"Scrap Your Boilerplate" Revolutions

RALF HINZE

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

July, 2006

Joint work with Andres Löh

(Pick up the slides at .../~ralf/talks.html#50.)

1 Introduction

Many of us (most of us?) will probably agree that type systems, especially, polymorphic type systems are a good thing.

A type system is like a suit of armour:

- it shields against the modern dangers of illegal instructions and memory violations, but
- it also restricts flexibility.

In Haskell 98, for instance, it is not possible to define an equality test that works for all types.

☞ Equality, comparison functions, pretty printers (Haskell's $show$), parsers (Haskell's $read$) have to become known as data-generic or polytypic functions.

☞ Broadly speaking, generic programming is about defining functions that work for all types but that also exhibit type-specific behaviour.

Support for generic programming consists of three essential ingredients:

- a type reflection mechanism,
- a type representation, and
- a generic view on data.

☞ Using type reflection we can program functions that exhibit type-specific behaviour, so-called overloaded functions. Using a generic view on data we can generalise overloaded functions to generic ones. The type representation determines the generic functions we can write: $*$-indexed functions such as '==', $read$; $* \longrightarrow *$-indexed functions such as map, $size$ etc.

For each dimension there are several choices: for instance, to reflect types we can use a type representation type, type classes or a type-safe cast.

☞ The purpose of this talk is to investigate the third dimension: the generic view on data.

- Classicism (1990 –): strong background in category theory.

  PolyP: a data type is viewed as a fixed point of a regular functor; the base functor is viewed as a lifted sum of products.

- Romanticism (1995 –): shift towards type-theoretic approaches.

  Generic Haskell: a data type is viewed as a sum of products.

- Realism (2000 –): compiler and library hacking.

  SYB: combinator-based, the user writes generic functions by combining a few generic primitives.

☞ SYB did not seem to fit into the picture, as it lacked the generic view. Also, it wasn't clear, whether SYB could express all the generic functions.

2. The spine view

To reflect types onto the value level, we introduce a type representation type.

$$\begin{aligned}
&\mathbf{data}\ \mathit{Type} :: * \to * \\
&\mathit{Char} :: \mathit{Type}\ \mathit{Char} \\
&\mathit{Int} \quad :: \mathit{Type}\ \mathit{Int} \\
&\mathit{Pair} :: \mathit{Type}\ \alpha \to \mathit{Type}\ \beta \to \mathit{Type}\ (\alpha, \beta) \\
&\mathit{List} :: \mathit{Type}\ \alpha \to \mathit{Type}\ [\alpha]
\end{aligned}$$

Each type has a unique representation: Int is represented by the constructor Int, $(\mathit{String}, \mathit{Int})$ is represented by $\mathit{Pair}\ (\mathit{List}\ \mathit{Char})\ \mathit{Int}$.

We shall often need to annotate an expression with its type representation.

$$\mathbf{data}\ \mathit{Typed}\ \alpha = (:)\{\mathit{val} :: \alpha, \mathit{type} :: \mathit{Type}\ \alpha\}$$

The definition, which makes use of Haskell's record syntax, introduces the colon ':' as an infix data constructor: $4711 : \mathit{Int}$ is an element of $\mathit{Typed}\ \mathit{Int}$.

Using the type of type representation we can define overloaded functions that exhibit type-specific behaviour.

Example: collecting integers.

$$
\begin{array}{ll}
ints & :: Typed\ \alpha \rightarrow [Int] \\
ints\ (c : Char) & = [\,] \\
ints\ (i : Int) & = [i] \\
ints\ ((x, y) : Pair\ a\ b) & = ints\ (x : a) +\!\!+ ints\ (y : b) \\
ints\ (xs : List\ a) & = concat\ [ints\ (x : a) \mid x \leftarrow xs]
\end{array}
$$

In order to define a generic function, we need to find a way to treat elements of a data type in a uniform way.

Consider an arbitrary element of some data type:

$$C\ e_1\ \cdots\ e_n$$

The idea is to make this applicative structure visible and accessible: we mark the constructor using Con and each function application using '$\lozenge$'.

- $Empty$ becomes $Con\ empty$,
- $Node\ l\ a\ r$ becomes $Con\ node \lozenge (l : Tree\ Int) \lozenge (a : Int) \lozenge (r : Tree\ Int)$.

☞ The arguments are additionally annotated with their types and the constructor itself with information on its syntax.

