Generalizing Generalized Tries

November 1998 (revised February 1999)

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* `ralf@informatik.uni-bonn.de`)

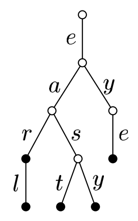
Abstract

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 C.P. Wadsworth and others, R.H. Connelly and F.L. Morris generalized the concept to permit indexing by elements of an arbitrary monomorphic datatype. Here we go one step further and define tries and operations on tries generically for arbitrary first-order polymorphic datatypes. The derivation is based on techniques recently developed in the context of polytypic programming. It is well known that for the implementation of generalized tries nested datatypes and polymorphic recursion are needed. Implementing tries for polymorphic datatypes places even greater demands on the type system: it requires rank-2 type signatures and higher-order polymorphic nested datatypes. Despite these requirements the definition of generalized tries for polymorphic datatypes is surprisingly simple which is mostly due to the framework of polytypic programming.

All generalizations are dangerous, even this one.
— Alexandre Dumas

1 Introduction

The concept of a trie was introduced by A. Thue in 1912 as a means to represent a set of strings, see (Knuth, 1998). 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 which 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 theme. As we shall see it is essential for the further development.



At a more abstract level a trie 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. For concreteness, programs will be given in the functional programming language Haskell 98 (Peyton Jones *et al.*, 1999). If strings are defined by the following datatype

$$\mathbf{data}\ \mathit{Str} \quad = \quad \mathit{Nil} \mid \mathit{Cons}\ \mathit{Char}\ \mathit{Str} \ ,$$

we can represent string-indexed tries as follows.

$$\mathbf{data}\ \mathit{MapStr}\ v \quad = \quad \mathit{TrieStr}\ (\mathit{Maybe}\ v)\ (\mathit{MapChar}\ (\mathit{MapStr}\ v))$$

The first component of the constructor *TrieStr* 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. In this case the first component equals *Nothing*. The second component corresponds to the edge map. To keep the example manageable we assume that a suitable data structure, *MapChar*, and an associated look-up function *lookupChar* are predefined. Now, to lookup a non-empty string, say, *Cons c s* we lookup *c* in the edge map obtaining a trie which is then recursively searched for *s*.

$$\begin{array}{lcl}
\mathit{lookupStr} & :: & \mathit{Str} \to \mathit{MapStr}\ v \to v \\
\mathit{lookupStr}\ \mathit{Nil}\ (\mathit{TrieStr}\ \mathit{Nothing}\ \mathit{tc}) & = & \mathit{error}\ \texttt{"not found"} \\
\mathit{lookupStr}\ \mathit{Nil}\ (\mathit{TrieStr}\ (\mathit{Just}\ v)\ \mathit{tc}) & = & v \\
\mathit{lookupStr}\ (\mathit{Cons}\ c\ s)\ (\mathit{TrieStr}\ \mathit{tn}\ \mathit{tc}) & = & (\mathit{lookupStr}\ s \circ \mathit{lookupChar}\ c)\ \mathit{tc}
\end{array}$$

Based on work by C.P. Wadsworth and others, R.H. Connelly and F.L. Morris (1995) have generalized the concept of a trie to permit indexing by elements of an arbitrary monomorphic datatype. The definition of *lookupStr* already gives a clue how a suitable generalization might look like: the trie *TrieStr tn tc* contains a finite map for each constructor of the datatype *Str*; to lookup *Cons c s* the look-up functions for the components, *c* and *s*, are simply composed. The type constructor *Maybe* can be seen as implementing finite maps over the unit datatype. Generally, if we have a datatype with *k* constructors, the corresponding trie has *k* components. To lookup a constructor with *n* components, we must select the corresponding finite map and compose *n* look-up functions of the appropriate types. To illustrate, consider the datatype of external search trees.

$$\mathbf{data}\ \mathit{Bin} \quad = \quad \mathit{Leaf}\ \mathit{Str} \mid \mathit{Node}\ \mathit{Bin}\ \mathit{Char}\ \mathit{Bin}$$

The trie for external search trees is given by

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{MapBin}\ v & = & \mathit{TrieBin}\ (\mathit{MapStr}\ v) \\
& & \qquad (\mathit{MapBin}\ (\mathit{MapChar}\ (\mathit{MapBin}\ v)))\ .
\end{array}$$

The type *MapBin* is an instance of a so-called *nested datatype* (*nest* for short). The term 'nested datatype' has been coined by R. Bird and L. Meertens (1998) and characterizes polymorphic datatypes whose definition involves 'recursive calls'—*MapBin* (*MapChar* (*MapBin v*)) in the example above—which are substitution instances of the defined type. Functions operating on nested datatypes are known to require a non-schematic form of recursion, called *polymorphic recursion* (Mycroft, 1984). The look-up function on external search trees may serve as an example.

$$\begin{array}{lcl} \mathit{lookupBin} & :: & \mathit{Bin} \to \mathit{MapBin}\ v \to v \\ \mathit{lookupBin}\ (\mathit{Leaf}\ s)\ (\mathit{TrieBin}\ \mathit{tl}\ \mathit{tn}) & = & \mathit{lookupStr}\ s\ \mathit{tl} \\ \mathit{lookupBin}\ (\mathit{Node}\ \ell\ c\ r)\ (\mathit{TrieBin}\ \mathit{tl}\ \mathit{tn}) & & \\ \qquad = \ (\mathit{lookupBin}\ \ell \circ \mathit{lookupChar}\ c \circ \mathit{lookupBin}\ r)\ \mathit{tn} & & \end{array}$$

Looking up a node involves two recursive calls. The second, *lookupBin* r, is of type $\mathit{Bin} \to \mathit{MapBin}\ (\mathit{MapChar}\ (\mathit{MapBin}\ v)) \to \mathit{MapChar}\ (\mathit{MapBin}\ v)$ which is a substitution instance of the declared type. Haskell allows polymorphic recursion only if an explicit type signature is provided for the function(s). The rationale behind this restriction is that type inference in the presence of polymorphic recursion is undecidable (Henglein, 1993).

Note that it is absolutely necessary that *MapBin* and *lookupBin* are parametric with respect to the codomain of the finite maps. If we restricted the type of *lookupBin* to $\mathit{Bin} \to \mathit{MapBin}\ s \to s$ for some fixed type s, the definition would no longer type-check. This also explains why the construction does not work for the finite set abstraction.

From the discussion above it should be clear how to define tries for arbitrary *monomorphic* datatypes. In this paper we go one step further and show how to generalize the concept to arbitrary *first-order polymorphic* datatypes. We will answer in particular the intriguing question what the *generalized trie of a nested datatype* looks like. Note that this question is not only of theoretical but also of practical interest. A number of data structures, such as 2-3 trees or red-black trees, have recently been shown to be expressible by nested declarations. R. Bird and R. Paterson (1998) use a nested datatype for expressing de Bruijn notation. Now, if a look-up structure for de Bruijn terms is required, say, to implement common subexpression elimination, we are confronted with the problem of constructing generalized tries for a nested datatype.

To develop generalized tries for polymorphic datatypes we will employ the framework of *polytypic programming*. In short, a generic or polytypic function is one which is defined by induction on the structure of types. A simple example for a polytypic function is $\mathit{flatten} :: f\ a \to [a]$ which traverses an element of $f\ a$ and collects all elements of type a from left to right in a list. The function *flatten* can sensibly be defined for each polymorphic type and it is usually a tiresome, routine matter to do so. A polytypic programming language enables the user to program *flatten* once and for all times. The specialization of *flatten* to concrete instances of f is then handled automatically by the system. Polytypic programming can be surprisingly simple. In a companion paper (Hinze, 1999) we show that it suffices to define a polytypic function on predefined types, type variables, sums, and products. This information is sufficient to specialize a polytypic function to arbitrary datatypes including mutually recursive and nested datatypes.

Generalized tries make a particularly interesting application of polytypic programming. The central insight is that a trie can be considered as a *type-indexed datatype*. This makes it possible to define tries and operations on tries generically for arbitrary polymorphic datatypes. 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 a product of tries and a trie for a product is a composition of tries. The extension to arbitrary datatypes is then uniquely defined.

We have seen that nested datatypes and polymorphic recursion are necessary for the implementation of generalized tries. Implementing tries for polymorphic datatypes, especially nested datatypes, places even greater demands on the type system: it requires rank-2 type signatures (McCracken, 1984), higher-order polymorphic datatypes (Jones, 1995), and higher-order polymorphic nests. Fortunately, all major Haskell system provide the necessary extensions.

The rest of this paper is structured as follows. In Section 2 we briefly review the theoretical background of polytypic programming. A more detailed account is given in (Hinze, 1999). Section 3 applies the technique to implement a finite map abstraction based on generalized tries. Section 4 discusses variations of the theme. Finally, Section 5 reviews related work and points out a direction for future work.

