Generic Programming, Now!

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

April 2006 — Revised June 2007

Joint work with Andres Löh

(Pick up the slides at .../~ralf/talks.html#T55.)

1 Introduction

Many of us (most of us?) will 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 an untyped, or rather, in a uni-typed language, it is straightforward to define a generic equality test.

$$
\begin{array}{l}
\quad \text{-- } equal :: Value \to Value \to Value \\
equal\ x_1\ x_2 \\
\quad \mid isChar\ x_1 \wedge isChar\ x_2 = equal_{Char}\ x_1\ x_2 \\
\quad \mid isInt\ \ x_1 \wedge isInt\ \ x_2 = equal_{Int}\ \ x_1\ x_2 \\
\quad \mid isPair\ x_1 \wedge isPair\ x_2 = equal\ (fst\ x_1)\ (fst\ x_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ equal\ (snd\ x_1)\ (snd\ x_2) \\
\quad \mid otherwise\ \ \ \ \ \ \ \ \ \ \ \ \ \ = false
\end{array}
$$

☞ Unfortunately, it is not straightforward to port the code to a typed language.

☞ Furthermore, dynamic type tests aren't helpful for defining generic functions that construct data.

- classicism (1990 –):
  strong background in category theory;
- romanticism (1995 –):
  shift towards type-theoretic approaches;
- realism (2000 –):
  compiler and library hacking.

- In these lectures, 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!
- Besides, we shall try to establish some terminology ...

polymorphic functions. A function of type

$$poly :: \forall \alpha \; . \; Poly \; \alpha$$

is called polymorphic or parametrically polymorphic.

- 1. Introduction
- 2. Preliminaries
- 3. A guided tour
- 4. Type representations
- 5. Views
- 6. Conclusion

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:	$Bool,\ [\alpha],\ \forall\alpha\ .\ \alpha \longrightarrow \alpha$
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

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

☞ 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 (plus generation of fresh type names).

The following alternative definition of the pair data type

$$\mathbf{data}\ \mathit{Pair}\ \alpha\ \beta = \mathit{Pair}\{\mathit{fst} :: \alpha, \mathit{snd} :: \beta\}$$

makes use of Haskell's record syntax: the declaration introduces the data constructor Pair and two accessor functions

$$\begin{aligned}\mathit{fst} &:: \forall \alpha\ \beta\ .\ \mathit{Pair}\ \alpha\ \beta \to \alpha\\ \mathit{snd} &:: \forall \alpha\ \beta\ .\ \mathit{Pair}\ \alpha\ \beta \to \beta\end{aligned}$$

Pairs and lists are examples of type constructors.

- The kind of a type is $*$.

$$\begin{array}{ll} \mathit{Char} & :: * \\ \mathit{Int} & :: * \end{array}$$

- The kind of a type constructor is a function of the kind of its parameters to $*$.

$$\begin{array}{ll} \mathit{Pair} & :: * \longrightarrow * \longrightarrow * \\ [\,] & :: * \longrightarrow * \end{array}$$

☞ Type constructors are written in *red*; kinds in *purple*.

The order of a kind is given by

$$\begin{aligned} order\ (*) &= 0 \\ order\ (\iota \to \kappa) &= max\{1 + order\ (\iota), order\ (\kappa)\}. \end{aligned}$$

☞ Haskell supports kinds of arbitrary order.

2.2 Generalised algebraic data types

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

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

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

In these lectures 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.

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

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.

$$string\ \_ = ""$$

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

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

3. A guided tour

- 3.1 Type-indexed functions
- 3.2 Introducing new data types
- 3.3 Generic functions
- 3.4 Dynamic values
- 3.5 Stocktaking

3.1 Type-indexed 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.

$$
\begin{array}{l}
\textbf{type}\ \mathit{Pretty}\ \alpha = \alpha \to \mathit{Text} \\
\\
\mathit{pretty}_{\mathit{Int}} :: \mathit{Pretty}\ \mathit{Int} \\
\mathit{pretty}_{\mathit{Int}}\ n = \mathit{text}\ (\mathit{show}_{\mathit{Int}}\ n) \\
\\
\mathit{pretty}_{\mathit{TreeInt}} :: \mathit{Pretty}\ (\mathit{Tree}\ \mathit{Int}) \\
\mathit{pretty}_{\mathit{TreeInt}}\ \mathit{Empty} = \mathit{text}\ \texttt{"Empty"} \\
\mathit{pretty}_{\mathit{TreeInt}}\ (\mathit{Node}\ l\ x\ r) = \mathit{align}\ \texttt{"(Node "}\ (\mathit{pretty}_{\mathit{TreeInt}}\ l \diamondsuit \mathit{nl} \diamondsuit \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{pretty}_{\mathit{Int}}\ x \diamondsuit \mathit{nl} \diamondsuit \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{pretty}_{\mathit{TreeInt}}\ r \diamondsuit \mathit{text}\ \texttt{")"}) \\
\\
\mathit{align} :: \mathit{String} \to \mathit{Text} \to \mathit{Text} \\
\mathit{align}\ s\ d = \mathit{indent}\ (\mathit{length}\ s)\ (\mathit{text}\ s \diamondsuit d)
\end{array}
$$

☞ 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.

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

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 lectures we explore a different route!

type-indexed functions. A simple approach to generic programming defines a family of functions indexed by type.

$$poly_\tau :: Poly\ \tau$$

The family contains a definition of $poly_\tau$ for each type τ of interest; the type of $poly_\tau$ is parametric in the type index τ. For brevity, we call $poly$ a type-indexed function (omitting the 'family of').

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

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

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

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

$$pretty :: Type \to \alpha \to Text$$

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

$$pretty :: Type\ \alpha \to \alpha \to Text$$

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.

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

☞ The derived constructor $String$, defined by a pattern definition, is equal to $List\ Char$ in all contexts.

Each type has a unique representation:

- the type Int is represented by the constructor Int,
- the type $(String, Int)$ is represented by $Pair\ String\ Int$,
- 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 $\mathit{Typed}\ \mathit{Int}$.
- $(47, \texttt{"hello"}) : \mathit{Pair}\ \mathit{Int}\ \mathit{String}$ is an element of $\mathit{Typed}\ (\mathit{Int}, \mathit{String})$.

☞ 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))
```

type-polymorphic functions. A function of type

$$poly :: \forall \alpha \; . \; Type \; \alpha \longrightarrow Poly \; \alpha$$

is called type-polymorphic or intensionally polymorphic. By contrast, a function of type $\forall \alpha \; . \; Poly \; \alpha$ is called parametrically polymorphic.

3.2 Introducing new data types

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 type-indexed functions is laborious and difficult (can you imagine a compiler doing that?).

☞ In the sequel 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 has 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*.

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

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

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.

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

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 $Typed\ \alpha \longrightarrow Spine\ \alpha$.

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.

