Embedding Generic Programming into Haskell

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#T51.)

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.

- In this talk, we show how to embed generic programming into Haskell.
- The embedding builds upon recent advances in type theory: generalised algebraic data types and open data types.
- Put differently, we propose and employ language features that are useful for generic programming.
- We will identify the basic building blocks of generic programming and we will provide an overview of the overall design space.
- We hope to convince you that generic programming is useful and that you can use generic programming techniques today!

2. Preliminaries

- 2.1 Values, types and kinds
- 2.2 Generalised algebraic data types
- 2.3 Open data types and open functions

2.1 Values, types and kinds

Haskell has the three level structure depicted below.

kinds:	$*,\ * \longrightarrow *$
types:	$Bool,\ List\ \alpha,\ \forall\alpha.\alpha \longrightarrow \alpha$
values:	$False,\ Nil,\ \lambda f\ x \longrightarrow f\ (f\ x)$

- The lowest level — where computations take place — consists of values.
- The second level, which imposes structure on the value level, is inhabited by types.
- On the third level, which imposes structure on the type level, we have so-called kinds. ☞ A kind is simply the 'type' of a type constructor.

In Haskell, new data types are declared using the **data** construct.

$$\begin{aligned}
&\textbf{data } \mathit{Bool} &&= \mathit{False} \mid \mathit{True} \\
&\textbf{data } [\alpha] &&= \mathit{Nil} \mid \mathit{Cons}\ \alpha\ [\alpha] \\
&\textbf{data } \mathit{Pair}\ \alpha\ \beta &&= (\alpha, \beta)
\end{aligned}$$

☞ Data constructors are written in *blue*; type constructors in *red*.

A data type declaration of the schematic form

$$\textbf{data}\ T\ \alpha_1\ \ldots\ \alpha_s = C_1\ \tau_{1,1}\ \ldots\ \tau_{1,m_1} \mid \cdots \mid C_n\ \tau_{n,1}\ \ldots\ \tau_{n,m_n}$$

introduces data constructors $C_1, \ldots, C_n$ with signatures

$$C_i :: \forall \alpha_1\ \ldots\ \alpha_s . \tau_{i,1} \to \cdots \to \tau_{i,m_i} \to T\ \alpha_1\ \ldots\ \alpha_s$$

☞ The **data** construct is a beast; it combines no less than four features: type abstraction, n-ary disjoint sums, n-ary cartesian products and type recursion.

2.2 Generalised algebraic data types

Using a recent version of GHC, there is an alternative way of defining data types:

$$\begin{array}{l}\mathbf{data}\ [\,] :: * \to * \ \mathbf{where} \\ \quad \mathit{Nil} \quad :: \forall\alpha.[\alpha] \\ \quad \mathit{Cons} :: \forall\alpha.\alpha \to [\alpha] \to [\alpha]\end{array}$$

The first line declares the kind of the new data type.

The type is then inhabited by listing the signatures of the data constructors.

Generalised algebraic data types (GADTs) lift the Haskell 98 restriction that the result type of the constructors must be of the form $T\ \alpha_1\ \ldots\ \alpha_s$.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} :: * \to * \ \textbf{where} \\
\quad \mathit{Num} :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\quad \mathit{Plus} :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\quad \mathit{Eq} :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool} \\
\quad \mathit{If} :: \forall \alpha.\mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ \alpha \to \mathit{Expr}\ \alpha \to \mathit{Expr}\ \alpha
\end{array}
$$

☞ The data type Expr represents typed expressions.

An evaluator for the $Expr$ data type:

$$
\begin{array}{l}
eval :: Expr\ \alpha \rightarrow \alpha \\
eval\ (Num\ i) \quad\quad = i \\
eval\ (Plus\ e_1\ e_2) = eval\ e_1 + eval\ e_2 \\
eval\ (Eq\ e_1\ e_2) \quad = eval\ e_1 == eval\ e_2 \\
eval\ (If\ e_1\ e_2\ e_3) = \textbf{if}\ eval\ e_1\ \textbf{then}\ eval\ e_2\ \textbf{else}\ eval\ e_3
\end{array}
$$

☞ For functions on GADTs, type signatures are mandatory.

