TRIES, POLYTYPICALLY!

RALF HINZE

Institut für Informatik III, Universität Bonn, Römerstraße 164
53117 Bonn, Germany
E-mail: ralf@informatik.uni-bonn.de

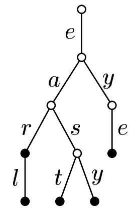
March, 1999

Tries

A trie is a search tree scheme due to A. Thue that employs the structure of search keys to organize information.

Originally, tries were used to represent sets of strings.

$$\{ear, earl, east, easy, eye\} \Longrightarrow$$



Generalized tries

R.H. Connelly and F.L. Morris generalized the concept of a trie to permit indexing by elements of an arbitrary *monomorphic* datatype.

The generalization is based on the *finite map abstraction*—each trie represents a finite map—and makes use of the following isomorphisms.

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

Generalizing generalized tries

In this talk we show how to define tries generically for arbitrary *polymorphic* datatypes.

To this end we make heavy use of the framework of *polytypic programming*.

The central insight is that a trie can be considered as a *type-indexed* datatype.

Polytypic programming

A polytypic or generic function is one that is parameterised by datatype and is defined by induction on the structure of types.

For instance:

$$\mathit{flatten}\langle f\rangle \quad :: \quad \forall a.f\ a \to \mathit{List}\ a$$

Meaning: $\mathit{flatten}\langle f\rangle$ traverses a value of type $f\ a$ and collects all values of type a from left to right in a list.

We must explicit the structure of types . . .

Polytypic programming—Datatypes

Running examples:

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{List}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\
\mathbf{data}\ \mathit{Tree}\ a_1\ a_2 & = & \mathit{Leaf}\ a_1 \mid \mathit{Node}\ (\mathit{Tree}\ a_1\ a_2)\ a_2\ (\mathit{Tree}\ a_1\ a_2) \\
\mathbf{data}\ \mathit{Fork}\ a & = & \mathit{Fork}\ a\ a \\
\mathbf{data}\ \mathit{Perfect}\ a & = & \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Perfect}\ (\mathit{Fork}\ a))
\end{array}$$

The structure of the datatype definitions becomes more apparent if we rewrite the definitions as *functor equations*.

$$\begin{array}{lcl}
\mathit{List} & = & K1 + \mathit{Id} \times \mathit{List} \\
\mathit{Tree} & = & \mathit{Fst} + \mathit{Tree} \times \mathit{Snd} \times \mathit{Tree} \\
\mathit{Fork} & = & \mathit{Id} \times \mathit{Id} \\
\mathit{Perfect} & = & \mathit{Id} + \mathit{Perfect} \cdot \mathit{Fork}
\end{array}$$

Polytypic programming–Modeling non-recursive types

Non-recursive datatypes are modeled simply by type expressions. *Type expression of arity* n are formed according to

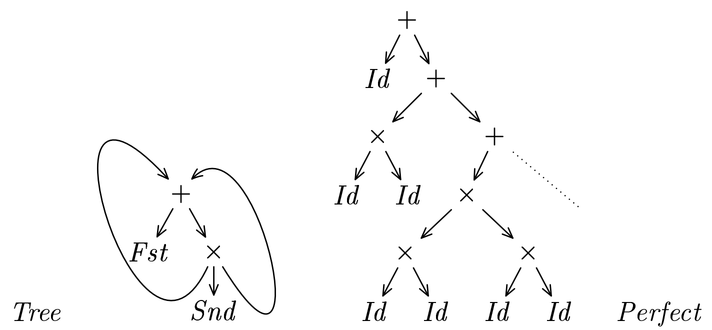
$$T^n \quad ::= \quad \Pi^n_i \mid P^k \cdot (T^n_1, \ldots, T^n_k) \ .$$

Here, P^k is the set of primitive type constructors of arity k, ie $P^0 = \{1, \mathit{Int}\}$ and $P^2 = \{+, \times\}$.

Notation: we write $K t$ for $t \cdot ()$ and $f_1 \oplus f_2$ for $\oplus \cdot (f_1, f_2)$. For unary and binary projection functors we use the following more familiar names: $\mathit{Id} = \Pi^1_1$, $\mathit{Fst} = \Pi^2_1$, and $\mathit{Snd} = \Pi^2_2$.

