{sa}\ (\mathit{OneS}\ k\ s, v) & \\
\quad = \ \mathit{TrieSequ}\ \mathit{Nothing}\ \mathit{NullSequ}\ (\mathit{sa}\ (k, \mathit{singleSequ}\ (\mathit{emptyFork}\ \mathit{ea})\ (\mathit{singleFork}\ \mathit{ea}\ \mathit{sa})\ (s, v))) &
\end{array}
$$

Again, we can simplify the 'mechanically' generated definitions: since the definition of *Fork* does not involve sums, *singleFork* does not require its first argument, *ea*, which can be safely removed.

2.6 Look up

The look-up function implements the scheme discussed in Section 2.1.

$$
\begin{array}{l}
\textbf{type}\ \mathit{Lookup}\langle\!\langle\star\rangle\!\rangle\ t \qquad = \forall v\,.\,t \to \mathit{FMap}\langle t\rangle\ v \to \mathit{Maybe}\ v \\
\textbf{type}\ \mathit{Lookup}\langle\!\langle\kappa \to \nu\rangle\!\rangle\ t = \forall a\,.\,\mathit{Lookup}\langle\!\langle\kappa\rangle\!\rangle\ a \to \mathit{Lookup}\langle\!\langle\nu\rangle\!\rangle\ (t\ a)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{lookup}\langle t :: \kappa\rangle & :: \mathit{Lookup}\langle\!\langle\kappa\rangle\!\rangle\ t \\
\mathit{lookup}\langle \mathit{Unit}\rangle\ \mathit{Unit}\ (\mathit{FMUnit}\ \mathit{fm}) & = \mathit{fm} \\
\mathit{lookup}\langle \mathit{Char}\rangle\ c\ (\mathit{FMChar}\ \mathit{fm}) & = \mathit{lookupChar}\ c\ \mathit{fm} \\
\mathit{lookup}\langle \mathit{Int}\rangle\ i\ (\mathit{FMInt}\ \mathit{fm}) & = \mathit{Patricia.lookup}\ i\ \mathit{fm} \\
\mathit{lookup}\langle \texttt{:+:}\rangle\ \mathit{la}\ \mathit{lb}\ (\mathit{Inl}\ a)\ (\mathit{FMEither}\ (\mathit{Pair}\ \mathit{fma}\ \mathit{fmb})) & = \mathit{la}\ \mathit{fma}\ a \\
\mathit{lookup}\langle \texttt{:+:}\rangle\ \mathit{la}\ \mathit{lb}\ (\mathit{Inr}\ b)\ (\mathit{FMEither}\ (\mathit{Pair}\ \mathit{fma}\ \mathit{fmb})) & = \mathit{lb}\ \mathit{fmb}\ b \\
\mathit{lookup}\langle \texttt{:*:}\rangle\ \mathit{la}\ \mathit{lb}\ (a\ \texttt{:*:}\ b)\ (\mathit{FMProd}\ \mathit{fma}) & = \textbf{do}\ \mathit{fmb} \leftarrow \mathit{la}\ \mathit{fma}\ a \\
& \qquad\ \ \mathit{lb}\ \mathit{fmb}\ b \\
\mathit{lookup}\langle \mathit{Con}\ d\rangle\ l\ (\mathit{Con}\ b)\ (\mathit{FMCon}\ \mathit{fm}) & = l\ \mathit{fm}\ b
\end{array}
$$

On sums the look-up function selects the appropriate map; on products it 'composes' the look-up functions for the components. Since *lookup* has result type *Maybe v*, we use the monadic composition.

Suppose we want to define a function *specialLookup* that is almost the same as function *lookup*, but uses a different function, say *lookupCharEfficient*, when looking up characters. Then we can use a default case [8] to define function *specialLookup* using function *lookup* as follows:

$$\begin{aligned}
&\mathit{specialLookup}\langle t :: \kappa\rangle && :: \mathit{Lookup}\langle\!\langle\kappa\rangle\!\rangle\ t\\
&\mathit{specialLookup}\langle \mathit{Char}\rangle\ c\ (\mathit{FMChar}\ \mathit{fm}) && = \mathit{lookupCharEfficient}\ c\ \mathit{fm}\\
&\mathit{specialLookup}\langle a\rangle && = \mathit{lookup}\langle a\rangle
\end{aligned}$$

So function *specialLookup* is equal to the function *lookup* in all cases except for the *Char* case, where it uses a special lookup function.

We also might want to use the special lookup function for just one kind of characters that appear in a particular data type. For example, suppose we have the following data type:

$$\mathbf{data}\ C = \mathit{C1}\ \mathit{Char} \mid \mathit{C2}\ \mathit{Char} \mid \ldots$$

and we want to use the efficient lookup function *lookupCharEfficient* only for characters under the constructor *C1*. Then we can use a constructor case [8] to define function *specialLookup*.

$$\begin{aligned}
&\mathit{specialLookup}\langle t :: \kappa\rangle && :: \mathit{Lookup}\langle\!\langle\kappa\rangle\!\rangle\ t\\
&\mathit{specialLookup}\langle \mathbf{case}\ \mathit{C1}\rangle\ (\mathit{C1}\ c)\ (\mathit{FMChar}\ \mathit{fm}) && = \mathit{lookupCharEfficient}\ c\ \mathit{fm}\\
&\mathit{specialLookup}\langle a\rangle && = \mathit{lookup}\langle a\rangle
\end{aligned}$$

Function *specialLookup* is still a generic function, but on values of the form *C1 c* of type *C* it uses a different lookup function.

Example 5. Specializing $\mathit{lookup}\langle K\rangle$ to concrete instances of K is by now probably a matter of routine. We obtain

$$\begin{aligned}
&\mathit{lookupList} :: \forall kT\ \mathit{fa}\,.\,(\forall w\,.\,kT \to \mathit{fa}\ w \to \mathit{Maybe}\ w)\\
&\qquad\qquad\qquad \to (\forall v\,.\,\mathit{List}\ kT \to \mathit{FMapList}\ \mathit{fa}\ v \to \mathit{Maybe}\ v)\\
&\mathit{lookupList}\ \mathit{la}\ \mathit{ks}\ \mathit{NullList} && = \mathit{Nothing}\\
&\mathit{lookupList}\ \mathit{la}\ \mathit{Nil}\ (\mathit{TrieList}\ \mathit{tn}\ \mathit{tc}) && = \mathit{tn}\\
&\mathit{lookupList}\ \mathit{la}\ (\mathit{Cons}\ k\ \mathit{ks})\ (\mathit{TrieList}\ \mathit{tn}\ \mathit{tc}) && = (\mathit{la}\ k \diamond \mathit{lookupList}\ \mathit{la}\ \mathit{ks})\ \mathit{tc}\\
&\mathit{lookupFork} :: \forall kT\ \mathit{fa}\,.\,(\forall w\,.\,kT \to \mathit{fa}\ w \to \mathit{Maybe}\ w)\\
&\qquad\qquad\qquad \to (\forall v\,.\,\mathit{Fork}\ kT \to \mathit{FMapFork}\ \mathit{fa}\ v \to \mathit{Maybe}\ v)\\
&\mathit{lookupFork}\ \mathit{la}\ (\mathit{ForkF}\ k_1\ k_2)\ (\mathit{TrieFork}\ \mathit{tf}) && = (\mathit{la}\ k_1 \diamond \mathit{la}\ k_2)\ \mathit{tf}\\
&\mathit{lookupSequ} :: \forall \mathit{fa}\ kT\,.\,(\forall w\,.\,kT \to \mathit{fa}\ w \to \mathit{Maybe}\ w)\\
&\qquad\qquad\qquad \to (\forall v\,.\,\mathit{Sequ}\ kT \to \mathit{FMapSequ}\ \mathit{fa}\ v \to \mathit{Maybe}\ v)\\
&\mathit{lookupSequ}\ \mathit{la}\ s\ \mathit{NullSequ} && = \mathit{Nothing}\\
&\mathit{lookupSequ}\ \mathit{la}\ \mathit{EndS}\ (\mathit{TrieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) && = \mathit{te}\\
&\mathit{lookupSequ}\ \mathit{la}\ (\mathit{ZeroS}\ s)\ (\mathit{TrieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) && = \mathit{lookupSequ}\ (\mathit{lookupFork}\ \mathit{la})\ s\ \mathit{tz}\\
&\mathit{lookupSequ}\ \mathit{la}\ (\mathit{OneS}\ a\ s)\ (\mathit{TrieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) && = (\mathit{la}\ a \diamond \mathit{lookupSequ}\ (\mathit{lookupFork}\ \mathit{la})\ s)\ \mathit{to}
\end{aligned}$$

The function *lookupList* generalizes *lookupString* defined in Section 2.1; we have $lookupString \approx lookupList\ lookupChar$.

2.7 Inserting and merging

Insertion is defined in terms of *merge* and *single*.

$$\begin{array}{ll} insert\langle t :: \star\rangle & :: (v \to v \to v) \to (t, v) \to FMap\langle t\rangle\ v \to FMap\langle t\rangle\ v \\ insert\langle t\rangle\ c\ (x, v)\ d & = merge\langle t\rangle\ c\ (single\langle t\rangle\ (x, v))\ d \end{array}$$

Function *insert* is defined as a *generic abstraction*. A generic abstraction lifts the restrictions that are normally imposed on the type of a generic function. For example, normally a type constructor of kind $\star \to \star$ is always translated to a function by the translation scheme of generic functions. When you use a generic abstraction, this can be circumvented. The abstracted type parameter is, however, restricted to types of a fixed kind. In the above case, *insert* only works for types of kind $\star$. In the exercise at the end of this section you will define *insert* as a type-indexed function that works for type constructors of all kinds.

