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 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() {...}
}
```

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;
}
```

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.

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.

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.

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

Datatype-Generic Programming

What is datatype-generic programming?

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

Datatype-Generic Programming

What is datatype-generic programming?

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

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

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

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`)

Generic Functions

Applications

- Pretty-printing (e.g. `show`), parsing (e.g. `read`)
- Compression, serialization, marshalling (and their inverses)

Generic Functions

Applications

- Pretty-printing (e.g. `show`), parsing (e.g. `read`)
- Compression, serialization, marshalling (and their inverses)
- Comparison, equality

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

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:

Generic Platforms

Many different implementations:

- Preprocessors:
  - PolyP

Generic Platforms

Many different implementations:

- Preprocessors:
  - PolyP
  - Generic Haskell

Generic Platforms

Many different implementations:

- Preprocessors:
  - PolyP
  - Generic Haskell
- Libraries

Generic Platforms

Many different implementations:

- Preprocessors:
  - PolyP
  - Generic Haskell
- Libraries
  - Scrap Your Boilerplate (SYB) – included with GHC for a long time

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)

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

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 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

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.

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

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

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 ($\geqslant 0$)

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

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 $(,)\ a\ b = (,)\ a\ b$

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

Modeling Other Structures

A constructor without fields:

```
data U = U
```

A constructor name:

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

Modeling Other Structures

A constructor without fields:

```haskell
data U = U
```

A constructor name:

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

A field type:

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

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
```

There are other features of datatypes, but we will consider only the above as a foundation for looking at the structure.

Modeling an Example

An example datatype:

$\textbf{data}\ \mathsf{E}\ \mathsf{a} = \mathsf{E}_1 \mid \mathsf{E}_2\ \mathsf{a}\ (\mathsf{E}\ \mathsf{a})\ \mathsf{Int}$

Modeling an Example

An example datatype:

$\textbf{data}\ \mathsf{E\ a} = \mathsf{E}_1 \mid \mathsf{E}_2\ \mathsf{a\ (E\ a)\ Int}$

The corresponding structure representation type:

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

Modeling an Example

An example datatype:

$\textbf{data}\ \mathsf{E\ a} = \mathsf{E}_1 \mid \mathsf{E}_2\ \mathsf{a}\ (\mathsf{E\ a})\ \mathsf{Int}$

The corresponding structure representation type:

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

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.

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.

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.

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.

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.

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

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.

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

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

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:

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:

We modify the representation to store constructor names:

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

$$\mathsf{from\_D'}\ \mathsf{Alt_1} \quad\quad = \mathsf{L}\ (\mathsf{C}\ \texttt{"Alt1"}\ \mathsf{U})$$

$$\mathsf{from\_D'}\ (\mathsf{Alt_2}\ \mathsf{i}\ \mathsf{p}) = \mathsf{R}\ (\mathsf{C}\ \texttt{"Alt2"}\ (\mathsf{i}\ {:}{\times}{:}\ \mathsf{p}))$$

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:

We modify the representation to store constructor names:

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

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:

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

Constructor name:

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

Binary product:

$$\mathsf{show_\times} :: (\mathsf{a} \to \mathsf{String}) \to (\mathsf{b} \to \mathsf{String}) \to \mathsf{a} :\!\times\!: \mathsf{b} \to \mathsf{String}$$
$$\mathsf{show_\times}\ \mathsf{show_a}\ \mathsf{show_b}\ (\mathsf{a} :\!\times\!: \mathsf{b}) = \mathsf{show_a}\ \mathsf{a} +\!\!+ \texttt{" "} +\!\!+ \mathsf{show_b}\ \mathsf{b}$$

Generic Functions: show

We define a show function for each possible structure case.

Unit:

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

Constructor name:

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

Binary product:

$$\mathsf{show_\times} :: (\mathsf{a} \to \mathsf{String}) \to (\mathsf{b} \to \mathsf{String}) \to \mathsf{a} :\!\times\!: \mathsf{b} \to \mathsf{String}$$
$$\mathsf{show_\times}\ \mathsf{show_a}\ \mathsf{show_b}\ (\mathsf{a} :\!\times\!: \mathsf{b}) = \mathsf{show_a}\ \mathsf{a} +\!\!+ \texttt{" "} +\!\!+ \mathsf{show_b}\ \mathsf{b}$$

Binary sum:

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

Generic Functions: show

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

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

Generic Functions: show

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

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

The show function for D is just a hop away:

$$
\begin{aligned}
&\mathrm{show_D} :: (\mathrm{p} \to \mathrm{String}) \to \mathrm{D\ p} \to \mathrm{String} \\
&\mathrm{show_D}\ \mathrm{show_p} = \mathrm{show_{Rep_D}}\ \mathrm{show_p} \circ \mathrm{from\_D'}
\end{aligned}
$$

Generic Functions: show

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

Some observations:

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

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.

Generic Functions: show

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

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

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

- 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

$$
\begin{aligned}
&\mathsf{show_U} :: && \mathsf{U} && \to \mathsf{String}\\
&\mathsf{show_C} :: \ldots \Rightarrow && \mathsf{C\ a} && \to \mathsf{String}\\
&\mathsf{show_+} :: \ldots \Rightarrow && \mathsf{a :+: b} && \to \mathsf{String}\\
&\quad \ldots
\end{aligned}
$$

- 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

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

- A common encoding for isomorphisms

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

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:

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

Constructor name:

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

Polymorphic Recursion

The instances for each structure representation case:

Unit:

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

Constructor name:

instance Show a $\Rightarrow$ Show (C a) **where**
  show = show_{C} show

Binary product:

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

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

Binary product:

instance (Show a, Show b) ⇒ Show (a :×: b) **where**
  show = $\text{show}_{\times}$ show show

Binary sum:

instance (Show a, Show b) ⇒ Show (a :+: b) **where**
  show = show_{+} show show

Polymorphic Recursion

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

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

Polymorphic Recursion

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

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

Compare to the new version that is now possible:

$$\mathsf{show}'_{\mathsf{Rep_D}} :: \mathsf{Show}\ \mathsf{p} \Rightarrow \mathsf{Rep_D}\ \mathsf{p} \rightarrow \mathsf{String}$$
$$\mathsf{show}'_{\mathsf{Rep_D}} = \mathsf{show}$$

Encoding Isomorphisms

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

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

Encoding Isomorphisms

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

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

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 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 :

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 $\mathsf{Rep_T}$:

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

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 $\mathsf{Rep_T}$:

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

- 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.

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 $\mathsf{Rep_T}$:

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

- 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
```

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) = Rep_D p
  from = from_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) $=$ $\mathrm{Rep}_{\mathrm{D}}$ 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:

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

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`