```
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* into a truly generic function.

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

$$
\begin{array}{l}
pretty\ x = pretty\_\ (toSpine\ x) \\
\\
pretty\_ :: Spine\ \alpha \to Text \\
pretty\_\ (Con\ c) = text\ (name\ c) \\
pretty\_\ (f \diamond x) \quad = pretty1\_\ f\ (pretty\ x) \\
\\
pretty1\_ :: Spine\ \alpha \to Text \to Text \\
pretty1\_\ (Con\ c)\ d = align\ (\texttt{"("} +\!\!+ name\ c +\!\!+ \texttt{"\ \ "})\ (d \diamondsuit text\ \texttt{")"}) \\
pretty1\_\ (f \diamond x) \quad d = pretty1\_\ f\ (pretty\ x \diamondsuit nl \diamondsuit d)
\end{array}
$$

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.

3.4 Dynamic values

Using *Typed* we can also represent dynamic values.

```
data Dynamic :: * where
  Dyn :: Typed α → Dynamic
```

☞ α does not appear in the result type: it is effectively existentially quantified.

☞ *Dynamic* is the union of all typed values.

```
misc :: [Dynamic]
misc = [Dyn (4711 : Int), Dyn ("hello world" : String)]
```

Can we apply *pretty* to dynamic values? Yes, we only have to turn *Dynamic* into a representable type.

First, we add *Type* and *Typed* to the type of representable types.

$$\begin{array}{l} \mathit{Type} \;\; :: \mathit{Type}\ \alpha \to \mathit{Type}\ (\mathit{Type}\ \alpha) \\ \mathit{Typed} :: \mathit{Type}\ \alpha \to \mathit{Type}\ (\mathit{Typed}\ \alpha) \end{array}$$

The first line exactly follows the scheme for unary type constructors: the representation of $T :: * \to *$ is $T :: \mathit{Type}\ \alpha \to \mathit{Type}\ (T\ \alpha)$.

Furthermore, we provide suitable instances of *toSpine*.

$$\begin{array}{ll} \mathit{toSpine}\ (\mathit{Char} : \mathit{Type}\ \mathit{Char}) & = \mathit{Con}\ \mathit{char} \\ \mathit{toSpine}\ (\mathit{List}\ t : \mathit{Type}\ (\mathit{List}\ a)) & = \mathit{Con}\ \mathit{list} \diamond (t : \mathit{Type}\ a) \\ \ldots & \\ \mathit{toSpine}\ ((x : t) : \mathit{Typed}\ a) & = \mathit{Con}\ \mathit{hastype} \diamond (x : t) \diamond (t : \mathit{Type}\ t) \end{array}$$

☞ *hastype* is the infix operator ':' augmented by additional information.

Second, we extend *Type* and *toSpine* by a *Dynamic* case.

$$Dynamic :: Type\ Dynamic$$

$$toSpine\ (Dyn\ x : Dynamic) = Con\ dyn \diamond (x : Typed\ (type\ x))$$

☞ This instance does not follow the general pattern for *toSpine*: *Dyn*'s argument is existentially quantified and the general scheme cannot cope with existentially quantified types.

Finally, we provide an ad-hoc type case for typed values (we want to use infix rather than prefix notation for ':').

```
pretty ((x : t) : Typed a) = align "( " (pretty (x : t)) ♢ nl ♢    -- t = a
                             align ": " (pretty (t : Type t)) ♢ text ")"
```

Here is an interactive session that illustrates pretty printing dynamic values.

```
Now⟩ pretty (misc : List Dynamic)
[ (Dyn (4711
        :Int))
, (Dyn ("hello world"
        :(List Char)))]
```

The constructor Dyn turns a static into a dynamic value. The other way round involves a dynamic type check:

$$\textbf{open}\ equal :: Type\ \alpha \to Type\ \beta \to Maybe\ (\alpha :=: \beta)$$

☞ $equal$ is an overloaded function that takes two type representations and possibly returns a proof of their equality (a simple truth value is not enough).

An element of $\tau_1 :=: \tau_2$ is a proof that τ_1 and τ_2 are equal.

$$\textbf{data}\ (:=:) :: * \to * \to * \ \textbf{where}\ \mathit{Refl} :: \alpha :=: \alpha$$

$$\mathit{apply} :: (\alpha :=: \beta) \to (\alpha \to \beta)$$
$$\mathit{apply}\ \mathit{Refl}\ x = x$$

☞ This type has the property that it is non-empty if and only if its argument types are equal.

The type equality type ':=:' has all the properties of a congruence relation.

$$\mathit{Refl} :: \alpha :=: \alpha$$

$$\mathit{ctx}_1 :: (\alpha :=: \beta) \to (\psi\ \alpha :=: \psi\ \beta)$$

$$\mathit{ctx}_2 :: (\alpha_1 :=: \beta_1) \to (\alpha_2 :=: \beta_2) \to (\psi\ \alpha_1\ \alpha_2 :=: \psi\ \beta_1\ \beta_2)$$

The type equality check is given by

$$
\begin{array}{lll}
\textbf{open}\ equal :: Type\ \alpha \to Type\ \beta \to Maybe\ (\alpha :=: \beta) & & \\
equal\ Int & Int & = return\ Refl \\
equal\ Char & Char & = return\ Refl \\
equal\ (Pair\ a_1\ a_2)\ & (Pair\ b_1\ b_2) & = liftM2\ ctx_2\ (equal\ a_1\ b_1)\ (equal\ a_2\ b_2) \\
equal\ (List\ a) & (List\ b) & = liftM\ \ ctx_1\ (equal\ a\ b)
\end{array}
$$

Since the equality check may fail, we lift the congruence proofs into the *Maybe* monad using *return*, *liftM*, and *liftM2*.

☞ The running time of the cast function that *equal* returns is linear in the size of the type (it is independent of the size of its argument structure).