Each equation has a more specific type: the first equation has type $Expr\ Int \rightarrow Int$ as Num constrains α to Int.

The interpreter is quite noticeable in that it is tag free.

2.3 Open data types and open functions

For embedding generic function into Haskell, we make use of open data types and open functions, data types and functions that can be freely extended.

An open data type is declared as follows:

$$\textbf{open data}\ \ \mathit{Expr} :: * \longrightarrow *$$

Constructors can then be introduced just by providing their type signatures, at any point in the program.

$$\begin{array}{ll} \mathit{Str} & :: \mathit{String} \longrightarrow \mathit{Expr}\ \mathit{String} \\ \mathit{Show} & :: \mathit{Expr}\ \mathit{Int} \longrightarrow \mathit{Expr}\ \mathit{String} \\ \mathit{Cat} & :: \mathit{Expr}\ \mathit{String} \longrightarrow \mathit{Expr}\ \mathit{String} \longrightarrow \mathit{Expr}\ \mathit{String} \end{array}$$

An open function is declared as follows:

$$\textbf{open}\ \mathit{eval} :: \mathit{Expr}\ \alpha \to \alpha$$

The definition of an open function needs not be contiguous; the defining equations may be scattered around the program.

$$\begin{aligned}
&\mathit{eval}\ (\mathit{Str}\ s) &&= s\\
&\mathit{eval}\ (\mathit{Show}\ e) &&= \mathit{show}_{\mathit{Int}}\ (\mathit{eval}\ e)\\
&\mathit{eval}\ (\mathit{Cat}\ e_1\ e_2) &&= \mathit{eval}\ e_1 +\!\!+ \mathit{eval}\ e_2
\end{aligned}$$

☞ The semantics of open data types and open functions is the same as if data types and functions had been defined closed, in a single place.

Open data types and open functions provide two dimensions of extensibility:

- we can add additional sorts of data, by providing new constructors,
- we can add additional operations, by defining new functions:

$$
\begin{aligned}
&\textbf{open } \mathit{string} :: \mathit{Expr}\ \alpha \to \mathit{String} \\
&\mathit{string}\ (\mathit{Num}\ i) &&= \texttt{"(Num"}\ \diamond \mathit{show}_{\mathit{Int}}\ i +\!\!+ \texttt{")"} \\
&\mathit{string}\ (\mathit{Plus}\ e_1\ e_2) &&= \texttt{"(Plus"}\ \diamond \mathit{string}\ e_1 \diamond \mathit{string}\ e_2 +\!\!+ \texttt{")"} \\
&\mathit{string}\ (\mathit{Eq}\ e_1\ e_2) &&= \texttt{"(Eq"}\ \diamond \mathit{string}\ e_1 \diamond \mathit{string}\ e_2 +\!\!+ \texttt{")"} \\
&\mathit{string}\ (\mathit{If}\ e_1\ e_2\ e_3) &&= \texttt{"(If"}\ \diamond \mathit{string}\ e_1 \diamond \mathit{string}\ e_2 \diamond \mathit{string}\ e_3 +\!\!+ \texttt{")"} \\
&\mathit{string}\ (\mathit{Str}\ s) &&= \texttt{"(Str"}\ \diamond \mathit{show}_{\mathit{String}}\ s +\!\!+ \texttt{")"} \\
&\mathit{string}\ (\mathit{Show}\ e) &&= \texttt{"(Show"}\ \diamond \mathit{string}\ e +\!\!+ \texttt{")"} \\
&\mathit{string}\ (\mathit{Cat}\ e_1\ e_2) &&= \texttt{"(Cat"}\ \diamond \mathit{string}\ e_1 \diamond \mathit{string}\ e_2 +\!\!+ \texttt{")"}
\end{aligned}
$$

$$s_1 \diamond s_2 = s_1 +\!\!+ \texttt{" "} +\!\!+ s_2$$

For open functions, first-fit pattern matching is not suitable.

$$string\ \_ = \text{""}$$

Using first-fit pattern matching, this equation effectively closes the definition of $string$.

