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
  - Uniplate – similar to SYB but faster and less expressive

Generic Platforms

Many different implementations:

- Preprocessors:
  - PolyP
  - Generic Haskell
- Libraries
  - Scrap Your Boilerplate (SYB) – included with GHC for a long time
  - Uniplate – similar to SYB but faster and less expressive
  - EMGM – fast sums-of-products

Generic Platforms

Many different implementations:

- Preprocessors:
  - PolyP
  - Generic Haskell
- Libraries
  - Scrap Your Boilerplate (SYB) – included with GHC for a long time
  - Uniplate – similar to SYB but faster and less expressive
  - EMGM – fast sums-of-products
  - Regular – recursion schemes

Generic Platforms

Many different implementations:

- Preprocessors:
  - PolyP
  - Generic Haskell
- Libraries
  - Scrap Your Boilerplate (SYB) – included with GHC for a long time
  - Uniplate – similar to SYB but faster and less expressive
  - EMGM – fast sums-of-products
  - 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
  - Uniplate – similar to SYB but faster and less expressive
  - EMGM – fast sums-of-products
  - 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
  - Uniplate – similar to SYB but faster and less expressive
  - EMGM – fast sums-of-products
  - 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

$\textbf{data}\ \mathsf{T}_{\mathsf{sum}} = \mathsf{A}_1 \mid \mathsf{A}_2$

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:

```
data U = U
```

A constructor name:

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

A field type:

```
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

Note: There are other features of datatypes, but we consider only the above.

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:

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

The corresponding structure representation type:

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

Notes:

- $:+:$ is $\textbf{infixr}\ 5$ and $:\times:$ is $\textbf{infixr}\ 6$.
- Operators nest to the right.

Converting Between Types: Isomorphism

- Generic functions work on the sums-of-products model.

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.

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

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.

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

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

Converting Between Types: Isomorphism

For convenience, we join the representation type and isomorphism in a type class `Generic` with an associated type synonym `Rep`.

```
class Generic a where
  type Rep a
  from :: a → Rep a
  to   :: Rep a → a
```

Converting Between Types: Isomorphism

For convenience, we join the representation type and isomorphism in a type class `Generic` with an associated type synonym `Rep`.

```
class Generic a where
  type Rep a
  from :: a → Rep a
  to   :: Rep a → a
```

The instance for `E`:

