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 $\approx$ parametric polymorphism
- C++ templates $\approx$ 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:

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

What about a type with < 2 alternatives?

$$\textbf{data}\ \mathsf{AltEx}_3 = B_1\ \mathsf{Int} \mid B_2\ \mathsf{Char} \mid B_3\ \mathsf{Float}$$

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

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

Note that:

$$b_3 :: \mathsf{Float} \rightarrow \mathsf{AltEx}'_3$$
$$b_3 = R \circ R$$

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:

```
data D p = Alt₁ | Alt₂ Int p
```

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

```
type Rep_D p = U :+: Int :×: p
```

Notes:

- We use the "unit" type `data U = U` (identical to the standard type `()`) to represent an alternative without fields.
- `:+:` is `infixr 5`, and `:×:` is `infixr 6`, so we can write `Rep_D` naturally, without unnecessary parentheses.

Structure of Datatypes: Isomorphism

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

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

$$
\begin{aligned}
&\mathsf{from}_{\mathsf{D}} :: \mathsf{D}\ \mathsf{p} \rightarrow \mathsf{Rep}_{\mathsf{D}}\ \mathsf{p}\\
&\mathsf{from}_{\mathsf{D}}\ \mathsf{Alt}_1 \qquad\quad = \mathsf{L}\ \mathsf{U}\\
&\mathsf{from}_{\mathsf{D}}\ (\mathsf{Alt}_2\ \mathsf{i}\ \mathsf{p}) \quad = \mathsf{R}\ (\mathsf{i} :\!\times\!: \mathsf{p})
\end{aligned}
$$

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

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:

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

Constructor name:

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

Binary product:

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

Binary sum:

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

Generic Functions: show

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

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

The show function for D is just a hop away:

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

Generic Functions: show

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

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

$$
\begin{array}{lll}
\text{show}_{\mathsf{U}} :: & \mathsf{U} & \to \text{String} \\
\text{show}_{\mathsf{C}} :: \ldots \Rightarrow & \mathsf{C}\ a & \to \text{String} \\
\text{show}_{+} :: \ldots \Rightarrow & a :+: b & \to \text{String} \\
\quad \ldots & &
\end{array}
$$

- A common encoding for isomorphisms

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{T} & = \ldots & \text{-- User-defined datatype} \\
\textbf{type}\ \text{Rep}_{\mathsf{T}} & = \ldots & \text{-- Structure representation} \\
\text{from} & :: \mathsf{T} \to \text{Rep}_{\mathsf{T}} & \\
\text{to} & :: \text{Rep}_{\mathsf{T}} \to \mathsf{T} &
\end{array}
$$

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:

instance Show U **where**
  show $=$ show_{U}

Constructor name:

instance Show a $\Rightarrow$ Show (C a) **where**
  show $=$ show_{C} show

Binary product:

instance (Show a, Show b) $\Rightarrow$ Show (a :$\times$: b) **where**
  show $=$ $\text{show}_{\times}$ show show

Binary sum:

instance (Show a, Show b) $\Rightarrow$ Show (a :+: b) **where**
  show $=$ show_{+} show show

Polymorphic Recursion

Now, recall $\mathsf{show}_{\mathsf{Rep}_\mathsf{D}}$:

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

Compare to the new version that is now possible:

$$\mathsf{show}'_{\mathsf{Rep}_\mathsf{D}} :: \mathsf{Show}\ \mathsf{p} \Rightarrow \mathsf{Rep}_\mathsf{D}\ \mathsf{p} \to \mathsf{String}$$
$$\mathsf{show}'_{\mathsf{Rep}_\mathsf{D}} = \mathsf{show}$$

Encoding Isomorphisms

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

$$\mathsf{show}'_{\mathsf{D}} :: \mathsf{Show}\ \mathsf{p} \Rightarrow \mathsf{D}\ \mathsf{p} \rightarrow \mathsf{String}$$
$$\mathsf{show}'_{\mathsf{D}} = \mathsf{show}'_{\mathsf{Rep}_{\mathsf{D}}} \circ \mathsf{from}_{\mathsf{D}}$$

Next goal:

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

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 Rep_T:

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

- Each requires two types, so each instance must have two types (unlike the Show instances which needed only the structure representation type).
- Rep_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:

$$
\begin{aligned}
&\textbf{instance}\ \mathrm{Generic}\ (\mathrm{D}\ \mathrm{p})\ \textbf{where}\\
&\quad \textbf{type}\ \mathrm{Rep}\ (\mathrm{D}\ \mathrm{p}) = \mathrm{Rep}_{\mathrm{D}}\ \mathrm{p}\\
&\quad \mathrm{from} = \mathrm{from}_{\mathrm{D}}\\
&\quad \mathrm{to} \quad\;\; = \mathrm{to}_{\mathrm{D}}
\end{aligned}
$$

- 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:

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

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:

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

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

```
instance Apply Id where
  type A Id a = a
  apply (Id x) = x
```

```
instance Apply (Arr a) where
  type A (Arr a) b = a → b
  apply (Arr f) = f
```

```
instance (Apply f, Apply g) ⇒ Apply (f :○: g) where
  type A (f :○: g) a = A f (A g a)
  apply (Comp fg) = 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...

$(\diamond) :: (\mathsf{Functor}\ f, \mathsf{Functor}\ g) \Rightarrow f\ \mathsf{String} \to g\ \mathsf{String} \to (\circledcirc)\ f\ g\ \mathsf{String}$
$f \diamond g = \mathsf{Comp}\ (\mathsf{fmap}\ (\lambda s \to \mathsf{fmap}\ (\lambda t \to s + t)\ g)\ f)$
infixr $8 \diamond$

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

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

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

$\mathsf{wrld} :: \mathsf{Show}\ a \Rightarrow \mathsf{String} \to a \to \mathsf{String}$
$\mathsf{wrld} = \mathsf{apply}\ \mathsf{wrldF}$

... 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:

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

And the `showf` function:

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

Format Descriptors

```
class Apply (F f) ⇒ Format f where
  type F f :: * → *
  showf′ :: f → F f String
```

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

```
instance Format String where
  type F String = Id
  showf′ s = Id s
```

- Some instances define argument types:

```
data IntF = Int

instance Format IntF where
  type F IntF = Arr Int
  showf′ Int = Arr show
```

Format Descriptors

```
class Apply (F f) ⇒ Format f where
  type F f :: ∗ → ∗
  showf′ :: f → F f String
```

- Some instances define recursive descriptors:

```
instance (Format f₁, Format f₂) ⇒ Format (f₁, f₂) where
  type F (f₁, f₂) = F f₁ :∘: F f₂
  showf′ (f₁, f₂) = showf′ f₁ ⋄ showf′ f₂
```

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

Format Descriptors

- Recall showf :

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

- 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 :

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

Summary of `xformat`

- This description is based on `Text.XFormat.Show`:
  - One difference: package uses the more efficient `String → String` instead of `String`.
- Also `Text.XFormat.Read`:

$$\text{readf} :: \text{Format } f \Rightarrow f \to \text{String} \to \text{Maybe } (R\ f)$$

  - `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`