The *cast* operation simply calls *equal* and then applies the conversion function to the dynamic value.

$$
\begin{aligned}
&\mathit{cast} && :: \mathit{Dynamic} \to \mathit{Type}\ \alpha \to \mathit{Maybe}\ \alpha \\
&\mathit{cast}\ (\mathit{Dyn}\ (x : a))\ t && = \mathit{fmap}\ (\lambda p \to \mathit{apply}\ p\ x)\ (\mathit{equal}\ a\ t)
\end{aligned}
$$

Here is an interactive session that illustrates the use of *cast*.

```
Now⟩ let d = Dyn (4711 : Int)
Now⟩ pretty (d : Dynamic)
(Dyn (4711
      :Int))
Now⟩ d `cast` Int
Just 4711
Now⟩ fromJust (d `cast` Int) + 289
5000
Now⟩ d `cast` Char
Nothing
```

☞ *cast* can be seen as the dynamic counterpart of the colon operator.

generic functions and dynamic values. Generics and dynamics are dual concepts:

$$\begin{array}{ll} \text{generic function:} & \forall\alpha \;.\; \mathit{Type}\ \alpha \longrightarrow \sigma \\ \text{dynamic value:} & \exists\alpha \;.\; \mathit{Type}\ \alpha \times \sigma \end{array}$$

This is analogous to first-order predicate logic where $\forall x{:}T \;.\; P(x)$ is shorthand for $\forall x \;.\; T(x) \Rightarrow P(x)$ and $\exists x{:}T \;.\; P(x)$ abbreviates $\exists x \;.\; T(x) \wedge P(x)$.

3.5 Stocktaking

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. Here we stick to $Type$.

☞ Type representations and generic views are investigated in the next two sections.

4. Type representations

- 4.1 Representation types for types of a fixed kind
- 4.2 Kind-indexed families of representation types
- 4.3 Representations of open type terms

4.1 Representation types for types of a fixed kind

Recap: Since type constructors are reflected onto the value level, the type of the data constructor T depends on the kind of the type constructor T.

$$
\begin{array}{l}
\mathit{Int} \;\; :: \mathit{Type}\ \mathit{Int} \\
\mathit{Pair} :: \forall\alpha\ .\ \mathit{Type}\ \alpha \to (\forall\beta\ .\ \mathit{Type}\ \beta \to \mathit{Type}\ (\alpha,\beta)) \\
\mathit{List} \;\; :: \forall\alpha\ .\ \mathit{Type}\ \alpha \to \mathit{Type}\ [\alpha]
\end{array}
$$

☞ A type constructor $T :: \kappa$ of higher kind is represented by a polymorphic function:

$$
\begin{array}{l}
T :: \mathit{Type}_\kappa\ T \\
\\
\textbf{type}\ \mathit{Type}_*\ \ \alpha = \mathit{Type}\ \alpha \\
\textbf{type}\ \mathit{Type}_{\iota\to\kappa}\ \varphi = \forall\alpha\ .\ \mathit{Type}_\iota\ \alpha \to \mathit{Type}_\kappa\ (\varphi\ \alpha)
\end{array}
$$

So far we have only seen first-order type constructors. Here is a second-order one:

$$\textbf{newtype}\ \mathit{Fix}\ \varphi = \mathit{In}\{\,\mathit{out} :: \varphi\ (\mathit{Fix}\ \varphi)\}$$

☞ $\mathit{Fix} :: (* \longrightarrow *) \longrightarrow *$ is a fixed point operator on the type level. Consequently, Fix has a rank-2 type: it takes a polymorphic function as an argument.

$$\mathit{Fix} :: \forall\varphi\ .\ (\forall\alpha\ .\ \mathit{Type}\ \alpha \longrightarrow \mathit{Type}\ (\varphi\ \alpha)) \longrightarrow \mathit{Type}\ (\mathit{Fix}\ \varphi)$$

$$\mathit{toSpine}\ (\mathit{In}\ x : \mathit{Fix}\ f) = \mathit{Con}\ \mathit{in} \diamond (x : f\ (\mathit{Fix}\ f))$$

Here, in is the annotated variant of In. Again, the definition of $\mathit{toSpine}$ pedantically follows the general scheme.

☞ However, we cannot extend equal to Fix as the argument of Fix is a polymorphic function. Also, $\mathit{Fix}\ \mathit{List}$ is not a legal pattern.

The generic pretty printer generalises functions of type

$$Char \to Text, \quad String \to Text, \quad [[Int]] \to Text$$

to a single generic function of type

$$Type\ \alpha \to \alpha \to Text \qquad \cong Typed\ \alpha \to Text$$

☞ A generic function may also abstract over a type constructor of higher kind: a generic *size* function generalises functions of type

$$[\alpha] \longrightarrow Int, \quad Tree\ \alpha \longrightarrow Int, \quad [Tree\ \alpha] \longrightarrow Int$$

to a single generic function of type

$$Type'\ \varphi \longrightarrow \varphi\ \alpha \longrightarrow Int \qquad \cong Typed'\ \varphi\ \alpha \longrightarrow Int$$

where $Type'$ is a representation type for types of kind $* \longrightarrow *$ and $Typed'$ is a suitable type for annotating values with these representations.

How can we represent type constructors of kind $* \longrightarrow *$?

We define a new tailor-made representation type:

$$\textbf{open data}\ \ Type' :: (* \longrightarrow *) \longrightarrow * \ \textbf{where}$$
$$\quad List\ :: Type'\ [\,]$$
$$\quad Tree :: Type'\ Tree$$

The type $Type'$ is only inhabited by two constructors since the other data types have kinds different from $* \longrightarrow *$.

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

☞ Think of the prime as shorthand for the kind index $* \longrightarrow *$.

An overloaded version of *size* is now straightforward to define.

$$
\begin{array}{l}
size :: Typed'\ \varphi\ \alpha \longrightarrow Int \\
size\ (Nil :'\ List) \qquad\qquad\ = 0 \\
size\ (Cons\ x\ xs :'\ List) \ = 1 + size\ (xs :'\ List) \\
size\ (Empty :'\ Tree) \qquad = 0 \\
size\ (Node\ l\ x\ r :'\ Tree) = size\ (l :'\ Tree) + 1 + size\ (r :'\ Tree)
\end{array}
$$

☞ However, *size* is not as flexible as *pretty*: if x is, say, a list of trees of integers, we can call $pretty\ (x : List\ (Tree\ Int))$, but we cannot call *size* to count the total number of integers.

Computing the size of a compound data structure is inherently ambiguous: if x is a list of trees of integers, shall we count the number of integers, the number of trees or the number of lists?

Formally, we have to solve the type equation $\varphi\,\tau = \mathit{List}\ (\mathit{Tree}\ \mathit{Int})$. The equation has four principal solutions:

- $\varphi = \Lambda\alpha \to \alpha$ and $\tau = \mathit{List}\ (\mathit{Tree}\ \mathit{Int})$,
- $\varphi = \Lambda\alpha \to \mathit{List}\ \alpha$ and $\tau = \mathit{Tree}\ \mathit{Int}$,
- $\varphi = \Lambda\alpha \to \mathit{List}\ (\mathit{Tree}\ \alpha)$ and $\tau = \mathit{Int}$, and
- $\varphi = \Lambda\alpha \to \mathit{List}\ (\mathit{Tree}\ \mathit{Int})$ and τ arbitrary.

