POLYTYPIC VALUES POSSESS POLYKINDED TYPES

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#T12.)

The name of the game

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 depending on types (called polytypic values).

This talk shows that it is also useful to have

- types depending on kinds (called polykinded types).

The first level: values

variables	a, b			
terms	t, u	$::=$	a	variable
		$\mid$	$t\ u$	application
		$\mid$	$\lambda a\,.\,t$	abstraction
		$\mid$	$rec\ a\,.\,t$	recursion
		$\mid$	$()$	empty tuple
		$\mid$	$inl\ t$	left injection
		$\mid$	$inr\ t$	right injection
		$\mid$	$\mathbf{case}\ t\ \mathbf{of}\ \{inl\ a_1 \to t_1; inr\ a_2 \to t_2\}$	
		$\mid$	(t, u)	pairing $\quad \lfloor$case analysis
		$\mid$	$outl\ t$	left projection
		$\mid$	$outr\ t$	right projection

The second level: types

type variables	A, B			
type terms	T, U	$::=$	A	type variable
		$\mid$	$T\ U$	type application
		$\mid$	$\Lambda A\,.\,T$	type abstraction
		$\mid$	$Rec\,A\,.\,T$	type recursion
		$\mid$	1	unit type
		$\mid$	$T + U$	sum type
		$\mid$	$T \times U$	product type
		$\mid$	$T \to U$	function type

The third level: kinds

kind terms	K, L	$::=$	$\star$	kind of types
		$\mid$	$K \to L$	function kind

Example types

The types of natural numbers, parametric lists, rose trees and generalized rose trees can be defined as follows:

$$
\begin{array}{ll}
Nat & :: \star \\
Nat & = Rec\, N\, .\, 1 + N \\
List & :: \star \to \star \\
List & = \Lambda A\, .\, Rec\, X\, .\, 1 + A \times X \\
Rose & :: \star \to \star \\
Rose & = \Lambda A\, .\, Rec\, T\, .\, A \times List\ T \\
GRose & :: (\star \to \star) \to (\star \to \star) \\
GRose & = \Lambda F\, .\, \Lambda A\, .\, Rec\, T\, .\, A \times F\ T.
\end{array}
$$

NB. All of these types can be defined in Haskell, as well.

$$
\begin{array}{ll}
\textbf{data}\ Nat & = Zero \mid Succ\ Nat \\
\textbf{data}\ List\ a & = Nil \mid Cons\ a\ (List\ a) \\
\textbf{data}\ Rose\ a & = Node\ a\ (List\ (Rose\ a)) \\
\textbf{data}\ GRose\ f\ a & = GNode\ a\ (f\ (GRose\ f\ a))
\end{array}
$$

NB. In Haskell we cannot distinguish between $List = \Lambda A\,.\,Rec\,X\,.\,1+A\times X$ and $List = Rec\,F\,.\,\Lambda A\,.\,1 + A \times F\ A$.

Recap: defining polytypic values

To define a polytypic value it suffices to specify cases for the primitive type constructors, that is, 1, '$+$', '$\times$' and '$\rightarrow$'.

Here is the polytypic equality function:

$$
\begin{array}{ll}
equal\langle T\rangle & :: T \to T \to Bool \\
equal\langle 1\rangle\ ()\ () & = True \\
equal\langle A+B\rangle\ (inl\ a_1)\ (inl\ a_2) & = equal\langle A\rangle\ a_1\ a_2 \\
equal\langle A+B\rangle\ (inl\ a_1)\ (inr\ b_2) & = False \\
equal\langle A+B\rangle\ (inr\ b_1)\ (inl\ a_2) & = False \\
equal\langle A+B\rangle\ (inr\ b_1)\ (inr\ b_2) & = equal\langle B\rangle\ b_1\ b_2 \\
equal\langle A\times B\rangle\ (a_1,b_1)\ (a_2,b_2) & = equal\langle A\rangle\ a_1\ a_2 \wedge equal\langle B\rangle\ b_1\ b_2.
\end{array}
$$

NB. Unfortunately, equality cannot be defined for '$\rightarrow$'.

Recap: specializing polytypic values

How can we specialize $equal\langle T \rangle$ for given instances of T?

$equal\langle List\ Bool \rangle$	$equal\langle List\ (List\ Nat) \rangle$

The specialization should be compositional, that is, the code generated for $List$ should be shared.

☞ *Basic idea:* Let function follow type.

$equal_{List}\ equal_{Bool}$	$equal_{List}\ (equal_{List}\ equal_{Nat})$

Since $List$ is a function on types, $equal_{List}$ is a function on 'equality functions': it maps $equal\langle A \rangle$ to $equal\langle List\ A \rangle$.