2 A polytypic programming primer

2.1 Datatypes

A polytypic function is one which is parameterised by a datatype. The polytypic programming primer therefore starts with a brief investigation of the structure of types. The following definitions will serve as running examples throughout the paper.

$$\begin{array}{lcl}
\textbf{data}\ \mathit{List}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\
\textbf{data}\ \mathit{Bintree}\ a_1\ a_2 & = & \mathit{Leaf}\ a_1 \mid \mathit{Node}\ (\mathit{Bintree}\ a_1\ a_2)\ a_2\ (\mathit{Bintree}\ a_1\ a_2) \\
\textbf{data}\ \mathit{Fork}\ a & = & \mathit{Fork}\ a\ a \\
\textbf{data}\ \mathit{Perfect}\ a & = & \mathit{Null}\ a \mid \mathit{Succ}\ (\mathit{Perfect}\ (\mathit{Fork}\ a)) \\
\textbf{data}\ \mathit{Sequ}\ a & = & \mathit{Empty} \mid \mathit{Zero}\ (\mathit{Sequ}\ (\mathit{Fork}\ a)) \mid \mathit{One}\ a\ (\mathit{Sequ}\ (\mathit{Fork}\ a))
\end{array}$$

The meaning of these datatypes in a nutshell: the first equation defines the ubiquitous datatype of polymorphic lists; *Bintree* encompasses external binary search trees. The types *Perfect* and *Sequ* are examples for nested datatypes: *Perfect* comprises perfect binary leaf trees (Dielissen & Kaldewaij, 1995) and *Sequ* implements binary random-access lists (Okasaki, 1998). Both definitions make use of the auxiliary datatype *Fork* whose elements may be interpreted as internal nodes.

Haskell's **data** construct combines several features in a single coherent form: sums, products, and recursion. Using more conventional notation ('+' for sums and '×' for products) and omitting constructor names we obtain the following emaciated recursion equations.

$$\begin{array}{lcl}
\mathit{List}\ a & = & 1 + a \times \mathit{List}\ a \\
\mathit{Bintree}\ a_1\ a_2 & = & a_1 + \mathit{Bintree}\ a_1\ a_2 \times a_2 \times \mathit{Bintree}\ a_1\ a_2 \\
\mathit{Fork}\ a & = & a \times a \\
\mathit{Perfect}\ a & = & a + \mathit{Perfect}\ (\mathit{Fork}\ a) \\
\mathit{Sequ}\ a & = & 1 + \mathit{Sequ}\ (\mathit{Fork}\ a) + a \times \mathit{Sequ}\ (\mathit{Fork}\ a)
\end{array}$$

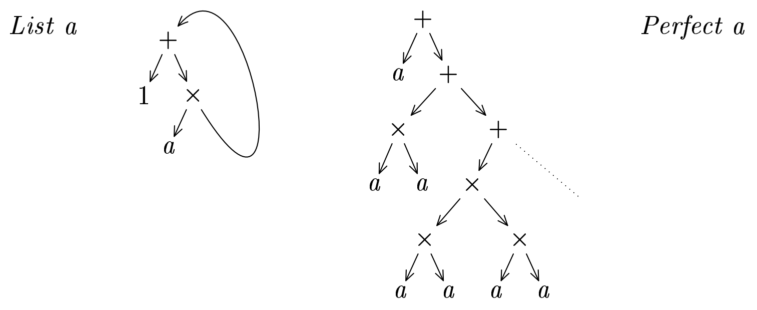


Fig. 1. Types interpreted as infinite type expressions.

In the following we treat 1, '+', and '×' as if they were given by the following datatype declarations.

$$\begin{array}{lcl}\textbf{data}\ 1 & = & () \\ \textbf{data}\ a_1 + a_2 & = & \mathit{Inl}\ a_1 \mid \mathit{Inr}\ a_2 \\ \textbf{data}\ a_1 \times a_2 & = & (a_1, a_2)\end{array}$$

Now, the central idea of polytypic programming is that the set of all types—or rather, the set of all type expressions itself can be modelled by a datatype. Assuming a fixed set of type variables $A = \{a_1, a_2, a_3, \ldots\}$ and a set of primitive type constructors $P = \{1, \mathit{Int}, \ldots, +, \times\}$ type expressions can be seen as defined by the following grammar.

$$T \quad = \quad A \mid P\ (T, \ldots, T)$$

The type $F(t_1, \ldots, t_n)$ denotes the application of an n-ary type constructor to n types. We omit the parenthesis when $n \leqslant 1$. We also write $t_1 + t_2$ for $+(t_1, t_2)$ and similarly $t_1 \times t_2$. Finally, we abbreviate a_1 to a when defining unary type constructors.

The question remains how recursive types are modelled. The answer probably comes as no surprise to the experienced Haskell programmer: recursive types are modelled by infinite type expressions! Figure 1 displays the infinite type expressions defined by the equations for *List* and *Perfect*.

2.2 Polytypic definitions

A polytypic function is defined by induction on the structure of types. In general, the definition takes the following form.

$$\begin{array}{rcl}\mathit{poly}\langle t\rangle & :: & \tau(t) \\ \mathit{poly}\langle a_i\rangle & = & \mathit{poly}_{a_i} \\ \mathit{poly}\langle F(t_1, \ldots, t_n)\rangle & = & \mathit{poly}_F\ (\mathit{poly}\langle t_1\rangle, \ldots, \mathit{poly}\langle t_n\rangle)\end{array}$$

The type parameter is written in angle brackets to distinguish it from ordinary parameters. If t is an n-ary type constructor, poly_{a_i} must be specified for $1 \leqslant i \leqslant n$. Furthermore, an equation must be given for each primitive type constructor $F \in P$.

As an example, the function *flatten*, which listifies a given structure, can be defined as follows.

$$\begin{array}{lcl}
\mathit{flatten}\langle t\rangle & :: & \forall a.t\ a \to [\,a\,] \\
\mathit{flatten}\langle 1\rangle\ a & = & [\,] \\
\mathit{flatten}\langle \mathit{Int}\rangle\ a & = & [\,] \\
\mathit{flatten}\langle a\rangle\ a & = & [\,a\,] \\
\mathit{flatten}\langle t_1 + t_2\rangle\ (\mathit{Inl}\ a_1) & = & \mathit{flatten}\langle t_1\rangle\ a_1 \\
\mathit{flatten}\langle t_1 + t_2\rangle\ (\mathit{Inr}\ a_2) & = & \mathit{flatten}\langle t_2\rangle\ a_2 \\
\mathit{flatten}\langle t_1 \times t_2\rangle\ (a_2, a_2) & = & \mathit{flatten}\langle t_1\rangle\ a_1 +\!\!\!+ \mathit{flatten}\langle t_2\rangle\ a_2
\end{array}$$

The first two equations specify the action of *flatten* on nullary type constructors, ie $\mathit{flatten}_t = \lambda a \to [\,]$ for each $t \in \{1, \mathit{Int}, \ldots\}$. The third equation defines $\mathit{flatten}_a = \lambda a \to [\,a\,]$. Finally, $\mathit{flatten}_+$ and $\mathit{flatten}_\times$ are given by

$$\begin{array}{lcl}
\mathit{flatten}_+(\varphi_1, \varphi_2) & = & \lambda a \to \mathbf{case}\ a\ \mathbf{of}\ \{\mathit{Inl}\ a_1 \to \varphi_1\ a_1; \mathit{Inr}\ a_2 \to \varphi_2\ a_2\} \\
\mathit{flatten}_\times(\varphi_1, \varphi_2) & = & \lambda(a_1, a_2) \to \varphi_1\ a_1 +\!\!\!+ \varphi_2\ a_2\ .
\end{array}$$

This information is sufficient to define a unique function $\mathit{flatten}\langle t\rangle$ for each (unary) type expression t (Courcelle, 1983). Of course, since t may be infinite—and usually is—we require that types are interpreted by complete partial orders and functions by continuous functions between them. Both conditions are usually met.

The use of infinite type expressions as index sets for polytypic functions distinguishes our approach from previous ones (Jeuring & Jansson, 1996; Jansson & Jeuring, 1997), which are based on the initial algebra semantics of datatypes. Briefly, our approach has two major advantages: it is simpler (the programmer must consider less cases) and it is more general (it covers all first-order polymorphic datatypes). We refer the interested reader to (Hinze, 1999) for a more detailed account of the pros and cons.