☞ How can we represent these different container types?

To represent the different container types, we can lift the type constructors and include the identity type Id as a representation of the type variable α:

$$\begin{array}{l}\textbf{newtype}\ Id\ \alpha = In_{Id}\{\, out_{Id} :: \alpha\,\}\\[2ex]
Id \quad :: Type'\ Id\\
Int' \ :: Type'\ Int'\\
List' :: Type'\ \varphi \longrightarrow Type'\ (List'\ \varphi)\\
Tree' :: Type'\ \varphi \longrightarrow Type'\ (Tree'\ \varphi)\end{array}$$

☞ $List'$ takes a type of kind $* \longrightarrow *$ to a type of kind $* \longrightarrow *$.

Using the lifted types we can specify the four different container types:

- Id,
- $List'\ Id$,
- $List'\ (Tree'\ Id)$, and
- $List'\ (Tree'\ Int')$.

Here are the lifted versions of the type constructors:

$$\begin{array}{ll}
\textbf{newtype}\ \mathit{Int}' & \chi = \mathit{In}_{\mathit{Int}'}\{\mathit{out}_{\mathit{Int}'} :: \mathit{Int}\} \\
\textbf{data}\ \mathit{List}'\ \alpha' & \chi = \mathit{Nil}' \mid \mathit{Cons}'\ (\alpha'\ \chi)\ (\mathit{List}'\ \alpha'\ \chi) \\
\textbf{data}\ \mathit{Pair}'\ \alpha'\ \beta' & \chi = \mathit{Pair}'\ (\alpha'\ \chi)\ (\beta'\ \chi) \\
\textbf{data}\ \mathit{Tree}'\ \alpha' & \chi = \mathit{Empty}' \mid \mathit{Node}'\ (\mathit{Tree}'\ \alpha'\ \chi)\ (\alpha'\ \chi)\ (\mathit{Tree}'\ \alpha'\ \chi)
\end{array}$$

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.

The function $size$ can be easily extended to Id and to the lifted types.

$$
\begin{array}{ll}
size\ (x :'\ Id) & = 1 \\
size\ (c :'\ Char') & = 0 \\
size\ (i :'\ Int') & = 0 \\
size\ (Nil' :'\ List'\ a') & = 0 \\
size\ (Cons'\ x\ xs :'\ List'\ a') & = size\ (x :'\ a') + size\ (xs :'\ List'\ a') \\
size\ (Empty' :'\ Tree'\ a') & = 0 \\
size\ (Node'\ l\ x\ r :'\ Tree'\ a') & \\
\quad = size\ (l :'\ Tree'\ a') + size\ (x :'\ a') + size\ (r :'\ Tree'\ a') &
\end{array}
$$

☞ Unfortunately, in Haskell $size$ no longer works on the original data types. We return to the problem later in Section 5.3.

☞ The representation type $Type'$ is similar to $Type$ except for the kinds:

$$T' :: Type'_\kappa\ T'$$

$$\mathbf{type}\ Type'_*\ \ \alpha = Type'\ \alpha$$
$$\mathbf{type}\ Type'_{\iota\to\kappa}\ \varphi = \forall\alpha\ .\ Type'_\iota\ \alpha \to Type'_\kappa\ (\varphi\ \alpha)$$

☞ Defining a $* \to *$-indexed function is similar to defining a $*$-indexed function except that one has to additionally consider the Id case.

4.2 Kind-indexed families of representation types

Type-indexed functions may abstract over arbitrary type constructors: $pretty$ abstracts over types of kind $*$, $size$ abstracts over types of kind $* \longrightarrow *$.

Sometimes a type-indexed function even makes sense for types of different kinds. A paradigmatic example is the mapping function:

- the mapping function of a type φ of kind $* \longrightarrow *$ lifts a function of type $\alpha_1 \longrightarrow \alpha_2$ to a function of type $\varphi\ \alpha_1 \longrightarrow \varphi\ \alpha_2$;
- the mapping function of a type ψ of kind $* \longrightarrow * \longrightarrow *$ takes two functions of type $\alpha_1 \longrightarrow \alpha_2$ and $\beta_1 \longrightarrow \beta_2$ respectively and returns a function of type $\psi\ \alpha_1\ \beta_1 \longrightarrow \psi\ \alpha_2\ \beta_2$;
- the mapping function of a type σ of kind $*$ is the identity of type $\sigma \longrightarrow \sigma$.

Since the type of the mapping functions depends on the kind of the type argument, we have, a kind-indexed family of overloaded functions.

We require a kind-indexed family of representation types.

$$\textbf{open data}\ \ \mathit{Type}_\kappa :: \kappa \longrightarrow * \ \textbf{where}$$
$$\quad \mathit{T}_\kappa :: \mathit{Type}_\kappa\ T$$

☞ $\mathit{Int} :: *$ is represented by a data constructor of Type_*; $\mathit{Tree} :: * \longrightarrow *$ is represented by a data constructor of type $\mathit{Type}_{*\to *}$.

☞ How can we represent the application of Tree to some type?

We also represent type application syntactically:

$$\mathit{App}_{\iota,\kappa} :: \mathit{Type}_{\iota\to\kappa}\ \varphi \longrightarrow \mathit{Type}_\iota\ \alpha \longrightarrow \mathit{Type}_\kappa\ (\varphi\ \alpha)$$

Theoretically, we need an infinite number of $App_{\iota,\kappa}$ constructors. Practically, only a few are needed:

$$
\begin{array}{ll}
\textbf{open data} & Type_{*} :: * \to * \textbf{ where} \\
\quad Int_{*} & :: Type_{*}\ Int \\
\quad App_{*,*} & :: Type_{*\to *}\ \varphi \to Type_{*}\ \alpha \to Type_{*}\ (\varphi\ \alpha) \\
 & \\
\textbf{open data} & Type_{*\to *} :: (* \to *) \to * \textbf{ where} \\
\quad List_{*\to *} & :: Type_{*\to *}\ [\,] \\
\quad Tree_{*\to *} & :: Type_{*\to *}\ Tree \\
\quad App_{*,*\to *} & :: Type_{*\to *\to *}\ \varphi \to Type_{*}\ \alpha \to Type_{*\to *}\ (\varphi\ \alpha) \\
 & \\
\textbf{open data} & Type_{*\to *\to *} :: (* \to * \to *) \to * \textbf{ where} \\
\quad Pair_{*\to *\to *} & :: Type_{*\to *\to *}\ (,)
\end{array}
$$

For example, $Tree\ Int$ is represented by $Tree_{*\to *}$ ‘$App_{*,*}$‘ Int_{*}.