Recap: specializing polytypic values (continued)

The following equations extend the definition of $equal$:

$$
\begin{array}{ll}
equal\langle T :: K\rangle & :: Equal\langle K\rangle\ T \\
equal\langle A\rangle & = equal_A \\
equal\langle T\ U\rangle & = (equal\langle T\rangle)\ (equal\langle U\rangle) \\
equal\langle \Lambda A\,.\,T\rangle & = \lambda equal_A\,.\,equal\langle T\rangle \\
equal\langle Rec\ A\,.\,T\rangle & = rec\ equal_A\,.\,equal\langle T\rangle,
\end{array}
$$

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

$$
\begin{array}{ll}
Equal\langle\star\rangle\ T & = T \to T \to Bool \\
Equal\langle K \to L\rangle\ T & = \forall A\,.\,Equal\langle K\rangle\ A \to Equal\langle L\rangle\ (T\ A).
\end{array}
$$

Recap: specializing polytypic values (continued)

Now we can specialize $equal$ for $Nat = Rec\, N\, .\, 1 + N$:

$$
\begin{aligned}
&equal_{Nat} :: Nat \to Nat \to Bool \\
&equal_{Nat} = rec\ equal_N\ .\ equal_+\ equal_1\ equal_N.
\end{aligned}
$$

where $equal_1 = equal\langle 1 \rangle$ and $equal_+ = equal\langle \Lambda A\, .\, \Lambda B\, .\, A + B \rangle$. Removing the abstract clothing we obtain the following Haskell function.

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

Recap: specializing polytypic values (continued)

Or we can specialize $equal$ for $List = \Lambda A \,.\, Rec\, X \,.\, 1 + A \times X$.

$$
\begin{array}{l}
equal_{List} :: \forall A \,.\, (A \to A \to Bool) \to (List\ A \to List\ A \to Bool) \\
equal_{List} = \lambda equal_A \,.\, rec\ equal_X \,.\, equal_+\ equal_1\ (equal_\times\ equal_A\ equal_X)
\end{array}
$$

Removing the abstract clothing we obtain:

$$
\begin{array}{l}
equalList\ equala\ Nil\ Nil = True \\
equalList\ equala\ Nil\ (Cons\ a_2\ x_2) = False \\
equalList\ equala\ (Cons\ a_1\ x_1)\ Nil = False \\
equalList\ equala\ (Cons\ a_1\ x_1)\ (Cons\ a_2\ x_2) \\
\qquad = equala\ a_1\ a_2 \wedge equalList\ equala\ x_1\ x_2.
\end{array}
$$

An uninteresting example: polytypic identity

The most uninteresting polytypic function is probably the identity function.

$$
\begin{array}{ll}
copy\langle T\rangle & :: T \to T \\
copy\langle 1\rangle\ () & = () \\
copy\langle A + B\rangle\ (inl\ a) & = inl\ (copy\langle A\rangle\ a) \\
copy\langle A + B\rangle\ (inr\ b) & = inr\ (copy\langle B\rangle\ b) \\
copy\langle A \times B\rangle\ (a, b) & = (copy\langle A\rangle\ a, copy\langle B\rangle\ b)
\end{array}
$$

Specializing polytypic identity

Nonetheless, let us specialize $copy$ for $List = \Lambda A \,.\, Rec\, X \,.\, 1 + A \times X$.

$$
\begin{array}{l}
copy_{List} :: \forall A \,.\, (A \to A) \to (List\ A \to List\ A) \\
copy_{List} = \lambda copy_A \,.\, rec\ copy_X \,.\, copy_+\ copy_1\ (copy_\times\ copy_A\ copy_X)
\end{array}
$$

Removing the abstract clothing we obtain:

$$
\begin{array}{lll}
copyList & :: \forall a \,.\, (a \to a) \to (List\ a \to List\ a) \\
copyList\ copya\ Nil & = Nil \\
copyList\ copya\ (Cons\ a\ x) & = Cons\ (copya\ a)\ (copyList\ copya\ x).
\end{array}
$$

H'm, the code looks familiar.

Specializing polytypic identity (continued)

Specializing $copy$ for $Rose = \Lambda A \,.\, Rec\ T \,.\, A \times List\ T$ yields:

$$
\begin{array}{l}
copy_{Rose} :: \forall A \,.\, (A \to A) \to (Rose\ A \to Rose\ A) \\
copy_{Rose} = \lambda copy_A \,.\, rec\ copy_T \,.\, copy_{\times}\ copy_A\ (copy_{List}\ copy_T).
\end{array}
$$

Removing the abstract clothing we obtain again a familiar piece of code:

$$
\begin{array}{l}
copyRose :: \forall a \,.\, (a \to a) \to (Rose\ a \to Rose\ a) \\
copyRose\ copya\ (Node\ a\ ts) \\
\qquad\qquad = Node\ (copya\ a)\ (copyList\ (copyRose\ copya)\ ts).
\end{array}
$$

This is the code of $Rose$'s mapping function — except for the type!

