J. Functional Programming **10** (4): 327–351, July 2000. Printed in the United Kingdom

Generalizing generalized tries

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 built according to an arbitrary signature. Here we go one step further, and define tries and operations on tries generically for arbitrary datatypes of first-order kind, including parameterized and nested datatypes. The derivation employs techniques recently developed in the context of polytypic programming and can be regarded as a comprehensive case study in this new programming paradigm. It is well known that for the implementation of generalized tries, nested datatypes and polymorphic recursion are needed. Implementing tries for first-order kinded datatypes places even greater demands on the type system: it requires rank-2 type signatures and second-order nested datatypes. Despite these requirements, the definition of tries is surprisingly simple, which is mostly due to the framework of polytypic programming.

Capsule Review

Implementing tries whose search keys are values of a complicated datatype is far from trivial – or is it?

Using a new approach to polytypic programming, this paper shows how to implement tries for arbitrary datatypes, including even nested datatypes. This problem is solved by the systematic, largely mechanical, application of simple rules.

If you are willing to be convinced of the advantages of the emerging paradigm of polytypic programming, in particular its conceptual simplicity, read this paper!

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 in figure 1. Searching in a trie starts at the root and proceeds by traversing the edge that matches the first

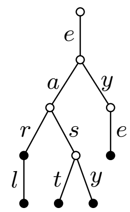

Fig. 1. A simple trie.

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 in figure 1. 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. For concreteness, programs will be given in the functional programming language Haskell 98 (Peyton Jones and Hughes, 1999). If strings are defined by the datatype

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

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

$$\begin{aligned}
\mathbf{data}\ \mathit{MapStr}\ v &\quad = \quad \mathit{TrieStr}\ (\mathit{Maybe}\ v)\ (\mathit{MapChar}\ (\mathit{MapStr}\ v))\\
\mathbf{data}\ \mathit{Maybe}\ v &\quad = \quad \mathit{Nothing} \mid \mathit{Just}\ v.
\end{aligned}$$

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 represented by the trie. In this case, the first component equals *Nothing*. The second component corresponds to the edge map. To keep the example manageable, we implement *MapChar* using association lists (note that in Haskell *List* t is written $[t]$).

$$\begin{aligned}
&\mathbf{type}\ \mathit{MapChar}\ v && = \quad \mathit{List}\ (\mathit{Char}, v)\\
&\mathit{lookupChar} && :: \quad \mathit{Char} \to \mathit{MapChar}\ v \to v\\
&\mathit{lookupChar}\ c\ [\,] && = \quad \mathit{error}\ \texttt{"not found"}\\
&\mathit{lookupChar}\ c\ ((c', v) : x) && = \quad \mathbf{if}\ c == c'\ \mathbf{then}\ v\ \mathbf{else}\ \mathit{lookupChar}\ c\ x
\end{aligned}$$

Building upon *lookupChar*, we can define a look-up function for strings. To lookup the empty string we access the first component of the trie. 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:

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

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

If the key is not in the domain of the finite map, a run-time error is raised. This will be remedied later.

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 built according to an arbitrary signature, i.e. by elements of an arbitrary non-parameterized datatype. The definition of *lookupStr* already gives a clue what 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. Generally, if we have a datatype 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 using *value*. Note that a nullary constructor of type T can be viewed as a function of type $() \to T$. Consequently, the type constructor *Maybe* can be seen as implementing finite maps over the unit datatype '()' with *value* as its look-up function.

As a second example, consider the datatype of external search trees.

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

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

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

The type *MapBin* is an instance of a so-called *nested datatype* (*nest* for short). The term 'nested datatype' has been coined by Bird and Meertens (1998), and characterizes parameterized datatypes whose definition involves 'recursive calls' – *MapBin* (*MapChar* (*MapBin* v)) in the example above – that 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:

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

Looking up a node involves two recursive calls. The second, *lookupBin* l, is of type $Bin \to MapBin\ (MapChar\ (MapBin\ v)) \to MapChar\ (MapBin\ v)$, which is a substitution instance of the declared type. Haskell 98 allows polymorphic recursion

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

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).

It is absolutely necessary that *MapBin* and *lookupBin* are parametric with respect to the codomain of the finite maps. Had we restricted the type of *lookupBin* to $Bin \to MapBin\ V \to V$ for some fixed type V, the definition would have no longer type-checked. This also explains why the construction does not work for the finite set abstraction.

Remark 1

Looking up a constructed value boils down to composing look-up functions. Interestingly, the order of composition is completely arbitrary: we are free to use either textual order or reverse textual order. For instance, *MapStr* and *lookupStr* can alternatively be defined by

$$\begin{array}{lcl}
\textbf{data}\ MapStr\ v & = & TrieStr\ (Maybe\ v)\ (MapStr\ (MapChar\ v)) \\
lookupStr & :: & Str \to MapStr\ v \to v \\
lookupStr\ Nil\ (TrieStr\ tn\ tc) & = & value\ tn \\
lookupStr\ (Cons\ c\ s)\ (TrieStr\ tn\ tc) & = & (lookupChar\ c \circ lookupStr\ s)\ tc.
\end{array}$$

These definitions employ reverse textual order – s is looked up first and then c – and correspond to the textual order implementation of tries for 'snoc' strings given by **data** $Rts = Lin \mid Snoc\ Rts\ Char$. That said, it becomes clear that both orders must work equally well. As an aside, note that *MapStr* is now a nested datatype and *lookupStr* requires polymorphic recursion. □

From the discussion above it should be clear how to define tries for arbitrary non-parameterized datatypes. In this paper we go one step further, and show how to generalize the concept to arbitrary datatypes of first-order kind, including parameterized and nested datatypes. Note that a datatype of first-order kind may be parameterized by types, but not by type constructors. In the sequel, the qualifier 'of first-order kind' will usually be omitted. Now, we are particularly interested in giving a *compositional* definition of tries. Let us briefly discuss what we mean by 'compositional'. In Haskell, strings are represented as lists of characters: $String = List\ Char$. This suggests that tries for strings should be compositionally defined in terms of tries for lists and tries for characters: $MapString = MapList\ MapChar$. Since *List* is a function on types (a so-called *functor*), *MapList* is consequently a function on tries – or rather, on trie types (a so-called *higher-order functor*). A note on terminology: though *MapList* is a function, we often refer to *MapList* simply as a trie just like *List* is often referred to as a type.

In generalizing tries to type constructors, we will answer in particular the intriguing question what the *generalized trie of a nested datatype* looks like. This question is not only of theoretical but also of practical interest, since a number of data structures, such as 2-3 trees or red-black trees, have recently been shown to be expressible by nested declarations. Bird and Paterson (1999) 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

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

problem of constructing generalized tries for a nested datatype (the solution to this problem will be presented in section 5).

To develop generalized tries we will employ the framework of *polytypic programming*. In fact, the following can be regarded as a comprehensive case study in this new programming paradigm. Briefly, a polytypic or generic function is one that is defined by induction on the structure of types. A simple example for a polytypic function is $encode :: T \to [Bit]$, which encodes an element of type T as a bit string. The function *encode* can sensibly be defined for each type, and it is usually a tiresome, routine matter to do so. A polytypic programming language enables the user to program *encode* once and for all times. The specialization of *encode* to concrete instances of T is then handled automatically by the system. Polytypic programming can be surprisingly simple. In a companion paper (Hinze, 1999b), we show that it suffices to define a polytypic function on predefined types, sums and products. This information is sufficient to specialize a polytypic function to arbitrary datatypes, including mutually recursive, parameterized 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 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. Mathematically speaking, generalized tries are based on the following isomorphisms:

$$\begin{aligned}(k_1 + k_2) \to_{\text{fin}} v &\cong (k_1 \to_{\text{fin}} v) \times (k_2 \to_{\text{fin}} v)\\ (k_1 \times k_2) \to_{\text{fin}} v &\cong k_1 \to_{\text{fin}} (k_2 \to_{\text{fin}} v).\end{aligned}$$

Here, $k \to_{\text{fin}} v$ denotes the set of all finite maps from k to v. Note that $k \to_{\text{fin}} v$ is sometimes written $v^{[k]}$, which explains why these equations are also known as the 'laws of exponentials'.

