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`
Homepage: `http://www.informatik.uni-bonn.de/~ralf`

January, 2000

(Pick the slides at `.../~ralf/talks.html#T10`.)

A brief outline of the talk

We propose a new framework for generic functional programming (GFP), which allows you to define functions generically for all datatypes definable in *Haskell*.

- ☐ Why GFP is useful.
- ☐ Why GFP might be difficult.
- ☐ GFP is simple.
- ☐ Bridging the gap.

Polymorphic type systems

Many of us (most of us?) will probably 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 $length$).

But, . . .

Polymorphic type systems are sometimes less flexible than one would wish.

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

$$\boxed{equal \quad :: \quad \forall a. a \to a \to Bool}$$

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

GFP is your friend

Generic functional programming addresses this problem.

☞ *Basic idea:* define the equality function by *induction on the structure of types*.

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

Generic programs are ubiquitous

- Equality and comparison functions.
- 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

- $\surd$ Why GFP is useful.
- $\square$ Why GFP might be difficult.
- $\square$ GFP is simple.
- $\square$ Bridging the gap.

The challenge of Haskell's type system

In Haskell new types are introduced via $\mathbf{data}$ declarations. Here are the datatypes of lists and rose trees (multiway trees).

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

$List$ and $Rose$ are *type constructors* while Nil, $Cons$ and $Node$ are *value constructors*.

☞ Both $List$ and $Rose$ are parameterized types. They have kind $\star \to \star$.

The challenge of Haskell's type system (continued)

$$
\begin{array}{lcl}
\mathbf{type}\ \mathit{Square}\ a & = & \mathit{Sq}\ \mathit{NilT}\ a \\
\mathbf{data}\ \mathit{Sq}\ f\ a & = & \mathit{Zero}\ (f\ (f\ a)) \mid \mathit{Succ}\ (\mathit{Sq}\ (\mathit{ConsT}\ f)\ a) \\
\mathbf{data}\ \mathit{NilT}\ a & = & \mathit{Nil} \\
\mathbf{data}\ \mathit{ConsT}\ f\ a & = & \mathit{Cons}\ a\ (f\ a)
\end{array}
$$

The type Square implements $n \times n$ *square matrices*. **NB.** The squareness constraint is enforced by the type system.

☞ Type arguments can range over type constructors: Sq and ConsT have kind $(\star \to \star) \to (\star \to \star)$.

Haskell's $\mathbf{data}$ declaration

In general, a $\mathbf{data}$ declaration has the form:

$$\boxed{\mathbf{data}\ T\ x_1\ \dots\ x_m \quad = \quad C_1\ t_{11}\ \dots\ t_{1m_1} \mid \cdots \mid C_n\ t_{n1}\ \dots\ t_{nm_n}\ .}$$

The $\mathbf{data}$ construct combines no less than four different features: type abstraction, type recursion, n-ary sums, and n-ary products.

The types on the right-hand side are built from type constants, type variables, and type application.

☞ Haskell's type system corresponds to the *simply typed λ-calculus* (with kinds playing the rôle of types).

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 $C = \{1, Int, (+), (\times)\}$ be a fixed set of primitive type constructors.

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

Translation of the **data** declaration:

$$\mu(\Lambda T\ x_1\ \ldots\ x_m \to t_{11} \times \cdots \times t_{1m_1} + \cdots + t_{n1} \times \cdots \times t_{nm_n})\ .$$

A brief outline of the talk

- $\surd$ Why GFP is useful.
- $\surd$ Why GFP might be difficult.
- $\square$ GFP is simple.
- $\square$ Bridging the gap.