From $copy$ to map

To obtain a polytypic mapping function we must merely generalize $copy$'s type from

$$\begin{aligned} Copy\langle\star\rangle\ T &= T \to T \\ Copy\langle K \to L\rangle\ T &= \forall A\,.\,Copy\langle K\rangle\ A \\ &\qquad \to Copy\langle L\rangle\ (T\ A) \end{aligned}$$

to

$$\begin{aligned} Map\langle\star\rangle\ T_1\ T_2 &= T_1 \to T_2 \\ Map\langle K \to L\rangle\ T_1\ T_2 &= \forall A_1\ A_2\,.\,Map\langle K\rangle\ A_1\ A_2 \\ &\qquad \to Map\langle L\rangle\ (T_1\ A_1)\ (T_2\ A_2)! \end{aligned}$$

A mapping function for all types of all kinds!

$$
\begin{aligned}
&Map\langle\star\rangle\ T_1\ T_2 &&= T_1 \to T_2\\
&Map\langle K \to L\rangle\ T_1\ T_2 &&= \forall A_1\ A_2\,.\,Map\langle K\rangle\ A_1\ A_2\\
&&&\quad \to Map\langle L\rangle\ (T_1\ A_1)\ (T_2\ A_2)
\end{aligned}
$$

$$
\begin{aligned}
&map\langle T :: K\rangle &&:: Map\langle K\rangle\ T\ T\\
&map\langle 1\rangle\ () &&= ()\\
&map\langle +\rangle\ mapA\ mapB\ (inl\ a) &&= inl\ (mapA\ a)\\
&map\langle +\rangle\ mapA\ mapB\ (inr\ b) &&= inr\ (mapB\ b)\\
&map\langle \times\rangle\ mapA\ mapB\ (a, b) &&= (mapA\ a, mapB\ b)
\end{aligned}
$$

Instances of Map

$$
\begin{aligned}
&Map\langle\star\rangle\ Nat\ Nat \\
&= Nat \to Nat \\
&Map\langle\star \to \star\rangle\ List\ List \\
&= \forall A_1\ A_2\,.\,(A_1 \to A_2) \to (List\ A_1 \to List\ A_2) \\
&Map\langle(\star \to \star) \to (\star \to \star)\rangle\ GRose\ GRose \\
&= \forall F_1\ F_2\,.\,(\forall B_1\ B_2\,.\,(B_1 \to B_2) \\
&\qquad\qquad \to (F_1\ B_1 \to F_2\ B_2)) \\
&\qquad \to (\forall A_1\ A_2\,.\,(A_1 \to A_2) \\
&\qquad\qquad \to (GRose\ F_1\ A_1 \to GRose\ F_2\ A_2))
\end{aligned}
$$

Specializing the polytypic mapping function

The mapping function for generalized rose trees is given by:

$$
\begin{aligned}
&mapGRose :: \forall f_1\ f_2\,.\,(\forall b_1\ b_2\,.\,(b_1 \to b_2) \to (f_1\ b_1 \to f_2\ b_2)) \\
&\qquad\qquad \to (\forall a_1\ a_2\,.\,(a_1 \to a_2) \\
&\qquad\qquad\qquad\qquad \to (GRose\ f_1\ a_1 \to GRose\ f_2\ a_2)) \\
&mapGRose\ mapf\ mapa\ (GNode\ a\ ts) \\
&\qquad = GNode\ (mapa\ a)\ (mapf\ (mapGRose\ mapf\ mapa)\ ts).
\end{aligned}
$$

The call $mapGRose\ mapList$ yields the mapping function for ordinary rose trees of type $Rose = GRose\ List = \Lambda A\,.\,Rec\ T\,.\,A \times List\ T$.

Polytypic equality, revisited

The type of $equal$

$$
\begin{aligned}
&Equal\langle\star\rangle\ T &&= T \to T \to Bool \\
&Equal\langle K \to L\rangle\ T &&= \forall A\,.\,Equal\langle K\rangle\ A \\
& && \quad \to Equal\langle L\rangle\ (T\ A)
\end{aligned}
$$

can be generalized to

$$
\begin{aligned}
&Equal\langle\star\rangle\ T_1\ T_2 &&= T_1 \to T_2 \to Bool \\
&Equal\langle K \to L\rangle\ T_1\ T_2 &&= \forall A_1\ A_2\,.\,Equal\langle K\rangle\ A_1\ A_2 \\
& && \quad \to Equal\langle L\rangle\ (T_1\ A_1)\ (T_2\ A_2)!
\end{aligned}
$$

NB. The second equation always has this characteristic form!

