A NEW APPROACH TO GENERIC FUNCTIONAL PROGRAMMING

RALF HINZE

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

September, 1999

A brief guide to the lecture

We propose a new framework for generic programming that allows you to define functions generically for all datatypes definable in Haskell.

Advertisement *or* Why you might want to become a generic programmer

My work on type-structure polymorphism was motivated by experiences with languages such as ML and Miranda which offered both polymorphic type inference and user-defined algebraic types. As I worked with these languages, I discovered that I had become spoiled in my earlier exposure to APL and LISP by the primitive availability of various useful functions on structures (e.g., map, length, etc.). The promise of polymorphism soured a bit when I realized I must re-define these useful functions at each new structural type I defined. The relevant functions were easy enough to define, but that was the problem: clearly the compiler ought to be able to derive the appropriate definitions automatically, given some generic template.

— Fritz Ruehr

Further examples for generic functions: pretty printers ($show$), parsers ($read$), data conversion, data compression, equality ($==$), comparison functions ($compare$).

Crazy types *or* The challenge of Haskell's type system

☞ Datatypes may have type arguments.

$$\begin{array}{lcl}\mathbf{data}\ List\ a &=& Nil \mid Cons\ a\ (List\ a)\\ \mathbf{data}\ Rose\ a &=& Node\ a\ (List\ (Rose\ a))\end{array}$$

NB. $List$ and $Rose$ have kind $\star \to \star$.

☞ The type arguments can range over type constructors.

$$\begin{array}{lcl}\mathbf{data}\ GRose\ f\ a &=& GNode\ a\ (f\ (GRose\ f\ a))\\ \mathbf{data}\ Fix\ f &=& In\ (f\ (Fix\ f))\\ \mathbf{data}\ BaseList\ a\ b &=& Nil \mid Cons\ a\ b\end{array}$$

NB. We have $GRose :: (\star \to \star) \to (\star \to \star)$, $Fix :: (\star \to \star) \to \star$ and $BaseList :: \star \to (\star \to \star)$.

Crazy types *or* The challenge of Haskell's type system

☞ Definitions can be mutually recursive.

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Rose}\ a & = & \mathit{Node}\ a\ (\mathit{Forest}\ a) \\ \mathbf{data}\ \mathit{Forest}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ (\mathit{Rose}\ a)\ (\mathit{Forest}\ a) \end{array}$$

☞ Type recursion can be nested.

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Fork}\ a & = & \mathit{Fork}\ a\ a \\ \mathbf{data}\ \mathit{Sequ}\ a & = & \mathit{Empty} \\ & \mid & \mathit{Zero}\ (\mathit{Sequ}\ (\mathit{Fork}\ a)) \\ & \mid & \mathit{One}\ a\ (\mathit{Sequ}\ (\mathit{Fork}\ a)) \end{array}$$

Crazy types *or* The challenge of Haskell's type system

☞ We can even combine abstraction over type constructors and nested recursion.

$$\begin{array}{l} \mathbf{data}\ \mathit{MapFork}\ m\ v \\ \quad =\ \ \mathit{TrieFork}\ (m\ (m\ v)) \\ \mathbf{data}\ \mathit{MapSequ}\ m\ v \\ \quad =\ \ \mathit{TrieSequ}\ v \\ \qquad\qquad\qquad (\mathit{MapSequ}\ (\mathit{MapFork}\ m)\ v) \\ \qquad\qquad\qquad (m\ (\mathit{MapSequ}\ (\mathit{MapFork}\ m)\ v)) \end{array}$$

NB. We have $\mathit{MapFork}, \mathit{MapSequ} :: (\star \to \star) \to (\star \to \star)$.

Do you want to become a generic programmer? *or* Defining generic functions

A polytypic function is defined by *induction on the structure of types*.

[By contrast, a monotypic function is typically defined by induction on the structure of values.]

Hence, we must first explicit the structure of types . . .

Modelling Haskell's data declarations

Haskell's **data** construct combines many features: type abstraction, n-ary sums, n-ary products, and type recursion. The structure becomes more apparent if we omit constructor names and use '+' and '×'.

$$\begin{array}{lcl} \mathit{List}\ a & = & 1 + a \times \mathit{List}\ a \\ \mathit{GRose}\ f\ a & = & a \times f\ (\mathit{GRose}\ f\ a) \\ \mathit{Fork}\ a & = & a \times a \\ \mathit{Sequ}\ a & = & 1 + \mathit{Sequ}\ (\mathit{Fork}\ a) + a \times \mathit{Sequ}\ (\mathit{Fork}\ a) \end{array}$$

We treat 1, '+', and '×' as if they were given by

$$\begin{array}{lcl} \mathbf{data}\ 1 & = & () \\ \mathbf{data}\ a_1 + a_2 & = & \mathit{Inl}\ a_1 \mid \mathit{Inr}\ a_2 \\ \mathbf{data}\ a_1 \times a_2 & = & (a_1, a_2)\ . \end{array}$$

Normal form of types of kind $\star$

We can view $List$, $GRose$ etc as (ordinary) functions over type terms with 1, Int, '$+$', and '$\times$' acting as (ordinary) constructors.

Normal form of type terms of kind $\star$ (we assume that Int is the only primitive type):

$$T \quad ::= \quad 1 \mid Int \mid (T + T) \mid (T \times T) \ .$$

☞ The normal form of a type term is usually infinite if the definition involves recursion. *Theoretical background:* Böhm trees.

Defining $\star$-indexed values

Now, to define a type-indexed value it suffices to specify cases for primitive types (1 and Int) and for primitive type constructors ('$+$' and '$\times$').