☞ The family $Type_{\kappa}$ is a faithful representation of Haskell's type system!

Let's tackle an example of a type-indexed function that works for types of different kinds. We re-implement the function $size$.

$$size :: Type_{*\to *}\ \varphi \to \varphi\ \alpha \to Int$$

How can we generalise $size$ so that it works for types of arbitrary kinds?

The essential step is to abstract away from $size$'s action on values of type α turning the action of type $\alpha \to Int$ into an additional argument:

$$count_{*\to *} :: Type_{*\to *}\ \varphi \to (\alpha \to Int) \to (\varphi\ \alpha \to Int)$$

We call $size$'s kind-indexed generalisation $count$.

Since $count_\kappa$ is indexed by kind it also has a kind-indexed type.

$$count_\kappa :: Type_\kappa\ \alpha \to Count_\kappa\ \alpha$$

$$\textbf{type}\ Count_*\ \ \alpha = \alpha \to Int$$
$$\textbf{type}\ Count_{\iota\to\kappa}\ \varphi = \forall\alpha\ .\ Count_\iota\ \alpha \to Count_\kappa\ (\varphi\ \alpha)$$

☞ The scheme dictates that $count_\kappa$ maps type application to application of generic functions:

$$count_\kappa\ (App_{\iota,\kappa}\ f\ \ a) = (count_{\iota\to\kappa}\ f)\ (count_\iota\ a)$$

This case for $App_{\iota,\kappa}$ is truly generic: it is the same for all kind-indexed generic functions and for all combinations of ι and κ.

The type-specific behaviour is solely determined by the cases for the different type constructors.

$$
\begin{array}{l}
\textbf{open}\ count_{*} :: Type_{*}\ \alpha \to Count_{*}\ \alpha \\
count_{*}\ (f\ `App_{*,*}`\ a) = (count_{*\to *}\ f)\ (count_{*}\ a) \\
count_{*}\ t \qquad\qquad\quad = const\ 0 \\
\\
\textbf{open}\ count_{*\to *} :: Type_{*\to *}\ \alpha \to Count_{*\to *}\ \alpha \\
count_{*\to *}\ List_{*\to *} \qquad\quad c = sum_{[]}\ .\ map_{[]}\ c \\
count_{*\to *}\ Tree_{*\to *} \qquad\quad c = count_{*\to *}\ List_{*\to *}\ c\ .\ inorder \\
count_{*\to *}\ (f\ `App_{*,*\to *}`\ a)\ c = (count_{*\to *\to *}\ f)\ (count_{*}\ a)\ c \\
\\
\textbf{open}\ count_{*\to *\to *} :: Type_{*\to *\to *}\ \alpha \to Count_{*\to *\to *}\ \alpha \\
count_{*\to *\to *}\ (Pair_{*\to *\to *})\ c_1\ c_2 = \lambda(x_1, x_2) \to c_1\ x_1 + c_2\ x_2
\end{array}
$$

☞ We have to repeat the generic $App_{\iota,\kappa}$ case for every instance of ι and κ.

Taking the size of a compound data structure such as a list of trees of integers is now much easier than before:

$$
\begin{array}{l}
Now\rangle\ \textbf{let}\ ts = [tree\ [0\,..\,i] \mid i \leftarrow [0\,..\,9]] \\
Now\rangle\ (const\ 1)\ ts \\
1 \\
Now\rangle\ count_{*\to *}\ List_{*\to *}\ (const\ 1)\ ts \\
10 \\
Now\rangle\ count_{*\to *}\ List_{*\to *}\ (count_{*\to *}\ Tree_{*\to *}\ (const\ 1))\ ts \\
55 \\
Now\rangle\ count_{*\to *}\ List_{*\to *}\ (count_{*\to *}\ Tree_{*\to *}\ (count_{*}\ Int_{*}))\ ts \\
0
\end{array}
$$

kind-indexed functions. A kind-indexed family of type-polymorphic functions

$$poly_\kappa :: \forall \alpha \; . \; Type_\kappa \; \alpha \longrightarrow Poly_\kappa \; \alpha$$

contains a definition of $poly_\kappa$ for each kind κ of interest. The type representation $Type_\kappa$ and the type $Poly_\kappa$ are indexed by kind, as well. For brevity, we call $poly_\kappa$ a kind-indexed function (omitting the 'family of type-polymorphic').

4.3 Representations of open type terms

Haskell's type system is somewhat peculiar as it features type application but not type abstraction.

If Haskell had type-level lambdas, we could determine the instances of $* \longrightarrow *$-indexed functions using suitable type abstractions:

- $\Lambda\alpha \longrightarrow \alpha$,
- $\Lambda\alpha \longrightarrow \mathit{List}\ \alpha$,
- $\Lambda\alpha \longrightarrow \mathit{List}\ (\mathit{Tree}\ \alpha)$, or
- $\Lambda\alpha \longrightarrow \mathit{List}\ (\mathit{Tree}\ \mathit{Int})$.

☞ Alternatively, we can represent an anonymous type function by an open type term: $\Lambda\alpha \longrightarrow \mathit{List}\ (\mathit{Tree}\ \alpha)$, for instance, is represented by $\mathit{List}\ (\mathit{Tree}\ a)$ where a is a suitable representation of α.

To motivate the representation of free type variables, consider the following version of *count* that is defined on *Type*, the original type of type representations.

$$
\begin{array}{l}
count :: Type\ \alpha \to (\alpha \to Int) \\
count\ (Int) \qquad\quad = const\ 0 \\
count\ (Pair\ a\ b) = \lambda(x, y) \to count\ a\ x + count\ b\ y \\
count\ (List\ a) \quad = sum_{[]}\ .\ map_{[]}\ (count\ a) \\
count\ (Tree\ a) \quad = sum_{[]}\ .\ map_{[]}\ (count\ a)\ .\ inorder
\end{array}
$$

☞ *count* is point-free but also pointless as it always returns the constant 0.

☞ We can make *count* more useful by adding a representation of unbound type variables to *Type*.

What constitutes a suitable representation of an unbound type variable?

An intriguing choice is to identify the type variable with its meaning: an unbound type variable is representated by a constructor that embeds a *count* instance into a type representation.

$$Count :: (\alpha \to Int) \to Type\ \alpha$$

Since the 'type variable' carries its meaning, the *count* instance is simple.

$$count\ (Count\ c) = c$$

☞ This approach is an instance of the 'embedding trick' for higher-order abstract syntax: *Count* is the inverse of *count*.

