Generic Programming in Haskell

Presented to the Lambda Luminaries

Sean Leather

Utrecht University

21 June 2012

What is Generic Programming?

The adjective "generic" is heavily overloaded.

- Java/C# generics
- C++ templates
- Ada generic packages

What is Generic Programming?

The goal is often the same.

A higher level of abstraction than "normally" available

The technique is also often the same.

Some form of parameterization and instantiation

Examples of Generic Programming

Java/C#:

```
public class Stack<T>
{
  public void push(T item) {...}
  public T pop() {...}
}
```

Examples of Generic Programming

C++ :

```
template<typename T, typename Compare>
T& min(T& a, T& b, Compare comp) {
  if (comp(b, a))
    return b;
  return a;
}
```

Generic Programming in Haskell

In other words:

- Java-style generics ≈ parametric polymorphism
- C++ templates ≈ ad-hoc polymorphism

In Haskell:

- Both forms already exist.
- We don't call them generics because they're native to the language.

Datatype-generic programming:

- Abstract over the *structure of a datatype*
- Also known as "polytypism" and "shape-/structure-polymorphism"

Datatypes

data D p = Alt_1 | Alt_2 Int p

A datatype can have:

- Parameters: type variables ($\geqslant 0$)
- Alternatives: unique constructors ($\geqslant 0$)
- Fields: types for each constructor ($\geqslant 0$)

Non-syntactic features:

- Recursion
- Nesting

There are other features of datatypes, but we will consider only the above as a foundation for looking at the structure.

Structure of Datatypes: Sums

First structural element: alternatives.

$\textbf{data}\ \mathsf{AltEx}_2 = \mathsf{A}_1\ \mathsf{Int} \mid \mathsf{A}_2\ \mathsf{Char}$

Note that the above is similar to a standard type:

$\textbf{data}\ \mathsf{Either}\ \mathsf{a}\ \mathsf{b} = \mathsf{Left}\ \mathsf{a} \mid \mathsf{Right}\ \mathsf{b}$

And we can, in fact, model AltEx_2 as:

$\textbf{type}\ \mathsf{AltEx}'_2 = \mathsf{Either}\ \mathsf{Int}\ \mathsf{Char}$

with the following "smart" constructors:

$\mathsf{a}_1 :: \mathsf{Int} \to \mathsf{AltEx}'_2$
$\mathsf{a}_1 = \mathsf{Left}$

$\mathsf{a}_2 :: \mathsf{Char} \to \mathsf{AltEx}'_2$
$\mathsf{a}_2 = \mathsf{Right}$

Structure of Datatypes: Sums

When talking about alternatives in structural sense, we often call them sums. `Either` is the basic binary sum type. For conciseness, we use this (identical) binary sum type:

$$\textbf{data}\ a :+: b = \mathsf{L}\ a \mid \mathsf{R}\ b$$

What about a type with <2 alternatives?

$$\textbf{data}\ \mathsf{AltEx}_3 = \mathsf{B}_1\ \mathsf{Int} \mid \mathsf{B}_2\ \mathsf{Char} \mid \mathsf{B}_3\ \mathsf{Float}$$

The simplest solution is to nest one binary sum inside another:

$$\textbf{type}\ \mathsf{AltEx}'_3 = \mathsf{Int} :+: (\mathsf{Char} :+: \mathsf{Float})$$

Note that:

$$\mathsf{b}_3 :: \mathsf{Float} \rightarrow \mathsf{AltEx}'_3$$
$$\mathsf{b}_3 = \mathsf{R} \circ \mathsf{R}$$

Structure of Datatypes: Products

Next: fields.

$\textbf{data}\ \mathsf{FldEx}_2 = \mathsf{FldEx}_2\ \mathsf{Int}\ \mathsf{Char}$

Again, note the similarity to a standard type, the pair:

$\textbf{data}\ (,)\ \mathsf{a}\ \mathsf{b} = (,)\ \mathsf{a}\ \mathsf{b}$

And again, we model FldEx_2 similarly:

$\textbf{type}\ \mathsf{FldEx}'_2 = (,)\ \mathsf{Int}\ \mathsf{Char}$

with the smart constructor:

$$\mathsf{fldEx}'_2 :: \mathsf{Int} \to \mathsf{Char} \to \mathsf{FldEx}'_2$$
$$\mathsf{fldEx}'_2 = (,)$$

Structure of Datatypes: Products

The pair type is the basic binary product type. For symmetry with sums, we will use the following type:

$\textbf{data}\ a :\times: b = a :\times: b$

And more than two fields...