Example. Binary encoding, ie a simple form of data compression.

$$\begin{array}{lcl}
\mathbf{type}\ Bin & = & [Bit] \\
\mathbf{data}\ Bit & = & 0 \mid 1 \\
enc\langle a :: \star\rangle & :: & a \to Bin \\
enc\langle 1\rangle\ () & = & [\,] \\
enc\langle Int\rangle\ n & = & encInt\ n \\
enc\langle a_1 + a_2\rangle\ (Inl\ x_1) & = & 0 : enc\langle a_1\rangle\ x_1 \\
enc\langle a_1 + a_2\rangle\ (Inr\ x_2) & = & 1 : enc\langle a_2\rangle\ x_2 \\
enc\langle a_1 \times a_2\rangle\ (x_1, x_2) & = & enc\langle a_1\rangle\ x_1 +\!\!+\ enc\langle a_2\rangle\ x_2
\end{array}$$

Defining $\star$-indexed values

Example. Decoding an encoded value.

$$\begin{array}{lcl}
dec\langle a :: \star\rangle & :: & Bin \to (a, Bin) \\
dec\langle 1\rangle\ bs & = & ((), bs) \\
dec\langle Int\rangle\ bs & = & decInt\ bs \\
dec\langle a_1 + a_2\rangle\ [\,] & = & error\ \texttt{"dec"} \\
dec\langle a_1 + a_2\rangle\ (0 : bs) & = & \mathbf{let}\ (x_1, bs') = dec\langle a_1\rangle\ bs\ \mathbf{in}\ (Inl\ x_1, bs') \\
dec\langle a_1 + a_2\rangle\ (1 : bs) & = & \mathbf{let}\ (x_2, bs') = dec\langle a_2\rangle\ bs\ \mathbf{in}\ (Inr\ x_2, bs') \\
dec\langle a_1 \times a_2\rangle\ bs & = & \mathbf{let}\ (x_1, bs_1) = dec\langle a_1\rangle\ bs \\
 & & \qquad (x_2, bs_2) = dec\langle a_2\rangle\ bs_1 \\
 & & \mathbf{in}\ ((x_1, x_2), bs_2)
\end{array}$$

Defining $\star$-indexed values

Example. A generic definition of Haskell's $compare$ function.

$$\begin{array}{lcl}
\mathbf{data}\ Ordering & = & LT \mid EQ \mid GT \\
cmp\langle a :: \star\rangle & :: & a \to a \to Ordering \\
cmp\langle 1\rangle\ ()\ () & = & EQ \\
cmp\langle Int\rangle\ m\ n & = & cmpInt\ m\ n \\
cmp\langle a_1 + a_2\rangle\ (Inl\ x_1)\ (Inl\ y_1) & = & cmp\langle a_1\rangle\ x_1\ y_1 \\
cmp\langle a_1 + a_2\rangle\ (Inl\ x_1)\ (Inr\ y_2) & = & LT \\
cmp\langle a_1 + a_2\rangle\ (Inr\ x_2)\ (Inl\ y_1) & = & GT \\
cmp\langle a_1 + a_2\rangle\ (Inr\ x_2)\ (Inr\ y_2) & = & cmp\langle a_2\rangle\ x_2\ y_2 \\
cmp\langle a_1 \times a_2\rangle\ (x_1, x_2)\ (y_1, y_2) & = & cmp\langle a_1\rangle\ x_1\ y_1 \lhd cmp\langle a_2\rangle\ x_2\ y_2 \\
\multicolumn{3}{l}{ord_1 \lhd ord_2 = \quad \mathbf{case}\ ord_1\ \mathbf{of}\ \{EQ \to ord_2; _ \to ord_1\}}
\end{array}$$

NB. $(\lhd)$ implements the lexicographic product of two orderings.

Defining $\star \to \star$-indexed values

Example. Count the number of values of type x in a given structure of type $f\ x$.

$$
\begin{array}{lcl}
size\langle f :: \star \to \star \rangle & :: & \forall x . f\ x \to Int \\
size\langle \Lambda a \to a \rangle\ x & = & 1 \\
size\langle \Lambda a \to 1 \rangle\ () & = & 0 \\
size\langle \Lambda a \to Int \rangle\ n & = & 0 \\
size\langle \Lambda a \to f_1\ a + f_2\ a \rangle\ (Inl\ x_1) & = & size\langle f_1 \rangle\ x_1 \\
size\langle \Lambda a \to f_1\ a + f_2\ a \rangle\ (Inr\ x_2) & = & size\langle f_2 \rangle\ x_2 \\
size\langle \Lambda a \to f_1\ a \times f_2\ a \rangle\ (x_1, x_2) & = & size\langle f_1 \rangle\ x_1 + size\langle f_2 \rangle\ x_2
\end{array}
$$

Defining $\star \to \star$-indexed values

Example. A generic definition of map (aka $fmap$ in Haskell 98).

$$
\begin{array}{lcl}
map\langle f :: \star \to \star \rangle & :: & \forall a, b . (a \to b) \to (f\ a \to f\ b) \\
map\langle Id \rangle\ \varphi\ x & = & \varphi\ x \\
map\langle K\ t \rangle\ \varphi\ x & = & x \\
map\langle \Lambda a \to f_1\ a + f_2\ a \rangle\ \varphi\ (Inl\ x_1) & = & Inl\ (map\langle f_1 \rangle\ \varphi\ x_1) \\
map\langle \Lambda a \to f_1\ a + f_2\ a \rangle\ \varphi\ (Inr\ x_2) & = & Inr\ (map\langle f_2 \rangle\ \varphi\ x_2) \\
map\langle \Lambda a \to f_1\ a \times f_2\ a \rangle\ \varphi\ (x_1, x_2) & = & (map\langle f_1 \rangle\ \varphi\ x_1, map\langle f_2 \rangle\ \varphi\ x_2)
\end{array}
$$

NB. Id abbreviates $\Lambda a \to a$ and $K\ t = \Lambda a \to t$ covers both $\Lambda a \to 1$ and $\Lambda a \to Int$.