The functions *Con* and '◊' are constructors of a data type called *Spine*.

$$\begin{array}{l}\textbf{data}\ \mathit{Spine} :: * \to * \ \textbf{where}\\ \quad \mathit{Con} :: \mathit{Constr}\ \alpha \to \mathit{Spine}\ \alpha\\ \quad (\lozenge) \quad :: \mathit{Spine}\ (\alpha \to \beta) \to \mathit{Typed}\ \alpha \to \mathit{Spine}\ \beta\end{array}$$

The type is called *Spine* because its elements represent the possibly partial spine of a constructor application.

Elements of type $\mathit{Constr}\ \alpha$ comprise an element of type α, namely the original data constructor, plus additional information about its syntax.

$$\textbf{data}\ \mathit{Constr}\ \alpha = \mathit{Descr}\{\, \mathit{constr} :: \alpha, \mathit{name} :: \mathit{String}\,\}$$

In order to use the $Spine$ data type as a generic view, we need for each data type functions that convert to and fro.

Given a value of type $Spine\ \alpha$, we can easily recover the original value of type α:

$$\begin{array}{ll} fromSpine & :: Spine\ \alpha \to \alpha \\ fromSpine\ (Con\ c) & = constr\ c \\ fromSpine\ (f \lozenge x) & = (fromSpine\ f)\ (val\ x) \end{array}$$

☞ *fromSpine* is parametrically polymorphic — one size fits all.

The inverse of $fromSpine$ is an overloaded function of type $Typed\ \alpha \longrightarrow Spine\ \alpha$.

Its definition, however, follows a trivial pattern: if the data type comprises a constructor C

$$C :: \tau_1 \longrightarrow \cdots \longrightarrow \tau_n \longrightarrow \tau_0$$

then the equation for $toSpine$ takes the form

$$toSpine\ (C\ x_1\ \ldots\ x_n : t_0) = Con\ c \Diamond (x_1 : t_1) \Diamond \cdots \Diamond (x_n : t_n)$$

where c is the annotated version of C and t_i is the type representation of τ_i.

As an example, here is the definition of $toSpine$ for binary trees.

$$
\begin{aligned}
&\textbf{data}\ Tree\ \alpha = Empty \mid Node\ (Tree\ \alpha)\ \alpha\ (Tree\ \alpha)\\
&\\
&toSpine :: Typed\ \alpha \to Spine\ \alpha\\
&toSpine\ (Empty : Tree\ a) = Con\ empty\\
&toSpine\ (Node\ l\ x\ r : Tree\ a)\\
&\quad = Con\ node \diamond (l : Tree\ a) \diamond (x : a) \diamond (r : Tree\ a)\\
&\\
&empty :: Constr\ (Tree\ \alpha)\\
&empty = Descr\{\, constr = Empty, name = \texttt{"Empty"}\,\}\\
&node\ \ :: Constr\ (Tree\ \alpha \to \alpha \to Tree\ \alpha \to Tree\ \alpha)\\
&node\ \ = Descr\{\, constr = Node,\ \ name = \texttt{"Node"}\,\}
\end{aligned}
$$

With all the machinery in place we can now turn *ints* into a truly generic function.

☞ The idea is to add a catch-all case that takes care of all the remaining type cases in a uniform manner.

$$
\begin{array}{lll}
ints & :: & Typed\ \alpha \to [Int] \\
ints\ (i : Int) & = & [i] \\
ints\ x & = & intsSpine\ (toSpine\ x)
\end{array}
$$

$$
\begin{array}{lll}
intsSpine & :: & Spine\ \alpha \to [Int] \\
intsSpine\ (Con\ c) & = & [\,] \\
intsSpine\ (f \diamond x) & = & intsSpine\ f +\!\!+ ints\ x
\end{array}
$$

- The spine view is easy to use: the generic part of a generic function only has to consider two cases: Con and '$\Diamond$'.
- A further advantage of the spine view is its generality: it is applicable to a large class of data types including generalized algebraic data types.
- On the other hand, the spine view restricts the class of functions we can write:
  - one can only define generic functions that consume or transform data (such as $show$) but not ones that produce data (such as $read$);
  - functions that abstract over type constructors (such as map or $size$) are out of reach.

3 The type spine view

In order to define generic producers, we require additional information about the data type, information that the spine view does not provide.

Consider the syntactic form of a generalized algebraic data type: a data type is essentially a sequence of signatures. This motivates the following definitions.

$$\begin{array}{l}\textbf{type}\ \mathit{Datatype}\ \alpha = [\mathit{Signature}\ \alpha] \\ \\ \textbf{data}\ \mathit{Signature} :: * \to * \to * \ \textbf{where} \\ \quad \mathit{Sig} :: \mathit{Constr}\ \alpha \to \mathit{Signature}\ \alpha \\ \quad (\Box) :: \mathit{Signature}\ (\alpha \to \beta) \to \mathit{Type}\ \alpha \to \mathit{Signature}\ \beta\end{array}$$

☞ The type *Signature* is almost identical to the *Spine* type, except for the second argument of '□', which is of type $\mathit{Type}\ \alpha$ rather than $\mathit{Typed}\ \alpha$.