Merging two tries is surprisingly simple. Given an auxiliary function for combining two values of type *Maybe*:

$$\begin{array}{ll} combine & :: \forall v\,.\,(v \to v \to v) \to \\ & \qquad (Maybe\ v \to Maybe\ v \to Maybe\ v) \\ combine\ c\ Nothing\ Nothing & = Nothing \\ combine\ c\ Nothing\ (Just\ v_2) & = Just\ v_2 \\ combine\ c\ (Just\ v_1)\ Nothing & = Just\ v_1 \\ combine\ c\ (Just\ v_1)\ (Just\ v_2) & = Just\ (c\ v_1\ v_2) \end{array}$$

and a function for merging two association lists

$$\begin{array}{ll} mergeChar & :: \forall v\,.\,(v \to v \to v) \to \\ & \qquad (FMapChar\ v \to FMapChar\ v \to FMapChar\ v) \\ mergeChar\ c\ [\,]\ x' = x' & \\ mergeChar\ c\ x\ [\,]\ = x & \\ mergeChar\ c\ ((k, v) : x)\ ((k', v') : x') & \\ \quad \mid k < k' & = (k, v) : mergeChar\ c\ x\ ((k', v') : x') \\ \quad \mid k == k' & = (k, c\ v\ v') : mergeChar\ c\ x\ x' \\ \quad \mid k > k' & = (k', v') : mergeChar\ c\ ((k, v) : x)\ x', \end{array}$$

we can define *merge* as follows.

$$\begin{array}{l} \textbf{type}\ Merge\langle\!\langle \star \rangle\!\rangle\ t \qquad = \forall v\,. \\ \quad (v \to v \to v) \to FMap\langle t\rangle\ v \to FMap\langle t\rangle\ v \to FMap\langle t\rangle\ v \\ \textbf{type}\ Merge\langle\!\langle \kappa \to \nu \rangle\!\rangle\ t = \forall a\,.\,Merge\langle\!\langle \kappa \rangle\!\rangle\ a \to Merge\langle\!\langle \nu \rangle\!\rangle\ (t\ a) \end{array}$$

$$\begin{array}{ll} merge\langle t :: k\rangle & :: Merge\langle\!\langle k \rangle\!\rangle\ t \\ merge\langle Unit\rangle\ c\ (FMUnit\ v)\ (FMUnit\ v') & = FMUnit\ (combine\ c\ v\ v') \\ merge\langle Char\rangle\ c\ (FMChar\ fm)\ (FMChar\ fm') & = FMChar\ (mergeChar\ fm'\ fm) \\ merge\langle Int\rangle\ c\ (FMInt\ fm)\ (FMInt\ fm') & = FMInt\ (Patricia.mergeInt\ fm'\ fm) \\ merge\langle Con\ d\rangle\ ma\ c\ (FMCon\ e)\ (FMCon\ e') & = FMCon\ (ma\ c\ e\ e') \end{array}$$

For the sum case, we have to distinguish all possible forms of the tries to be merged:

$$\begin{aligned}
&\mathit{merge}\langle\texttt{:+:}\rangle\ \mathit{ma}\ \mathit{mb}\ c\ d\ (\mathit{FMEither}\ \mathit{Null}) && = d\\
&\mathit{merge}\langle\texttt{:+:}\rangle\ \mathit{ma}\ \mathit{mb}\ c\ (\mathit{FMEither}\ \mathit{Null})\ d && = d\\
&\mathit{merge}\langle\texttt{:+:}\rangle\ \mathit{ma}\ \mathit{mb}\ c\ (\mathit{FMEither}\ (\mathit{Pair}\ x\ y))\ (\mathit{FMEither}\ (\mathit{Pair}\ v\ w)) = \\
&\quad \mathit{FMEither}\ (\mathit{Pair}\ (\mathit{ma}\ c\ x\ v)\ (\mathit{mb}\ c\ y\ w))
\end{aligned}$$

The most interesting equation is the product case. The tries d and d' are of type $\mathit{FMap}\langle t_1\rangle\ (\mathit{FMap}\langle t_2\rangle\ v)$, for some types t_1 and t_2. To merge them we can recursively call *ma*; we must, however, supply a combining function of type $\forall v\,.\,\mathit{FMap}\langle t_2\rangle\ v \to \mathit{FMap}\langle t_2\rangle\ v \to \mathit{FMap}\langle t_2\rangle\ v$. A moment's reflection reveals that *mb c* is the desired combining function.

$$\mathit{merge}\langle\texttt{:*:}\rangle\ \mathit{ma}\ \mathit{mb}\ c\ (\mathit{FMProd}\ d)\ (\mathit{FMProd}\ d') = \mathit{FMProd}\ (\mathit{ma}\ (\mathit{mb}\ c)\ d\ d')$$

The definition of *merge* shows that it is sometimes necessary to implement operations more general than immediately needed. If $\mathit{Merge}\langle\!\langle\star\rangle\!\rangle\ t$ had been the simpler type $\forall v\,.\,\mathit{FMap}\langle t\rangle\ v \to \mathit{FMap}\langle t\rangle\ v \to \mathit{FMap}\langle t\rangle\ v$, then we would not have been able to give a defining equation for `:*:`.

Example 6. To complete the picture let us again specialize the merging operation for lists and binary random-access lists. The different instances of *merge* are surprisingly concise (only the types look complicated).

$$\begin{aligned}
&\mathit{mergeList} :: \forall \mathit{fa}\,.\,(\forall w\,.\,(w \to w \to w) \to (\mathit{fa}\ w \to \mathit{fa}\ w \to \mathit{fa}\ w))\\
&\qquad\qquad \to (\forall v\,.\,(v \to v \to v)\\
&\qquad\qquad \to (\mathit{FMapList}\ \mathit{fa}\ v \to \mathit{FMapList}\ \mathit{fa}\ v \to \mathit{FMapList}\ \mathit{fa}\ v))\\
&\mathit{mergeList}\ \mathit{ma}\ c\ \mathit{NullList}\ t\ = t\\
&\mathit{mergeList}\ \mathit{ma}\ c\ t\ \mathit{NullList}\ = t\\
&\mathit{mergeList}\ \mathit{ma}\ c\ (\mathit{TrieList}\ \mathit{tn}\ \mathit{tc})\ (\mathit{TrieList}\ \mathit{tn}'\ \mathit{tc}')\\
&\qquad\qquad = \mathit{TrieList}\ (\mathit{combine}\ c\ \mathit{tn}\ \mathit{tn}')\\
&\qquad\qquad\qquad (\mathit{ma}\ (\mathit{mergeList}\ \mathit{ma}\ c)\ \mathit{tc}\ \mathit{tc}')\\
&\mathit{mergeFork} :: \forall \mathit{fa}\,.\,(\forall w\,.\,(w \to w \to w) \to (\mathit{fa}\ w \to \mathit{fa}\ w \to \mathit{fa}\ w))\\
&\qquad\qquad \to (\forall v\,.\,(v \to v \to v)\\
&\qquad\qquad \to (\mathit{FMapFork}\ \mathit{fa}\ v \to \mathit{FMapFork}\ \mathit{fa}\ v \to \mathit{FMapFork}\ \mathit{fa}\ v))\\
&\mathit{mergeFork}\ \mathit{ma}\ c\ (\mathit{TrieFork}\ \mathit{tf})\ (\mathit{TrieFork}\ \mathit{tf}')\\
&\qquad\qquad = \mathit{TrieFork}\ (\mathit{ma}\ (\mathit{ma}\ c)\ \mathit{tf}\ \mathit{tf}')\\
&\mathit{mergeSequ} :: \forall \mathit{fa}\,.\,(\forall w\,.\,(w \to w \to w) \to (\mathit{fa}\ w \to \mathit{fa}\ w \to \mathit{fa}\ w))\\
&\qquad\qquad \to (\forall v\,.\,(v \to v \to v)\\
&\qquad\qquad \to (\mathit{FMapSequ}\ \mathit{fa}\ v \to \mathit{FMapSequ}\ \mathit{fa}\ v \to \mathit{FMapSequ}\ \mathit{fa}\ v))\\
&\mathit{mergeSequ}\ \mathit{ma}\ c\ \mathit{NullSequ}\ t = t\\
&\mathit{mergeSequ}\ \mathit{ma}\ c\ t\ \mathit{NullSequ} = t\\
&\mathit{mergeSequ}\ \mathit{ma}\ c\ (\mathit{TrieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to})\ (\mathit{TrieSequ}\ \mathit{te}'\ \mathit{tz}'\ \mathit{to}')\\
&\qquad\qquad = \mathit{TrieSequ}\ (\mathit{combine}\ c\ \mathit{te}\ \mathit{te}')\\
&\qquad\qquad\qquad (\mathit{mergeSequ}\ (\mathit{mergeFork}\ \mathit{ma})\ c\ \mathit{tz}\ \mathit{tz}')\\
&\qquad\qquad\qquad (\mathit{ma}\ (\mathit{mergeSequ}\ (\mathit{mergeFork}\ \mathit{ma})\ c)\ \mathit{to}\ \mathit{to}') \qquad \square
\end{aligned}$$

2.8 Deleting

Function $delete\langle t\rangle$ takes a key and a trie, and removes the binding for the key from the trie. For the *Char* case we need a help function that removes an element from an association list:

$$deleteChar :: \forall v\,.\, Char \to FMapChar\ v \to FMapChar\ v$$

and similarly for the *Int* case. Function *delete* is defined as follows:

$$\begin{aligned}
&delete\langle t :: \kappa\rangle && :: Delete\langle\!\langle\kappa\rangle\!\rangle\ t\\
&delete\langle Unit\rangle\ Unit\ (FMUnit\ v) && = FMUnit\ Nothing\\
&delete\langle Char\rangle\ c\ (FMChar\ fm) && = FMChar\ (deleteChar\ c\ fm)\\
&delete\langle Int\rangle\ i\ (FMInt\ fm) && = FMInt\ (Patricia.delete\ i\ fm)
\end{aligned}$$

All *delete* cases except the product case are relatively straightforward. In the product case, we have to remove a binding for a product a `:*:` b. We do this by using a to lookup the trie d in which there is a binding for b. Then we remove the binding for b in d, obtaining a trie d'. If d' is empty, then we delete the complete binding for a in d, otherwise we insert the binding (a, d') in the original trie d. Here we assume insertion overwrites existing bindings in a trie. Function *delete* depends on functions *lookup*, *insert*, and *isempty*:

$$\textbf{dependency}\ delete \leftarrow delete\ lookup\ insert\ isempty$$

Here we need the kind-indexed typed version of function *insert*, as defined in the exercise at the end of this section.