Polytypic programming–Modeling recursive types

Recursive datatypes are modeled by infinite type expressions!



NB Tree is a *rational tree*, $\mathit{Perfect}$ is an *algebraic tree*.

Polytypic programming—Polytypic $\mathit{flatten}\langle f\rangle$

Here is the generic definition of $\mathit{flatten}\langle f\rangle$.

$$
\begin{array}{lcl}
\mathit{flatten}\langle f\rangle & :: & \forall a.f\ a \to \mathit{List}\ a \\
\mathit{flatten}\langle \mathit{Id}\rangle & = & \mathit{wrap} \\
\mathit{flatten}\langle K t\rangle & = & \mathit{const}\ [\,] \\
\mathit{flatten}\langle f_1 + f_2\rangle & = & \mathit{flatten}\langle f_1\rangle \triangledown \mathit{flatten}\langle f_2\rangle \\
\mathit{flatten}\langle f_1 \times f_2\rangle & = & \mathit{flatten}\langle f_1\rangle \bowtie \mathit{flatten}\langle f_2\rangle \\
\mathit{wrap}\ a & = & [a] \\
(\varphi_1 \bowtie \varphi_2)\ (a_1, a_2) & = & \varphi_1\ a_1 +\!\!+\ \varphi_2\ a_2
\end{array}
$$

NB $\mathit{flatten}\langle f\rangle$ is inductive on the structure of type expressions.

Polytypic programming—Polytypic definitions

In general, a polytypic function which is parameterised by an m-ary type constructor is given by the following data.

$$
\begin{array}{rcl}
\mathit{poly}\langle f\rangle & :: & \tau(f) \\
\mathit{poly}\langle \Pi_i\rangle & = & \mathit{poly}_{\Pi_i} \\
\mathit{poly}\langle F \cdot (f_1, \ldots, f_k)\rangle & = & \mathit{poly}_F\ (\mathit{poly}\langle f_1\rangle, \ldots, \mathit{poly}\langle f_k\rangle)
\end{array}
$$

We have one equation for each projection functor Π_i ($1 \leqslant i \leqslant m$) and one equation for each primitive type constructor F ($F \in \{1, \mathit{Int}, +, \times\}$).

Polytypic programming—Specialization

The specialization *cannot* be based on the structure of type expressions, because a type is in general given by an algebraic tree, which may have an infinite number of different subtrees.

Instead, the specialization must be based on the *representation* of type expressions, ie on functor equations.

Functor expressions of arity n are given by

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

Polytypic programming—Specialization

Assume that we want to specialize $\mathit{flatten}\langle F\rangle$ for a given F.

Problem: F may be defined in terms of k-ary functors, for instance, $F = G \cdot (e_1, e_2)$. How can we generate code for the right-hand side?

Solution: We define, for each arity n, an auxiliary function $\mathit{poly}_n\langle g\rangle$ satisfying

$$
\mathit{poly}_n\langle g\rangle\ (\mathit{poly}\langle g_1\rangle, \ldots, \mathit{poly}\langle g_n\rangle) \quad = \quad \mathit{poly}\langle g \cdot (g_1, \ldots, g_n)\rangle \ .
$$

Note the close correspondence between functors and functions.

Polytypic programming—Specialization

The following definition of $poly_n\langle g\rangle$ satisfies the specification.

$$\begin{array}{rcl}
poly\langle f\rangle & :: & \tau(f) \\
poly\langle f\rangle & = & poly_m\langle f\rangle\ (poly_{\Pi_1}, \ldots, poly_{\Pi_m}) \\
poly_n\langle g\rangle & :: & (\tau(g_1), \ldots, \tau(g_n)) \to \tau(g \cdot (g_1, \ldots, g_n)) \\
poly_n\langle \Pi_i\rangle & = & \pi_i \\
poly_n\langle F\rangle & = & poly_F \\
poly_n\langle h \cdot (h_1, \ldots, h_k)\rangle & = & poly_k\langle h\rangle \star (poly_n\langle h_1\rangle, \ldots, poly_n\langle h_k\rangle)
\end{array}$$

where $\pi_i\,(\varphi_1, \ldots, \varphi_n) = \varphi_i$ and $(\varphi \star (\varphi_1, \ldots, \varphi_n))\,a = \varphi\,(\varphi_1\ a, \ldots, \varphi_n\ a)$.

