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. Generic Haskell is an extension of Haskell that supports the construction of generic programs. These lecture notes discuss three advanced generic programming applications: generic dictionaries, compressing XML documents, and the zipper: a data structure 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. 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 generic (or polytypic, type-indexed) function is a function that can be instantiated on many data types to obtain data type specific functionality. Examples of generic functions are the functions that can be derived in Haskell [45], such as *show*, *read*, and '=='. In the first part of these lecture notes, entitled Generic Haskell: practice and theory [22], we have introduced generic functions, and we have shown how to implement them in Generic Haskell [9]. We assume the reader is familiar with the first part of these notes. For an introduction to a related but different kind of generic programming, see Backhouse et al [2].

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.

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.

These lecture notes describe three advanced generic programming applications: generic dictionaries, compressing XML documents, and the zipper. The 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 [23]. It is the equivalent of a type-indexed function 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 [10]. 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 [10]. 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 [10].

We will introduce these concepts as we go along.

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

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

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

$$\begin{aligned}\textbf{data}\ \mathsf{FMap\_String\ v} = {} & \mathit{Null\_String} \\ \mid {} & \mathit{Node\_String}\ (\mathsf{Maybe\ v}) \\ & \qquad (\mathsf{FMapChar}\ (\mathsf{FMap\_String\ v})).\end{aligned}$$

Such a trie for strings would typically be used for an index on texts. The constructor *Null_String* represents the empty trie. The first component of the constructor *Node_String* contains the value associated with *Nil*. The second component of *Node_String* is derived from the constructor $\mathit{Cons} :: \mathsf{Char} \to \mathsf{String} \to \mathsf{String}$. We assume that a suitable data structure, FMapChar, and an associated lookup function $\mathit{lookupChar} :: \forall \mathsf{v}\,.\,\mathsf{Char} \to \mathsf{FMapChar\ v} \to \mathsf{Maybe\ v}$ for characters are

predefined. Given these prerequisites we can define a look-up function for strings as follows:

$$
\begin{aligned}
&\mathit{lookup\_String} :: \mathsf{String} \to \mathsf{FMap\_String\ v} \to \mathsf{Maybe\ v} \\
&\mathit{lookup\_String}\ s\ \mathit{Null\_String} &&= \mathit{Nothing} \\
&\mathit{lookup\_String}\ \mathit{Nil}\ (\mathit{Node\_String}\ tn\ tc) &&= tn \\
&\mathit{lookup\_String}\ (\mathit{Cons}\ c\ s)\ (\mathit{Node\_String}\ tn\ tc) \\
&\quad = (\mathit{lookupChar}\ c \diamond \mathit{lookup\_String}\ s)\ tc.
\end{aligned}
$$

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

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

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

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

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

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

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

$$
\begin{aligned}
&\mathit{lookup\_Bush} :: \mathsf{Bush} \to \mathsf{FMap\_Bush\ v} \to \mathsf{Maybe\ v} \\
&\mathit{lookup\_Bush}\ b\ \mathit{Null\_Bush} &&= \mathit{Nothing} \\
&\mathit{lookup\_Bush}\ (\mathit{Leaf}\ c)\ (\mathit{Node\_Bush}\ tl\ tf) &&= \mathit{lookupChar}\ c\ tl \\
&\mathit{lookup\_Bush}\ (\mathit{Bin}\ bl\ br)\ (\mathit{Node\_Bush}\ tl\ tf) \\
&\quad = (\mathit{lookup\_Bush}\ bl \diamond \mathit{lookup\_Bush}\ br)\ tf.
\end{aligned}
$$

One can easily recognize that not only the look-up functions, but also the data types for the tries are instances of an underlying generic pattern. In the following section we will show how to define a trie and associated functions generically for arbitrary data types.

Example 2: Compressing XML documents. The extensible markup language XML [49] is used to mark up text with structure information. XML documents may become (very) large because of the markup that is added to the content. A good XML compressor can compress an XML document 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 of the document.

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

If we know how many elements and attributes, say n, appear in the DTD (the DTD above document contains 6 elements), we can replace the markup of an element in an XML file which is valid with respect to the DTD by a natural number between 0 and $n-1$, or by $log_2\ n$ bits. This is one of the main ideas behind XMill [36]. We improve on XMill by only recording the choices made in an XML document. In the above document, there is a choice for the language of the book, and the number of chapters it has. All the other elements are not encoded, since they can be inferred from the DTD. Section 3 describes a tool based on this idea, which was first described by Jansson and Jeuring in the context of data conversion [26, 30]. We use HaXml [51] 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 are 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 [24] is a data structure that is used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up, or down the tree. For example, the zipper corresponding to the data type Bush, called Loc_Bush, is defined by

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

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

$$
\begin{array}{ll}
\mathit{down\_Bush} & :: \mathsf{Loc\_Bush} \to \mathsf{Loc\_Bush} \\
\mathit{down\_Bush}\ (\mathit{Leaf}\ a, c) & = (\mathit{Leaf}\ a, c) \\
\mathit{down\_Bush}\ (\mathit{Bin}\ tl\ tr, c) & = (tl, \mathit{BinL}\ c\ tr) \\
\mathit{right\_Bush} & :: \mathsf{Loc\_Bush} \to \mathsf{Loc\_Bush} \\
\mathit{right\_Bush}\ (tl, \mathit{BinL}\ c\ tr) & = (tr, \mathit{BinR}\ tl\ c) \\
\mathit{right\_Bush}\ m & = m.
\end{array}
$$

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

Huet [24] defines the zipper data structure for rose trees and for the data type Bush, 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 (equivalents 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, test tools [7, 34], etc.
- *Typed term processing.* Functions like pattern matching, term rewriting and unification are generic functions, and have been implemented as generic functions in [31, 28, 29].

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* [41], together with a special constructor case for variables [10]. For more examples of generic traversals, see Lämmel and Peyton Jones [35].

The Generic Haskell code for the programs discussed in these lecture notes can be downloaded from the applications page on `http://www.generic-haskell.org/`.

On notation. To improve readability, the notation for generic functions we use in these notes slightly differs from the notation used in the first part of these notes [22]. For example, in the first part of these lecture notes we write:

$$\begin{aligned}
&equal\{\!|\mathsf{Char}|\!\} && = eqChar\\
&equal\{\!|\mathsf{Int}|\!\} && = eqInt\\
&equal\{\!|\mathsf{Unit}|\!\}\ Unit\ Unit && = True\\
&equal\{\!|\mathtt{:+:}|\!\}\ eqa\ eqb\ (Inl\ a)\ (Inl\ a') && = eqa\ a\ a'\\
&equal\{\!|\mathtt{:+:}|\!\}\ eqa\ eqb\ (Inl\ a)\ (Inr\ b') && = False\\
&equal\{\!|\mathtt{:+:}|\!\}\ eqa\ eqb\ (Inr\ b)\ (Inl\ a') && = False\\
&equal\{\!|\mathtt{:+:}|\!\}\ eqa\ eqb\ (Inr\ b)\ (Inr\ b') && = eqb\ b\ b'\\
&equal\{\!|\mathtt{:*:}|\!\}\ eqa\ eqb\ (a\ \mathtt{:*:}\ b)\ (a'\ \mathtt{:*:}\ b') && = eqa\ a\ a' \wedge eqb\ b\ b'\\
&equal\{\!|\mathsf{Con}\ c|\!\}\ eqa\ (Con\ a)\ (Con\ b) && = eqa\ a\ b.
\end{aligned}$$

The function *equal* is a generic function which recurses over the type structure of the argument type. The recursion is implicit in the arguments *eqa* and *eqb* in the `:+:` and `:*:` cases. In this part of the lecture notes we will instead write:

$$\begin{aligned}
&equal\{\!|\mathsf{Char}|\!\} && = eqChar\\
&equal\{\!|\mathsf{Int}|\!\} && = eqInt\\
&equal\{\!|\mathsf{Unit}|\!\}\ Unit\ Unit && = True\\
&equal\{\!|\mathsf{a}\ \mathtt{:+:}\ \mathsf{b}|\!\}\ (Inl\ a)\ (Inl\ a') && = equal\{\!|\mathsf{a}|\!\}\ a\ a'\\
&equal\{\!|\mathsf{a}\ \mathtt{:+:}\ \mathsf{b}|\!\}\ (Inl\ a)\ (Inr\ b') && = False\\
&equal\{\!|\mathsf{a}\ \mathtt{:+:}\ \mathsf{b}|\!\}\ (Inr\ b)\ (Inl\ a') && = False\\
&equal\{\!|\mathsf{a}\ \mathtt{:+:}\ \mathsf{b}|\!\}\ (Inr\ b)\ (Inr\ b') && = equal\{\!|\mathsf{b}|\!\}\ b\ b'\\
&equal\{\!|\mathsf{a}\ \mathtt{:*:}\ \mathsf{b}|\!\}\ (a\ \mathtt{:*:}\ b)\ (a'\ \mathtt{:*:}\ b') && = equal\{\!|\mathsf{a}|\!\}\ a\ a' \wedge equal\{\!|\mathsf{b}|\!\}\ b\ b'\\
&equal\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (Con\ a)\ (Con\ b) && = equal\{\!|\mathsf{a}|\!\}\ a\ b.
\end{aligned}$$

Here the recursion over the type structure is explicit: $equal\{\!|\mathsf{a}\ \mathtt{:*:}\ \mathsf{b}|\!\}$ is expressed in terms of $equal\{\!|\mathsf{a}|\!\}$ and $equal\{\!|\mathsf{b}|\!\}$. We think this style is more readable, especially when a generic function depends on another generic function. Functions written in the latter style can be translated to the former style and vice versa, so no expressiveness is lost or gained; the only difference is readability. The Generic Haskell compiler does not accept explicit recursive functions yet, but probably will do so in the near future. A formal description of Generic Haskell with explicit recursion is given by Löh et al. [37].

Organization. The rest of this paper is organized as follows. Section 2 introduces generic dictionaries, and implements them in Generic Haskell. Section 3 describes XCOMPREZ, a compressor for XML documents. Section 4 develops a generic zipper data structure. Section 5 summarizes the main points and concludes.

These lecture notes contain exercises. Solutions to the exercises can be found on the webpage for the Generic Haskell project: `www.generic-haskell.org`.

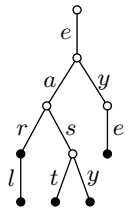
2 Generic dictionaries

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. [11] 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 material in this section is largely taken from [18].

2.1 Introduction

The concept of a trie was introduced by Thue in 1912 as a means to represent a set of strings, see [33]. 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:

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

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

```
data FMap_String v = Null_String
                   | Node_String (Maybe v)
                                 (FMapChar (FMap_String v))
```

Here, *Null_String* represents the empty trie. The first component of the constructor *Node_String* contains the value associated with *Nil*. Its type is Maybe v instead of v since *Nil* 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.

```
type FMapChar v = [(Char, v)]
lookupChar        :: ∀v . Char → FMapChar v → Maybe v
lookupChar c []   = Nothing
lookupChar c ((c', v) : x)
  | c < c'        = Nothing
  | c == c'       = Just v
  | c > c'        = lookupChar c x
```

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, *Cons c s* we look up c in the edge map obtaining a trie, which is then recursively searched for s.

```
lookup_String :: ∀v . String → FMap_String v → Maybe v
lookup_String s Null_String                  = Nothing
lookup_String Nil (Node_String tn tc)        = tn
lookup_String (Cons c s) (Node_String tn tc) =
  (lookupChar c ◇ lookup_String s) tc
```

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. [11] 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 *lookup_String* already gives a clue what a suitable generalization might look like: the trie *Node_String tn tc* contains a finite map for each constructor of the data type String; to look up *Cons 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 *Nil*), we extract the associated value.