We have seen that nested datatypes and polymorphic recursion are necessary for the implementation of generalized tries. Implementing tries for datatypes of first-order kind, especially nested datatypes, places even greater demands on the type system: it requires rank-2 type signatures (McCracken, 1984), datatypes of second-order kind (Jones, 1995) and second-order nests. Since Haskell 98 does not offer rank-2 types, we will give the examples in an ideal, Haskell-like language. In particular, we will write polymorphic types using explicit universal quantifiers. The simple changes necessary to make the examples run under GHC (GHC Team, 1999) or Hugs 98 (Jones and Peterson, 1999) are given at the end of section 5.

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 the companion paper (Hinze, 1999b). Section 3 applies the technique to implement a finite map abstraction based on generalized tries. Section 4 discusses variations on the theme. Generalized tries for de Bruijn terms are presented in section 5. Finally, section 6 reviews related work, and points out a direction for future work.

2 A polytypic programming primer

2.1 Datatypes

A polytypic function is one that is parameterized by 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 lists; *Bintree* encompasses external binary search trees. The types *Perfect* and *Sequ* are examples for nested datatypes: *Perfect* comprises perfectly balanced, binary leaf trees (Hinze, 1999a); 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 '$\times$' for products) and omitting constructor names, we obtain the following emaciated recursion equations:

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

In the following, we treat 1, '$+$' and '$\times$' as if they were given by the following datatype declarations (note that in Haskell, 1 and '$\times$' have an extra element, $\perp$, which we simply ignore):

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

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 primitive type constructors $\{1, \mathit{Int}, +, \times\}$ type expressions can be seen as being defined by the following grammar (which is akin to a **data** declaration except that the latter does not allow us to use '1', '$+$', and '$\times$' as constructor names):

$$
T \quad ::= \quad 1 \mid \mathit{Int} \mid (T + T) \mid (T \times T).
$$

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

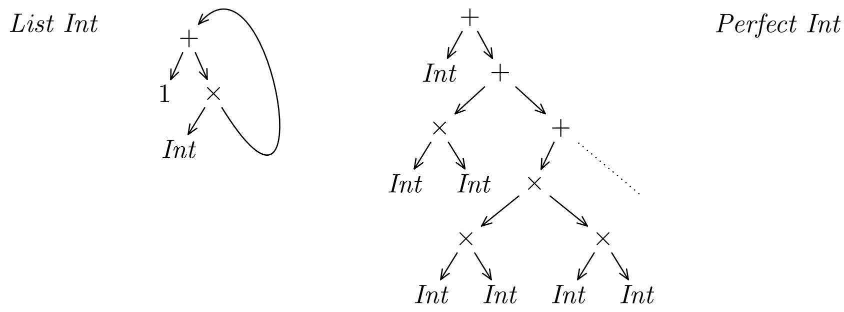

Fig. 2. Types interpreted as infinite type expressions.

In the sequel, we let t range over type expressions, and we agree upon that '$\times$' binds more tightly than '$+$'.

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 2 displays the infinite type expressions *List Int* and *Perfect Int* in a tree-like form. The expressions are obtained by unrolling the equations for *List* and *Perfect* ad infinitum. The recursion equations above can be regarded as defining functions over type expressions. Note that *List Int* is a *rational tree* while *Perfect Int* is an *algebraic tree*. A rational tree is a possibly infinite tree that has only a finite number of subtrees. Algebraic trees are obtained as solutions of so-called algebraic equations (Courcelle, 1983), which are akin to datatype declarations. In general, we obtain rational trees for regular types such as *List*, *Bintree* and *Fork*, and algebraic trees for nested types such as *Perfect* and *Sequ*.

2.2 Polytypic definitions

A polytypic value is defined by induction on the structure of type expressions. In general, the definition takes the following form:

$$\begin{aligned}
poly\langle a\rangle &:: \tau\langle a\rangle\\
poly\langle 1\rangle &= poly_1\\
poly\langle Int\rangle &= poly_{Int}\\
poly\langle a_1 + a_2\rangle &= poly_+ \,(poly\langle a_1\rangle, poly\langle a_2\rangle)\\
poly\langle a_1 \times a_2\rangle &= poly_\times \,(poly\langle a_1\rangle, poly\langle a_2\rangle).
\end{aligned}$$

Here, *poly* is the name of the polytypic value; a, a_1 and a_2 are type variables; τ, $poly_1$, $poly_{Int}$, $poly_+$ and $poly_\times$ are the ingredients that have to be supplied by the polytypic programmer. The type of $poly\langle a\rangle$ is given by the type scheme $\tau\langle a\rangle$, which may contain function types and universally quantified types. Note that type parameters are always written in angle brackets, to distinguish them from ordinary value parameters.

Example 1
The function $encode\langle a\rangle$, which encodes elements of type a as bit strings implementing a simple form of data compression (Jansson and Jeuring, 1999), can be defined as follows:

$$
\begin{array}{lcl}
\mathbf{data}\ Bit & = & \mathtt{0} \mid \mathtt{1} \\
encode\langle a\rangle & :: & a \to [Bit] \\
encode\langle 1\rangle\ x & = & [\,] \\
encode\langle Int\rangle\ x & = & encodeInt\ x \\
encode\langle a_1 + a_2\rangle\ (Inl\ x_1) & = & \mathtt{0} : encode\langle a_1\rangle\ x_1 \\
encode\langle a_1 + a_2\rangle\ (Inr\ x_2) & = & \mathtt{1} : encode\langle a_2\rangle\ x_2 \\
encode\langle a_1 \times a_2\rangle\ (x_1, x_2) & = & encode\langle a_1\rangle\ x_1 +\!\!+ encode\langle a_2\rangle\ x_2.
\end{array}
$$

To encode the single element of the unit type no bits are required. Integers are encoded using the primitive function *encodeInt*, whose existence we assume. To encode an element of a sum, we emit one bit for the constructor followed by the encoding of its argument. Finally, the encoding of a pair is given by the concatenation of the component's encodings. The code above implicitly defines the type scheme $\tau\langle a\rangle = a \to [Bit]$ and the functions $encode_1$, $encode_{Int}$, $encode_+$ and $encode_\times$:

$$
\begin{array}{lcl}
encode_1 & = & \lambda x \to [\,] \\
encode_{Int} & = & \lambda x \to encodeInt\ x \\
encode_+\ (\varphi_1, \varphi_2) & = & \lambda x \to \mathbf{case}\ x\ \mathbf{of}\ \{Inl\ x_1 \to \mathtt{0} : \varphi_1\ x_1; Inr\ x_2 \to \mathtt{1} : \varphi_2\ x_2\} \\
encode_\times\ (\varphi_1, \varphi_2) & = & \lambda x \to \varphi_1\ (fst\ x) +\!\!+ \varphi_2\ (snd\ x). \quad \square
\end{array}
$$

The inductive definition of *poly* induces a unique function $poly\langle t\rangle$ for each 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 met, since types and functions are given by Haskell programs, which are interpreted in these domains.