Defining $\star \to \star$-indexed values

Example. Lambert Meertens' $crush$ function, which collapses a structure of values of type a into a single value of type a.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{crush\langle f :: \star \to \star \rangle :: \forall a . a \to (a \to a \to a) \to (f\ a \to a)} \\
crush\langle f \rangle\ e\ op & = & red\langle f \rangle \\
\quad \mathbf{where} & & \\
\quad red\langle f :: \star \to \star \rangle & :: & f\ a \to a \\
\quad red\langle Id \rangle\ x & = & x \\
\quad red\langle K\ t \rangle\ x & = & e \\
\quad red\langle \Lambda a \to f_1\ a + f_2\ a \rangle\ (Inl\ x_1) & = & red\langle f_1 \rangle\ x_1 \\
\quad red\langle \Lambda a \to f_1\ a + f_2\ a \rangle\ (Inr\ x_2) & = & red\langle f_2 \rangle\ x_2 \\
\quad red\langle \Lambda a \to f_1\ a \times f_2\ a \rangle\ (x_1, x_2) & = & red\langle f_1 \rangle\ x_1\ `op`\ red\langle f_2 \rangle\ x_2
\end{array}
$$

NB. We have $size\langle f \rangle = crush\langle f \rangle\ 1\ (+)$.

Defining $\star \to \star$-indexed values

☞ Are the type patterns complete? Yes, they are!

Recall the normal form of type terms of kind $\star$:

$$T \ ::= \ 1 \mid Int \mid (T + T) \mid (T \times T)\ .$$

Assume that we are given a type t of kind $\star \to \star$. Applying η-expansion we have $t = \Lambda a \to t\ a$. Normal form of $t\ a$:

$$T \ ::= \ a \mid 1 \mid Int \mid (T + T) \mid (T \times T)\ .$$

Now, to make the passing of a explicit abstract a out.

Defining $(\star \to \star) \to (\star \to \star)$-indexed values

Example. A higher-order map.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{hmap\langle h :: (\star \to \star) \to (\star \to \star)\rangle} \\
\multicolumn{3}{l}{\quad :: (Functor\ g, Functor\ g') \Rightarrow (\forall x. g\ x \to g'\ x) \to (\forall x. h\ g\ x \to h\ g'\ x)} \\
hmap\langle \Lambda f\ \ a \to f\ (h_1\ f\ \ a)\rangle\ \varphi & = & \varphi \cdot fmap\ (hmap\langle h_1\rangle\ \varphi) \\
hmap\langle \Lambda f\ \ a \to a\rangle\ \varphi & = & id \\
hmap\langle \Lambda f\ \ a \to 1\rangle\ \varphi & = & id \\
hmap\langle \Lambda f\ \ a \to Int\rangle\ \varphi & = & id \\
hmap\langle \Lambda f\ \ a \to h_1\ f\ \ a + h_2\ f\ \ a\rangle\ \varphi\ (Inl\ x_1) & = & Inl\ (hmap\langle h_1\rangle\ \varphi\ x_1) \\
hmap\langle \Lambda f\ \ a \to h_1\ f\ \ a + h_2\ f\ \ a\rangle\ \varphi\ (Inr\ x_2) & = & Inr\ (hmap\langle h_2\rangle\ \varphi\ x_2) \\
\multicolumn{3}{l}{hmap\langle \Lambda f\ \ a \to h_1\ f\ \ a \times h_2\ f\ \ a\rangle\ \varphi\ (x_1, x_2)} \\
\multicolumn{3}{l}{\quad = \ (hmap\langle h_1\rangle\ \varphi\ x_1, hmap\langle h_2\rangle\ \varphi\ x_2)}
\end{array}
$$

Defining $(\star \to \star) \to (\star \to \star)$-indexed values

☞ Are the type patterns complete? Yes, they are!

Recall the normal form of type terms of kind $\star$:

$$T \ ::= \ 1 \mid Int \mid (T + T) \mid (T \times T)\ .$$

Assume that we are given a type t of kind $(\star \to \star) \to (\star \to \star)$. Applying η-expansion we have $t = \Lambda f\ \ a \to t\ f\ \ a$. Normal form of $t\ f\ \ a$:

$$T \ ::= \ a \mid f\ \ T \mid 1 \mid Int \mid (T + T) \mid (T \times T)\ .$$

Now, to make the passing of f and a explicit abstract them out.

Relax *or* Specializing generic functions

☞ How can we specialize $enc\langle a\rangle$, $size\langle f\rangle$, and $hmap\langle h\rangle$ for given instances of a, f, and h?

Examples.

$$
\begin{array}{l}
enc\langle GRose\ List\ Int\rangle \\
enc\langle Fix\ (BaseList\ Bool)\rangle \\
size\langle \Lambda a \to List\ (Rose\ a)\rangle \\
size\langle GRose\ List\rangle \\
size\langle \Lambda a \to Fix\ (BaseList\ a)\rangle \\
hmap\langle GRose\rangle
\end{array}
$$

Relax *or* Specializing generic functions

☞ The specialization should be compositional.

$$enc\langle List\ Bool\rangle \qquad\qquad enc\langle List\ (List\ Int)\rangle$$

The code generated for $List$ should be shared.

Idea: Let function follow type.

$$encList\ encBool \qquad\qquad encList\ (encList\ encInt)$$

Since $List$ is a function on types, $encList$ is consequently a function on encoders: it maps $enc\langle a\rangle$ to $enc\langle List\ a\rangle$.