Specializing polytypic equality

Since $equal$ has a polykinded type we can also specialize it for, say, unary type constructors.

$$equal\langle F :: \star \to \star \rangle :: \forall A_1\ A_2\,.\,(A_1 \to A_2 \to Bool) \to (F\ A_1 \to F\ A_2 \to Bool)$$

This gives us an extra degree of flexibility: $equal\langle F \rangle\ op\ x_1\ x_2$ checks whether corresponding elements in x_1 and x_2 are related by op.

NB. op need not be an equality operator and the element types need not be identical.

Another uninteresting example: a polytypic size function

$$
\begin{aligned}
&Count\langle\star\rangle\ T &&= T \to Nat \\
&Count\langle K \to L\rangle\ T &&= \forall A\,.\,Count\langle K\rangle\ A \to Count\langle L\rangle\ (T\ A)
\end{aligned}
$$

$$
\begin{aligned}
&count\langle T :: K\rangle && :: Count\langle K\rangle\ T \\
&count\langle 1\rangle\ () && = 0 \\
&count\langle +\rangle\ countA\ countB\ (inl\ a) && = countA\ a \\
&count\langle +\rangle\ countA\ countB\ (inr\ b) && = countB\ b \\
&count\langle \times\rangle\ countA\ countB\ (a, b) && = countA\ a + countB\ b
\end{aligned}
$$

Specializing the polytypic size function

For unary type constructors $count$ has type:

$$count\langle F :: \star \to \star \rangle :: \forall A \,.\, (A \to Nat) \to (F\ A \to Nat).$$

Two applications of $count\langle F \rangle$:

$$\begin{array}{lll} sum\langle F :: \star \to \star \rangle & :: & F\ Nat \to Nat \\ sum\langle F \rangle & = & count\langle F \rangle\ id \\ size\langle F :: \star \to \star \rangle & :: & \forall A \,.\, F\ A \to Nat \\ size\langle F \rangle & = & count\langle F \rangle\ (const\ 1). \end{array}$$

Summary

- Previous approaches to polytypic programming (PolyP: Jansson, Jeuring, POPL 97) were restricted in that they only allowed to parameterize values by types of one fixed kind.

- However, specializing a polytypic value must work and, indeed, does work for arbitrary types of arbitrary kinds (Hinze, POPL 00).

- Giving the programmer the ability to assign polykinded types to polytypic values greatly enhances genericity: a single definition yields, for instance, mapping functions for arbitrary types.

- Furthermore, using a variant of logical relations we can prove properties of these polytypic values (we can show, for instance, that the mapping function preserves identity and composition).

Appendix: polytypic proofs

The mapping function of a unary type constructor must satisfy:

$$\begin{aligned} map\langle F\rangle\ id &= id \\ map\langle F\rangle\ (\varphi \cdot \psi) &= map\langle F\rangle\ \varphi \cdot map\langle F\rangle\ \psi. \end{aligned}$$

If the type constructor is binary, the functor laws take the form:

$$\begin{aligned} map\langle G\rangle\ id\ id &= id \\ map\langle G\rangle\ (\varphi_1 \cdot \psi_1)\ (\varphi_2 \cdot \psi_2) &= map\langle G\rangle\ \varphi_1\ \varphi_2 \cdot map\langle G\rangle\ \psi_1\ \psi_2. \end{aligned}$$

For types we have:

$$\begin{aligned} map\langle T\rangle &= id \\ map\langle T\rangle &= map\langle T\rangle \cdot map\langle T\rangle. \end{aligned}$$

Polytypic proofs (continued)

The following 'logical relation' generalizes the first functor law to types of arbitrary kinds.

$$
\begin{array}{lll}
\mathcal{I}d\langle K\rangle\ T & \subseteq & Map\langle K\rangle\ T\ T \\
t \in \mathcal{I}d\langle\star\rangle\ T & \Longleftrightarrow & t = id :: T \to T \\
t \in \mathcal{I}d\langle K \to L\rangle\ T & \Longleftrightarrow & \forall A :: K\,.\,\forall a :: Map\langle K\rangle\ A\ A\,. \\
 & & \quad a \in \mathcal{I}d\langle K\rangle\ A \Longrightarrow t\ a \in \mathcal{I}d\langle L\rangle\ (T\ A)
\end{array}
$$

Now, if $map\langle C\rangle \in \mathcal{I}d\langle K\rangle\ C$ for all primitive type constructors $C :: K$, then

$$map\langle T\rangle \in \mathcal{I}d\langle K\rangle\ T$$

for all closed type terms $T :: K$.