2.3 Specializing polytypic definitions

The main purpose of a polytypic programming system is to specialize a polytypic function $\mathit{poly}\langle t\rangle$ for different instances of t. Unfortunately, the specialization cannot be based on the inductive definition of $\mathit{poly}\langle t\rangle$—at least, not directly. Consider the following attempt to specialize $\mathit{poly}\langle \mathit{Perfect}\ a\rangle$:

$$\begin{array}{ll}
 & \mathit{poly}\langle \mathit{Perfect}\ a\rangle \\
= & \mathit{poly}\langle a + \mathit{Perfect}\ (\mathit{Fork}\ a)\rangle \\
= & \mathit{poly}_+(\mathit{poly}_a, \mathit{poly}\langle \mathit{Perfect}\ (\mathit{Fork}\ a)\rangle) \\
= & \mathit{poly}_+(\mathit{poly}_a, \mathit{poly}\langle \mathit{Fork}\ a + \mathit{Perfect}\ (\mathit{Fork}\ (\mathit{Fork}\ a))\rangle) \\
= & \mathit{poly}_+(\mathit{poly}_a, \mathit{poly}_+(\mathit{poly}\langle \mathit{Fork}\ a\rangle, \mathit{poly}\langle \mathit{Perfect}\ (\mathit{Fork}\ (\mathit{Fork}\ a))\rangle)) \\
= & \ldots
\end{array}$$

To define $\mathit{poly}\langle \mathit{Perfect}\ a\rangle$ we require $\mathit{poly}\langle \mathit{Perfect}\ (\mathit{Fork}^n\ a)\rangle$ for each $n \geqslant 1$. It is probably clear that in general we cannot hope to obtain a *finite* representation of

$poly\langle t\rangle$ this way. Instead, we must base the specialization on the *representation of types*, ie on the datatype declarations themselves, which are by necessity finite.

To exhibit the structure of datatype declarations more clearly we shall rewrite them as *functor equations*. Functor expression of arity n are given by the following grammar.

$$F^n \quad = \quad \Pi_i^n \mid P^n \mid F^k \cdot (F_1^n, \ldots, F_k^n)$$

By Π_i^n we denote the n-ary projection functor selecting its i-th component. For $n = 1$ and $n = 2$ we use the following more familiar names: $Id = \Pi_1^1$, $Fst = \Pi_1^2$, and $Snd = \Pi_2^2$. Elements of P^n are predefined functors of arity n, ie $P^0 = \{1, Int, \ldots\}$ and $P^2 = \{+, \times\}$. The expression $F \cdot (F_1, \ldots, F_k)$ denotes the composition of an k-ary functor F with functors F_i, all of arity n. We omit the parenthesis when $n = 1$ and we write $K\,t$ instead of $t \cdot ()$ when $n = 0$. Finally, we write $f_1 + f_2$ for $+ \cdot (f_1, f_2)$ and similarly $f_1 \times f_2$.

Here are the datatype definitions of Section 2.1 rewritten as functor equations.

$$\begin{array}{lcl} List & = & K\,1 + Id \times List \\ Bintree & = & Fst + Bintree \times Snd \times Bintree \\ Fork & = & Id \times Id \\ Perfect & = & Id + Perfect \cdot Fork \\ Sequ & = & K\,1 + Sequ \cdot Fork + Id \times Sequ \cdot Fork \end{array}$$

In essence, functor equations are written in a compositional or 'point-free' style while **data** definitions are written in an applicative or 'pointwise' style.

Now, the central idea is to define, for each arity n, an n-ary function $poly_n\langle f\rangle$ satisfying

$$poly_n\langle f\rangle\ (poly\langle t_1\rangle, \ldots, poly\langle t_n\rangle) \quad = \quad poly\langle f(t_1, \ldots, t_n)\rangle\ .$$

We let function follow type: f is an n-ary functor mapping the types $t_1, \ldots, t_n$ to $f(t_1, \ldots, t_n)$; likewise $poly_n\langle f\rangle$ is an n-ary function mapping the polytypic functions $poly\langle t_1\rangle, \ldots, poly\langle t_n\rangle$ to $poly\langle f(t_1, \ldots, t_n)\rangle$. It can be shown that the following definition satisfies the condition above:

$$\begin{array}{rcl} poly_n\langle f\rangle & :: & \tau(t_1) \times \cdots \times \tau(t_n) \to \tau(f\,(t_1, \ldots, t_n)) \\ poly_n\langle \Pi_i\rangle & = & \pi_i \\ poly_n\langle F\rangle & = & poly_F \\ poly_n\langle f \cdot (g_1, \ldots, g_k)\rangle & = & poly_k\langle f\rangle \star (poly_n\langle g_1\rangle, \ldots, poly_n\langle g_k\rangle)\ . \end{array}$$

where $\pi_i\,(\varphi_1, \ldots, \varphi_n) = \varphi_i$ is the i-th projection function and '$\star$' denotes n-ary composition defined by $\varphi \star (\varphi_1, \ldots, \varphi_n) = \lambda a \to \varphi\,(\varphi_1\ a, \ldots, \varphi_n\ a)$. Note that $\varphi \star (\varphi_1) = \varphi \circ \varphi_1$ when $n = 1$. Furthermore note, that the definition of $poly_n\langle f\rangle$ is inductive on the structure of functor expressions. On a more abstract level we can view $poly_n$ as an interpretation of functor expressions: Π_i is interpreted by π_i, F by $poly_F$, and '$\cdot$' by '$\star$'.

Now, setting $t_i = a_i$ we can define *poly* in terms of poly_n.

$$\begin{array}{rcl}\mathit{poly}\langle f\rangle & :: & \tau(f)\\ \mathit{poly}\langle f(a_1,\ldots,a_n)\rangle & = & \mathit{poly}_n\langle f\rangle\ (\mathit{poly}_{a_1},\ldots,\mathit{poly}_{a_n})\end{array}$$

By now we have the necessary prerequisites at hand to define the specialization of a polytypic function $\mathit{poly}\langle f(a_1,\ldots,a_n)\rangle$ for a given instance of f. Assume that the type constructor is defined by the system of equations $\langle f_1 = e_1,\ldots,f_m = e_m\rangle$ with $f = f_i$ for some i. For each equation $f_i = e_i$, where f_i is an k-ary type constructor, a function definition of the form $\mathit{poly}_k\langle f_i\rangle = \mathit{poly}_k\langle e_i\rangle$ is generated. The expression $\mathit{poly}_k\langle e_i\rangle$ is given by the inductive definition above, additionally setting $\mathit{poly}_k\langle f_i\rangle = \mathit{poly\_k\_f}_i$, where $\mathit{poly\_k\_f}_i$ is a new function symbol. Finally, the defining equation for $\mathit{poly\_f}$, ie $\mathit{poly\_f} = \mathit{poly\_n\_f}\ (\mathit{poly}_{a_1},\ldots,\mathit{poly}_{a_n})$, must be added.

Let us apply the above framework to specialize $\mathit{flatten}\langle t\rangle$ for $t = \mathit{Perfect}$. Since $\mathit{flatten}\langle t\rangle$ has a polymorphic type, the auxiliary functions $\mathit{flatten}_n\langle f\rangle$ take polymorphic functions to polymorphic functions. We have, for instance,

$$\mathit{flatten}_1\langle f\rangle\ ::\ (\forall a.t\ a\to[a])\to(\forall a.f\ (t\ a)\to[a])\ .$$

In other words, $\mathit{flatten}_1\langle f\rangle$ has a rank-2 type signature (McCracken, 1984).

The specialization proceeds entirely mechanically. Using the original constructor names and abbreviating type names to their first letter we obtain

$$\begin{array}{lcl}\mathit{flattenP} & :: & \mathit{Perfect}\ a\to[a]\\ \mathit{flattenP} & = & \mathit{flatten1P}\ (\lambda a\to[a])\\ \mathit{flatten1F} & :: & (\forall a.t\ a\to[a])\to(\forall a.\mathit{Fork}\ (t\ a)\to[a])\\ \mathit{flatten1F}\ f\ (\mathit{Fork}\ a_1\ a_2) & = & f\ a_1 +\!\!+ f\ a_2\\ \mathit{flatten1P} & :: & (\forall a.t\ a\to[a])\to(\forall a.\mathit{Perfect}\ (t\ a)\to[a])\\ \mathit{flatten1P}\ f\ (\mathit{Null}\ a) & = & f\ a\\ \mathit{flatten1P}\ f\ (\mathit{Succ}\ a) & = & \mathit{flatten1P}\ (\mathit{flatten1F}\ f)\ a\ .\end{array}$$

Flattening a perfect tree operates in two stages: while recursing *flatten1P* constructs a tailor-made flattening function $\mathit{flatten1F}^i\ f$ of type $\forall a.\mathit{Fork}^i\ t\ a\to[a]$ which is eventually applied in the base case.

