Memo functions, polytypically!

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
`ralf@informatik.uni-bonn.de`
`http://www.informatik.uni-bonn.de/~ralf/`

Abstract. This paper presents a polytypic implementation of memo functions that are based on digital search trees. A memo function can be seen as the composition of a tabulation function that creates a memo table and a look-up function that queries the table. We show that tabulation can be derived from look-up by inverse function construction. The type of memo tables is defined by induction on the structure of argument types and is parametric with respect to the result type of memo functions. A memo table for a fixed argument type is then a functor and look-up and tabulation are natural isomorphisms. We provide simple polytypic proofs of these properties.

1 Introduction

A *memo function* [11] is like an ordinary function except that it caches previously computed values. If it is applied a second time to a particular argument, it immediately returns the cached result, rather than recomputing it. For storing arguments and results a memo function internally employs an index structure, the so-called *memo table*. In fact, a memo function can be seen as the composition of a *tabulation* function that creates a memo table and a *look-up* function that queries the table. The memo table can be implemented in a variety of ways using, for instance, hashing or comparison-based search tree schemes. These approaches, however, have their drawbacks if the argument to a memo function is a compound value such as a list or a tree. Since comparing compound values is expensive, search tree schemes based on ordering are prohibitive. Hash tables are no viable alternative as hashing compound values is difficult. Furthermore, in case of collisions values must be checked for equality (though a hash-consing garbage collector [1] may alleviate this problem). For memo functions with compound argument types *digital search trees*, also known as *tries*, are the data structure of choice. Looking up a value in a trie takes time proportional to the size of the value. In particular, the running time is independent of the number of memoized values. In combination with lazy evaluation tries provide an elegant and efficient implementation of memo functions.

This paper is a direct descendant of my earlier work on generalized tries [5], which in turn relies heavily on the framework of *polytypic programming* [7, 6, 8]. The central insight is that a trie can be considered as a *type-indexed datatype* that is defined by induction on the structure of types. The look-up function then enjoys a straightforward polytypic definition. We show that from this definition one can systematically derive its inverse, the tabulation function. Like the functions involved the derivation is parametric in the underlying datatype, the argument type of memo functions. Note that the work reported here generalizes the approach of [5] in that we define tries for arbitrary datatypes of arbitrary kinds (the precursor was restricted to types of first-order kind). A second, but minor difference is that for memo tables we require infinite tries whereas [5] was concerned with finite tries.

The rest of this paper is structured as follows. Section 2 briefly reviews the paradigm of polytypic programming. Section 3 gives polytypic definitions of memo tables and associated look-up and tabulation functions. The naturality of these functions is shown in Section 4. Finally, Section 5 concludes and points out a direction for future work.

Examples are given in the functional programming language Haskell 98 [14]. Throughout, we use Haskell as an abbreviation for Haskell 98.

2 Polytypic programming

This section briefly reviews the concept of polytypic programming. For a more thorough treatment the interested reader is referred to [7, 6, 8]. The cognoscenti may safely skip this section.

The central idea of polytypism is to provide the programmer with the ability to define a function by induction on the structure of types. Since types play a central rôle in this undertaking, let us first take a closer look at Haskell's type system. Haskell offers one basic construct for defining new types: datatype declarations. A **data** declaration takes the following general form:

$$\mathbf{data}\ D\ x_1\ \ldots\ x_m = K_1\ t_{11}\ \ldots\ t_{1m_1} \mid \cdots \mid K_n\ t_{n1}\ \ldots\ t_{nm_n}\ .$$

Here, D is the defined type constructor (the K_i are value constructors). From the perspective of language design the **data** construct is quite a monster as it comprises no less than four different features: type recursion, type abstraction, n-ary sums, and n-ary products. Thus, Haskell's type system is covered by the following language of types (we do not consider functional types, that is, no higher-order memo functions yet, but see Section 5).

$$\begin{array}{ll} \text{type variables} & a, b \\ \text{type terms} & t, u ::= 1 \mid (t + u) \mid (t \times u) \mid a \mid (t\ u) \mid (\Lambda a.t) \mid (\mu a.t) \end{array}$$

Here, 1 is the unit datatype, $t\ u$ denotes type application, $\Lambda a.t$ denotes type abstraction, and $\mu a.t$ is the least fixpoint of $\Lambda a.t$. Not every type term denotes a sensible type, consider, for instance, 1 1. To exclude these terms we require type terms to be *well-kinded*, where the language of kinds is given by

$$\text{kind terms} \qquad K, L ::= * \mid K \to L\ .$$

Here, '$*$' is the kind of manifest types such as 1; $K \to L$ is the kind of type constructors that map type constructors of kind K to those of kind L. The straightforward typing rules (or rather, 'kinding' rules) are omitted for reasons of space, but see [6, 8]. Now, given this type language we can easily translate **data** declarations into type terms: the type D defined above becomes

$$\mu D.\Lambda x_1.\dots\Lambda x_m.(t_{11} \times \cdots \times t_{1m_1}) + \cdots + (t_{n1} \times \cdots \times t_{nm_n}) \ ,$$

where $t_1 \times \cdots \times t_k = 1$ for $k = 0$. For simplicity, n-ary sums are reduced to binary sums and n-ary products to binary products.

Though the type language is quite complex, defining a polytypic value is comparatively simple. It suffices to specify cases for the three primitive type constructors 1, '+' and '$\times$'. We treat these type constructors as if they were given by the following datatype declarations.

$$
\begin{aligned}
&\mathbf{data}\ 1 &&= () \\
&\mathbf{data}\ a + b &&= \mathit{Inl}\ a \mid \mathit{Inr}\ b \\
&\mathbf{data}\ a \times b &&= (a, b)
\end{aligned}
$$

Example 1. The polytypic equality function is defined by the following equations. For clarity, the type argument is enclosed in angle brackets.

$$
\begin{aligned}
&\mathit{equal}\langle a\rangle && ::\ a \to a \to \mathit{Bool} \\
&\mathit{equal}\langle 1\rangle\ x\ y && = \mathit{True} \\
&\mathit{equal}\langle t + u\rangle\ (\mathit{Inl}\ x_1)\ (\mathit{Inl}\ x_2) && = \mathit{equal}\langle t\rangle\ x_1\ x_2 \\
&\mathit{equal}\langle t + u\rangle\ (\mathit{Inl}\ x_1)\ (\mathit{Inr}\ y_2) && = \mathit{False} \\
&\mathit{equal}\langle t + u\rangle\ (\mathit{Inr}\ y_1)\ (\mathit{Inl}\ x_2) && = \mathit{False} \\
&\mathit{equal}\langle t + u\rangle\ (\mathit{Inr}\ y_1)\ (\mathit{Inr}\ y_2) && = \mathit{equal}\langle u\rangle\ y_1\ y_2 \\
&\mathit{equal}\langle t \times u\rangle\ (x_1, y_1)\ (x_2, y_2) && = \mathit{equal}\langle t\rangle\ x_1\ x_2 \wedge \mathit{equal}\langle u\rangle\ y_1\ y_2
\end{aligned}
$$

Since 1 has only one proper element, $\mathit{equal}\langle 1\rangle\ x\ y$ trivially yields *True*. Elements of a sum are equal if they have the same constructor and the arguments of the constructor are equal. Finally, pairs are equal if the corresponding components are equal. □

It may seem surprising at first sight that a polytypic function such as *equal* is completely determined by giving cases for the three primitive type constructors. However, using standard reduction rules for type terms, that is, $(\Lambda a.t)\ u = t\,[\,a := u\,]$ and $\mu a.t = t\,[\,a := \mu a.t\,]$ every type term of kind $*$ can be reduced to a term of the form 1, $t + u$, or $t \times u$, which are exactly the cases covered by *equal*.