```
instance Generic (E a) where
  type Rep (E a) = Rep_E a
  from = from_E
  to   = to_E
```

Generic Functions

A generic function

- Is defined on each case of the structure representation and

Generic Functions

A generic function

- Is defined on each case of the structure representation and
- Works for every datatype that has a structure representation and isomorphism.

Generic Functions

A generic function

- Is defined on each case of the structure representation and
- Works for every datatype that has a structure representation and isomorphism.

Generic Functions

A generic function

- Is defined on each case of the structure representation and
- Works for every datatype that has a structure representation and isomorphism.

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

- We define a show function for each case.

Defining show

Unit:

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

Defining show

Unit:

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

Constructor name:

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

Defining show

Unit:

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

Constructor name:

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

Field:

$$\mathsf{show_K} :: (\mathsf{a} \to \mathsf{String}) \to \mathsf{K\ a} \to \mathsf{String}$$
$$\mathsf{show_K}\ \mathsf{show_a}\ (\mathsf{K\ a}) = \mathsf{show_a}\ \mathsf{a}$$

Defining show

Binary sum:

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

Defining show

Binary sum:

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

Binary product:

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

Defining show

Recall:

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

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

$$\begin{aligned}
\mathrm{show_{Rep_E}} ::\ & (a \to \mathrm{String}) \to ((a \to \mathrm{String}) \to \mathrm{E}\ a \to \mathrm{String}) \\
& \to \mathrm{Rep_E}\ a \to \mathrm{String}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{show_{Rep_E}}\ \mathrm{show_a}\ \mathrm{show_E} = \\
&\quad \mathrm{show_+}\ (\mathrm{show_C}\ \mathrm{show_U}) \\
&\qquad (\mathrm{show_C}\ (\mathrm{show_\times}\ (\mathrm{show_K}\ \mathrm{show_a}) \\
&\qquad\qquad (\mathrm{show_\times}\ (\mathrm{show_K}\ (\mathrm{show_E}\ \mathrm{show_a}))\ (\mathrm{show_K}\ \mathrm{show_{Int}}))))
\end{aligned}$$

Defining show

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

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

The show_E function itself is just an isomorphism away:

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

Defining show

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

Some observations:

- This is **not** a generic function.

Defining show

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

Some observations:

- This is **not** a generic function.
- It is defined on the structure of E, not on datatypes in general.

Defining show

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

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

Some observations:

- This is **not** a generic function.
- It is defined on the structure of E, not on datatypes in general.
- It demonstrates a predictable pattern for defining the generic function.

Defining show

Consider these typical expressions and their types:

$$\begin{array}{lll} \mathsf{show}_{\mathsf{C}}\ \mathsf{show}_{\mathsf{U}} & :: \mathsf{C}\ \mathsf{U} & \to \mathsf{String} \\ \mathsf{show}_{\times}\ (\mathsf{show}_{\mathsf{K}}\ \mathsf{show}_{\mathsf{Int}})\ (\mathsf{show}_{\mathsf{K}}\ \mathsf{show}_{\mathsf{Char}}) & :: (\mathsf{K}\ \mathsf{Int}\ {:}{\times}{:}\ \mathsf{K}\ \mathsf{Char}) & \to \mathsf{String} \end{array}$$

- $\mathsf{show}_?$ functions call other $\mathsf{show}_?$ functions.

Defining show

Consider these typical expressions and their types:

$$
\begin{array}{lll}
\mathsf{show}_{\mathsf{C}}\ \mathsf{show}_{\mathsf{U}} & :: \mathsf{C}\ \mathsf{U} & \rightarrow \mathsf{String} \\
\mathsf{show}_{\times}\ (\mathsf{show}_{\mathsf{K}}\ \mathsf{show}_{\mathsf{Int}})\ (\mathsf{show}_{\mathsf{K}}\ \mathsf{show}_{\mathsf{Char}}) & :: (\mathsf{K}\ \mathsf{Int}\ {:}{\times}{:}\ \mathsf{K}\ \mathsf{Char}) & \rightarrow \mathsf{String}
\end{array}
$$

- $\mathsf{show}_?$ functions call other $\mathsf{show}_?$ functions.
- They can be considered recursive but not in the usual way.

Defining show

Consider these typical expressions and their types:

$$
\begin{array}{lll}
\mathsf{show}_{\mathsf{C}}\ \mathsf{show}_{\mathsf{U}} & :: \mathsf{C}\ \mathsf{U} & \to \mathsf{String} \\
\mathsf{show}_{\times}\ (\mathsf{show}_{\mathsf{K}}\ \mathsf{show}_{\mathsf{Int}})\ (\mathsf{show}_{\mathsf{K}}\ \mathsf{show}_{\mathsf{Char}}) & :: (\mathsf{K}\ \mathsf{Int} :\!\times\!: \mathsf{K}\ \mathsf{Char}) & \to \mathsf{String}
\end{array}
$$

- $\mathsf{show}_?$ functions call other $\mathsf{show}_?$ functions.
- They can be considered recursive but not in the usual way.
- Polymorphic recursion – functions with different types that have a common scheme that reference each other

Defining `show`

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

Defining show

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

- The class declaration specifies the type signature.

Defining show

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

- The class declaration specifies the type signature.
- Each recursive (type) case is specified by an instance of the class.

Defining show

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

- The class declaration specifies the type signature.
- Each recursive (type) case is specified by an instance of the class.

Defining `show`

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

- The class declaration specifies the type signature.
- Each recursive (type) case is specified by an instance of the class.

A simplified definition of the `Show` class:

```
class Show a where
  show :: a → String
```

Defining show

Some of the instances for each structure representation case:

Defining show

Some of the instances for each structure representation case:

Constructor name:

instance Show a $\Rightarrow$ Show (C a) **where**
  show $=$ show_C show

Defining show

Some of the instances for each structure representation case:

Constructor name:

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

Binary sum:

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

Defining show

Some of the instances for each structure representation case:

Constructor name:

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

Binary sum:

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

The remaining instances are straightforward.

Defining show

Now, compare:

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

$$
\begin{aligned}
& \mathsf{show}_{\mathsf{Rep_E}}\ \mathsf{show_a}\ \mathsf{show_E} = \\
& \quad \mathsf{show}_{+}\ (\mathsf{show_C}\ \mathsf{show_U}) \\
& \qquad (\mathsf{show_C}\ (\mathsf{show}_{\times}\ (\mathsf{show_K}\ \mathsf{show_a}) \\
& \qquad\qquad (\mathsf{show}_{\times}\ (\mathsf{show_K}\ (\mathsf{show_E}\ \mathsf{show_a}))\ (\mathsf{show_K}\ \mathsf{show_{Int}}))))
\end{aligned}
$$

Defining show

Now, compare:

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

To:

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

Defining show

Finally, we can use a slightly different Show class to support generic functions for any type that has a representation.

Defining show

Finally, we can use a slightly different Show class to support generic functions for any type that has a representation.

class Show a **where**

  show :: a $\rightarrow$ String

  default show :: (Generic a, Show (Rep a)) $\Rightarrow$ a $\rightarrow$ String
  show = show $\circ$ from

- This uses the DefaultSignatures language extension: if type a has the instances Show (Rep a) and Generic a, then the given definition is used.

Defining show

Finally, we can use a slightly different Show class to support generic functions for any type that has a representation.

$$
\begin{aligned}
&\textbf{class}\ \mathrm{Show}\ a\ \textbf{where} \\
&\quad \mathrm{show} :: a \to \mathrm{String} \\
&\quad \textbf{default}\ \mathrm{show} :: (\mathrm{Generic}\ a, \mathrm{Show}\ (\mathrm{Rep}\ a)) \Rightarrow a \to \mathrm{String} \\
&\quad \mathrm{show} = \mathrm{show} \circ \mathrm{from}
\end{aligned}
$$

- This uses the DefaultSignatures language extension: if type a has the instances Show (Rep a) and Generic a, then the given definition is used.

The instance for E:

$$\textbf{instance}\ \mathrm{Show}\ a \Rightarrow \mathrm{Show}\ (E\ a)$$

Sums-of-Products and Beyond

We presented a sums-of-products view.

- We used Haskell2010 plus a few GHC language extensions:

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE DefaultSignatures #-}
```

Sums-of-Products and Beyond

We presented a sums-of-products view.

- We used Haskell2010 plus a few GHC language extensions:

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE DefaultSignatures #-}
```

- Typically, a GP library does not support another view.

Sums-of-Products and Beyond

We presented a sums-of-products view.

- We used Haskell2010 plus a few GHC language extensions:

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE DefaultSignatures #-}
```

- Typically, a GP library does not support another view.
- But we can, with a few more extensions:

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverlappingInstances #-}
```

The Uniplate View

- Uniplate uses a simplified spine view.

The Uniplate View

- Uniplate uses a simplified spine view.
- The spine is the sequence of fields in a constructor.

The Uniplate View

- Uniplate uses a simplified spine view.
- The spine is the sequence of fields in a constructor.
- SYB models the "full" spine, i.e. all fields (which can naturally have different types).

The Uniplate View

- Uniplate uses a simplified spine view.
- The spine is the sequence of fields in a constructor.
- SYB models the "full" spine, i.e. all fields (which can naturally have different types).
- Uniplate models only a list of the (recursive) children (which have the same type).

Defining descend

- We define the function descend from Uniplate to demonstrate that our library can model the simplified spine view.

Defining `descend`

- We define the function `descend` from Uniplate to demonstrate that our library can model the simplified spine view.
- `descend` performs a traversal of the children and applies a function to each one.

Defining descend

- We define the function descend from Uniplate to demonstrate that our library can model the simplified spine view.
- descend performs a traversal of the children and applies a function to each one.
- We use the following signature:

```
class Uniplate a where
  descend :: (a → a) → a → a
```

Defining descend

- We define the function descend from Uniplate to demonstrate that our library can model the simplified spine view.
- descend performs a traversal of the children and applies a function to each one.
- We use the following signature:

```
class Uniplate a where
  descend :: (a → a) → a → a
```

- Note that we must traverse every field to determine whether that field is a child or not. (Uniplate does this in an ad-hoc way.)

Defining descend

- We define the function descend from Uniplate to demonstrate that our library can model the simplified spine view.
- descend performs a traversal of the children and applies a function to each one.
- We use the following signature:

class Uniplate a **where**
  descend :: (a $\rightarrow$ a) $\rightarrow$ a $\rightarrow$ a

- Note that we must traverse every field to determine whether that field is a child or not. (Uniplate does this in an ad-hoc way.)
- Our generic function must support:

Defining descend

- We define the function descend from Uniplate to demonstrate that our library can model the simplified spine view.
- descend performs a traversal of the children and applies a function to each one.
- We use the following signature:

```
class Uniplate a where
  descend :: (a → a) → a → a
```

- Note that we must traverse every field to determine whether that field is a child or not. (Uniplate does this in an ad-hoc way.)
- Our generic function must support:
  - Polymorphic recursion on the structure (as usual) *and*

Defining descend

- We define the function descend from Uniplate to demonstrate that our library can model the simplified spine view.
- descend performs a traversal of the children and applies a function to each one.
- We use the following signature:

```haskell
class Uniplate a where
  descend :: (a → a) → a → a
```

- Note that we must traverse every field to determine whether that field is a child or not. (Uniplate does this in an ad-hoc way.)
- Our generic function must support:
  - Polymorphic recursion on the structure (as usual) *and*
  - A function parameter whose type matches only some of the fields.

Defining $\mathsf{descend}'$

Consequently, we use a signature with different types for the function parameter and the structure representation:

$$
\begin{array}{l}
\textbf{class } \mathsf{Uniplate}' \ \mathsf{a}\ \mathsf{r} \ \textbf{where} \\
\quad \mathsf{descend}' :: (\mathsf{r} \rightarrow \mathsf{r}) \rightarrow \mathsf{a} \rightarrow \mathsf{a}
\end{array}
$$

Defining descend′

Consequently, we use a signature with different types for the function parameter and the structure representation:

```
class Uniplate′ a r where
  descend′ :: (r → r) → a → a
```

- We need the function parameter type (r) in the Uniplate′ class.

Defining descend′

Consequently, we use a signature with different types for the function parameter and the structure representation:

class Uniplate′ a r **where**
  descend′ :: $(r \to r) \to a \to a$

- We need the function parameter type (r) in the Uniplate′ class.
- We will come back to Uniplate later.

Defining $\mathsf{descend}'$

Most of the instances are straightforward:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Uniplate}'\ \mathsf{U}\ \mathsf{a}\ \textbf{where} \\
\quad \mathsf{descend}'\ \_\ \mathsf{U} = \mathsf{U}
\end{array}
$$

Defining descend′

Most of the instances are straightforward:

instance Uniplate′ U a **where**
  descend′ _ U = U

instance Uniplate′ a r ⇒ Uniplate′ (C a) r **where**
  descend′ f (C nm a) = C nm (descend′ f a)

Defining descend′

Most of the instances are straightforward:

```
instance Uniplate′ U a where
  descend′ _ U = U
```

```
instance Uniplate′ a r ⇒ Uniplate′ (C a) r where
  descend′ f (C nm a) = C nm (descend′ f a)
```