Remark. Unfortunately, the above definitions pass neither the Hugs nor the GHC type-checker though both accept rank-2 type signatures. The reason is that Haskell provides only a limited form of *type constructor polymorphism*. Consider the subexpression $\mathit{flatten1F}\ f$ in the last equation. It has type $\forall a.\mathit{Fork}\ (t\ a)\to[a]$ which is not unifiable with the expected type $\forall a.t'\ a\to[a]$. Since Haskell deliberately omits type abstractions from the language of type constructors (Jones, 1995), we cannot instantiate t' to $\lambda u\to\mathit{Fork}\ (t\ u)$. Fortunately, there is a way out of this dilemma. If we assign the following types to *flatten1F* and *flatten1P*

$$\begin{array}{lcl}\mathit{flatten1F} & :: & (v\to[w])\to(\mathit{Fork}\ v\to[w])\\ \mathit{flatten1P} & :: & (v\to[w])\to(\mathit{Perfect}\ v\to[w])\ ,\end{array}$$

the above definitions type-check. This trick works as long as the definition of $\mathit{poly}\langle t\rangle$

does not involve polymorphic recursion (in Section 3.2 we will get to know a polytypic function which is polymorphically recursive). □

3 Tries generically

In this section we apply the framework of polytypic programming to implement generalized tries generically for all first-order polymorphic datatypes. We have already mentioned the basic idea that generalized tries can be considered as a type-indexed datatype. To put this idea in concrete terms we will define a datatype

$$\mathit{Map}\langle * \rangle \quad :: \quad * \rightarrow * \ ,$$

which assigns a type constructor of kind $* \rightarrow *$ to each type constructor of kind $*$. The type $\mathit{Map}\langle k\rangle\ v$ represents the set of finite maps from k to v. Based on this representation we will implement the following operations.

$$\begin{array}{lcl}
\mathit{empty}\langle k\rangle & :: & \forall v.\mathit{Map}\langle k\rangle\ v \\
\mathit{single}\langle k\rangle & :: & \forall v.k \times v \rightarrow \mathit{Map}\langle k\rangle\ v \\
\mathit{lookup}\langle k\rangle & :: & \forall v.k \rightarrow \mathit{Map}\langle k\rangle\ v \rightarrow \mathit{Maybe}\ v \\
\mathit{insert}\langle k\rangle & :: & \forall v.(v \rightarrow v \rightarrow v) \rightarrow k \times v \rightarrow (\mathit{Map}\langle k\rangle\ v \rightarrow \mathit{Map}\langle k\rangle\ v) \\
\mathit{merge}\langle k\rangle & :: & \forall v.(v \rightarrow v \rightarrow v) \rightarrow (\mathit{Map}\langle k\rangle\ v \rightarrow \mathit{Map}\langle k\rangle\ v \rightarrow \mathit{Map}\langle k\rangle\ v)
\end{array}$$

The signature of $\mathit{lookup}\langle k\rangle$ deviates slightly from the one used in the introduction to this paper: the look-up function returns a value of type $\mathit{Maybe}\ v$ instead of v to be able to signal that a key is unbound. The functions $\mathit{insert}\langle k\rangle$ and $\mathit{merge}\langle k\rangle$ take as a first argument a so-called *combining function*, which is applied whenever two bindings have the same key. Typically, the combining form is fst or snd. For finite maps of type $\mathit{Map}\langle k\rangle\ \mathit{Int}$ addition may also be a sensible choice. Interestingly, we will see that the combining function is not only a convenient feature for the user, it is also necessary for defining $\mathit{insert}\langle k\rangle$ and $\mathit{merge}\langle k\rangle$ generically for all types!

3.1 Type-indexed tries

Mathematically speaking, generalized tries are based on the following isomorphisms, also known as laws of exponentials.

$$\begin{array}{rcl}
1 \rightarrow_{\mathrm{fin}} v & \cong & \mathit{Maybe}\ v \\
(k_1 + k_2) \rightarrow_{\mathrm{fin}} v & \cong & (k_1 \rightarrow_{\mathrm{fin}} v) \times (k_2 \rightarrow_{\mathrm{fin}} v) \\
(k_1 \times k_2) \rightarrow_{\mathrm{fin}} v & \cong & k_1 \rightarrow_{\mathrm{fin}} (k_2 \rightarrow_{\mathrm{fin}} v)
\end{array}$$

As $\mathit{Map}\langle k\rangle\ v$ represents the set of finite maps from k to v, ie $k \rightarrow_{\mathrm{fin}} v$, the isomorphisms above can be rewritten as defining equations for $\mathit{Map}\langle k\rangle$.

$$\begin{array}{lcl}
\mathit{Map}\langle 1\rangle & = & \mathit{Maybe} \\
\mathit{Map}\langle \mathit{Int}\rangle & = & \mathit{Patricia.Dict} \\
\mathit{Map}\langle k_1 + k_2\rangle & = & \mathit{Map}\langle k_1\rangle \times \mathit{Map}\langle k_2\rangle \\
\mathit{Map}\langle k_1 \times k_2\rangle & = & \mathit{Map}\langle k_1\rangle \cdot \mathit{Map}\langle k_2\rangle
\end{array}$$

We assume the existence of 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* (Okasaki & Gill, 1998). 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*.

Building upon the techniques developed in Section 2.3 we can now specialize $Map\langle k\rangle$ for a given instance of k. That is, for each functor f of arity n we will define an n-ary *higher-order functor* $Map_n\langle f\rangle$. For $n = 1$ we have, for instance,

$$Map_1\langle * \to *\rangle \quad :: \quad (* \to *) \to (* \to *) \ .$$

The type constructor $Map_1\langle f\rangle$ is *the generalized trie of a polymorphic datatype*. It takes as argument the generalized trie of the base type, say, a and yields the generalized trie of $f\ a$. It may come as a surprise that the framework for specializing type-indexed functions is also applicable to type-indexed datatypes. The reason is quite simple: the definition of $poly_n\langle f\rangle$ requires only two operations, namely projection and composition. However, both operations are also available in the world of functors and higher-order functors.

Let us specialize $Map_n\langle f\rangle$ to the datatypes listed in Section 2.1. For better readability we abbreviate type names to their first letter and omit the arity of functors, ie we write *MapL* instead of *Map1List*.

$$\begin{array}{lcl}
MapL\ m & = & Maybe \times m \cdot MapL\ m \\
MapB\ (m_1, m_2) & = & m_1 \times MapB\ (m_1, m_2) \cdot m_2 \cdot MapB\ (m_1, m_2) \\
MapF\ m & = & m \cdot m \\
MapP\ m & = & m \times MapP\ (MapF\ m) \\
MapS\ m & = & Maybe \times MapS\ (MapF\ m) \times m \cdot MapS\ (MapF\ m)
\end{array}$$

Since Haskell permits the definition of higher-order polymorphic datatypes, the higher-order functors above can be directly coded as datatypes. All we have to do is to bring the equations into an applicative form.

```
data MapL m v       =  TrieL (Maybe v) (m (MapL m v))
data MapB m1 m2 v   =  TrieB (m1 v)
                              (MapB m1 m2 (m2 (MapB m1 m2 v)))
```

These types are the polymorphic variants of *MapStr* and *MapBin* defined in the introduction, ie we have $MapStr \cong MapL\ MapChar$ (since $Str \cong List\ Char$) and $MapBin \cong MapB\ MapStr\ MapChar$ (since $Bin \cong Bintree\ Str\ Char$). Things become interesting if we consider nested datatypes.

$$\begin{array}{lcl}
\textbf{data}\ \mathit{MapF}\ m\ v & = & \mathit{TrieF}\ (m\ (m\ v)) \\
\textbf{data}\ \mathit{MapP}\ m\ v & = & \mathit{TrieP}\ (m\ v) \\
 & & \qquad (\mathit{MapP}\ (\mathit{MapF}\ m)\ v) \\
\textbf{data}\ \mathit{MapS}\ m\ v & = & \mathit{TrieS}\ (\mathit{Maybe}\ v) \\
 & & \qquad (\mathit{MapS}\ (\mathit{MapF}\ m)\ v) \\
 & & \qquad (m\ (\mathit{MapS}\ (\mathit{MapF}\ m)\ v))
\end{array}$$

The generalized trie of a nested datatype is a higher-order polymorphic nested datatype! The nest is higher-order polymorphic since the type parameter which is instantiated in a recursive call ranges over type constructors of kind $* \to *$. By contrast, *MapB* is a first-order polymorphic nest since its instantiated type parameter has kind $*$. It is quite easy to produce generalized tries which are both first- and higher-order nests. If we change the type of *Sequ*'s third constructor to $\mathit{One}\ (\mathit{Sequ}\ (\mathit{Fork}\ a))\ a$, then the third component of *TrieS* has type $\mathit{MapS}\ (\mathit{MapF}\ m)\ (m\ v)$ and *MapS* is consequently both a first- and a higher-order nest.