NB $poly_n$ is inductive on the structure of functor expressions.

Polytypic programming—Specializing $flatten\langle f\rangle$

Specializing a polytypic function works by transforming a system of functor equations into a system of function definitions.

$$\begin{array}{rcl}
Fork & = & Id \times Id \\
Perfect & = & Id + Perfect \cdot Fork \\
Tree & = & Fst + Tree \times Snd \times Tree
\end{array}$$

$$\Downarrow$$

$$\begin{array}{rcl}
flatten\langle Perfect\rangle & = & flatten_1\langle Perfect\rangle\ wrap \\
flatten_1\langle Fork\rangle\ \varphi & = & \varphi \bowtie \varphi \\
flatten_1\langle Perfect\rangle\ \varphi & = & \varphi \bigtriangledown flatten_1\langle Perfect\rangle\ (flatten_1\langle Fork\rangle\ \varphi) \\
flatten_2\langle Tree\rangle\ (\varphi_1, \varphi_2) & = & \varphi_1 \bigtriangledown flatten_2\langle Tree\rangle\ (\varphi_1, \varphi_2) \bowtie \varphi_2 \bowtie \\
 & & \quad flatten_2\langle Tree\rangle\ (\varphi_1, \varphi_2)
\end{array}$$

Tries—Signature

We will define

$$\begin{array}{lcl}
Map\langle *\rangle & :: & * \to * \\
empty\langle k\rangle & :: & \forall v.Map\langle k\rangle\ v \\
lookup\langle k\rangle & :: & \forall v.k \to Map\langle k\rangle\ v \to Maybe\ v \\
merge\langle k\rangle & :: & \forall v.(v \to v \to v) \to \\
 & & \quad (Map\langle k\rangle\ v \to Map\langle k\rangle\ v \to Map\langle k\rangle\ v)\ .
\end{array}$$

$Map\langle k\rangle\ v$ represents the set of finite maps from k to v, ie $k \to_{\mathrm{fin}} v$.

Tries—Polytypic $Map\langle k\rangle$

Recall the laws of exponentials.

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

Since $Map\langle k\rangle\ v$ represents $k \to_{\mathrm{fin}} v$, we define

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

Tries—Specializing $Map\langle k\rangle$

Amazingly, the technique of polytypic programming works equally well for polytypic datatypes!

In order to specialize $Map\langle k\rangle$ we define, for each arity n, an n-ary *higher-order functor* Map_n. For $n = 1$ we have, for instance,

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

The type constructor $Map_1\langle g\rangle$ is *the generalized trie of the polymorphic datatype* g.

Tries—Specializing $Map\langle k\rangle$

$$\begin{aligned}
List &= K1 + Id \times List \\
Tree &= Fst + Tree \times Snd \times Tree \\
Fork &= Id \times Id \\
Perfect &= Id + Perfect \cdot Fork
\end{aligned}$$

$$\Downarrow$$

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

Tries—Specializing $Map\langle k\rangle$

The generalized tries can be written as Haskell datatype definitions.

$$\begin{aligned}
\mathbf{data}\ MapL\ m\ v &= TrieL\ (Maybe\ v)\ (m\ (MapL\ m\ v)) \\
\mathbf{data}\ MapT\ m_1\ m_2\ v &= TrieT\ (m_1\ v) \\
&\qquad\quad (MapT\ m_1\ m_2 \\
&\qquad\qquad\quad (m_2\ (MapT\ m_1\ m_2\ v))) \\
\mathbf{data}\ MapF\ m\ v &= TrieF\ (m\ (m\ v)) \\
\mathbf{data}\ MapP\ m\ v &= TrieP\ (m\ v) \\
&\qquad\quad (MapP\ (MapF\ m)\ v)
\end{aligned}$$

NB $MapT$ is a first-order nest, $MapP$ is a higher-order nest.

Tries—Empty tries

The empty trie is defined as follows.

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

NB $empty\langle k\rangle$ makes use of *polymorphic recursion*.