Modelling Haskell's type system: kinds and types

Kinds.

$$K \quad ::= \quad \star \mid (K \to K)$$

Types: let X be a fixed set of type variables, let $C = \{1, \mathit{Int}, (+), (\times)\}$ be a fixed set of primitive type constructors.

$$T \quad ::= \quad X \mid C \mid (T\ T) \mid (\Lambda X \to T) \mid (\mu T)$$

NB. Instead of μ one can alternatively use systems of recursion equations.

NB. C could additionally include the function space constructor '$\to$'.

Modelling Haskell's type system: kinds and types

Examples.

$$\begin{array}{lcl}
\mathit{List} & = & \Lambda a \to \mu(\Lambda \ell \to 1 + a \times \ell) \\
\mathit{GRose} & = & \Lambda f \to \Lambda a \to \mu(\Lambda g \to a \times f\ g) \\
\mathit{Fork} & = & \Lambda a \to a \times a \\
\mathit{Sequ} & = & \mu(\Lambda s\ a \to 1 + s\ (\mathit{Fork}\ a) + a \times s\ (\mathit{Fork}\ a))
\end{array}$$

Specializing $\star$-indexed values

Recall: Type-indexed definitions are inductive on the structure of types.

$$\begin{array}{lcl}
\mathit{poly}\langle a :: \star \rangle & :: & F\ a \\
\mathit{poly}\langle 1 \rangle & = & \mathit{poly}_1 \\
\mathit{poly}\langle \mathit{Int} \rangle & = & \mathit{poly}_{\mathit{Int}} \\
\mathit{poly}\langle a_1 + a_2 \rangle & = & \mathit{poly}_+\ (\mathit{poly}\langle a_1 \rangle)\ (\mathit{poly}\langle a_2 \rangle) \\
\mathit{poly}\langle a_1 \times a_2 \rangle & = & \mathit{poly}_\times\ (\mathit{poly}\langle a_1 \rangle)\ (\mathit{poly}\langle a_2 \rangle)
\end{array}$$

Here, poly is the name of the type-indexed value. The type scheme F and the functions poly_1, $\mathit{poly}_{\mathit{Int}}$, poly_+, and $\mathit{poly}_\times$ are the ingredients that are supplied by the generic programmer.

NB. F may also contain function types, $t_1 \to t_2$, and universally quantified types, $\forall x.t$.

Specializing $\star$-indexed values

☞ Specialization can be seen as an interpretation of the simply typed λ-calculus.

$$\begin{array}{lcl}
\mathit{poly}\langle\!\langle t :: \kappa \rangle\!\rangle\ \varrho & :: & F\langle\!\langle t :: \kappa \rangle\!\rangle \\
\mathit{poly}\langle\!\langle c \rangle\!\rangle\ \varrho & = & \mathit{poly}_c \\
\mathit{poly}\langle\!\langle x \rangle\!\rangle\ \varrho & = & \varrho\ x \\
\mathit{poly}\langle\!\langle t_1\ t_2 \rangle\!\rangle\ \varrho & = & (\mathit{poly}\langle\!\langle t_1 \rangle\!\rangle\ \varrho)\ (\mathit{poly}\langle\!\langle t_2 \rangle\!\rangle\ \varrho) \\
\mathit{poly}\langle\!\langle \Lambda x \to t \rangle\!\rangle\ \varrho & = & \lambda v \to \mathit{poly}\langle\!\langle t \rangle\!\rangle\ (\varrho(x := v)) \\
\mathit{poly}\langle\!\langle \mu t \rangle\!\rangle\ \varrho & = & \mathit{fix}\ (\mathit{poly}\langle\!\langle t \rangle\!\rangle\ \varrho)
\end{array}$$

The type of $\mathit{poly}\langle\!\langle t :: \kappa \rangle\!\rangle\ \varrho$ is defined by induction on the structure of kinds.

$$\begin{array}{lcl}
F\langle\!\langle u :: \star \rangle\!\rangle & = & F\ u \\
F\langle\!\langle u :: \kappa_1 \to \kappa_2 \rangle\!\rangle & = & \forall x. F\langle\!\langle x :: \kappa_1 \rangle\!\rangle \to F\langle\!\langle u\ x :: \kappa_2 \rangle\!\rangle
\end{array}$$

Specializing enc

$$
\begin{array}{lcl}
encList & :: & \forall a.(a \to Bin) \to (List\ a \to Bin) \\
encList\ enca & = & encl \\
\quad \mathbf{where}\ encl\ Nil & = & [0] \\
\qquad\quad encl\ (Cons\ x\ xs) & = & 1 : enca\ x +\!\!+\ encl\ xs \\
encRose & :: & \forall a.(a \to Bin) \to (Rose\ a \to Bin) \\
encRose\ enca & = & encr \\
\quad \mathbf{where}\ encr\ (Node\ x\ ts) & = & enca\ x +\!\!+\ encList\ encr\ ts \\
encGRose & :: & \forall f.(\forall b.(b \to Bin) \to (f\ b \to Bin)) \\
 & & \quad \to (\forall a.(a \to Bin) \to (GRose\ f\ a \to Bin)) \\
encGRose\ encf\ enca & = & encg \\
\quad \mathbf{where}\ encg\ (GNode\ x\ ts) & = & enca\ x +\!\!+\ encf\ encg\ ts \\
encFix & :: & \forall f.(\forall a.(a \to Bin) \to (f\ a \to Bin)) \to (Fix\ f \to Bin) \\
encFix\ encf & = & encr \\
\quad \mathbf{where}\ encr\ (In\ x) & = & encf\ encr\ x \\
encBaseList & :: & \forall a.(a \to Bin) \to (\forall b.(b \to Bin) \to (BaseList\ a\ b \to Bin)) \\
encBaseList\ enca\ encb\ Nil & = & [0] \\
encBaseList\ enca\ encb\ (Cons\ x\ y) & & \\
 & = & 1 : enca\ x +\!\!+\ encb\ y
\end{array}
$$

