MEMO FUNCTIONS, POLYTYPICALLY!

RALF HINZE

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

July, 2000

(Pick the slides at `.../~ralf/talks.html#T14`.)

Say, you want to memoize the function

$$\boxed{\varphi :: Nat \times Nat \to V}$$

so that it caches previously computed values.

Let us assume that we already have a memo table for caching functions on the natural numbers:

$$\begin{aligned}
&\mathbf{data}\ \mathit{TableNat}\ v\\
&\mathit{applyNat} \quad\ :: \forall v.\,\mathit{TableNat}\ v \to (\mathit{Nat} \to v)\\
&\mathit{tabulateNat} :: \forall v.(\mathit{Nat} \to v) \to \mathit{TableNat}\ v.
\end{aligned}$$

Given this interface we can memoize φ as follows

$$
\begin{array}{l}
\mathit{memo}\text{-}\varphi :: \mathit{Nat} \times \mathit{Nat} \to V \\
\mathit{memo}\text{-}\varphi = \mathit{apply}\ (\mathit{tabulate}\ \varphi)
\end{array}
$$

where

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{type}\ \mathit{Table}\ v = \mathit{TableNat}\ (\mathit{TableNat}\ v)} \\
\mathit{apply} & :: \forall v. \mathit{Table}\ v \to (\mathit{Nat} \times \mathit{Nat} \to v) \\
\mathit{apply}\ t\ (i, j) & = \mathit{applyNat}\ (\mathit{applyNat}\ t\ i)\ j \\
\mathit{tabulate} & :: \forall v. (\mathit{Nat} \times \mathit{Nat} \to v) \to \mathit{Table}\ v \\
\mathit{tabulate}\ \psi & = \mathit{tabulateNat}\ (\lambda i \to \mathit{tabulateNat}\ (\lambda j \to \psi\ (i, j))).
\end{array}
$$

Now, let us construct a memo table for the natural numbers.

Since we aim at extending the technique to arbitrary types, we will employ the structure of the type Nat to organize the memo table (in fact, the memo tables are based on so-called *generalized tries*).

However, Peano, Leibniz and Knuth suggest different representations of the natural numbers . . .

Peano numbers

Here is the simplest representation of the natural numbers:

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

Semantics:

$$\begin{aligned} [\![Zero]\!] &= 0 \\ [\![Succ\ n]\!] &= [\![n]\!] + 1. \end{aligned}$$

A memo table for Peano numbers

The memo table for Peano numbers is an infinite list:

$$\begin{aligned}&\mathbf{data}\ \mathit{TableNat}\ v = \mathit{Node}\{\mathit{trieZero} :: v,\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathit{trieSucc} :: \mathit{TableNat}\ v\}.\end{aligned}$$

The look-up function corresponds to list indexing:

$$\begin{aligned}&\mathit{applyNat}\ (\mathit{Node}\{\mathit{trieZero} = t\})\ \mathit{Zero} &&= t\\ &\mathit{applyNat}\ (\mathit{Node}\{\mathit{trieSucc} = t\})\ (\mathit{Succ}\ n) &&= \mathit{applyNat}\ t\ n.\end{aligned}$$

Leibniz numbers

A binary representation of the natural numbers:

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

Semantics:

$$\begin{aligned}
[\![End]\!] &= 0 \\
[\![One\ n]\!] &= 2 * [\![n]\!] + 1 \\
[\![Two\ n]\!] &= 2 * [\![n]\!] + 2.
\end{aligned}$$

Exercise. What is the representation of 10?

A memo table for Leibniz numbers

The memo table for Leibniz numbers is an infinite binary tree:

$$\begin{aligned}\mathbf{data}\ \mathit{TableBin}\ v = \mathit{Node}\{ & \mathit{trieEnd} :: v,\\ & \mathit{trieOne} :: \mathit{TableBin}\ v,\\ & \mathit{trieTwo} :: \mathit{TableBin}\ v\}.\end{aligned}$$