$$\begin{aligned}
&\textbf{type}\ Delete\langle\!\langle\star\rangle\!\rangle\ t && = \forall v\,.\, t \to FMap\langle t\rangle\ v \to FMap\langle t\rangle\ v\\
&\textbf{type}\ Delete\langle\!\langle\kappa \to \nu\rangle\!\rangle\ t && = \forall a\,.\, Delete\langle\!\langle\kappa\rangle\!\rangle\ a\\
& && \quad \to Lookup\langle\!\langle\kappa\rangle\!\rangle\ a\\
& && \quad \to Insert\langle\!\langle\kappa\rangle\!\rangle\ a\\
& && \quad \to IsEmpty\langle\!\langle\kappa\rangle\!\rangle\ a\\
& && \quad \to Delete\langle\!\langle\nu\rangle\!\rangle\ (t\ a)
\end{aligned}$$

$$\begin{aligned}
&delete\langle \texttt{:+:}\rangle\ da\ la\ ia\ iea\ db\ lb\ ib\ ieb\ (Inl\ a)\ (FMEither\ (Pair\ x\ y)) =\\
&\quad FMEither\ (Pair\ (da\ a\ x)\ y)\\
&delete\langle \texttt{:+:}\rangle\ da\ la\ ia\ iea\ db\ lb\ ib\ ieb\ (Inr\ b)\ (FMEither\ (Pair\ x\ y)) =\\
&\quad FMEither\ (Pair\ x\ (db\ b\ y))\\
&delete\langle \texttt{:*:}\rangle\ da\ la\ ia\ iea\ db\ lb\ ib\ ieb\ (a\ \texttt{:*:}\ b)\ (FMProd\ d) =\\
&\quad \textbf{let}\ Just\ d' = la\ a\ d\\
&\qquad\qquad d'' = db\ b\ d'\\
&\quad \textbf{in if}\ ieb\ d''\ \textbf{then}\ FMProd\ (da\ a\ d)\ \textbf{else}\ FMProd\ (ia\ (a, d'')\ d)\\
&delete\langle Con\ c\rangle\ da\ la\ ia\ iea\ (Con\ b)\ (FMCon\ d) =\\
&\quad FMCon\ (da\ b\ d)
\end{aligned}$$

2.9 Properties

The functions on tries enjoy several properties which hold generically for all instances of t and which can be proved by fixed point induction.

$$\begin{aligned} lookup\langle t\rangle\ k\ (empty\langle t\rangle) &= Nothing \\ lookup\langle t\rangle\ k\ (single\langle t\rangle\ (k_1, v_1)) &= \textbf{if}\ k == k_1\ \textbf{then}\ Just\ v_1\ \textbf{else}\ Nothing \\ lookup\langle t\rangle\ k\ (merge\langle t\rangle\ c\ t_1\ t_2) &= combine\ c\ (lookup\langle t\rangle\ k\ t_1)\ (lookup\langle t\rangle\ k\ t_2) \end{aligned}$$

The last law, for instance, states that looking up a key in the merge of two tries yields the same result as looking up the key in each trie separately and then combining the results. If the combining form c is associative,

$$c\ v_1\ (c\ v_2\ v_3) = c\ (c\ v_1\ v_2)\ v_3,$$

then $merge\langle t\rangle\ c$ is associative, as well. Furthermore, $empty\langle t\rangle$ is the left and the right unit of $merge\langle t\rangle\ c$:

$$\begin{aligned} merge\langle t\rangle\ c\ (empty\langle t\rangle)\ x &= x \\ merge\langle t\rangle\ c\ x\ (empty\langle t\rangle) &= x \\ merge\langle t\rangle\ c\ x_1\ (merge\langle t\rangle\ c\ x_2\ x_3) &= merge\langle t\rangle\ c\ (merge\langle t\rangle\ c\ x_1\ x_2)\ x_3. \end{aligned}$$

2.10 Related work

Knuth [39] attributes the idea of a trie to Thue who introduced it in a paper about strings that do not contain adjacent repeated substrings [54]. De la Briandais [13] recommended tries for computer searching. The generalization of tries from strings to elements built according to an arbitrary signature was discovered by Wadsworth [57] and others independently since. Connelly et al. [10] formalized the concept of a trie in a categorical setting: they showed that a trie is a functor and that the corresponding look-up function is a natural transformation.

The first implementation of generalized tries was given by Okasaki in his recent textbook on functional data structures [48]. Tries for parameterized types like lists or binary trees are represented as Standard ML functors. While this approach works for regular data types, it fails for nested data types such as *Sequ*. In the latter case data types of second-order kind are indispensable. The material in this section has been taken from Hinze [22].

Exercise 1. (Simple exercise to experiment with Generic Haskell.) Define function *depth*, which returns the depth of a value. The depth of a value is the maximum number of constructors encountered on a path from the root to a leaf. For example, given the data type *Tree*:

$$\textbf{data}\ Tree\ a = Leaf\ a \mid Node\ (Tree\ a)\ a\ (Tree\ a)$$

the depth of

$$exTree = Node\ (Leaf\ 1)\ 2\ (Node\ (Node\ (Leaf\ 6)\ 0\ (Leaf\ (-11)))\ 4\ (Leaf\ 4))$$

is 4.

Exercise 2. (More difficult exercise about dictionaries.) Define function *insert* as a type-indexed function with a kind-indexed kind. You can download the code for the functions described in this section from `http://www.generic-haskell.org`, including the solution to this exercise. You might want to avoid looking at the implementation of *insert* while solving this exercise.

3 Generic Programming for XML tools

An XML document is usually structured according to a *document type definition* (DTD). DTDs are another formalism for specifying data types (or grammars, abstract syntax). This section shows how an XML compressor is implemented as a generic program, and it discusses which other classes of XML tools would profit from an implementation as a generic program. The example shows how generic programming can be used to implement XML tools such as XML editors, databases, and compressors, that depend on the DTD of an input XML document. The resulting tools usually perform better because knowledge of the DTD can be used to optimise the tools, and are smaller, because all DTD handling is dealt with in the generic programming compiler.

The Generic Haskell code for this section can be downloaded from the applications page on `http://www.generic-haskell.org/`. It is also distributed together with the Generic Haskell compiler.

3.1 Introduction

XML Tools. Since W3C released XML [55], the de facto data format standard on the web, hundreds of XML tools have been developed. There exist XML editors, XML databases, XML converters, XML parsers, XML validators, XML search engines, XML encryptors, etc. Information about XML tools is available from many sites, see for example [18, 20]. Flynn's book [17] provides a description of some older tools.

Usage of DTDs in XML Tools. An XML document is usually structured according to a Document Type Definition (DTD) or a schema. An XML document is valid with respect to a DTD if it is structured according to the rules (elements) specified in the DTD. So a validator is a tool that critically depends on a DTD. Some other classes of tools, such as the class of XML editors, also critically depend on the presence of a DTD. An XML editor can only support editing of an XML document well, for example by suggesting possible children or listing attributes of an element, if it knows about the element structure and attributes of elements. These classes of tools depend on a DTD, and do the same thing (modulo structure differences) for different DTDs. We claim that many classes of XML tools are generic programs, or would benefit from being viewed as generic programs. We call such tools *DTD-indexed XML tools.*

Generic Programming for XML Tools. Since DTD-indexed XML tools are generic programs, it should help to implement such tools as generic programs. Implementing an XML tool as a generic program has several advantages:

- *Development time*. Generic programming supports the construction of type- (or DTD-) indexed programs. So all processing of DTDs and programs defined on DTDs can be left to the compiler, and does not have to be implemented by the tool developer. Furthermore, the existsing library of often used basic generic programs, for example, for comparing, encoding, etc., can be used in generic programs for XML tools.
- *Correctness*. An instance of a typable generic program is typeable. This implies that valid documents will be transformed to valid documents, possibly structured according to another DTD. Thus generic programming supports constructing type correct XML tools.
- *Efficiency*. The generic programming compiler may perform all kinds of optimisations on the code, such as deforestation, partial evaluation or fusion, which are difficult to conceive or implement by an XML tool developer.

This section discusses which classes of XML tools are DTD indexed, and how they can be implemented as generic programs.

Generic programming can also be used for XML tools that are not DTD indexed, but then most of the above advantages no longer apply.

3.2 XML compressors

Compression for XML documents. XML documents may become (very) large because of the markup that is added to the content. Because of the repetitive structure of many XML documents, these documents can be compressed by quite a large factor.

Existing XML compressors. We know of four XML compressors[2]:

- XMLZip [12]. XMLzip cuts its argument XML file (viewed as a tree) at a certain depth, and compresses the upper part separately from the lower part, both using a variant of zip or LZW [59]. This allows fast access to documents, but results in worse compression ratios compared with the following compressors.
- XMill [40]. XMill is a compressor that separates the structure of the XML document from the contents, and compresses structure and contents separately. Furthermore, it groups related data items (such as dates), and it applies semantic compressors to data items with a particular structure.

[2] Actually, after we wrote this paragraph, we found two more XML compressors. This field seems to develop fast. The final version of these lecture notes will contain the references.

– ICT's XML-Xpress [30] is a commercial compression system for XML files that uses 'Schema model files' to provide support for files conforming to a specific XML schema. The basic idea of this system is the same as the idea underlying the compressor we will describe below.
– Millau [19] is a system for efficient encoding and streaming of XML structures. It also separates structure and content, and uses the associated schema (if present) for compressing the structure.

XML compression and DTDs. XML compressors are DTD indexed. For example, consider the following small XML file:

```
<book lang="English">
<title>  Dead Famous </title>
<author> Ben Elton   </author>
<date>   2001        </date>
</book>
```

This file may be compressed by separating the structure from the data, and compressing the two parts separately. For compressing the structure we can make good use of the DTD. If we know how many elements, say n, appear in the DTD (the DTD for the above document contains at least 4 elements), we can replace each occurrence of the markup of an element in an XML file which is valid with respect to the DTD by $log_2\ n$ bits. This simple idea is the main idea behind the following tool, and has been described in the context of data conversion by Jansson and Jeuring [31, 35].

3.3 Implementing an XML compressor as a generic program

We have implemented an XML compressor, called XComprez, as a generic program. XComprez separates structure from contents, compresses the structure using knowledge about the DTD, and compresses the contents using a variant of zip [59]. Thus we replace each element, or rather, the pair of open and close keywords of the element, by the minimal number of bits required for the element given the DTD. We distinguish four components in the tool:

– a component that translates a DTD to a data type,
– a component that separates a value of any data type into its structure and its contents,
– a component that encodes the structure replacing constructors by bits,
– and a component for compressing the contents.

Of course, we have also implemented a decompressor, but since it is dual, hence very similar, to the compressor, we omit its description. See the website for XComprez [37] for the latest developments on XComprez. The Generic Haskell source code for XComprez can be obtained from the website.