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

```
data Dict = Tip String | Node Dict String Dict.
```

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

```
data FMap_Dict v = Null_Dict
                 | Node_Dict (FMap_String v)
                             (FMap_Dict (FMap_String (FMap_Dict v))).
```

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

$$
\begin{array}{l}
\mathit{lookup\_Dict} :: \forall \mathsf{v} \,.\, \mathsf{Dict} \to \mathsf{FMap\_Dict\ v} \to \mathsf{Maybe\ v} \\
\mathit{lookup\_Dict}\ d\ \mathit{Null\_Dict} \qquad\qquad\qquad\quad = \mathit{Nothing} \\
\mathit{lookup\_Dict}\ (\mathit{Tip}\ s)\ (\mathit{Node\_Dict}\ tl\ tn) \quad = \mathit{lookup\_String}\ s\ tl \\
\mathit{lookup\_Dict}\ (\mathit{Node}\ m\ s\ r)\ (\mathit{Node\_Dict}\ tl\ tn) = \\
\quad (\mathit{lookup\_Dict}\ m \diamond \mathit{lookup\_String}\ s \diamond \mathit{lookup\_Dict}\ r)\ tn
\end{array}
$$

Looking up a node involves two recursive calls. The first, *lookup_Dict* m, is of type $\mathsf{Dict} \to \mathsf{FMap\_Dict\ X} \to \mathsf{Maybe\ X}$ where $\mathsf{X} = \mathsf{FMap\_String\ (FMap\_Dict\ v)}$, which is a substitution instance of the declared type.

Note that it is absolutely necessary that FMap_Dict and *lookup_Dict* are parametric with respect to the codomain of the finite maps. If we restrict the type of *lookup_Dict* to $\mathsf{Dict} \to \mathsf{FMap\_Dict\ T} \to \mathsf{Maybe\ T}$ for some fixed type T, the definition no longer type-checks. 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 renders 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.

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

Here, $\mathsf{t} \to_{\mathrm{fin}} \mathsf{v}$ denotes the set of all finite maps from t to v. Note that $\mathsf{t} \to_{\mathrm{fin}} \mathsf{v}$ is sometimes written $\mathsf{v}^{[\mathsf{t}]}$, 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 kind for types t of kind $\star$.

$$\mathsf{FMap}\{\!|\mathsf{t} :: \star|\!\} :: \star \to \star$$

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

We will implement the following operations on tries.

$$
\begin{array}{ll}
\mathit{empty}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v} \,.\, \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \\
\mathit{isempty}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v} \,.\, \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{Bool} \\
\mathit{single}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v} \,.\, (\mathsf{t}, \mathsf{v}) \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \\
\mathit{lookup}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v} \,.\, \mathsf{t} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v} \\
\mathit{insert}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v} \,.\, (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \to (\mathsf{t}, \mathsf{v}) \to (\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v}) \\
\mathit{merge}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v} \,.\, (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \to (\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v}) \\
\mathit{delete}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v} \,.\, \mathsf{t} \to (\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v})
\end{array}
$$

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

2.3 Properties

Each of the operations specified in the previous section satisfies a number of laws, which hold generically for all instances of t. These properties formally specify parts of the informal descriptions of the operations given there, and can be proved for the definitions given in the following sections using fixed point induction. See Hinze [19, 21] for examples of proofs of properties of generic functions.

$$
\begin{aligned}
\mathit{lookup}\{\!|\mathsf{t}|\!\}\ k\ (\mathit{empty}\{\!|\mathsf{t}|\!\}) &= \mathit{Nothing} \\
\mathit{lookup}\{\!|\mathsf{t}|\!\}\ k\ (\mathit{single}\{\!|\mathsf{t}|\!\}\ (k_1, v_1)) &= \mathbf{if}\ k == k_1\ \mathbf{then}\ \mathit{Just}\ v_1\ \mathbf{else}\ \mathit{Nothing} \\
\mathit{lookup}\{\!|\mathsf{t}|\!\}\ k\ (\mathit{merge}\{\!|\mathsf{t}|\!\}\ c\ t_1\ t_2) &= \mathit{combine}\ c\ (\mathit{lookup}\{\!|\mathsf{t}|\!\}\ k\ t_1)\ (\mathit{lookup}\{\!|\mathsf{t}|\!\}\ k\ t_2),
\end{aligned}
$$

where *combine* combines two values of type Maybe:

$$
\begin{aligned}
&\mathit{combine} :: \forall \mathsf{v} \,.\, (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \to (\mathsf{Maybe}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}) \\
&\mathit{combine}\ c\ \mathit{Nothing}\ \mathit{Nothing} = \mathit{Nothing} \\
&\mathit{combine}\ c\ \mathit{Nothing}\ (\mathit{Just}\ v_2) = \mathit{Just}\ v_2 \\
&\mathit{combine}\ c\ (\mathit{Just}\ v_1)\ \mathit{Nothing} = \mathit{Just}\ v_1 \\
&\mathit{combine}\ c\ (\mathit{Just}\ v_1)\ (\mathit{Just}\ v_2) = \mathit{Just}\ (c\ v_1\ v_2).
\end{aligned}
$$

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 $merge\{\!|\mathsf{t}|\!\}\ c$ is associative, as well. Furthermore, $empty\{\!|\mathsf{t}|\!\}$ is the left and the right unit of $merge\{\!|\mathsf{t}|\!\}\ c$:

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

The functions *insert* and *delete* satisfy the following laws: for all k, v, and c,

$$\begin{aligned} &insert\{\!|\mathsf{t}|\!\}\ c\ (k,v)\ (empty\{\!|\mathsf{t}|\!\}) = single\{\!|\mathsf{t}|\!\}\ (k,v) \\ &delete\{\!|\mathsf{t}|\!\}\ k\ (single\{\!|\mathsf{t}|\!\}\ (k,v)) = empty\{\!|\mathsf{t}|\!\}. \end{aligned}$$

The operations satisfy many more laws, but we will omit them here.

2.4 Type-indexed tries

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

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

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* $\bot_{\mathsf{v}}$, see [11]. A finite map is then a function whose value is $\bot_{\mathsf{v}}$ 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, our first example of a type-indexed data type.

$$\begin{aligned} &\mathsf{FMap}\{\!|\mathsf{t} :: \star|\!\} && :: \star \to \star \\ &\mathsf{FMap}\{\!|\mathsf{Unit}|\!\}\ \mathsf{v} && = \mathsf{Maybe}\ \mathsf{v} \\ &\mathsf{FMap}\{\!|\mathsf{Int}|\!\}\ \mathsf{v} && = Patricia.\mathsf{Dict}\ \mathsf{v} \\ &\mathsf{FMap}\{\!|\mathsf{Char}|\!\}\ \mathsf{v} && = \mathsf{FMapChar}\ \mathsf{v} \\ &\mathsf{FMap}\{\!|\mathsf{t}_1 :+: \mathsf{t}_2|\!\}\ \mathsf{v} && = \mathsf{FMap}\{\!|\mathsf{t}_1|\!\}\ \mathsf{v} \times_\bullet \mathsf{FMap}\{\!|\mathsf{t}_2|\!\}\ \mathsf{v} \\ &\mathsf{FMap}\{\!|\mathsf{t}_1 :*: \mathsf{t}_2|\!\}\ \mathsf{v} && = \mathsf{FMap}\{\!|\mathsf{t}_1|\!\}\ (\mathsf{FMap}\{\!|\mathsf{t}_2|\!\}\ \mathsf{v}) \\ &\mathsf{FMap}\{\!|\mathsf{Con}\ \mathsf{t}|\!\}\ \mathsf{v} && = \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \end{aligned}$$

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

$$\mathbf{data}\ \mathsf{a} \times_\bullet \mathsf{b} = Null \mid Pair\ \mathsf{a}\ \mathsf{b}$$

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

$$\mathsf{FMap}\{\!|\mathsf{t}_1 :+: \mathsf{t}_2|\!\}\ \mathsf{v} = \mathsf{FMap}\{\!|\mathsf{t}_1|\!\}\ \mathsf{v} :*: \mathsf{FMap}\{\!|\mathsf{t}_2|\!\}\ \mathsf{v}.$$

This representation has, however, two major drawbacks: (i) it relies in an essential way on lazy evaluation and (ii) it is inefficient, see [18].

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* [44]. 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*.

A few remarks are in order. FMap is a type-indexed data type [23]. The only way to construct values of type FMap is by means of the functions in the interface.

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 a value, so the constructor description can never be used in the definition of a type-indexed data type.

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, Unit $\cong$ Unit :∗: Unit (ignoring laziness) but FMap{|Unit|} v = Maybe v $\ncong$ Maybe (Maybe v) = FMap{|Unit :∗: Unit|} 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, Maybe v $\times_\bullet$ 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 kind of FMap for types of kind $\star$ is $\star \to \star$. For type constructors with higher-order kinds, the kind 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:

$$\mathsf{FMap}\{|\mathsf{f} :: \star \to \star|\} :: (\star \to \star) \to (\star \to \star).$$

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

$$\begin{aligned}
&\mathsf{FMap}\{|\mathsf{f} :: \kappa|\} &&:: \mathsf{FMAP}\{[\kappa]\}\ \mathsf{f}\\
&\mathsf{FMAP}\{[\kappa :: \Box]\} &&:: \Box\\
&\mathsf{FMAP}\{[\star]\} &&= \star \to \star\\
&\mathsf{FMAP}\{[\kappa \to \nu]\} &&= \mathsf{FMAP}\{[\kappa]\} \to \mathsf{FMAP}\{[\nu]\},
\end{aligned}$$

where the box $\Box$ is the type of a kind, a so-called *super-kind*.

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

```
data List a   = Nil | Cons a (List a)
data Tree a b = Tip a | Node (Tree a b) b (Tree a b)
data Fork a   = Fork a a
data Sequ a   = EndS | ZeroS (Sequ (Fork a)) | OneS a (Sequ (Fork a))
```

These types are represented by (see also section 3.1 in the first part of these lecture notes [22]):

```
List  = Fix (ΛList . Λa . Unit :+: a :*: List a)
Tree  = Fix (ΛTree . Λa b . a :+: Tree a b :*: b :*: Tree a b)
Fork  = Λa . a :*: a
Sequ  = Fix (ΛSequ . Λa . Unit :+: Sequ (Fork a) :+: a :*: Sequ (Fork a)).
```

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

```
FMap_List  = Fix (ΛFMap_List . Λfa . Maybe ×• fa · FMap_List fa)
FMap_Tree  = Fix (ΛFMap_Tree . Λfa fb .
                    fa ×•
                    FMap_Tree fa fb · fb · FMap_Tree fa fb)
FMap_Fork  = Λfa . fa · fa
FMap_Sequ  = Fix (ΛFMap_Sequ . Λfa .
                    Maybe ×•
                    FMap_Sequ (FMap_Fork fa) ×•
                    fa · FMap_Sequ (FMap_Fork fa)).
```

As an aside, note that we interpret `a ×• b ×• c` as the type of optional triples and not as nested optional pairs:

$$\textbf{data}\ \mathsf{a} \times_\bullet \mathsf{b} \times_\bullet \mathsf{c} = \mathit{Null} \mid \mathit{Triple}\ \mathsf{a}\ \mathsf{b}\ \mathsf{c}.$$

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.

```
data FMap_List fa v      = Null_List
                         | Node_List (Maybe v)
                                     (fa (FMap_List fa v))
data FMap_Tree fa fb v = Null_Tree
                         | Node_Tree (fa v)
                                     (FMap_Tree fa fb
                                           (fb (FMap_Tree fa fb v)))
```

These types are the parametric variants of FMap_String and FMap_Dict defined in Section 2.1: we have FMap_String ≈ FMap_List FMapChar (corresponding to String ≈ List Char) and FMap_Dict ≈ FMap_Tree FMap_String FMap_String (corresponding to Dict ≈ Tree String String). Things become interesting if we consider nested data types.

```
data FMap_Fork fa v  = Node_Fork (fa (fa v))
data FMap_Sequ fa v = Null_Sequ
                    | Node_Sequ (Maybe v)
                                (FMap_Sequ (FMap_Fork fa) v)
                                (fa (FMap_Sequ (FMap_Fork fa) v))
```

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 FMap_Sequ is a second-order nest since the parameter fa of kind $\star \to \star$ is changed in the recursive calls. By contrast, FMap_Tree 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 FMap_Sequ has type FMap_Sequ (FMap_Fork fa) (fa v) and since both fa and v are instantiated, FMap_Sequ is consequently both a first- and a second-order nest.