The use of infinite type expressions as index sets for polytypic values distinguishes our approach from previous ones that are based on the initial algebra semantics of datatypes (Jeuring and Jansson, 1996; Jansson and Jeuring, 1997). Briefly, our approach has two major advantages: it is simpler (the programmer must consider fewer cases); and it is more general (it covers all datatypes of first-order kind). As an aside, note that our approach also allows to define polytypic values that are indexed by type constructors rather than types. The archetypical example for such a function is $size\langle f\rangle :: \forall a.f\ a \to Int$, which counts the number of values of type a in a given structure of type $f\ a$. Further details can be found in the companion paper (Hinze, 1999b).

2.3 Specializing polytypic definitions

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

$$
\begin{aligned}
& \mathit{encode}\langle \mathit{Perfect}\ \mathit{Int}\rangle \\
= \;& \mathit{encode}\langle \mathit{Int} + \mathit{Perfect}\ (\mathit{Fork}\ \mathit{Int})\rangle \\
= \;& \mathit{encode}_{+}\ (\mathit{encode}_{\mathit{Int}}, \mathit{encode}\langle \mathit{Perfect}\ (\mathit{Fork}\ \mathit{Int})\rangle) \\
= \;& \mathit{encode}_{+}\ (\mathit{encode}_{\mathit{Int}}, \mathit{encode}\langle \mathit{Fork}\ \mathit{Int} + \mathit{Perfect}\ (\mathit{Fork}^2\ \mathit{Int})\rangle) \\
= \;& \mathit{encode}_{+}\ (\mathit{encode}_{\mathit{Int}}, \mathit{encode}_{+}\ (\mathit{encode}\langle \mathit{Fork}\ \mathit{Int}\rangle, \mathit{encode}\langle \mathit{Perfect}\ (\mathit{Fork}^2\ \mathit{Int})\rangle)) \\
= \;& \ldots
\end{aligned}
$$

To define $\mathit{encode}\langle \mathit{Perfect}\ \mathit{Int}\rangle$ we require $\mathit{encode}\langle \mathit{Perfect}\ (\mathit{Fork}^n\ \mathit{Int})\rangle$ for each $n \geqslant 1$. It is probably clear that in general we cannot hope to obtain a *finite* representation of $\mathit{poly}\langle t\rangle$ this way. Instead, we must base the specialization on the *representation of types*, i.e. 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*. Roughly speaking, a functor can be seen as a function on types. Functor expressions 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 ith component. For $n = 1$ and $n = 2$ we use the following more familiar names: $\mathit{Id} = \Pi_1^1$, $\mathit{Fst} = \Pi_1^2$ and $\mathit{Snd} = \Pi_2^2$. Elements of P^n are predefined functors of arity n, i.e. $P^0 = \{1, \mathit{Int}\}$ – we identify types and nullary functors – and $P^2 = \{+, \times\}$. The expression $f \cdot (f_1, \ldots, f_k)$ denotes the composition of a k-ary functor f with functors f_i, all of arity n. We omit parentheses when $k = 1$, and we write $K t$ instead of $t \cdot ()$ when $k = 0$. Note that, in $K t$, the component t is a type viewed as a nullary functor; $K t$ is then an n-ary functor. Furthermore, we write $f_1 + f_2$ for $+ \cdot (f_1, f_2)$, and similarly, $f_1 \times f_2$. We agree upon that '$\cdot$' binds more tightly than '$\times$', which in turn takes precedence over '$+$'. For instance, $f + g \times h \cdot h$ means $f + (g \times (h \cdot h))$. Finally, we let f, g, and h range over functor expressions and p over primitive functors.

Here are the datatype definitions of section 2.1 rewritten as functor equations:

$$
\begin{aligned}
\mathit{List} &= K 1 + \mathit{Id} \times \mathit{List} \\
\mathit{Bintree} &= \mathit{Fst} + \mathit{Bintree} \times \mathit{Snd} \times \mathit{Bintree} \\
\mathit{Fork} &= \mathit{Id} \times \mathit{Id} \\
\mathit{Perfect} &= \mathit{Id} + \mathit{Perfect} \cdot \mathit{Fork} \\
\mathit{Sequ} &= K 1 + \mathit{Sequ} \cdot \mathit{Fork} + \mathit{Id} \times \mathit{Sequ} \cdot \mathit{Fork}.
\end{aligned}
$$

In essence, functor equations are written in a compositional or 'point-free' style, while **data** definitions are written in an applicative or 'pointwise' style. A system of functor equations has the general form $x_1 = f_1; \ldots; x_m = f_m$, where the x_i are functor variables (acting as unknowns) and the f_i are functor expressions.

Now, the central idea of the specialization is to mimic the structure of datatypes on the value level. For instance, $\mathit{encode}\langle \mathit{Perfect}\ \mathit{Int}\rangle$ will be compositionally defined in terms of the specializations for the constituent datatypes *Perfect* and *Int*. Since *Perfect* is a function on types, the 'encoder' for *Perfect* is consequently a function on encoders: it takes an encoder for values of type t, and yields an encoder for values of type *Perfect* t, i.e. it takes $encode\langle t\rangle$ to $encode\langle Perfect\ t\rangle$. In general, we define, for each functor f of arity n, an n-ary function $poly_n\langle f\rangle$ satisfying

$$poly_n\langle f\rangle\ (poly\langle t_1\rangle,\ldots,poly\langle t_n\rangle) \quad = \quad poly\langle f(t_1,\ldots,t_n)\rangle\ , \tag{1}$$

for all type expressions $t_1, \ldots, t_n$. It can be shown that the following definition satisfies this specification:

$$\begin{aligned}
poly_n\langle f\rangle &\ ::\ \forall a_1\ldots a_n.\tau\langle a_1\rangle\times\cdots\times\tau\langle a_n\rangle\to\tau\langle f\,(a_1,\ldots,a_n)\rangle\\
poly_n\langle \Pi_i^n\rangle &\ =\ \pi_i^n\\
poly_n\langle p\rangle &\ =\ poly_p\\
poly_n\langle g\cdot(h_1,\ldots,h_k)\rangle &\ =\ poly_k\langle g\rangle\star(poly_n\langle h_1\rangle,\ldots,poly_n\langle h_k\rangle),
\end{aligned}$$

where $\pi_i^n\,(\varphi_1,\ldots,\varphi_n) = \varphi_i$ is the ith projection function, and '$\star$' denotes n-ary composition defined by $\varphi\star(\varphi_1,\ldots,\varphi_n) = \lambda v \to \varphi\,(\varphi_1\ v,\ldots,\varphi_n\ v)$. 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^n is interpreted by π_i^n, p by $poly_p$, and '$\cdot$' by '$\star$'.

Finally, we can define *poly* in terms of $poly_0$:

$$\begin{aligned}
poly\langle t\rangle &\ ::\ \tau\langle t\rangle\\
poly\langle t\rangle &\ =\ poly_0\langle t\rangle\ ()
\end{aligned}$$

In the sequel we will identify *poly* and $poly_0$ just like we identify types and nullary functors.

By now we have the necessary prerequisites at hand to define the specialization of a polytypic value $poly\langle t\rangle$ for a given instance of t. Assume that the type is defined by the system of equations $x_1 = f_1;\ldots;x_m = f_m$, with $t = x_i$ for some i. For each equation $x_i = f_i$, where f_i is a k-ary functor expression, a function definition of the form $poly_k\langle x_i\rangle = poly_k\langle f_i\rangle$ is generated. The expression $poly_k\langle f_i\rangle$ is given by the inductive definition above, additionally setting $poly_k\langle x_i\rangle = poly\_x_i$, where $poly\_x_i$ is a new function symbol.