To be able to use the type spine view, we require an overloaded function that maps a type representation to an element of type $Datatype\ \alpha$.

$$
\begin{array}{ll}
datatype & :: Type\ \alpha \to Datatype\ \alpha \\
datatype\ (Bool) & = [Sig\ false, Sig\ true] \\
datatype\ (Char) & = [Sig\ (char\ c) \mid c \leftarrow [minBound\,..\,maxBound]] \\
datatype\ (Int) & = [Sig\ (int\ i) \mid i \leftarrow [minBound\,..\,maxBound]] \\
datatype\ (List\ a) & = [Sig\ nil, Sig\ cons \,\square\, a \,\square\, List\ a] \\
datatype\ (Pair\ a\ b) & = [Sig\ pair \,\square\, a \,\square\, b] \\
datatype\ (Tree\ a) & = [Sig\ empty, Sig\ node \,\square\, Tree\ a \,\square\, a \,\square\, Tree\ a]
\end{array}
$$

☞ *datatype* plays the same role for producers as *toSpine* plays for consumers.

Here is an example of a generic producer: a test-data generator.

$$
\begin{array}{lll}
generate & :: & Type\ \alpha \to Int \to [\alpha] \\
generate\ a\ 0 & = & [\,] \\
generate\ a\ (d+1) & = & concat\ [generateSig\ s\ d \mid s \leftarrow datatype\ a]
\end{array}
$$

$$
\begin{array}{lll}
generateSig & :: & Signature\ \alpha \to Int \to [\alpha] \\
generateSig\ (Sig\ c)\ d & = & [constr\ c] \\
generateSig\ (s \square a)\ d & = & [f\ x \mid f \leftarrow generateSig\ s\ d, x \leftarrow generate\ a\ d]
\end{array}
$$

The helper function *generateSig* constructs all terms that conform to a given signature.

- The type spine view is complementary to the spine view, but independent of it. The latter is used for generic producers, the former for generic consumers or transformers.
- The type spine view shares the major advantage of the spine view: it is applicable to a large class of data types including generalized algebraic data types.

4 Lifted spine view

To represent container types of kind $* \longrightarrow *$, we lift the type constructors and include Id as a representation of the type variable α:

$$\begin{array}{l}\textbf{data}\ Type' :: (* \longrightarrow *) \longrightarrow * \\ \\ Id \quad :: Type'\ Id \\ Char' :: Type'\ Char' \\ Int' \quad :: Type'\ Int' \\ List' \ :: Type'\ \varphi \longrightarrow Type'\ (List'\ \varphi)\end{array}$$

☞ $List'$ takes a type of kind $* \longrightarrow *$ to a type of kind $* \longrightarrow *$. The container type $\Lambda\alpha.[[\alpha]]$ is represented by $List'\ (List'\ Id)$.

$$\textbf{data}\ Typed'\ \varphi\ \alpha = (:')\{\, val' :: \varphi\ \alpha, type' :: Type'\ \varphi\,\}$$

Using the type $Type'$ of type representations we can define overloaded functions that abstract over type constructors of kind $* \longrightarrow *$.

Example: the size of a container.

$$
\begin{array}{ll}
size & :: Typed'\ \varphi\ \alpha \longrightarrow Int \\
size\ (x :'\ Id) & = 1 \\
size\ (c :'\ Char') & = 0 \\
size\ (i :'\ Int') & = 0 \\
size\ (Nil' :'\ List'\ a') & = 0 \\
size\ (Cons'\ x\ xs :'\ List'\ a') & = size\ (x :'\ a') + size\ (xs :'\ List'\ a')
\end{array}
$$

☞ Nil' etc are the constructors of the lifted types.

The lifted **data** definitions follow a simple scheme: each data constructor C

$$C :: \tau_1 \to \cdots \to \tau_n \to \tau_0$$

is replaced by a polymorphic data constructor C'

$$C' :: \forall\chi.\tau_1'\ \chi \to \cdots \to \tau_n'\ \chi \to \tau_0'\ \chi$$

where τ_i' is the lifted variant of τ_i.