```
instance (Uniplate′ a r, Uniplate′ b r) ⇒ Uniplate′ (a :+: b) r where
  descend′ f (L a) = L (descend′ f a)
  descend′ f (R b) = R (descend′ f b)
```

Defining descend′

Most of the instances are straightforward:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Uniplate}'\ \mathsf{U}\ \mathsf{a}\ \textbf{where}\\
\quad \mathsf{descend}'\ \_\ \mathsf{U} = \mathsf{U}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Uniplate}'\ \mathsf{a}\ \mathsf{r} \Rightarrow \mathsf{Uniplate}'\ (\mathsf{C}\ \mathsf{a})\ \mathsf{r}\ \textbf{where}\\
\quad \mathsf{descend}'\ \mathsf{f}\ (\mathsf{C}\ \mathsf{nm}\ \mathsf{a}) = \mathsf{C}\ \mathsf{nm}\ (\mathsf{descend}'\ \mathsf{f}\ \mathsf{a})
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{Uniplate}'\ \mathsf{a}\ \mathsf{r}, \mathsf{Uniplate}'\ \mathsf{b}\ \mathsf{r}) \Rightarrow \mathsf{Uniplate}'\ (\mathsf{a} :+: \mathsf{b})\ \mathsf{r}\ \textbf{where}\\
\quad \mathsf{descend}'\ \mathsf{f}\ (\mathsf{L}\ \mathsf{a}) = \mathsf{L}\ (\mathsf{descend}'\ \mathsf{f}\ \mathsf{a})\\
\quad \mathsf{descend}'\ \mathsf{f}\ (\mathsf{R}\ \mathsf{b}) = \mathsf{R}\ (\mathsf{descend}'\ \mathsf{f}\ \mathsf{b})
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{Uniplate}'\ \mathsf{a}\ \mathsf{r}, \mathsf{Uniplate}'\ \mathsf{b}\ \mathsf{r}) \Rightarrow \mathsf{Uniplate}'\ (\mathsf{a} :\times: \mathsf{b})\ \mathsf{r}\ \textbf{where}\\
\quad \mathsf{descend}'\ \mathsf{f}\ (\mathsf{a} :\times: \mathsf{b}) = \mathsf{descend}'\ \mathsf{f}\ \mathsf{a} :\times: \mathsf{descend}'\ \mathsf{f}\ \mathsf{b}
\end{array}
$$

Defining $\mathsf{descend'}$

It is the K instance that is interesting.

Defining descend′

It is the K instance that is interesting.

Because there is a fall-back instance:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Uniplate}'\ (\mathsf{K}\ \mathsf{a})\ \mathsf{r}\ \textbf{where} \\
\quad \mathsf{descend}'\ \_\ (\mathsf{K}\ \mathsf{a}) = \mathsf{K}\ \mathsf{a}
\end{array}
$$

Defining descend′

It is the K instance that is interesting.

Because there is a fall-back instance:

```
instance Uniplate′ (K a) r where
  descend′ _ (K a) = K a
```

And an instance where we apply the function parameter:

```
instance Uniplate′ (K a) a where
  descend′ f (K a) = K (f a)
```

- Note the matching types a in the header.

Defining descend′

It is the K instance that is interesting.

Because there is a fall-back instance:

```
instance Uniplate′ (K a) r where
  descend′ _ (K a) = K a
```

And an instance where we apply the function parameter:

```
instance Uniplate′ (K a) a where
  descend′ f (K a) = K (f a)
```

- Note the matching types a in the header.
- Overlapping instances implies type equality.

Defining descend′

It is the K instance that is interesting.

Because there is a fall-back instance:

```
instance Uniplate′ (K a) r where
  descend′ _ (K a) = K a
```

And an instance where we apply the function parameter:

```
instance Uniplate′ (K a) a where
  descend′ f (K a) = K (f a)
```

- Note the matching types a in the header.
- Overlapping instances implies type equality.
- This is the “trick” that allows us to determine when to choose this instance.

Defining descend

Coming back to an improved Uniplate class:

$$
\begin{array}{l}
\textbf{class}\ \mathrm{Uniplate}\ a\ \textbf{where} \\
\quad \mathrm{descend} :: (a \to a) \to a \to a \\
\quad \textbf{default}\ \mathrm{descend} :: (\mathrm{Generic}\ a, \mathrm{Uniplate}'\ (\mathrm{Rep}\ a)\ a) \Rightarrow (a \to a) \to a \to a \\
\quad \mathrm{descend}\ f = \mathrm{to} \circ \mathrm{descend}'\ f \circ \mathrm{from}
\end{array}
$$

- We again use DefaultSignatures to simplify instantiation.

Defining descend

Coming back to an improved Uniplate class:

$$\textbf{class}\ \mathsf{Uniplate}\ a\ \textbf{where}$$
$$\quad \mathsf{descend} :: (a \to a) \to a \to a$$
$$\quad \textbf{default}\ \mathsf{descend} :: (\mathsf{Generic}\ a, \mathsf{Uniplate}'\ (\mathsf{Rep}\ a)\ a) \Rightarrow (a \to a) \to a \to a$$
$$\quad \mathsf{descend}\ f = \mathsf{to} \circ \mathsf{descend}'\ f \circ \mathsf{from}$$

- We again use DefaultSignatures to simplify instantiation.
- The types of the function parameter and generic parameter are the same.

Defining descend

Coming back to an improved Uniplate class:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Uniplate}\ \mathsf{a}\ \textbf{where} \\
\quad \mathsf{descend} :: (\mathsf{a} \to \mathsf{a}) \to \mathsf{a} \to \mathsf{a} \\
\quad \textbf{default}\ \mathsf{descend} :: (\mathsf{Generic}\ \mathsf{a}, \mathsf{Uniplate'}\ (\mathsf{Rep}\ \mathsf{a})\ \mathsf{a}) \Rightarrow (\mathsf{a} \to \mathsf{a}) \to \mathsf{a} \to \mathsf{a} \\
\quad \mathsf{descend}\ \mathsf{f} = \mathsf{to} \circ \mathsf{descend'}\ \mathsf{f} \circ \mathsf{from}
\end{array}
$$

- We again use DefaultSignatures to simplify instantiation.
- The types of the function parameter and generic parameter are the same.
- They only differ "behind the scenes."

Uniplate View and Beyond

We presented a traversal function

- From Uniplate (which is not a sums-of-products library)

Uniplate View and Beyond

We presented a traversal function

- From Uniplate (which is not a sums-of-products library)
- In a library with a sums-of-products view

Uniplate View and Beyond

We presented a traversal function

- From Uniplate (which is not a sums-of-products library)
- In a library with a sums-of-products view
- Extended with overlapping instances (and type equality in particular).

Uniplate View and Beyond

We presented a traversal function

- From Uniplate (which is not a sums-of-products library)
- In a library with a sums-of-products view
- Extended with overlapping instances (and type equality in particular).

Uniplate View and Beyond

We presented a traversal function

- From Uniplate (which is not a sums-of-products library)
- In a library with a sums-of-products view
- Extended with overlapping instances (and type equality in particular).

With a bit more work, we can also define functions that work on all fields and not just the recursive children, e.g.:

$$\mathsf{topDown} :: \mathsf{C}\ \mathsf{b}\ \mathsf{a} \Rightarrow (\mathsf{a} \to \mathsf{a}) \to \mathsf{b} \to \mathsf{b}$$

- For a class `C` that supports matching on any type `T` for which there is an instance `C T T`

Uniplate View and Beyond

We presented a traversal function

- From Uniplate (which is not a sums-of-products library)
- In a library with a sums-of-products view
- Extended with overlapping instances (and type equality in particular).

With a bit more work, we can also define functions that work on all fields and not just the recursive children, e.g.:

$$\text{topDown} :: \text{C b a} \Rightarrow (\text{a} \rightarrow \text{a}) \rightarrow \text{b} \rightarrow \text{b}$$

- For a class C that supports matching on any type T for which there is an instance C T T
- Similar to the function $\text{everywhere}'$ in SYB

The Fixed-Point View

- We can use the type equality trick to model the fixed-point view in our library.

The Fixed-Point View

- We can use the type equality trick to model the fixed-point view in our library.
- The fixed-point view typically extends the sums-of-products view with an explicit indicator of recursive points in the structure.

The Fixed-Point View

- We can use the type equality trick to model the fixed-point view in our library.
- The fixed-point view typically extends the sums-of-products view with an explicit indicator of recursive points in the structure.
- In the basic sums-of-products view, recursion occurs on the structure but not on the datatype.

The Fixed-Point View

- We can use the type equality trick to model the fixed-point view in our library.
- The fixed-point view typically extends the sums-of-products view with an explicit indicator of recursive points in the structure.
- In the basic sums-of-products view, recursion occurs on the structure but not on the datatype.
- In the basic fixed-point view, we define one case of a generic function on the recursive points structural element.

The Fixed-Point View

- We can use the type equality trick to model the fixed-point view in our library.
- The fixed-point view typically extends the sums-of-products view with an explicit indicator of recursive points in the structure.
- In the basic sums-of-products view, recursion occurs on the structure but not on the datatype.
- In the basic fixed-point view, we define one case of a generic function on the recursive points structural element.
- In our library, we pass the top-level type T through the type cases.

The Fixed-Point View

- We can use the type equality trick to model the fixed-point view in our library.
- The fixed-point view typically extends the sums-of-products view with an explicit indicator of recursive points in the structure.
- In the basic sums-of-products view, recursion occurs on the structure but not on the datatype.
- In the basic fixed-point view, we define one case of a generic function on the recursive points structural element.
- In our library, we pass the top-level type `T` through the type cases.
- The case at which we can match on `T` is the recursive point.

Defining fold

- We define the function fold (catamorphism).

Defining fold

- We define the function fold (catamorphism).
- fold iterates from the root of a value to its leaves and builds up a new result based on the recursive structure of the input.

Defining fold

- We define the function fold (catamorphism).
- fold iterates from the root of a value to its leaves and builds up a new result based on the recursive structure of the input.
- We use the following signature:

```
class Fold a where
  fold :: Alg (Rep a) r → a → r
```

Defining fold

- We define the function fold (catamorphism).
- fold iterates from the root of a value to its leaves and builds up a new result based on the recursive structure of the input.
- We use the following signature:

```
class Fold a where
  fold :: Alg (Rep a) r → a → r
```

- Given an algebra and a value, compute the result of applying the algebra to the structure of the value.

Defining Alg

The algebra of the fold is a type family:

```
type family Alg a r
type instance Alg U           r = r
type instance Alg (K a)       r = Either a r → r
type instance Alg (C a)       r = Alg a r
type instance Alg (a :+: b)   r = (Alg a r, Alg b r)
type instance Alg (K a :×: b) r = Either a r → Alg b r
```

- Alg is indexed on the representation type of the input type a.

Defining Alg

The algebra of the fold is a type family:

```
type family Alg a r
type instance Alg U          r = r
type instance Alg (K a)      r = Either a r → r
type instance Alg (C a)      r = Alg a r
type instance Alg (a :+: b)  r = (Alg a r, Alg b r)
type instance Alg (K a :×: b) r = Either a r → Alg b r
```

- Alg is indexed on the representation type of the input type a.
- The type r is the result of the fold.

Defining Alg

The algebra of the fold is a type family:

```
type family Alg a r
type instance Alg U           r = r
type instance Alg (K a)       r = Either a r → r
type instance Alg (C a)       r = Alg a r
type instance Alg (a :+: b)   r = (Alg a r, Alg b r)
type instance Alg (K a :×: b) r = Either a r → Alg b r
```

- Alg is indexed on the representation type of the input type a.
- The type r is the result of the fold.
- K types can be either non-recursive (a) or recursive (r) points.

Defining Alg

For the example type:

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

The algebra type is:

$\textbf{type instance}\ \mathrm{Alg}\ (\mathrm{Rep}\ (\mathrm{E}\ a))\ r =$
$\quad (r, \mathrm{Either}\ a\ r \to \mathrm{Either}\ (\mathrm{E}\ a)\ r \to \mathrm{Either}\ \mathrm{Int}\ r \to r)$

- E a is the recursive point, even though it does not appear so in the type.

Defining Alg

For the example type:

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

The algebra type is:

$\textbf{type instance}\ Alg\ (Rep\ (E\ a))\ r =$
$\quad (r, Either\ a\ r \to Either\ (E\ a)\ r \to Either\ Int\ r \to r)$

- E a is the recursive point, even though it does not appear so in the type.
- The instances of the generic function ensure the separation of non-recursive and recursive K cases.

Defining Fold′

We again define a helper generic function:

```
class Fold′ a t where
  fold′ :: proxy t → Alg (Rep t) r → Alg a r → a → r
```

- a is the structure type.

Defining Fold′

We again define a helper generic function:

$$\textbf{class}\ \mathsf{Fold}'\ \mathsf{a}\ \mathsf{t}\ \textbf{where}$$
$$\quad \mathsf{fold}' :: \mathsf{proxy}\ \mathsf{t} \to \mathsf{Alg}\ (\mathsf{Rep}\ \mathsf{t})\ \mathsf{r} \to \mathsf{Alg}\ \mathsf{a}\ \mathsf{r} \to \mathsf{a} \to \mathsf{r}$$

- a is the structure type.
- t is the recursive type.

Defining Fold′

We again define a helper generic function:

class Fold′ a t **where**
  fold′ :: proxy t → Alg (Rep t) r → Alg a r → a → r

- a is the structure type.
- t is the recursive type.
- The "proxy" provides proof of t while preventing the instances of Fold′ from using it.

Defining Fold′

The instances that do not have recursion:

```
instance Fold′ U t where
  fold′ _ _ alg U = alg
```

```
instance Fold′ a t ⇒ Fold′ (C a) t where
  fold′ p palg alg (C _ a) = fold′ p palg alg a
```