Example 2. The type of natural numbers is given by

$$\mathbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}\ \ .$$

Using *equal* we can test two naturals for equality.

$$
\begin{aligned}
\mathit{equal}\langle \mathit{Nat}\rangle\ (\mathit{Succ}\ \mathit{Zero})\ (\mathit{Succ}\ \mathit{Zero}) &= \mathit{equal}\langle 1 + \mathit{Nat}\rangle\ (\mathit{Succ}\ \mathit{Zero})\ (\mathit{Succ}\ \mathit{Zero}) \\
&= \mathit{equal}\langle \mathit{Nat}\rangle\ \mathit{Zero}\ \mathit{Zero} \\
&= \mathit{equal}\langle 1 + \mathit{Nat}\rangle\ \mathit{Zero}\ \mathit{Zero} \\
&= \mathit{True}
\end{aligned}
$$

Note that *Zero* equals *Inl* () and *Succ n* equals *Inr n*. □

The example suggests a simple way of implementing polytypic functions: if types are represented by an algebraic datatype (covering the cases 1, '+' and '×'), then *equal* can proceed by ordinary pattern matching. Alternatively, one can *specialize* or *partially evaluate* a polytypic value for a given closed type term. This has the advantage that passing representations of types at run-time is not necessary. The key idea for a *compositional* definition of *equal* is to mimic the structure of types on the value level. Consider, for instance, the specialization of $equal\langle t\ u\rangle$. How can we define $equal\langle t\ u\rangle$ compositionally in terms of specializations for the constituent types, $equal\langle t\rangle$ and $equal\langle u\rangle$? Now, since t is a mapping on types, the idea suggests itself that $equal\langle t\rangle$ is a mapping on equality functions. Then $equal\langle t\ u\rangle$ is given by the application of $equal\langle t\rangle$ to $equal\langle u\rangle$. In a nutshell, type abstraction is mapped to value abstraction, type application to value application, and type recursion to value recursion.

Example 3. The following equations extend the definition of *equal* given in Example 1 (note that $equal_a$ is a fresh variable associated with a).

$$\begin{aligned}
equal\langle a\rangle &= equal_a \\
equal\langle t\ u\rangle &= (equal\langle t\rangle)\ (equal\langle u\rangle) \\
equal\langle \Lambda a.t\rangle &= \lambda equal_a . equal\langle t\rangle \\
equal\langle \mu a.t\rangle &= fix\ (\lambda equal_a . equal\langle t\rangle)
\end{aligned}$$

Here, *fix* is the fixpoint operator on the value level. Note that *equal*'s type argument is no longer restricted to types of kind $*$. For that reason we must generalize its type signature:

$$equal\langle a :: K\rangle :: Equal\langle K\rangle\ a\ \ ,$$

where $Equal\langle K\rangle$ is defined by induction on the structure of kinds.

$$\begin{aligned}
Equal\langle *\rangle\ t &= t \to t \to Bool \\
Equal\langle K \to L\rangle\ t &= \forall a. Equal\langle K\rangle\ a \to Equal\langle L\rangle\ (t\ a)
\end{aligned}$$

As an example, $Equal\langle * \to *\rangle\ F = \forall a.(a \to a \to Bool) \to (F\ a \to F\ a \to Bool)$. Given these definitions we can specialize *equal* for $Nat = \mu n.1 + n$.

$$equal_{Nat} = fix\ (\lambda equal_n . equal_+\ equal_1\ equal_n)$$

where $equal_1 = equal\langle 1\rangle$ and $equal_+ = equal\langle \lambda a.\lambda b.a + b\rangle$. If we remove the abstract clothing, we obtain the familiar Haskell function

$$\begin{aligned}
&equalNat && :: Nat \to Nat \to Bool \\
&equalNat\ Zero\ Zero && = True \\
&equalNat\ Zero\ (Succ\ n_2) && = False \\
&equalNat\ (Succ\ n_1)\ Zero && = False \\
&equalNat\ (Succ\ n_1)\ (Succ\ n_2) && = equalNat\ n_1\ n_2 \quad \square
\end{aligned}$$

It is worth noting that the development is by no means special to *equal*. Rather it works for arbitrary polytypic values that are indexed by types of kind $*$.

3 Memo functions

In this section we apply the framework of polytypic programming to implement trie-based memo tables with associated look-up and tabulation functions.

$$\begin{aligned}
&\mathit{Table}\langle k :: *\rangle :: * \to * \\
&\mathit{apply}\langle k\rangle \quad :: \forall v.\mathit{Table}\langle k\rangle\ v \to (k \to v) \\
&\mathit{tabulate}\langle k\rangle \ :: \forall v.(k \to v) \to \mathit{Table}\langle k\rangle\ v
\end{aligned}$$

The type $\mathit{Table}\langle k\rangle\ v$ represents memo tables that are indexed by values of type k and store values of type v. In Section 3.1 we show how to define *Table* by induction on the structure of k. The function $\mathit{apply}\langle k\rangle$ is the associated look-up function: it takes a memo table and a key of type k and returns the associated value of type v. Its converse, $\mathit{tabulate}\langle k\rangle$, tabulates a given function with argument type k. Given this interface we can easily memoize a function of type $k \to v$:

$$\begin{aligned}
&\mathit{memo}\langle k\rangle \quad :: \forall v.(k \to v) \to (k \to v) \\
&\mathit{memo}\langle k\rangle\ \varphi = \mathit{apply}\langle k\rangle\ (\mathit{tabulate}\langle k\rangle\ \varphi)\ .
\end{aligned}$$

The memoized version of φ is simply $\mathit{memo}\langle k\rangle\ \varphi$. It is worth noting that this technique depends in an essential way on *lazy evaluation*: if the type of keys is infinite, then $\mathit{tabulate}\langle k\rangle\ \varphi$ produces a potentially infinite tree. We also require *full laziness* so that $\mathit{tabulate}\langle k\rangle\ \varphi$ is evaluated only once even if it is queried several times. Haskell meets both requirements.

3.1 Memo tables

Tries, or rather, *generalized tries* [4] enjoy a firm mathematical foundation: they are based on the *laws of exponentials*.

$$\begin{aligned}
1 \to v &\cong v \\
(k_1 + k_2) \to v &\cong (k_1 \to v) \times (k_2 \to v) \\
(k_1 \times k_2) \to v &\cong k_1 \to (k_2 \to v)
\end{aligned}$$

Note that the last equation captures the idea of *currying*. From these equations we can immediately derive a polytypic definition of *Table*.

$$\begin{aligned}
\mathit{Table}\langle 1\rangle\ v &= v \\
\mathit{Table}\langle k_1 + k_2\rangle\ v &= \mathit{Table}\langle k_1\rangle\ v \times \mathit{Table}\langle k_2\rangle\ v \\
\mathit{Table}\langle k_1 \times k_2\rangle\ v &= \mathit{Table}\langle k_1\rangle\ (\mathit{Table}\langle k_2\rangle\ v)
\end{aligned}$$

The type constructor $\mathit{Table}\langle k\rangle$ has kind $* \to *$. In fact, we will see in Section 4 that $\mathit{Table}\langle k\rangle$ satisfies the properties of a *functor*. In particular, the trie for the unit type is the identity functor, the trie for sums is a product of functors, and the trie for products is a composition of functors.