☞ Instead we use best-fit left-to-right pattern matching: the most specific match rather than the first match wins.

3. Overloaded functions

In Haskell, showing values of a data type is particularly easy:

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha)$$
$$\textbf{deriving}\ (\mathit{Show})$$

The compiler automatically generates a suitable $show$ function.

This function is used, for instance, in interactive sessions:

$$\mathit{Now}\rangle\ \ \mathit{tree}\ [0\,..\,3]$$
$$\mathit{Node}\ (\mathit{Node}\ (\mathit{Node}\ \mathit{Empty}\ 0\ \mathit{Empty})\ 1\ \mathit{Empty})\ 2\ (\mathit{Node}\ \mathit{Empty}\ 3\ \mathit{Empty})$$

Here $\mathit{tree} :: [\alpha] \to \mathit{Tree}\ \alpha$ transforms a list into a balanced tree.

The function $show$ can be seen as a pretty printer.

The display of larger structures, however, is not especially pretty, due to lack of indentation.

$$\begin{array}{l}Now\rangle\ \ tree\ [0\,..\,9]\\ Node\ (Node\ (Node\ (Node\ Empty\ 0\ Empty)\ 1\ Empty)\ 2\ (Node\ (Node\ Em\\ pty\ 3\ Empty)\ 4\ Empty))\ 5\ (Node\ (Node\ (Node\ Empty\ 6\ Empty)\ 7\ Empt\\ y)\ 8\ (Node\ Empty\ 9\ Empty))\end{array}$$

In the sequel we shall develop a replacement for $show$, a generic prettier printer.

We use a basic pretty printing library, which just offers support for indentation.

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Text} & \\
\mathit{text} & :: \mathit{String} \to \mathit{Text} \\
\mathit{nl} & :: \mathit{Text} \\
\mathit{indent} & :: \mathit{Int} \to \mathit{Text} \to \mathit{Text} \\
(\diamondsuit) & :: \mathit{Text} \to \mathit{Text} \to \mathit{Text}
\end{array}
$$

- Text is type of documents with indentation.
- text converts a string to a text.
- The string passed to text must not contain newline characters. The constant nl has to be used for that purpose.
- $\mathit{indent}\ i$ adds i spaces after each newline.
- '$\diamondsuit$' concatenates two pieces of text.

It is a simple exercise to write a prettier printer for trees of integers.

$$
\begin{aligned}
&pretty_{Int} :: Int \to Text \\
&pretty_{Int}\ n = text\ (show_{Int}\ n) \\
\\
&pretty_{TreeInt} :: Tree\ Int \to Text \\
&pretty_{TreeInt}\ Empty = text\ \texttt{"Empty"} \\
&pretty_{TreeInt}\ (Node\ l\ x\ r) = align\ \texttt{"(Node "}\ (pretty_{TreeInt}\ l \diamondsuit nl \diamondsuit \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad pretty_{Int}\ x \diamondsuit nl \diamondsuit \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad pretty_{TreeInt}\ r \diamondsuit text\ \texttt{")"}) \\
\\
&align :: String \to Text \to Text \\
&align\ s\ d = indent\ (length\ s)\ (text\ s \diamondsuit d)
\end{aligned}
$$

☞ While the program does the job, it is not very general: we can print trees of integers, but not, say, trees of characters.

Of course, it is easy to add another two ad-hoc definitions.

$$
\begin{aligned}
&pretty_{Char} :: Char \to Text \\
&pretty_{Char}\ c = text\ (show_{Char}\ c) \\
\\
&pretty_{Tree\,Char} :: Tree\ Char \to Text \\
&pretty_{Tree\,Char}\ Empty = text\ \texttt{"Empty"} \\
&pretty_{Tree\,Char}\ (Node\ l\ x\ r) = align\ \texttt{"(Node "}\ (pretty_{Tree\,Char}\ l \diamondsuit nl \diamondsuit \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad pretty_{Char}\ x \diamondsuit nl \diamondsuit \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad pretty_{Tree\,Char}\ r \diamondsuit text\ \texttt{")"})
\end{aligned}
$$

The code of $pretty_{Tree\,Char}$ is almost identical to that of $pretty_{Tree\,Int}$.