$\textbf{data}\ \mathsf{FldEx}_3 = \mathsf{FldEx}_3\ \mathsf{Int}\ \mathsf{Char}\ \mathsf{Float}$

... are modeled by nested binary products:

$\textbf{type}\ \mathsf{FldEx}_3' = \mathsf{Int} :\times: (\mathsf{Char} :\times: \mathsf{Float})$

with the smart constructor:

$\mathsf{fldEx}_3' :: \mathsf{Int} \to \mathsf{Char} \to \mathsf{Float} \to \mathsf{FldEx}_3'$
$\mathsf{fldEx}_3'\ x\ y\ z = x :\times: (y :\times: z)$

Structure of Datatypes: Sums of Products

To "sum" it all up, recall the first datatype example:

$\textbf{data}\ \mathsf{D}\ \mathsf{p} = \mathsf{Alt}_1 \mid \mathsf{Alt}_2\ \mathsf{Int}\ \mathsf{p}$

We can define an identical type using the sum and product types we have just discussed:

$\textbf{type}\ \mathsf{Rep}_{\mathsf{D}}\ \mathsf{p} = \mathsf{U} :\!+\!: \mathsf{Int} :\!\times\!: \mathsf{p}$

Notes:

- We use the "unit" type $\textbf{data}\ \mathsf{U} = \mathsf{U}$ (identical to the standard type $()$) to represent an alternative without fields.
- $:\!+\!:$ is $\textbf{infixr}\ 5$, and $:\!\times\!:$ is $\textbf{infixr}\ 6$, so we can write $\mathsf{Rep}_{\mathsf{D}}$ naturally, without unnecessary parentheses.

Structure of Datatypes: Isomorphism

So, we think we can model datatypes. But how do we know $\mathsf{Rep_D}$ accurately models D?

We define an isomorphism: two total functions that convert between types.

$$
\begin{aligned}
&\mathsf{from_D} :: \mathsf{D}\ \mathsf{p} \to \mathsf{Rep_D}\ \mathsf{p}\\
&\mathsf{from_D}\ \mathsf{Alt}_1 \qquad\quad\ = \mathsf{L}\ \mathsf{U}\\
&\mathsf{from_D}\ (\mathsf{Alt}_2\ \mathsf{i}\ \mathsf{p}) \quad = \mathsf{R}\ (\mathsf{i} :\times: \mathsf{p})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{to_D} :: \mathsf{Rep_D}\ \mathsf{p} \to \mathsf{D}\ \mathsf{p}\\
&\mathsf{to_D}\quad (\mathsf{L}\ \mathsf{U}) \qquad\quad = \mathsf{Alt}_1\\
&\mathsf{to_D}\quad (\mathsf{R}\ (\mathsf{i} :\times: \mathsf{p})) = \mathsf{Alt}_2\ \mathsf{i}\ \mathsf{p}
\end{aligned}
$$

This allows us to convert terms between (1) the familiar datatype and (2) the structure representation used for generic operations.

Structure of Datatypes: Constructors

Oh, but there's one more thing...
You may have noticed the representation lacked any information about the constructors (e.g. the names).

That's easily repaired with another datatype:

$$\textbf{data}\ \mathsf{C}\ \mathsf{a} = \mathsf{C}\ \mathsf{String}\ \mathsf{a}$$

We modify the representation to store constructor names:

$$\begin{aligned}
&\textbf{type}\ \mathsf{Rep_D}\ \mathsf{p} = \mathsf{C}\ \mathsf{U}\ {:+:}\ \mathsf{C}\ (\mathsf{Int}\ {:\times:}\ \mathsf{p})\\
&\mathsf{from_D}\ \mathsf{Alt}_1 \qquad\ = \mathsf{L}\ (\mathsf{C}\ \texttt{"Alt1"}\ \mathsf{U})\\
&\mathsf{from_D}\ (\mathsf{Alt}_2\ \mathsf{i}\ \mathsf{p}) = \mathsf{R}\ (\mathsf{C}\ \texttt{"Alt2"}\ (\mathsf{i}\ {:\times:}\ \mathsf{p}))
\end{aligned}$$

We could also put additional metadata (e.g. fixity) into C.