$$
\begin{array}{lcl}
encFork & :: & \forall a.(a \to Bin) \to (Fork\ a \to Bin) \\
encFork\ enca\ (Fork\ x_1\ x_2) & = & enca\ x_1 +\!\!+\ enca\ x_2 \\
encSequ & :: & \forall a.(a \to Bin) \to (Sequ\ a \to Bin) \\
encSequ\ enca\ Empty & = & [0] \\
encSequ\ enca\ (Zero\ s) & = & 1 : 0 : encSequ\ (encFork\ enca)\ s \\
encSequ\ enca\ (One\ x\ s) & = & 1 : 1 : enca\ x +\!\!+\ encSequ\ (encFork\ enca)\ s \\
encMapFork & :: & \forall m.(\forall w.(w \to Bin) \to (m\ w \to Bin)) \\
 & & \quad \to (\forall v.(v \to Bin) \to (MapFork\ m\ v \to Bin)) \\
encMapFork\ encm\ encv\ (TrieFork\ tf) & & \\
 & = & encm\ (encm\ encv)\ tf \\
encMapSequ & :: & \forall m.(\forall w.(w \to Bin) \to (m\ w \to Bin)) \\
 & & \quad \to (\forall v.(v \to Bin) \to (MapSequ\ m\ v \to Bin)) \\
encMapSequ\ encm\ encv\ (TrieSequ\ te\ tz\ to) & & \\
 & = & encv\ te \\
 & +\!\!+ & encMapSequ\ (encMapFork\ encm)\ encv\ tz \\
 & +\!\!+ & encm\ (encMapSequ\ (encMapFork\ encm)\ encv)\ to
\end{array}
$$

We require polymorphic recursion and rank-n types.

Specializing $\star \to \star$-indexed values

Let

$$
\begin{array}{lcl@{\qquad}lcl}
\underline{1} & = & \Lambda a \to 1 & f_1 \mathbin{\underline{+}} f_2 & = & \Lambda a \to f_1\ a + f_2\ a \\
\underline{Int} & = & \Lambda a \to Int & f_1 \mathbin{\underline{\times}} f_2 & = & \Lambda a \to f_1\ a \times f_2\ a
\end{array}
$$

then we have the following scheme for $\star \to \star$-indexed definitions.

$$
\begin{array}{lcl}
poly\langle f :: \star \to \star \rangle & :: & H\ f \\
poly\langle Id \rangle & = & poly_{Id} \\
poly\langle \underline{1} \rangle & = & poly_{\underline{1}} \\
poly\langle \underline{Int} \rangle & = & poly_{\underline{Int}} \\
poly\langle f_1 \mathbin{\underline{+}} f_2 \rangle & = & poly_{\underline{+}}\ (poly\langle f_1 \rangle)\ (poly\langle f_2 \rangle) \\
poly\langle f_1 \mathbin{\underline{\times}} f_2 \rangle & = & poly_{\underline{\times}}\ (poly\langle f_1 \rangle)\ (poly\langle f_2 \rangle)
\end{array}
$$

Specializing $\star \to \star$-indexed values

Step 1: Lift the types into a higher realm.

$$
\begin{array}{lcl}
\uparrow c & = & \underline{c} \\
\uparrow x & = & \underline{x} \\
\uparrow t_1\ t_2 & = & (\uparrow t_1)\ (\uparrow t_2) \\
\uparrow \Lambda x \to t & = & \Lambda \underline{x} \to \uparrow t \\
\uparrow \mu t & = & \mu(\uparrow t)
\end{array}
$$

Lifting maps a type $t :: \kappa$ to a type $\uparrow t :: \uparrow \kappa$.

$$
\begin{array}{lcl}
\uparrow \star & = & \star \to \star \\
\uparrow \kappa_1 \to \kappa_2 & = & (\uparrow \kappa_1) \to (\uparrow \kappa_2)
\end{array}
$$

Lifting types

Example. Lifting List and GRose yields:

$$
\begin{array}{lcl}
\underline{\mathit{List}} & = & \Lambda\underline{a} \to \mu(\Lambda\underline{\ell} \to \underline{1} \mathbin{\underline{+}} \underline{a} \mathbin{\underline{\times}} \underline{\ell}) \\
\underline{\mathit{GRose}} & = & \Lambda\underline{f} \to \Lambda\underline{a} \to \mu(\Lambda\underline{g} \to \underline{a} \mathbin{\underline{\times}} \underline{f}\ \underline{g})\ .
\end{array}
$$

Expanding the lifted primitives we obtain

$$
\begin{array}{lcl}
\underline{\mathit{List}} & = & \Lambda\underline{a} \to \mu(\Lambda\underline{\ell}\ z \to 1 + \underline{a}\ z \times \underline{\ell}\ z) \\
\underline{\mathit{GRose}} & = & \Lambda\underline{f} \to \Lambda\underline{a} \to \mu(\Lambda\underline{g}\ z \to \underline{a}\ z \times \underline{f}\ \underline{g}\ z)\ .
\end{array}
$$

☞ Crucial property: if t has kind $\star \to \star$, then

$$
t \quad = \quad (\uparrow t)\ \mathit{Id}\ .
$$

Specializing $\star \to \star$-indexed values

Step 2: Define $\mathit{poly}\langle\!\langle\cdot\rangle\!\rangle$ on the lifted types.