☞ We actually need a family of pretty printers. A typical case for type classes?

☞ In the sequel we explore a different route!

We could define a single function that receives the type as an additional argument and suitably dispatches on this type argument.

$$pretty :: (\alpha :: *) \longrightarrow \alpha \longrightarrow Text$$

☞ Haskell doesn't permit the explicit passing of types.

We could pass the pretty printer an additional argument that represents the type.

$$pretty :: Type \longrightarrow \alpha \longrightarrow Text$$

☞ This is too simple-minded: a function of this type must necessarily ignore its second parameter (parametricity, "free theorem").

$$pretty :: Type\ \alpha \longrightarrow \alpha \longrightarrow Text$$

An element of type $Type\ \tau$ is a representation of the type τ.

Using a generalised algebraic data type, we can define $Type$ directly in Haskell.

$$\begin{array}{l}\textbf{open data}\ \ Type :: * \to * \ \textbf{where}\\ \quad Char :: Type\ Char\\ \quad Int\ \ :: Type\ Int\\ \quad Pair :: Type\ \alpha \to Type\ \beta \to Type\ (\alpha, \beta)\\ \quad List\ \ :: Type\ \alpha \to Type\ [\alpha]\\ \quad Tree\ \ :: Type\ \alpha \to Type\ (Tree\ \alpha)\end{array}$$

Each type has a unique representation:

- the type Int is represented by the constructor Int,
- the type $(Char, Int)$ is represented by $Pair\ Char\ Int$,
- the type $[Tree\ Char]$ is represented by $List\ (Tree\ Char)$.

☞ Recall: type constructors are written in red; data constructors in $blue$.

☞ For any given τ the type $Type\ \tau$ comprises exactly one element: $Type\ \tau$ is a so-called singleton type.

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

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

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 $\mathit{Typed}\ \mathit{Int}$.
- $(47, \texttt{'R'}) : \mathit{Pair}\ \mathit{Int}\ \mathit{Char}$ is an element of $\mathit{Typed}\ (\mathit{Int}, \mathit{Char})$.

☞ Note the difference between $x : t$ and $x :: \tau$.

- $x : t$ is a pair consisting of a value x and a representation t of its type.
- $x :: \tau$ is Haskell syntax for 'x has type τ'.

Given these prerequisites, we can finally define the desired pretty printer.

$$
\begin{aligned}
&\textbf{open}\ \mathit{pretty} :: \mathit{Typed}\ \alpha \to \mathit{Text}\\
&\mathit{pretty}\ (c : \mathit{Char}) &&= \mathit{pretty}_{\mathit{Char}}\ c\\
&\mathit{pretty}\ (n : \mathit{Int}) &&= \mathit{pretty}_{\mathit{Int}}\ n\\
&\mathit{pretty}\ ((x, y) : \mathit{Pair}\ a\ b) &&= \mathit{align}\ \texttt{"(\ "}\ (\mathit{pretty}\ (x : a)) \diamondsuit \mathit{nl} \diamondsuit\\
&&&\quad \mathit{align}\ \texttt{",\ "}\ (\mathit{pretty}\ (y : b)) \diamondsuit \mathit{text}\ \texttt{")"}\\
&\mathit{pretty}\ (\mathit{xs} : \mathit{List}\ a) &&= \mathit{bracketed}\ [\mathit{pretty}\ (x : a) \mid x \leftarrow \mathit{xs}]\\
&\mathit{pretty}\ (\mathit{Empty} : \mathit{Tree}\ a) &&= \mathit{text}\ \texttt{"Empty"}\\
&\mathit{pretty}\ (\mathit{Node}\ l\ x\ r : \mathit{Tree}\ a)\\
&\quad = \mathit{align}\ \texttt{"(Node\ "}\ (\mathit{pretty}\ (l : \mathit{Tree}\ a) \diamondsuit \mathit{nl} \diamondsuit\\
&\qquad\qquad \mathit{pretty}\ (x : a) \diamondsuit \mathit{nl} \diamondsuit\\
&\qquad\qquad \mathit{pretty}\ (r : \mathit{Tree}\ a) \diamondsuit \mathit{text}\ \texttt{")"})
\end{aligned}
$$