Normal form of types

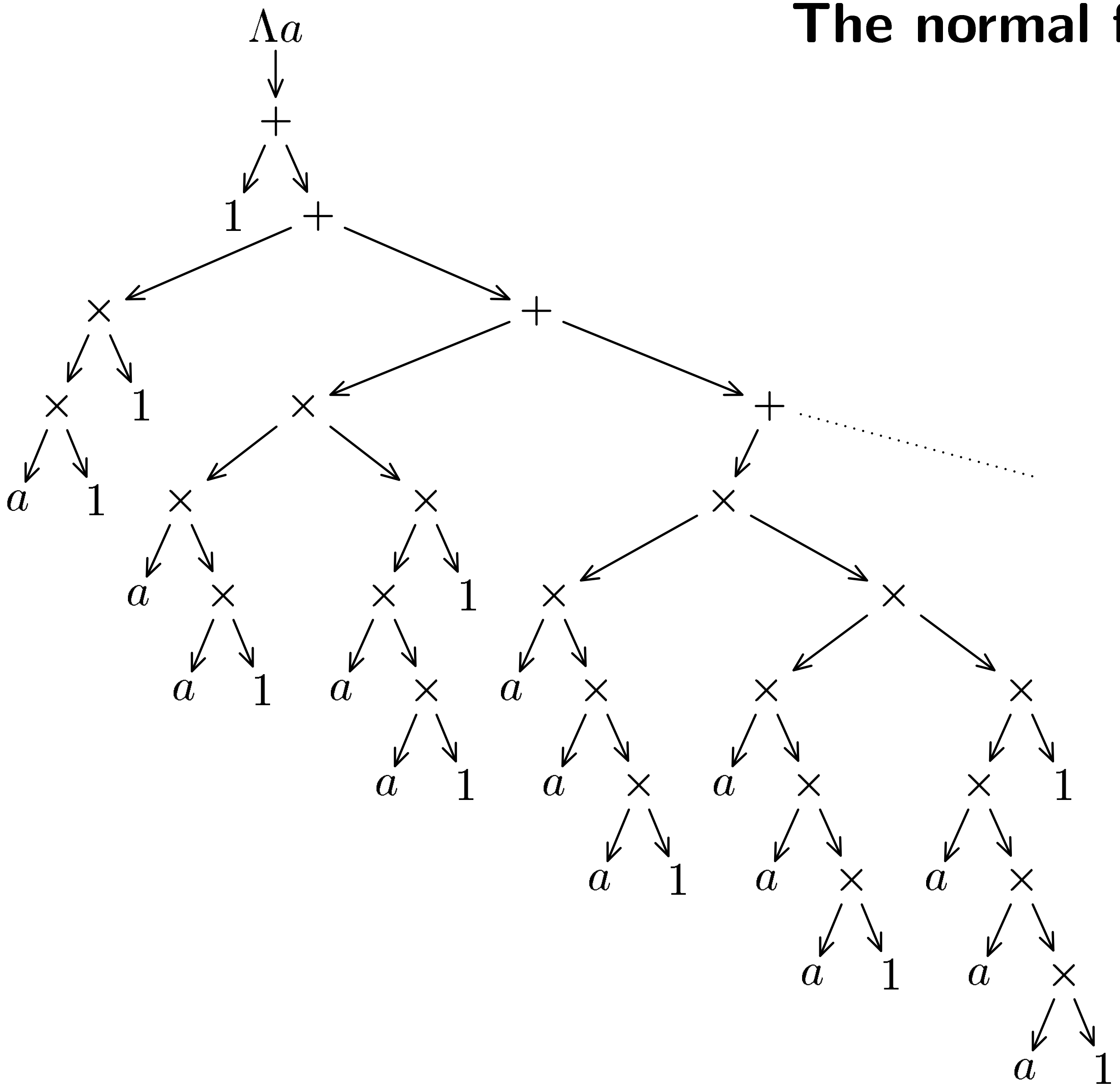
Recall the basic idea: a generic value is defined by induction on the structure of types.

The good news: it is sufficient to consider types in normal form.

$$T_\star \quad ::= \quad 1 \mid Int \mid (T_\star + T_\star) \mid (T_\star \times T_\star)$$

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

The normal form of $Square$



Example: generic data compression

The generic function $enc\langle a\rangle$ encodes elements of type a as bit lists.

$$
\begin{array}{lcl}
\textbf{type}\ Bin & = & [\,Bit\,] \\
\textbf{data}\ Bit & = & \mathsf{0} \mid \mathsf{1}
\end{array}
$$

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

NB. $[\,a\,]$ is Haskell syntax for $List\ a$, $[\,]$ for Nil, $b : bs$ for $Cons\ b\ bs$, and $(+\!\!+)$ denotes concatenation.

Example: generic equality

$$
\begin{array}{lcl}
\mathit{equal}\langle a\rangle & :: & a \to a \to \mathit{Bool} \\
\mathit{equal}\langle 1\rangle\ x\ y & = & \mathit{True} \\
\mathit{equal}\langle \mathit{Int}\rangle\ m\ n & = & \mathit{eqInt}\ m\ n \\
\mathit{equal}\langle a_1 + a_2\rangle\ (\mathit{Inl}\ x_1)\ (\mathit{Inl}\ y_1) & = & \mathit{equal}\langle a_1\rangle\ x_1\ y_1 \\
\mathit{equal}\langle a_1 + a_2\rangle\ (\mathit{Inl}\ x_1)\ (\mathit{Inr}\ y_2) & = & \mathit{False} \\
\mathit{equal}\langle a_1 + a_2\rangle\ (\mathit{Inr}\ x_2)\ (\mathit{Inl}\ y_1) & = & \mathit{False} \\
\mathit{equal}\langle a_1 + a_2\rangle\ (\mathit{Inr}\ x_2)\ (\mathit{Inr}\ y_2) & = & \mathit{equal}\langle a_2\rangle\ x_2\ y_2 \\
\mathit{equal}\langle a_1 \times a_2\rangle\ (x_1, x_2)\ (y_1, y_2) & = & \mathit{equal}\langle a_1\rangle\ x_1\ y_1 \\
 & & \quad \wedge\ \mathit{equal}\langle a_2\rangle\ x_2\ y_2
\end{array}
$$

