Dissecting Different Flavors of Generic Programming in One Haskell Universe

Presented to Galois

Sean Leather

Utrecht University

August 27, 2013

What is Generic Programming?

What is Generic Programming?

In programming languages, the adjective "generic" is heavily overloaded.

What is Generic Programming?

In programming languages, the adjective "generic" is heavily overloaded.

- Java/C# generics

What is Generic Programming?

In programming languages, the adjective "generic" is heavily overloaded.

- Java/C# generics
- C++ templates

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

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

Generic Programming in Haskell

In other words:

- Java-style generics $\approx$ parametric polymorphism
- C++ templates $\approx$ ad-hoc polymorphism

Generic Programming in Haskell

In other words:

- Java-style generics $\approx$ parametric polymorphism
- C++ templates $\approx$ ad-hoc polymorphism

Generic Programming in Haskell

In other words:

- Java-style generics ≈ parametric polymorphism
- C++ templates ≈ ad-hoc polymorphism

In Haskell:

- Both forms already exist.

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.

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.

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*

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

Datatypes

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

A datatype can have:

- Parameters: type variables ($\geqslant 0$)

Datatypes

data D p = Alt_1 | Alt_2 Int p

A datatype can have:

- Parameters: type variables ($\geqslant 0$)
- Alternatives: unique constructors ($\geqslant 0$)

Datatypes

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

A datatype can have:

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

Datatypes

$\textbf{data}\ D\ p = Alt_1 \mid Alt_2\ Int\ p$

A datatype can have:

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

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

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

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

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 $\mathsf{AltEx}_2 = \mathsf{A}_1\ \mathsf{Int} \mid \mathsf{A}_2\ \mathsf{Char}$

Structure of Datatypes: Sums

First structural element: alternatives.

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:

data $\mathsf{Either}\ a\ b = \mathsf{Left}\ a \mid \mathsf{Right}\ b$

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:

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

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?

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

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:

$$\textbf{data}\ a \mathrel{:+:} b = \mathsf{L}\ a \mid \mathsf{R}\ b$$

What about a type with < 2 alternatives?

$$\textbf{data}\ \mathsf{AltEx}_3 = \mathsf{B}_1\ \mathsf{Int} \mid \mathsf{B}_2\ \mathsf{Char} \mid \mathsf{B}_3\ \mathsf{Float}$$

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

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

Note that:

$$\mathsf{b}_3 :: \mathsf{Float} \rightarrow \mathsf{AltEx}'_3$$
$$\mathsf{b}_3 = \mathsf{R} \circ \mathsf{R}$$

Structure of Datatypes: Products

Next: fields.

```
data FldEx₂ = FldEx₂ Int Char
```

Structure of Datatypes: Products

Next: fields.

data $\mathsf{FldEx}_2 = \mathsf{FldEx}_2$ Int Char

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

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

Structure of Datatypes: Products

Next: fields.

$\mathbf{data}\ \mathsf{FldEx}_2 = \mathsf{FldEx}_2\ \mathsf{Int}\ \mathsf{Char}$

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

$\mathbf{data}\ (,)\ a\ b = (,)\ a\ b$

And again, we model FldEx_2 similarly:

$\mathbf{type}\ \mathsf{FldEx}'_2 = (,)\ \mathsf{Int}\ \mathsf{Char}$

with the smart constructor:

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

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
```

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...

$$\textbf{data}\ \mathsf{FldEx}_3 = \mathsf{FldEx}_3\ \mathsf{Int}\ \mathsf{Char}\ \mathsf{Float}$$

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:

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

Structure of Datatypes: Sums of Products

To "sum" it all up, recall the first datatype example:

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

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

$\textbf{type}\ \mathsf{Rep_D}\ \mathsf{p} = \mathsf{U}\ {:}{+}{:}\ \mathsf{Int}\ {:}{\times}{:}\ \mathsf{p}$

Structure of Datatypes: Sums of Products

To “sum” it all up, recall the first datatype example:

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

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

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

Notes:

- We use the “unit” type $\textbf{data}\ \mathrm{U} = \mathrm{U}$ (identical to the standard type `()`) to represent an alternative without fields.

Structure of Datatypes: Sums of Products

To "sum" it all up, recall the first datatype example:

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

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

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

Notes:

- We use the "unit" type $\textbf{data}\ \mathrm{U} = \mathrm{U}$ (identical to the standard type $()$) to represent an alternative without fields.
- $:+:$ is $\textbf{infixr}\ 5$, and $:\times:$ is $\textbf{infixr}\ 6$, 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 $\mathsf{Rep}_{\mathsf{D}}$ accurately models D?

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.

Structure of Datatypes: Isomorphism

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

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

$$
\begin{array}{lll}
\mathsf{from}_{\mathsf{D}} :: \mathsf{D}\ \mathsf{p} \rightarrow \mathsf{Rep}_{\mathsf{D}}\ \mathsf{p} & & \\
\mathsf{from}_{\mathsf{D}}\ \mathsf{Alt}_1 & = & \mathsf{L}\ \mathsf{U} \\
\mathsf{from}_{\mathsf{D}}\ (\mathsf{Alt}_2\ \mathsf{i}\ \mathsf{p}) & = & \mathsf{R}\ (\mathsf{i} :\times: \mathsf{p}) \\
\mathsf{to}_{\mathsf{D}} :: \mathsf{Rep}_{\mathsf{D}}\ \mathsf{p} \rightarrow \mathsf{D}\ \mathsf{p} & & \\
\mathsf{to}_{\mathsf{D}}\ \ (\mathsf{L}\ \mathsf{U}) & = & \mathsf{Alt}_1 \\
\mathsf{to}_{\mathsf{D}}\ \ (\mathsf{R}\ (\mathsf{i} :\times: \mathsf{p})) & = & \mathsf{Alt}_2\ \mathsf{i}\ \mathsf{p}
\end{array}
$$

Structure of Datatypes: Isomorphism

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

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

$$
\begin{aligned}
&\mathsf{from_D} :: \mathsf{D\ p} \rightarrow \mathsf{Rep_D\ p} \\
&\mathsf{from_D\ Alt_1} &&= \mathsf{L\ U} \\
&\mathsf{from_D\ (Alt_2\ i\ p)} &&= \mathsf{R\ (i :\!\times\!: p)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{to_D} :: \mathsf{Rep_D\ p} \rightarrow \mathsf{D\ p} \\
&\mathsf{to_D\ \ (L\ U)} &&= \mathsf{Alt_1} \\
&\mathsf{to_D\ \ (R\ (i :\!\times\!: p))} &&= \mathsf{Alt_2\ i\ p}
\end{aligned}
$$

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...

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).

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:

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

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:

$$\begin{aligned}
&\textbf{type}\ \mathsf{Rep_D}\ \mathsf{p} = \mathsf{C}\ \mathsf{U} :+: \mathsf{C}\ (\mathsf{Int} :\times: \mathsf{p})\\
&\mathsf{from_D}\ \mathsf{Alt}_1 \qquad\ \ = \mathsf{L}\ (\mathsf{C}\ \texttt{"Alt1"}\ \mathsf{U})\\
&\mathsf{from_D}\ (\mathsf{Alt}_2\ \mathsf{i}\ \mathsf{p}) = \mathsf{R}\ (\mathsf{C}\ \texttt{"Alt2"}\ (\mathsf{i} :\times: \mathsf{p}))
\end{aligned}$$

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:

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

We modify the representation to store constructor names:

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

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

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

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

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

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

Generic Functions: `show`

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

Generic Functions: show

We define a show function for each possible structure case.

Unit:

$\text{show}_{\text{U}} :: \text{U} \rightarrow \text{String}$
$\text{show}_{\text{U}}\ \text{U} = \texttt{""}$

Generic Functions: show

We define a show function for each possible structure case.

Unit:

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

Constructor name:

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

Generic Functions: show

We define a show function for each possible structure case.

Unit:

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

Constructor name:

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

Binary product:

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

Generic Functions: show

We define a show function for each possible structure case.

Unit:

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

Constructor name:

$$\mathsf{show}_C :: (a \to \mathsf{String}) \to C\ a \to \mathsf{String}$$
$$\mathsf{show}_C\ \mathsf{show}_a\ (C\ nm\ a) =$$
$$\quad \texttt{"("} + nm + \texttt{" "} + \mathsf{show}_a\ a + \texttt{")"}$$

Binary product:

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

Binary sum:

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

Generic Functions: show

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

$$\mathsf{show_{Rep_D}} :: (\mathsf{p} \rightarrow \mathsf{String}) \rightarrow \mathsf{Rep_D}\ \mathsf{p} \rightarrow \mathsf{String}$$
$$\mathsf{show_{Rep_D}}\ \mathsf{show_p} =$$
$$\quad \mathsf{show_+}\ (\mathsf{show_C}\ \mathsf{show_U})\ (\mathsf{show_C}\ (\mathsf{show_\times}\ \mathsf{show_{Int}}\ \mathsf{show_p}))$$

Generic Functions: show

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

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

The show function for D is just a hop away:

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

Generic Functions: show

$$\mathsf{show}_{\mathsf{Rep}_\mathsf{D}} :: (\mathsf{p} \rightarrow \mathsf{String}) \rightarrow \mathsf{Rep}_\mathsf{D}\ \mathsf{p} \rightarrow \mathsf{String}$$
$$\mathsf{show}_{\mathsf{Rep}_\mathsf{D}}\ \mathsf{show}_\mathsf{p} =$$
$$\quad \mathsf{show}_+\ (\mathsf{show}_\mathsf{C}\ \mathsf{show}_\mathsf{U})\ (\mathsf{show}_\mathsf{C}\ (\mathsf{show}_\times\ \mathsf{show}_\mathsf{Int}\ \mathsf{show}_\mathsf{p}))$$

Some observations:

- This is a sort of predictable pattern (or recipe) for defining show functions on structure representations.

Generic Functions: show

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

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.

Generic Functions: show

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

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:

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

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

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

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

- A common encoding for isomorphisms

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

$$
\begin{array}{llll}
\mathsf{show}_{\mathsf{U}} :: & & \mathsf{U} & \to \mathsf{String} \\
\mathsf{show}_{\mathsf{C}} :: & \ldots \Rightarrow & \mathsf{C}\ \mathsf{a} & \to \mathsf{String} \\
\mathsf{show}_{+} :: & \ldots \Rightarrow & \mathsf{a} :\!+\!: \mathsf{b} & \to \mathsf{String} \\
\quad \ldots & & &
\end{array}
$$

- A common encoding for isomorphisms

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

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

- A common encoding for isomorphisms

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

Polymorphic Recursion

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

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.

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.

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.

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.

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:

Polymorphic Recursion

The instances for each structure representation case:

Unit:

instance Show U **where**
  show = show_{U}

Polymorphic Recursion

The instances for each structure representation case:

Unit:

```
instance Show U where
  show = show_U
```

Constructor name:

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

Polymorphic Recursion

The instances for each structure representation case:

Unit:

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

Constructor name:

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

Binary product:

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

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:

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

Binary product:

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

Binary sum:

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

Polymorphic Recursion

Now, recall $\mathsf{show}_{\mathsf{Rep_D}}$:

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

Polymorphic Recursion

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

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

Compare to the new version that is now possible:

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

Encoding Isomorphisms

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

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

Encoding Isomorphisms

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

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

Next goal:

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

Encoding Isomorphisms

We define a class of function pairs.

- We again use a type class, but with the addition of a *type family*.

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 $\mathsf{Rep}_{\mathsf{T}}$:

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 $\mathsf{Rep_T}$:

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 $\mathsf{Rep}_{\mathsf{T}}$:

$$\mathsf{from} :: \mathsf{T} \to \mathsf{Rep}_{\mathsf{T}}$$

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 $\mathsf{Rep_T}$:

$$\mathsf{from} :: \mathsf{T} \rightarrow \mathsf{Rep_T}$$

$$\mathsf{to} :: \mathsf{Rep_T} \rightarrow \mathsf{T}$$

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).

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 $\mathsf{Rep_T}$:

  $\mathsf{from} :: \mathsf{T} \rightarrow \mathsf{Rep_T}$

  $\mathsf{to} :: \mathsf{Rep_T} \rightarrow \mathsf{T}$

- 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.

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:

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

- 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
```

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.

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`.

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.

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.

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}

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.

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:

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

GP in General

- Datatype-generic programming:

GP in General

- Datatype-generic programming:
  - Datatype is the parameter

GP in General

- Datatype-generic programming:
  - Datatype is the parameter
  - Instantiation gives you a large class of generic functions

GP in General

- Datatype-generic programming:
  - Datatype is the parameter
  - Instantiation gives you a large class of generic functions
- Many generic functions:

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`)

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

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

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

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

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:

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

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

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

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.

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.

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

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`