3.2 Empty and singleton tries

The empty trie is given by

$$\begin{array}{lcl}
\mathit{empty}\langle k\rangle & :: & \forall v.\mathit{Map}\langle k\rangle\ v \\
\mathit{empty}\langle 1\rangle & = & \mathit{Nothing} \\
\mathit{empty}\langle \mathit{Int}\rangle & = & \mathit{Patricia.empty} \\
\mathit{empty}\langle k_1 + k_2\rangle & = & (\mathit{empty}\langle k_1\rangle, \mathit{empty}\langle k_2\rangle) \\
\mathit{empty}\langle k_1 \times k_2\rangle & = & \mathit{empty}\langle k_1\rangle\ .
\end{array}$$

The definition already illustrates several interesting aspects of programming with generalized tries. To begin with the polymorphic type of $\mathit{empty}\langle k\rangle$ is necessary to make the definition work. Consider the last equation: $\mathit{empty}\langle k_1 \times k_2\rangle$, which is of type $\forall v.\mathit{Map}\langle k_1\rangle\ (\mathit{Map}\langle k_2\rangle\ v)$, is defined in terms of $\mathit{empty}\langle k_1\rangle$, which is of type $\forall v.\mathit{Map}\langle k_1\rangle\ v$. That means that $\mathit{empty}\langle k_1\rangle$ is used polymorphically. In other words $\mathit{empty}\langle k\rangle$ makes use of polymorphic recursion! By contrast, the definition of $\mathit{flatten}\langle t\rangle$ given in Section 2.2 also type-checks when the type is restricted to $t\ s \to [s]$ for some fixed type s.

Since $\mathit{empty}\langle k\rangle$ has a polymorphic type, $\mathit{empty}_n\langle f\rangle$ takes polymorphic values to polymorphic values. We have, for instance,

$$\mathit{empty}_1\langle f\rangle \quad :: \quad (\forall v.\mathit{Map}\langle k\rangle\ v) \to (\forall v.\mathit{Map}\langle f\ k\rangle\ v)$$

To obtain a signature which is expressible in Haskell we employ the specification of $\mathit{Map}_n\langle f\rangle$, ie $\mathit{Map}\langle f(k_1,\ldots,k_n)\rangle = \mathit{Map}_n\langle f\rangle\ (\mathit{Map}\langle k_1\rangle,\ldots,\mathit{Map}\langle k_n\rangle)$, additionally setting $\mathit{Map}\langle k\rangle = m$ where m is a fresh type variable.

$$\mathit{empty}_1\langle f\rangle \quad :: \quad (\forall v.m\ v) \to (\forall v.\mathit{Map}_1\langle f\rangle\ m\ v)$$

Let us take a look at some examples.[1]

$$\begin{array}{lcl}
\mathit{emptyL} & :: & (\forall v.m\ v) \to \mathit{MapL}\ m\ w \\
\mathit{emptyL}\ e & = & \mathit{TrieL}\ \mathit{Nothing}\ e \\
\mathit{emptyF} & :: & (\forall v.m\ v) \to \mathit{MapF}\ m\ w \\
\mathit{emptyF}\ e & = & \mathit{TrieF}\ e \\
\mathit{emptyP} & :: & (\forall v.m\ v) \to \mathit{MapP}\ m\ w \\
\mathit{emptyP}\ e & = & \mathit{TrieP}\ e\ (\mathit{emptyP}\ (\mathit{emptyF}\ e))
\end{array}$$

The second function, *emptyF*, illustrates the polymorphic use of the parameter: e has type $\forall v.m\ v$ but is used as an element of $m\ (m\ w)$. The last definition employs 'higher-order polymorphic' recursion: the recursive call is of type $(\forall v.\mathit{MapF}\ m\ v) \to \mathit{MapP}\ (\mathit{MapF}\ m)\ w$ which is a substitution instance of the declared type. The function *emptyP* illustrates another point: the implementation of generalized tries relies in an essential way on lazy evaluation. As an example, consider the empty trie for *Perfect Int*, which is represented by the infinite tree (abbreviating *Patricia.empty* to e)

$$\mathit{TrieP}\ e\ (\mathit{TrieP}\ (\mathit{TrieF}\ e)\ (\mathit{TrieP}\ (\mathit{TrieF}\ (\mathit{TrieF}\ e))\ \ldots))\ .$$

In Section 4.1 we shall discuss a slightly modified representation of generalized tries which avoids this problem.

The singleton trie which contains only a single binding is defined as follows.

$$\begin{array}{lcl}
\mathit{single}\langle k\rangle & :: & \forall v.k \times v \to \mathit{Map}\langle k\rangle\ v \\
\mathit{single}\langle 1\rangle\ ((), v) & = & \mathit{Just}\ v \\
\mathit{single}\langle \mathit{Int}\rangle\ (i, v) & = & \mathit{Patricia.single}\ (i, v) \\
\mathit{single}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1, v) & = & (\mathit{single}\langle k_1\rangle\ (i_1, v), \mathit{empty}\langle k_2\rangle) \\
\mathit{single}\langle k_1 + k_2\rangle\ (\mathit{Inr}\ i_2, v) & = & (\mathit{empty}\langle k_1\rangle, \mathit{single}\langle k_2\rangle\ (i_2, v)) \\
\mathit{single}\langle k_1 \times k_2\rangle\ ((i_1, i_2), v) & = & \mathit{single}\langle k_1\rangle\ (i_1, \mathit{single}\langle k_2\rangle\ (i_2, v))
\end{array}$$

The definition of $\mathit{single}\langle k\rangle$ is interesting because it falls back on $\mathit{empty}\langle k\rangle$ in the third and the fourth equation. This necessitates that $\mathit{single}_n\langle f\rangle$ is parameterised both with $\mathit{single}\langle k\rangle$ and with $\mathit{empty}\langle k\rangle$. For $n = 1$ we obtain the type signature

$$\begin{array}{lcl}
\mathit{single}_1\langle f\rangle & :: & (\forall v.m\ v) \\
 & \to & (\forall v.k \times v \to m\ v) \\
 & \to & (\forall v.f\ k \times v \to \mathit{Map}_1\langle f\rangle\ m\ v)\ .
\end{array}$$

Let us again specialize the polytypic function to lists and perfect trees. To improve readability we will henceforth present the instances without their type signatures—which are, nonetheless, mandatory.

$$\begin{array}{lcl}
\mathit{singleL}\ e\ s\ (\mathit{Nil}, v) & = & \mathit{TrieL}\ (\mathit{Just}\ v)\ e \\
\mathit{singleL}\ e\ s\ (\mathit{Cons}\ i\ \mathit{is}, v) & = & \mathit{TrieL}\ \mathit{Nothing}\ (s\ (i, \mathit{singleL}\ e\ s\ (\mathit{is}, v))) \\
\mathit{singleF}\ e\ s\ (\mathit{Fork}\ i_1\ i_2, v) & = & \mathit{TrieF}\ (s\ (i_1, s\ (i_2, v)))
\end{array}$$

[1] Note that we use Hugs/GHC syntax for universal quantification (Peyton Jones, 1998), which forces us to write $(\forall v.m\ v) \to \mathit{MapL}\ m\ w$ instead of $(\forall v.m\ v) \to (\forall v.\mathit{MapL}\ m\ v)$.

$$\begin{aligned}
\mathit{singleP}\ e\ s\ (\mathit{Null}\ i, v) &= \mathit{TrieP}\ (s\ (i, v))\ (\mathit{emptyP}\ (\mathit{emptyF}\ e)) \\
\mathit{singleP}\ e\ s\ (\mathit{Succ}\ i, v) &= \mathit{TrieP}\ e\ (\mathit{singleP}\ (\mathit{emptyF}\ e)\ (\mathit{singleF}\ e\ s)\ (i, v))
\end{aligned}$$

The function *singleF* illustrates that the 'mechanically' generated definitions can sometimes be slightly improved. Since the definition of *Fork* does not involve sums, *singleF* does not require its first argument which could be safely removed.