Now, we can specify the action on the free type variable when we call *count*:

```
Now⟩ let ts = [tree [0..i] | i ← [0..9 :: Int]]
Now⟩ let a = Count (const 1)
Now⟩ count a ts
1
Now⟩ count (List a) ts
10
Now⟩ count (List (Tree a)) ts
55
Now⟩ count (List (Tree Int)) ts
0
```

The approach would work perfectly well if *count* were the only generic function.

```
Now⟩ pretty (4711 : a)
*** Exception: Non-exhaustive patterns in function pretty
```

A safer approach is to parameterise $Type$ by the type of generic functions.

$$\begin{aligned}&\textbf{open data}\ PType :: (* \to *) \to * \to * \ \textbf{where}\\ &\quad PInt \ :: PType\ poly\ Int\\ &\quad PPair :: PType\ poly\ \alpha \to PType\ poly\ \beta \to PType\ poly\ (\alpha, \beta)\\ &\quad PList \ :: PType\ poly\ \alpha \to PType\ poly\ [\alpha]\\ &\quad PTree :: PType\ poly\ \alpha \to PType\ poly\ (Tree\ \alpha)\end{aligned}$$

A generic function now has type $PType\ Poly\ \alpha \to Poly\ \alpha$.

An unbound type variable is represented by a constructor of the inverse type:

$$PVar :: poly\ \alpha \to PType\ poly\ \alpha$$

☞ Since we abstract over $Poly$, we make do with a single constructor: $PVar$ can be used to embed instances of arbitrary generic functions.

The definition of *count* can be easily adapted to the new representation.

$$\textbf{newtype}\ Count\ \alpha = In_{Count}\{\, out_{Count} :: \alpha \to Int\,\}$$

$$\begin{aligned}
&pcount :: PType\ Count\ \alpha \to (\alpha \to Int)\\
&pcount\ (PVar\ c) &&= out_{Count}\ c\\
&pcount\ (PInt) &&= const\ 0\\
&pcount\ (PPair\ a\ b) &&= \lambda(x,y) \to pcount\ a\ x + pcount\ b\ y\\
&pcount\ (PList\ a) &&= sum_{[]}\ .\ map_{[]}\ (pcount\ a)\\
&pcount\ (PTree\ a) &&= sum_{[]}\ .\ map_{[]}\ (pcount\ a)\ .\ inorder
\end{aligned}$$

The code is almost identical to what we have seen before except that the type signature is more precise.

```
Now⟩ let ts = [tree [0 .. i] | i ← [0 .. 9 :: Int]]
Now⟩ let a = PVar (In_Count (const 1))
Now⟩ :type a
a :: ∀α . PType Count α
Now⟩ pcount (a) ts
1
Now⟩ pcount (PList a) ts
10
Now⟩ pcount (PList (PTree a)) ts
55
Now⟩ pcount (PList (PTree PInt)) ts
0
```

☞ The type of a limits the applicability of the unbound type variable: passing it to *pretty* would result in a static type error.

5. Views

It is useful to make the concept of a view explicit.

$$\begin{array}{l}\textbf{data}\ \mathit{View} :: * \to * \ \textbf{where}\\ \quad \mathit{View} :: \mathit{Type}\ \beta \to (\alpha \to \beta) \to (\alpha \leftarrow \beta) \to \mathit{View}\ \alpha\end{array}$$

A view consists of three ingredients: a so-called structure type that constitutes the actual view on the original data type and two functions that convert to and fro.

To define a view the generic programmer simply provides a view function

$$\mathit{view} :: \mathit{Type}\ \alpha \to \mathit{View}\ \alpha$$

The view function is then used in the catch-all case of a generic function.

$$\begin{array}{l}\mathit{pretty}\ (x : t) = \textbf{case}\ \mathit{view}\ t\ \textbf{of}\\ \qquad\qquad\qquad \mathit{View}\ u\ \mathit{fromData}\ \mathit{toData} \to \mathit{pretty}\ (\mathit{fromData}\ x : u)\end{array}$$

For the type $Type'$ of lifted type representations we can set up a similar machinery.

$$\mathbf{type}\ \varphi \dot{\rightarrow} \psi = \forall \alpha\ .\ \varphi\ \alpha \rightarrow \psi\ \alpha$$

$$\begin{aligned}&\mathbf{data}\ View' :: (* \rightarrow *) \rightarrow * \ \mathbf{where}\\ &\quad View' :: Type'\ \psi \rightarrow (\varphi \dot{\rightarrow} \psi) \rightarrow (\varphi \dot{\leftarrow} \psi) \rightarrow View'\ \varphi\end{aligned}$$

The view function is now of type

$$view' :: Type'\ \varphi \rightarrow View'\ \varphi$$

and is used as follows:

$$\begin{aligned}&size\ (x :'\ a') = \mathbf{case}\ view\ a'\ \mathbf{of}\\ &\qquad\qquad View'\ b'\ fromData\ toData \rightarrow size\ (fromData\ x :'\ b')\end{aligned}$$

5.1 Spine view

The spine view of the type τ is simply $Spine\ \tau$:

$$\begin{aligned} &spine \quad :: Type\ \alpha \longrightarrow View\ \alpha \\ &spine\ a = View\ (Spine\ a)\ (\lambda x \longrightarrow toSpine\ (x : a))\ fromSpine \end{aligned}$$

☞ Recall that $fromSpine$ is parametrically polymorphic, while $toSpine$ is an overloaded function.

The definition of $toSpine$ follows a simple 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.

☞ The equation is only valid if $vars\ (t_1) \cup \cdots \cup vars\ (t_n) \subseteq vars\ (t_0)$, that is, if C's type signature contains no existentially quantified type variables.

- 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 $read$).
- Furthermore, functions that abstract over type constructors (such as $size$) are out of reach.

5.2 The type spine view

A generic consumer is a function of type $Type\ \alpha \to \alpha \to \tau$. The generic part of a consumer follows the general pattern:

$$
\begin{aligned}
&consume :: Type\ \alpha \to \alpha \to \tau \\
&\ldots \\
&consume\ a\ x\ \ = consume\_\ (toSpine\ (x : a)) \\
&\\
&consume\_\ \ :: Spine\ \alpha \to \tau \\
&consume\_\ \ldots\ = \ldots
\end{aligned}
$$

The element x is converted to the spine representation, over which the helper function $consume\_$ then recurses.