Translating a DTD to a data type. A DTD can be translated to one or more Haskell data types. For example, the following DTD:

```
<!ELEMENT book    (title,author,date,(chapter)*)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT author  (#PCDATA)>
<!ELEMENT date    (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
<!ATTLIST book lang (English | Dutch) #REQUIRED>
```

can be translated to the following data types:

```
data Book           = Book Book_Attrs Title Author Date [Chapter]
data Book_Attrs     = Book_Attrs{bookLang :: Lang}
data Lang           = English | Dutch
newtype Title       = Title String
newtype Author      = Author String
newtype Date        = Date String
newtype Chapter     = Chapter String
```

We have used the Haskell library HaXml [58], in particular the functionality in the module DtdToHaskell to obtain a data type from a DTD, together with functions for reading (parsing) and writing (pretty printing) valid XML documents to and from a value of the generated data type. For example, the following value of the above DTD:

```
<book lang="English">
<title>  Dead Famous  </title>
<author> Ben Elton    </author>
<date>   2001         </date>
<chapter>Introduction </chapter>
<chapter>Preliminaries</chapter>
</book>
```

is translated to the following value of the data type *Book*:

```
Book Book_Attrs{bookLang = English}
     (Title "␣␣Dead␣Famous␣␣")
     (Author "␣Ben␣Elton␣␣␣␣")
     (Date "␣␣␣2001␣␣␣␣␣␣␣␣␣")
     [Chapter "Introduction␣"
     , Chapter "Preliminaries"
     ]
```

An element is translated to a value of a data type using just constructors and no labelled fields. An attribute is translated to a value that contains a labelled field for the attribute. Thus we can use the Generic Haskell constructs *Con* and *Label* to distinguish between elements and attributes in generic programs.

Separating structure and contents. The contents of an XML document is obtained by extracting all `PCData` and all `CData` from the document. In Generic Haskell, the contents of a value of a data type is obtained by extracting all strings from the value. For the above example value, we obtain the following result:

```
["␣␣Dead␣Famous␣␣"
,"␣Ben␣Elton␣␣␣␣"
,"␣␣␣2001␣␣␣␣␣␣␣␣␣"
,"Introduction␣"
,"Preliminaries"
]
```

The generic function *extract*, which extracts all strings from a value of a data type, is defined as follows:

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Extract}\langle\!\langle \star \rangle\!\rangle\ t & = t \rightarrow [\mathit{String}] \\
\textbf{type}\ \mathit{Extract}\langle\!\langle \kappa \rightarrow \nu \rangle\!\rangle\ t & = \forall a\,.\,\mathit{Extract}\langle\!\langle \kappa \rangle\!\rangle\ a \rightarrow \mathit{Extract}\langle\!\langle \nu \rangle\!\rangle\ (t\ a) \\
\mathit{extract}\langle t :: \kappa \rangle & :: \mathit{Extract}\langle\!\langle \kappa \rangle\!\rangle\ t \\
\mathit{extract}\langle \mathit{Unit} \rangle\ \mathit{Unit} & = [\,] \\
\mathit{extract}\langle \mathit{String} \rangle\ s & = [s] \\
\mathit{extract}\langle \mathtt{:+:} \rangle\ eA\ eB\ (\mathit{Inl}\ x) & = eA\ x \\
\mathit{extract}\langle \mathtt{:+:} \rangle\ eA\ eB\ (\mathit{Inr}\ y) & = eB\ y \\
\mathit{extract}\langle \mathtt{:*:} \rangle\ eA\ eB\ (x\ \mathtt{:*:}\ y) & = eA\ x +\!\!+\ eB\ y \\
\mathit{extract}\langle \mathit{Con}\ c \rangle\ e\ (\mathit{Con}\ b) & = e\ b \\
\mathit{extract}\langle \mathit{Label}\ l \rangle\ e\ (\mathit{Label}\ b) & = e\ b
\end{array}
$$

Note that it is possible to give special instances of a type-indexed function on a particular type, as with $\mathit{extract}\langle \mathit{String} \rangle$ in the above definition. Furthermore, because DtdToHaskell translates any DTD to a data type of kind $\star$, we could have defined *extract* just on data types of kind $\star$. However, higher-order kinds pose no problems. Finally, note that the operator $+\!\!+$ in the product case is a source of inefficiency. It can be removed using the standard lifting to the function level approach.

The structure from an XML document is obtained by removing all `PCData` and `CData` from the document. In Generic Haskell, the structure, or *shape*, of a value of a data type is obtained by replacing all strings by units (empty tuples). Thus we obtain a value of a new data type, in which occurrences of the type *String* have been replaced by the type (). This is another example of a type-indexed data type [24]. For example, the type we obtain from the data type

Book is isomorphic to the following data type:

```
data SHAPEBook          = SHAPEBook SHAPEBook_Attrs
                                    SHAPETitle
                                    SHAPEAuthor
                                    SHAPEDate
                                    [SHAPEChapter]
data SHAPEBook_Attrs    = SHAPEBook_Attrs{bookLang :: SHAPELang}
data SHAPELang          = SHAPEEnglish | SHAPEDutch
newtype SHAPETitle      = SHAPETitle ()
newtype SHAPEAuthor     = SHAPEAuthor ()
newtype SHAPEDate       = SHAPEDate ()
newtype SHAPEChapter    = SHAPEChapter ()
```

and the structure of the example value is

```
shapeBook = SHAPEBook  ( SHAPEBook_Attrs{bookLang = SHAPEEnglish})
                       ( SHAPETitle ())
                       ( SHAPEAuthor ())
                       ( SHAPEDate ())
                       [ SHAPEChapter ()
                       , SHAPEChapter ()
                       ]
```

The type-indexed data type *SHAPE* replaces occurrences of *String* in a data type by *Unit*.

```
type SHAPE⟨Unit⟩        = SH1 Unit
type SHAPE⟨String⟩      = SHString Unit
type SHAPE⟨:+:⟩ sa sb   = SHEither (Sum sa sb)
type SHAPE⟨:*:⟩ sa sb   = SHProd (Prod sa sb)
type SHAPE⟨Con⟩ sa      = SHCon (Con sa)
type SHAPE⟨Label⟩ sa    = SHLabel (Label sa)
```

The generic function *shape* returns the shape of a value of any data type, using the constructors of the type-indexed data type *SHAPE*.

```
type Shape⟪⋆⟫ t       = t → SHAPE⟨t⟩
type Shape⟪κ → ν⟫ t = ∀a . Shape⟪κ⟫ a → Shape⟪ν⟫ (t a)

shape⟨t :: κ⟩                   :: Shape⟪κ⟫ t
shape⟨Unit⟩ u                   = SH1 Unit
shape⟨String⟩ s                 = SHString Unit
shape⟨:+:⟩ sa sb (Inl a)        = SHEither (Inl (sa a))
shape⟨:+:⟩ sa sb (Inr b)        = SHEither (Inr (sb b))
shape⟨:*:⟩ sa sb (a :*: b)      = SHProd ((sa a) :*:(sb b))
shape⟨Con c⟩ sa (Con b)         = SHCon (Con (sa b))
shape⟨Label l⟩ sa (Label b)     = SHLabel (Label (sa b))
```

Given the shape and the contents (obtained by means of function *extract*) of a value we obtain the original value by means of function *insert*:

$$insert\langle t :: \star\rangle :: SHAPE\langle t\rangle \to [String] \to t$$

The type-indexed definition (with a kind-indexed type) of function *insert* is omitted.

Encoding constructors. A constructor of a value of a data type is encoded as follows. First calculate the number n of constructors of the data type. Then calculate the position of the constructor in the list of constructors of the data type. Finally, replace the constructor by the bit representation of its position, using $\log_2\ n$ bits. For example, in a data type with 6 constructors, the third constructor is encoded by `010`. Note that we start counting with 0. Furthermore, note that a value of a data type with a single constructor is represented using 0 bits. So the values of all types except for *String* and *Lang* in the example are represented using 0 bits.

All constructor descriptions of a data type can be obtained by means of function *constructors* from the module Collect, which can be found in the library of Generic Haskell.

$$constructors\langle t :: \star\rangle :: [ConDescr]$$

Function *constructors* is defined for arbitrary kinds in module Collect. The generic function *encode* takes a shape value, and encodes it. Since it needs the constructors of a data type to encode a shape value, function *encode* depends on function *constructors*.

$$\textbf{dependency}\ encode \leftarrow constructors\ encode$$

On types of kind $\star$, *encode* takes maybe a list of constructor descriptions (the constructor descriptions currently in scope: note that this list may change when traversing a value of a data type that refers to other data types) and a shape value, and returns a list of bits.

$$\begin{aligned}
&\textbf{type}\ Encode\langle\!\langle\star\rangle\!\rangle\ t = Maybe\ [ConDescr] \to SHAPE\langle t\rangle \to [Bit]\\
&\textbf{type}\ Encode\langle\!\langle\kappa \to \nu\rangle\!\rangle\ t =\\
&\quad \forall u\,.\ Collect0\langle\!\langle\kappa\rangle\!\rangle\ [ConDescr] \to Encode\langle\!\langle\kappa\rangle\!\rangle\ u \to Encode\langle\!\langle\nu\rangle\!\rangle\ (t\ u)\\
&\textbf{type}\ Collect0\langle\!\langle\star\rangle\!\rangle\ a = a\\
&\textbf{type}\ Collect0\langle\!\langle\kappa \to \nu\rangle\!\rangle\ a = \forall u\,.\ Collect0\langle\!\langle\kappa\rangle\!\rangle\ a \to Collect0\langle\!\langle\nu\rangle\!\rangle\ a
\end{aligned}$$

The only interesting cases in the definition of function *encode* are the sum and constructor case. We first give the uninteresting cases:

$$\begin{aligned}
&encode\langle t :: \kappa\rangle :: Encode\langle\!\langle\kappa\rangle\!\rangle\ t\\
&encode\langle Unit\rangle = \lambda_\ _ \to [\,]\\
&encode\langle String\rangle = \lambda_\ _ \to [\,]\\
&encode\langle{:}{*}{:}\rangle\ cA\ eA\ cB\ eB =\\
&\quad \lambda_\ (SHProd\ (a :\!*\!: b)) \to eA\ Nothing\ a +\!\!+\ eB\ Nothing\ b\\
&encode\langle Label\ l\rangle\ cA\ eA = \lambda_\ (SHLabel\ (Label\ a)) \to eA\ Nothing\ a
\end{aligned}$$

For *Unit* and *String* there is nothing to encode. Note that the product case takes four arguments, which correspond to the constructors and encoding of the left and right argument of the product, respectively. The encoding functions for the arguments are called with no constructor descriptions (*Nothing*), since whenever a new constructor is encountered in a product, it might be from another data type, and the constructors have to be recalculated.