2.5 Empty tries

The empty trie is defined as follows.

$$
\begin{array}{lll}
\textbf{type}\ \mathsf{Empty}\{\![\star]\!\}\ \mathsf{t} & = \forall \mathsf{v}\,.\,\mathsf{FMap}\{\![\mathsf{t}]\!\}\ \mathsf{v} \\
\textbf{type}\ \mathsf{Empty}\{\![\kappa \to \nu]\!\}\ \mathsf{t} & = \forall \mathsf{a}\,.\,\mathsf{Empty}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Empty}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a}) \\
\mathit{empty}\{\![\mathsf{t} :: \kappa]\!\} & :: & \mathsf{Empty}\{\![\kappa]\!\}\ \mathsf{t} \\
\mathit{empty}\{\![\mathsf{Unit}]\!\} & = & \mathit{Nothing} \\
\mathit{empty}\{\![\mathsf{Char}]\!\} & = & [\,] \\
\mathit{empty}\{\![\mathsf{Int}]\!\} & = & \mathit{Patricia.empty} \\
\mathit{empty}\{\![\mathsf{a} :+: \mathsf{b}]\!\} & = & \mathit{Null} \\
\mathit{empty}\{\![\mathsf{a} :*: \mathsf{b}]\!\} & = & \mathit{empty}\{\![\mathsf{a}]\!\} \\
\mathit{empty}\{\![\mathsf{Con}\ c\ \mathsf{a}]\!\} & = & \mathit{empty}\{\![\mathsf{a}]\!\}
\end{array}
$$

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 $\mathit{empty}\{\![\mathsf{a} :*: \mathsf{b}]\!\}$, which is of type $\forall \mathsf{v}\,.\,\mathsf{FMap}\{\![\mathsf{a}]\!\}\ (\mathsf{FMap}\{\![\mathsf{b}]\!\}\ \mathsf{v})$. It is defined in terms of $\mathit{empty}\{\![\mathsf{a}]\!\}$, which is of type $\forall \mathsf{v}\,.\,\mathsf{FMap}\{\![\mathsf{a}]\!\}\ \mathsf{v}$. That means that $\mathit{empty}\{\![\mathsf{a}]\!\}$ is used polymorphically. In other words, *empty* makes use of polymorphic recursion!

Suppose we want to define a function *emptyI* that is almost the same as the function *empty*, but uses a different value, say *emptyIntTrie*, for the empty trie for integers. The definition of FMap says that *emptyIntTrie* has to be a *Patricia* tree, but that might be changed in the definition of FMap. Then we can use a default case [10] to define *emptyI* in terms of *empty* as follows:

$$
\begin{array}{lll}
\mathit{emptyI}\{\![\mathsf{t} :: \kappa]\!\} & :: \mathit{Empty}\ (\kappa)\ \mathsf{t} \\
\mathit{emptyI}\{\![\mathsf{Int}]\!\} & = \mathit{emptyIntTrie} \\
\mathit{emptyI}\{\![\mathsf{a}]\!\} & = \mathit{empty}\{\![\mathsf{a}]\!\}.
\end{array}
$$

So the function *emptyI* is equal to the function *empty* in all cases except for the Int case, where it uses a special empty trie.

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

$$
\begin{aligned}
&\mathit{empty\_List} :: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, \mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{v} \,.\, \mathsf{FMap\_List}\ \mathsf{fa}\ \mathsf{v}) \\
&\mathit{empty\_List}\ \mathit{ea} = \mathit{Null\_List} \\
&\mathit{empty\_Fork} :: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, \mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{v} \,.\, \mathsf{FMap\_Fork}\ \mathsf{fa}\ \mathsf{v}) \\
&\mathit{empty\_Fork}\ \mathit{ea} = \mathit{Node\_Fork}\ \mathit{ea} \\
&\mathit{empty\_Sequ} :: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, \mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{v} \,.\, \mathsf{FMap\_Sequ}\ \mathsf{fa}\ \mathsf{v}) \\
&\mathit{empty\_Sequ}\ \mathit{ea} = \mathit{Null\_Sequ}
\end{aligned}
$$

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

The function $\mathit{isempty}\{\!|t|\!\}$ takes a trie and determines whether it is empty.

$$
\begin{aligned}
&\mathbf{type}\ \mathsf{IsEmpty}\{\!|\star|\!\}\ \mathsf{t} &&= \forall \mathsf{v} \,.\, \mathsf{FMap}\{\!|t|\!\}\ \mathsf{v} \to \mathsf{Bool} \\
&\mathbf{type}\ \mathsf{IsEmpty}\{\!|\kappa \to \nu|\!\}\ \mathsf{t} &&= \forall \mathsf{a} \,.\, \mathsf{IsEmpty}\{\!|\kappa|\!\}\ \mathsf{a} \to \mathsf{IsEmpty}\{\!|\nu|\!\}\ (\mathsf{t}\ \mathsf{a})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{isempty}\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{IsEmpty}\{\!|\kappa|\!\}\ \mathsf{t} \\
&\mathit{isempty}\{\!|\mathsf{Unit}|\!\}\ v &&= \mathit{isNothing}\ v \\
&\mathit{isempty}\{\!|\mathsf{Char}|\!\}\ m &&= \mathit{null}\ m \\
&\mathit{isempty}\{\!|\mathsf{Int}|\!\}\ m &&= \mathit{Patricia.isempty}\ m \\
&\mathit{isempty}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ \mathit{Null} &&= \mathit{True} \\
&\mathit{isempty}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ d &&= \mathit{False} \\
&\mathit{isempty}\{\!|\mathsf{a} :*: \mathsf{b}|\!\}\ d &&= \mathit{isempty}\{\!|\mathsf{a}|\!\}\ d \\
&\mathit{isempty}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ d &&= \mathit{isempty}\{\!|\mathsf{a}|\!\}\ d
\end{aligned}
$$

Function *isempty* assumes that tries are in 'normal form', so the empty trie is always represented by *Null*, and not, for example, by *Pair Null Null*.

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

$$
\begin{aligned}
&\mathit{isempty\_List} &&:: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, \mathsf{fa}\ \mathsf{w} \to \mathsf{Bool}) \to \\
&&&\qquad (\forall \mathsf{v} \,.\, \mathsf{FMap\_List}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{Bool}) \\
&\mathit{isempty\_List}\ \mathit{iea}\ \mathit{Null\_List} &&= \mathit{True} \\
&\mathit{isempty\_List}\ \mathit{iea}\ (\mathit{Node\_List}\ \mathit{tn}\ \mathit{tc}) &&= \mathit{False} \\
&\mathit{isempty\_Fork} &&:: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, \mathsf{fa}\ \mathsf{w} \to \mathsf{Bool}) \to \\
&&&\qquad (\forall \mathsf{v} \,.\, \mathsf{FMap\_Fork}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{Bool}) \\
&\mathit{isempty\_Fork}\ \mathit{iea}\ (\mathit{Node\_Fork}\ \mathit{tf}) &&= \mathit{iea}\ \mathit{tf} \\
&\mathit{isempty\_Sequ} &&:: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, \mathsf{fa}\ \mathsf{w} \to \mathsf{Bool}) \to \\
&&&\qquad (\forall \mathsf{v} \,.\, \mathsf{FMap\_Sequ}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{Bool}) \\
&\mathit{isempty\_Sequ}\ \mathit{iea}\ \mathit{Null\_Sequ} &&= \mathit{True} \\
&\mathit{isempty\_Sequ}\ \mathit{iea}\ (\mathit{Node\_Sequ}\ \mathit{tv}\ \mathit{tf}\ \mathit{ts}) &&= \mathit{False}
\end{aligned}
$$

2.6 Singleton tries

The function $single\{\!|\mathsf{t}|\!\}\ (t, v)$ 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 means that *single depends on empty*. Generic Haskell supports dependencies, so we can use both the empty trie and the single trie in the sum, product, and constructor cases of function *single*. The dependency shows in the type of the function *single*: on higher-order kinds, the type mentions the type of *empty*.

$$\begin{aligned}&\textbf{type}\ \mathsf{Single}\{\!|\star|\!\}\ \mathsf{t} &&= \forall \mathsf{v}\,.\,(\mathsf{t},\mathsf{v}) \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v}\\ &\textbf{type}\ \mathsf{Single}\{\!|\kappa \to \nu|\!\}\ \mathsf{t} &&= \forall \mathsf{a}\,.\,\mathsf{Empty}\{\!|\kappa|\!\}\ \mathsf{a} \to \mathsf{Single}\{\!|\kappa|\!\}\ \mathsf{a} \to \mathsf{Single}\{\!|\nu|\!\}\ (\mathsf{t}\ \mathsf{a})\end{aligned}$$

Plain generic functions can be seen as catamorphisms [38, 42] over the structure of data types. With dependencies, we also get the power of paramorphisms [40].

$$\begin{aligned}&single\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{Single}\{\!|\kappa|\!\}\ \mathsf{t}\\ &single\{\!|\mathsf{Unit}|\!\}\ (Unit, v) &&= Just\ v\\ &single\{\!|\mathsf{Char}|\!\}\ (c, v) &&= [(c, v)]\\ &single\{\!|\mathsf{Int}|\!\}\ (i, v) &&= Patricia.single\ (i, v)\\ &single\{\!|\mathsf{a}\ \mathsf{:+:}\ \mathsf{b}|\!\}\ (Inl\ a, v) &&= Pair\ (single\{\!|\mathsf{a}|\!\}\ (a, v))\ (empty\{\!|b|\!\})\\ &single\{\!|\mathsf{a}\ \mathsf{:+:}\ \mathsf{b}|\!\}\ (Inr\ b, v) &&= Pair\ (empty\{\!|\mathsf{a}|\!\})\ (single\{\!|\mathsf{b}|\!\}\ (b, v))\\ &single\{\!|\mathsf{a}\ \mathsf{:*:}\ \mathsf{b}|\!\}\ (a\ \mathsf{:*:}\ b, v) &&= single\{\!|\mathsf{a}|\!\}\ (a, single\{\!|\mathsf{b}|\!\}\ (b, v))\\ &single\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (Con\ b, v) &&= single\{\!|\mathsf{a}|\!\}\ (b, v)\end{aligned}$$

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

$$\begin{aligned}&single\_List :: \forall \mathsf{k}\ \mathsf{fa}\,.\,(\forall \mathsf{w}\,.\,\mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{w}\,.\,(\mathsf{k},\mathsf{w}) \to \mathsf{fa}\ \mathsf{w})\\ &\qquad\qquad \to (\forall \mathsf{v}\,.\,(\mathsf{List}\ \mathsf{k},\mathsf{v}) \to \mathsf{FMap\_List}\ \mathsf{fa}\ \mathsf{v})\\ &single\_List\ ea\ sa\ (Nil, v) = Node\_List\ (Just\ v)\ ea\\ &single\_List\ ea\ sa\ (Cons\ k\ ks, v) =\\ &\quad Node\_List\ Nothing\ (sa\ (k, single\_List\ ea\ sa\ (ks, v)))\\ &single\_Fork :: \forall \mathsf{k}\ \mathsf{fa}\,.\,(\forall \mathsf{w}\,.\,\mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{w}\,.\,(\mathsf{k},\mathsf{w}) \to \mathsf{fa}\ \mathsf{w})\\ &\qquad\qquad \to (\forall \mathsf{v}\,.\,(\mathsf{Fork}\ \mathsf{k},\mathsf{v}) \to \mathsf{FMap\_Fork}\ \mathsf{fa}\ \mathsf{v})\\ &single\_Fork\ ea\ sa\ (Fork\ k_1\ k_2, v) = Node\_Fork\ (sa\ (k_1, sa\ (k_2, v)))\\ &single\_Sequ :: \forall \mathsf{k}\ \mathsf{fa}\,.\,(\forall \mathsf{w}\,.\,\mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{w}\,.\,(\mathsf{k},\mathsf{w}) \to \mathsf{fa}\ \mathsf{w})\\ &\qquad\qquad \to (\forall \mathsf{v}\,.\,(\mathsf{Sequ}\ \mathsf{k},\mathsf{v}) \to \mathsf{FMap\_Sequ}\ \mathsf{fa}\ \mathsf{v})\\ &single\_Sequ\ ea\ sa\ (EndS, v) = Node\_Sequ\ (Just\ v)\ Null\_Sequ\ ea\\ &single\_Sequ\ ea\ sa\ (ZeroS\ s, v) =\\ &\quad Node\_Sequ\ Nothing\\ &\qquad\qquad (single\_Sequ\ (empty\_Fork\ ea)\ (single\_Fork\ ea\ sa)\ (s, v))\\ &\qquad\qquad ea\\ &single\_Sequ\ ea\ sa\ (OneS\ k\ s, v) =\\ &\quad Node\_Sequ\ Nothing\\ &\qquad\qquad Null\_Sequ\\ &\qquad\qquad (sa\ (k, single\_Sequ\ (empty\_Fork\ ea)\ (single\_Fork\ ea\ sa)\ (s, v)))\end{aligned}$$