Generic Functions

Okay, so we have a structure representation. But what can we *do* with it?

Generic functions

- Defined on each possible case of the structure representation
- Work for every datatype that has an isomorphism with a structure representation

Example: $\text{show} :: a \rightarrow \text{String}$

Generic Functions: show

We define a show function for each possible structure case.

Unit:

$$\text{show}_U :: U \to \text{String}$$
$$\text{show}_U\ U = \texttt{""}$$

Constructor name:

$$\text{show}_C :: (a \to \text{String}) \to C\ a \to \text{String}$$
$$\text{show}_C\ \text{show}_a\ (C\ nm\ a) =$$
$$\quad \texttt{"("} + nm + \texttt{" "} + \text{show}_a\ a + \texttt{")"}$$

Binary product:

$$\text{show}_\times :: (a \to \text{String}) \to (b \to \text{String}) \to a :\times: b \to \text{String}$$
$$\text{show}_\times\ \text{show}_a\ \text{show}_b\ (a :\times: b) = \text{show}_a\ a + \texttt{" "} + \text{show}_b\ b$$

Binary sum:

$$\text{show}_+ :: (a \to \text{String}) \to (b \to \text{String}) \to a :+: b \to \text{String}$$
$$\text{show}_+\ \text{show}_a\ \_\ (L\ a) = \text{show}_a\ a$$
$$\text{show}_+\ \_\ \text{show}_b\ (R\ b) = \text{show}_b\ b$$

Generic Functions: show

We can define a show function for $\mathsf{Rep_D}$ (assuming $\mathsf{show_{Int}}$):

$$\mathsf{show_{Rep_D}} :: (\mathsf{p} \to \mathsf{String}) \to \mathsf{Rep_D}\ \mathsf{p} \to \mathsf{String}$$
$$\mathsf{show_{Rep_D}}\ \mathsf{show_p} =$$
$$\quad \mathsf{show_+}\ (\mathsf{show_C}\ \mathsf{show_U})\ (\mathsf{show_C}\ (\mathsf{show_\times}\ \mathsf{show_{Int}}\ \mathsf{show_p}))$$

The show function for D is just a hop away:

$$\mathsf{show_D} :: (\mathsf{p} \to \mathsf{String}) \to \mathsf{D}\ \mathsf{p} \to \mathsf{String}$$
$$\mathsf{show_D}\ \mathsf{show_p} = \mathsf{show_{Rep_D}}\ \mathsf{show_p} \circ \mathsf{from_D}$$

Generic Functions: show

$\text{show}_{\text{Rep}_\text{D}} :: (\text{p} \rightarrow \text{String}) \rightarrow \text{Rep}_\text{D}\ \text{p} \rightarrow \text{String}$
$\text{show}_{\text{Rep}_\text{D}}\ \text{show}_\text{p} =$
$\quad \text{show}_+\ (\text{show}_\text{C}\ \text{show}_\text{U})\ (\text{show}_\text{C}\ (\text{show}_\times\ \text{show}_{\text{Int}}\ \text{show}_\text{p}))$

Some observations:

- This is a sort of predictable pattern (or recipe) for defining show functions on structure representations.
- The functions are recursive but not in the usual way because the argument types differ.
- Each datatype can have a unique structure representation, and we want to support all combinations, *generically*.