The look-up function corresponds to tree indexing:

$$\begin{aligned}&\mathit{applyBin}\ (\mathit{Node}\{\mathit{trieEnd} = t\})\ \mathit{End} &&= t\\ &\mathit{applyBin}\ (\mathit{Node}\{\mathit{trieOne} = t\})\ (\mathit{One}\ n) &&= \mathit{applyBin}\ t\ n\\ &\mathit{applyBin}\ (\mathit{Node}\{\mathit{trieTwo} = t\})\ (\mathit{Two}\ n) &&= \mathit{applyBin}\ t\ n.\end{aligned}$$

Knuth numbers

A different representation of the natural numbers (humungous numbers):

$$\textbf{data}\ Hum = Leaf \mid Fork\ Hum\ Hum.$$

Semantics:

$$\begin{aligned} [\![Leaf]\!] &= 0 \\ [\![Fork\ l\ r]\!] &= 2 \uparrow [\![l]\!] + [\![r]\!]. \end{aligned}$$

Exercise. What is a representation of 10?

A memo table for Knuth numbers

The memo table for Knuth numbers is an infinite 'bushy list':

$$\begin{aligned} \mathbf{data}\ \mathit{TableHum}\ v = \mathit{Node}\{ & \mathit{trieLeaf} :: v, \\ & \mathit{trieFork} :: \mathit{TableHum}\ (\mathit{TableHum}\ v)\}. \end{aligned}$$

The look-up function corresponds to 'bushy list' indexing:

$$\begin{aligned} & \mathit{applyHum}\ (\mathit{Node}\{\mathit{trieLeaf} = t\})\ \mathit{Leaf} \\ & \qquad = t \\ & \mathit{applyHum}\ (\mathit{Node}\{\mathit{trieFork} = t\})\ (\mathit{Fork}\ l\ r) \\ & \qquad = \mathit{applyHum}\ (\mathit{applyHum}\ t\ l)\ r. \end{aligned}$$

NB. $\mathit{TableHum}$ is a 'nest' and $\mathit{applyHum}$ requires polymorphic recursion.

A polytypic implementation of memo tables

Now, let us apply the framework of polytypic programming to implement trie-based memo tables with associated look-up and tabulation functions.

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

NB. Table is a type-indexed type and apply and $\mathit{tabulate}$ are type-indexed values.

Memo tables

Tries are based on the *laws of exponentials*.

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

From these equations we can immediately derive a definition of $Table$:

$$\begin{array}{lcl} Table\langle 1\rangle\ v & = & v \\ Table\langle k_1 + k_2\rangle\ v & = & Table\langle k_1\rangle\ v \times Table\langle k_2\rangle\ v \\ Table\langle k_1 \times k_2\rangle\ v & = & Table\langle k_1\rangle\ (Table\langle k_2\rangle\ v). \end{array}$$

Table look-up

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

$$
\begin{array}{ll}
\mathit{apply}\langle k\rangle & :: \forall v. \mathit{Table}\langle k\rangle\ v \rightarrow (k \rightarrow v)\\
\mathit{apply}\langle 1\rangle\ t\ () & = t\\
\mathit{apply}\langle k_1 + k_2\rangle\ (t_1, t_2)\ (\mathit{inl}\ i_1) & = \mathit{apply}\langle k_1\rangle\ t_1\ i_1\\
\mathit{apply}\langle k_1 + k_2\rangle\ (t_1, t_2)\ (\mathit{inr}\ i_2) & = \mathit{apply}\langle k_2\rangle\ t_2\ i_2\\
\mathit{apply}\langle k_1 \times k_2\rangle\ t\ (i_1, i_2) & = \mathit{apply}\langle k_2\rangle\ (\mathit{apply}\langle k_1\rangle\ t\ i_1)\ i_2
\end{array}
$$

Deriving $tabulate$

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

$$apply\langle k\rangle\ (tabulate\langle k\rangle\ \varphi) = \varphi$$

For the derivation we use a slight reformulation of $apply$ that allows for more structured calculations.