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

2.7 Look up

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

$$\begin{aligned}
&\textbf{type}\ \mathsf{Lookup}\{\![\star]\!\}\ \mathsf{t} &&= \forall \mathsf{v}\,.\,\mathsf{t} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}\\
&\textbf{type}\ \mathsf{Lookup}\{\![\kappa \to \nu]\!\}\ \mathsf{t} &&= \forall \mathsf{a}\,.\,\mathsf{Lookup}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Lookup}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})
\end{aligned}$$

$$\begin{aligned}
&\mathit{lookup}\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{Lookup}\{\![\kappa]\!\}\ \mathsf{t}\\
&\mathit{lookup}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit}\ \mathit{fm} &&= \mathit{fm}\\
&\mathit{lookup}\{\!|\mathsf{Char}|\!\}\ c\ \mathit{fm} &&= \mathit{lookupChar}\ c\ \mathit{fm}\\
&\mathit{lookup}\{\!|\mathsf{Int}|\!\}\ i\ \mathit{fm} &&= \mathit{Patricia.lookup}\ i\ \mathit{fm}\\
&\mathit{lookup}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ t\ \mathit{Null} &&= \mathit{Nothing}\\
&\mathit{lookup}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inl}\ a)\ (\mathit{Pair}\ \mathit{fma}\ \mathit{fmb}) &&= \mathit{lookup}\{\!|\mathsf{a}|\!\}\ a\ \mathit{fma}\\
&\mathit{lookup}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inr}\ b)\ (\mathit{Pair}\ \mathit{fma}\ \mathit{fmb}) &&= \mathit{lookup}\{\!|\mathsf{b}|\!\}\ b\ \mathit{fmb}\\
&\mathit{lookup}\{\!|\mathsf{a} :*: \mathsf{b}|\!\}\ (a :*: b)\ \mathit{fma} &&= (\mathit{lookup}\{\!|\mathsf{a}|\!\}\ a \diamond \mathit{lookup}\{\!|\mathsf{b}|\!\}\ b)\ \mathit{fma}\\
&\mathit{lookup}\{\!|\mathsf{Con}\ d\ \mathsf{a}|\!\}\ (\mathit{Con}\ b)\ \mathit{fm} &&= \mathit{lookup}\{\!|\mathsf{a}|\!\}\ b\ \mathit{fm}
\end{aligned}$$

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 monadic composition.

Example 5. Specializing $\mathit{lookup}\{\!|\mathsf{t}|\!\}$ to concrete instances of t is by now probably a matter of routine.

$$\begin{aligned}
&\mathit{lookup_List} :: \forall \mathsf{k}\ \mathsf{fa}\,.\,(\forall \mathsf{w}\,.\,\mathsf{k} \to \mathsf{fa}\ \mathsf{w} \to \mathsf{Maybe}\ \mathsf{w})\\
&\qquad\qquad \to (\forall \mathsf{v}\,.\,\mathsf{List}\ \mathsf{k} \to \mathsf{FMap_List}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\mathit{lookup_List}\ \mathit{la}\ \mathit{ks}\ \mathit{Null_List} &&= \mathit{Nothing}\\
&\mathit{lookup_List}\ \mathit{la}\ \mathit{Nil}\ (\mathit{Node_List}\ \mathit{tn}\ \mathit{tc}) &&= \mathit{tn}\\
&\mathit{lookup_List}\ \mathit{la}\ (\mathit{Cons}\ k\ \mathit{ks})\ (\mathit{Node_List}\ \mathit{tn}\ \mathit{tc}) &&= (\mathit{la}\ k \diamond \mathit{lookup_List}\ \mathit{la}\ \mathit{ks})\ \mathit{tc}\\
&\mathit{lookup_Fork} :: \forall \mathsf{k}\ \mathsf{fa}\,.\,(\forall \mathsf{w}\,.\,\mathsf{k} \to \mathsf{fa}\ \mathsf{w} \to \mathsf{Maybe}\ \mathsf{w})\\
&\qquad\qquad \to (\forall \mathsf{v}\,.\,\mathsf{Fork}\ \mathsf{k} \to \mathsf{FMap_Fork}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\mathit{lookup_Fork}\ \mathit{la}\ (\mathit{Fork}\ k_1\ k_2)\ (\mathit{Node_Fork}\ \mathit{tf}) &&= (\mathit{la}\ k_1 \diamond \mathit{la}\ k_2)\ \mathit{tf}\\
&\mathit{lookup_Sequ} :: \forall \mathsf{fa}\ \mathsf{k}\,.\,(\forall \mathsf{w}\,.\,\mathsf{k} \to \mathsf{fa}\ \mathsf{w} \to \mathsf{Maybe}\ \mathsf{w})\\
&\qquad\qquad \to (\forall \mathsf{v}\,.\,\mathsf{Sequ}\ \mathsf{k} \to \mathsf{FMap_Sequ}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v})\\
&\mathit{lookup_Sequ}\ \mathit{la}\ s\ \mathit{Null_Sequ} &&= \mathit{Nothing}\\
&\mathit{lookup_Sequ}\ \mathit{la}\ \mathit{EndS}\ (\mathit{Node_Sequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) &&= \mathit{te}\\
&\mathit{lookup_Sequ}\ \mathit{la}\ (\mathit{ZeroS}\ s)\ (\mathit{Node_Sequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) &&=\\
&\quad \mathit{lookup_Sequ}\ (\mathit{lookup_Fork}\ \mathit{la})\ s\ \mathit{tz}\\
&\mathit{lookup_Sequ}\ \mathit{la}\ (\mathit{OneS}\ a\ s)\ (\mathit{Node_Sequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) =\\
&\quad (\mathit{la}\ a \diamond \mathit{lookup_Sequ}\ (\mathit{lookup_Fork}\ \mathit{la})\ s)\ \mathit{to}
\end{aligned}$$

The function *lookup_List* generalizes *lookup_String* defined in Section 2.1; we have $\mathit{lookup\_String} \approx \mathit{lookup\_List}\ \mathit{lookupChar}$.

2.8 Inserting and merging

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

$$
\begin{aligned}
&\mathit{insert}\{\!|\mathsf{t} :: \star|\!\} \quad :: \forall \mathsf{v} \,.\, (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \to (\mathsf{t}, \mathsf{v}) \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \\
&\mathit{insert}\{\!|\mathsf{t}|\!\}\ c\ (x, v)\ d = \mathit{merge}\{\!|\mathsf{t}|\!\}\ c\ (\mathit{single}\{\!|\mathsf{t}|\!\}\ (x, v))\ d
\end{aligned}
$$

Function *insert* is defined as a *generic abstraction*. A generic abstraction is a generic function that is defined in terms of another generic function. 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 generic function that works for type constructors of all kinds.

Merging two tries is surprisingly simple. Given the function *combine* defined in section 2.3, and a function for merging two association lists

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

we can define *merge* as follows.

$$
\begin{aligned}
&\mathbf{type}\ \mathsf{Merge}\{\![\star]\!\}\ \mathsf{t} = \forall \mathsf{v} \,. \\
&\quad (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \\
&\mathbf{type}\ \mathsf{Merge}\{\![\kappa \to \nu]\!\}\ \mathsf{t} = \forall \mathsf{a} \,.\, \mathsf{Merge}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Merge}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{merge}\{\!|\mathsf{t} :: \kappa|\!\} \quad :: \mathsf{Merge}\{\![\kappa]\!\}\ \mathsf{t} \\
&\mathit{merge}\{\!|\mathsf{Unit}|\!\}\ c\ v\ v' \quad = \mathit{combine}\ c\ v\ v' \\
&\mathit{merge}\{\!|\mathsf{Char}|\!\}\ c\ \mathit{fm}\ \mathit{fm}' = \mathit{mergeChar}\ c\ \mathit{fm}'\ \mathit{fm} \\
&\mathit{merge}\{\!|\mathsf{Int}|\!\}\ c\ \mathit{fm}\ \mathit{fm}' \quad = \mathit{Patricia.merge}\ c\ \mathit{fm}'\ \mathit{fm} \\
&\mathit{merge}\{\!|\mathsf{Con}\ d\ \mathsf{a}|\!\}\ c\ e\ e' = \mathit{merge}\{\!|\mathsf{a}|\!\}\ c\ e\ e'
\end{aligned}
$$

For the sum case, we have to distinguish between empty and nonempty tries:

$$
\begin{aligned}
&\mathit{merge}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ c\ d\ \mathit{Null} \quad\quad\quad\quad\quad\quad = d \\
&\mathit{merge}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ c\ \mathit{Null}\ d \quad\quad\quad\quad\quad\quad = d \\
&\mathit{merge}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ c\ (\mathit{Pair}\ x\ y)\ (\mathit{Pair}\ v\ w) = \\
&\quad \mathit{Pair}\ (\mathit{merge}\{\!|\mathsf{a}|\!\}\ c\ x\ v)\ (\mathit{merge}\{\!|\mathsf{b}|\!\}\ c\ y\ w).
\end{aligned}
$$

The most interesting equation is the product case. The tries d and d' are of type $\mathsf{FMap}\{\!|\mathsf{a}|\!\}\ (\mathsf{FMap}\{\!|\mathsf{b}|\!\}\ \mathsf{v})$. To merge them we can recursively call $\mathit{merge}\{\!|\mathsf{a}|\!\}$; we must, however, supply a combining function of type $\forall \mathsf{v} \,.\, \mathsf{FMap}\{\!|\mathsf{b}|\!\}\ \mathsf{v} \to$

$\mathsf{FMap}\{\!|\mathsf{b}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{b}|\!\}\ \mathsf{v}$. A moment's reflection reveals that $merge\{\!|\mathsf{b}|\!\}\ c$ is the desired combining function.

$$merge\{\!|\mathsf{a}\ \mathtt{:*:}\ \mathsf{b}|\!\}\ c\ d\ d' = merge\{\!|\mathsf{a}|\!\}\ (merge\{\!|\mathsf{b}|\!\}\ c)\ d\ d'$$

The definition of *merge* shows that it is sometimes necessary to implement operations more general than immediately needed. If $\mathsf{Merge}\{\!|\star|\!\}\ \mathsf{t}$ had been the simpler type $\forall \mathsf{v} \,.\, \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v}$, 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).

