

# Dissecting Different Flavors of Generic Programming in One Haskell Universe

Presented to Galois

Sean Leather

Utrecht University

August 27, 2013

# What is Generic Programming?

In programming languages, 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 = Alt1 | Alt2 Int p
```

A datatype can have:

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

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.

```
data AltEx2 = A1 Int | A2 Char
```

Note that the above is similar to a standard type:

```
data Either a b = Left a | Right b
```

And we can, in fact, model `AltEx2` as:

```
type AltEx'2 = Either Int Char
```

with the following “smart” constructors:

```
a1 :: Int → AltEx'2  
a1 = Left
```

```
a2 :: Char → AltEx'2  
a2 = 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:

```
data a :+: b = L a | R b
```

What about a type with  $< 2$  alternatives?

```
data AltEx3 = B1 Int | B2 Char | B3 Float
```

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

```
type AltEx'3 = Int :+: (Char :+: Float)
```

Note that:

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

# Structure of Datatypes: Products

Next: fields.

```
data FldEx2 = FldEx2 Int Char
```

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

```
data (,) a b = (,) a b
```

And again, we model `FldEx2` similarly:

```
type FldEx'2 = (,) Int Char
```

with the smart constructor:

```
fldEx'2 :: Int → Char → FldEx'2  
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:

```
data a × b = a × b
```

And more than two fields...

```
data FldEx3 = FldEx3 Int Char Float
```

... are modeled by nested binary products:

```
type FldEx'3 = Int × (Char × Float)
```

with the smart constructor:

```
fldEx'3 :: Int → Char → Float → FldEx'3  
fldEx'3 x y z = x × (y × 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 RepD 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 **infix** 5, and `:×:` is **infix** 6, so we can write `RepD` naturally, without unnecessary parentheses.

# Structure of Datatypes: Isomorphism

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

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

```
fromD :: D p → RepD p
fromD Alt1          = L U
fromD (Alt2 i p)    = R (i :: p)

toD :: RepD p → D p
toD (L U)           = Alt1
toD (R (i :: p))    = Alt2 i p
```

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:

```
data C a = C String a
```

We modify the representation to store constructor names:

```
type RepD p = C U :+: C (Int :×: p)
fromD Alt1      = L (C "Alt1" U)
fromD (Alt2 i p) = R (C "Alt2" (i :×: p))
```

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: `show :: a → String`

# Generic Functions: `show`

We define a `show` function for each possible structure case.

Unit:

```
showU :: U → String
showU U = ""
```

Constructor name:

```
showC :: (a → String) → C a → String
showC showa (C nm a) =
  "(" ++ nm ++ " " ++ showa a ++ ")"
```

Binary product:

```
show× :: (a → String) → (b → String) → a :× b → String
show× showa showb (a :× b) = showa a ++ " " ++ showb b
```

Binary sum:

```
show+ :: (a → String) → (b → String) → a :+ b → String
show+ showa _ (L a) = showa a
show+ _ showb (R b) = showb b
```



# Generic Functions: `show`

We can define a `show` function for `RepD` (assuming `showInt`):

```
showRepD :: (p → String) → RepD p → String
showRepD showp =
    show+ (showC showU) (showC (show× showInt showp))
```

The `show` function for `D` is just a hop away:

```
showD :: (p → String) → D p → String
showD showp = showRepD showp ∘ fromD
```

# Generic Functions: `show`

```
showRepD :: (p → String) → RepD p → String  
showRepD showp =  
  show+ (showC showU) (showC (show× showInt showp))
```

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 additional things:

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

```
showU ::      U      → String
showC :: ... ⇒ C a    → String
show+ :: ... ⇒ a :+: b → String
...
```

- A common encoding for isomorphisms

```
data T      = ...  -- User-defined datatype
type RepT = ...  -- Structure representation
from :: T → RepT
to    :: RepT → T
```

# 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 = showU
```

Constructor name:

```
instance Show a  $\Rightarrow$  Show (C a) where  
  show = showC show
```

Binary product:

```
instance (Show a, Show b)  $\Rightarrow$  Show (a  $\times$  b) where  
  show = show $\times$  show show
```

Binary sum:

```
instance (Show a, Show b)  $\Rightarrow$  Show (a  $+$  b) where  
  show = show+ show show
```

# Polymorphic Recursion

Now, recall `showRepD` :

```
showRepD :: (p → String) → RepD p → String
showRepD showp =
  show+ (showC showU) (showC (show× showInt showp))
```

Compare to the new version that is now possible:

```
show'RepD :: Show p ⇒ RepD p → String
show'RepD = show
```

# Encoding Isomorphisms

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

$$\begin{aligned} \text{show}'_D &:: \text{Show } p \Rightarrow D \text{ } p \rightarrow \text{String} \\ \text{show}'_D &= \text{show}'_{\text{Rep}_D} \circ \text{from}_D \end{aligned}$$

Next goal:

- Define one `show` function that knows how to convert any type `T` to its structure representation type `RepT`, given an isomorphism between `T` and `RepT`.

# 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  $\text{Rep}_T$ :

$\text{from} :: T \rightarrow \text{Rep}_T$

$\text{to} :: \text{Rep}_T \rightarrow T$

- Each requires two types, so each instance must have two types (unlike the `Show` instances which needed only the structure representation type).
- $\text{Rep}_T$  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:

```
instance Generic (D p) where  
  type Rep (D p) = RepD p  
  from = fromD  
  to   = toD
```

- 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:

```
gshow :: (Show (Rep a), Generic a) => a -> String
gshow = show o 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  $\geq$  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
  - ▶ ...

# 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>