Example 2
Let us apply the above framework to specialize $encode\langle t\rangle$ for $t = Perfect\ Int$. The specialization proceeds entirely mechanically. Using the original constructor names and abbreviating type names to their first letter, we obtain

$$\begin{aligned}
&encodePI &&::\ Perfect\ Int\to[Bit]\\
&encodePI\ x &&=\ encodeP\ encodeInt\ x\\
&encodeF &&::\ \forall a.(a\to[Bit])\to(Fork\ a\to[Bit])\\
&encodeF\ enca\ (Fork\ x_1\ x_2) &&=\ enca\ x_1 +\!\!+\ enca\ x_2\\
&encodeP &&::\ \forall a.(a\to[Bit])\to(Perfect\ a\to[Bit])\\
&encodeP\ enca\ (Null\ x) &&=\ \mathtt{0} : enca\ x\\
&encodeP\ enca\ (Succ\ x) &&=\ \mathtt{1} : encodeP\ (encodeF\ enca)\ x.
\end{aligned}$$

Encoding a perfect tree operates in two stages: while recursing *encodeP* constructs

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

a tailor-made encoding function $encodeF^i\ enca$ of type $Fork^i\ a \to [Bit]$, which is eventually applied in the base case. □

3 Tries generically

In this section we apply the framework of polytypic programming to implement generalized tries generically for all datatypes of first-order kind. 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 define a scheme for constructing datatypes

$$Map\langle k :: *\rangle \quad :: \quad * \to *,$$

which assigns a type constructor of kind $* \to *$ to each key type k of kind $*$. The kind system of Haskell specifies the 'type' of a type constructor (Jones, 1995). The '$*$' kind comprises nullary constructors like Int. The kind $\kappa_1 \to \kappa_2$ comprises type constructors that map type constructors of kind κ_1 to those of kind κ_2. The order of a kind is given by $order(*) = 0$ and $order(\kappa_1 \to \kappa_2) = max\{1 + order(\kappa_1), order(\kappa_2)\}$.

The type $Map\langle k\rangle\ v$ represents the set $k \to_{\text{fin}} v$ of finite maps from k to v. It is worth noting that the two arguments of '$\to_{\text{fin}}$' are treated in a different way: the key type k is used as a type index, i.e. Map will be defined by induction on the structure of k, whereas v is a type parameter, i.e. Map will be parametric in the value type v, and the operations on tries will be polymorphic with respect to v.

We will implement the following operations on tries.

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

The signature of $lookup\langle k\rangle$ deviates slightly from that used in the introduction: the look-up function returns a value of type $Maybe\ v$ instead of v to be able to signal that a key is unbound. The functions $insert\langle k\rangle$ and $merge\langle k\rangle$ take as a first argument a so-called *combining function*, which is applied whenever two bindings have the same key. For instance, $\lambda new\ old \to new$ is used as the combining function for $insert\langle k\rangle$ if the new binding is to override an old binding with the same key. For finite maps of type $Map\langle k\rangle\ Int$, 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 $insert\langle k\rangle$ and $merge\langle k\rangle$ generically for all types!

3.1 Type-indexed tries

We have already noted in the introduction that generalized tries are based on the laws of exponentials:

$$\begin{array}{rcl}
1 \to_{\text{fin}} v & \cong & v \\
(k_1 + k_2) \to_{\text{fin}} v & \cong & (k_1 \to_{\text{fin}} v) \times (k_2 \to_{\text{fin}} v) \\
(k_1 \times k_2) \to_{\text{fin}} v & \cong & k_1 \to_{\text{fin}} (k_2 \to_{\text{fin}} v)
\end{array}$$

To define the notion of finite map, it is customary to assume that each value type v contains a distinguished element or *base point* $\perp_v$ – see Connelly and Morris (1995). A finite map is then a function whose value is $\perp_v$ for all but finitely many arguments. For the implementation of tries it is, however, inconvenient to make such a strong assumption (though one could use type classes for this purpose). Instead, we explicitly add when necessary a base point using *Maybe*. It appears that this is only required for the unit type motivating the following definition of $Map\langle k\rangle$:

$$\begin{aligned}
Map\langle 1\rangle\ v &= Maybe\ v\\
Map\langle Int\rangle\ v &= Patricia.Dict\ v\\
Map\langle k_1 + k_2\rangle\ v &= Map\langle k_1\rangle\ v \times Map\langle k_2\rangle\ v\\
Map\langle k_1 \times k_2\rangle\ v &= Map\langle k_1\rangle\ (Map\langle k_2\rangle\ v).
\end{aligned}$$

We take for granted 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 and 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*.

Note that $Map\langle k\rangle$ is a unary functor. Using the functorial notation of section 2.3, we can define $Map\langle k\rangle$ more succinctly as

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

Since the trie for the unit type is given by *Maybe* rather than *Id*, tries for isomorphic types are, in general, not isomorphic. We have, for instance, $1 \cong 1\times 1$, but $Map\langle 1\rangle = Maybe \not\cong Maybe\cdot Maybe = Map\langle 1\times 1\rangle$. The trie type $Maybe\cdot Maybe$ has two different representations of the empty trie: *Nothing* and *Just Nothing*. However, only the first one will be used in our implementation.

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,

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

The type constructor $Map_1\langle f\rangle$ is *the generalized trie of the unary type constructor* f. It takes as argument the generalized trie of the base type, say, t and yields the generalized trie of $f\ t$. In general, $Map_n\langle f\rangle$ satisfies

$$Map_n\langle f\rangle\ (Map\langle t_1\rangle,\ldots,Map\langle t_n\rangle) \quad = \quad Map\langle f(t_1,\ldots,t_n)\rangle, \tag{2}$$

for all type expressions $t_1, \ldots, t_n$. It may come as a surprise that the framework for specializing type-indexed values 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, both of which are available in the world of functors and higher-order functors. Consequently, $Map_n\langle f\rangle$ is given by

$$\begin{aligned}
Map_n\langle f\rangle &:: (* \to *) \times \cdots \times (* \to *) \to (* \to *) \\
Map_n\langle \Pi_i^n\rangle &= \Pi_i^n \\
Map_n\langle p\rangle &= Map_p \\
Map_n\langle g \cdot (h_1, \ldots, h_k)\rangle &= Map_k\langle g\rangle \cdot (Map_n\langle h_1\rangle, \ldots, Map_n\langle h_k\rangle).
\end{aligned}$$

Let us specialize $Map_n\langle f\rangle$ to the datatypes listed in section 2.1. As before, we abbreviate type names to their first letter, i.e. we write *MapL* instead of *MapList*:

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

Since Haskell 98 permits the definition of higher-order kinded datatypes, the second-order functors above can be directly coded as datatypes.[1] All we have to do is to bring the equations into an applicative form:

$$\begin{aligned}
\textbf{data}\ MapL\ m\ v &= TrieL\ (Maybe\ v)\ (m\ (MapL\ m\ v)) \\
\textbf{data}\ MapB\ m_1\ m_2\ v &= TrieB\ (m_1\ v) \\
&\qquad\qquad (MapB\ m_1\ m_2\ (m_2\ (MapB\ m_1\ m_2\ v))).
\end{aligned}$$

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

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

The generalized trie of a nested datatype is a second-order nested datatype! 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 tries *MapP* and *MapS* are second-order nests since the parameter m of kind $* \to *$ is changed in the recursive calls. By contrast, *MapB* is a first-order nest since its instantiated parameter v has kind $*$. 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 –

[1] Note that Miranda (trademark of Research Software Ltd), Standard ML and previous versions of Haskell (1.2 and before) only have first-order kinded datatypes.