$$\begin{aligned}
&merge_List :: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, (\mathsf{w} \to \mathsf{w} \to \mathsf{w}) \to (\mathsf{fa}\ \mathsf{w} \to \mathsf{fa}\ \mathsf{w} \to \mathsf{fa}\ \mathsf{w})) \\
&\qquad \to (\forall \mathsf{v} \,.\, (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \\
&\qquad\quad \to \mathsf{FMap_List}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{FMap_List}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{FMap_List}\ \mathsf{fa}\ \mathsf{v}) \\
&merge_List\ ma\ c\ Null_List\ t = t \\
&merge_List\ ma\ c\ t\ Null_List = t \\
&merge_List\ ma\ c\ (Node_List\ tn\ tc)\ (Node_List\ tn'\ tc') = \\
&\quad Node_List\ (combine\ c\ tn\ tn') \\
&\qquad (ma\ (merge_List\ ma\ c)\ tc\ tc')
\end{aligned}$$

$$\begin{aligned}
&merge_Fork :: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, (\mathsf{w} \to \mathsf{w} \to \mathsf{w}) \to (\mathsf{fa}\ \mathsf{w} \to \mathsf{fa}\ \mathsf{w} \to \mathsf{fa}\ \mathsf{w})) \\
&\qquad \to (\forall \mathsf{v} \,.\, (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \\
&\qquad\quad \to \mathsf{FMap_Fork}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{FMap_Fork}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{FMap_Fork}\ \mathsf{fa}\ \mathsf{v}) \\
&merge_Fork\ ma\ c\ (Node_Fork\ tf)\ (Node_Fork\ tf') = \\
&\quad Node_Fork\ (ma\ (ma\ c)\ tf\ tf')
\end{aligned}$$

$$\begin{aligned}
&merge_Sequ :: \forall \mathsf{fa} \,.\, (\forall \mathsf{w} \,.\, (\mathsf{w} \to \mathsf{w} \to \mathsf{w}) \to (\mathsf{fa}\ \mathsf{w} \to \mathsf{fa}\ \mathsf{w} \to \mathsf{fa}\ \mathsf{w})) \\
&\qquad \to (\forall \mathsf{v} \,.\, (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \\
&\qquad\quad \to \mathsf{FMap_Sequ}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{FMap_Sequ}\ \mathsf{fa}\ \mathsf{v} \to \mathsf{FMap_Sequ}\ \mathsf{fa}\ \mathsf{v}) \\
&merge_Sequ\ ma\ c\ Null_Sequ\ t = t \\
&merge_Sequ\ ma\ c\ t\ Null_Sequ = t \\
&merge_Sequ\ ma\ c\ (Node_Sequ\ te\ tz\ to)\ (Node_Sequ\ te'\ tz'\ to') = \\
&\quad Node_Sequ\ (combine\ c\ te\ te') \\
&\qquad (merge_Sequ\ (merge_Fork\ ma)\ c\ tz\ tz') \\
&\qquad (ma\ (merge_Sequ\ (merge_Fork\ ma)\ c)\ to\ to')
\end{aligned}$$

2.9 Deleting

Function $delete\{\!|\mathsf{t}|\!\}$ 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:

$$deleteChar :: \forall \mathsf{v} \,.\, \mathsf{Char} \to \mathsf{FMapChar}\ \mathsf{v} \to \mathsf{FMapChar}\ \mathsf{v},$$

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

$$
\begin{array}{ll}
\mathit{delete}\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Delete}\{\![\kappa]\!\}\ \mathsf{t} \\
\mathit{delete}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit}\ \mathit{fm} & = \mathit{Nothing} \\
\mathit{delete}\{\!|\mathsf{Char}|\!\}\ c\ \mathit{fm} & = \mathit{deleteChar}\ c\ \mathit{fm} \\
\mathit{delete}\{\!|\mathsf{Int}|\!\}\ i\ \mathit{fm} & = \mathit{Patricia.delete}\ i\ \mathit{fm}.
\end{array}
$$

All cases except the product case are straightforward. In the product case, we have to remove a binding for a product $(a\ \mathsf{:*:}\ b)$. 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 pass as a combining function $\lambda x\ y \to x$, which overwrites existing bindings in a trie. From this description it follows that the function *delete* depends on the functions *lookup*, *insert* (which depends on function *empty*), and *isempty*. Here we need the kind-indexed typed version of function *insert*, as defined in the exercise at the end of this section.

$$
\begin{array}{ll}
\textbf{type}\ \mathsf{Delete}\{\![\star]\!\}\ \mathsf{t} & = \forall \mathsf{v}\,.\,\mathsf{t} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \\
\textbf{type}\ \mathsf{Delete}\{\![\kappa \to \nu]\!\}\ \mathsf{t} = & \forall \mathsf{a}\,.\,\mathsf{Lookup}\{\![\kappa]\!\}\ \mathsf{a} \\
& \quad \to \mathsf{Insert}\{\![\kappa]\!\}\ \mathsf{a} \\
& \quad \to \mathsf{IsEmpty}\{\![\kappa]\!\}\ \mathsf{a} \\
& \quad \to \mathsf{Empty}\{\![\kappa]\!\}\ \mathsf{a} \\
& \quad \to \mathsf{Delete}\{\![\kappa]\!\}\ \mathsf{a} \\
& \quad \to \mathsf{Delete}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{delete}\{\!|\mathsf{a}\ \mathsf{:+:}\ \mathsf{b}|\!\}\ t\ \mathit{Null} & = \mathit{Null} \\
\mathit{delete}\{\!|\mathsf{a}\ \mathsf{:+:}\ \mathsf{b}|\!\}\ (\mathit{Inl}\ a)\ (\mathit{Pair}\ x\ y) & = \mathit{Pair}\ (\mathit{delete}\{\!|\mathsf{a}|\!\}\ a\ x)\ y \\
\mathit{delete}\{\!|\mathsf{a}\ \mathsf{:+:}\ \mathsf{b}|\!\}\ (\mathit{Inr}\ b)\ (\mathit{Pair}\ x\ y) & = \mathit{Pair}\ x\ (\mathit{delete}\{\!|\mathsf{b}|\!\}\ b\ y) \\
\mathit{delete}\{\!|\mathsf{a}\ \mathsf{:*:}\ \mathsf{b}|\!\}\ (a\ \mathsf{:*:}\ b)\ d & = \\
\quad \textbf{case}\ (\mathit{lookup}\{\!|\mathsf{a}|\!\}\ a \diamond \mathit{delete}\{\!|\mathsf{b}|\!\}\ b)\ d\ \textbf{of} & \\
\qquad \mathit{Nothing} & \to d \\
\qquad \mathit{Just}\ d' \mid \mathit{isempty}\{\!|\mathsf{b}|\!\}\ d' & \to \mathit{delete}\{\!|\mathsf{a}|\!\}\ a\ d \\
\qquad\quad \mid \mathit{otherwise} & \to \mathit{insert}\{\!|\mathsf{a}|\!\}\ (\lambda x\ y \to x)\ (a, d')\ d \\
\mathit{delete}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ b)\ d & = \mathit{delete}\{\!|\mathsf{a}|\!\}\ b\ d
\end{array}
$$

Function *delete* should also maintain the invariant that the empty trie is represented by *Null*, and not by *Pair Null Null*, for example. It is easy to adapt the above definition such that this invariant is maintained.

Since the type of *delete* is rather complex because of the dependencies, we only give the instance of *delete* on List.

```
delete_List :: ∀k fa . (∀w . k → fa w → Maybe w)
                     → (∀w . (w → w → w) → (k, w) → fa w → fa w)
                     → (∀w . fa w → Bool)
                     → (∀w . fa w)
                     → (∀w . (k → fa w → fa w))
                     → (∀v . List k → FMap_List fa v → FMap_List fa v)
delete_List la ia iea ea da Nil Null_List                 = Null_List
delete_List la ia iea ea da Nil (Node_List tn tc)         = Node_List Nothing tc
delete_List la ia iea ea da (Cons a as) Null_List         = Null_List
delete_List la ia iea ea da (Cons a as) (Node_List tn tc) =
  case (la a ◇ delete_List la ia iea ea da as) tc of
    Nothing → tc
    Just tc' | iea tc' → da a tc
             | otherwise → ia (λx y → x) (a, tc') tc
```

2.10 Related work

Knuth [33] attributes the idea of a trie to Thue who introduced it in a paper about strings that do not contain adjacent repeated substrings [47]. 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 [50] and others independently since. Connelly et al. [11] 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 [43]. 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.

Exercise 1. Define function *insert* as a generic 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.

Exercise 2. Define a function *update*, which updates a binding in a trie.

$$update\{|t|\} :: \forall v\,.\,(t, v) \to FMap\{|t|\}\ v \to Maybe\ (FMap\{|t|\}\ v)$$

If there is no binding for the value of type `t` in trie of type FMap{|t|} v, *update* returns *Nothing*.

3 XComprez: a generic XML compressor

The extensible markup language XML is a popular standard for describing documents with markup (or structure). XML documents may become (very) large because of the markup that is added to the content. A lot of diskspace and bandwidth is used to store and send XML documents. XML compressors reduce the size of an XML document, sometimes by a considerable factor. This section describes a generic XML compressor based on the ideas described in the context of data conversion by Jansson and Jeuring [26, 30].

This section shows how an XML compressor is implemented as a generic program, and it briefly discusses which other classes of XML tools would profit from an implementation as a generic program. The example shows how Generic Haskell can be used to implement XML tools whose behaviour depends on the DTD or Schema of the input XML document. Example tools include XML editors, databases, and compressors.

Generic Haskell is ideally suited for implementing XML tools:

- Knowledge of the DTD can be used to provide more precise functionality, such as manipulations of an XML document that preserve validity in an XML editor, or better compression in an XML compressor.
- Generic Haskell programs are typed. Consequently, valid documents are transformed to valid documents, possibly structured according to another DTD. Thus Generic Haskell supports constructing type correct XML tools.
- The generic features of Generic Haskell make XML tools easier to implement in a surprisingly small amount of code.
- The Generic Haskell compiler may perform all kinds of advanced optimisations on the code, such as partial evaluation or deforestation, which are difficult to conceive or implement by an XML tool developer.

3.1 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 the Unix compress utility [52]. 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 a 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 very similar to the compressor, we omit its description. See the website for XCOMPREZ [32] 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 [51], 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. We have not introduced the Label construct in these lecture notes. It it used to represent record labels in data types, and is very similar to the Con construct.

Separating structure and contents. The contents of an XML document is obtained by extracting all `PCData` (Parsable Character Data: characters without tags) and all `CData` (Character Data: characters with possibly tags, starting with '`<![CDATA[`' ending in '`]]>`') 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:

$$
\begin{array}{ll}
\textbf{type}\ \mathsf{Extract}\{\![\star]\!\}\ \mathsf{t} & = \mathsf{t} \to [\mathsf{String}] \\
\textbf{type}\ \mathsf{Extract}\{\![\kappa \to \nu]\!\}\ \mathsf{t} & = \forall \mathsf{a}\,.\,\mathsf{Extract}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Extract}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a}) \\
\mathit{extract}\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Extract}\{\![\kappa]\!\}\ \mathsf{t} \\
\mathit{extract}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit} & = [\,] \\
\mathit{extract}\{\!|\mathsf{String}|\!\}\ s & = [s] \\
\mathit{extract}\{\!|\mathsf{a}\ \mathtt{:+:}\ \mathsf{b}|\!\}\ (\mathit{Inl}\ x) & = \mathit{extract}\{\!|\mathsf{a}|\!\}\ x \\
\mathit{extract}\{\!|\mathsf{a}\ \mathtt{:+:}\ \mathsf{b}|\!\}\ (\mathit{Inr}\ y) & = \mathit{extract}\{\!|\mathsf{b}|\!\}\ y \\
\mathit{extract}\{\!|\mathsf{a}\ \mathtt{:*:}\ \mathsf{b}|\!\}\ (x\ \mathtt{:*:}\ y) & = \mathit{extract}\{\!|\mathsf{a}|\!\}\ x +\!\!+ \mathit{extract}\{\!|\mathsf{b}|\!\}\ y \\
\mathit{extract}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ b) & = \mathit{extract}\{\!|\mathsf{a}|\!\}\ b.
\end{array}
$$

Note that it is possible to give special instances of a generic function on a particular type, as with $\mathit{extract}\{\!|\mathsf{String}|\!\}$ 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, the operator $+\!\!+$ in the product case is a source of inefficiency. It can be removed using a standard transformation, see Exercise 8 in the first part of these lecture notes.