Defining generic values

In general, a generic definition has the form:

$$\begin{array}{lcl} poly\langle a\rangle & :: & Poly\langle a\rangle \\ poly\langle 1\rangle & = & poly_{1} \\ poly\langle Int\rangle & = & poly_{Int} \\ poly\langle a_1 + a_2\rangle & = & poly_{+}\ (poly\langle a_1\rangle)\ (poly\langle a_2\rangle) \\ poly\langle a_1 \times a_2\rangle & = & poly_{\times}\ (poly\langle a_1\rangle)\ (poly\langle a_2\rangle)\ . \end{array}$$

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

☞ $poly$ can be uniquely extended to a continuous function on Böhm trees (setting $poly\langle 0\rangle = \bot$).

A brief outline of the talk

$\surd$ Why GFP is useful.

$\surd$ Why GFP might be difficult.

$\surd$ GFP is simple.

$\square$ Bridging the gap.

Specializing generic values

How can we specialize $enc\langle a\rangle$ or $equal\langle a\rangle$ for given instances of a?

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

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

☞ *Basic 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 $enc\langle List\ a\rangle$.

Specializing generic values (continued)

The specialization of $poly$ to types of arbitrary kinds is given by

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

Type application is mapped to value application, type abstraction to value abstraction, and type recursion to value recursion.

☞ Specialization amounts to an *interpretation* of the simply typed λ-calculus.

Specializing generic values (continued)

The type of $poly\langle\!\langle t :: \kappa \rangle\!\rangle\ \varrho$ is given by

$$poly\langle\!\langle t :: \kappa \rangle\!\rangle\ \varrho \quad :: \quad Poly\langle\!\langle t :: \kappa \rangle\!\rangle \ ,$$

where $Poly\langle\!\langle t :: \kappa \rangle\!\rangle$ is defined by induction on the structure of kinds.

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

Example: specializing enc

$$
\begin{array}{lll}
encList & :: & \forall a.(a \to Bin) \to (List\ a \to Bin) \\
encRose & :: & \forall a.(a \to Bin) \to (Rose\ a \to Bin)
\end{array}
$$

$$
\begin{array}{lll}
encSquare & :: & \forall a.(a \to Bin) \to (Square\ a \to Bin) \\
encSq & :: & \forall f.(\forall b.(b \to Bin) \to (f\ b \to Bin)) \\
 & & \qquad \to (\forall a.(a \to Bin) \to (Sq\ f\ a \to Bin)) \\
encNilT & :: & \forall a.(a \to Bin) \to (NilT\ a \to Bin) \\
encConsT & :: & \forall f.(\forall b.(b \to Bin) \to (f\ b \to Bin)) \\
 & & \qquad \to (\forall a.(a \to Bin) \to (ConsT\ f\ a \to Bin))
\end{array}
$$

☞ $encSq$ and $encConsT$ have rank-2 type signatures.

Example: specializing enc (continued)

$$
\begin{array}{lcl}
encList\ enca\ Nil & = & [\mathtt{0}] \\
encList\ enca\ (Cons\ x\ xs) & = & \mathtt{1} : enca\ x +\!\!+\ encList\ enca\ xs \\
encRose\ enca\ (Node\ x\ ts) & = & enca\ x +\!\!+\ encList\ (encRose\ enca)\ ts
\end{array}
$$

$$
\begin{array}{lcl}
encSquare\ enca\ m & = & encSq\ encNil\ enca\ m \\
encSq\ encf\ enca\ (Zero\ m) & = & \mathtt{0} : encf\ (encf\ enca)\ m \\
encSq\ encf\ enca\ (Succ\ m) & = & \mathtt{1} : encSq\ (encConsT\ encf)\ enca\ m \\
encNilT\ enca\ Nil & = & [\,] \\
\multicolumn{3}{l}{encConsT\ encf\ enca\ (Cons\ x\ xs)} \\
 & = & enca\ x +\!\!+\ encf\ enca\ xs
\end{array}
$$

A brief outline of the talk

$\surd$ Why GFP is useful.

$\surd$ Why GFP might be difficult.

$\surd$ GFP is simple.

$\surd$ Bridging the gap.

Summary

Three things to remember:

- □ Haskell's type system can be modelled by the simply typed λ-calculus.
- □ Defining a generic value works by giving cases for primitive types.
- □ Specializing a generic value amounts to an interpretation of the simply typed λ-calculus.

The paper furthermore explains how to generalize the approach to

- □ values indexed by type constructors of first-order kind (such as $\star \to \star$).

Current and future work

- Generic reasoning using logical relations on the type level.
- It remains to implement the framework.

Appendix

We treat 1, '$+$', and '$\times$' as if they were given by the following datatype declarations.

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