In the sum case we calculate the constructors, if necessary (implemented by *maybe*), and encode the arguments of sum with the constructors. The encoding happens in the constructor case of function *encode*. We use function *intinrange2bits* to calculate the bits for the position of the argument constructor in the constructor list, given the number of constructors of the data type currently in scope. The definition of *intinrange2bits* is omitted.

```
encode⟨:+:⟩ cA eA cB eB =
  let cR = cA ++ cB
  in λcs → let cs' = maybe cR id cs
           in eA (Just cs') `shapejunc` eB (Just cs')
encode⟨Con c⟩ cA eA      =
  λcs (SHCon (Con a)) →
    let cs' = maybe [c] id cs
    in intinrange2bits (length cs') (fromJust (elemIndex c cs'))
       ++ eA Nothing a

  shapejunc :: (a → c) → (b → c) → (SHAPE⟨:+:⟩ a b) → c
  shapejunc f g (SHEither (Inl x)) = f x
  shapejunc f g (SHEither (Inr x)) = g x

  intinrange2bits :: Int → Int → [Bit]
```

We omit the definitions of the functions to decode a list of bits into a value of a data type. These functions are the inverses of the functions defined in this section.

Compressing the contents. Finally, the contents of an XML document have to be compressed. At the moment we use zip to compress the strings obtained from the document. In the future, we envisage more sophisticated compression methods for the contents, similar to the methods used in XMill.

Huffman coding. A relatively simple way to improve XComprez it is to analyze some source files that are valid with respect to the DTD, count the number of occurrences of the different elements (constructors), and apply Huffman coding. We have implemented this rather simple extension [37].

Analysis. How does the compressor described in the previous subsection compare with the existing XML compressors? The following analysis is limited, because we have not been able to obtain the executables or the source code of some of the existing compressors. Since the goal of XMLZip is different from our and the other compressors goal (fast access to compressed documents), we do not compare with XMLZip.

Compression ratio. XML-Xpress has been tested extensively against XMill, and achieves compression results that are about 80% better than XMill. We have performed some initial tests comparing XComprez and XMill. The tests are not representative, and it is impossible to draw hard conclusions from the results. However, on our test examples XComprez is 40% to 50% better than XMill. We think this improvement in compression ratio is considerable. As a schema contains more information about an XML document than a DTD, it is not surprising that our compressor does not achieve the same compression ratios as XML-Xpress. However, when we replace HaXml by a tool that generates a data type for a schema, we expect that we can achieve similar compression ratios as XML-Xpress. We have not been able to test against Millau, but from its description we expect that Millau achieves compression ratios that are a bit worse than the compression ratios achieved by XComprez, as Millau uses a fixed number of bits for some elements or attributes, independent of the DTD or Schema.

Code size. With respect to code size, the difference between XMill and XComprez is dramatic: XMill is written in almost 20k lines of C++. The main functionality of XComprez is less than 300 lines of Generic Haskell code. Of course, for a fair comparison we have to add some of the HaXml code (which is a library distributed together with almost all compiler and interpreters for Haskell), the code for handling bits, and the code for implementing the as yet unimplemented features of XMill. We expect to be able implement all of XMill's features in about 20% of the code size of XMill. We have not been able to obtain the source code of the (commercial) XML-Xpress.

3.4 DTD-indexed XML Tools

This section discusses whether or not several classes of XML tools are DTD indexed. We briefly introduce each of the classes of tools, and we discuss whether the class is DTD indexed and whether the available tools make use of this fact. Furthermore, whenever applicable, we discuss where HaXml and Generic Haskell might help in implementing an XML tool. Note that some (classes of) XML tools develop very fast, and that some of the information given in this section may be out of date.

We will discuss the following classes of tools:

- XML converters, parsers, and validators
- XML databases and search tools
- XML editors
- XML encryptors
- XML publishing tools
- XML version management tools

The class of XML compressors does not appear in this list, but has been discussed in the previous section. This is not a complete list of classes of XML tools, but this list includes many of the XML tools in use today.

XML converters, parsers, and validators. Since they are very similar, we discuss the classes of XML converters, parsers, and validators together.

For an overview of XML parsers, see [18]. There are several variants of XML parsers. Most XML parsers parse an arbitrary XML document to a universal tree (a DOM). DTDs play no role when parsing: validity of an XML document with respect to a DTD is checked in a separate phase, for example by an XML validator. These parsers are not DTD indexed.

Using Generic Haskell to develop an XML parser we would obtain a tool that takes a DTD as argument, and returns a parser for documents of the argument DTD. Thus the parser is automatically a validator. Any element that would turn the document into an invalid document would lead to a parse error. The HaXml library, in particular the module DtdToHaskell, contains a generic parser of this kind. Since this technology lies at the basis of our tools, we want to reimplement the read and show functions from DtdToHaskell in Generic Haskell, and add a module SchemaToHaskell.

For an overview of XML converters, see [18]. There exist two classes of XML converters: XML to XML converters, and non-XML to XML converters. For both of these classes there exist specific and generic (that is DTD-indexed) tools. An example of a specific non-XML to XML converter is RTF2XML [28]. Examples of generic non-XML to XML converters are Some2xml and Jedi [53, 26]. In both Some2xml and Jedi it is possible to specify patterns that are to be mapped on XML. If it were also possible to specify the result DTD, we would obtain a generic parser. At the moment Generic Haskell is of little use here, but future versions of Generic Haskell might offer functionality that is useful for implementing generic converters. The converters from XML to another format are discussed in the XML publishing tools section.

XML databases and search tools. XML documents can be searched by means of queries. Since XML query languages also play an important role in XML databases, we discuss XML databases and search tools under a common heading.

XML databases are used to store XML documents in a database. There are several ways to store an XML document in a database, but each of these can be classified as either structured or unstructured. The XML databases that store documents in a structured way are DTD-indexed XML tools. The DTD is used to determine the tables in the database, and may be used to optimise queries etc. Being DTD indexed can be of great help when searching or querying XML documents: indexes can be built based on DTDs, subtrees can be skipped when searching, etc. According to Abiteboul et al [1] current databases are not DTD indexed. However, the field of XML databases is developing fast, and we expect that there may already be DTD-indexed XML databases. Since most of these tools are commercial tools, see for example [11], it is difficult to check whether or not they are DTD indexed, and to compare implementations.

XML editors. An XML editor supports editing an XML document. Most XML editors support viewing XML documents in different ways, and they suggest elements and attributes that may be inserted at a given position. There are too many XML editors to list here. An incomplete list of XML editors can be found at the Proxima site [38]. An XML editor is a nice example of a DTD-indexed XML tool. Most XML editors are DTD indexed, and we think they should be. We have started on the core of a generic editor [14], but a lot of work remains in order to obtain a full-fledged XML editor. Again, as most of the existing XML editors are commercial tools, see for example [52, 2], it is difficult to compare implementations.

XML encryptors. An XML encryptor encrypts an XML document. Since the encrypted document gives nothing away of the structure of the input document, we see no application for generic programming here. Indeed, encryption would be weaker if it were based on the structure of a document.

XML publishing tools. An XML publishing tool, like for example Cocoon [3], takes an XML document and a target type on which to publish the document, and maybe a style sheet for this type, and returns a document which can be published. The tool traverses the input document, using the style sheet. Existing publishing tools are not DTD indexed. DTD indexing might help in constructing a publishing tool, since knowledge about the DTD can be used to optimise the traversal.

XML version management tools. IBM has developed a tool called treediff [29] which compares two XML files and points out the differences between the two files. A similar tool has been developed by Dommit [15]. We think that it would not be difficult to implement such a tool in Generic Haskell. Chawathe [5] has developed algorithms for comparing hierarchally structured data (such as XML documents). It is easy to implement the minimum-cost edit distance algorithm given by Chawathe as a generic program, by printing values to the format expected by the algorithm, and parsing, to a value of the original type, the tree obtained by applying the minimum cost edit script to the printed argument. Types do not play an essential role in this algorithm.

3.5 Related work

Most XML tools are built using the DOM or the SAX for manipulating XML documents. Using the DOM or the SAX usually implies that an XML document does not have a type. It follows that these standards are not a lot of help when developing tools that critically depend on the type (DTD) of a document.

There are a number of XML-specific (query) languages, such as for example XDuce [25], XMλ [47, 51], XSLT [56], XML query algebras [16], Yatl [9]. In many of these languages, XML documents are native values. Each of these languages has a number of features, such as regular expression pattern matching, type inference, or regular expression types, that support the construction of programs that manipulate XML documents. However, none of these languages have features that support the construction of DTD-indexed XML tools. We expect that extending XMλ with a construct that supports defining DTD-indexed functions would result in a very useful language for developing XML tools, but unfortunately an implementation of XMλ does not seem to exist.

3.6 Conclusions

We have shown that the combination of HaXml and generic programming as in Generic Haskell is very useful for implementing DTD-indexed XML tools. Using generic programming, such tools become easier to write, because a lot of the code pertaining to DTD handling and optimisation is obtained from the generic programming compiler, and the resulting tools are more effective, because they directly depend on the DTD. For example, DTD-indexed XML compressors, such as XComprez described in this paper, compress considerably better than XML compressors that don't take the DTD into account, such as XMill. Furthermore, our compressor is much smaller than XMill.

It remains to develop other DTD-indexed XML tools, and a library that supports the development of XML tools using Generic Haskell. We have started on an XML editor in Generic Haskell, see [14], but a lot of work remains to be done. However, we hope to (further) develop at least our XML compressor, an XML editor, part of an XML version management tool, and an XML database this year.

Although we think Generic Haskell is very useful for developing DTD-indexed XML tools, there are some features of XML tools that are harder to express in Generic Haskell. Some of the functionality in the DOM, such as the methods `childNodes` and `firstChild` in the `Node` interface, is hard to express in a typed way. Flexible extensions of type-indexed data types [24] might offer a solution to this problem. We think fusing HaXml, or a tool based on Schemas, with Generic Haskell, obtaining a 'domain-specific' language [6] for generic programming on DTDs or Schemas is a promising approach.

For tools that do not depend on a DTD we can use the untyped approach from HaXml to obtain a tool that works for any document. However, most of the advantages of Generic Programming no longer apply.

Exercise 3. (Easy exercise about XComprez.) In order to implement Huffman coding for XComprez, we have to analyse representative documents of a data type. So we want to count the constructors that appear in a value of a data type. For example,

$$\textbf{data}\ \mathit{Tree} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Node}\ \mathit{Tree}\ \mathit{Int}\ \mathit{Tree}$$
$$?\mathit{countCon}\ (\mathit{Node}\ (\mathit{Leaf}\ 1)\ 3\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ 1\ (\mathit{Leaf}\ 5)))$$
$$[(\texttt{"Leaf"}, 3), (\texttt{"Node"}, 2)]$$
$$\textbf{data}\ \mathit{List} = \mathit{Nil} \mid \mathit{Cons}\ \mathit{Char}\ \mathit{List}$$
$$?\mathit{countCon}\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 6\ \mathit{Nil})))$$
$$[(\texttt{"Nil"}, 1), (\texttt{"Cons"}, 3)]$$

Define the type-indexed function *countCon*, together with its kind-indexed type.