Generic Functions, Generically

In order to jump into "true" genericity (where the structure is a parameter instead of a pattern), we need several addtional things:

- Polymorphic recursion – functions with a common scheme that reference each other and allow types to change in the calls

$$
\begin{array}{llll}
show_U :: & U & \to String \\
show_C :: \ldots \Rightarrow & C\ a & \to String \\
show_+ :: \ldots \Rightarrow & a :+: b & \to String \\
\ \ \ldots & & &
\end{array}
$$

- A common encoding for isomorphisms

$$
\begin{array}{ll}
\textbf{data}\ T \quad = \ldots & \text{-- User-defined datatype} \\
\textbf{type}\ Rep_T = \ldots & \text{-- Structure representation} \\
from\ \ :: T \to Rep_T & \\
to\ \ \ \ \ :: Rep_T \to T &
\end{array}
$$

Polymorphic Recursion

There are several ways to encode polymorphic recursion. We will use type classes.

- Standard classes already use polymorphic recursion for deriving instances: `Show`, `Eq`, etc.
- The class declaration specifies the type signature.
- Each recursive case is specified by an instance of the class.

A simplified definition of the `Show` class:

```
class Show a where
  show :: a → String
```

Polymorphic Recursion

The instances for each structure representation case:

Unit:

$$\textbf{instance}\ \mathsf{Show\ U}\ \textbf{where}$$
$$\quad \mathsf{show} = \mathsf{show}_{\mathsf{U}}$$

Constructor name:

$$\textbf{instance}\ \mathsf{Show\ a} \Rightarrow \mathsf{Show\ (C\ a)}\ \textbf{where}$$
$$\quad \mathsf{show} = \mathsf{show}_{\mathsf{C}}\ \mathsf{show}$$

Binary product:

$$\textbf{instance}\ (\mathsf{Show\ a}, \mathsf{Show\ b}) \Rightarrow \mathsf{Show\ (a} :\times: \mathsf{b)}\ \textbf{where}$$
$$\quad \mathsf{show} = \mathsf{show}_{\times}\ \mathsf{show\ show}$$

Binary sum:

$$\textbf{instance}\ (\mathsf{Show\ a}, \mathsf{Show\ b}) \Rightarrow \mathsf{Show\ (a} :+: \mathsf{b)}\ \textbf{where}$$
$$\quad \mathsf{show} = \mathsf{show}_{+}\ \mathsf{show\ show}$$

Polymorphic Recursion

Now, recall $\mathsf{show}_{\mathsf{Rep_D}}$:

$$\begin{aligned}&\mathsf{show}_{\mathsf{Rep_D}} :: (\mathsf{p} \rightarrow \mathsf{String}) \rightarrow \mathsf{Rep_D}\ \mathsf{p} \rightarrow \mathsf{String}\\ &\mathsf{show}_{\mathsf{Rep_D}}\ \mathsf{show_p} =\\ &\quad \mathsf{show}_{+}\ (\mathsf{show_C}\ \mathsf{show_U})\ (\mathsf{show_C}\ (\mathsf{show}_{\times}\ \mathsf{show_{Int}}\ \mathsf{show_p}))\end{aligned}$$

Compare to the new version that is now possible:

$$\begin{aligned}&\mathsf{show}'_{\mathsf{Rep_D}} :: \mathsf{Show}\ \mathsf{p} \Rightarrow \mathsf{Rep_D}\ \mathsf{p} \rightarrow \mathsf{String}\\ &\mathsf{show}'_{\mathsf{Rep_D}} = \mathsf{show}\end{aligned}$$

Encoding Isomorphisms

To define the show function for D, we still need to define another function:

$$\mathrm{show}'_{\mathrm{D}} :: \mathrm{Show\ p} \Rightarrow \mathrm{D\ p} \rightarrow \mathrm{String}$$
$$\mathrm{show}'_{\mathrm{D}} = \mathrm{show}'_{\mathrm{Rep_D}} \circ \mathrm{from_D}$$

Next goal:

- Define one show function that knows how to convert any type T to its structure representation type $\mathrm{Rep_T}$, given an isomorphism between T and $\mathrm{Rep_T}$.

Encoding Isomorphisms

We define a class of function pairs.