We declare *pretty* to be open so that we can later extend it.

☞ $\mathit{Typed}\ \alpha \to \mathit{Text}$ is an uncurried version of $\mathit{Type}\ \alpha \to \alpha \to \mathit{Text}$.

The pretty printer produces output in the following style.

```
Now⟩ pretty (tree [0..3] : Tree Int)
(Node (Node (Node Empty
                  0
                  Empty)
            1
            Empty)
      2
      (Node Empty
            3
            Empty))
```

The type of type representations is, of course, by no means special to pretty printing.

A second example: collecting integers.

$$
\begin{array}{ll}
\textbf{open}\ \mathit{ints} :: \mathit{Typed}\ \alpha \to [\mathit{Int}] & \\
\mathit{ints}\ (i : \mathit{Int}) & = [i] \\
\mathit{ints}\ (c : \mathit{Char}) & = \mathit{Nil} \\
\mathit{ints}\ ((x, y) : \mathit{Pair}\ a\ b) & = \mathit{ints}\ (x : a) +\!\!+ \mathit{ints}\ (y : b) \\
\mathit{ints}\ (xs : \mathit{List}\ a) & = \mathit{concat}\ [\mathit{ints}\ (x : a) \mid x \leftarrow xs] \\
\mathit{ints}\ (t : \mathit{Tree}\ a) & = \mathit{ints}\ (\mathit{inorder}\ t : \mathit{List}\ a)
\end{array}
$$

We have declared $Type$ to be open so that we can freely add new constructors to the $Type$ data type and that we can freely add new equations to existing open functions on $Type$.

Whenenver we define a new data type

$$\mathbf{data}\ Perfect\ \alpha = Zero\ \alpha \mid Succ\ (Perfect\ (\alpha, \alpha))$$

we extend $Type$ by a new constructor.

$$Perfect :: Type\ \alpha \longrightarrow Type\ (Perfect\ \alpha)$$

☞ $Perfect$ is a so-called nested data type.

Then we extend *pretty* by suitable equations.

$$
\begin{aligned}
&\mathit{pretty}\ (\mathit{Zero}\ x : \mathit{Perfect}\ a) \\
&\quad = \mathit{align}\ \texttt{"(Zero "}\ (\mathit{pretty}\ (x : a) \diamondsuit \mathit{text}\ \texttt{")"}) \\
&\mathit{pretty}\ (\mathit{Succ}\ x : \mathit{Perfect}\ a) \\
&\quad = \mathit{align}\ \texttt{"(Succ "}\ (\mathit{pretty}\ (x : \mathit{Perfect}\ (\mathit{Pair}\ a\ a)) \diamondsuit \mathit{text}\ \texttt{")"})
\end{aligned}
$$

```
Now⟩ pretty (perfect 4 1 : Perfect Int)
(Succ (Succ (Succ (Succ (Zero ((((1
                                  ,1)
                                 ,(1
                                  ,1))
                                ,((1
                                  ,1)
                                 ,(1
                                  ,1)))
                               ,(((1
                                  ,1)
                                 ,(1
                                  ,1))
                                ,((1
                                  ,1)
                                 ,(1
                                  ,1)))))))))
```

Whenever we define a new data type,

- we add a constructor of the same name to the type of type representations,
- we add corresponding equations to all generic functions.