To specialize $\mathit{Table}\langle k\rangle$ for a given type term k we can apply the techniques sketched in Section 2 (though the techniques have been developed for type-indexed values they work equally well for type-indexed types). The following equations extend *Table* to arbitrary type terms of arbitrary kinds.

$$
\begin{array}{ll}
\mathit{TABLE}\langle *\rangle & = * \to * \\
\mathit{TABLE}\langle K \to L\rangle & = \mathit{TABLE}\langle K\rangle \to \mathit{TABLE}\langle L\rangle \\
\mathit{Table}\langle k :: K\rangle & ::\ \mathit{TABLE}\langle K\rangle \\
\mathit{Table}\langle a\rangle & = \mathit{table}_a \\
\mathit{Table}\langle t\ u\rangle & = (\mathit{Table}\langle t\rangle)\ (\mathit{Table}\langle u\rangle) \\
\mathit{Table}\langle \Lambda a.t\rangle & = \Lambda \mathit{table}_a.\mathit{Table}\langle t\rangle \\
\mathit{Table}\langle \mu a.t\rangle & = \mu \mathit{table}_a.\mathit{Table}\langle t\rangle
\end{array}
$$

Note that the kind of $\mathit{Table}\langle k\rangle$ depends on the kind of k. Consequently, *TABLE* is a kind-indexed kind.

Example 4. The memo table for the type of natural numbers

$$\mathbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

is an *infinite list*.

$$
\begin{array}{ll}
\mathit{Nat} & = 1 + \mathit{Nat} \\
\mathit{Table}\langle \mathit{Nat}\rangle\ v & = v \times \mathit{Table}\langle \mathit{Nat}\rangle\ v
\end{array}
$$

In Haskell notation $\mathit{Table}\langle \mathit{Nat}\rangle$ reads

$$\mathbf{data}\ \mathit{TNat}\ v = \mathit{NNat}\ v\ (\mathit{TNat}\ v)\ .$$

If we replace *NNat* by *Cons* and add a case for *Nil*, we obtain the familiar type of lists (see Example 7). Note that this instance, the use of infinite lists for memoizing functions on the natural numbers, already appears in the paper on ‘The Semantic Elegance of Applicative Languages’ by D. Turner [17]. □

Example 5. The following alternative definition of the natural numbers is based on the binary number system (using the digits 1 and 2).

$$\mathbf{data}\ \mathit{Bin} = \mathit{End} \mid \mathit{One}\ \mathit{Bin} \mid \mathit{Two}\ \mathit{Bin}$$

The associated memo table is an infinite binary tree

$$
\begin{array}{ll}
\mathit{Bin} & = 1 + \mathit{Bin} + \mathit{Bin} \\
\mathit{Table}\langle \mathit{Bin}\rangle\ v & = v \times \mathit{Table}\langle \mathit{Bin}\rangle\ v \times \mathit{Table}\langle \mathit{Bin}\rangle\ v
\end{array}
$$

and the corresponding Haskell type is given by

$$\mathbf{data}\ \mathit{TBin}\ v = \mathit{NBin}\ v\ (\mathit{TBin}\ v)\ (\mathit{TBin}\ v) \quad \square$$

Example 6. The memo table for an unlabelled binary tree

$$\mathbf{data}\ \mathit{Tree} = \mathit{Leaf} \mid \mathit{Fork}\ \mathit{Tree}\ \mathit{Tree}$$

has a somewhat mind-boggling type.

$$\begin{aligned}
\mathit{Tree} &= 1 + \mathit{Tree} \times \mathit{Tree} \\
\mathit{Table}\langle \mathit{Tree} \rangle\ v &= v \times \mathit{Table}\langle \mathit{Tree} \rangle\ (\mathit{Table}\langle \mathit{Tree} \rangle\ v)
\end{aligned}$$

Note that the two occurrences of $\mathit{Table}\langle \mathit{Tree} \rangle$ on the right-hand side are nested. Indeed, the Haskell type TTree

$$\mathbf{data}\ \mathit{TTree}\ v = \mathit{NTree}\ v\ (\mathit{TTree}\ (\mathit{TTree}\ v))$$

is an example for a so-called *nested datatype* [3]. An element of type $\mathit{TTree}\ v$ is like an infinite list except that the n-th entry has type $\mathit{TTree}^n\ v$ (a similar type appears in the seminal paper on nested datatypes [3]). □

Example 7. Finally, let us consider a parameterized datatype, the ubiquitous datatype of lists.

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

Since List is a type constructor, $\mathit{Table}\langle \mathit{List} \rangle$ is a 'higher-order' memo table that takes a trie for the base type a and yields a trie for $\mathit{List}\ a$.

$$\begin{aligned}
\mathit{List}\ a &= 1 + a \times \mathit{List}\ a \\
\mathit{Table}\langle \mathit{List} \rangle\ \mathit{tablea}\ v &= v \times \mathit{tablea}\ (\mathit{Table}\langle \mathit{List} \rangle\ \mathit{tablea}\ v)
\end{aligned}$$

The type constructor $\mathit{Table}\langle \mathit{List} \rangle$ is a so-called *generalized rose tree*. The corresponding Haskell type reads

$$\mathbf{data}\ \mathit{TList}\ \mathit{ta}\ v = \mathit{NList}\ v\ (\mathit{ta}\ (\mathit{TList}\ \mathit{ta}\ v)) \quad \Box$$

3.2 Table look-up

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

$$\begin{aligned}
&\mathit{apply}\langle k \rangle && :: \forall v. \mathit{Table}\langle k \rangle\ v \to (k \to v) \\
&\mathit{apply}\langle 1 \rangle\ t\ () && = t \\
&\mathit{apply}\langle k_1 + k_2 \rangle\ (t_1, t_2)\ (\mathit{Inl}\ i_1) && = \mathit{apply}\langle k_1 \rangle\ t_1\ i_1 \\
&\mathit{apply}\langle k_1 + k_2 \rangle\ (t_1, t_2)\ (\mathit{Inr}\ i_2) && = \mathit{apply}\langle k_2 \rangle\ t_2\ i_2 \\
&\mathit{apply}\langle k_1 \times k_2 \rangle\ t\ (i_1, i_2) && = \mathit{apply}\langle k_2 \rangle\ (\mathit{apply}\langle k_1 \rangle\ t\ i_1)\ i_2
\end{aligned}$$

The structure of *apply* becomes more visible if we swap the two value arguments (the new function is called *lookup*).

$$\begin{aligned}
&\mathit{lookup}\langle k \rangle && :: \forall v. k \to \mathit{Table}\langle k \rangle\ v \to v \\
&\mathit{lookup}\langle 1 \rangle\ () && = \mathit{id} \\
&\mathit{lookup}\langle k_1 + k_2 \rangle\ (\mathit{Inl}\ i_1) && = \mathit{lookup}\langle k_1 \rangle\ i_1 \cdot \mathit{outl} \\
&\mathit{lookup}\langle k_1 + k_2 \rangle\ (\mathit{Inr}\ i_2) && = \mathit{lookup}\langle k_2 \rangle\ i_2 \cdot \mathit{outr} \\
&\mathit{lookup}\langle k_1 \times k_2 \rangle\ (i_1, i_2) && = \mathit{lookup}\langle k_2 \rangle\ i_2 \cdot \mathit{lookup}\langle k_1 \rangle\ i_1
\end{aligned}$$

Thus, on the unit type the lookup function is the identity, on sums it selects the appropriate memo table, and on products it composes the lookup functions for the components.