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 is obtained by replacing all strings by empty tuples. Thus we obtain a value that has a different type, in which occurrences of the type String have been replaced by the type (). This is another example of a type-indexed data type [23]. 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.

```
Shape{|Unit|}    = Unit
Shape{|String|}  = ()
Shape{|a :+: b|} = Shape{|a|} :+: Shape{|b|}
Shape{|a :*: b|} = Shape{|a|} :*: Shape{|b|}
Shape{|Con a|}   = Con (Shape{|a|})
```

The generic function *shape* returns the shape of a value of any data type. It has the following kind-indexed type.

```
type Shape{|*|} t       = t → Shape{|t|}
type Shape{|κ → ν|} t = ∀a . Shape{|κ|} a → Shape{|ν|} (t a)
```

Note that we use the same name both for the kind-indexed type of function *shape*, as well as the type-indexed data type Shape. They can be distinguished based on their index.

```
shape{|t :: κ|}              :: Shape{|κ|} t
shape{|Unit|} Unit           = Unit
shape{|String|} s            = ()
shape{|a :+: b|} (Inl a)     = Inl (shape{|a|} a)
shape{|a :+: b|} (Inr b)     = Inr (shape{|b|} b)
shape{|a :*: b|} (a :*: b)   = (shape{|a|} a :*: shape{|b|} b)
shape{|Con c a|} (Con b)     = Con (shape{|a|} 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*:

$$insert\{\!|\mathsf{t} :: \star|\!\} :: \mathsf{Shape}\{\!|\mathsf{t}|\!\} \to [\mathsf{String}] \to \mathsf{t}.$$

The generic definition (with a kind-indexed type) of *insert* is left as an exercise.

Encoding constructors. The constructor of a value 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`. We start counting with 0. Furthermore, a value of a data type with a single constructor is represented using 0 bits. Consequently, the values of all types except for String and Lang in the running example are represented using 0 bits.

We assume there exists a function *constructorPosition* which given a constructor returns a pair of integers: its position in the list of constructors of the data type, and the number of constructors of the data type.

$$constructorPosition :: \mathsf{ConDescr} \to (\mathsf{Int}, \mathsf{Int})$$

Function *constructorPosition* can be defined by means of function *constructors*, which returns the constructor descriptions of a data type. This function is defined in the module *Collect*, which can be found in the library of Generic Haskell.

$$constructors\{\!|\mathsf{t} :: \star|\!\} :: [\mathsf{ConDescr}]$$

Function *constructors* is defined for arbitrary kinds in module Collect. We omit the definitions of both function *constructors* and function *constructorPosition*.

The function *encode* takes a value, and encodes it as a value of type Bin, a list of bits, defined in the first part of these lecture notes. The difference with the function *encode* that is defined in the first part of these lecture notes is that here we encode the constructors of the value, and not the choices made in the sum. On average, the function *encode* given here compresses much better than the function *encode* from the first part of these lecture notes.

$$\begin{aligned}
&\mathbf{type}\ \mathsf{Encode}\{\![\star]\!\}\ \mathsf{t} &&= \mathsf{Shape}\{\!|\mathsf{t}|\!\} \to \mathsf{Bin}\\
&\mathbf{type}\ \mathsf{Encode}\{\![\kappa \to \nu]\!\}\ \mathsf{t} &&= \forall \mathsf{a}\ .\ \mathsf{Encode}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Encode}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})
\end{aligned}$$

The interesting case in the definition of function *encode* is the constructor case. We first give the simple cases:

$$\begin{aligned}
&encode\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{Encode}\{\![\kappa]\!\}\ \mathsf{t}\\
&encode\{\!|\mathsf{Unit}|\!\}\ _ &&= [\,]\\
&encode\{\!|\mathsf{String}|\!\}\ _ &&= [\,]\\
&encode\{\!|\mathsf{a} :\!*\!: \mathsf{b}|\!\}\ (a :\!*\!: b) &&= encode\{\!|\mathsf{a}|\!\}\ a +\!\!+ encode\{\!|\mathsf{b}|\!\}\ b\\
&encode\{\!|\mathsf{a} :\!+\!: \mathsf{b}|\!\}\ (Inl\ a) &&= encode\{\!|\mathsf{a}|\!\}\ a\\
&encode\{\!|\mathsf{a} :\!+\!: \mathsf{b}|\!\}\ (Inr\ b) &&= encode\{\!|\mathsf{b}|\!\}\ b.
\end{aligned}$$

For Unit and String there is nothing to encode. The product case encodes the components of the product, and concatenates the results. The sum case strips of the *Inl* or *Inr* constructor, and encodes the argument.

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.

$$
\begin{aligned}
&\mathit{encode}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ a) = \\
&\quad \mathit{intinrange2bits}\ (\mathit{constructorPosition}\ c) +\!\!+ \mathit{encode}\{\!|\mathsf{a}|\!\}\ a \\
&\mathit{intinrange2bits} :: (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Bin}
\end{aligned}
$$

We omit the definition of the function to decode a list of bits into a value of a data type. This function is the inverse of function *encode* defined in this section, and is very similar to the function *decodes* given in the first part of these lecture notes.

Compressing the contents. Finally, it remains to compress the contents of an XML document. At the moment we use the Unix compress utility [52] to compress the strings obtained from the document.

3.2 Analysis

How does XCompreZ perform, and how does it compare with other XML compressors? The analysis given in this section is limited: XCompreZ is used as an example of a generic program, and not as the latest development in XML compression. Furthermore, we have not been able to obtain the executables or the source code of most of the existing XML compressors.

Existing XML compressors. Structure-specific compression methods give much better compression results [4, 15, 14, 46] than conventional compression methods such as the Unix compress utility [52]. There exist many XML compressors; we know of XMLZip [12], XMill [36], ICT's XML-Xpress [25], Millau [16], XMLPPM [6], XGrind [48], and lossy XML compression [5]. We will not perform an exhaustive comparison between our compressor and these compressors, but we will briefly compare our compressor with XMill.

Compression ratio. 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. When we replace HaXml by a tool that generates a data type for a schema, we expect that we can achieve better compression ratios.

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.

Extensions of XCOMPREZ. 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 [32].

XMill stores the strings in the elements in so-called containers. The standard approach in XMill is to use different containers for different elements, so that, for example, all authors are stored in the author container, all dates in the date container, etc. Since the strings in for example the container for dates are very similar, standard compression methods can compress the containers with a larger factor than the single file obtained by storing all strings that appear in the XML document. Again, it is easy to implement this feature as a generic program [32].

Finally, we have also used (Adaptive) Arithmetic Coding [3] to compress the constructors.

Specializing XCOMPREZ. XCOMPREZ can not only be used for compressing XML documents, but also for compressing values of arbitrary data types that have not necessarily been generated by DtdToHaskell.

Suppose we have a data type that contains a constructor *Age*, which takes an integer as argument and denotes the age of a human being. Since 128 is currently a safe upperbound for the age of a human being, it suffices to use 7 bits for the integer argument of *Age*. Suppose *compressAge* calculates these 7 bits from an age. Then we can reuse the definition of *encode* together with a constructor case [10] to define a function *specialEncode*.

$$
\begin{array}{ll}
\mathit{specialEncode}\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Encode}\{\![\kappa]\!\}\ \mathsf{t} \\
\mathit{specialEncode}\{\!|\mathbf{case}\ \mathit{Age}|\!\}\ (\mathit{Age}\ i) & = \mathit{compressAge}\ i \\
\mathit{specialEncode}\{\!|\mathsf{a}|\!\} & = \mathit{encode}\{\!|\mathsf{a}|\!\}
\end{array}
$$

Function *specialEncode* is still a generic encoding function, but on values of the form *Age* i it uses a different compress function.

3.3 Conclusions

We have shown how to implement an XML compressor as a generic program. XCOMPREZ compresses better than for example XMill because it uses the information about an XML document present in a DTD.

There exist several other classes of XML tools that can be implemented as generic programs, and that would benefit from such an implementation. Examples of such tools are XML editors and XML databases [17]. The combination of HaXml and generic programming as in Generic Haskell is very useful for implementing the kind of XML tools for which DTDs play an important rôle. 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, a DTD-aware XML compressor, such as XCOMPREZ described in this paper, compresses considerably better than XML compressors that don't take the DTD into account, such as XMill. Furthermore, our compressor is much smaller than XMill.