Observations:

- the extension of $Type$ is cheap and easy (a compiler could do this for us),
- the extension of all functions on $Type$ is laborious and difficult (can you imagine a compiler doing that?).

☞ In the next section we shall develop a scheme so that it suffices to extend one or two particular overloaded functions. The remaining functions adapt themselves.

overloaded and generic functions. An overloaded function works for a fixed family of types. By contrast, a generic function works for all types, including types that the programmer is yet to define.

3.3 Generic functions

☞ 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*.

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

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

The following table illustrates the stepwise construction of a spine.

$$
\begin{array}{l}
node :: Constr\ (Tree\ Int \to Int \to Tree\ Int \to Tree\ Int) \\
Con\ node :: Spine\ (Tree\ Int \to Int \to Tree\ Int \to Tree\ Int) \\
Con\ node \lozenge (l : Tree\ Int) :: Spine\ (Int \to Tree\ Int \to Tree\ Int) \\
Con\ node \lozenge (l : Tree\ Int) \lozenge (a : Int) :: Spine\ (Tree\ Int \to Tree\ Int) \\
Con\ node \lozenge (l : Tree\ Int) \lozenge (a : Int) \lozenge (r : Tree\ Int) :: Spine\ (Tree\ Int)
\end{array}
$$

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

$$\begin{aligned}&\textbf{data}\ Constr\ \alpha = Descr\{\ constr :: \alpha\\ &\qquad\qquad\qquad\qquad\quad ,\ name\ \ :: String\ \}\end{aligned}$$

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

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

☞ *fromSpine* is parametrically polymorphic.

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

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

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

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.

```
open toSpine :: Typed α → Spine α
toSpine (Empty : Tree a) = Con empty
toSpine (Node l x r : Tree a)
    = Con node ◊ (l : Tree a) ◊ (x : a) ◊ (r : Tree a)

empty :: Constr (Tree α)
empty = Descr{ constr = Empty,
               name   = "Empty"}
node  :: Constr (Tree α → α → Tree α → Tree α)
node  = Descr{ constr = Node,
               name   = "Node"}
```

With all the machinery in place we can now turn *pretty* and *ints* into truly generic functions.

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

$$
\begin{array}{l}
ints :: Typed\ \alpha \rightarrow [Int] \\
ints\ (i : Int) = [i] \\
ints\ x \qquad\quad = intsSpine\ (toSpine\ x)
\end{array}
$$

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

The catch-all case for pretty is almost as easy. We only have to take care that we do not parenthesize nullary constructors.

$$
\begin{aligned}
&pretty\ x = prettySpine\ (toSpine\ x)\\
&\\
&prettySpine :: Spine\ \alpha \to Text\\
&prettySpine\ (Con\ c) = text\ (name\ c)\\
&prettySpine\ (f \lozenge x)\ \ = prettySpine1\ \ f\ (pretty\ x)\\
&\\
&prettySpine1 :: Spine\ \alpha \to Text \to Text\\
&prettySpine1\ (Con\ c)\ d = align\ (\texttt{"("} +\!\!+ name\ c +\!\!+ \texttt{"\ \ "})\ (d \diamondsuit text\ \texttt{")"})\\
&prettySpine1\ (f \lozenge x)\ \ \ d = prettySpine1\ \ f\ (pretty\ x \diamondsuit nl \diamondsuit d)
\end{aligned}
$$

Now, why are we in a better situation than before?

- When we introduce a new data type we still have to extend $Type$ and provide cases for the data constructors in the $toSpine$ function.
- This has to be done only once per data type.
- The code for the generic functions (of which there can be many) is completely unaffected by the addition of a new data type.
- As a further plus, the generic functions are unaffected by changes to a given data type. Only the function $toSpine$ must be adapted to the new definition.

4. Conclusion

Using reflected types we can program overloaded functions. Using a uniform view on data we can generalise overloaded functions to generic ones.

Support for generic programming consists of three essential ingredients:

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

For each dimension there are several choices: instead of the data type $Type$ we could use type classes or a type-safe cast.

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	–