The extension of *apply* works essentially as before. Applying the scheme of Section 2 we obtain

$$\begin{array}{ll}
\mathit{Apply}\langle * \rangle\ u & = \forall v.\mathit{Table}\langle u\rangle\ v \to (u \to v) \\
\mathit{Apply}\langle K \to L\rangle\ u & = \forall a.\mathit{Apply}\langle K\rangle\ a \to \mathit{Apply}\langle L\rangle\ (u\ a) \\
\mathit{apply}\langle k :: K\rangle & :: \mathit{Apply}\langle K\rangle\ k \\
\mathit{apply}\langle a\rangle & = \mathit{apply}_a \\
\mathit{apply}\langle t\ u\rangle & = (\mathit{apply}\langle t\rangle)\ (\mathit{apply}\langle u\rangle) \\
\mathit{apply}\langle \Lambda a.t\rangle & = \lambda \mathit{apply}_a.\mathit{apply}\langle t\rangle \\
\mathit{apply}\langle \mu a.t\rangle & = \mathit{fix}\ (\lambda \mathit{apply}_a.\mathit{apply}\langle t\rangle)\ .
\end{array}$$

There is one small glitch, however. Consider the type signature of $\mathit{apply}\langle F\rangle$ where F is a type constructor of kind $* \to *$.

$$\mathit{apply}\langle F\rangle :: \forall a.(\forall v.\mathit{Table}\langle a\rangle\ v \to (a \to v)) \to (\forall w.\mathit{Table}\langle F\ a\rangle\ w \to (F\ a \to w))$$

The type signature contains two occurrences of *Table*. Of course, if we want to specialize $\mathit{apply}\langle F\rangle$ for a given F, we must specialize its type signature, as well. To this end we replace $\mathit{Table}\langle F\ a\rangle$ by $\mathit{Table}\langle F\rangle\ (\mathit{Table}\langle a\rangle)$ and generalize $\mathit{Table}\langle a\rangle$ to a fresh type variable, say, ta.

$$\begin{array}{l}
\mathit{apply}\langle F\rangle :: \forall ta\ a.(\forall v.ta\ v \to (a \to v)) \\
\qquad\qquad\qquad \to (\forall w.\mathit{Table}\langle F\rangle\ ta\ w \to (F\ a \to w))
\end{array}$$

The following refined definition of *Apply* captures this generalization.

$$\begin{array}{ll}
\mathit{Apply}\langle * \rangle\ tu\ u & = \forall v.tu \to (u \to v) \\
\mathit{Apply}\langle K \to L\rangle\ tu\ u & = \forall ta\ a.\mathit{Apply}\langle K\rangle\ ta\ a \to \mathit{Apply}\langle L\rangle\ (tu\ ta)\ (u\ a)
\end{array}$$

It is not hard to see that $\mathit{Apply}\langle K\rangle\ (\mathit{Table}\langle k\rangle)\ k$ is a valid type of $\mathit{apply}\langle k :: K\rangle$.

Example 8. Querying a memo table for the natural numbers works as follows.

$$\begin{array}{ll}
\mathit{applyNat} & :: \forall v.\mathit{TNat}\ v \to (\mathit{Nat} \to v) \\
\mathit{applyNat}\ (\mathit{NNat}\ tz\ ts)\ \mathit{Zero} & = tz \\
\mathit{applyNat}\ (\mathit{NNat}\ tz\ ts)\ (\mathit{Succ}\ n) & = \mathit{applyNat}\ ts\ n
\end{array}$$

Recall that elements of *TNat* are infinite lists. Consequently, *applyNat* corresponds to list indexing (written (!!) in Haskell). □

Example 9. The look-up function for binary numbers corresponds to tree indexing (a binary number is interpreted as a path into a binary tree).

$$\begin{array}{ll}
\mathit{applyBin} & :: \forall v.\mathit{TBin}\ v \to (\mathit{Bin} \to v) \\
\mathit{applyBin}\ (\mathit{NBin}\ tn\ to\ tt)\ \mathit{End} & = tn \\
\mathit{applyBin}\ (\mathit{NBin}\ tn\ to\ tt)\ (\mathit{One}\ b) & = \mathit{applyBin}\ to\ b \\
\mathit{applyBin}\ (\mathit{NBin}\ tn\ to\ tt)\ (\mathit{Two}\ b) & = \mathit{applyBin}\ tt\ b \quad \Box
\end{array}$$

Example 10. The look-up function for memo tables of type *TTree* is somewhat hard to grasp. Its definition is, however, a simple instance of the general scheme.

$$
\begin{aligned}
&\mathit{applyTree} && :: \forall v.\mathit{TTree}\ v \to (\mathit{Tree} \to v)\\
&\mathit{applyTree}\ (\mathit{NTree}\ \mathit{tl}\ \mathit{tf})\ \mathit{Leaf} && = \mathit{tl}\\
&\mathit{applyTree}\ (\mathit{NTree}\ \mathit{tl}\ \mathit{tf})\ (\mathit{Fork}\ l\ r) && = \mathit{applyTree}\ (\mathit{applyTree}\ \mathit{tf}\ l)\ r
\end{aligned}
$$

Since *TTree* is a nested type, *applyTree* requires *polymorphic recursion* [12]. □

Example 11. As the final example, consider the look-up function for lists.

$$
\begin{aligned}
&\mathit{applyList} :: \forall \mathit{ta}\ a.(\forall v.\mathit{ta}\ v \to (a \to v))\\
&\qquad\qquad\qquad \to (\forall w.\mathit{TList}\ \mathit{ta}\ w \to (\mathit{List}\ a \to w))\\
&\mathit{applyList}\ \mathit{applya}\ (\mathit{NList}\ \mathit{tn}\ \mathit{tc})\ \mathit{Nil}\\
&\qquad = \mathit{tn}\\
&\mathit{applyList}\ \mathit{applya}\ (\mathit{NList}\ \mathit{tn}\ \mathit{tc})\ (\mathit{Cons}\ a\ \mathit{as})\\
&\qquad = \mathit{applyList}\ \mathit{applya}\ (\mathit{applya}\ \mathit{tc}\ a)\ \mathit{as}
\end{aligned}
$$

Since *List* is a parametric type, *applyList* is a 'higher-order' look-up function that takes a look-up function for the base type a and yields a lookup function for *List* a. Note that *applyList* has a rank-2 type signature [10], which is not legal Haskell. However, recent versions of the Glasgow Haskell Compiler GHC [16] and the Haskell interpreter Hugs [9] support rank-2 types. □

3.3 Tabulation

Tabulation is the inverse of look-up and, in fact, we can derive its definition by inverse function construction. For the derivation we use a slight reformulation of *apply* that allows for more structured calculations ('$\triangledown$' is the *junk* combinator, see, for instance [2]).

