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:

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

For syntactic elegance:

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

Modeling Other Structures

A constructor without fields:

$\textbf{data}\ \mathsf{U} = \mathsf{U}$

A constructor name:

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

A field type:

$\textbf{data}\ \mathsf{K}\ \mathsf{a} = \mathsf{K}\ \mathsf{a}$

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 $\textbf{infixr}\ 5$ and $:\times:$ is $\textbf{infixr}\ 6$.
- 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.

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

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

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: $\mathsf{show}_{\mathsf{Rep\ a}} :: \mathsf{a} \rightarrow \mathsf{String}$

- We will define a show function for each case.

Defining a Generic Function: show

Unit:

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

Constructor name:

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

Field:

$$\mathsf{show}_{\mathsf{K}} :: (\mathsf{a} \to \mathsf{String}) \to \mathsf{K}\ \mathsf{a} \to \mathsf{String}$$
$$\mathsf{show}_{\mathsf{K}}\ \mathsf{show}_{\mathsf{a}}\ (\mathsf{K}\ \mathsf{a}) = \mathsf{show}_{\mathsf{a}}\ \mathsf{a}$$

Defining a Generic Function: show

Binary sum:

$$\begin{array}{l} \mathrm{show}_{+} :: (a \to \mathrm{String}) \to (b \to \mathrm{String}) \to a :\!+\!: b \to \mathrm{String} \\ \mathrm{show}_{+}\ \mathrm{show}_{a}\ \_\ (L\ a) = \mathrm{show}_{a}\ a \\ \mathrm{show}_{+}\ \_\ \mathrm{show}_{b}\ (R\ b) = \mathrm{show}_{b}\ b \end{array}$$

Binary product:

$$\begin{array}{l} \mathrm{show}_{\times} :: (a \to \mathrm{String}) \to (b \to \mathrm{String}) \to a :\!\times\!: b \to \mathrm{String} \\ \mathrm{show}_{\times}\ \mathrm{show}_{a}\ \mathrm{show}_{b}\ (a :\!\times\!: b) = \mathrm{show}_{a}\ a +\!\!+ \texttt{"\ \ "} +\!\!+ \mathrm{show}_{b}\ b \end{array}$$

Defining a Generic Function: show

Recall:

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

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

$$\begin{aligned}
&\mathsf{show_{Rep_E}} :: (a \to \mathsf{String}) \to ((a \to \mathsf{String}) \to \mathsf{E}\ a \to \mathsf{String}) \\
&\qquad \to \mathsf{Rep_E}\ a \to \mathsf{String} \\
&\mathsf{show_{Rep_E}}\ \mathsf{show_a}\ \mathsf{show_E} = \\
&\quad \mathsf{show_+}\ (\mathsf{show_C}\ \mathsf{show_U}) \\
&\qquad (\mathsf{show_C}\ (\mathsf{show_\times}\ (\mathsf{show_K}\ \mathsf{show_a}) \\
&\qquad\qquad (\mathsf{show_\times}\ (\mathsf{show_K}\ (\mathsf{show_E}\ \mathsf{show_a}))\ (\mathsf{show_K}\ \mathsf{show_{Int}}))))
\end{aligned}$$

Defining a Generic Function: show

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

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

The show_E function itself is just an isomorphism away:

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

Defining a Generic Function: show

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

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

$$
\begin{aligned}
\mathsf{show}_{\mathsf{U}} &:: \ \mathsf{U} \qquad\ \ \rightarrow \mathsf{String} \\
\mathsf{show}_{\mathsf{C}} &:: \ldots \Rightarrow \mathsf{C\ a} \qquad \rightarrow \mathsf{String} \\
\mathsf{show}_{+} &:: \ldots \Rightarrow \mathsf{a} :\!+\!: \mathsf{b} \rightarrow \mathsf{String} \\
&\ldots
\end{aligned}
$$

- The show_... functions can be thought of as recursive but not in the usual way because the argument types differ.
- Polymorphic recursion – functions with different types that have a common scheme that reference each other

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:

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

Constructor name:

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

Binary product:

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

Binary sum:

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

Polymorphic Recursion

Now, recall show_Rep_D :

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

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

Compare to the new version that is now possible:

$$
\begin{aligned}
&\mathsf{show\_Rep\_D}' :: \mathsf{Show}\ p \Rightarrow \mathsf{Rep\_D}'\ p \to \mathsf{String} \\
&\mathsf{show\_Rep\_D}' = \mathsf{show}
\end{aligned}
$$

Encoding Isomorphisms

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

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

Next goal:

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

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 Rep_T:

  $$\mathrm{from} :: T \to \mathrm{Rep}_T \qquad \mathrm{to} :: \mathrm{Rep}_T \to T$$

- 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:

```
instance Generic (D p) where
  type Rep (D p) = Rep_D′ p
  from = from_D′
  to   = to_D′
```

- 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:

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

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
  - ...

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`