- We again use a type class, but with the addition of a *type family*.
- Each function pair implements an isomorphism between a datatype T and its structure representation $\mathsf{Rep}_{\mathsf{T}}$:

$$\mathsf{from} :: \mathsf{T} \rightarrow \mathsf{Rep}_{\mathsf{T}} \qquad\qquad \mathsf{to} :: \mathsf{Rep}_{\mathsf{T}} \rightarrow \mathsf{T}$$

- Each requires two types, so each instance must have two types (unlike the Show instances which needed only the structure representation type).
- $\mathsf{Rep}_{\mathsf{T}}$ is precisely determined by T, so really we only need one unique type and a second type derivable from the first.
- In this case, a (1) multiparameter type class with a functional dependency and a (2) type class with a type family are equally expressive. (It's a matter of taste, really.)

Encoding Isomorphisms

The type class:

```
class Generic a where
  type Rep a
  from :: a → Rep a
  to   :: Rep a → a
```

- `Rep` is a type family or, more precisely, an associated type synonym.
- Think of `Rep` as a function on types. Given a unique type (index) `T`, you get a type (synonym) `Rep T`.
- Note that `Rep T` need not be different from `Rep U` even though `T` and `U` are different.
- Concretely: two datatypes may have the same representation.

Encoding Isomorphisms

We need Generic instances for every datatype that we want to use with generic functions.

The instance for D uses definitions that we've already seen:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Generic}\ (\mathsf{D}\ \mathsf{p})\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{Rep}\ (\mathsf{D}\ \mathsf{p}) = \mathsf{Rep}_{\mathsf{D}}\ \mathsf{p} \\
\quad \mathsf{from} = \mathsf{from}_{\mathsf{D}} \\
\quad \mathsf{to} \quad\;\; = \mathsf{to}_{\mathsf{D}}
\end{array}
$$

- Other instances are defined similarly.
- In fact, Rep T, from, and to are precisely determined by the definition of T, so these instances can be automatically generated (e.g. using Template Haskell or a preprocessor).

The Generic show Function

Finally:

$$
\begin{aligned}
&\mathsf{gshow} :: (\mathsf{Show}\ (\mathsf{Rep}\ a), \mathsf{Generic}\ a) \Rightarrow a \rightarrow \mathsf{String} \\
&\mathsf{gshow} = \mathsf{show} \circ \mathsf{from}
\end{aligned}
$$

GP in General

- Datatype-generic programming:
  - Datatype is the parameter
  - Instantiation gives you a large class of generic functions
- Many generic functions:
  - Pretty-printing (`show`) and parsing (`read`)
  - Compression, serialization, and the reverse
  - Comparison, equality
  - Folds (catamorphisms), unfolds (anamorphisms), maps, zips, zippers
  - Traversals, updates, queries
- Many different libraries:
  - Instant Generics – presented here
  - Generic Deriving – GHC $\geqslant$ 7.2, similar to Instant Generics
  - EMGM – maintained by me
  - Regular – folds, etc.
  - Multirec – mutually recursive datatypes, folds, etc.
  - Scrap Your Boilerplate (SYB) – GHC, traversals, queries
  - ...

A Different Application

- Now, I want to present a different approach to using the structure of datatypes.
- This applies to the HollingBerries specification.
- The problem we look at is `printf`.

Presenting printf

In C and related languages, we have a function such as:

```
int printf(const char *format, ...)
```

The following code works:

```
printf("%s W%drld!\n", "Hello", 0); // Hello W0rld!
```

But so does the following:

```
printf("%s W%drld!\n", 0, "Hello"); // (null) W134514152rld!
```

As does this:

```
printf("%s W%drld!\n", "Hello"); // Hello W134514152rld!
```

And we can't do this (where `Y` is a new format descriptor):

```
printf("%Y\n", x); // Y
```

Problems With `printf`

- No type-checking of arguments
- No arity check
- Restricted set of format descriptors
  - Not extensible

Text.Printf

Is this the solution in Haskell?

printf :: PrintfType r ⇒ String → r

- From `Text.Printf` in `base`
- Abstract type class `PrintfType`
- `String` format descriptor

Try it in GHCi:

```
ghci> printf "%s W%drld!\n" "Hello" 0

ghci> printf "%s W%drld!\n" 0 "Hello"

ghci> printf "%s W%drld!\n" "Hello"
```