$$
\begin{aligned}
&\mathit{apply}\langle k\rangle && :: \forall v.\mathit{Table}\langle k\rangle\ v \to (k \to v)\\
&\mathit{apply}\langle 1\rangle\ t && = \lambda().t\\
&\mathit{apply}\langle k_1 + k_2\rangle\ t && = \mathit{apply}\langle k_1\rangle\ (\mathit{outl}\ t) \triangledown \mathit{apply}\langle k_2\rangle\ (\mathit{outr}\ t)\\
&\mathit{apply}\langle k_1 \times k_2\rangle\ t && = \mathit{uncurry}\ (\mathit{apply}\langle k_2\rangle \cdot \mathit{apply}\langle k_1\rangle\ t)
\end{aligned}
$$

We specify *tabulate* as the right inverse of *apply*.

$$\mathit{apply}\langle k\rangle\ (\mathit{tabulate}\langle k\rangle\ \varphi) = \varphi$$

Since we are seeking a polytypic definition of *tabulate*, we proceed by case analysis on k. **Case** $k = 1$:

$$
\begin{aligned}
&\quad \mathit{apply}\langle 1\rangle\ (\mathit{tabulate}\langle 1\rangle\ \varphi) = \varphi\\
\iff &\quad \{\ \text{definition } \mathit{apply}\langle 1\rangle\ \}\\
&\quad \lambda().\mathit{tabulate}\langle 1\rangle\ \varphi = \varphi\\
\iff &\quad \{\ \text{extensionality: } \psi_1 = \psi_2 :: 1 \to A \iff \psi_1\ () = \psi_2\ () :: A\ \}\\
&\quad \mathit{tabulate}\langle 1\rangle\ \varphi = \varphi\ ()\ \ .
\end{aligned}
$$

Case $k = k_1 + k_2$: let $t = \mathit{tabulate}\langle k_1 + k_2\rangle\ \varphi$, then

$$
\begin{aligned}
& \mathit{apply}\langle k_1 + k_2\rangle\ t = \varphi \\
\iff & \quad \{ \text{ definition } \mathit{apply}\langle k_1 + k_2\rangle \ \} \\
& \mathit{apply}\langle k_1\rangle\ (\mathit{outl}\ t) \mathbin{\triangledown} \mathit{apply}\langle k_2\rangle\ (\mathit{outr}\ t) = \varphi \\
\iff & \quad \{ \text{ coproducts: } \psi = \psi_1 \mathbin{\triangledown} \psi_2 \iff \psi \cdot \mathit{Inl} = \psi_1 \wedge \psi \cdot \mathit{Inr} = \psi_2 \ \} \\
& \mathit{apply}\langle k_1\rangle\ (\mathit{outl}\ t) = \varphi \cdot \mathit{Inl} \wedge \mathit{apply}\langle k_2\rangle\ (\mathit{outr}\ t) = \varphi \cdot \mathit{Inr} \\
\Longleftarrow & \quad \{ \text{ specification } \} \\
& \mathit{outl}\ t = \mathit{tabulate}\langle k_1\rangle\ (\varphi \cdot \mathit{Inl}) \wedge \mathit{outr}\ t = \mathit{tabulate}\langle k_2\rangle\ (\varphi \cdot \mathit{Inr}) \\
\iff & \quad \{ \text{ surjective pairing: } z = (x_1, x_2) \iff \mathit{outl}\ z = x_1 \wedge \mathit{outr}\ z = x_2 \ \} \\
& t = (\mathit{tabulate}\langle k_1\rangle\ (\varphi \cdot \mathit{Inl}), \mathit{tabulate}\langle k_2\rangle\ (\varphi \cdot \mathit{Inr})) \ .
\end{aligned}
$$

Note that we use both the universal property of coproducts and the universal property of products (of which surjective pairing is a special case). **Case** $k = k_1 \times k_2$: let $t = \mathit{tabulate}\langle k_1 \times k_2\rangle\ \varphi$, then

$$
\begin{aligned}
& \mathit{apply}\langle k_1 \times k_2\rangle\ t = \varphi \\
\iff & \quad \{ \text{ definition } \mathit{apply}\langle k_1 \times k_2\rangle \ \} \\
& \mathit{uncurry}\ (\mathit{apply}\langle k_2\rangle \cdot \mathit{apply}\langle k_1\rangle\ t) = \varphi \\
\Longleftarrow & \quad \{ \text{ exponentials: } \mathit{uncurry}\ (\mathit{curry}\ \psi) = \psi \ \} \\
& \mathit{apply}\langle k_2\rangle \cdot \mathit{apply}\langle k_1\rangle\ t = \mathit{curry}\ \varphi \\
\Longleftarrow & \quad \{ \text{ specification } \} \\
& \mathit{apply}\langle k_1\rangle\ t = \mathit{tabulate}\langle k_2\rangle \cdot \mathit{curry}\ \varphi \\
\Longleftarrow & \quad \{ \text{ specification } \} \\
& t = \mathit{tabulate}\langle k_1\rangle\ (\mathit{tabulate}\langle k_2\rangle \cdot \mathit{curry}\ \varphi) \ .
\end{aligned}
$$

To summarize, we have calculated the following definition of *tabulate*.

$$
\begin{aligned}
& \mathit{tabulate}\langle k\rangle && :: \forall v.(k \to v) \to \mathit{Table}\langle k\rangle\ v \\
& \mathit{tabulate}\langle 1\rangle\ \varphi && = \varphi\ () \\
& \mathit{tabulate}\langle k_1 + k_2\rangle\ \varphi && = (\mathit{tabulate}\langle k_1\rangle\ (\varphi \cdot \mathit{Inl}), \mathit{tabulate}\langle k_2\rangle\ (\varphi \cdot \mathit{Inr})) \\
& \mathit{tabulate}\langle k_1 \times k_2\rangle\ \varphi && = \mathit{tabulate}\langle k_1\rangle\ (\mathit{tabulate}\langle k_2\rangle \cdot \mathit{curry}\ \varphi)
\end{aligned}
$$

The last equation becomes more readable if we convert it into a pointwise style.

$$
\mathit{tabulate}\langle k_1 \times k_2\rangle\ \varphi = \mathit{tabulate}\langle k_1\rangle\ (\lambda i_1 . \mathit{tabulate}\langle k_2\rangle\ (\lambda i_2 . \varphi\ (i_1, i_2)))
$$

The extension of *tabulate* to arbitrary type terms, which works exactly as for *apply*, is omitted for reasons of space.

Two points are in order.

First, the second calculation makes essential use of the universal property of coproducts. Alas, Haskell's natural semantic model, the category *Cpo* of pointed, complete partial orders and continuous functions, has no categorical coproduct.

In other words, in Haskell $\mathit{apply}\langle k\rangle\ (\mathit{tabulate}\langle k\rangle\ \varphi) = \varphi$ is only valid for so-called *hyper-strict* functions that completely evaluate their arguments. In the context of a lazy language this need for hyper-strictness is somewhat ironic. The intuition is that *all* information about the result of a memoized function is in the leaves of the corresponding trie.

