MAKING GENERIC PROGRAMMING MORE GENERIC

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`

April, 2000

(Pick the slides at `.../~ralf/talks.html#T11`.)

A brief outline of the talk

Generic programming (GFP for short) is about making programming easier and more economical. This talk shows how to make programming even more economical by increasing genericity.

- ☐ Why? (2 – 5)
- ☐ How? (7 – 17)
- ☐ Making generic programming more generic (19 – 28).

Polymorphic type systems

I hope we all agree that *type systems*, especially, *polymorphic type systems* are a good thing. For reasons of

Security soundness results guarantee that well-typed programs cannot 'go wrong'.

Flexibility polymorphism allows the definition of functions that behave uniformly over all types (such as $length$).

NB. This is an uninteresting talk for addicts of untyped languages.

However, . . .

. . . even polymorphic type systems are sometimes less flexible than one would wish.

For instance, it is not possible to define a polymorphic *equality function*.

$$equal \quad :: \quad \forall a \, . \, a \to a \to Bool$$

Parametricity implies that a function of this type must necessarily be constant (roughly speaking, the arguments cannot be inspected).

As a consequence, the programmer is forced to program a separate equality function for each type from scratch.

GFP is your friend

Generic functional programming (aka structural polymorphism, type parametric programming, shape polymorphism, polytypic programming) addresses this problem.

☞ *Basic idea:* define the equality function by *induction on the structure of types* ('one size fits all').

NB. Instances of $equal$ are typically defined by induction on the structure of values.

Generic programs are ubiquitous

- ☐ Equality and comparison functions.
- ☐ Mapping and zipping functions.
- ☐ Reductions (such as $size$, sum, $listify$).
- ☐ Pretty printers (such as Haskell's $show$ function).
- ☐ Parsers (such as Haskell's $read$ function).
- ☐ Data compression.
- ☐ Data conversion (Jansson and Jeuring, ESOP'99).
- ☐ Digital searching (Hinze, JFP).

A brief outline of the talk

- √ Why? (2 – 5)
- ☐ How? (7 – 17)
- ☐ Making generic programming more generic (19 – 28).

The structure of functional languages

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

- ☐ values
- ☐ types (imposing structure on the value level)
- ☐ kinds (imposing structure on the type level)

Sometimes there is even a fourth level.

- ☐ superkinds (imposing structure on the kind 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
		$\mid$	$\mathbf{case}\ t\ \mathbf{of}\ \{Inl\ a_1 \rightarrow t_1; Inr\ a_2 \rightarrow t_2\}$	
				$\lfloor$case analysis
		$\mid$	(t, u)	pairing
		$\mid$	$outl\ t$	left projection
		$\mid$	$outr\ t$	right projection

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
		$\mid$	$T \rightarrow U$	function type

Kinds

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

Example types

Natural numbers, parametric lists, and generalized rose trees are given by

$$\begin{aligned} Nat &= Rec\,N\,.\,1 + N \\ List &= \Lambda A\,.\,Rec\,L\,.\,1 + A \times L \\ GRose &= \Lambda F\,.\,\Lambda A\,.\,Rec\,T\,.\,A \times F\ \ T\ \ . \end{aligned}$$

We have $Nat :: \star$, $List :: \star \rightarrow \star$, and $GRose :: (\star \rightarrow \star) \rightarrow (\star \rightarrow \star)$.

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

$$\begin{aligned} \mathbf{data}\ Nat &= Zero \mid Succ\ Nat \\ \mathbf{data}\ List\ a &= Nil \mid Cons\ a\ (List\ a) \\ \mathbf{data}\ GRose\ f\ a &= Node\ a\ (f\ (GRose\ f\ a)) \end{aligned}$$

Defining generic equality

To define a generic function it suffices to specify cases for primitive types, that is, 1 and for primitive type constructors, that is, '$+$', '$\times$' and '$\rightarrow$' (equality cannot be defined for '$\rightarrow$').

$$
\begin{array}{lcl}
equal\langle A\rangle & :: & A \rightarrow A \rightarrow Bool \\
equal\langle 1\rangle\ ()\ () & = & True \\
equal\langle A+B\rangle\ (Inl\ a_1)\ (Inl\ a_2) & = & equal\langle A\rangle\ a_1\ a_2 \\
equal\langle A+B\rangle\ (Inl\ a_1)\ (Inr\ b_2) & = & False \\
equal\langle A+B\rangle\ (Inr\ b_1)\ (Inl\ a_2) & = & False \\
equal\langle A+B\rangle\ (Inr\ b_1)\ (Inr\ b_2) & = & equal\langle B\rangle\ b_1\ b_2 \\
equal\langle A\times B\rangle\ (a_1,b_1)\ (a_2,b_2) & = & equal\langle A\rangle\ a_1\ a_2 \wedge equal\langle B\rangle\ b_1\ b_2
\end{array}
$$

NB. The type argument is enclosed in angle brackets.

Applying generic equality

Using $equal$ we can, for instance, test naturals for equality (note that $Zero = Inl\ ()$ and $Succ\ n = Inr\ n$).

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

NB. Reductions take place both at the value and at the type level.