Apparently, `Text.Printf` is not the solution.

xformat

The solution:

$$\text{printf} :: \text{Format } f \Rightarrow f \to A\ (F\ f)\ (IO\ ())$$

- From `Text.XFormat.Show` in `xformat`
- `f` is the format descriptor
- `A` and `F` are type families
- Also: $\text{showf} :: \text{Format } f \Rightarrow f \to A\ (F\ f)\ \text{String}$
- `IO ()` (or `String`) is the result type

Try it in GHCi:

```
ghci> printf (String, " W", Int, "rld!\n") "Hello" 0

ghci> printf (String, " W", Int, "rld!\n") 0 "Hello"

ghci> printf (String, " W", Int, "rld!\n") "Hello"
```

Application to HollingBerries

Text.Printf :

```
printf "R%8.2f%d/%02d/%02d%-31s\n" price year month day desc
```

Text.XFormat.Show :

```
priceF = ("R", fillL 8 (Prec 2))
```

```
dateF = Int % "/" % zero 2 Int % "/" % zero 2 Int
```

```
descF = fillR' 31 String
```

```
printf (priceF, dateF, descF, "\n") price year month day desc
```

Advantages of XFormat

Just from this example, there are several obvious advantages:

- Type-safe: see types in GHCi
- Computable
  - Compute numbers (e.g. column widths) instead of strings
- Modular and composable
  - Reuse formats in different combinations

There is also a non-obvious advantage:

- Flexible
  - Use any `Real` for `Prec` (e.g. `Int`, `Float`, or `Double`)

Behind the Scenes

How does `xformat` work?

- Polyvariadic functions
  - Variable number of arguments (of different types)
- Format descriptors may...
  - Require an argument:

```
printf String "Sharp!"
```

  - Not allow any argument:

```
printf "Yebo."
```

  - Compose for multiple arguments (or not):

```
printf (String, " yebo ", String) "Ja" "yes!"
```

Polyvariadic Functions

- In the end, we construct a function, $T_1 \to \ldots \to T_n$, given a composition, $f_1 \diamond \ldots \diamond f_n$, of format descriptors.
- We track the argument (and result) type expected by each descriptor using functors.
- We combine multiple descriptors using functor composition.

Polyvariadic Functions

- Identity – no argument

```
newtype Id a = Id a
```

- Arrow – one argument

```
newtype Arr a b = Arr (a → b)
```

- Composition – combining

```
newtype (:○:) f g a = Comp (f (g a))
infixr 8 :○:
```

Polyvariadic Functions

The Functor instances:

```
instance Functor Id where
  fmap f (Id x) = Id (f x)
instance Functor (Arr a) where
  fmap f (Arr g) = Arr (f ∘ g)
instance (Functor f, Functor g) ⇒ Functor (f :○: g) where
  fmap f (Comp fga) = Comp (fmap (fmap f) fga)
```

Polyvariadic Functions

- The functors give us terms for the components of a function composition.
- We need a way to "lift" the functor to get the actual function type.

```
class Functor f ⇒ Apply f where
  type A f a
  apply :: f a → A f a
```

- `A` is the polyvariadic function type derived from the functor `f`.

Polyvariadic Functions

The Apply instances:

$$\textbf{instance}\ \mathsf{Apply\ Id}\ \textbf{where}$$
$$\quad \textbf{type}\ \mathsf{A\ Id\ a} = \mathsf{a}$$
$$\quad \mathsf{apply\ (Id\ x)} = \mathsf{x}$$

$$\textbf{instance}\ \mathsf{Apply\ (Arr\ a)}\ \textbf{where}$$
$$\quad \textbf{type}\ \mathsf{A\ (Arr\ a)\ b} = \mathsf{a} \rightarrow \mathsf{b}$$
$$\quad \mathsf{apply\ (Arr\ f)} = \mathsf{f}$$

$$\textbf{instance}\ (\mathsf{Apply\ f}, \mathsf{Apply\ g}) \Rightarrow \mathsf{Apply\ (f \mathbin{:\circ:} g)}\ \textbf{where}$$
$$\quad \textbf{type}\ \mathsf{A\ (f \mathbin{:\circ:} g)\ a} = \mathsf{A\ f\ (A\ g\ a)}$$
$$\quad \mathsf{apply\ (Comp\ fg)} = \mathsf{apply\ (fmap\ apply\ fg)}$$

