# 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 similar.**

Some form of parameterization and instantiation

# Examples of Generic Programming

Java/C#:

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

In other words:

- Java-style generics ≈ parametric polymorphism

# 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;
}
```

In other words:

- C++ templates ≈ ad-hoc polymorphism

# Generic Programming in Haskell

"Generic programming":

- For other languages, the term tends to be used for late additions.
- Parametric and ad-hoc polymorphism were available in Haskell from the beginning.

In Haskell, we have come to use "generic programming" for datatype-generic programming (a.k.a. "polytypism" or "shape/structure polymorphism").

# Datatype-Generic Programming

What is datatype-generic programming?

- Parameterize a function over the *structure* of datatypes
- Instantiate the function with a particular type

The result is a function that

- **works with many types** (polymorphism) but
- **uses knowledge of the type** (unlike parametric) and
- **need not be redefined for every type** (unlike ad-hoc).

# Generic Functions

Applications

- Pretty-printing (e.g. `show`), parsing (e.g. `read`)
- Compression, serialization, marshalling (and their inverses)
- Comparison, equality
- (Co-)recursion, map, zip, zippers
- Traversals, queries, updates

# Generic Platforms

Many different implementations:

- Preprocessors:
  - PolyP
  - Generic Haskell
- Libraries
  - Scrap Your Boilerplate (SYB) – included with GHC for a long time
  - Extensible and Modular Generics for the Masses (EMGM)
  - Regular – recursion schemes
  - Multirec – mutually recursive datatypes
  - Generic Deriving – available in GHC $\geqslant$ 7.2, similar to Instant Generics
  - (and many, many more)

# Generic Flavors

The implementations can be grouped into flavors depending on how they view the structure of a datatype.

Some flavors (or views):

**Spine** A constructor is a sequence of types.
Example: SYB

**Sums-of-products** A datatype is a collection of alternative tuples of types.
Example: Generic Deriving

**Fixed point** A datatype is a sums-of-products with recursive points.
Example: Multirec

# Dissecting a Datatype: Sums-of-Products

**data** $\mathsf{T}_{\mathsf{sum}} = \mathsf{A}_1 \mid \mathsf{A}_2$

A datatype can have:

- Alternatives: unique constructors ($\geqslant 0$)

# Dissecting a Datatype: Sums-of-Products

**data** $\mathsf{T}_{\mathsf{prod}} = \mathsf{P}_2$ Char Int

A datatype can have:

- Fields: types for each constructor ($\geqslant 0$)

# Dissecting a Datatype: Sums-of-Products

Other features that are modeled:

- Constant types: each type in a field
- Parameters: type variables ($\geqslant 0$)

Features that are not modeled:

- Recursion
- Nesting (though it can be)

# Modeling a Sum

To model (nested) alternatives:

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

For syntactic elegance:

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

# Modeling a Product

To model (nested) fields:

**data** $(,)$ a b $=$ $(,)$ a b

For syntactic elegance:

**data** a $:\times:$ b $=$ a $:\times:$ b

# Modeling Other Structures

A constructor without fields:

```
data U = U
```

A constructor name:

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

A field type:

```
data K a = K a
```

Note: There are other features of datatypes, but we consider only the above.

# Modeling an Example

An example datatype:

$$\textbf{data}\ \mathsf{E\ a} = \mathsf{E_1} \mid \mathsf{E_2\ a\ (E\ a)\ Int}$$

The corresponding structure representation type:

$$\textbf{type}\ \mathsf{Rep_E\ a} = \mathsf{C\ U} :+: \mathsf{C\ (K\ a} :\times: \mathsf{K\ (E\ a)} :\times: \mathsf{K\ Int)}$$

Notes:

- $:+:$ is **infixr** $5$ and $:\times:$ is **infixr** $6$.
- Operators nest to the right.

# Converting Between Types: Isomorphism

- Generic functions work on the sums-of-products model.
- But first we need to convert between the model and the actual value of the datatype.
- We define an isomorphism: two total, dual functions.

$$
\begin{aligned}
&\mathsf{from}_{\mathsf{E}} :: \mathsf{E}\ \mathsf{a} \rightarrow \mathsf{Rep}_{\mathsf{E}}\ \mathsf{a} \\
&\mathsf{from}_{\mathsf{E}}\ \mathsf{E}_1 = \mathsf{L}\ (\mathsf{C}\ \texttt{"E1"}\ \mathsf{U}) \\
&\mathsf{from}_{\mathsf{E}}\ (\mathsf{E}_2\ \mathsf{x}\ \mathsf{e}\ \mathsf{i}) = \mathsf{R}\ (\mathsf{C}\ \texttt{"E2"}\ ((\mathsf{K}\ \mathsf{x}) :\times: (\mathsf{K}\ \mathsf{e}) :\times: (\mathsf{K}\ \mathsf{i})))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{to}_{\mathsf{E}} :: \mathsf{Rep}_{\mathsf{E}}\ \mathsf{a} \rightarrow \mathsf{E}\ \mathsf{a} \\
&\mathsf{to}_{\mathsf{E}}\ (\mathsf{L}\ (\mathsf{C}\ \texttt{"E1"}\ \mathsf{U})) = \mathsf{E}_1 \\
&\mathsf{to}_{\mathsf{E}}\ (\mathsf{R}\ (\mathsf{C}\ \texttt{"E2"}\ ((\mathsf{K}\ \mathsf{x}) :\times: (\mathsf{K}\ \mathsf{e}) :\times: (\mathsf{K}\ \mathsf{i})))) = \mathsf{E}_2\ \mathsf{x}\ \mathsf{e}\ \mathsf{i}
\end{aligned}
$$

# Converting Between Types: Isomorphism

For convenience, we join the representation type and isomorphism in a type class `Generic` with an associated type synonym `Rep`.

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

The instance for `E`:

```
instance Generic (E a) where
  type Rep (E a) = Rep_E a
  from = from_E
  to   = to_E
```

# Generic Functions

A generic function

- Is defined on each case of the structure representation and
- Works for every datatype that has a structure representation and isomorphism.

Example: $\text{show}_{\text{Rep a}} :: a \to \text{String}$

- We will define a show function for each case.

# Defining a Generic Function: show

Unit:

$$\mathsf{show_U} :: \mathsf{U} \rightarrow \mathsf{String}$$
$$\mathsf{show_U}\ \mathsf{U} = \texttt{""}$$

Constructor name:

$$\mathsf{show_C} :: (\mathsf{a} \rightarrow \mathsf{String}) \rightarrow \mathsf{C\ a} \rightarrow \mathsf{String}$$
$$\mathsf{show_C}\ \mathsf{show_a}\ (\mathsf{C\ nm\ a}) = \texttt{"("} + \!\!+ \mathsf{nm} + \!\!+ \texttt{" "} + \!\!+ \mathsf{show_a}\ \mathsf{a} + \!\!+ \texttt{")"}$$

Field:

$$\mathsf{show_K} :: (\mathsf{a} \rightarrow \mathsf{String}) \rightarrow \mathsf{K\ a} \rightarrow \mathsf{String}$$
$$\mathsf{show_K}\ \mathsf{show_a}\ (\mathsf{K\ a}) = \mathsf{show_a}\ \mathsf{a}$$

# Defining a Generic Function: show

Binary sum:

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

Binary product:

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

# Defining a Generic Function: show

Recall:

$$\textbf{type}\ \mathsf{Rep_E\ a = C\ U :\!+\!: C\ (K\ a :\!\times\!: K\ (E\ a) :\!\times\!: K\ Int)}$$

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

$$\begin{aligned}
\mathsf{show_{Rep_E}} &:: (\mathsf{a} \to \mathsf{String}) \to ((\mathsf{a} \to \mathsf{String}) \to \mathsf{E\ a} \to \mathsf{String}) \\
&\quad \to \mathsf{Rep_E\ a} \to \mathsf{String}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{show_{Rep_E}\ show_a\ show_E} = \\
&\quad \mathsf{show_+\ (show_C\ show_U)} \\
&\quad\quad \mathsf{(show_C\ (show_\times\ (show_K\ show_a)} \\
&\quad\quad\quad \mathsf{(show_\times\ (show_K\ (show_E\ show_a))\ (show_K\ show_{Int}))))}
\end{aligned}$$

# Defining a Generic Function: show

$$
\begin{aligned}
&\mathsf{show}_{\mathsf{Rep}_\mathsf{E}} :: (\mathsf{a} \to \mathsf{String}) \to ((\mathsf{a} \to \mathsf{String}) \to \mathsf{E\ a} \to \mathsf{String}) \\
&\qquad \to \mathsf{Rep}_\mathsf{E}\ \mathsf{a} \to \mathsf{String} \\
&\mathsf{show}_{\mathsf{Rep}_\mathsf{E}}\ \mathsf{show}_\mathsf{a}\ \mathsf{show}_\mathsf{E} = \\
&\quad \mathsf{show}_+\ (\mathsf{show}_\mathsf{C}\ \mathsf{show}_\mathsf{U}) \\
&\qquad (\mathsf{show}_\mathsf{C}\ (\mathsf{show}_\times\ (\mathsf{show}_\mathsf{K}\ \mathsf{show}_\mathsf{a}) \\
&\qquad\qquad (\mathsf{show}_\times\ (\mathsf{show}_\mathsf{K}\ (\mathsf{show}_\mathsf{E}\ \mathsf{show}_\mathsf{a}))\ (\mathsf{show}_\mathsf{K}\ \mathsf{show}_{\mathsf{Int}}))))
\end{aligned}
$$

The $\mathsf{show}_\mathsf{E}$ function itself is just an isomorphism away:

$$
\begin{aligned}
&\mathsf{show}_\mathsf{E} :: (\mathsf{a} \to \mathsf{String}) \to \mathsf{E\ a} \to \mathsf{String} \\
&\mathsf{show}_\mathsf{E}\ \mathsf{show}_\mathsf{a} = \mathsf{show}_{\mathsf{Rep}_\mathsf{E}}\ \mathsf{show}_\mathsf{a}\ \mathsf{show}_\mathsf{E} \circ \mathsf{from}_\mathsf{E}
\end{aligned}
$$

# Defining a Generic Function: show

$$
\begin{aligned}
\mathsf{show}_{\mathsf{Rep}_\mathsf{E}} &:: (\mathsf{a} \to \mathsf{String}) \to ((\mathsf{a} \to \mathsf{String}) \to \mathsf{E}\ \mathsf{a} \to \mathsf{String}) \\
&\quad \to \mathsf{Rep}_\mathsf{E}\ \mathsf{a} \to \mathsf{String}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{show}_{\mathsf{Rep}_\mathsf{E}}\ \mathsf{show}_\mathsf{a}\ \mathsf{show}_\mathsf{E} = \\
&\quad \mathsf{show}_+\ (\mathsf{show}_\mathsf{C}\ \mathsf{show}_\mathsf{U}) \\
&\qquad (\mathsf{show}_\mathsf{C}\ (\mathsf{show}_\times\ (\mathsf{show}_\mathsf{K}\ \mathsf{show}_\mathsf{a}) \\
&\qquad\qquad (\mathsf{show}_\times\ (\mathsf{show}_\mathsf{K}\ (\mathsf{show}_\mathsf{E}\ \mathsf{show}_\mathsf{a}))\ (\mathsf{show}_\mathsf{K}\ \mathsf{show}_{\mathsf{Int}}))))