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

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

$$
\begin{array}{ll}
& \mathit{apply}\langle k_1 + k_2\rangle\ t = \varphi \\
\iff & \quad \{\ \text{definition } \mathit{apply}\ \} \\
& \mathit{apply}\langle k_1\rangle\ (\mathit{outl}\ t) \triangledown \mathit{apply}\langle k_2\rangle\ (\mathit{outr}\ t) = \varphi \\
\iff & \quad \{\ \text{universal property of coproducts}\ \} \\
& \mathit{apply}\langle k_1\rangle\ (\mathit{outl}\ t) = \varphi \cdot \mathit{inl} \wedge \mathit{apply}\langle k_2\rangle\ (\mathit{outr}\ t) = \varphi \cdot \mathit{inr} \\
\Longleftarrow & \quad \{\ \text{specification}\ \} \\
& \mathit{outl}\ t = \mathit{tabulate}\langle k_1\rangle\ (\varphi \cdot \mathit{inl}) \wedge \mathit{outr}\ t = \mathit{tabulate}\langle k_2\rangle\ (\varphi \cdot \mathit{inr}) \\
\iff & \quad \{\ \text{surjective pairing}\ \} \\
& t = (\mathit{tabulate}\langle k_1\rangle\ (\varphi \cdot \mathit{inl}), \mathit{tabulate}\langle k_2\rangle\ (\varphi \cdot \mathit{inr})).
\end{array}
$$

Tabulation

Using similar calculations for $k = 1$ and $k = k_1 \times k_2$ we obtain the following definition of $tabulate$.

$$
\begin{array}{ll}
tabulate\langle k\rangle & :: \forall v.(k \to v) \to Table\langle k\rangle\ v \\
tabulate\langle 1\rangle\ \varphi & = \varphi\ () \\
tabulate\langle k_1 + k_2\rangle\ \varphi & = (tabulate\langle k_1\rangle\ (\varphi \cdot inl), tabulate\langle k_2\rangle\ (\varphi \cdot inr)) \\
tabulate\langle k_1 \times k_2\rangle\ \varphi & = tabulate\langle k_1\rangle\ (tabulate\langle k_2\rangle \cdot curry\ \varphi)
\end{array}
$$

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

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

Examples

$$
\begin{aligned}
&\mathit{tabulateNat} && :: \forall v.(\mathit{Nat} \to v) \to \mathit{TableNat}\ v\\
&\mathit{tabulateNat}\ \varphi && = \mathit{Node}\{\, \mathit{trieZero} = \varphi\ \mathit{Zero},\\
& && \qquad\qquad \mathit{trieSucc} = \mathit{tabulateNat}\ (\varphi \cdot \mathit{Succ})\}\\
&\mathit{tabulateBin} && :: \forall v.(\mathit{Bin} \to v) \to \mathit{TableBin}\ v\\
&\mathit{tabulateBin}\ \varphi && = \mathit{Node}\{\, \mathit{trieEnd} = \varphi\ \mathit{End},\\
& && \qquad\qquad \mathit{trieOne} = \mathit{tabulateBin}\ (\varphi \cdot \mathit{One}),\\
& && \qquad\qquad \mathit{trieTwo} = \mathit{tabulateBin}\ (\varphi \cdot \mathit{Two})\}\\
&\mathit{tabulateHum} && :: \forall v.(\mathit{Hum} \to v) \to \mathit{TableHum}\ v\\
&\mathit{tabulateHum}\ \varphi && = \mathit{Node}\{\, \mathit{trieLeaf} = \varphi\ \mathit{Leaf},\\
& && \qquad\qquad \mathit{trieFork} = \mathit{tabulateHum}\ (\lambda l \to\\
& && \qquad\qquad\qquad\qquad \mathit{tabulateHum}\ (\lambda r \to\\
& && \qquad\qquad\qquad\qquad\qquad \varphi\ (\mathit{Fork}\ l\ r)))\}
\end{aligned}
$$

Generalizing to arbitrary types

Haskell's type system is covered by the following language of types.

$$t, u \quad ::= \quad 1 \mid (t + u) \mid (t \times u) \mid a \mid (t\ u) \mid (\Lambda a.t) \mid (Rec\ a.t)$$

Type terms must be *well-kinded*, where the language of kinds is given by

$$K, L \quad ::= \quad * \mid K \to L.$$

Example: parametric lists

The memo table of parametric lists

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

is given by:

$$\begin{aligned}&\textbf{data}\ \mathit{TableList}\ \mathit{tablea}\ v\\ &= \mathit{Node}\{\mathit{trieNil} :: v,\\ &\qquad\qquad \mathit{trieCons} :: \mathit{tablea}\ (\mathit{TableList}\ \mathit{tablea}\ v)\}.\end{aligned}$$

NB. $\mathit{TableList}$ is a 'higher-order' memo table that takes a trie for the base type a and yields a trie for $\mathit{List}\ a$.