Exercise 4. (More involved exercise about adapting a generic program.) Adapt the current version of XCOMPREZ such that it can use Huffman coding instead of the standard constructor encoding used in this section. Make sure also other encodings can be used. Solutions to these exercises can be found on the webpage for XCOMPREZ.

4 The zipper

This section shows how to define a zipper for an arbitrary data type. This is an advanced example demonstrating the full power of a type-indexed data type 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*:

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

If the left subtree of a *Node* constructor is selected, moving right means moving to the right tree, not to the label of type a. 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.

The Generic Haskell code for this section can be downloaded from the applications page on `http://www.generic-haskell.org/`. It is also distributed together with the Generic Haskell compiler.

4.1 Preliminaries

Data types as fixed points. As mentioned above, in order to use the zipper, we have to be explicit about the fixed points in data type definitions.

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

For example, the data types of natural numbers and trees can be defined as fixed points as follows:

$$\begin{aligned}
&\textbf{data}\ \mathit{NatF}\ a &&= \mathit{ZeroF} \mid \mathit{SuccF}\ a\\
&\textbf{type}\ \mathit{Nat} &&= \mathit{Fix}\ \mathit{NatF}\\
&\textbf{data}\ \mathit{TreeF}\ a &&= \mathit{LeafF}\ \mathit{Char} \mid \mathit{ForkF}\ a\ a\\
&\textbf{type}\ \mathit{Tree} &&= \mathit{Fix}\ \mathit{TreeF}
\end{aligned}$$

It is easy to convert between data types as fixed points and the original data type definitions of natural numbers and trees. Note that nested data types and mutually recursive data types cannot be defined in terms of this particular definition of *Fix*.

The identity type-indexed data type. In the following subsections we will frequently use the identity type-indexed data type *Gid*. The definition of *Gid* is omitted. We assume there exist functions *mkid* and *unid*, which turn a value of type t into a value of type $\mathit{Gid}\langle t\rangle$ and vice versa:

$$\begin{aligned}
&\mathit{mkid}\langle t :: \kappa\rangle &&:: \mathit{MkId}\langle\!\langle\kappa\rangle\!\rangle\ t\\
&\textbf{type}\ \mathit{MkId}\langle\!\langle\star\rangle\!\rangle\ t &&= t \to \mathit{Gid}\langle t\rangle\\
&\textbf{type}\ \mathit{MkId}\langle\!\langle\kappa \to \nu\rangle\!\rangle\ t &&= \forall u\,.\,\mathit{MkId}\langle\!\langle\kappa\rangle\!\rangle\ u \to \mathit{MkId}\langle\!\langle\nu\rangle\!\rangle\ (t\ u)\\
&\mathit{unid}\langle t :: \kappa\rangle &&:: \mathit{UnId}\langle\!\langle\kappa\rangle\!\rangle\ t\\
&\textbf{type}\ \mathit{UnId}\langle\!\langle\star\rangle\!\rangle\ t &&= \mathit{Gid}\langle t\rangle \to t\\
&\textbf{type}\ \mathit{UnId}\langle\!\langle\kappa \to \nu\rangle\!\rangle\ t &&= \forall u\,.\,\mathit{UnId}\langle\!\langle\kappa\rangle\!\rangle\ u \to \mathit{UnId}\langle\!\langle\nu\rangle\!\rangle\ (t\ u)
\end{aligned}$$

Function *mkid* and *unid* are each others inverse.

Lifted Maybe. The lifted *Maybe* type is called *LMaybe*, and is defined by:

$$\textbf{data}\ \mathit{LMaybe}\ f\ a = \ \mathit{LNothing} \mid \mathit{LJust}\ (f\ a)$$

Functions $\mathit{unlift} :: \mathit{LMaybe}\ f\ a \to \mathit{Maybe}\ (f\ a)$ and $\mathit{lift} :: \mathit{Maybe}\ (f\ a) \to \mathit{LMaybe}\ f\ a$ convert between the two types. We will use the functions out_{-} and in_{-}

$$\begin{aligned}
\mathit{out}_{-} &= \mathit{unlift}\,.\,\mathit{out}\\
\mathit{in}_{-} &= \mathit{In}\,.\,\mathit{lift}
\end{aligned}$$

4.2 Locations

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

$$\begin{aligned}
&\mathit{Loc}\langle f :: \star \to \star\rangle &&:: \star\\
&\textbf{type}\ \mathit{Loc}\langle f\rangle &&= (\mathit{Fix}\ f, \mathit{Context}\langle f\rangle\ (\mathit{Fix}\ f))\\
&\mathit{Context}\langle f :: \star \to \star\rangle &&:: \star \to \star\\
&\textbf{type}\ \mathit{Context}\langle f\rangle\ r &&= \mathit{Fix}\ (\mathit{LMaybe}\ (\mathit{Ctx}\langle f\rangle\ r)).
\end{aligned}$$

The type *Loc* is defined in terms *Context*, which constructs the context parameterized by the original tree type. Note that we use generic abstraction on types here, instead of on generic functions. This feature has not been added to Generic Haskell yet, so in the actual Generic Haskell code, *Loc* and *Context* are replaced by their right-hand sides. The *Context* of a value is either empty (represented by *LNothing* in the *LMaybe* type), or it is a path from the root down into the tree. Such a path is constructed by means of the argument type of *LMaybe*: the type-indexed data type *Ctx*. The type-indexed data type *Ctx* is defined by induction on the pattern functor f of the original data type. It can be seen as the *derivative* (as in calculus) of the type f. If the derivative of f is denoted by f', we have

$$\begin{aligned} \mathit{const}' &= 0 \\ (f + g)' &= f' + g' \\ (f \times g)' &= f' \times g + f \times g' \end{aligned}$$

It follows that *Ctx* depends on the identity type-indexed data type *Gid*. Dependencies on type-indexed data types work in the same way as dependencies on type-indexed functions. The identity type-indexed data type is only used in the product case for *Ctx*.

```
dependency Ctx ← Gid Ctx
Ctx⟨f :: ⋆ → ⋆⟩                 :: ⋆ → ⋆ → ⋆
type Ctx⟨Unit⟩                  = CTXUnit 0
type Ctx⟨Int⟩                   = CTXInt 0
type Ctx⟨Char⟩                  = CTXChar 0
type Ctx⟨:+:⟩ iA cA iB cB = CTXSum (Sum cA cB)
type Ctx⟨:*:⟩ iA cA iB cB = CTXProd (Sum (Prod cA iB) (Prod iA cB))
type Ctx⟨Con⟩ iA cA             = CTXCon cA
type Ctx⟨Label⟩ iA cA           = CTXLab cA
```

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. 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 and trees we obtain ismorphic versions of the following context types:

```
type ContextNat r  = Fix (LMaybe (NatC r))
type ContextTree r = Fix (LMaybe (TreeC r))
data NatC r a      = ZeroC 0 | SuccF a
data TreeC r a     = LeafC 0 | NodeF (a, r) (r, a)
```

Note that if we assume 0 is a unit of + 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 *Tree* applied to the data type *Tree* itself is isomorphic to the type *Ctx_Tree* introduced in Section 1.

McBride [43] also defines a type-indexed zipper data type. His zipper slightly deviates from Huet's and our zipper: the navigation functions on McBride's zipper are not constant time anymore. The observation that the context of a data type is its derivative is due to McBride.

4.3 Navigation functions

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

Function down. The function *down* is a type-indexed function that moves down to the leftmost recursive child of the current node, if such a child exists. Otherwise, if the current node is a leaf node, then *down* returns the location unchanged. The instantiation of *down* to the data type *Tree* 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 $down\langle f\rangle \cdot up\langle f\rangle = id$ 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.

$$\begin{array}{ll} down\langle f :: \star \to \star\rangle & ::\ Loc\langle f\rangle \to Loc\langle f\rangle \\ down\langle f\rangle\ (t, c) & = \textbf{case}\ first\langle f\rangle\ (out\ t)\ c\ \textbf{of} \\ & \qquad Just\ (t', c') \to (t', in_{\_}\ (Just\ c')) \\ & \qquad Nothing \to (t, c). \end{array}$$

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 $Ctx\langle f\rangle\ r\ (Fix\ f)$) 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 *Just*, and applying the fixed point constructor $in_{\_}$ to it.

The value $out\ t$ is of type $f\ (Fix\ f)$. We want to obtain the leftmost occurrence of type $Fix\ f$ in $out\ t$. For this purpose we define *first* as a generic abstraction of a function $first'$.

$$\begin{array}{ll} first\langle f :: \star \to \star\rangle & ::\ f\ (Fix\ f) \to c \to Maybe\ (Fix\ f, Ctx\langle f\rangle\ (Fix\ f)\ c) \\ first\langle f\rangle\ x\ c & = first'\ (f)\ first'Rec\ id\ x\ c \end{array}$$

The first argument of $first'\ (f)$, function $first'Rec$, is the function that is applied to the values of type $Fix\ f$, when $x :: f\ (Fix\ f)$. Since we want to return a value of type *Fix f*, together with its context, function *first′Rec* is the curried version of *Just*:

$$first'Rec\ t\ c = Just\ (t, c)$$

Function *first′* does the real work. It depends on the identity function *mkid*.

$$\begin{aligned}
&\textbf{dependency}\ first' \leftarrow first'\ mkid\\
&first'\ (t :: \kappa) && :: \forall a\ c\,.\, First\langle\!\langle \kappa \rangle\!\rangle\ t\ a\ c\\
&\textbf{type}\ First\langle\!\langle \star \rangle\!\rangle\ t\ a\ c && = t \rightarrow c \rightarrow Maybe\ (a, Ctx\langle t \rangle)\\
&\textbf{type}\ First\langle\!\langle \kappa \rightarrow \nu \rangle\!\rangle\ t\ a\ c && = \forall u\,.\, First\langle\!\langle \kappa \rangle\!\rangle\ u\ a\ c \rightarrow MkId\langle\!\langle \kappa \rangle\!\rangle\ u \rightarrow First\langle\!\langle l \rangle\!\rangle\ (t\ u)\ a\ c
\end{aligned}$$

Because of the dependency, *first′* is also applied to the identity function in the generic abstraction above.