\end{aligned}
$$

Some observations:

- This is **not** a generic function.
- It is defined on the structure of E, not on datatypes in general.
- It demonstrates a predictable pattern for defining the generic function.

# Defining a Generic Function: show

Consider these typical expressions and their types:

$$\mathrm{show}_{\mathrm{C}}\ \mathrm{show}_{\mathrm{U}} :: \mathrm{C}\ \mathrm{U} \rightarrow \mathrm{String}$$
$$\mathrm{show}_{\times}\ (\mathrm{show}_{\mathrm{K}}\ \mathrm{show}_{\mathrm{Int}})\ (\mathrm{show}_{\mathrm{K}}\ \mathrm{show}_{\mathrm{Char}}) :: (\mathrm{K}\ \mathrm{Int} :\times: \mathrm{K}\ \mathrm{Char}) \rightarrow \mathrm{String}$$

- $\mathrm{show}_{?}$ functions call other $\mathrm{show}_{?}$ functions.
- They can be considered recursive but not in the usual way.
- Polymorphic recursion – functions with different types that have a common scheme that reference each other

# Defining a Generic Function: show

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

- The class declaration specifies the type signature.
- Each recursive (type) case is specified by an instance of the class.

A simplified definition of the Show class:

**class** Show a **where**
  show :: a $\rightarrow$ String

# Defining a Generic Function: show

Some of the instances for each structure representation case:

Constructor name:

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

Binary sum:

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

The remaining instances are straightforward.

# Defining a Generic Function: show

Now, compare:

$$
\begin{aligned}
&\mathsf{show}_{\mathsf{Rep}_\mathsf{E}} :: (\mathsf{a} \to \mathsf{String}) \to ((\mathsf{a} \to \mathsf{String}) \to \mathsf{E}\ \mathsf{a} \to \mathsf{String}) \\
&\qquad \to \mathsf{Rep}_\mathsf{E}\ \mathsf{a} \to \mathsf{String}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{show}_{\mathsf{Rep}_\mathsf{E}}\ \mathsf{show}_\mathsf{a}\ \mathsf{show}_\mathsf{E} = \\
&\quad \mathsf{show}_+\ (\mathsf{show}_\mathsf{C}\ \mathsf{show}_\mathsf{U}) \\
&\qquad (\mathsf{show}_\mathsf{C}\ (\mathsf{show}_\times\ (\mathsf{show}_\mathsf{K}\ \mathsf{show}_\mathsf{a}) \\
&\qquad\qquad (\mathsf{show}_\times\ (\mathsf{show}_\mathsf{K}\ (\mathsf{show}_\mathsf{E}\ \mathsf{show}_\mathsf{a}))\ (\mathsf{show}_\mathsf{K}\ \mathsf{show}_{\mathsf{Int}}))))
\end{aligned}
$$

To:

$$
\begin{aligned}
&\mathsf{show}_{\mathsf{Rep}_\mathsf{E}} :: (\mathsf{Show}\ \mathsf{a}, \mathsf{Show}\ (\mathsf{E}\ \mathsf{a})) \Rightarrow \mathsf{Rep}_\mathsf{E}\ \mathsf{a} \to \mathsf{String} \\
&\mathsf{show}_{\mathsf{Rep}_\mathsf{E}} = \mathsf{show}
\end{aligned}
$$

# Defining a Generic Function: `show`

Finally, we can use a slightly different `Show` class to support generic functions for any type that has a representation.

```
class Show a where
  show :: a → String
  default show :: (Show (Rep a), Generic a) ⇒ a → String
  show = show ∘ from
```

- This uses default signatures: if type `a` has the instances `Show (Rep a)` and `Generic a`, then the given definition is used.

The instance for `E`:

```
instance Show a ⇒ Show (E a)
```

# ???

```
class Uniplate′ a r where
  descend′ :: (r → r) → a → a

instance Uniplate′ U a where
  descend′ _ U = U

instance Uniplate a ⇒ Uniplate′ (K a) a where
  descend′ f (K a) = K (f a)

instance Uniplate′ (K a) r where
  descend′ _ (K a) = K a

instance Uniplate′ a r ⇒ Uniplate′ (C a) r where
  descend′ f (C nm a) = C nm (descend′ f a)

instance (Uniplate′ a r, Uniplate′ b r) ⇒ Uniplate′ (a :+: b) r where
  descend′ f (L a) = L (descend′ f a)
  descend′ f (R b) = R (descend′ f b)

instance (Uniplate′ a r, Uniplate′ b r) ⇒ Uniplate′ (a :×: b) r where
  descend′ f (a :×: b) = descend′ f a :×: descend′ f b
```

# 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
  - ...

# 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`