$$
\begin{array}{lcl}
\mathit{poly}\langle\!\langle t :: \kappa\rangle\!\rangle\ \varrho & :: & H\langle\!\langle t :: \kappa\rangle\!\rangle \\
\mathit{poly}\langle\!\langle \underline{c}\rangle\!\rangle\ \varrho & = & \mathit{poly}_{\underline{c}} \\
\mathit{poly}\langle\!\langle \underline{x}\rangle\!\rangle\ \varrho & = & \varrho\ \underline{x} \\
\mathit{poly}\langle\!\langle t_1\ t_2\rangle\!\rangle\ \varrho & = & (\mathit{poly}\langle\!\langle t_1\rangle\!\rangle\ \varrho)\ (\mathit{poly}\langle\!\langle t_2\rangle\!\rangle\ \varrho) \\
\mathit{poly}\langle\!\langle \Lambda\underline{x} \to t\rangle\!\rangle\ \varrho & = & \lambda v \to \mathit{poly}\langle\!\langle t\rangle\!\rangle\ (\varrho(\underline{x} := v))\ . \\
\mathit{poly}\langle\!\langle \mu t\rangle\!\rangle\ \varrho & = & \mathit{fix}\ (\mathit{poly}\langle\!\langle t\rangle\!\rangle\ \varrho)
\end{array}
$$

The type of $\mathit{poly}\langle\!\langle t :: \kappa\rangle\!\rangle\ \varrho$ is defined by induction on the structure of kinds.

$$
\begin{array}{lcl}
H\langle\!\langle u :: \uparrow\star\rangle\!\rangle & = & H\ u \\
H\langle\!\langle u :: \uparrow\kappa_1 \to \kappa_2\rangle\!\rangle & = & \forall x. H\langle\!\langle x :: \uparrow\kappa_1\rangle\!\rangle \to H\langle\!\langle u\ x :: \uparrow\kappa_2\rangle\!\rangle
\end{array}
$$

Specializing $\star \to \star$-indexed values

Step 3: Define $\mathit{poly}\langle\cdot\rangle$ in terms of $\mathit{poly}\langle\!\langle\cdot\rangle\!\rangle$.

$$
\mathit{poly}\langle t :: \star \to \star\rangle \quad = \quad \mathit{poly}\langle\!\langle \uparrow t :: \uparrow\star \to \star\rangle\!\rangle\ \epsilon\ \mathit{poly}_{\mathit{Id}}
$$

NB. ϵ is the empty environment.

Specializing size

$$
\begin{array}{lcl}
\mathit{sizeList} & :: & \forall\underline{a}.(\forall x.\underline{a}\ x \to \mathit{Int}) \to (\forall x.\underline{\mathit{List}}\ \underline{a}\ x \to \mathit{Int}) \\
\mathit{sizeList}\ \mathit{sizea} & = & \mathit{sizel} \\
\quad \textbf{where}\ \mathit{sizel}\ \mathit{Nil} & = & 0 \\
\qquad\qquad \mathit{sizel}\ (\mathit{Cons}\ x\ \mathit{xs}) & = & \mathit{sizea}\ x + \mathit{sizel}\ \mathit{xs} \\
\mathit{sizeRose} & :: & \forall\underline{a}.(\forall x.\underline{a}\ x \to \mathit{Int}) \to (\forall x.\underline{\mathit{Rose}}\ \underline{a}\ x \to \mathit{Int}) \\
\mathit{sizeRose}\ \mathit{sizea} & = & \mathit{sizer} \\
\quad \textbf{where}\ \mathit{sizer}\ (\mathit{Node}\ x\ \mathit{ts}) & = & \mathit{sizea}\ x + \mathit{sizeList}\ \mathit{sizer}\ \mathit{ts} \\
\mathit{sizeGRose} & :: & \forall\underline{f}.(\forall\underline{b}.(\forall x.\underline{b}\ x \to \mathit{Int}) \to (\forall x.\underline{f}\ \underline{b}\ x \to \mathit{Int})) \\
 & & \to (\forall\underline{a}.(\forall x.\underline{a}\ x \to \mathit{Int}) \to (\forall x.\underline{\mathit{GRose}}\ \underline{f}\ \underline{a}\ x \to \mathit{Int})) \\
\mathit{sizeGRose}\ \mathit{sizef}\ \mathit{sizea} & = & \mathit{sizeg} \\
\quad \textbf{where}\ \mathit{sizeg}\ (\mathit{GNode}\ x\ \mathit{ts}) & = & \mathit{sizea}\ x + \mathit{sizef}\ \mathit{sizeg}\ \mathit{ts} \\
\mathit{sizeFix} & :: & \forall\underline{f}.(\forall\underline{b}.((\forall x.\underline{b}\ x \to \mathit{Int}) \to (\forall x.\underline{f}\ \underline{b}\ x \to \mathit{Int}))) \\
 & & \qquad\qquad \to (\forall x.\underline{\mathit{Fix}}\ \underline{f}\ x \to \mathit{Int}) \\
\mathit{sizeFix}\ \mathit{sizef} & = & \mathit{sizer}\ \textbf{where}\ \mathit{sizer}\ (\mathit{In}\ x) = \mathit{sizef}\ \mathit{sizer}\ x \\
\mathit{sizeBaseList} & :: & \forall\underline{a}.(\forall x.\underline{a}\ x \to \mathit{Int}) \to (\forall\underline{b}.(\forall x.\underline{b}\ x \to \mathit{Int}) \\
 & & \to (\forall x.\underline{\mathit{BaseList}}\ \underline{a}\ \underline{b}\ x \to \mathit{Int})) \\
\mathit{sizeBaseList}\ \mathit{sizea}\ \mathit{sizeb}\ \mathit{Nil} & = & 0 \\
\mathit{sizeBaseList}\ \mathit{sizea}\ \mathit{sizeb}\ (\mathit{Cons}\ x\ y) & & \\
 & = & \mathit{sizea}\ x + \mathit{sizeb}\ y
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{sizeFork} & :: & \forall \underline{a}.(\forall x.\underline{a}\ x \to \mathit{Int}) \to (\forall x.\underline{\mathit{Fork}}\ \underline{a}\ x \to \mathit{Int}) \\
\mathit{sizeFork}\ \mathit{sizea}\ (\mathit{Fork}\ x_1\ x_2) & = & \mathit{sizea}\ x_1 + \mathit{sizea}\ x_2 \\
\mathit{sizeSequ} & :: & \forall \underline{a}.(\forall x.\underline{a}\ x \to \mathit{Int}) \to (\forall x.\underline{\mathit{Sequ}}\ \underline{a}\ x \to \mathit{Int}) \\
\mathit{sizeSequ}\ \mathit{sizea}\ \mathit{Empty} & = & 0 \\
\mathit{sizeSequ}\ \mathit{sizea}\ (\mathit{Zero}\ s) & = & \mathit{sizeSequ}\ (\mathit{sizeFork}\ \mathit{sizea})\ s \\
\mathit{sizeSequ}\ \mathit{sizea}\ (\mathit{One}\ x\ s) & = & \mathit{sizea}\ x + \mathit{sizeSequ}\ (\mathit{sizeFork}\ \mathit{sizea})\ s \\
\mathit{sizeMapFork} & :: & \forall \underline{m}.(\forall \underline{w}.(\forall x.\underline{w}\ x \to \mathit{Int}) \to (\forall x.\underline{m}\ \underline{w}\ x \to \mathit{Int})) \\
 & & \to (\forall \underline{v}.(\forall x.\underline{v}\ x \to \mathit{Int}) \to (\forall x.\underline{\mathit{MapFork}}\ \underline{m}\ \underline{v}\ x \to \mathit{Int})) \\
\multicolumn{3}{l}{\mathit{sizeMapFork}\ \mathit{sizem}\ \mathit{sizev}\ (\mathit{TrieFork}\ \mathit{tf})} \\
 & = & \mathit{sizem}\ (\mathit{sizem}\ \mathit{sizev})\ \mathit{tf} \\
\mathit{sizeMapSequ} & :: & \forall \underline{m}.(\forall \underline{w}.(\forall x.\underline{w}\ x \to \mathit{Int}) \to (\forall x.\underline{m}\ \underline{w}\ x \to \mathit{Int})) \\
 & & \to (\forall \underline{v}.(\forall x.\underline{v}\ x \to \mathit{Int}) \to (\forall x.\underline{\mathit{MapSequ}}\ \underline{m}\ \underline{v}\ x \to \mathit{Int})) \\
\multicolumn{3}{l}{\mathit{sizeMapSequ}\ \mathit{sizem}\ \mathit{sizev}\ (\mathit{TrieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to})} \\
 & = & \mathit{sizev}\ \mathit{te} \\
 & + & \mathit{sizeMapSequ}\ (\mathit{sizeMapFork}\ \mathit{sizem})\ \mathit{sizev}\ \mathit{tz} \\
 & + & \mathit{sizem}\ (\mathit{sizeMapSequ}\ (\mathit{sizeMapFork}\ \mathit{sizem})\ \mathit{sizev})\ \mathit{to}
\end{array}
$$