Specializing generic equality

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

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

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

☞ *Basic idea:* Let function follow type.

$$eqList\ eqBool \qquad eqList\ (eqList\ eqNat)$$

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

Specializing generic equality (continued)

The following equations extend the definition of $equal$:

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

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

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

Specializing generic equality (continued)

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

$$\begin{array}{lcl} equal_{Nat} & :: & Nat \to Nat \to Bool \\ equal_{Nat} & = & rec\; equal_N \,.\, equal_+ \; equal_1 \; equal_N \end{array}$$

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

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

Specializing generic equality (continued)

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

$$\begin{array}{lcl} equal_{List} & :: & \forall A \,.\, (A \to A \to Bool) \to (List\; A \to List\; A \to Bool) \\ equal_{List} & = & \lambda equal_A \,.\, rec\; equal_L \,.\, equal_+ \; equal_1 \\ & & \qquad (equal_\times \; equal_A \; equal_L) \end{array}$$

Removing the abstract clothing we obtain

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

A brief outline of the talk

- √ Why? (2 – 5)
- √ How? (7 – 17)
- ☐ Making generic programming more generic (19 – 28).

An uninteresting example: generic identity

The most uninteresting generic function is probably the identity function.

$$\begin{array}{lcl} copy\langle A \rangle & :: & A \to A \\ copy\langle 1 \rangle\; () & = & () \\ copy\langle A + B \rangle\; (Inl\; a) & = & Inl\; (copy\langle A \rangle\; a) \\ copy\langle A + B \rangle\; (Inr\; b) & = & Inr\; (copy\langle B \rangle\; b) \\ copy\langle A \times B \rangle\; (a, b) & = & (copy\langle A \rangle\; a, copy\langle B \rangle\; b) \end{array}$$

Would you agree?

Specializing generic identity

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

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

Removing the abstract clothing we obtain

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

Hmm, the code looks familiar.

From $copy$ to map

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

$$
\begin{array}{lcl}
COPY\langle\star\rangle\ T & = & T \to T \\
COPY\langle K \to L\rangle\ T & = & \forall A\,.\,COPY\langle K\rangle\ A \\
 & & \quad \to COPY\langle L\rangle\ (T\ A)
\end{array}
$$

to

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

A mapping function for all types of all kinds!

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

$$
\begin{array}{lcl}
map\langle A :: K\rangle & :: & Map\langle K\rangle\ A\ A \\
map\langle 1\rangle\ () & = & () \\
map\langle +\rangle\ mapA\ mapB\ (Inl\ a) & = & Inl\ (mapA\ a) \\
map\langle +\rangle\ mapA\ mapB\ (Inr\ b) & = & Inr\ (mapB\ b) \\
map\langle \times\rangle\ mapA\ mapB\ (a, b) & = & (mapA\ a, mapB\ b)
\end{array}
$$

Instances of Map

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

Specializing the generic mapping function

The mapping function for generalized rose trees is given by

$$
\begin{array}{lll}
mapGRose & :: & \forall f_1\ f_2 \,.\, (\forall b_1\ b_2 \,.\, (b_1 \to b_2) \to (f_1\ b_1 \to f_2\ b_2)) \\
& & \quad \to (\forall a_1\ a_2 \,.\, (a_1 \to a_2) \\
& & \qquad \to (GRose\ f_1\ a_1 \to GRose\ f_2\ a_2)) \\
\multicolumn{3}{l}{mapGRose\ mapf\ mapa\ (Node\ a\ ts)} \\
& = & Node\ (mapa\ a)\ (mapf\ (mapGRose\ mapf\ mapa)\ ts) \ .
\end{array}
$$

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

Generic equality, revisited

The type of $equal$

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

can be generalized to

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

NB. The second equation is always of this form!

Specializing generic equality

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

$$
\begin{array}{lll}
equal\langle F :: \star \to \star \rangle & :: & \forall A_1\ A_2 \,.\, (A_1 \to A_2 \to Bool) \\
& & \quad \to (F\ A_1 \to F\ A_2 \to Bool)
\end{array}
$$

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.

A generic size function

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

$$
\begin{array}{lll}
count\langle A :: K \rangle & :: & Count\langle K \rangle\ A \\
count\langle 1 \rangle\ () & = & 0 \\
count\langle + \rangle\ countA\ countB\ (Inl\ a) & = & countA\ a \\
count\langle + \rangle\ countA\ countB\ (Inr\ b) & = & countB\ b \\
count\langle \times \rangle\ countA\ countB\ (a, b) & = & countA\ a + countB\ b
\end{array}
$$

Specializing the generic size function

For unary type constructors $count$ has type

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

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

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

A brief outline of the talk

- √ Why? (2 – 5)
- √ How? (7 – 17)
- √ Making generic programming more generic (19 – 28).

Summary

- ☐ Previous approaches to polytypic programming (PolyP) were restricted in that they only allowed to parameterize values by types of one fixed kind.
- ☐ However, specializing a polytypic value works for all types of all kinds (Hinze, POPL 00).
- ☐ Now, giving the programmer the ability to assign polykinded types to polytypic values greatly enhances genericity: a single polytypic definition, for instance, yields mapping functions for types of arbitrary kinds (Hinze, MPC 2000).

Making generic programming more generic

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)

Making generic programming more generic

In 'ordinary' programming we may define

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

Generic 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).