Polyvariadic Functions

Resolve these types in GHCi (using `kind!` from $\geqslant$ 7.4):

apply (Id T1) :: A Id T1

apply (Arr (λT1 $\rightarrow$ T2)) :: A (Arr T1) T2

apply (Comp (Arr (λT1 $\rightarrow$ Arr (λT2 $\rightarrow$ T3)))) :: A (Arr T1 :○: Arr T2) T3

apply (Comp (Arr (λT1 $\rightarrow$ Id T2))) :: A (Arr T1 :○: Id) T2
apply (Comp (Id (Arr (λT1 $\rightarrow$ T2)))) :: A (Id :○: Arr T1) T2

Format Functors

With the help of...

$(\diamond) :: (\mathrm{Functor}\ f, \mathrm{Functor}\ g) \Rightarrow f\ \mathrm{String} \rightarrow g\ \mathrm{String} \rightarrow (\circ)\ f\ g\ \mathrm{String}$
$f \diamond g = \mathrm{Comp}\ (\mathrm{fmap}\ (\lambda s \rightarrow \mathrm{fmap}\ (\lambda t \rightarrow s +\!\!+ t)\ g)\ f)$
infixr $8 \diamond$

... we can easily compose functors, ...

$\mathrm{wrldF} :: \mathrm{Show}\ a \Rightarrow (\mathrm{Arr}\ \mathrm{String} \circ \mathrm{Id} \circ \mathrm{Arr}\ a \circ \mathrm{Id})\ \mathrm{String}$
$\mathrm{wrldF} = \mathrm{Arr}\ \mathrm{id} \diamond \mathrm{Id}$ `"  W"` $\diamond\ \mathrm{Arr}\ \mathrm{show} \diamond \mathrm{Id}$ `"rld!\n"`

... lift the resulting functor to a function type, ...

$\mathrm{wrld} :: \mathrm{Show}\ a \Rightarrow \mathrm{String} \rightarrow a \rightarrow \mathrm{String}$
$\mathrm{wrld} = \mathrm{apply}\ \mathrm{wrldF}$

... and greet the w0rld.

```
ghci> putStr $ wrld "Hello" 0
Hello W0rld!
```

Format Descriptors

But this isn't quite good enough.

We can go a step further by using meaningful descriptors instead of `Id`, `Arr`, and `Comp`.

Introducing the `Format` class:

$$\textbf{class}\ \mathsf{Apply}\ (\mathsf{F}\ \mathsf{f}) \Rightarrow \mathsf{Format}\ \mathsf{f}\ \textbf{where}$$
$$\quad \textbf{type}\ \mathsf{F}\ \mathsf{f} :: * \to *$$
$$\quad \mathsf{showf}' :: \mathsf{f} \to \mathsf{F}\ \mathsf{f}\ \mathsf{String}$$

And the `showf` function:

$$\mathsf{showf} :: \mathsf{Format}\ \mathsf{f} \Rightarrow \mathsf{f} \to \mathsf{A}\ (\mathsf{F}\ \mathsf{f})\ \mathsf{String}$$
$$\mathsf{showf} = \mathsf{apply} \circ \mathsf{showf}'$$

Format Descriptors

$$\textbf{class}\ \mathsf{Apply}\ (\mathsf{F}\ \mathsf{f}) \Rightarrow \mathsf{Format}\ \mathsf{f}\ \textbf{where}$$
$$\quad \textbf{type}\ \mathsf{F}\ \mathsf{f} :: * \to *$$
$$\quad \mathsf{showf}' :: \mathsf{f} \to \mathsf{F}\ \mathsf{f}\ \mathsf{String}$$