3.3 Lookup

The look-up function implements the scheme discussed in the introduction.

$$\begin{aligned}
&\mathit{lookup}\langle k\rangle && :: && \forall v.k \to \mathit{Map}\langle k\rangle\ v \to \mathit{Maybe}\ v \\
&\mathit{lookup}\langle 1\rangle\ ()\ t && = && t \\
&\mathit{lookup}\langle \mathit{Int}\rangle\ i\ t && = && \mathit{Patricia.lookup}\ i\ t \\
&\mathit{lookup}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1)\ (t_1, t_2) && = && \mathit{lookup}\langle k_1\rangle\ i_1\ t_1 \\
&\mathit{lookup}\langle k_1 + k_2\rangle\ (\mathit{Inr}\ i_2)\ (t_1, t_2) && = && \mathit{lookup}\langle k_2\rangle\ i_2\ t_2 \\
&\mathit{lookup}\langle k_1 \times k_2\rangle\ (i_1, i_2)\ t_1 && = && \mathbf{do}\ \{ t_2 \leftarrow \mathit{lookup}\langle k_1\rangle\ i_1\ t_1; \mathit{lookup}\langle k_2\rangle\ i_2\ t_2 \}
\end{aligned}$$

On sums the look-up function selects the appropriate map; on products it 'composes' the look-up functions for the components. Since $\mathit{lookup}\langle k\rangle$ has result type *Maybe v*, composition amounts to monad or *Kleisli composition* (Bird, 1998). Defining

$$(m_1 \diamond m_2)\ a_1 \quad = \quad \mathbf{do}\ \{ a_2 \leftarrow m_1\ a_1; m_2\ a_2 \}$$

we may write the last equation more succinctly as

$$\mathit{lookup}\langle k_1 \times k_2\rangle\ (i_1, i_2) \quad = \quad \mathit{lookup}\langle k_1\rangle\ i_1 \diamond \mathit{lookup}\langle k_2\rangle\ i_2 \ .$$

Specializing $\mathit{lookup}\langle k\rangle$ to concrete instances of k is by now probably a matter of routine. Here is $\mathit{lookup}_1\langle f\rangle$'s type signature.

$$\begin{aligned}
\mathit{lookup}_1\langle f\rangle \quad &:: \quad (\forall v.k \to m\ v \to \mathit{Maybe}\ v) \\
&\to \quad (\forall v.f\ k \to \mathit{Map}_1\langle f\rangle\ m\ v \to \mathit{Maybe}\ v) \ .
\end{aligned}$$

For lists and perfect trees we obtain

$$\begin{aligned}
\mathit{lookupL}\ \ell\ \mathit{Nil}\ (\mathit{TrieL}\ \mathit{tn}\ \mathit{tc}) &= \mathit{tn} \\
\mathit{lookupL}\ \ell\ (\mathit{Cons}\ i\ \mathit{is})\ (\mathit{TrieL}\ \mathit{tn}\ \mathit{tc}) &= (\mathit{lookupL}\ \ell\ \mathit{is} \diamond \ell\ i)\ \mathit{tc} \\
\mathit{lookupF}\ \ell\ (\mathit{Fork}\ i_1\ i_2)\ (\mathit{TrieF}\ \mathit{tf}) &= (\ell\ i_2 \diamond \ell\ i_1)\ \mathit{tf} \\
\mathit{lookupP}\ \ell\ (\mathit{Null}\ i)\ (\mathit{TrieP}\ \mathit{ts}\ \mathit{tc}) &= \ell\ i\ \mathit{ts} \\
\mathit{lookupP}\ \ell\ (\mathit{Succ}\ i)\ (\mathit{TrieP}\ \mathit{ts}\ \mathit{tc}) &= \mathit{lookupP}\ (\mathit{lookupF}\ \ell)\ i\ \mathit{tc} \ .
\end{aligned}$$

Note that *lookupL* generalizes *lookupStr* defined in the introduction to this paper; we have $\mathit{lookupStr}\ s \cong \mathit{fromJust} \circ \mathit{lookupL}\ \mathit{lookupChar}\ s$ where *fromJust* is given by $\mathit{fromJust}\ (\mathit{Just}\ a) = a$. The definition of *lookupP* employs the same recursion scheme as *flatten1P*: while recursing, *lookupP* constructs a tailor-made look-up function $\mathit{lookupF}^i\ \ell$ of type $\mathit{Fork}^i\ k \to \mathit{MapF}^i\ w \to \mathit{Maybe}\ w$ which is finally applied in the base case.

3.4 Inserting and merging

Insertion is defined in terms of $\mathit{merge}\langle k\rangle$ and $\mathit{single}\langle k\rangle$.

$$\begin{array}{lcl}
\mathit{insert}\langle k\rangle & :: & \forall v.(v \to v \to v) \to k \times v \to (\mathit{Map}\langle k\rangle\ v \to \mathit{Map}\langle k\rangle\ v) \\
\mathit{insert}\langle k\rangle\ c\ (i, v)\ t & = & \mathit{merge}\langle k\rangle\ c\ (\mathit{single}\langle k\rangle\ (i, v))\ t
\end{array}$$

Note that this is not the most efficient implementation of $\mathit{insert}\langle k\rangle$ since singleton tries are in general given by infinite trees. This implies that the running time of $\mathit{insert}\langle k\rangle$ is *not* proportional to the size of the inserted element as one would expect. The problem vanishes, however, if we employ the alternative representation of generalized tries to be introduced in Section 4.1.

Merging two tries is surprisingly simple. Given an auxiliary function for combining two values of type $\mathit{Maybe}\ a$

$$\begin{array}{lcl}
\mathit{combine} & :: & (a \to a \to a) \\
 & \to & (\mathit{Maybe}\ a \to \mathit{Maybe}\ a \to \mathit{Maybe}\ a) \\
\mathit{combine}\ c\ \mathit{Nothing}\ \mathit{Nothing} & = & \mathit{Nothing} \\
\mathit{combine}\ c\ \mathit{Nothing}\ (\mathit{Just}\ a') & = & \mathit{Just}\ a' \\
\mathit{combine}\ c\ (\mathit{Just}\ a)\ \mathit{Nothing} & = & \mathit{Just}\ a \\
\mathit{combine}\ c\ (\mathit{Just}\ a)\ (\mathit{Just}\ a') & = & \mathit{Just}\ (c\ a\ a')\ ,
\end{array}$$

we can define $\mathit{merge}\langle k\rangle$ as follows.

$$\begin{array}{lcl}
\mathit{merge}\langle k\rangle & :: & \forall v.(v \to v \to v) \\
 & \to & (\mathit{Map}\langle k\rangle\ v \to \mathit{Map}\langle k\rangle\ v \to \mathit{Map}\langle k\rangle\ v) \\
\mathit{merge}\langle 1\rangle\ c\ t\ t' & = & \mathit{combine}\ c\ t\ t' \\
\mathit{merge}\langle \mathit{Int}\rangle\ c\ t\ t' & = & \mathit{Patricia.merge}\ c\ t\ t' \\
\mathit{merge}\langle k_1 + k_2\rangle\ c\ (t_1, t_2)\ (t_1', t_2') & = & (\mathit{merge}\langle k_1\rangle\ c\ t_1\ t_1', \mathit{merge}\langle k_2\rangle\ c\ t_2\ t_2') \\
\mathit{merge}\langle k_1 \times k_2\rangle\ c\ t\ t' & = & \mathit{merge}\langle k_1\rangle\ (\mathit{merge}\langle k_2\rangle\ c)\ t\ t'
\end{array}$$

The most interesting equation is the last one. The tries t and t' are of type $\mathit{Map}\langle k_1 \times k_2\rangle\ v = \mathit{Map}\langle k_1\rangle\ (\mathit{Map}\langle k_2\rangle\ v)$. To merge them we can use $\mathit{merge}\langle k_1\rangle$; we must, however, supply a combining function of type $\mathit{Map}\langle k_2\rangle\ v \to \mathit{Map}\langle k_2\rangle\ v \to \mathit{Map}\langle k_2\rangle\ v$. A moment's reflection reveals that that $\mathit{merge}\langle k_2\rangle\ c$ is the desired combining function. Using functional composition we can write the last equation quite succinctly as

$$\mathit{merge}\langle k_1 \times k_2\rangle \quad = \quad \mathit{merge}\langle k_1\rangle \circ \mathit{merge}\langle k_2\rangle\ .$$

The definition of $\mathit{merge}\langle k\rangle$ shows that it is sometimes necessary to implement operations more general than actually needed. If $\mathit{merge}\langle k\rangle$ had the simplified type $\mathit{Map}\langle k\rangle\ v \to \mathit{Map}\langle k\rangle\ v \to \mathit{Map}\langle k\rangle\ v$, then we would not be able to give a defining equation for $k = k_1 \times k_2$.