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

One a (*Sequ* (*Fork a*)) becomes *One* (*Sequ* (*Fork a*)) *a* – then the third component of *TrieS* has type *MapS* (*MapF m*) (*m v*), and since both *m* and *v* are instantiated, *MapS* is consequently both a first- and a second-order nest.

3.2 Empty and singleton tries

The empty trie is defined as follows:

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

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!

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,

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

The type signature contains two occurrences of *Map*. Of course, if we want to specialize $\mathit{empty}_1\langle f\rangle$ for a given f we must specialize its type signature, as well. In a first step, we use the specification of Map_n, equation (2), to replace $\mathit{Map}\langle f\ k\rangle$ by $\mathit{Map}_1\langle f\rangle\ (\mathit{Map}\langle k\rangle)$.

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

In a second step, we generalize $\mathit{Map}\langle k\rangle$ to a fresh type variable, say, m.

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

Note that $\mathit{empty}_1\langle f\rangle$ has a so-called rank-2 type signature (McCracken, 1984).

Let us take a look at some examples:

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

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 $(\forall v.\mathit{MapF}\ m\ v) \rightarrow$

$(\forall v.MapP\ (MapF\ m)\ v)$, 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)

$$TrieP\ e\ (TrieP\ (TrieF\ e)\ (TrieP\ (TrieF\ (TrieF\ e))\ \ldots)).$$

In section 4.1, we shall discuss a slightly modified representation of generalized tries that avoids this problem.

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

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

The definition of $single\langle k\rangle$ is interesting because it falls back on $empty\langle k\rangle$ in the third and the fourth equations. This requires a small extension of the theory of section 2: the specialization $single_n\langle f\rangle$ must be parameterized both with $single\langle k\rangle$ and with $empty\langle k\rangle$. For $n = 1$ we obtain the type signature

$$\begin{array}{lcl}
single_1\langle f\rangle & :: & \forall m.(\forall v.m\ v) \to (\forall v.k \times v \to m\ v) \\
 & & \quad \to (\forall v.f\ k \times v \to Map_1\langle f\rangle\ m\ v).
\end{array}$$

Let us again specialize the polytypic function to lists and perfect trees:

$$\begin{array}{lcl}
singleL & :: & \forall m.(\forall v.m\ v) \to (\forall v.k \times v \to m\ v) \\
 & & \quad \to (\forall v.List\ k \times v \to MapL\ m\ v) \\
singleL\ e\ s\ (Nil, v) & = & TrieL\ (Just\ v)\ e \\
singleL\ e\ s\ (Cons\ i\ is, v) & = & TrieL\ Nothing\ (s\ (i, singleL\ e\ s\ (is, v))) \\
singleF & :: & \forall m.(\forall v.m\ v) \to (\forall v.k \times v \to m\ v) \\
 & & \quad \to (\forall v.Fork\ k \times v \to MapF\ m\ v) \\
singleF\ e\ s\ (Fork\ i_1\ i_2, v) & = & TrieF\ (s\ (i_1, s\ (i_2, v))) \\
singleP & :: & \forall m.(\forall v.m\ v) \to (\forall v.k \times v \to m\ v) \\
 & & \quad \to (\forall v.Perfect\ k \times v \to MapP\ m\ v) \\
singleP\ e\ s\ (Null\ i, v) & = & TrieP\ (s\ (i, v))\ (emptyP\ (emptyF\ e)) \\
singleP\ e\ s\ (Succ\ i, v) & = & TrieP\ e\ (singleP\ (emptyF\ e)\ (singleF\ e\ s)\ (i, v)).
\end{array}$$

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 Look-up

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

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

On sums the look-up function selects the appropriate map; on products it 'composes' the look-up functions for the components. Since $lookup\langle k\rangle$ has the result type $Maybe\ v$, the composition must take care of the error signal $Nothing$:

$$
\begin{array}{lcl}
(\diamond) & :: & (a \to Maybe\ b) \to (b \to Maybe\ c) \to (a \to Maybe\ c) \\
(m_1 \diamond m_2)\ a_1 & = & \textbf{case}\ m_1\ a_1\ \textbf{of}\ \{Nothing \to Nothing\,; Just\ a_2 \to m_2\ a_2\}.
\end{array}
$$

The operation $(\diamond)$ amounts to the monad or *Kleisli composition* (Bird, 1998). As an aside, note that the arguments are not in the same order as with functional composition.

Specializing $lookup\langle k\rangle$ to concrete instances of k is by now probably a matter of routine. Here is $lookup_1\langle f\rangle$'s type signature:

$$
\begin{array}{lcl}
lookup_1\langle f\rangle & :: & \forall m.(\forall v.k \to m\ v \to Maybe\ v) \\
& & \quad \to (\forall v.f\ k \to Map_1\langle f\rangle\ m\ v \to Maybe\ v).
\end{array}
$$

For lists and perfect trees we obtain

$$
\begin{array}{lcl}
lookupL & :: & \forall m.(\forall v.k \to m\ v \to Maybe\ v) \\
& & \quad \to (\forall v.List\ k \to MapL\ m\ v \to Maybe\ v) \\
lookupL\ l\ Nil\ (TrieL\ tn\ tc) & = & tn \\
lookupL\ l\ (Cons\ i\ is)\ (TrieL\ tn\ tc) & = & (l\ i \diamond lookupL\ l\ is)\ tc \\
lookupF & :: & \forall m.(\forall v.k \to m\ v \to Maybe\ v) \\
& & \quad \to (\forall v.Fork\ k \to MapF\ m\ v \to Maybe\ v) \\
lookupF\ l\ (Fork\ i_1\ i_2)\ (TrieF\ tf) & = & (l\ i_1 \diamond l\ i_2)\ tf \\
lookupP & :: & \forall m.(\forall v.k \to m\ v \to Maybe\ v) \\
& & \quad \to (\forall v.Perfect\ k \to MapP\ m\ v \to Maybe\ v) \\
lookupP\ l\ (Null\ i)\ (TrieP\ tn\ ts) & = & l\ i\ tn \\
lookupP\ l\ (Succ\ i)\ (TrieP\ tn\ ts) & = & lookupP\ (lookupF\ l)\ i\ ts.
\end{array}
$$

The function $lookupL$ generalizes $lookupStr$ defined in the introduction to this paper; we have $lookupStr\ s \cong value \circ lookupL\ lookupChar\ s$. The definition of $lookupP$ employs the same recursion scheme as $encodeP$: while recursing, $lookupP$ constructs a tailor-made look-up function $lookupF^i\ l$ of type $\forall v.Fork^i\ k \to MapF^i\ v \to Maybe\ v$, which is finally applied in the base case.