Unlifting types

☞ The lifted types, $\underline{F}$, appear in the type signatures. Can we express $\underline{F}$ in terms of F?

Haskell B. Curry helps. We require combinators on types.

$$
\begin{array}{lcl}
K\ t & = & \Lambda x \to t \\
S\ t_1\ t_2 & = & \Lambda x \to (t_1\ x)\ (t_2\ x)
\end{array}
$$

NB. K and S are polymorphic wrt the kinds.

Unlifting types

We relate types of kind $\uparrow\kappa$ to types of kind $\star \to \kappa$.

Definition.

$$
\begin{array}{rcl}
t \sim_{\star} t' & \iff & t = t' \\
t \sim_{\kappa_1 \to \kappa_2} t' & \iff & \forall x, x'.(x \sim_{\kappa_1} x' \Rightarrow t\ x \sim_{\kappa_2} S\ t'\ x')
\end{array}
$$

Proposition.

$$
\uparrow t \quad \sim_{\kappa} \quad K\ t
$$

Unlifting types

Examples.

$$
\begin{array}{r}
\underline{\mathit{List}}\ a = \Lambda x \to \mathit{List}\ (a\ x) \\
\underline{\mathit{GRose}}\ \underline{f}\ a = \Lambda x \to \mathit{GRose}\ (f\ x)\ (a\ x) \quad \Leftarrow \quad \underline{f}\ b = \Lambda x \to (f\ x)\ (b\ x)
\end{array}
$$

NB. $S\ (K\ t)\ u = t \cdot u$.