- Descriptors are instances of Format
- Some instances define primitives:

$$\textbf{instance}\ \mathsf{Format}\ \mathsf{String}\ \textbf{where}$$
$$\quad \textbf{type}\ \mathsf{F}\ \mathsf{String} = \mathsf{Id}$$
$$\quad \mathsf{showf}'\ \mathsf{s} = \mathsf{Id}\ \mathsf{s}$$

- Some instances define argument types:

$$\textbf{data}\ \mathsf{IntF} = \mathsf{Int}$$

$$\textbf{instance}\ \mathsf{Format}\ \mathsf{IntF}\ \textbf{where}$$
$$\quad \textbf{type}\ \mathsf{F}\ \mathsf{IntF} = \mathsf{Arr}\ \mathsf{Int}$$
$$\quad \mathsf{showf}'\ \mathsf{Int} = \mathsf{Arr}\ \mathsf{show}$$

Format Descriptors

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Apply}\ (\mathsf{F}\ \mathsf{f}) \Rightarrow \mathsf{Format}\ \mathsf{f}\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{F}\ \mathsf{f} :: * \rightarrow * \\
\quad \mathsf{showf}' :: \mathsf{f} \rightarrow \mathsf{F}\ \mathsf{f}\ \mathsf{String}
\end{array}
$$

- Some instances define recursive descriptors:

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{Format}\ \mathsf{f}_1, \mathsf{Format}\ \mathsf{f}_2) \Rightarrow \mathsf{Format}\ (\mathsf{f}_1, \mathsf{f}_2)\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{F}\ (\mathsf{f}_1, \mathsf{f}_2) = \mathsf{F}\ \mathsf{f}_1 :\circ: \mathsf{F}\ \mathsf{f}_2 \\
\quad \mathsf{showf}'\ (\mathsf{f}_1, \mathsf{f}_2) = \mathsf{showf}'\ \mathsf{f}_1 \diamond \mathsf{showf}'\ \mathsf{f}_2
\end{array}
$$

- And yet other instances are even more interesting. See code!

Format Descriptors

- Recall showf :

$$\text{showf} :: \text{Format f} \Rightarrow \text{f} \rightarrow \text{A (F f) String}$$
$$\text{showf} = \text{apply} \circ \text{showf}'$$

- How do we define printf ?
- Note that we cannot simply do $\text{putStr} \circ \text{showf}$:

```
ghci> :t putStr . showf
putStr . showf
  :: (Format a, A (F a) String ~ String) => a -> IO ()
```

- But F f is a Functor :

```
ghci> :t fmap putStr . showf'
fmap putStr . showf' :: Format a => a -> F a (IO ())
```

- So, we can now define printf :

$$\text{printf} :: \text{Format f} \Rightarrow \text{f} \rightarrow \text{A (F f) (IO ())}$$
$$\text{printf} = \text{apply} \circ \text{fmap putStr} \circ \text{showf}'$$

Summary of `xformat`

- This description is based on `Text.XFormat.Show`:
  - One difference: package uses the more efficient $\mathrm{String} \rightarrow \mathrm{String}$ instead of `String`.
- Also `Text.XFormat.Read`:

$$\mathrm{readf} :: \mathrm{Format}\ f \Rightarrow f \rightarrow \mathrm{String} \rightarrow \mathrm{Maybe}\ (\mathrm{R}\ f)$$

  - `R` is a type family that determines the structure of the result from the format descriptor `f`.
  - No functors involved. Simpler.

References

Generic Programming in Haskell:

- Johan Jeuring, Sean Leather, José Pedro Magalhães, Alexey Rodriguez Yakushev. *Libraries for Generic Programming in Haskell*. AFP 2008. pp. 165-229, 2009.
- Generic deriving: `http://www.haskell.org/haskellwiki/GHC.Generics`

`xformat`:

- Olivier Danvy. *Functional Unparsing*. Journal of Functional Programming. v8, N6, pp. 621-625. 1998.
- Ralf Hinze. *Formatting: A Class Act*. Journal of Functional Programming. v13, N5, pp. 935-944. 2003.
- Oleg Kiselyov: `http://okmij.org/ftp/typed-formatting/FPrintScan.html`
- More: `http://www.citeulike.org/user/spl/tag/printf`