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

3.4 Inserting and merging

Insertion is defined in terms of $merge\langle k\rangle$ and $single\langle k\rangle$:

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

Unfortunately, this is not the most efficient implementation of $insert\langle k\rangle$, since singleton tries are in general given by infinite trees. This implies that the running time of $insert\langle k\rangle$ is *not* proportional to the size of the inserted key, 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 *Maybe a*

$$\begin{array}{lcl}
combine & :: & \forall v.(v \to v \to v)\\
 & & \quad \to (Maybe\ v \to Maybe\ v \to Maybe\ v)\\
combine\ c\ Nothing\ Nothing & = & Nothing\\
combine\ c\ Nothing\ (Just\ v') & = & Just\ v'\\
combine\ c\ (Just\ v)\ Nothing & = & Just\ v\\
combine\ c\ (Just\ v)\ (Just\ v') & = & Just\ (c\ v\ v'),
\end{array}$$

we can define $merge\langle k\rangle$ as follows:

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

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

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

The definition of $merge\langle k\rangle$ shows that it is sometimes necessary to implement operations more general than immediately needed. If $merge\langle k\rangle$ had the simplified type $\forall v.Map\langle k\rangle\ v \to Map\langle k\rangle\ v \to Map\langle k\rangle\ v$, 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 $merge_1\langle f\rangle$'s type signature:

$$\begin{array}{lcl}
merge_1\langle f\rangle & :: & \forall m.(\forall v.(v \to v \to v) \to (m\ v \to m\ v \to m\ v))\\
 & & \quad \to (\forall v.(v \to v \to v)\\
 & & \qquad \to (Map_1\langle f\rangle\ m\ v \to Map_1\langle f\rangle\ m\ v \to Map_1\langle f\rangle\ m\ v)).
\end{array}$$

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

The different instances of $merge_1\langle f\rangle$ are surprisingly concise:

$$
\begin{aligned}
&mergeL \quad :: \quad \forall m.(\forall v.(v \to v \to v) \to (m\ v \to m\ v \to m\ v)) \\
&\qquad\qquad\qquad \to (\forall v.(v \to v \to v) \to (MapL\ m\ v \to MapL\ m\ v \to MapL\ m\ v)) \\
&mergeL\ m\ c\ (TrieL\ tn\ tc)\ (TrieL\ tn'\ tc') \\
&\qquad = \quad TrieL\ (combine\ c\ tn\ tn')\ (m\ (mergeL\ m\ c)\ tc\ tc') \\
&mergeF \quad :: \quad \forall m.(\forall v.(v \to v \to v) \to (m\ v \to m\ v \to m\ v)) \\
&\qquad\qquad\qquad \to (\forall v.(v \to v \to v) \to (MapF\ m\ v \to MapF\ m\ v \to MapF\ m\ v)) \\
&mergeF\ m\ c\ (TrieF\ tf)\ (TrieF\ tf') \\
&\qquad = \quad TrieF\ (m\ (m\ c)\ tf\ tf') \\
&mergeP \quad :: \quad \forall m.(\forall v.(v \to v \to v) \to (m\ v \to m\ v \to m\ v)) \\
&\qquad\qquad\qquad \to (\forall v.(v \to v \to v) \to (MapP\ m\ v \to MapP\ m\ v \to MapP\ m\ v)) \\
&mergeP\ m\ c\ (TrieP\ tn\ ts)\ (TrieP\ tn'\ ts') \\
&\qquad = \quad TrieP\ (m\ c\ tn\ tn')\ (mergeP\ (mergeF\ m)\ c\ ts\ ts').
\end{aligned}
$$

3.5 Laws

Polytypic functions enjoy polytypic properties. The following laws hold generically for all instances of k, and can be proved by fixpoint induction over the structure of type expressions:

$$
\begin{aligned}
lookup\langle k\rangle\ i\ (empty\langle k\rangle) &= Nothing \\
lookup\langle k\rangle\ i\ (single\langle k\rangle\ (i_1, v_1)) &= \mathbf{if}\ i == i_1\ \mathbf{then}\ Just\ v_1\ \mathbf{else}\ Nothing \\
lookup\langle k\rangle\ i\ (merge\langle k\rangle\ c\ t_1\ t_2) &= combine\ c\ (lookup\langle k\rangle\ i\ t_1)\ (lookup\langle k\rangle\ i\ t_2).
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
merge\langle k\rangle\ c\ (empty\langle k\rangle)\ t &= t \\
merge\langle k\rangle\ c\ t\ (empty\langle k\rangle) &= t \\
merge\langle k\rangle\ c\ t_1\ (merge\langle k\rangle\ c\ t_2\ t_3) &= merge\langle k\rangle\ c\ (merge\langle k\rangle\ c\ t_1\ t_2)\ t_3.
\end{aligned}
$$

4 Variations on the theme

4.1 Spotted tries

The representation of tries as defined in section 3.1 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

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

it would be worth devising a special representation for the empty trie. Technically, this is achieved using so-called *spot products* (Connelly and 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* since $a_1 \times_\bullet a_2 \cong \mathit{Maybe}\ (a_1 \times a_2)$. 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.

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

To ensure that the representation is unique, 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:

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

4.2 Skinny tries

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

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

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

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

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 $\mathit{merge}\langle k\rangle$. Here are a few of the cases:

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

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

The remaining cases are defined accordingly.

5 Sample application: Generalized tries for de Bruijn terms

As a slightly larger application, let us construct generalized tries for de Bruijn terms (de Bruijn, 1972) building upon the representation given in section 4.1. These tries may be useful for performing common subexpression elimination on lambda terms, or for implementing a memoizing interpreter for the lambda calculus. de Bruijn notation is a special encoding of lambda terms, where a bound variable is represented by a natural number, giving the number of abstractions lying between the variable and its binding abstraction. For instance, $\lambda x \to x$ is represented by $\lambda 0$ and $\lambda x \to \lambda y \to x$ by $\lambda(\lambda 1)$. Recently, Bird and Paterson (1999) devised a nested implementation of de Bruijn terms, which nicely captures the 'distance invariant':

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Term}\ v & = & \mathit{Var}\ v \mid \mathit{App}\ (\mathit{Term}\ v)\ (\mathit{Term}\ v) \mid \mathit{Lam}\ (\mathit{Term}\ (\mathit{Incr}\ v)) \\
\mathbf{data}\ \mathit{Incr}\ v & = & \mathit{Zero} \mid \mathit{Succ}\ v.
\end{array}
$$

Closed de Bruijn terms can be represented as elements of *Term Void*, where *Void* is the empty type. For instance, $\lambda 0$ and $\lambda(\lambda 1)$ are written as $\mathit{Lam}\ (\mathit{Var}\ \mathit{Zero})$ and $\mathit{Lam}\ (\mathit{Lam}\ (\mathit{Var}\ (\mathit{Succ}\ \mathit{Zero})))$. Non-closed terms where the free variables are drawn from the type *Int* are given by elements of *Term Int*. For example, $\mathit{Var}\ 0$ and $\mathit{Lam}\ (\mathit{Lam}\ (\mathit{Var}\ (\mathit{Succ}\ (\mathit{Succ}\ 0))))$ correspond to the lambda terms z and $\lambda x \to \lambda y \to z$, in which the variable z appears free.

Figures 3, 4 and 5 contain the complete code for generalized tries on non-closed de Bruijn terms of type *Term Int*.[2]

Some remarks are appropriate. First of all, the datatype *MapT* in figure 4 is based on the functor equation

$$\mathit{MapT}\ m \quad = \quad m \times_\bullet \mathit{MapT}\ m \cdot \mathit{MapT}\ m \times_\bullet \mathit{MapT}\ (\mathit{MapI}\ m).$$

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.

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

All the definitions with the notable exception of the *empty* instances have been

[2] The source code is available from the *Journal of Functional Programming* Internet home page (`http://www.dcs.gla.ac.uk/jfp/online/jfpvol10.4/hinze/index.html`).