By duality, we would expect that a generic producer of type $Type\ \alpha \to \tau \to \alpha$ takes on the following form:

```
produce :: Type α → τ → α
...
produce a t   = fromSpine (produce_ t)

produce_   :: τ → Spine α   -- this does not work
produce_ ... = ...
```

The helper function $produce\_$ generates an element in spine representation, which $fromSpine$ converts back.

☞ Unfortunately, this approach does not work. If it were possible to define $produce\_ :: \forall\alpha\ .\ \tau \to Spine\ \alpha$, then the composition $fromSpine\ .\ produce\_$ would yield a parametrically polymorphic function of type $\forall\alpha\ .\ \tau \to \alpha$ (an unsafe cast).

We require additional information about the data type, information that the spine view does not provide.

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

$$\textbf{type } \mathit{Datatype}\ \alpha = [\mathit{Signature}\ \alpha]$$

$$\begin{aligned}&\textbf{data } \mathit{Signature} :: * \to * \textbf{ where}\\ &\quad \mathit{Sig} :: \mathit{Constr}\ \alpha \to \mathit{Signature}\ \alpha\\ &\quad (\Box) :: \mathit{Signature}\ (\alpha \to \beta) \to \mathit{Type}\ \alpha \to \mathit{Signature}\ \beta\end{aligned}$$

☞ The type $\mathit{Signature}$ is almost identical to the Spine type, except for the second argument of '$\Box$', 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 $Datatype\ \alpha$.

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

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

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

$$
\begin{aligned}
&generate :: Type\ \alpha \to Int \to [\alpha] \\
&generate\ a\ 0 \qquad\quad = Nil \\
&generate\ a\ (d+1) \;= concat\ [generate\_\ s\ d \mid s \leftarrow datatype\ a] \\
\\
&generate\_ :: Signature\ \alpha \to Int \to [\alpha] \\
&generate\_\ (Sig\ c)\ d = [constr\ c] \\
&generate\_\ (s \square a)\ \ d = [f\ x \mid f \leftarrow generate\_\ s\ d, x \leftarrow generate\ a\ d]
\end{aligned}
$$

The helper function $generate\_$ constructs all terms that conform to a given signature.

The scheme can even be extended to generalised algebraic data types.

Since $Datatype\ \alpha$ is a homogeneous list, we have to partition the constructors according to their result types.

$$
\begin{array}{l}
datatype\ (Expr\ Bool) \\
\quad = [Sig\ eq \square Expr\ Int \square Expr\ Int, \\
\qquad Sig\ if \square Expr\ Bool \square Expr\ Bool \square Expr\ Bool] \\
datatype\ (Expr\ Int) \\
\quad = [Sig\ num \square Int, \\
\qquad Sig\ plus \square Expr\ Int \square Expr\ Int, \\
\qquad Sig\ if \square Expr\ Bool \square Expr\ Int \square Expr\ Int] \\
datatype\ (Expr\ a) \\
\quad = [Sig\ if \square Expr\ Bool \square Expr\ a \square Expr\ a]
\end{array}
$$

☞ The equations are ordered from specific to general; each right-hand side lists all the constructors that have the given result type or a more general one.

- 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.

5.3 Lifted spine view

The spine view is not suitable for defining $* \longrightarrow *$-indexed functions. To illustrate, consider a variant of $Tree$ whose inner nodes are annotated with a balance factor.

$$\textbf{data}\ BalTree\ \alpha = Empty \mid Node\ Int\ (BalTree\ \alpha)\ \alpha\ (BalTree\ \alpha)$$

If we call the generic function on a value of type $BalTree\ Int$, then the spine view handles the two integer components in a uniform way.

☞ A generic version of sum, however, must consider the label of type $\alpha = Int$, but ignore the balance factor of type Int.

A constructor of a lifted type has the signature $\forall\chi\ .\ \tau'_1\ \chi \to \cdots \to \tau'_n\ \chi \to \tau'_0\ \chi$ where χ marks the parametric components.

We can write the signature more perspicuously as $\forall\chi\ .\ (\tau'_1 \to' \cdots \to' \tau'_n \to' \tau'_0)\ \chi$, using the lifted function space:

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

As an example, here are variants of Nil' and Cons':

$$\begin{array}{l} \mathit{nil}' \ :: \forall\alpha'\ .\ \forall\chi\ .\ (\mathit{List}'\ \alpha')\ \chi \\ \mathit{nil}' \ = \mathit{Nil}' \\ \\ \mathit{cons}' :: \forall\alpha'\ .\ \forall\chi\ .\ (\alpha' \to' \mathit{List}'\ \alpha' \to' \mathit{List}'\ \alpha')\ \chi \\ \mathit{cons}' = \mathit{Fun}\ (\lambda x \to \mathit{Fun}\ (\lambda xs \to \mathit{Cons}'\ x\ xs)) \end{array}$$

☞ 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.

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

☞ 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.

$$
\begin{aligned}
&fromSpine' :: Spine'\ \varphi\ \alpha \to \varphi\ \alpha \\
&fromSpine'\ (Con'\ c) = c \\
&fromSpine'\ (f \diamond' x) \quad = fromSpine'\ f\ `app`\ val'\ x
\end{aligned}
$$

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.

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

The lifted spine view of φ is simply $Spine'\ \varphi$.

$$Spine' :: Type'\ \varphi \longrightarrow Type'\ (Spine'\ \varphi)$$

$$spine' :: Type'\ \varphi \longrightarrow View'\ \varphi$$
$$spine'\ a' = View'\ (Spine'\ a')\ (\lambda x \longrightarrow toSpine'\ (x :'\ a'))\ fromSpine'$$

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

$$\begin{aligned} size\ (x :'\ Spine'\ a') &= size\_\ x \\ size\ (x :'\ a') &= \mathbf{case}\ spine'\ a'\ \mathbf{of} \\ &\quad\ View'\ b'\ fromData\ toData \rightarrow size\ (fromData\ x :'\ b') \end{aligned}$$

The catch-all case applies the spine view: the argument x is converted to the structure type, on which $size$ is called recursively.

The implementation of $size\_$ is entirely straightforward: it traverses the spine summing up the sizes of the constructors arguments.

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

Recall: the generic size function does not work on the original, unlifted types as they are different from the lifted ones.

☞ However, both are closely related: if τ' is the lifted variant of τ, then

$$\tau'\ Id \cong \tau$$

☞ This relation only holds for Haskell 98 types, not for GADTs.