Specializing $size$, revisited

$$
\begin{array}{lcl}
\mathit{sizeList} & :: & \forall a.(\forall x.a\ x \to \mathit{Int}) \to (\forall x.\mathit{List}\ (a\ x) \to \mathit{Int}) \\
\mathit{sizeList}\ \mathit{sizea} & = & \mathit{sizel} \\
\ \mathbf{where}\ \mathit{sizel}\ \mathit{Nil} & = & 0 \\
\qquad \mathit{sizel}\ (\mathit{Cons}\ x\ \mathit{xs}) & = & \mathit{sizea}\ x + \mathit{sizel}\ \mathit{xs} \\
\mathit{sizeRose} & :: & \forall a.(\forall x.a\ x \to \mathit{Int}) \to (\forall x.\mathit{Rose}\ (a\ x) \to \mathit{Int}) \\
\mathit{sizeRose}\ \mathit{sizea} & = & \mathit{sizer} \\
\ \mathbf{where}\ \mathit{sizer}\ (\mathit{Node}\ x\ \mathit{ts}) & = & \mathit{sizea}\ x + \mathit{sizeList}\ \mathit{sizer}\ \mathit{ts} \\
\mathit{sizeGRose} & :: & \forall f.(\forall b.(\forall x.b\ x \to \mathit{Int}) \to (\forall x.(f\ x)\ (b\ x) \to \mathit{Int})) \\
 & & \to (\forall a.(\forall x.a\ x \to \mathit{Int}) \to (\forall x.\mathit{GRose}\ (f\ x)\ (a\ x) \to \mathit{Int})) \\
\mathit{sizeGRose}\ \mathit{sizef}\ \mathit{sizea} & = & \mathit{sizeg} \\
\ \mathbf{where}\ \mathit{sizeg}\ (\mathit{GNode}\ x\ \mathit{ts}) & = & \mathit{sizea}\ x + \mathit{sizef}\ \mathit{sizeg}\ \mathit{ts} \\
\mathit{sizeFix} & :: & \forall f.(\forall b.((\forall x.b\ x \to \mathit{Int}) \to (\forall x.(f\ x)\ (b\ x) \to \mathit{Int}))) \\
 & & \quad \to (\forall x.\mathit{Fix}\ (f\ x) \to \mathit{Int}) \\
\mathit{sizeFix}\ \mathit{sizef} & = & \mathit{sizer}\ \mathbf{where}\ \mathit{sizer}\ (\mathit{In}\ x) = \mathit{sizef}\ \mathit{sizer}\ x \\
\mathit{sizeBaseList} & :: & \forall a.(\forall x.a\ x \to \mathit{Int}) \to (\forall b.(\forall x.b\ x \to \mathit{Int}) \\
 & & \to (\forall x.\mathit{BaseList}\ (a\ x)\ (b\ x) \to \mathit{Int})) \\
\mathit{sizeBaseList}\ \mathit{sizea}\ \mathit{sizeb}\ \mathit{Nil} & = & 0 \\
\mathit{sizeBaseList}\ \mathit{sizea}\ \mathit{sizeb}\ (\mathit{Cons}\ x\ y) & & \\
 & = & \mathit{sizea}\ x + \mathit{sizeb}\ y
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{sizeFork} & :: & \forall a.(\forall x.a\ x \to \mathit{Int}) \to (\forall x.\mathit{Fork}\ (a\ x) \to \mathit{Int}) \\
\mathit{sizeFork}\ \mathit{sizea}\ (\mathit{Fork}\ x_1\ x_2) & = & \mathit{sizea}\ x_1 + \mathit{sizea}\ x_2 \\
\mathit{sizeSequ} & :: & \forall a.(\forall x.a\ x \to \mathit{Int}) \to (\forall x.\mathit{Sequ}\ (a\ x) \to \mathit{Int}) \\
\mathit{sizeSequ}\ \mathit{sizea}\ \mathit{Empty} & = & 0 \\
\mathit{sizeSequ}\ \mathit{sizea}\ (\mathit{Zero}\ s) & = & \mathit{sizeSequ}\ (\mathit{sizeFork}\ \mathit{sizea})\ s \\
\mathit{sizeSequ}\ \mathit{sizea}\ (\mathit{One}\ x\ s) & = & \mathit{sizea}\ x + \mathit{sizeSequ}\ (\mathit{sizeFork}\ \mathit{sizea})\ s \\
\mathit{sizeMapFork} & :: & \forall m.(\forall w.(\forall x.w\ x \to \mathit{Int}) \to (\forall x.(m\ x)\ (w\ x) \to \mathit{Int})) \\
 & & \to (\forall v.(\forall x.v\ x \to \mathit{Int}) \to (\forall x.\mathit{MapFork}\ (m\ x)\ (v\ x) \to \mathit{Int})) \\
\mathit{sizeMapFork}\ \mathit{sizem}\ \mathit{sizev}\ (\mathit{TrieFork}\ \mathit{tf}) & & \\
 & = & \mathit{sizem}\ (\mathit{sizem}\ \mathit{sizev})\ \mathit{tf} \\
\mathit{sizeMapSequ} & :: & \forall m.(\forall w.(\forall x.w\ x \to \mathit{Int}) \to (\forall x.(m\ x)\ (w\ x) \to \mathit{Int})) \\
 & & \to (\forall v.(\forall x.v\ x \to \mathit{Int}) \to (\forall x.\mathit{MapSequ}\ (m\ x)\ (v\ x) \to \mathit{Int})) \\
\mathit{sizeMapSequ}\ \mathit{sizem}\ \mathit{sizev}\ (\mathit{TrieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) & & \\
 & = & \mathit{sizev}\ \mathit{te} \\
 & + & \mathit{sizeMapSequ}\ (\mathit{sizeMapFork}\ \mathit{sizem})\ \mathit{sizev}\ \mathit{tz} \\
 & + & \mathit{sizem}\ (\mathit{sizeMapSequ}\ (\mathit{sizeMapFork}\ \mathit{sizem})\ \mathit{sizev})\ \mathit{to}
\end{array}
$$

We require polymorphic recursion, rank-n types, and a strong form of type-constructor polymorphism.

Concluding remarks

- The new approach to generic functional programming covers definitions indexed by type constructors up to *second-order* kinds.

- Generic values can be specialized to arbitrary Haskell datatypes.

- It is easy to include 'ad-hoc' definitions for type constructors up to *first-order* kinds.

$$
\begin{array}{lcl}
\mathit{enc}\langle \mathit{List}\ a\rangle & = & \ldots\ (\mathit{enc}\langle a\rangle)\ \ldots \\
\mathit{size}\langle \Lambda a \to \mathit{Set}\ (f\ a)\rangle & = & \ldots\ (\mathit{size}\langle f\rangle)\ \ldots
\end{array}
$$

- The approach has been used to specify Haskell's derivable methods: $(==)$, $\mathit{compare}$, show, read, etc.