$$\begin{aligned}
&first'\langle Unit \rangle\ t\ c && = Nothing\\
&first'\langle Int \rangle\ t\ c && = Nothing\\
&first'\langle Char \rangle\ t\ c && = Nothing\\
&first'\langle \texttt{:+:} \rangle\ fA\ mA\ fB\ mB\ (Inl\ x)\ c && = \textbf{do}\ (t, cx) \leftarrow fA\ x\ c\\
& && \quad\quad return\ (t, CTXSum\ (Inl\ cx))\\
&first'\langle \texttt{:+:} \rangle\ fA\ mA\ fB\ mB\ (Inr\ y)\ c && = \textbf{do}\ (t, cy) \leftarrow fB\ y\ c\\
& && \quad\quad return\ (t, CTXSum\ (Inr\ cy))\\
&first'\langle \texttt{:*:} \rangle\ fA\ mA\ fB\ mB\ (x \texttt{ :*: } y)\ c && = (\textbf{do}\ (t, cx) \leftarrow fA\ x\ c\\
& && \quad\quad\quad return\ (t, CTXProd\ (Inl\ (cx \texttt{ :*: } mB\ y)))\\
& && \quad)\\
& && \quad `mplus`\\
& && \quad (\textbf{do}\ (t, cy) \leftarrow fB\ y\ c\\
& && \quad\quad\quad return\ (t, CTXProd\ (Inr\ (mA\ x \texttt{ :*: } cy)))\\
& && \quad)\\
&first'\langle Con\ d \rangle\ fA\ mA\ (Con\ t)\ c && = \textbf{do}\ (t, cx) \leftarrow fA\ t\ c\\
& && \quad\quad return\ (t, CTXCon\ cx)
\end{aligned}$$

Here, *return* is obtained from the *Maybe* monad, and *mplus* is the standard monadic plus, given by

$$\begin{aligned}
&mplus && :: Maybe\ a \rightarrow Maybe\ a \rightarrow Maybe\ a\\
&Nothing\ `mplus`\ m && = m\\
&Just\ a\ `mplus`\ m && = Just\ a.
\end{aligned}$$

The function *first* returns the value and the context at the leftmost recursive 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.

$$
\begin{aligned}
&up\langle f :: \star \to \star\rangle :: Loc\langle f\rangle \to Loc\langle f\rangle \\
&up\langle f\rangle\ (t, c) \quad = \mathbf{case}\ out\ c\ \mathbf{of} \\
&\qquad\qquad Nothing \to (t, c) \\
&\qquad\qquad Just\ c' \to fromJust\$ \\
&\qquad\qquad\qquad \mathbf{do}\ \{ft \leftarrow insert\langle f\rangle\ c'\ t; \\
&\qquad\qquad\qquad\quad c'' \leftarrow extract\langle f\rangle\ c'; \\
&\qquad\qquad\qquad\quad return\ (In\ ft, c'')\}.
\end{aligned}
$$

Remember that *Nothing* 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 $Ctx\langle f\rangle\ t\ c$ 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. Just as function *first*, function *extract* is defined as a generic abstraction of function *extract'*.

$$
\begin{aligned}
&extract\langle f :: \star \to \star\rangle && :: Ctx\langle f\rangle\ t\ c \to Maybe\ c \\
&extract\langle f\rangle\ c && = extract'\langle f\rangle\ extract'Rec\ c \\
&extract'Rec\ c && = Just\ c \\
&extract'\langle t :: \kappa\rangle && :: \forall a\,.\, Extract\langle\!\langle \kappa\rangle\!\rangle\ t\ a \\
&\mathbf{type}\ Extract\langle\!\langle \star\rangle\!\rangle\ t\ a && = Ctx\langle t\rangle \to Maybe\ a \\
&\mathbf{type}\ Extract\langle\!\langle \kappa \to \nu\rangle\!\rangle\ t\ a && = \forall u\,.\, Extract\langle\!\langle \kappa\rangle\!\rangle\ u\ a \to Extract\langle\!\langle \nu\rangle\!\rangle\ (t\ u)\ a
\end{aligned}
$$

Note that *extract* is polymorphic in t and in c. Function *extract'* is a simple function that traverses a context value.

$$
\begin{aligned}
&extract'\langle Unit\rangle\ c && = Nothing \\
&extract'\langle Int\rangle\ c && = Nothing \\
&extract'\langle Char\rangle\ c && = Nothing \\
&extract'\langle \mathtt{:+:}\rangle\ eA\ eB\ (CTXSum\ (Inl\ cx)) && = eA\ cx \\
&extract'\langle \mathtt{:+:}\rangle\ eA\ eB\ (CTXSum\ (Inr\ cy)) && = eB\ cy \\
&extract'\langle \mathtt{:*:}\rangle\ eA\ eB\ (CTXProd\ (Inl\ (cx\ \mathtt{:*:}\ y))) && = eA\ cx \\
&extract'\langle \mathtt{:*:}\rangle\ eA\ eB\ (CTXProd\ (Inr\ (x\ \mathtt{:*:}\ cy))) && = eB\ cy \\
&extract'\langle Con\ c\rangle\ eA\ (CTXCon\ cx) && = eA\ cx \\
&extract'\langle Label\ l\rangle\ eA\ (CTXLab\ cx) && = eA\ cx
\end{aligned}
$$

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. To obtain such a tree, we have to remove the occurrences of constructors of the identity type-indexed data type, for which we use function *unid*.

$$\mathbf{dependency}\ insert' \leftarrow insert'\ unid$$

Function *insert* is defined in a similar fashion as *extract* and *first*. We first give a generic abstraction:

$$\begin{aligned}
&insert\langle f :: \star \to \star\rangle && :: Ctx\langle f\rangle\ (Fix\ f)\ c \to Fix\ f \to Maybe\ (f\ (Fix\ f))\\
&insert\langle f\rangle\ c\ t && = insert'\langle f\rangle\ insert'Rec\ id\ c\\
&\quad \mathbf{where}\ insert'Rec\ c && = Just\ t\\
&insert'\langle t :: \kappa\rangle && :: Insert\langle\!\langle \kappa\rangle\!\rangle\ t\\
&\mathbf{type}\ Insert\langle\!\langle \star\rangle\!\rangle\ t && = Ctx\langle t\rangle \to Maybe\ t\\
&\mathbf{type}\ Insert\langle\!\langle \kappa \to \nu\rangle\!\rangle\ t && = \forall u\,.\, Insert\langle\!\langle \kappa\rangle\!\rangle\ u \to UnId\langle\!\langle \kappa\rangle\!\rangle\ u \to Insert\langle\!\langle \nu\rangle\!\rangle\ (t\ u)
\end{aligned}$$

and then define the type-indexed function *insert'*

$$\begin{aligned}
&insert'\langle Unit\rangle\ c && = Nothing\\
&insert'\langle Int\rangle\ c && = Nothing\\
&insert'\langle Char\rangle\ c && = Nothing\\
&insert'\langle :+: \rangle\ iA\ uA\ iB\ uB\ (CTXSum\ (Inl\ cx)) && = \mathbf{do}\ x \leftarrow iA\ cx\\
& && \qquad\ return\ (Inl\ x)\\
&insert'\langle :+: \rangle\ iA\ uA\ iB\ uB\ (CTXSum\ (Inr\ cy)) && = \mathbf{do}\ y \leftarrow iB\ cy\\
& && \qquad\ return\ (Inr\ y)\\
&insert'\langle :*: \rangle\ iA\ uA\ iB\ uB\ (CTXProd\ (Inl\ (cx :*: y))) && = \mathbf{do}\ x \leftarrow iA\ cx\\
& && \qquad\ return\ (x :*: uB\ y)\\
&insert'\langle :*: \rangle\ iA\ uA\ iB\ uB\ (CTXProd\ (Inr\ (x :*: cy))) && = \mathbf{do}\ y \leftarrow iB\ cy\\
& && \qquad\ return\ (uA\ x :*: y)\\
&insert'\langle Con\ c\rangle\ iA\ uA\ (CTXCon\ cx) && = \mathbf{do}\ x \leftarrow iA\ cx\\
& && \qquad\ return\ (Con\ x)\\
&insert'\langle Label\ l\rangle\ iA\ uA\ (CTXLab\ cx) && = \mathbf{do}\ x \leftarrow iA\ cx\\
& && \qquad\ return\ (Label\ x)
\end{aligned}$$

Note that the extraction and insertion is happening in the application of the generic abstraction to the *Rec* case (such as *insert'Rec* and *extract'rec*): the helper functions 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

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

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 *Tree* 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.

$$
\begin{array}{lll}
right\langle f :: \star \to \star\rangle & :: & Loc\langle f\rangle \to Loc\langle f\rangle \\
right\langle f\rangle\ (t, c) & = & \mathbf{case}\ out_{-}\ c\ \mathbf{of} \\
& & \quad Nothing \to (t, c) \\
& & \quad Just\ c' \to \mathbf{case}\ next\langle f\rangle\ t\ c'\ \mathbf{of} \\
& & \qquad\qquad Just\ (t', c'') \to (t', in_{-}\ (Just\ c'')) \\
& & \qquad\qquad Nothing \to (t, c).
\end{array}
$$

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

$$
\begin{array}{l}
\mathbf{do}\ \{(t', c') \leftarrow next\langle f\rangle\ t\ c; \\
\quad\ (t'', c'') \leftarrow previous\langle f\rangle\ t'\ c'; \\
\quad\ return\ (c == c'' \wedge t == t'')\}
\end{array}
$$

returns *True* (on locations in which it is possible to go to the right). We will give the heading of function *next*, and omit the definitions of $next'$ and *previous*.

$$
\begin{array}{lll}
next\langle f :: \star \to \star\rangle & :: & Fix\ f \to Ctx\langle f\rangle\ (Fix\ f)\ c \to Maybe\ (Fix\ f, Ctx\langle f\rangle\ (Fix\ f)\ c) \\
next\langle f\rangle\ t\ c & = & next'\langle f\rangle\ next'Rec \\
& & \qquad\qquad extract'Rec \\
& & \qquad\qquad insert'Rec \\
& & \qquad\qquad first'Rec \\
& & \qquad\qquad id\ id \\
& & \qquad\qquad c \\
next'Rec\ t & = & Just\ t
\end{array}
$$

$$\mathbf{dependency}\ next' \leftarrow next'\ extract'\ insert'\ first'\ mkid\ unid$$

The dependency shows that $next'$ is a rather complicated function that depends on five other generic functions. This is reflected in its type:

$$
\begin{array}{lll}
\mathbf{type}\ Next\langle\!\langle\star\rangle\!\rangle\ t\ a\ c & = & Ctx\langle t\rangle \to Maybe\ (a, Ctx\langle t\rangle) \\
\mathbf{type}\ Next\langle\!\langle\kappa \to \nu\rangle\!\rangle\ t\ a\ c & = & \forall u\,.\,Next\langle\!\langle\kappa\rangle\!\rangle\ u\ a\ c \to \\
& & \qquad Extract\langle\!\langle\kappa\rangle\!\rangle\ u\ c \to \\
& & \qquad Insert\langle\!\langle\kappa\rangle\!\rangle\ u \to \\
& & \qquad First\langle\!\langle\kappa\rangle\!\rangle\ u\ a\ c \to \\
& & \qquad MkId\langle\!\langle\kappa\rangle\!\rangle\ u \to \\
& & \qquad UnId\langle\!\langle\kappa\rangle\!\rangle\ u \to Next\langle\!\langle\nu\rangle\!\rangle\ (t\ u)\ a\ c
\end{array}
$$

The definition of function $next'$ can be found in [24].