Note that an appropriate theoretical setting for the calculations is the category $\mathit{Cpo}_\perp$ of pointed, complete partial orders and *strict* continuous functions, which has categorical products (the cartesian product '$\times$'), categorical coproducts (the coalesced sum '$\oplus$') and is monoidally closed (the smash product '$\otimes$' and the space '$\multimap$' of strict continuous functions form a monoidal closure[1]). Thus, memo tables are actually based on the following isomorphisms:

$$\begin{aligned}
1 \multimap v &\cong v \\
(k_1 \oplus k_2) \multimap v &\cong (k_1 \multimap v) \times (k_2 \multimap v) \\
(k_1 \otimes k_2) \multimap v &\cong k_1 \multimap (k_2 \multimap v)\ ,
\end{aligned}$$

where $1 = \{\perp, ()\}$. The isomorphisms make precise that memoization operates on strict functions but its implementation requires lazy evaluation: a trie for a 'strict' sum is a 'lazy' pair of tries. We could maintain this distinction in Haskell using strictness annotations (*TNat* is really the memo table for the flat domain $\mathbb{N}_\perp$ given by **data** $\mathit{Nat} = \mathit{Zero} \mid\ !\mathit{Succ}$) but we refrain from being that pedantic.

Second, the calculations show that tabulation is the right inverse of look-up. The converse can be shown using a straightforward *fixpoint induction*. We require fixpoint induction in order to cope with recursive types. That said it becomes clear that the case $k = 0$, where $0 = \{\perp\}$ is the 'bottom' type, is missing in the derivation above. Fortunately, $\mathit{apply}\langle 0\rangle\ (\mathit{tabulate}\langle 0\rangle\ \varphi) = \varphi$ holds trivially since 0 is the initial object in $\mathit{Cpo}_\perp$, that is, for each type V there is a unique *strict* function of type $0 \to V$.

Example 12. The tabulation function for natural numbers is a one-liner.

$$\begin{aligned}
&\mathit{tabulateNat} &&:: \forall v.(\mathit{Nat} \to v) \to \mathit{TNat}\ v \\
&\mathit{tabulateNat}\ \varphi &&= \mathit{NNat}\ (\varphi\ \mathit{Zero})\ (\mathit{tabulateNat}\ (\varphi \cdot \mathit{Succ}))
\end{aligned}$$

The standard toy example for memoization is the Fibonacci function.

$$\begin{aligned}
&\mathit{fib} &&:: \mathit{Nat} \to \mathit{Nat} \\
&\mathit{fib}\ \mathit{Zero} &&= \mathit{Zero} \\
&\mathit{fib}\ (\mathit{Succ}\ \mathit{Zero}) &&= \mathit{Succ}\ \mathit{Zero} \\
&\mathit{fib}\ (\mathit{Succ}\ (\mathit{Succ}\ n)) &&= \mathit{fib}\ n + \mathit{fib}\ (\mathit{Succ}\ n)
\end{aligned}$$

[1] Monoidal closure is similar to cartesian closure except that the product (here, the smash product) is not a categorical product but a *tensor product*.

Its time complexity can be improved from exponential to quadratic if the recursive calls are replaced by table lookups.

$$
\begin{array}{lcl}
\mathit{fib} & :: & \mathit{Nat} \to \mathit{Nat} \\
\mathit{fib}\ \mathit{Zero} & = & \mathit{Zero} \\
\mathit{fib}\ (\mathit{Succ}\ \mathit{Zero}) & = & \mathit{Succ}\ \mathit{Zero} \\
\mathit{fib}\ (\mathit{Succ}\ (\mathit{Succ}\ n)) & = & \mathit{memo\text{-}fib}\ n + \mathit{memo\text{-}fib}\ (\mathit{Succ}\ n) \\
\mathit{memo\text{-}fib} & :: & \mathit{Nat} \to \mathit{Nat} \\
\mathit{memo\text{-}fib} & = & \mathit{applyNat}\ (\mathit{tabulateNat}\ \mathit{fib}) \quad \square
\end{array}
$$

Example 13. Tabulating a function of type $Bin \to V$ is equally easy.

$$
\begin{array}{lcl}
\mathit{tabulateBin} & :: & \forall v.(\mathit{Bin} \to v) \to \mathit{TBin}\ v \\
\mathit{tabulateBin}\ \varphi & = & \mathit{NBin}\ (\varphi\ \mathit{End})\ (\mathit{tabulateBin}\ (\varphi \cdot \mathit{One}))\ (\mathit{tabulateBin}\ (\varphi \cdot \mathit{Two}))
\end{array}
$$

$\square$

Example 14. Like its inverse *tabulateTree* requires polymorphic recursion (note that $\lambda x \to e$ is Haskell notation for the lambda abstraction $\lambda x.e$).

$$
\begin{array}{lcl}
\mathit{tabulateTree} & :: & \forall v.(\mathit{Tree} \to v) \to \mathit{TTree}\ v \\
\mathit{tabulateTree}\ \varphi & = & \mathit{NTree}\ (\varphi\ \mathit{Leaf})\ (\mathit{tabulateTree}\ (\lambda l \to \\
& & \qquad \mathit{tabulateTree}\ (\lambda r \to \varphi\ (\mathit{Fork}\ l\ r)))) \quad \square
\end{array}
$$

Example 15. Finally, for parametric lists we obtain a 'higher-order' tabulation function.

$$
\begin{array}{l}
\mathit{tabulateList} :: \forall ta\ a.(\forall v.(a \to v) \to ta\ v) \\
\qquad\qquad \to (\forall w.(\mathit{List}\ a \to w) \to \mathit{TList}\ ta\ w) \\
\mathit{tabulateList}\ \mathit{tabulatea}\ \varphi \\
\qquad = \mathit{NList}\ (\varphi\ \mathit{Nil})\ (\mathit{tabulatea}\ (\lambda a \to \\
\qquad\qquad \mathit{tabulateList}\ \mathit{tabulatea}\ (\lambda as \to \varphi\ (\mathit{Cons}\ a\ as))))
\end{array}
$$

Using *TList* we can memoize functions that operate on lists. The following dynamic programming problem, *optimal matrix multiplication*, may serve as an example. Given a sequence of matrix dimensions $[d_0, \ldots, d_n]$, the problem is to find the least cost for multiplying out a sequence of matrices $M_1 * \cdots * M_n$ where the dimension of M_i is $d_{i-1} \times d_i$. We assume that multiplying an $i \times j$ matrix by an $j \times k$ matrix costs $i * j * k$. The following Haskell program implements a straightforward, but exponential solution.

$$
\begin{array}{l}
\mathit{cost} \qquad\qquad :: \mathit{List}\ \mathit{Nat} \to \mathit{Nat} \\
\mathit{cost}\ d \\
\quad \mid n \leqslant 1 \qquad = 0 \\
\quad \mid \mathit{otherwise} = \mathit{minimum}\ [\mathit{cost}\ (\mathit{take}\ (i+1)\ d) \\
\qquad\qquad\qquad\qquad + d\ !!\ 0 * d\ !!\ i * d\ !!\ n \\
\qquad\qquad\qquad\qquad + \mathit{cost}\ (\mathit{drop}\ i\ d) \mid i \leftarrow [1 \mathinner{..} n-1]] \\
\quad \mathbf{where}\ n \quad = \mathit{length}\ d - 1
\end{array}
$$

Memoizing the recursive calls improves the complexity from exponential to polynomial in the size of the input (the modified version of *cost* is omitted for reasons of space).

$$\begin{aligned}
&\mathit{memo\text{-}cost} :: \mathit{List}\ \mathit{Nat} \to \mathit{Nat}\\
&\mathit{memo\text{-}cost} = (\mathit{applyList}\ \mathit{applyNat})\ ((\mathit{tabulateList}\ \mathit{tabulateNat})\ \mathit{cost})
\end{aligned}$$