We can write the signature more perspicuously as

$$C' :: \forall\chi.(\tau_1' \to' \cdots \to' \tau_n' \to' \tau_0')\ \chi$$

using the lifted function space:

$$\mathbf{newtype}\ (\varphi \to' \psi)\ \chi = \mathit{Fun}\{\, \mathit{app} :: \varphi\ \chi \to \psi\ \chi\}$$

☞ An element of a lifted type can always be put into the applicative form

$$c' \; `app` \; e_1 \; `app` \cdots `app` \; e_n$$

As in the first-order case we can make this structure visible and accessible by marking the constructor and the function applications.

$$\begin{array}{l} \mathbf{data}\ \mathit{Spine}' :: (* \to *) \to * \to * \ \mathbf{where} \\ \quad \mathit{Con}' :: (\forall \chi.\varphi\ \chi) \to \mathit{Spine}'\ \varphi\ \alpha \\ \quad (\Diamond') \quad :: \mathit{Spine}'\ (\varphi \to' \psi)\ \alpha \to \mathit{Typed}'\ \varphi\ \alpha \to \mathit{Spine}'\ \psi\ \alpha \end{array}$$

☞ The structure of Spine' is very similar to that of Spine except that we are now working in a higher realm: Con' takes a polymorphic function of type $\forall \chi.\varphi\ \chi$ to an element of $\mathit{Spine}'\ \varphi$.

Turning to the conversion functions, $fromSpine'$ is again polymorphic.

$$
\begin{array}{ll}
fromSpine' & :: Spine'\ \varphi\ \alpha \to \varphi\ \alpha \\
fromSpine'\ (Con'\ c) & = c \\
fromSpine'\ (f \mathbin{\Diamond'} x) & = fromSpine'\ f\ \text{`}app\text{`}\ val'\ x
\end{array}
$$

Its inverse is an overloaded function that follows a similar pattern as $toSpine$: each constructor C'

$$C' :: \forall\chi.\tau'_1\ \chi \to \cdots \to \tau'_n\ \chi \to \tau'_0\ \chi$$

gives rise to an equation of the form

$$toSpine'\ (C'\ x_1\ \ldots\ x_n :'\ t'_0) = Con\ c'\ \Diamond\ (x_1 : t'_1)\ \Diamond \cdots \Diamond\ (x_n : t'_n)$$

where c' is the variant of C' that uses the lifted function space and t'_i is the type representation of the lifted type τ'_i.

As an example, here is the instance for lifted lists.

$$\begin{array}{ll} toSpine' & :: Typed'\ \varphi\ \alpha \to Spine'\ \varphi\ \alpha \\ toSpine'\ (Nil' :'\ List'\ a') & = Con'\ nil' \\ toSpine'\ (Cons'\ x\ xs :'\ List'\ a') & = Con'\ cons'\ \Diamond'\ (x :'\ a')\ \Diamond'\ (xs :'\ List'\ a') \end{array}$$

Given these prerequisites we can turn *size* into a truely generic function.

$$
\begin{aligned}
&size && :: Typed'\ \varphi\ \alpha \to Int \\
&size\ (x :'\ Id) && = 1 \\
&size\ (x :'\ a') && = sizeSpine\ (toSpine'\ (x :'\ a'))
\end{aligned}
$$

The implementation of *sizeSpine* is entirely straightforward: it traverses the spine summing up the sizes of the constructors arguments.

$$
\begin{aligned}
&sizeSpine && :: Spine'\ \varphi\ \alpha \to Int \\
&sizeSpine\ (Con'\ c) && = 0 \\
&sizeSpine\ (f \diamond'\ x) && = sizeSpine\ f + size\ x
\end{aligned}
$$

- The lifted spine view is almost as general as the original spine view: it is applicable to all data types that are definable in Haskell 98.
- The lifted spine view is not applicable to generalised algebraic data types, as it is not possible to generalise $size$ to GADTs.
- For generic producers we need a lifted spine view.
- The spine view can even be lifted to kind indices of arbitrary kinds.

  The generic programmer then has to consider two cases for the spine view and additionally n cases, one for each of the n projection types $Out_1, \ldots, Out_n$.

5. Conclusion

The original SYB approach is combinator-based: the user writes generic functions by combining a few generic primitives such as *gfoldl* and *gunfold*.

- *gfoldl* is essentially the catamorphism of the *Spine* data type: *gfoldl* equals the catamorphism composed with *toSpine*.
- *gunfold* is the catamorphism of the *Signature* data type.

view(s)	representation of overloaded functions			
	type reflection	type classes	type-safe cast	specialisation
none	ITA	–	–	–
fixed point	Reloaded	PolyP	–	PolyP
sum-of-products	LIGD	DTC, GC, GM	–	GH
spine	Reloaded, Revolutions	SYB, Reloaded	SYB	–