To complete the picture let us again specialize the merging operation for lists and perfect trees. To begin with here is $\mathit{merge}_1\langle f\rangle$'s type signature.

$$\begin{array}{lcl}
\mathit{merge}_1\langle f\rangle & :: & (\forall v.(v \to v \to v) \to (m\ v \to m\ v \to m\ v)) \\
 & \to & (\forall v.(v \to v \to v) \to (\mathit{Map}_1\langle f\rangle\ m\ v \to \mathit{Map}_1\langle f\rangle\ m\ v \to \mathit{Map}_1\langle f\rangle\ m\ v))
\end{array}$$

The different instances of $\mathit{merge}_1\langle f\rangle$ are surprisingly concise.

$$\begin{aligned}
&\mathit{mergeL}\ m\ c\ (\mathit{TrieL}\ \mathit{tn}_1\ \mathit{tc}_1)\ (\mathit{TrieL}\ \mathit{tn}_2\ \mathit{tc}_2)\\
&\quad =\ \mathit{TrieL}\ (\mathit{combine}\ c\ \mathit{tn}_1\ \mathit{tn}_2)\ (m\ (\mathit{mergeL}\ m\ c)\ \mathit{tc}_1\ \mathit{tc}_2)\\
&\mathit{mergeF}\ m\ c\ (\mathit{TrieF}\ \mathit{tf}_1)\ (\mathit{TrieF}\ \mathit{tf}_2)\\
&\quad =\ \mathit{TrieF}\ (m\ (m\ c)\ \mathit{tf}_1\ \mathit{tf}_2)\\
&\mathit{mergeP}\ m\ c\ (\mathit{TrieP}\ \mathit{ts}_1\ \mathit{tc}_1)\ (\mathit{TrieP}\ \mathit{ts}_2\ \mathit{tc}_2)\\
&\quad =\ \mathit{TrieP}\ (m\ c\ \mathit{ts}_1\ \mathit{ts}_2)\ (\mathit{mergeP}\ (\mathit{mergeF}\ m)\ c\ \mathit{tc}_1\ \mathit{tc}_2)
\end{aligned}$$

4 Variations of the theme

4.1 Spotted tries

The representation of tries as defined in the previous section has two major drawbacks: (i) it relies in an essential way on lazy evaluation and (ii) it is inefficient. Both disadvantages have their roots in the representation of tries on sums. A trie on $k_1 + k_2$ is a pair of tries irrespective of whether the trie is empty or not. This suggests to devise a special representation for the empty trie. Technically, this is achieved using so-called *spot products* (Connelly & Lockwood Morris, 1995).

$$\textbf{data}\ a_1 \times_\bullet a_2 \quad = \quad \mathit{Spot} \mid \mathit{Pair}\ a_1\ a_2$$

Spot products are also known as *optional pairs*. Changing $\mathit{Map}\langle k\rangle$'s definition to

$$\mathit{Map}\langle k_1 + k_2\rangle \quad = \quad \mathit{Map}\langle k_1\rangle \times_\bullet \mathit{Map}\langle k_2\rangle$$

we can now represent the empty trie in constant space.

$$\mathit{empty}\langle k_1 + k_2\rangle \quad = \quad \mathit{Spot}$$

This representation is, of course, no longer unique. Therefore, we require that the empty trie on sums is always represented by *Spot*. Maintaining this invariant in our implementation is, however, trivial since tries never shrink. The situation would be different if we additionally supplied an operation for removing bindings from a trie.

The remaining operations must be modified accordingly.

$$\begin{aligned}
&\mathit{single}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1, v) && = && \mathit{Pair}\ (\mathit{single}\langle k_1\rangle\ (i_1, v))\ (\mathit{empty}\langle k_2\rangle)\\
&\mathit{single}\langle k_1 + k_2\rangle\ (\mathit{Inr}\ i_2, v) && = && \mathit{Pair}\ (\mathit{empty}\langle k_1\rangle)\ (\mathit{single}\langle k_2\rangle\ (i_2, v))\\
&\mathit{lookup}\langle k_1 + k_2\rangle\ k\ \mathit{Spot} && = && \mathit{Nothing}\\
&\mathit{lookup}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1)\ (\mathit{Pair}\ t_1\ t_2) && = && \mathit{lookup}\langle k_1\rangle\ i_1\ t_1\\
&\mathit{lookup}\langle k_1 + k_2\rangle\ (\mathit{Inr}\ i_2)\ (\mathit{Pair}\ t_1\ t_2) && = && \mathit{lookup}\langle k_2\rangle\ i_2\ t_2\\
&\mathit{merge}\langle k_1 + k_2\rangle\ c\ \mathit{Spot}\ t' && = && t'\\
&\mathit{merge}\langle k_1 + k_2\rangle\ c\ t\ \mathit{Spot} && = && t\\
&\mathit{merge}\langle k_1 + k_2\rangle\ c\ (\mathit{Pair}\ t_1\ t_2)\ (\mathit{Pair}\ t_1'\ t_2') && && \\
& && = && \mathit{Pair}\ (\mathit{merge}\langle k_1\rangle\ c\ t_1\ t_1')\ (\mathit{merge}\langle k_2\rangle\ c\ t_2\ t_2')
\end{aligned}$$

Figure 2 contains the complete code for generalized tries on binary random-access lists building on the above representation. Some remarks are appropriate. First of

```
{- Generalized tries for polymorphic binary random-access lists -}
data MapS m v           =   SpotS | TrieS (Maybe v)
                                          (MapS (MapF m) v)
                                          (m (MapS (MapF m) v))
emptyS                  ::  MapS m w
emptyS                  =   SpotS
singleS                 ::  (∀v.m v)
                        →   (∀v.k × v → m v)
                        →   Sequ k × w → MapS m w
singleS e s (Empty, v)  =   TrieS (Just v) emptyS e
singleS e s (Zero x, v) =   TrieS Nothing
                                  (singleS (emptyF e) (singleF e s) (x, v))
                                  e
singleS e s (One i x, v) =  TrieS Nothing
                                  emptyS
                                  (s (i, singleS (emptyF e) (singleF e s) (x, v)))
lookupS                 ::  (∀v.k → m v → Maybe v)
                        →   Sequ k → MapS m w → Maybe w
lookupS ℓ Empty (TrieS te tz to)
                        =   te
lookupS ℓ (Zero x) (TrieS te tz to)
                        =   lookupS (lookupF ℓ) x tz
lookupS ℓ (One i x) (TrieS te tz to)
                        =   (lookupS (lookupF ℓ) x ◇ ℓ i) to
mergeS                  ::  (∀v.(v → v → v) → (m v → m v → m v))
                        →   (w → w → w) → (MapS m w → MapS m w → MapS m w)
mergeS m c SpotS t'     =   t'
mergeS m c t SpotS      =   t
mergeS m c (TrieS te₁ tz₁ to₁) (TrieS te₂ tz₂ to₂)
                        =   TrieS (combine c te₁ te₂)
                                  (mergeS (mergeF m) c tz₁ tz₂)
                                  (m (mergeS (mergeF m) c) to₁ to₂)
{- Generalized tries for binary random-access lists over integers -}
type MapSI              =   MapS Patricia.Dict
emptySI                 ::  MapSI v
emptySI                 =   emptyS
singleSI                ::  Sequ Int × v → MapSI v
singleSI                =   singleS Patricia.empty Patricia.single
lookupSI                ::  Sequ Int → MapSI w → Maybe w
lookupSI                =   lookupS Patricia.lookup
insertSI                ::  (v → v → v) → Sequ Int × v → (MapSI v → MapSI v)
insertSI c b t          =   mergeSI c (singleSI b) t
mergeSI                 ::  (v → v → v) → (MapSI v → MapSI v → MapSI v)
mergeSI                 =   mergeS Patricia.merge
```

Fig. 2. Generalized tries for binary random-access lists.

all, the datatype *MapS* is based on the functor equation

$$\mathit{MapS}\ m \quad = \quad \mathit{Maybe} \times_\bullet \mathit{MapS}\ (\mathit{MapF}\ m) \times_\bullet m \cdot \mathit{MapS}\ (\mathit{MapF}\ m)\ .$$

For simplicity we interpret $a_1 \times_\bullet a_2 \times_\bullet a_3$ as the type of optional triples and not as nested optional pairs.

$$\mathbf{data}\ a_1 \times_\bullet a_2 \times_\bullet a_3 \quad = \quad \mathit{Spot} \mid \mathit{Triple}\ a_1\ a_2\ a_3$$

The definition of *emptyS* has been simplified by omitting its parameter, which is not required. Finally, note that we have not listed the implementation of generalized tries for the datatype *Fork*. Since *Fork*'s definition does not involve sums, the code is identical to that given in Section 3.