Exercise 5. (Easy exercise about constructor selection.) If we don't want to use the zipper, we can also keep track of the path to the current focus. Suppose we want to use the path to determine the name of the top constructor of the current focus in a value of a data type. The path determines which child of a value is selected. Since the products used in our representations of data types are binary, a path has the following structure:

$$\begin{aligned}&\textbf{data}\ \mathit{Dir} &&= L \mid R\\ &\textbf{type}\ \mathit{Path} &&= [\mathit{Dir}]\end{aligned}$$

Function $\mathit{selectCon}\langle\rangle$ takes a value of a data type and a path, and returns the constructor name at the position denoted by the path. For example,

$\textbf{data}\ \mathit{List} = \mathit{Nil} \mid \mathit{Cons}\ \mathit{Char}\ \mathit{List}$

$?\mathit{selectCon}\langle \mathit{List}\rangle\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 6\ \mathit{Nil})))\ [R, R, R]$

```
"Nil"
```

$\textbf{data}\ \mathit{Tree} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Node}\ \mathit{Tree}\ \mathit{Int}\ \mathit{Tree}$

$?\ \mathit{selectCon}\langle \mathit{Tree}\rangle\ (\mathit{Node}\ (\mathit{Leaf}\ 1\ )\ \ 3\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ 1\ (\mathit{Leaf}\ 5)))\ [R, R]$

```
"Node"
```

$?\ \mathit{selectCon}\langle \mathit{Tree}\rangle\ (\mathit{Node}\ (\mathit{Leaf}\ 1\ )\ \ 3\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ 1\ (\mathit{Leaf}\ 5)))\ [R, R, L]$

```
"Leaf"
```

Define the type-indexed function $\mathit{selectCon}\langle\rangle$, together with its kind-indexed type.

Exercise 6. (Difficult exercise, in which you need (dual versions of) most of the functions defined in this section.) Define the function *left*, which takes a location, and returns the location to the left of the argument location, if possible.

Exercise 7. (Rather complicated exercise about an alternative representation of values for editing purposes.) For several applications we have to extend a data type such that it is possible to represent a place holder. For example, from the data type *Tree* defined by

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

we would like to obtain a type isomorphic to the following type:

$$\textbf{data}\ \mathit{HoleTree}\ a = \mathit{Hole} \mid \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{HoleTree}\ a)\ (\mathit{HoleTree}\ a)$$

– Define a type-indexed data type *Hole* that takes a data type and returns a data type in which also holes can be specified. Also give the kind-indexed kind of this type-indexed data type. (The kind-indexed kind cannot and does not have to be defined in Generic Haskell though.)
– Define a type-indexed function *toHole* which translates a value of a data type t to a value of the data type $\mathit{Hole}\langle t\rangle$, and a function *fromHole* that does the inverse for values that do not contain holes anymore, so:

$$\begin{aligned}&\mathit{toHole}\langle t :: \kappa\rangle &&:: \mathit{ToHole}\langle\!\langle \kappa\rangle\!\rangle\ t\\ &\mathit{fromHole}\langle t :: \kappa\rangle &&:: \mathit{FromHole}\langle\!\langle \kappa\rangle\!\rangle\ t\end{aligned}$$

$$\textbf{type } \mathit{ToHole}\langle\!\langle\star\rangle\!\rangle\ t \quad = t \rightarrow \mathit{Hole}\langle t\rangle$$
$$\textbf{type } \mathit{FromHole}\langle\!\langle\star\rangle\!\rangle\ t = \mathit{Hole}\langle t\rangle \rightarrow t$$

5 Conclusions

We have developed three advanced applications in Generic Haskell. In these examples we use, besides type-indexed functions with kind-indexed kinds, type-indexed data types, dependencies between and generic abstractions of generic functions, and default and constructor cases. Some of the latest developments of Generic Haskell have been guided by requirements from these applications.

We hope to develop more applications using Generic Haskell in the future, both to develop the theory and the language. Current candidate applications are more XML tools and editors.

Acknowledgements. Andres Löh implemented Generic Haskell and the zipper example, and contributed to almost all other examples. We want to thank the 2001 XML and Generic Programming class for investigating different classes of XML tools. Paul Hagg contributed to the implementation of the XML compressor. Dave Clarke, Andres Löh, Ralf Lämmel, Doaitse Swierstra and Jan de Wit commented on or contributed to (parts of) previous versions of this paper.

References

1. Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web.* Morgan Kaufmann Publishers, 2000.
2. Altova. XML Spy. Whitepaper available from `http://www.xmlspy.com`, 2002.
3. The Apache Project. Cocoon. Available from `http://xml.apache.org/cocoon/`, 2002.
4. 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.
5. S. Chawathe. Comparing hierarchical data in external memory. In *The Twenty-fifth International Conference on Very Large Data Bases*, pages 90–101, 1999.
6. Dave Clarke. Towards GH(XML). Talk at the Generic Haskell meeting, see `http://www.generic-haskell.org/talks.html`, 2001.
7. Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001. Also available from `http://www.generic-haskell.org/`.
8. Dave Clarke and Andres Löh. Generic Haskell, specifically. In *Proceedings Working Conference on Generic Programming*. Kluwer Academic Publishers, 2002. To appear.
9. Sophie Cluet and Jérôme Siméon. YATL: a functional and declarative language for XML, 2000.
10. Richard H. Connelly and F. Lockwood Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, September 1995.

11. X-Hive Corporation. X-Hive. Available from `http://www.xhive.com`, 2002.
12. XMLSolutions Corporation. XMLZip. Available from `http://www.xmlzip.com/`, 1999.
13. René de la Briandais. File searching using variable length keys. In *Proc. Western Joint Computer Conference*, volume 15, pages 295–298. AFIPS Press, 1959.
14. Jan de Wit. A technical overview of Generic Haskell. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.
15. Dommitt. XML Diff and Merge Tool. Available from `http://www.dommitt.com/`.
16. Mary Fernandez, Jérôme Siméon, and Philip Wadler. A semistructured monad for semistructured data. In *Proceedings ICDT*, 2001. Also available via `http://www.research.avayalabs.com/user/wadler/`.
17. Peter Flynn. *Understanding SGML and XML Tools*. Kluwer Academic Publishers, 1998.
18. Lars M. Garshol. Free XML tools and software. Available from `http://www.garshol.priv.no/download/xmltools/`.
19. Marc Girardot and Neel Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. In *IEEE International Conference on Multimedia and Expo (I) 2000*, pages 747–765, 2000.
20. Google. Web Directory on XML tools. `http://www.google.com/`.
21. Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, July 2000.
22. Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.
23. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory, 2002. To appear.
24. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th Mathematics of Program Construction Conference*, volume 2386 of *LNCS*, pages 148–174, 2002.
25. Haruo Hosoya and Benjamin C. Pierce. Xduce: A typed XML processing language. In *Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science*, pages 226–244, 1997,2000.
26. Gerald Huck, Peter Fankhauser, Karl Aberer, and Erich J. Neuhold. Jedi: Extracting and synthesizing information from the web. In *Conference on Cooperative Information Systems*, pages 32–43, 1998.
27. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
28. AlphaWorks IBM. RTF2XML. Available from `http://www.alphaworks.ibm.com/`, 2000.
29. AlphaWorks IBM. XML treediff. Available from `http://www.alphaworks.ibm.com/tech/xmltreediff`, 2000.
30. INC Intelligent Compression Technologies. XML-Xpress. Whitepaper available from `http://www.ictcompress.com/products_xmlxpress.html`, 2001.
31. P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In Doaitse Swierstra, editor, *ESOP'99*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.
32. Patrik Jansson. The WWW home page for polytypic programming. Available from `http://www.cs.chalmers.se/~patrikj/poly/`, 2001.
33. Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.
34. 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.
35. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
36. 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.
37. Johan Jeuring and Paul Hagg. XComprez. Available from `http://www.generic-haskell.org/xmltools/XComprez/`, 2002.
38. Johan Jeuring, Lambert Meertens, Steven Pemberton, Martijn Schrage, Gert van der Steen, and Doaitse Swierstra. Proxima - a generic presentation-oriented XML editor. Available from `http://www.cs.uu.nl/research/projects/proxima/`, 2002.
39. Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley Publishing Company, 2nd edition, 1998.
40. Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.
41. G. Malcolm. *Algebraic data types and program transformation.* PhD thesis, University of Groningen, 1990.
42. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
43. Connor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.
44. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.
45. Lambert Meertens. Functor pulling. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.
46. 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.
47. Erik Meijer and Mark Shields. XMLambda: A functional language for constructing and manipulating XML documents. Available from `http://www.cse.ogi.edu/~mbs/`, 1999.
48. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.
49. Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland*, pages 77–86, 1998.
50. 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.
51. Mark Shields and Erik Meijer. Type-indexed rows. In *The 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 261–275, 2001. Also available from `http://www.cse.ogi.edu/~mbs/`.
52. SoftQuad Software. XMetal. Available from `http://www.xmetal.com`, 2002.
53. Paul A. Tchistopolskii. Some2xml. Available from `http://www.pault.com/pault/old/Some2xml.html`, 1999.

54. Axel Thue. Über die gegenseitige lage gleicher teile gewisser zeichenreihen. *Skrifter udgivne af Videnskaps-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse*, 1:1–67, 1912. Reprinted in Thue's "Selected Mathematical Papers" (Oslo: Universitetsforlaget, 1977), 413–477.
55. W3C. XML 1.0. Available from `http://www.w3.org/XML/`, 1998.
56. W3C. XSL Transformations 1.0. Available from `http://www.w3.org/TR/xslt`, 1999.
57. C.P. Wadsworth. Recursive type operators which are more than type schemes. *Bulletin of the EATCS*, 8:87–88, 1979. Abstract of a talk given at the 2nd International Workshop on the Semantics of Programming Languages, Bad Honnef, Germany, 19–23 March 1979.
58. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159, 1999.
59. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.