As an example, the functions $fromList\ In_{Id}$ and $toList\ out_{Id}$ exhibit the isomorphism between $[\,]$ and $List'\ Id$.

$$
\begin{array}{l}
fromList :: (\alpha \to \alpha'\ \chi) \to ([\alpha] \to List'\ \alpha'\ \chi) \\
fromList\ from\ Nil\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = Nil' \\
fromList\ from\ (Cons\ x\ xs) = Cons'\ (from\ x)\ (fromList\ from\ xs)
\end{array}
$$

$$
\begin{array}{l}
toList\ \ \ \ :: (\alpha'\ \chi \to \alpha) \to (List'\ \alpha'\ \chi \to [\alpha]) \\
toList\ to\ Nil'\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = Nil \\
toList\ to\ (Cons'\ x\ xs) = Cons\ (to\ x)\ (toList\ to\ xs)
\end{array}
$$

We can use the isomorphism to broaden the scope of generic functions to unlifted types. To this end we simply re-use the view mechanism.

$$spine'\ List = View'\ (List'\ Id)\ (fromList\ In_{Id})\ (toList\ out_{Id})$$

- 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 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 $Out_1, \ldots, Out_n$.

5.4 Sum of products

The sum-of-products view is inspired by the semantics of data types.

Consider the schematic form of a Haskell 98 **data** declaration.

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

The **data** construct combines several features in a single coherent form: type abstraction, n-ary disjoint sums, n-ary cartesian products and type recursion.

We have already the machinery in place to deal with type abstraction, type application and type recursion: using type reflection the type-level constructs are mapped onto value abstraction, value application and value recursion.

☞ It remains to model n-ary sums and n-ary products.

We reduce the n-ary constructs to binary sums and binary products.

$$\textbf{data}\ \mathit{Zero}$$

$$\textbf{data}\ \mathit{Unit} = \mathit{Unit}$$

$$\textbf{data}\ \alpha + \beta = \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta$$

$$\textbf{data}\ \alpha \times \beta = \mathit{Pair}\{\mathit{outl} :: \alpha, \mathit{outr} :: \beta\}$$

☞ Zero, the empty sum, is used for encoding data types with no constructors; Unit, the empty product, is used for encoding constructors with no arguments.

If a data type has more than two alternatives or a constructor more than two arguments, then the binary constructors '$+$' and '$\times$' are nested accordingly.

☞ With respect to the nesting there are several choices: we can use a right-deep or a left-deep nesting, a list-like nesting or a (balanced) tree-like nesting.

As usual, we add suitable constructors to the type of type representations.

$$
\begin{array}{ll}
\mathtt{0} & :: \mathit{Type}\ \mathit{Zero} \\
\mathtt{1} & :: \mathit{Type}\ \mathit{Unit} \\
(+) & :: \mathit{Type}\ \alpha \longrightarrow \mathit{Type}\ \beta \longrightarrow \mathit{Type}\ (\alpha + \beta) \\
(\times) & :: \mathit{Type}\ \alpha \longrightarrow \mathit{Type}\ \beta \longrightarrow \mathit{Type}\ (\alpha \times \beta)
\end{array}
$$

The view function for the sum-of-products view is more elaborate than the one for the spine view as each data type has a tailor-made structure type.

```
open structure :: Type α → View α
structure Bool = View (1 + 1) fromBool toBool
  where
  fromBool :: Bool → Unit + Unit
  fromBool False    = Inl Unit
  fromBool True     = Inr Unit

  toBool    :: Unit + Unit → Bool
  toBool (Inl Unit) = False
  toBool (Inr Unit) = True
```

```
structure (List a) = View (1 + a × List a) fromList toList
  where
  fromList :: [α] → Unit + α × [α]
  fromList Nil          = Inl Unit
  fromList (Cons x xs)  = Inr (Pair x xs)
  toList   :: Unit + α × [α] → [α]
  toList (Inl Unit)           = Nil
  toList (Inr (Pair x xs))    = Cons x xs
structure (Tree a) = View (1 + Tree a × a × Tree a) fromTree toTree
  where
  fromTree :: Tree α → Unit + Tree α × α × Tree α
  fromTree Empty              = Inl Unit
  fromTree (Node l x r)       = Inr (Pair l (Pair x r))
  toTree   :: Unit + Tree α × α × Tree α → Tree α
  toTree (Inl Unit)           = Empty
  toTree (Inr (Pair l (Pair x r))) = Node l x r
```

The sum-of-products view can be used for defining both consumers and producers.

$$
\begin{array}{lll}
memo :: Type\ \alpha \to (\alpha \to \nu) \to (\alpha \to \nu) & & \\
memo\ Int & f\ i & = f\ i \quad \text{-- no memoisation} \\
memo\ \mathbb{1} & f\ Unit & = f_{Unit} \\
\quad \textbf{where}\ f_{Unit} & & = f\ Unit \\
memo\ (a + b)\ f\ (Inl\ x) & & = f_{Inl}\ x \\
\quad \textbf{where}\ f_{Inl} & & = memo\ a\ (\lambda x \to f\ (Inl\ x)) \\
memo\ (a + b)\ f\ (Inr\ y) & & = f_{Inr}\ y \\
\quad \textbf{where}\ f_{Inr} & & = memo\ b\ (\lambda y \to f\ (Inr\ y)) \\
memo\ (a \times b)\ f\ (Pair\ x\ y) & & = (f_{Pair}\ x)\ y \\
\quad \textbf{where}\ f_{Pair} & & = memo\ a\ (\lambda x \to memo\ b\ (\lambda y \to f\ (Pair\ x\ y))) \\
memo\ a & f\ x & = f_{View}\ x \\
\quad \textbf{where}\ f_{View} & & = \textbf{case}\ structure\ a\ \textbf{of} \\
 & & \quad View\ b\ from\ to \to memo\ b\ (f\ .\ to)\ .\ from
\end{array}
$$

☞ The helper definitions f_{Unit}, f_{Inl}, f_{Inr}, f_{Pair} and f_{View} do not depend on the actual argument of f. Thus, once f is given, they can be readily computed.

- The sum-of-products view provides more information than the spine view as it represents the complete data type, not just a single constructor application.

  It is type-oriented; the spine view is value-oriented.

- Consequently, it can be used both for defining consumers and producers.

- The sum-of-products view is preferable when the generic function has to relate different elements of a data type, the paradigmatic example being ordering.

- The sum-of-products view in its original form is only applicable to Haskell 98 data types.

  However, using a similar technique as for the type spine view we can broaden its scope to generalised algebraic data types.

6. Conclusion

Support for generic programming consists of three essential ingredients:

- a type reflection mechanism: $Type$ data type, type classes, type-safe cast;
- a type representation: $Type$, $Type'$, $Type_\kappa$;
- a generic view on data: spine view, type spine view, lifted spine view, sum-of-products view.

For each dimension there are several choices.

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	–