Tries—Specializing $empty\langle k\rangle$

We obtain

$$
\begin{array}{lcl}
emptyL & :: & (\forall v.m\ v) \to (\forall v.MapL\ m\ v) \\
emptyL\ e & = & TrieL\ Nothing\ e \\
emptyT & :: & (\forall v.m_1\ v) \to (\forall v.m_2\ v) \to (\forall v.MapT\ m_1\ m_2\ v) \\
emptyT\ e_1\ e_2 & = & TrieT\ e_1\ (emptyT\ e_1\ e_2) \\
emptyF & :: & (\forall v.m\ v) \to (\forall v.MapF\ m\ v) \\
emptyF\ e & = & TrieF\ e \\
emptyP & :: & (\forall v.m\ v) \to (\forall v.MapP\ m\ v) \\
emptyP\ e & = & TrieP\ e\ (emptyP\ (emptyF\ e))\ .
\end{array}
$$

NB We require rank-2 type signatures.

Tries—Lookup

The look-up function is defined as follows.

$$
\begin{array}{lcl}
lookup\langle k\rangle & :: & \forall v.k \to Map\langle k\rangle\ v \to Maybe\ v \\
lookup\langle 1\rangle\ ()\ t & = & t \\
lookup\langle Int\rangle\ i\ t & = & Patricia.lookup\ i\ t \\
lookup\langle k_1 + k_2\rangle\ (inl\ i_1)\ (t_1, t_2) & = & lookup\langle k_1\rangle\ i_1\ t_1 \\
lookup\langle k_1 + k_2\rangle\ (inr\ i_2)\ (t_1, t_2) & = & lookup\langle k_2\rangle\ i_2\ t_2 \\
lookup\langle k_1 \times k_2\rangle\ (i_1, i_2)\ t_1 & = & (lookup\langle k_1\rangle\ i_1 \diamond lookup\langle k_2\rangle\ i_2)\ t_1 \\
(\varphi_1 \diamond \varphi_2)\ a_1 & = & apply\ \varphi_2\ (\varphi_1\ a_1) \\
apply\ \varphi\ Nothing & = & Nothing \\
apply\ \varphi\ (Just\ a) & = & \varphi\ a
\end{array}
$$

Tries—Specializing $lookup\langle k\rangle$

We obtain

$$
\begin{array}{lcl}
lookupL\ \ell\ Nil\ (TrieL\ tn\ tc) & = & tn \\
lookupL\ \ell\ (Cons\ i\ is)\ (TrieL\ tn\ tc) & = & (\ell\ i \diamond lookupL\ \ell\ is)\ tc \\
lookupT\ \ell_1\ \ell_2\ (Leaf\ i_1)\ (TrieT\ tl\ tn) & = & \ell_1\ i_1\ tl \\
\multicolumn{3}{l}{lookupT\ \ell_1\ \ell_2\ (Node\ t_1\ i_2\ t_2)\ (TrieT\ tl\ tn)} \\
\multicolumn{3}{l}{\quad = \ (lookupT\ \ell_1\ \ell_2\ t_1 \diamond \ell_2\ i_2 \diamond lookupT\ \ell_1\ \ell_2\ t_2)\ tn} \\
lookupF\ \ell\ (Fork\ i_1\ i_2)\ (TrieF\ tf) & = & (\ell\ i_1 \diamond \ell\ i_2)\ tf \\
lookupP\ \ell\ (Zero\ i)\ (TrieP\ ts\ tc) & = & \ell\ i\ ts \\
lookupP\ \ell\ (Succ\ i)\ (TrieP\ ts\ tc) & = & lookupP\ (lookupF\ \ell)\ i\ tc\ .
\end{array}
$$

NB To improve readability the type signatures are omitted. They are, nonetheless, mandatory.

Tries—Merging

Merging two tries is defined as follows.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{merge\langle k\rangle :: \forall v.(v \to v \to v) \to} \\
\multicolumn{3}{l}{\qquad\qquad (Map\langle k\rangle\ v \to Map\langle k\rangle\ v \to Map\langle k\rangle\ v)} \\
merge\langle 1\rangle\ c\ t\ t' & = & combine\ c\ t\ t' \\
merge\langle Int\rangle\ c\ t\ t' & = & Patricia.merge\ c\ t\ t' \\
merge\langle k_1 + k_2\rangle\ c\ (t_1, t_2)\ (t_1', t_2') & = & (merge\langle k_1\rangle\ c\ t_1\ t_1', \\
 & & \ merge\langle k_2\rangle\ c\ t_2\ t_2') \\
merge\langle k_1 \times k_2\rangle\ c\ t\ t' & = & merge\langle k_1\rangle\ (merge\langle k_2\rangle\ c)\ t\ t' \\
combine\ c\ Nothing\ Nothing & = & Nothing \\
combine\ c\ Nothing\ (Just\ a') & = & Just\ a' \\
combine\ c\ (Just\ a)\ Nothing & = & Just\ a \\
combine\ c\ (Just\ a)\ (Just\ a') & = & Just\ (c\ a\ a')
\end{array}
$$

