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.

```
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 AltEx_2 as:

```
type AltEx2' = Either Int Char
```

with the following "smart" constructors:

```
a1 :: Int -> AltEx2'
a1 = Left
```

```
a2 :: Char -> AltEx2'
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 AltEx3' = Int :+: (Char :+: Float)
```

Note that:

```
b3 :: Float → AltEx3'
b3 = R ∘ 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 FldEx2' = (,) Int Char
```

with the smart constructor:

```
fldEx2' :: Int → Char → FldEx2'
fldEx2' = (,)
```

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 FldEx3' = Int :×: (Char :×: Float)
```

with the smart constructor:

```
fldEx3' :: Int → Char → Float → FldEx3'
fldEx3' 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 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 Rep_D accurately models D?

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

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

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:

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

We modify the representation to store constructor names:

```
type Rep_D p = C U :+: C (Int :×: p)
from_D Alt1       = L (C "Alt1" U)
from_D (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: $\text{show} :: a \rightarrow \text{String}$

Generic Functions: show

We define a show function for each possible structure case.

Unit:

$$\begin{aligned}&\text{show}_U :: U \to \text{String}\\&\text{show}_U\ U = ""\end{aligned}$$

Constructor name:

$$\begin{aligned}&\text{show}_C :: (a \to \text{String}) \to C\ a \to \text{String}\\&\text{show}_C\ \text{show}_a\ (C\ nm\ a) =\\&\quad "("\ +\!\!+\ nm\ +\!\!+\ "\ \ "\ +\!\!+\ \text{show}_a\ a\ +\!\!+\ ")"\end{aligned}$$

Binary product:

$$\begin{aligned}&\text{show}_\times :: (a \to \text{String}) \to (b \to \text{String}) \to a :\times: b \to \text{String}\\&\text{show}_\times\ \text{show}_a\ \text{show}_b\ (a :\times: b) = \text{show}_a\ a\ +\!\!+\ "\ \ "\ +\!\!+\ \text{show}_b\ b\end{aligned}$$

Binary sum:

$$\begin{aligned}&\text{show}_+ :: (a \to \text{String}) \to (b \to \text{String}) \to a :+: b \to \text{String}\\&\text{show}_+\ \text{show}_a\ \_\ (L\ a) = \text{show}_a\ a\\&\text{show}_+\ \_\ \text{show}_b\ (R\ b) = \text{show}_b\ b\end{aligned}$$

Generic Functions: show

We can define a show function for Rep_D (assuming show_Int):

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

The show function for D is just a hop away:

$$
\begin{aligned}
&\text{show}_\text{D} :: (\text{p} \to \text{String}) \to \text{D}\ \text{p} \to \text{String} \\
&\text{show}_\text{D}\ \text{show}_\text{p} = \text{show}_{\text{Rep}_\text{D}}\ \text{show}_\text{p} \circ \text{from}_\text{D}
\end{aligned}
$$

Generic Functions: show

$$
\begin{aligned}
&\text{show}_{\text{Rep}_D} :: (p \to \text{String}) \to \text{Rep}_D\ p \to \text{String} \\
&\text{show}_{\text{Rep}_D}\ \text{show}_p = \\
&\quad \text{show}_+\ (\text{show}_C\ \text{show}_U)\ (\text{show}_C\ (\text{show}_\times\ \text{show}_{\text{Int}}\ \text{show}_p))
\end{aligned}
$$

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

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

- A common encoding for isomorphisms

$$
\begin{aligned}
&\textbf{data}\ T \quad = \ldots \quad \text{-- User-defined datatype} \\
&\textbf{type}\ \text{Rep}_T = \ldots \quad \text{-- Structure representation} \\
&\text{from} :: T \to \text{Rep}_T \\
&\text{to} \quad :: \text{Rep}_T \to T
\end{aligned}
$$

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:

```
instance Show U where
  show = showU
```

Constructor name:

```
instance Show a ⇒ Show (C a) where
  show = showC show
```

Binary product:

```
instance (Show a, Show b) ⇒ Show (a :×: b) where
  show = show× show show
```

Binary sum:

```
instance (Show a, Show b) ⇒ Show (a :+: b) where
  show = show+ show show
```

Polymorphic Recursion

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

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

Compare to the new version that is now possible:

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

Encoding Isomorphisms

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

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

Next goal:

- Define one `show` function that knows how to convert any type `T` to its structure representation type Rep_{T}, given an isomorphism between `T` and 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:

$$from :: T \rightarrow Rep_T \qquad to :: Rep_T \rightarrow T$$

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

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

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`