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{MapI}\ m\ v & = & \mathit{SpotI} \mid \mathit{TrieI}\ (\mathit{Maybe}\ v)\ (m\ v) \\
\mathit{emptyI} & :: & \forall m.(\forall v.\mathit{MapI}\ m\ v) \\
\mathit{emptyI} & = & \mathit{SpotI} \\
\mathit{singleI} & :: & \forall m.(\forall v.m\ v) \to (\forall v.k \times v \to m\ v) \\
 & & \quad \to (\forall v.\mathit{Incr}\ k \times v \to \mathit{MapI}\ m\ v) \\
\mathit{singleI}\ e\ s\ (\mathit{Zero}, v) & = & \mathit{TrieI}\ (\mathit{Just}\ v)\ e \\
\mathit{singleI}\ e\ s\ (\mathit{Succ}\ i, v) & = & \mathit{TrieI}\ \mathit{Nothing}\ (s\ (i, v)) \\
\mathit{lookupI} & :: & \forall m.(\forall v.k \to m\ v \to \mathit{Maybe}\ v) \\
 & & \quad \to (\forall v.\mathit{Incr}\ k \to \mathit{MapI}\ m\ v \to \mathit{Maybe}\ v) \\
\mathit{lookupI}\ l\ i\ \mathit{SpotI} & = & \mathit{Nothing} \\
\mathit{lookupI}\ l\ \mathit{Zero}\ (\mathit{TrieI}\ tz\ ts) & = & tz \\
\mathit{lookupI}\ l\ (\mathit{Succ}\ i)\ (\mathit{TrieI}\ tz\ ts) & = & l\ i\ ts \\
\mathit{mergeI} & :: & \forall m.(\forall v.(v \to v \to v) \to (m\ v \to m\ v \to m\ v)) \\
 & & \quad \to (\forall v.(v \to v \to v) \\
 & & \qquad\quad \to (\mathit{MapI}\ m\ v \to \mathit{MapI}\ m\ v \to \mathit{MapI}\ m\ v)) \\
\mathit{mergeI}\ m\ c\ \mathit{SpotI}\ t' & = & t' \\
\mathit{mergeI}\ m\ c\ t\ \mathit{SpotI} & = & t \\
\mathit{mergeI}\ m\ c\ (\mathit{TrieI}\ tz\ ts)\ (\mathit{TrieI}\ tz'\ ts') & & \\
 & = & \mathit{TrieI}\ (\mathit{combine}\ c\ tz\ tz')\ (m\ c\ ts\ ts')
\end{array}
$$

Fig. 3. Generalized tries for variables of type *Incr v*.

mechanically derived from the generic definitions given in this and in the previous sections. The definition of *emptyI* has been simplified by omitting its parameter, which is not required. The same remark applies to *emptyT* and *emptyTI*.

Let us stress that the code does not conform to the Haskell 98 standard (Peyton Jones and Hughes, 1999), which neither provides explicit universal quantifiers nor rank-2 type signatures. However, both GHC 4.04 (GHC Team, 1999) and Hugs 98 (as of September 1999 (Jones and Peterson, 1999)) implement the necessary extensions (Peyton Jones, 1998). We only have to adjust the type signatures. To exemplify, the signature

$$\mathit{lookupI} \quad :: \quad \forall m.(\forall v.k \to m\ v \to \mathit{Maybe}\ v) \to (\forall v.\mathit{Incr}\ k \to \mathit{MapI}\ m\ v \to \mathit{Maybe}\ v)$$

must be changed to

$$\mathit{lookupI} \quad :: \quad (\forall v.k \to m\ v \to \mathit{Maybe}\ v) \to (\mathit{Incr}\ k \to \mathit{MapI}\ m\ w \to \mathit{Maybe}\ w).$$

The rewrite involves two steps: (i) use $t \to \forall v.u = \forall w.t \to u[v := w]$, where w is a fresh type variable to push quantifiers to the top-level; and (ii) discard top-level quantifiers. Both steps are meaning preserving (recall that every free type variable in a signature is implicitly universally quantified).

6 Related and future work

Knuth (1998) attributes the idea of a trie to Thue (1912), who introduced it in a paper about strings that do not contain adjacent repeated substrings. de la Briandais (1959) recommended tries for computer searching. The generalization of tries from strings to elements built according to an arbitrary signature was discovered by Wadsworth

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{MapT}\ m\ v & = & \mathit{SpotT} \mid \mathit{TrieT}\ (m\ v) \\
 & & \qquad\qquad\qquad (\mathit{MapT}\ m\ (\mathit{MapT}\ m\ v)) \\
 & & \qquad\qquad\qquad (\mathit{MapT}\ (\mathit{MapI}\ m)\ v) \\
\mathit{emptyT} & :: & \forall m.(\forall v.\mathit{MapT}\ m\ v) \\
\mathit{emptyT} & = & \mathit{SpotT} \\
\mathit{singleT} & :: & \forall m.(\forall v.m\ v) \to (\forall v.k \times v \to m\ v) \\
 & & \qquad \to (\forall v.\mathit{Term}\ k \times v \to \mathit{MapT}\ m\ v) \\
\mathit{singleT}\ e\ s\ (\mathit{Var}\ i, v) & = & \mathit{TrieT}\ (s\ (i, v))\ \mathit{emptyT}\ \mathit{emptyT} \\
\mathit{singleT}\ e\ s\ (\mathit{App}\ i_1\ i_2, v) & = & \mathit{TrieT}\ e\ (\mathit{singleT}\ e\ s\ (i_1, \mathit{singleT}\ e\ s\ (i_2, v)))\ \mathit{emptyT} \\
\mathit{singleT}\ e\ s\ (\mathit{Lam}\ i, v) & = & \mathit{TrieT}\ e\ \mathit{emptyT}\ (\mathit{singleT}\ \mathit{emptyI}\ (\mathit{singleI}\ e\ s)\ (i, v)) \\
\mathit{lookupT} & :: & \forall m.(\forall v.k \to m\ v \to \mathit{Maybe}\ v) \\
 & & \qquad \to (\forall v.\mathit{Term}\ k \to \mathit{MapT}\ m\ v \to \mathit{Maybe}\ v) \\
\mathit{lookupT}\ l\ i\ \mathit{SpotT} & = & \mathit{Nothing} \\
\multicolumn{3}{l}{\mathit{lookupT}\ l\ (\mathit{Var}\ i)\ (\mathit{TrieT}\ \mathit{tv}\ \mathit{ta}\ \mathit{tl})} \\
 & = & l\ i\ \mathit{tv} \\
\multicolumn{3}{l}{\mathit{lookupT}\ l\ (\mathit{App}\ i_1\ i_2)\ (\mathit{TrieT}\ \mathit{tv}\ \mathit{ta}\ \mathit{tl})} \\
 & = & (\mathit{lookupT}\ l\ i_1 \diamond \mathit{lookupT}\ l\ i_2)\ \mathit{ta} \\
\multicolumn{3}{l}{\mathit{lookupT}\ l\ (\mathit{Lam}\ i)\ (\mathit{TrieT}\ \mathit{tv}\ \mathit{ta}\ \mathit{tl})} \\
 & = & \mathit{lookupT}\ (\mathit{lookupI}\ l)\ i\ \mathit{tl} \\
\mathit{mergeT} & :: & \forall m.(\forall v.(v \to v \to v) \to (m\ v \to m\ v \to m\ v)) \\
 & & \qquad \to (\forall v.(v \to v \to v) \\
 & & \qquad\qquad \to (\mathit{MapT}\ m\ v \to \mathit{MapT}\ m\ v \to \mathit{MapT}\ m\ v)) \\
\mathit{mergeT}\ m\ c\ \mathit{SpotT}\ t' & = & t' \\
\mathit{mergeT}\ m\ c\ t\ \mathit{SpotT} & = & t \\
\multicolumn{3}{l}{\mathit{mergeT}\ m\ c\ (\mathit{TrieT}\ \mathit{tv}\ \mathit{ta}\ \mathit{tl})\ (\mathit{TrieT}\ \mathit{tv}'\ \mathit{ta}'\ \mathit{tl}')} \\
 & = & \mathit{TrieT}\ (m\ c\ \mathit{tv}\ \mathit{tv}') \\
 & & \qquad (\mathit{mergeT}\ m\ (\mathit{mergeT}\ m\ c)\ \mathit{ta}\ \mathit{ta}') \\
 & & \qquad (\mathit{mergeT}\ (\mathit{mergeI}\ m)\ c\ \mathit{tl}\ \mathit{tl}')
\end{array}
$$

Fig. 4. Generalized tries for de Bruijn terms of type *Term v*.