4.2 Skinny tries

Extending the idea of the previous section one step further we could additionally devise a special representation for singleton tries.

$$\mathbf{data}\ a_1 {}_\bullet\!\times_\bullet a_2 \quad = \quad \mathit{None} \mid \mathit{Onlyl}\ a_1 \mid \mathit{Onlyr}\ a_2 \mid \mathit{Both}\ a_1\ a_2$$

Using ${}_\bullet\!\times_\bullet$ instead of $\times_\bullet$ has the advantage that $\mathit{single}\langle k\rangle$ need not refer to $\mathit{empty}\langle k\rangle$.

$$\begin{aligned}
\mathit{single}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1, v) &\quad = \quad \mathit{Onlyl}\ (\mathit{single}\langle k_1\rangle\ (i_1, v))\\
\mathit{single}\langle k_1 + k_2\rangle\ (\mathit{Inr}\ i_2, v) &\quad = \quad \mathit{Onlyr}\ (\mathit{single}\langle k_2\rangle\ (i_2, v))
\end{aligned}$$

This representation is furthermore a bit more space economical. A potential disadvantage is the increased number of cases one must consider when defining $\mathit{lookup}\langle k\rangle$ and $\mathit{merge}\langle k\rangle$.

$$\begin{aligned}
\mathit{lookup}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1)\ \mathit{None} &\quad = \quad \mathit{Nothing}\\
\mathit{lookup}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1)\ (\mathit{Onlyl}\ t_1) &\quad = \quad \mathit{lookup}\langle k_1\rangle\ i_1\ t_1\\
\mathit{lookup}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1)\ (\mathit{Onlyr}\ t_2) &\quad = \quad \mathit{Nothing}\\
\mathit{lookup}\langle k_1 + k_2\rangle\ (\mathit{Inl}\ i_1)\ (\mathit{Both}\ t_1\ t_2) &\quad = \quad \mathit{lookup}\langle k_1\rangle\ i_1\ t_1\\
\mathit{merge}\langle k_1 + k_2\rangle\ c\ (\mathit{Onlyl}\ t_1)\ \mathit{None} &\quad = \quad \mathit{Onlyl}\ t_1\\
\mathit{merge}\langle k_1 + k_2\rangle\ c\ (\mathit{Onlyl}\ t_1)\ (\mathit{Onlyl}\ t_1') &\quad = \quad \mathit{Onlyl}\ (\mathit{merge}\langle k_1\rangle\ c\ t_1\ t_1')\\
\mathit{merge}\langle k_1 + k_2\rangle\ c\ (\mathit{Onlyl}\ t_1)\ (\mathit{Onlyr}\ t_2') &\quad = \quad \mathit{Both}\ t_1\ t_2'\\
\mathit{merge}\langle k_1 + k_2\rangle\ c\ (\mathit{Onlyl}\ t_1)\ (\mathit{Both}\ t_1'\ t_2') &\quad = \quad \mathit{Both}\ (\mathit{merge}\langle k_1\rangle\ c\ t_1\ t_1')\ t_2'
\end{aligned}$$

The remaining cases are defined accordingly.

5 Related and future work

D.E. Knuth (1998) attributes the idea of a trie to A. Thue who introduced it in a paper about strings that do not contain adjacent repeated substrings. R. de la Briandais recommended tries for computer searching (1959). The generalization of tries from strings to elements of an arbitrary datatype was discovered by C.P. Wadsworth (1979) and others independently since. R.H. Connelly and F.L. Morris (1995) 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. Interestingly, despite the framework of category theory they base the development on many-sorted signatures which makes the definitions somewhat unwieldy. This paper shows that the construction of generalized tries is much simpler if we replace to concept of a many-sorted signature by its categorical counterpart, the concept of a functor.

The first implementation of generalized tries was given by C. Okasaki in his recent textbook on functional data structures (1998). Tries for polymorphic types like lists or binary trees are represented as Standard ML functors. While this approach works for regular datatypes it fails for nested datatypes such as *Perfect* or *Sequ*. In the latter case higher-order polymorphic datatypes are indispensable.

That said, a direction for future work suggests itself, namely to generalize tries to arbitrary *higher-order polymorphic datatypes*. To give an impression of the extensions consider the standard definition of rose trees.

$$\mathbf{data}\ \mathit{Rose}\ k \quad = \quad \mathit{Branch}\ k\ (\mathit{List}\ (\mathit{Rose}\ k))$$

Its trie is given by

$$\mathbf{data}\ \mathit{MapR}\ \mathit{mk}\ v \quad = \quad \mathit{TrieR}\ (\mathit{mk}\ (\mathit{MapL}\ (\mathit{MapR}\ \mathit{mk})\ v))\ .$$

Now, abstracting the list functor away we obtain the following generalization of rose trees.

$$\mathbf{data}\ \mathit{GRose}\ t\ k \quad = \quad \mathit{GBranch}\ k\ (t\ (\mathit{GRose}\ t\ k))$$

The trie of *Rose* can be generalized in a similar way.

$$\mathbf{data}\ \mathit{MapGR}\ \mathit{mt}\ \mathit{mk}\ v \quad = \quad \mathit{TrieGR}\ (\mathit{mk}\ (\mathit{mt}\ (\mathit{MapGR}\ \mathit{mt}\ \mathit{mk})\ v))$$

Note that *GRose* is a type constructor of kind $(* \to *) \to (* \to *)$ while its trie has kind $((* \to *) \to (* \to *)) \to ((* \to *) \to (* \to *))$. Now, the same systematics can be applied to generalize the operations on *MapR* to operations on *MapGR*. Currently, the author is working on a suitable extension of the framework which allows to define polytypic functions generically for all datatypes expressible in Haskell.

6 Acknowledgement

Thanks are due to Chris Okasaki for his helpful comments on an earlier draft of this paper.

References

Bird, Richard. (1998). *Introduction to functional programming using Haskell.* 2nd edn. London: Prentice Hall Europe.

Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. *Pages 52–67 of:* Jeuring, J. (ed), *Fourth international conference on mathematics of program construction, MPC'98, Marstrand, Sweden.* Lecture Notes in Computer Science, vol. 1422. Springer Verlag.

Bird, Richard, & Paterson, Ross. (1998). De Bruijn notation as a nested datatype. *Journal of functional programming.* To appear.

Connelly, Richard H., & Lockwood Morris, F. (1995). A generalization of the trie data structure. *Mathematical structures in computer science*, **5**(3), 381–418.

Courcelle, Bruno. (1983). Fundamental properties of infinite trees. *Theoretical computer science*, **25**(2), 95–169.

de la Briandais, René. (1959). File searching using variable length keys. *Pages 295–298 of: Proc. western joint computer conference*, vol. 15. AFIPS Press.

Dielissen, Victor J., & Kaldewaij, Anne. (1995). A simple, efficient, and flexible implementation of flexible arrays. *Pages 232–241 of: Third international conference on mathematics of program construction (MPC'95)*. Lecture Notes in Computer Science, vol. 947. Springer Verlag.

Henglein, Fritz. (1993). Type inference with polymorphic recursion. *ACM transactions on programming languages and systems*, **15**(2), 253–289.

Hinze, Ralf. 1999 (February). *Polytypic programming with ease*. Tech. rept. IAI-TR-99-2. Institut für Informatik III, Universität Bonn.

Jansson, Patrik, & Jeuring, Johan. (1997). PolyP—a polytypic programming language extension. *Pages 470–482 of: Conf. record 24th ACM SIGPLAN-SIGACT symp. on principles of programming languages, POPL'97, Paris, France*. New York: ACM Press.

Jeuring, Johan, & Jansson, Patrik. (1996). Polytypic programming. *Pages 68–114 of:* Launchbury, J., Meijer, E., & Sheard, T. (eds), *Tutorial text 2nd international school on advanced functional programming, Olympia, WA, USA*. Lecture Notes in Computer Science, vol. 1129. Springer Verlag.

Jones, Mark P. (1995). Functional programming with overloading and higher-order polymorphism. *Pages 97–136 of: First international spring school on advanced functional programming techniques*. Lecture Notes in Computer Science, vol. 925. Springer Verlag.

Knuth, Donald E. (1998). *The art of computer programming, Volume 3: Sorting and searching*. 2nd edn. Addison-Wesley Publishing Company.

McCracken, Nancy Jean. (1984). The typechecking of programs with implicit type structure. *Pages 301–315 of:* Kahn, Gilles, MacQueen, David B., & Plotkin, Gordon D. (eds), *Semantics of data types: International symposium, Sophia-Antipolis, France*. Lecture Notes in Computer Science, vol. 173. Berlin: Springer Verlag.

Mycroft, Alan. (1984). Polymorphic type schemes and recursive definitions. Paul, M., & Robinet, B. (eds), *International symposium on programming, 6th colloquium Toulouse*. LNCS 167.

Okasaki, Chris. (1998). *Purely functional data structures*. Cambridge University Press.

Okasaki, Chris, & Gill, Andy. (1998). Fast mergeable integer maps. *Pages 77–86 of: Workshop on ML*.

Peyton Jones, Simon. (1998). *Explicit quantification in Haskell*. ⟨URL: http://research.microsoft.com/ Users/ simonpj/ Haskell/ quantification.html⟩.

Wadsworth, C.P. (1979). Recursive type operators which are more than type schemes. *Bulletin of the EATCS*, **8**, 87–88. Abstract of a talk given at the 2nd International Workshop on the Semantics of Programming Languages, Bad Honnef, Germany, 19–23 March 1979.