Dissecting Different Flavors of Generic Programming in One Haskell Universe

Presented to Galois

Sean Leather

Utrecht University

August 27, 2013

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

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

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:

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

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

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?

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

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

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

Note that:

$$b_3 :: \mathrm{Float} \to \mathrm{AltEx}'_3$$
$$b_3 = R \circ R$$

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:

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

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:

$\mathbf{data}\ \mathrm{D}\ p = \mathrm{Alt}_1 \mid \mathrm{Alt}_2\ \mathrm{Int}\ p$

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

$\mathbf{type}\ \mathrm{Rep}_{\mathrm{D}}\ p = \mathrm{U} :+: \mathrm{Int} :\times: p$

Notes:

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

Structure of Datatypes: Isomorphism

So, we think we can model datatypes. But how do we know Rep_D accurately models D?

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

$$
\begin{array}{lll}
\mathrm{from_D} :: \mathrm{D}\ \mathrm{p} \rightarrow \mathrm{Rep_D}\ \mathrm{p} & & \\
\mathrm{from_D}\ \mathrm{Alt}_1 & = & \mathrm{L}\ \mathrm{U} \\
\mathrm{from_D}\ (\mathrm{Alt}_2\ \mathrm{i}\ \mathrm{p}) & = & \mathrm{R}\ (\mathrm{i} :\times: \mathrm{p}) \\
\mathrm{to_D} :: \mathrm{Rep_D}\ \mathrm{p} \rightarrow \mathrm{D}\ \mathrm{p} & & \\
\mathrm{to_D}\ \ (\mathrm{L}\ \mathrm{U}) & = & \mathrm{Alt}_1 \\
\mathrm{to_D}\ \ (\mathrm{R}\ (\mathrm{i} :\times: \mathrm{p})) & = & \mathrm{Alt}_2\ \mathrm{i}\ \mathrm{p}
\end{array}
$$

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:

$\textbf{type}\ \mathsf{Rep}_{\mathsf{D}}\ \mathsf{p} = \mathsf{C}\ \mathsf{U} :+: \mathsf{C}\ (\mathsf{Int} :\times: \mathsf{p})$
$\mathsf{from}_{\mathsf{D}}\ \mathsf{Alt}_1 \qquad = \mathsf{L}\ (\mathsf{C}\ \texttt{"Alt1"}\ \mathsf{U})$
$\mathsf{from}_{\mathsf{D}}\ (\mathsf{Alt}_2\ \mathsf{i}\ \mathsf{p}) = \mathsf{R}\ (\mathsf{C}\ \texttt{"Alt2"}\ (\mathsf{i} :\times: \mathsf{p}))$

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: $\mathrm{show} :: a \rightarrow \mathrm{String}$

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:

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

Binary product:

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

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 $\mathsf{show_{Int}}$):

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

The show function for D is just a hop away:

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

Generic Functions: show

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

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

$$
\begin{aligned}
&\mathsf{show}_U :: && U && \to \mathsf{String}\\
&\mathsf{show}_C :: \ldots \Rightarrow && C\ a && \to \mathsf{String}\\
&\mathsf{show}_+ :: \ldots \Rightarrow && a :+: b && \to \mathsf{String}\\
&\quad \ldots
\end{aligned}
$$

- A common encoding for isomorphisms

$$
\begin{aligned}
&\mathbf{data}\ T && = \ldots && \text{-- User-defined datatype}\\
&\mathbf{type}\ Rep_T && = \ldots && \text{-- Structure representation}\\
&\mathsf{from} && :: T \to Rep_T\\
&\mathsf{to} && :: Rep_T \to T
\end{aligned}
$$

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 $\mathsf{show}_{\mathsf{Rep}_\mathsf{D}}$:

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

Compare to the new version that is now possible:

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

Encoding Isomorphisms

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

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

Next goal:

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

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 $\mathsf{Rep_T}$:

  $\mathsf{from} :: \mathsf{T} \to \mathsf{Rep_T}$ $\qquad$ $\mathsf{to} :: \mathsf{Rep_T} \to \mathsf{T}$

- Each requires two types, so each instance must have two types (unlike the Show instances which needed only the structure representation type).
- $\mathsf{Rep_T}$ 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) = $\mathrm{Rep}_{\mathrm{D}}$ p
  from = $\mathrm{from}_{\mathrm{D}}$
  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:

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

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
  - ...

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`