Tries—Specializing $merge\langle k\rangle$

We obtain

$$
\begin{aligned}
& \mathit{mergeL}\ m\ c\ (\mathit{TrieL}\ tn_1\ tc_1)\ (\mathit{TrieL}\ tn_2\ tc_2) \\
& \quad = \quad \mathit{TrieL}\ (\mathit{combine}\ c\ tn_1\ tn_2)\ (m\ (\mathit{mergeL}\ m\ c)\ tc_1\ tc_2) \\
& \mathit{mergeT}\ m_1\ m_2\ c\ (\mathit{TrieT}\ tl_1\ tn_1)\ (\mathit{TrieT}\ tl_2\ tn_2) \\
& \quad = \quad \mathit{TrieT}\ (m_1\ c\ tl_1\ tl_2) \\
& \qquad\qquad\qquad (\mathit{mergeT}\ m_1\ m_2\ (m_2\ (\mathit{mergeT}\ m_1\ m_2\ c))\ tn_1\ tn_2) \\
& \mathit{mergeF}\ m\ c\ (\mathit{TrieF}\ tf_1)\ (\mathit{TrieF}\ tf_2) \\
& \quad = \quad \mathit{TrieF}\ (m\ (m\ c)\ tf_1\ tf_2) \\
& \mathit{mergeP}\ m\ c\ (\mathit{TrieP}\ ts_1\ tc_1)\ (\mathit{TrieP}\ ts_2\ tc_2) \\
& \quad = \quad \mathit{TrieP}\ (m\ c\ ts_1\ ts_2)\ (\mathit{mergeP}\ (\mathit{mergeF}\ m)\ c\ tc_1\ tc_2)\ .
\end{aligned}
$$

Résumé

- A polytypic function is defined by induction on the structure of type expressions.
- Types are modeled by algebraic trees.
- Specializing a polytypic function works by transforming a system of functor equations into a system of function definitions.
- A trie can be considered as a type-indexed datatype. The framework of polytypic programming works equally well for polytypic datatypes.
- Polytypic programming is simpler than monotypic programming!

Future work

It remains to generalize tries to arbitrary *higher-order* polymorphic datatypes.

$$
\mathbf{data}\ \mathit{GRose}\ f\ a \quad = \quad \mathit{Node}\ a\ (f\ (\mathit{GRose}\ f\ a))
$$

Basic idea: higher-order polymorphic datatypes are modeled by possibly infinite expressions of the simply typed λ-calculus (with kinds playing the rôle of types).

Appendix—Unit, sums, and products

We treat 1, '$+$', and '$\times$' as if they were given by

$$
\begin{aligned}
\mathbf{data}\ 1 \quad & = \quad () \\
\mathbf{data}\ a_1 + a_2 \quad & = \quad \mathit{inl}\ a_1 \mid \mathit{inr}\ a_2 \\
\mathbf{data}\ a_1 \times a_2 \quad & = \quad (a_1, a_2)\ .
\end{aligned}
$$

Appendix—Combining forms

The following combining forms capture common programming idioms.

$$
\begin{array}{lcl}
id\ a & = & a \\
const\ a\ b & = & a \\
(f_1 \triangledown f_2)\ (inl\ a_1) & = & f_1\ a_1 \\
(f_1 \triangledown f_2)\ (inr\ a_2) & = & f_2\ a_2 \\
f_1 + f_2 & = & (inl \circ f_1) \triangledown (inr \circ f_2) \\
outl\ (a_1, a_2) & = & a_1 \\
outr\ (a_1, a_2) & = & a_2 \\
(f_1 \vartriangle f_2)\ a & = & (f_1\ a, f_2\ a) \\
f_1 \times f_2 & = & (f_1 \circ outl) \vartriangle (f_2 \circ outr)
\end{array}
$$