Memo tables

The following equations extend $Table$ to arbitrary type terms.

$$
\begin{array}{ll}
TABLE\langle K :: \Box \rangle & :: \Box \\
TABLE\langle * \rangle & = * \to * \\
TABLE\langle K \to L \rangle & = TABLE\langle K \rangle \to TABLE\langle L \rangle \\
Table\langle k :: K \rangle & :: TABLE\langle K \rangle \\
Table\langle a \rangle & = table_a \\
Table\langle t\ u \rangle & = (Table\langle t \rangle)\ (Table\langle u \rangle) \\
Table\langle \Lambda a.t \rangle & = \Lambda table_a . Table\langle t \rangle \\
Table\langle Rec\ a.t \rangle & = Rec\ table_a . Table\langle t \rangle
\end{array}
$$

NB. $TABLE$ is a kind-indexed kind.

Example: parametric lists

$$
\begin{aligned}
&\mathit{applyList} :: \forall \mathit{tablea}\ a.(\forall v.\mathit{tablea}\ v \to (a \to v))\\
&\qquad\qquad\qquad\qquad \to (\forall w.\mathit{TableList}\ \mathit{tablea}\ w \to (\mathit{List}\ a \to w))\\
&\mathit{applyList}\ \mathit{applya}\ (\mathit{Node}\{\mathit{trieNil} = t\})\ \mathit{Nil}\\
&\qquad = t\\
&\mathit{applyList}\ \mathit{applya}\ (\mathit{Node}\{\mathit{trieCons} = t\})\ (\mathit{Cons}\ a\ \mathit{as})\\
&\qquad = \mathit{applyList}\ \mathit{applya}\ (\mathit{applya}\ t\ a)\ \mathit{as}
\end{aligned}
$$

Since List is a parametric type, $\mathit{applyList}$ is a 'higher-order' look-up function that takes a look-up function for the base type a and yields a look-up function for $\mathit{List}\ a$.

Table look-up

The following equations extend $apply$ to arbitrary type terms.

$$
\begin{array}{ll}
Apply\langle K :: \Box\rangle & :: \ TABLE\langle K\rangle \to K \to * \\
Apply\langle *\rangle\ tu\ u & = \forall v.tu\ v \to (u \to v) \\
Apply\langle K \to L\rangle\ tu\ u & = \forall ta\ a.Apply\langle K\rangle\ ta\ a \to Apply\langle L\rangle\ (tu\ ta)\ (u\ a) \\
apply\langle k :: K\rangle & :: \ Apply\langle K\rangle\ (Table\langle k\rangle)\ k \\
apply\langle a\rangle & = apply_a \\
apply\langle t\ u\rangle & = (apply\langle t\rangle)\ (apply\langle u\rangle) \\
apply\langle \Lambda a.t\rangle & = \lambda apply_a.apply\langle t\rangle \\
apply\langle Rec\ a.t\rangle & = rec\ apply_a.apply\langle t\rangle.
\end{array}
$$

NB. $Apply$ is a kind-indexed type.

Polytypic programming: a broader perspective

Modern functional programming languages such as Haskell 98 typically have a three level structure (ignoring the module system).

- values
- types — imposing structure on the value level
- kinds — imposing structure on the type level

In ‘ordinary’ programming we define

- values depending on values (called functions) and
- types depending on types (called type constructors).

Polytypic programming adds to this list the possibility of defining

- values such as $apply$ that are defined by induction on the structure of types (called polytypic values).
- types such as $Apply$ that are defined by induction on the structure of kinds (called polykinded types).
- types such as $Table$ that are defined by induction on the structure of types (called polytypic types).
- kinds such as $TABLE$ that are defined by induction on the structure of kinds (called polykinded kinds).