An ad-hoc variant of this code appears in [13]. □

Example 16. The function *memo-cost* defined in the previous example maintains a global memo table. This comes at a considerable cost: recall that functions on the natural numbers are memoized using infinite lists and note that the matrix dimensions $d_0, \ldots, d_n$ index these lists. A more efficient alternative both in time and in space is to maintain a local memo table.

$$\begin{array}{ll}
\mathit{cost} & :: \mathit{List}\ \mathit{Int} \to \mathit{Int}\\
\mathit{cost}\ d & = \mathit{memo\text{-}c}\ (0, n)\\
\quad \mathbf{where} & \\
\quad n & = \mathit{length}\ d - 1\\
\quad c & :: (\mathit{Nat}, \mathit{Nat}) \to \mathit{Int}\\
\quad c\ (i, j) & \\
\qquad \mid i + 1 \geqslant j & = 0\\
\qquad \mid \mathit{otherwise} & = \mathit{minimum}\ [\, \mathit{memo\text{-}c}\ (i, k)\\
 & \qquad\qquad + d\ !!\ i * d\ !!\ k * d\ !!\ j\\
 & \qquad\qquad + \mathit{memo\text{-}c}\ (k, j) \mid k \leftarrow [i + 1 \mathinner{..} j - 1]]\\
\quad \mathit{memo\text{-}c} & :: (\mathit{Nat}, \mathit{Nat}) \to \mathit{Int}\\
\quad \mathit{memo\text{-}c}\ (i, j) & = \mathit{applyNat}\ (\mathit{applyNat}\ (\\
 & \qquad \mathit{tabulateNat}\ (\lambda i' \to \mathit{tabulateNat}\ (\lambda j' \to c\ (i', j'))))\ i)\ j
\end{array}$$

Since the sequence of matrix dimensions d is fixed in the body of *cost*, sublists of d can be represented by pairs of list indices. Consequently, a much smaller memo table suffices: *memo-c* uses a table of type *TNat* (*TNat Int*) that is indexed by pairs of list indices (which are small) rather than by sequences of matrix dimensions (which may be be very large). The resulting code corresponds closely to the standard dynamic programming solution, see, for instance [15]. □

4 Properties

For a fixed k, the type constructor $\mathit{Table}\langle k\rangle$ satisfies the properties of a functor (it is an endo functor of $\mathcal{C}po_\perp$). Its functorial action on arrows is given by

$$\begin{array}{ll}
\mathit{table}\langle k\rangle & :: \forall v\ w.(v \to w) \to (\mathit{Table}\langle k\rangle\ v \to \mathit{Table}\langle k\rangle\ w)\\
\mathit{table}\langle 1\rangle\ \varphi & = \varphi\\
\mathit{table}\langle k_1 + k_2\rangle\ \varphi & = \mathit{table}\langle k_1\rangle\ \varphi \times \mathit{table}\langle k_2\rangle\ \varphi\\
\mathit{table}\langle k_1 \times k_2\rangle\ \varphi & = \mathit{table}\langle k_1\rangle\ (\mathit{table}\langle k_2\rangle\ \varphi)\ .
\end{array}$$

The functor laws

$$\begin{aligned} table\langle k\rangle\ id &= id \\ table\langle k\rangle\ (\varphi \cdot \psi) &= table\langle k\rangle\ \varphi \cdot table\langle k\rangle\ \psi \end{aligned}$$

can be shown using straightforward fixpoint inductions, see, for instance [7].

The functions $apply\langle k\rangle$ and $tabulate\langle k\rangle$ are then natural isomorphisms between $(k \rightarrow)$ and $Table\langle k\rangle$. Note that the functor $(k \rightarrow)$ is sometimes written $(-)^k$. Its functorial action is postcomposition given by $post\ \varphi = curry\ (\varphi \cdot eval)$ where $eval$ is function application. The naturality conditions are

$$\begin{aligned} apply\langle k\rangle \cdot table\langle k\rangle\ \varphi &= post\ \varphi \cdot apply\langle k\rangle \\ tabulate\langle k\rangle \cdot post\ \varphi &= table\langle k\rangle\ \varphi \cdot tabulate\langle k\rangle\ . \end{aligned}$$

The proofs below are based on the following pointwise variants.

$$\begin{aligned} apply\langle k\rangle\ (table\langle k\rangle\ \varphi\ t) &= \varphi \cdot apply\langle k\rangle\ t \\ tabulate\langle k\rangle\ (\varphi \cdot \psi) &= table\langle k\rangle\ \varphi\ (tabulate\langle k\rangle\ \psi) \end{aligned}$$

An immediate consequence of the second naturality property is, for instance,

$$tabulate\langle k\rangle\ \varphi = table\langle k\rangle\ \varphi\ (tabulate\langle k\rangle\ id)\ .$$

Thus, instead of tabulating φ we can tabulate id and then map φ on the resulting memo table. Since some types allow for a more efficient implementation of $tabulate\langle k\rangle\ id$, applying the law from left to right may be an optimization. We prove $apply\langle k\rangle\ (table\langle k\rangle\ \varphi\ t) = \varphi \cdot apply\langle k\rangle\ t$ by fixpoint induction on k. The second naturality property then follows immediately since $apply\langle k\rangle$ and $tabulate\langle k\rangle$ are mutually inverse. **Case** $k = 0$: the proposition holds trivially for strict φ since polytypic functions are strict in their type arguments, that is, $apply\langle 0\rangle = \bot$ and $tabulate\langle 0\rangle = \bot$. **Case** $k = 1$:

$$\begin{aligned} & apply\langle 1\rangle\ (table\langle 1\rangle\ \varphi\ t) \\ = \quad & \{\ \text{definition } apply\langle 1\rangle\ \} \\ & \lambda().table\langle 1\rangle\ \varphi\ t \\ = \quad & \{\ \text{definition } table\langle 1\rangle\ \} \\ & \lambda().\varphi\ t \\ = \quad & \{\ \text{extensionality: } \psi_1 = \psi_2 :: 1 \rightarrow A \Longleftrightarrow \psi_1\ () = \psi_2\ () :: A\ \} \\ & \varphi \cdot (\lambda().t) \\ = \quad & \{\ \text{definition } apply\langle 1\rangle\ \} \\ & \varphi \cdot apply\langle 1\rangle\ t\ . \end{aligned}$$