Although we think Generic Haskell is very useful for developing DTD-aware XML tools, there are some features of XML tools that are difficult 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. A flexible extension of type-indexed data types [23] might offer a solution to this problem. We believe that fusing HaXml, or a tool based on Schemas, with Generic Haskell, obtaining a 'domain-specific' language [8] 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. Adapt the function *extract* such that it returns a list of containers, where a container is a list of strings. Return a (possibly empty) container for every constructor name.

Exercise 4. There might be many empty containers when using the approach from the previous exercise. Analyse a data type for occurrences of the type String under constructors. Use this analysis to only return containers for constructor names that might contain strings.

Exercise 5. Function *insert* takes the shape and the contents (a list of strings) of a value, and inserts the strings at the right positions in the shape. Define function *insert* as a a generic function with a kind-indexed type.

Exercise 6. 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 other encodings can be used as well.

4 The zipper

This section shows how to define a so-called 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 generic functions working on it.

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

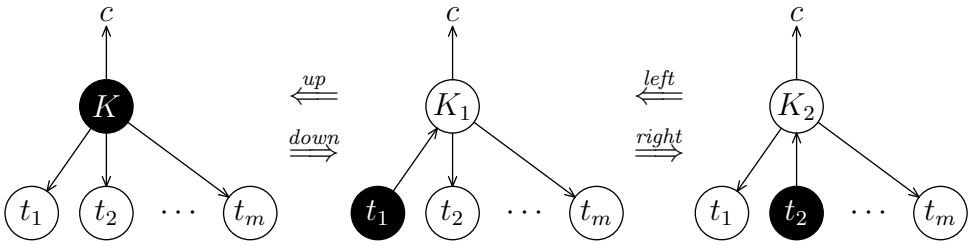
$$\textbf{data}\ \mathsf{Tree\ a\ b} = \mathit{Tip}\ \mathsf{a} \mid \mathit{Node}\ (\mathsf{Tree\ a\ b})\ \mathsf{b}\ (\mathsf{Tree\ a\ b}).$$

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

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

4.1 The basic idea

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



4.2 Data types as fixed points of pattern functors

As mentioned above, in order to use the zipper, we have to be explicit about the fixed points in data type definitions. Therefore, we introduce the data type Fix,

which is used to define a data type as a fixed point of a pattern functor. The pattern functor makes the recursion explicit in a data type.

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

This is a labelled variant of the data type Fix defined in Section 1.2 of the first part of these lecture notes. For example, the data types of natural numbers and bushes can be defined using explicit fixed points as follows:

$$\begin{array}{ll}
\textbf{data}\ \mathsf{NatF\ a} & = \mathit{ZeroF} \mid \mathit{SuccF}\ \mathsf{a} \\
\textbf{type}\ \mathsf{Nat} & = \mathsf{Fix\ NatF} \\
\textbf{data}\ \mathsf{BushF\ a} & = \mathit{LeafF}\ \mathsf{Char} \mid \mathit{BinF}\ \mathsf{a\ a} \\
\textbf{type}\ \mathsf{Bush} & = \mathsf{Fix\ BushF}.
\end{array}$$

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

4.3 Type indices of higher kinds

The types that occur in the indices of a generic function have kind $\star$ as their base kind. For example, Int, Char and Unit are all of kind $\star$, and :+: and :*: have kind $\star \to \star \to \star$. In this section we are going to define generic functions which have $\star \to \star$ as their base kind. We need slightly different type indices for generic functions operating on types of kind $\star \to \star$:

$$\begin{array}{ll}
\mathsf{K\ t} & = \Lambda \mathsf{a}\ .\ \mathsf{t} \\
\mathsf{f1\ :+:\ f2} & = \Lambda \mathsf{a}\ .\ \mathsf{f1\ a\ :+:\ f2\ a} \\
\mathsf{f1\ :*:\ f2} & = \Lambda \mathsf{a}\ .\ \mathsf{f1\ a\ :*:\ f2\ a} \\
\mathsf{Con}\ c\ \mathsf{f} & = \Lambda \mathsf{a}\ .\ \mathsf{Con}\ c\ \mathsf{(f\ a)} \\
\mathsf{Id} & = \Lambda \mathsf{a}\ .\ \mathsf{a}.
\end{array}$$

We have the constant functor K, which lifts a type of kind $\star$ to kind $\star \to \star$. We will need K Unit as well as K Char (or more general, K t for all primitive types). We overload :+:, :*:, and Con, to be lifted versions of their previously defined counterparts. The only new type index in this set of indices of kind $\star \to \star$ is the identity functor Id. Hinze [20] shows that these types are the normal forms of types of kind $\star \to \star$.

4.4 Locations

A location is a subtree, together with a context, which encodes the path from the top of the original tree to the selected subtree. The type-indexed data type

Loc returns a type for locations given an argument pattern functor.

$$\begin{array}{lcl}
\mathsf{Loc}\{\!|\mathsf{f} :: \star \to \star|\!\} & :: & \star \\
\mathsf{Loc}\{\!|\mathsf{f}|\!\} & = & (\mathsf{Fix}\ \mathsf{f}, \mathsf{Context}\{\!|\mathsf{f}|\!\}\ (\mathsf{Fix}\ \mathsf{f})) \\
\mathsf{Context}\{\!|\mathsf{f} :: \star \to \star|\!\} & :: & \star \to \star \\
\mathsf{Context}\{\!|\mathsf{f}|\!\}\ \mathsf{r} & = & \mathsf{Fix}\ (\mathsf{LMaybe}\ (\mathsf{Ctx}\{\!|\mathsf{f}|\!\}\ \mathsf{r})) \\
\mathbf{data}\ \mathsf{LMaybe}\ \mathsf{f}\ \mathsf{a} & = & \mathit{LNothing} \mid \mathit{LJust}\ (\mathsf{f}\ \mathsf{a}),
\end{array}$$

where LMaybe is the lifted version of Maybe. The type Loc is defined in terms of Context, which constructs the context parameterized by the original tree type. The Context of a value is either empty (represented by *LNothing* in the LMaybe type), or it is a path from the root down into the tree. Such a path is constructed by means of the argument type of LMaybe: the type-indexed data type Ctx. The type-indexed data type Ctx is defined by induction on the pattern functor of the original data type. It can be seen as the *derivative* (as in calculus) of the pattern functor f [39, 1]. If the derivative of f is denoted by f', we have

$$\begin{array}{lcl}
\mathsf{const}' & = & 0 \\
(\mathsf{f} + \mathsf{g})' & = & \mathsf{f}' + \mathsf{g}' \\
(\mathsf{f} \times \mathsf{g})' & = & \mathsf{f}' \times \mathsf{g} + \mathsf{f} \times \mathsf{g}'.
\end{array}$$

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

$$\begin{array}{lcl}
\mathsf{Ctx}\{\!|\mathsf{f} :: \star \to \star|\!\} & :: & \star \to \star \to \star \\
\mathsf{Ctx}\{\!|\mathsf{Id}|\!\}\ \mathsf{r}\ \mathsf{c} & = & \mathsf{c} \\
\mathsf{Ctx}\{\!|\mathsf{K}\ \mathsf{Unit}|\!\}\ \mathsf{r}\ \mathsf{c} & = & \mathsf{Void} \\
\mathsf{Ctx}\{\!|\mathsf{K}\ \mathsf{Char}|\!\}\ \mathsf{r}\ \mathsf{c} & = & \mathsf{Void} \\
\mathsf{Ctx}\{\!|\mathsf{f1} :+: \mathsf{f2}|\!\}\ \mathsf{r}\ \mathsf{c} & = & \mathsf{Ctx}\{\!|\mathsf{f1}|\!\}\ \mathsf{r}\ \mathsf{c} :+: \mathsf{Ctx}\{\!|\mathsf{f2}|\!\}\ \mathsf{r}\ \mathsf{c} \\
\mathsf{Ctx}\{\!|\mathsf{f1} :*: \mathsf{f2}|\!\}\ \mathsf{r}\ \mathsf{c} & = & (\mathsf{Ctx}\{\!|\mathsf{f1}|\!\}\ \mathsf{r}\ \mathsf{c} :*: \mathsf{f2}\ \mathsf{r}) :+: (\mathsf{f1}\ \mathsf{r} :*: \mathsf{Ctx}\{\!|\mathsf{f2}|\!\}\ \mathsf{r}\ \mathsf{c})
\end{array}$$

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

For example, for natural numbers with pattern functor K Unit :+: Id, and for trees of type Bush with pattern functor BushF, which can be represented by K Char :+: (Id :*: Id) we obtain

$$\begin{array}{ll}
\mathsf{Context}\{\!|\mathsf{K}\ \mathsf{Unit} :+: \mathsf{Id}|\!\}\ \mathsf{r} & = \mathsf{Fix}\ (\mathsf{LMaybe}\ (\mathsf{NatC}\ \mathsf{r})) \\
\mathsf{Context}\{\!|\mathsf{K}\ \mathsf{Char} :+: \mathsf{Id} :*: \mathsf{Id}|\!\}\ \mathsf{r} & = \mathsf{Fix}\ (\mathsf{LMaybe}\ (\mathsf{BushC}\ \mathsf{r})) \\
\mathbf{data}\ \mathsf{NatC}\ \mathsf{r}\ \mathsf{c} & = \mathit{ZeroC}\ \mathsf{Void} \mid \mathit{SuccC}\ \mathsf{c} \\
\mathbf{data}\ \mathsf{BushC}\ \mathsf{r}\ \mathsf{c} & = \mathit{LeafC}\ \mathsf{Void} \mid \mathit{BinCL}\ (\mathsf{c}, \mathsf{r}) \mid \mathit{BinCR}\ (\mathsf{r}, \mathsf{c}).
\end{array}$$

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

McBride [39, 1] also defines a type-indexed zipper data type. His zipper slightly deviates from Huet's and our zipper: the navigation functions on McBride's zipper are not constant time anymore. The observation that the Context of a data type is its derivative (as in calculus) is due to McBride.

4.5 Navigation functions

We define generic 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 generic function that moves down to the leftmost recursive child of the current node, if such a child exists. Otherwise, if the current node is a leaf node, then *down* returns the location unchanged.

$$down\{\!|\mathsf{f} :: \star \to \star|\!\} :: \mathsf{Loc}\{\!|\mathsf{f}|\!\} \to \mathsf{Loc}\{\!|\mathsf{f}|\!\}$$

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

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

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

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

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

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

The function *down* is defined as follows.

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

To find the leftmost recursive child, we have to pattern match on the pattern functor f, and find the first occurrence of Id. The helper function *first* is a generic function that possibly returns the leftmost recursive child of a node, together with the context (a value of type $\mathsf{Ctx}\{\!|\mathsf{f}|\!\}\ \mathsf{c}\ \mathsf{t}$) of the selected child. The function

down then turns this context into a value of type Context by inserting it in the right ('non-top') component of a sum by means of *LJust*, and applying the fixed point constructor *In* to it.