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{MapTI} & = & \mathit{MapT}\ \mathit{Patricia.Dict} \\
\mathit{emptyTI} & :: & \forall v.\mathit{MapTI}\ v \\
\mathit{emptyTI} & = & \mathit{emptyT} \\
\mathit{singleTI} & :: & \forall v.\mathit{Term}\ \mathit{Int} \times v \to \mathit{MapTI}\ v \\
\mathit{singleTI} & = & \mathit{singleT}\ \mathit{Patricia.empty}\ \mathit{Patricia.single} \\
\mathit{lookupTI} & :: & \forall v.\mathit{Term}\ \mathit{Int} \to \mathit{MapTI}\ v \to \mathit{Maybe}\ v \\
\mathit{lookupTI} & = & \mathit{lookupT}\ \mathit{Patricia.lookup} \\
\mathit{insertTI} & :: & \forall v.(v \to v \to v) \to \mathit{Term}\ \mathit{Int} \times v \to (\mathit{MapTI}\ v \to \mathit{MapTI}\ v) \\
\mathit{insertTI}\ c\ (i, v)\ t & = & \mathit{mergeTI}\ c\ (\mathit{singleTI}\ (i, v))\ t \\
\mathit{mergeTI} & :: & \forall v.(v \to v \to v) \to (\mathit{MapTI}\ v \to \mathit{MapTI}\ v \to \mathit{MapTI}\ v) \\
\mathit{mergeTI} & = & \mathit{mergeT}\ \mathit{Patricia.merge}
\end{array}
$$

Fig. 5. Generalized tries for non-closed de Bruijn terms of type *Term Int*.

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press

(1979) and others independently since. Connelly and 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 the concept of a many-sorted signature by its categorical counterpart, the concept of a functor.

The first implementation of generalized tries was given by Okasaki (1998) in his recent textbook on functional data structures. Tries for parameterized 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 *Term*. In the latter case, datatypes of the second-order kind are indispensable.

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

$$\textbf{data}\ Rose\ k \quad = \quad Branch\ k\ (List\ (Rose\ k)).$$

Its trie is given by

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

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

$$\textbf{data}\ GRose\ t\ k \quad = \quad GBranch\ k\ (t\ (GRose\ t\ k)).$$

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

$$\textbf{data}\ MapGR\ mt\ mk\ v \quad = \quad TrieGR\ (mk\ (mt\ (MapGR\ mt\ mk)\ v)).$$

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 that makes it possible to define polytypic values generically for all datatypes expressible in Haskell 98.

Acknowledgements

Thanks are due to Chris Okasaki for his helpful comments on an earlier draft of this paper. Suggestions by Patrik Jansson, Colin Runciman, Phil Wadler and three anonymous referees were very valuable and greatly improved the quality of the paper.

References

Bird, R. (1998) *Introduction to Functional Programming using Haskell. 2nd ed.* Prentice Hall.

Bird, R. and Meertens, L. (1998) Nested datatypes. In: Jeuring, J. (editor), *Fourth International Conference on Mathematics of Program Construction* (*MPC'98*), pp. 52–67. Marstrand, Sweden. *Lecture Notes in Computer Science* **1422**. Springer-Verlag.

Bird, R. and Paterson, R. (1999) de Bruijn notation as a nested datatype. *J. Functional Programming*, **9**(1), 77–91.

Connelly, R. H. and Morris, F. L. (1995) A generalization of the trie data structure. *Mathematical Structures in Comput. Sci.* **5**(3), 381–418.

Courcelle, B. (1983) Fundamental properties of infinite trees. *Theor. Comput. Sci.*, **25**(2), 95–169.

de Bruijn, N. G. (1972) A lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Mathematicae*, **34**, 381–392.

de la Briandais, R. (1959) File searching using variable length keys. *Proc. Western Joint Computer Conference*, **15**, pp. 295–298. AFIPS Press.

GHC Team (1999) *The Glasgow Haskell Compiler User's Guide, Version 4.04*. Available from `http://www.haskell.org/ghc/documentation.html`.

Henglein, F. (1993) Type inference with polymorphic recursion. *ACM Trans. Programming Languages and Systems*, **15**(2), 253–289.

Hinze, R. (1999a) Functional Pearl: Perfect trees and bit-reversal permutations. *J. Functional Programming*, **10**(3) 305–317.

Hinze, R. (1999b) Polytypic programming with ease (extended abstract). *4th Fuji International Symposium on Functional and Logic Programming* (*FLOPS'99*), pp. 21–36. Tsukuba, Japan. *Lecture Notes in Computer Science* **1722**. Springer-Verlag.

Jansson, P. and Jeuring, J. (1997) PolyP—a polytypic programming language extension. *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97*, pp. 470–482. Paris, France. ACM-Press.

Jansson, P. and Jeuring, J. (1999) Polytypic compact printing and parsing. In: Swierstra, S. D. (editor), *Proceedings European Symposium on Programming, ESOP'99*. *Lecture Notes in Computer Science* **1576**. Springer-Verlag.

Jeuring, J. and Jansson, P. (1996) Polytypic programming. In: Launchbury, J., Meijer, E. and Sheard, T. (editors), *Tutorial Text 2nd International School on Advanced Functional Programming*, pp. 68–114. Olympia, WA. *Lecture Notes in Computer Science* **1129**. Springer-Verlag.

Jones, M. P. (1995) Functional programming with overloading and higher-order polymorphism. *First International Spring School on Advanced Functional Programming Techniques*, pp. 97–136. *Lecture Notes in Computer Science* **925**. Springer-Verlag.

Jones, M. P. and Peterson, J. C. (1999) *Hugs 98 user manual*. Available from `http://www.haskell.org/hugs`.

Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd ed.* Addison-Wesley.

McCracken, N. J. (1984) The typechecking of programs with implicit type structure. In: Kahn, G., MacQueen, D. B. and Plotkin, G. D. (editors), *Semantics of Data Types: International symposium*, , pp. 301–315. Sophia-Antipolis, France. *Lecture Notes in Computer Science* **173**. Springer-Verlag.

Mycroft, A. (1984) Polymorphic type schemes and recursive definitions. In: Paul, M. and Robinet, B. (editors), *Proceedings of the International Symposium on Programming, 6th Colloquium*, pp. 217–228, Toulouse, France. *Lecture Notes in Computer Science* **167**. Springer-Verlag.

Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press.

Okasaki, C. and Gill, A. (1998) Fast mergeable integer maps. *The 1998 ACM SIGPLAN Workshop on ML*, pp. 77–86. Baltimore, MD.

Peyton Jones, S. (1998) *Explicit quantification in Haskell.* Available from `http://research.microsoft.com/Users/simonpj/Haskell/quantification.html`.

Peyton Jones, S. and Hughes, J. (editors) (1999) *Haskell 98 — A non-strict, purely functional language.* Available from `http://www.haskell.org/definition/`.

Thue, A. (1912) Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. *Skrifter udgivne af Videnskaps-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse*, **1**. Reprinted in Thue's "Selected Mathematical Papers", pp. 413–477 (Oslo: Universitetsforlaget, 1977).

Wadsworth, C. P. (1979) Recursive type operators which are more than type schemes. *Bulletin of the EATCS*, **8**, 87–88.

https://doi.org/10.1017/S0956796800003713 Published online by Cambridge University Press