Case $k = k_1 + k_2$:

$$apply\langle k_1 + k_2\rangle\ (table\langle k_1 + k_2\rangle\ \varphi\ t)$$

$$
\begin{aligned}
= \quad & \{ \text{ definition } \mathit{apply}\langle k_1 + k_2\rangle \ \} \\
& \mathit{apply}\langle k_1\rangle\ (\mathit{outl}\ (\mathit{table}\langle k_1 + k_2\rangle\ \varphi\ t)) \mathbin{\nabla} \mathit{apply}\langle k_2\rangle\ (\mathit{outr}\ (\mathit{table}\langle k_1 + k_2\rangle\ \varphi\ t)) \\
= \quad & \{ \text{ definition } \mathit{table}\langle k_1 + k_2\rangle, \\
& \qquad \mathit{outl} \cdot (\psi_1 \times \psi_2) = \psi_1 \cdot \mathit{outl} \text{ and } \mathit{outr} \cdot (\psi_1 \times \psi_2) = \psi_2 \cdot \mathit{outr} \ \} \\
& \mathit{apply}\langle k_1\rangle\ (\mathit{table}\langle k_1\rangle\ \varphi\ (\mathit{outl}\ t)) \mathbin{\nabla} \mathit{apply}\langle k_2\rangle\ (\mathit{table}\langle k_2\rangle\ \varphi\ (\mathit{outr}\ t)) \\
= \quad & \{ \text{ ex hypothesi } \} \\
& (\varphi \cdot \mathit{apply}\langle k_1\rangle\ (\mathit{outl}\ t)) \mathbin{\nabla} (\varphi \cdot \mathit{apply}\langle k_2\rangle\ (\mathit{outr}\ t)) \\
= \quad & \{ \text{ coproduct fusion law: } \psi \cdot (\psi_1 \mathbin{\nabla} \psi_2) = (\psi \cdot \psi_1) \mathbin{\nabla} (\psi \cdot \psi_2) \ \} \\
& \varphi \cdot (\mathit{apply}\langle k_1\rangle\ (\mathit{outl}\ t) \mathbin{\nabla} \mathit{apply}\langle k_2\rangle\ (\mathit{outr}\ t)) \\
= \quad & \{ \text{ definition } \mathit{apply}\langle k_1 + k_2\rangle \ \} \\
& \varphi \cdot \mathit{apply}\langle k_1 + k_2\rangle\ t \quad .
\end{aligned}
$$

Case $k = k_1 \times k_2$:

$$
\begin{aligned}
& \mathit{apply}\langle k_1 \times k_2\rangle\ (\mathit{table}\langle k_1 \times k_2\rangle\ \varphi\ t) \\
= \quad & \{ \text{ definition } \mathit{apply}\langle k_1 \times k_2\rangle \ \} \\
& \mathit{uncurry}\ (\mathit{apply}\langle k_2\rangle \cdot \mathit{apply}\langle k_1\rangle\ (\mathit{table}\langle k_1 \times k_2\rangle\ \varphi\ t)) \\
= \quad & \{ \text{ definition } \mathit{table}\langle k_1 \times k_2\rangle \ \} \\
& \mathit{uncurry}\ (\mathit{apply}\langle k_2\rangle \cdot \mathit{apply}\langle k_1\rangle\ (\mathit{table}\langle k_1\rangle\ (\mathit{table}\langle k_2\rangle\ \varphi)\ t)) \\
= \quad & \{ \text{ ex hypothesi } \} \\
& \mathit{uncurry}\ (\mathit{apply}\langle k_2\rangle \cdot \mathit{table}\langle k_2\rangle\ \varphi \cdot \mathit{apply}\langle k_1\rangle\ t) \\
= \quad & \{ \text{ ex hypothesi } \} \\
& \mathit{uncurry}\ (\mathit{post}\ \varphi \cdot \mathit{apply}\langle k_2\rangle \cdot \mathit{apply}\langle k_1\rangle\ t) \\
= \quad & \{ \text{ proof obligation, see below } \} \\
& \varphi \cdot \mathit{uncurry}\ (\mathit{apply}\langle k_2\rangle \cdot \mathit{apply}\langle k_1\rangle\ t) \\
= \quad & \{ \text{ definition } \mathit{apply}\langle k_1 \times k_2\rangle \ \} \\
& \varphi \cdot \mathit{apply}\langle k_1 \times k_2\rangle\ t \quad .
\end{aligned}
$$

It remains to show $\varphi \cdot \mathit{uncurry}\ f = \mathit{uncurry}\ (\mathit{post}\ \varphi \cdot f)$, which is equivalent to $\mathit{curry}\ (\varphi \cdot \mathit{uncurry}\ f) = \mathit{post}\ \varphi \cdot f$.

$$
\begin{aligned}
& \mathit{curry}\ (\varphi \cdot \mathit{uncurry}\ f) \\
= \quad & \{ \text{ definition } \mathit{uncurry} \ \} \\
& \mathit{curry}\ (\varphi \cdot \mathit{eval} \cdot (f \times \mathit{id})) \\
= \quad & \{ \text{ curry fusion law: } \mathit{curry}\ \psi \cdot g = \mathit{curry}\ (\psi \cdot (g \times \mathit{id})) \ \} \\
& \mathit{curry}\ (\varphi \cdot \mathit{eval}) \cdot f \\
= \quad & \{ \text{ definition } \mathit{post} \ \} \\
& \mathit{post}\ \varphi \cdot f
\end{aligned}
$$

5 Conclusion and future work

Memo functions make an interesting case study in polytypic programming. In implementing trie-based memo functions we have encountered kind-indexed kinds (TABLE), kind-indexed types (Apply), type-indexed types (Table), and type-indexed values (apply). It is quite remarkable that all of these concepts show up in a single application.

A direction for future work suggests itself. It remains to extend memoization to higher-order functions. Recall that we have based tries on the law of exponentials. Unfortunately, there is no obvious way of rewriting the function space $(k_1 \to k_2) \to v$. A possible way out of this dilemma is to apply memoization 'recursively': since $k_1 \to k_2 \cong \mathit{Table}\langle k_1 \rangle\ k_2$, we may set

$$\mathit{Table}\langle k_1 \to k_2 \rangle\ v = \mathit{Table}\langle \mathit{Table}\langle k_1 \rangle\ k_2 \rangle\ v = \mathit{Table}\langle \mathit{Table}\langle k_1 \rangle \rangle\ (\mathit{Table}\langle k_2 \rangle)\ v$$

The author is currently exploring this approach.

Acknowledgement

I am grateful to Colin Runciman for stimulating the research reported in this paper ("What about the function $\mathit{build}\langle k \rangle :: \forall v.(k \to v) \to \mathit{Map}\langle k \rangle\ v$ which transfers a function to trie form?"). Thanks are furthermore due to three anonymous referees for many valuable comments.

References

1. Andrew W. Appel and Marcelo J. R. Goncalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, February 1993.
2. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming — An Introduction —. In S. Doaitse Swierstra, Pedro R. Henriques, and Jose N. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming, Braga, Portugal*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115, Berlin, 1999. Springer-Verlag.
3. Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, June 1998.
4. Richard H. Connelly and F. Lockwood Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, September 1995.
5. Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 2000. Accepted for publication.
6. Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.

7. Ralf Hinze. Polytypic programming with ease, January 2000. In submission.
8. Ralf Hinze. Polytypic values possess polykinded types. In Phil Wadler, editor, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000), July 3-5, 2000*, July 2000.
9. M.P. Jones and J.C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs`.
10. Daniel Leivant. Polymorphic type inference. In *Proc. 10th Symposium on Principles of Programming Languages*, 1983.
11. Donald Michie. "Memo" functions and machine learning. *Nature*, (218):19–22, April 1968.
12. Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228, 1984.
13. Michael J. O'Donnell. *Equational Logic as a Programming Language*. Foundations of Computing Series. MIT Press, Cambridge, Mass., 1985.
14. Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999.
Available from `http://www.haskell.org/definition/`.
15. Fethi Rabhi and Guy Lapalme. *Algorithms: a Functional Programming Approach*. Addison-Wesley Publishing Company, second edition, 1999.
16. The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 4.04*, September 1999.
Available from `http://www.haskell.org/ghc/documentation.html`.
17. David Turner. The Semantic Elegance of Applicative Languages. In *Functional Programming Languages and Computer Architecture (FPCA '81), Portsmouth, New Hampshire*, pages 85–92. ACM, New York, October 1981.