```
first{|f :: * -> *|}              :: ∀c t . f t → c → Maybe (t, Ctx{|f|} c t)
first{|Id|} t c                   = return (t, c)
first{|K Unit|} t c               = Nothing
first{|K Char|} t c               = Nothing
first{|f1 :+: f2|} (Inl x) c      = do {(t, cx) ← first{|f1|} x c; return (t, Inl cx)}
first{|f1 :+: f2|} (Inr y) c      = do {(t, cy) ← first{|f2|} y c; return (t, Inr cy)}
first{|f1 :*: f2|} (x :*: y) c    = do {(t, cx) ← first{|f1|} x c
                                        ; return (t, Inl (cx, y))}
                                  ++ do {(t, cy) ← first{|f2|} y c
                                        ; return (t, Inr (x, cy))}
```

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

```
(++)              :: ∀a . Maybe a → Maybe a → Maybe a
Nothing ++ m = m
Just a ++ m  = Just a.
```

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

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

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

```
up{|f :: * -> *|} :: Loc{|f|} → Loc{|f|}
up{|f|} (t, c)    = case out c of
                      LNothing → (t, c)
                      LJust c' → do {ft ← insert{|f|} c' t;
                                     c'' ← extract{|f|} c';
                                     return (In ft, c'')}
```

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

this context.

$$
\begin{array}{ll}
\mathit{extract}\{\!|\mathsf{f} :: \star \to \star|\!\} & :: \forall \mathsf{c}\ \mathsf{t}\,.\,\mathsf{Ctx}\{\!|\mathsf{f}|\!\}\ \mathsf{c}\ \mathsf{t} \to \mathsf{Maybe}\ \mathsf{c} \\
\mathit{extract}\{\!|\mathsf{Id}|\!\}\ c & = \mathit{return}\ c \\
\mathit{extract}\{\!|\mathsf{K\ Unit}|\!\}\ c & = \mathit{Nothing} \\
\mathit{extract}\{\!|\mathsf{K\ Char}|\!\}\ c & = \mathit{Nothing} \\
\mathit{extract}\{\!|\mathsf{f1} :\!+\!: \mathsf{f2}|\!\}\ (\mathit{Inl}\ cx) & = \mathit{extract}\{\!|\mathsf{f1}|\!\}\ cx \\
\mathit{extract}\{\!|\mathsf{f1} :\!+\!: \mathsf{f2}|\!\}\ (\mathit{Inr}\ cy) & = \mathit{extract}\{\!|\mathsf{f2}|\!\}\ cy \\
\mathit{extract}\{\!|\mathsf{f1} :\!*\!: \mathsf{f2}|\!\}\ (\mathit{Inl}\ (cx, y)) & = \mathit{extract}\{\!|\mathsf{f1}|\!\}\ cx \\
\mathit{extract}\{\!|\mathsf{f1} :\!*\!: \mathsf{f2}|\!\}\ (\mathit{Inr}\ (x, cy)) & = \mathit{extract}\{\!|\mathsf{f2}|\!\}\ cy
\end{array}
$$

The function *extract* is polymorphic in c and in t.

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

$$
\begin{array}{ll}
\mathit{insert}\{\!|\mathsf{f} :: \star \to \star|\!\} & :: \forall \mathsf{c}\ \mathsf{t}\,.\,\mathsf{Ctx}\{\!|\mathsf{f}|\!\}\ \mathsf{c}\ \mathsf{t} \to \mathsf{t} \to \mathsf{Maybe}\ (\mathsf{f}\ \mathsf{t}) \\
\mathit{insert}\{\!|\mathsf{Id}|\!\}\ c\ t & = \mathit{return}\ t \\
\mathit{insert}\{\!|\mathsf{K\ Unit}|\!\}\ c\ t & = \mathit{Nothing} \\
\mathit{insert}\{\!|\mathsf{K\ Char}|\!\}\ c\ t & = \mathit{Nothing} \\
\mathit{insert}\{\!|\mathsf{f1} :\!+\!: \mathsf{f2}|\!\}\ (\mathit{Inl}\ cx)\ t & = \mathbf{do}\ \{x \leftarrow \mathit{insert}\{\!|\mathsf{f1}|\!\}\ cx\ t; \mathit{return}\ (\mathit{Inl}\ x)\} \\
\mathit{insert}\{\!|\mathsf{f1} :\!+\!: \mathsf{f2}|\!\}\ (\mathit{Inr}\ cy)\ t & = \mathbf{do}\ \{y \leftarrow \mathit{insert}\{\!|\mathsf{f2}|\!\}\ cy\ t; \mathit{return}\ (\mathit{Inr}\ y)\} \\
\mathit{insert}\{\!|\mathsf{f1} :\!*\!: \mathsf{f2}|\!\}\ (\mathit{Inl}\ (cx, y))\ t & = \mathbf{do}\ \{x \leftarrow \mathit{insert}\{\!|\mathsf{f1}|\!\}\ cx\ t; \mathit{return}\ (x, y)\} \\
\mathit{insert}\{\!|\mathsf{f1} :\!*\!: \mathsf{f2}|\!\}\ (\mathit{Inr}\ (x, cy))\ t & = \mathbf{do}\ \{y \leftarrow \mathit{insert}\{\!|\mathsf{f2}|\!\}\ cy\ t; \mathit{return}\ (x, y)\}
\end{array}
$$

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

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

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

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

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

$$
\forall m\,.\,\mathit{right}\{\!|\mathsf{f}|\!\}\ m \neq m \implies (\mathit{left}\{\!|\mathsf{f}|\!\} \cdot \mathit{right}\{\!|\mathsf{f}|\!\})\ m = m,
$$

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

of m, and the context of $right\{\!|\mathsf{f}|\!\}\ m$ is the context of m in which the context is replaced by the selected subtree of m, and the first subtree to the right of the context of m is replaced by the context of m.

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

$$
\begin{array}{lcl}
right\{\!|\mathsf{f} :: \star \to \star|\!\} :: & \mathsf{Loc}\{\!|\mathsf{f}|\!\} \to \mathsf{Loc}\{\!|\mathsf{f}|\!\} & \\
right\{\!|\mathsf{f}|\!\}\ (t, c) & = & \mathbf{case}\ out\ c\ \mathbf{of} \\
& & \quad LNothing \to (t, c) \\
& & \quad LJust\ c' \to \mathbf{case}\ next\{\!|\mathsf{f}|\!\}\ t\ c'\ \mathbf{of} \\
& & \qquad\qquad Just\ (t', c'') \to (t', In\ (LJust\ c'')) \\
& & \qquad\qquad Nothing \to (t, c)
\end{array}
$$

The helper function *next* is a generic 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 $left\{\!|\mathsf{f}|\!\} \cdot right\{\!|\mathsf{f}|\!\} = id$ (on locations in which it is possible to go to the right), there exists a function *previous*, such that

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

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

$$
\begin{array}{l}
next\{\!|\mathsf{f} :: \star \to \star|\!\} :: \forall \mathsf{c}\ \mathsf{t}\ .\ \mathsf{t} \to \mathsf{Ctx}\{\!|\mathsf{f}|\!\}\ \mathsf{c}\ \mathsf{t} \to \mathsf{Maybe}\ (\mathsf{t}, \mathsf{Ctx}\{\!|\mathsf{f}|\!\}\ \mathsf{c}\ \mathsf{t}) \\
next\{\!|\mathsf{Id}|\!\}\ t\ c \qquad\qquad\qquad\qquad = Nothing \\
next\{\!|\mathsf{K}\ \mathsf{Unit}|\!\}\ t\ c \qquad\qquad\qquad = Nothing \\
next\{\!|\mathsf{K}\ \mathsf{Char}|\!\}\ t\ c \qquad\qquad\qquad = Nothing \\
next\{\!|\mathsf{f1} :+: \mathsf{f2}|\!\}\ t\ (Inl\ cx) \\
\quad = \mathbf{do}\ \{(t', cx') \leftarrow next\{\!|\mathsf{f1}|\!\}\ t\ cx; return\ (t', Inl\ cx')\} \\
next\{\!|\mathsf{f1} :+: \mathsf{f2}|\!\}\ t\ (Inr\ cy) \\
\quad = \mathbf{do}\ \{(t', cy') \leftarrow next\{\!|\mathsf{f2}|\!\}\ t\ cy; return\ (t', Inr\ cy')\} \\
next\{\!|\mathsf{f1} :*: \mathsf{f2}|\!\}\ t\ (Inl\ (cx, y)\ ) \\
\quad = \mathbf{do}\ \{(t', cx') \leftarrow next\{\!|\mathsf{f1}|\!\}\ t\ cx; return\ (t', Inl\ (cx', y))\} \\
\qquad +\!\!+\mathbf{do}\ \{c \leftarrow extract\{\!|\mathsf{f1}|\!\}\ cx; \\
\qquad\qquad\ x \leftarrow insert\{\!|\mathsf{f1}|\!\}\ cx\ t; \\
\qquad\qquad\ (t', cy) \leftarrow first\{\!|\mathsf{f2}|\!\}\ y\ c; \\
\qquad\qquad\ return\ (t', Inr\ (x, cy))\} \\
next\{\!|\mathsf{f1} :*: \mathsf{f2}|\!\}\ t\ (Inr\ (x, cy)) \\
\quad = \mathbf{do}\ \{(t', cy') \leftarrow next\{\!|\mathsf{f2}|\!\}\ t\ cy; return\ (t', Inr\ (x, cy'))\}
\end{array}
$$

The first three lines in this definition show that it is impossible to go to the right in an identity or constant context. If the context argument is a value of a sum, we select the next element in the appropriate component of the sum. The product case is the most interesting one. If the context is in the right component of a pair, *next* returns the next value of that context, properly combined with

the left component of the tuple. On the other hand, if the context is in the left component of a pair, the next value may be either in that left component (the context), or it may be in the right component (the value). If the next value is in the left component, it is returned by the first line in the definition of the product case. If it is not, *next* extracts the context c (the context of the parent) from the left context cx, it inserts the given value in the context cx giving a 'tree' value x, and selects the first component in the right component of the pair, using the extracted context c for the new context. The new context that is thus obtained is combined with x into a context for the selected tree.

Exercise 7. Define the function *left*:

$$left\{\!|\mathsf{f} :: \star \to \star|\!\} :: \mathsf{Loc}\{\!|\mathsf{f}|\!\} \to \mathsf{Loc}\{\!|\mathsf{f}|\!\}.$$

Exercise 8. If you don't want to use the zipper, you can alternatively 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:

$$\begin{array}{l}\textbf{data}\ \mathsf{Dir}\ \ = L \mid R\\ \textbf{type}\ \mathsf{Path} = [\mathsf{Dir}].\end{array}$$

The to be defined function *selectCon* takes a value of a data type and a path, and returns the constructor name at the position denoted by the path. For example,

$\textbf{data}\ \mathsf{List}\ = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{Char}\ \mathsf{List}$
$\mathit{selectCon}\{\!|\mathsf{List}|\!\}\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 6\ \mathit{Nil})))\ [R, R, R]$
$\Longrightarrow$ `"Nil"`
$\textbf{data}\ \mathsf{Tree} = \mathit{Leaf}\ \mathsf{Int} \mid \mathit{Node}\ \mathsf{Tree}\ \mathsf{Int}\ \mathsf{Tree}$
$\mathit{selectCon}\{\!|\mathsf{Tree}|\!\}\ (\mathit{Node}\ (\mathit{Leaf}\ 1)\ 3\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ 1\ (\mathit{Leaf}\ 5)))\ [R, R]$
$\Longrightarrow$ `"Node"`
$\mathit{selectCon}\{\!|\mathsf{Tree}|\!\}\ (\mathit{Node}\ (\mathit{Leaf}\ 1)\ 3\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ 1\ (\mathit{Leaf}\ 5)))\ [R, R, L]$
$\Longrightarrow$ `"Leaf"`.

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

Exercise 9. Define the function *left*, which takes a location, and returns the location to the left of the argument location, if possible.

Exercise 10. 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

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

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

$$\textbf{data}\ \mathsf{HoleTree\ a\ b} = \mathit{Hole} \mid \mathit{Tip}\ \mathsf{a} \mid \mathit{Node}\ (\mathsf{HoleTree\ a\ b})\ \mathsf{b}\ (\mathsf{HoleTree\ a\ b}).$$

- 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 generic function *toHole* which translates a value of a data type t to a value of the data type Hole{|t|}, and a function *fromHole* that does the inverse for values that do not contain holes anymore:

$$
\begin{array}{lll}
\mathit{toHole}\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{ToHole}\{\!|\kappa|\!\}\ \mathsf{t} \\
\mathit{fromHole}\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{FromHole}\{\!|\kappa|\!\}\ \mathsf{t}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathsf{ToHole}\{\!|\star|\!\}\ \mathsf{t} \quad = \mathsf{t} \to \mathsf{Hole}\{\!|\mathsf{t}|\!\} \\
\textbf{type}\ \mathsf{FromHole}\{\!|\star|\!\}\ \mathsf{t} = \mathsf{Hole}\{\!|\mathsf{t}|\!\} \to \mathsf{t}.
\end{array}
$$

5 Conclusions

We have developed three advanced applications in Generic Haskell. In these examples we use, besides generic 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. 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. Michael Abott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications, TLCA 2003*, 2003. To appear.
2. 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.
3. Richard Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International Summer School, Oxford, UK*, volume 2638 of *LNCS*. Springer-Verlag, 2003. To appear.
4. Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.

5. Mario Cannataro, Gianluca Carelli, Andrea Pugliese, and Domenico Sacca. Semantic lossy compression of XML data. In *Knowledge Representation Meets Databases*, 2001.
6. James Cheney. Compressing xml with multiplexed hierarchical models. In *Proceedings of the 2001 IEEE Data Compression Conference, DCC'01*, pages 163–172, 2001.
7. Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and tracing lazy functional programs using Quickcheck and Hat. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional programming, 4th International Summer School, Oxford, UK*, volume 2638 of *LNCS*. Springer-Verlag, 2003. To appear.
8. Dave Clarke. Towards GH(XML). Talk at the Generic Haskell meeting, see `http://www.generic-haskell.org/talks.html`, 2001.
9. 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/`.
10. Dave Clarke and Andres Löh. Generic Haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming*, volume 243 of *IFIP*, pages 21–48. Kluwer Academic Publishers, January 2003.
11. 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.
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. William S. Evans and Christopher W. Fraser. Bytecode compression via profiled grammar rewriting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 148–155, 2001.
15. Michael Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In *Mobile Object Systems: Towards the Programmable Internet*, pages 263–276. Springer-Verlag: Heidelberg, Germany, 1997.
16. 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.
17. Paul Hagg. A framework for developing generic XML Tools. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.
18. Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.
19. Ralf Hinze. *Generic Programs and Proofs*. 2000. Habilitationsschrift, Bonn University.
20. Ralf Hinze. A new approach to generic functional programming. In *Conference Record of POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 2000.
21. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.
22. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory, 2003. To appear.

23. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th Mathematics of Program Construction Conference, MPC'02*, volume 2386 of *LNCS*, pages 148–174, 2002.
24. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
25. INC Intelligent Compression Technologies. XML-Xpress. Whitepaper available from `http://www.ictcompress.com/products_xmlxpress.html`, 2001.
26. 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.
27. Patrik Jansson. The WWW home page for polytypic programming. Available from `http://www.cs.chalmers.se/~patrikj/poly/`, 2001.
28. Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.
29. 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.
30. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
31. 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.
32. Johan Jeuring and Paul Hagg. XCOMPREZ. Available from `http://www.generic-haskell.org/xmltools/XComprez/`, 2002.
33. Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, 2nd edition, 1998.
34. Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. Submitted for publication, 2002.
35. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, 2003.
36. 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.
37. Andres Löh, Dave Clarke, and Johan Jeuring. Generic Haskell, naturally: The language and its type system. In preparation, 2003.
38. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
39. Connor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.
40. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.
41. Lambert Meertens. Functor pulling. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.
42. 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.
43. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
44. Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland*, pages 77–86, 1998.

45. 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.
46. C.H. Stork, V. V. Haldar, and M. Franz. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42, Department of Information and Computer Science, University of California, Irvine, 2000.
47. 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.
48. Pankaj Tolani and Jayant R. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, 2002.
49. W3C. XML 1.0. Available from `http://www.w3.org/XML/`, 1998.
50. 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.
51. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159, 1999.
52. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.