```
instance (Fold′ a t, Fold′ b t) ⇒ Fold′ (a :+: b) t where
  fold′ p palg (alg, _) (L a) = fold′ p palg alg a
  fold′ p palg (_, alg) (R b) = fold′ p palg alg b
```

Defining Fold′

The fall-back K instance:

```
instance Fold′ (K a) t where
  fold′ p _ alg (K a) = alg (Left a)
```

Defining Fold′

The fall-back K instance:

```
instance Fold′ (K a) t where
  fold′ p _ alg (K a) = alg (Left a)
```

The recursive K instance:

```
instance Fold t ⇒ Fold′ (K t) t where
  fold′ p palg alg (K t) = alg (Right (fold palg t))
```

Defining Fold′

The fall-back :×: instance:

$\textbf{instance}\ \mathrm{Fold}'\ \mathrm{b}\ \mathrm{t} \Rightarrow \mathrm{Fold}'\ (\mathrm{K}\ \mathrm{a}\ {:}{\times}{:}\ \mathrm{b})\ \mathrm{t}\ \textbf{where}$
$\quad \mathrm{fold}'\ \mathrm{p}\ \mathrm{palg}\ \mathrm{alg}\ (\mathrm{K}\ \mathrm{a}\ {:}{\times}{:}\ \mathrm{b}) = \mathrm{fold}'\ \mathrm{p}\ \mathrm{palg}\ (\mathrm{alg}\ (\mathrm{Left}\ \mathrm{a}))\ \mathrm{b}$

Defining Fold'

The fall-back $:\times:$ instance:

$$\textbf{instance}\ \mathsf{Fold}'\ b\ t \Rightarrow \mathsf{Fold}'\ (\mathsf{K}\ a\ :\times:\ b)\ t\ \textbf{where}$$
$$\quad \mathsf{fold}'\ p\ palg\ alg\ (\mathsf{K}\ a\ :\times:\ b) = \mathsf{fold}'\ p\ palg\ (alg\ (\mathsf{Left}\ a))\ b$$

The recursive $:\times:$ instance:

$$\textbf{instance}\ (\mathsf{Fold}\ t, \mathsf{Fold}'\ b\ t) \Rightarrow \mathsf{Fold}'\ (\mathsf{K}\ t\ :\times:\ b)\ t\ \textbf{where}$$
$$\quad \mathsf{fold}'\ p\ palg\ alg\ (\mathsf{K}\ a\ :\times:\ b) = \mathsf{fold}'\ p\ palg\ (alg\ (\mathsf{Right}\ (\mathsf{fold}\ palg\ a)))\ b$$

Defining fold

The improved Fold class:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Fold}\ \mathsf{a}\ \textbf{where} \\
\quad \mathsf{fold} :: \mathsf{Alg}\ (\mathsf{Rep}\ \mathsf{a})\ \mathsf{r} \to \mathsf{a} \to \mathsf{r} \\
\quad \textbf{default}\ \mathsf{fold} :: (\mathsf{Generic}\ \mathsf{a}, \mathsf{Fold'}\ (\mathsf{Rep}\ \mathsf{a})\ \mathsf{a}) \Rightarrow \mathsf{Alg}\ (\mathsf{Rep}\ \mathsf{a})\ \mathsf{r} \to \mathsf{a} \to \mathsf{r} \\
\quad \mathsf{fold}\ \mathsf{alg}\ \mathsf{x} = \mathsf{fold'}\ (\mathsf{Just}\ \mathsf{x})\ \mathsf{alg}\ \mathsf{alg}\ (\mathsf{from}\ \mathsf{x})
\end{array}
$$

- We use Maybe as a simple proxy.

Defining fold

The improved Fold class:

```
class Fold a where
  fold :: Alg (Rep a) r → a → r
  default fold :: (Generic a, Fold′ (Rep a) a) ⇒ Alg (Rep a) r → a → r
  fold alg x = fold′ (Just x) alg alg (from x)
```

- We use Maybe as a simple proxy.
- The algebra is needed twice: the second argument is pattern-matched by the instances of Fold′.

Fold and Beyond

We presented a generic recursive pattern in a library that would not typically have it.

- We can also define many other (co-)recursive functions, including the generic zipper.

Fold and Beyond

We presented a generic recursive pattern in a library that would not typically have it.

- We can also define many other (co-)recursive functions, including the generic zipper.
- The unfortunate aspect of `Alg` is that we must use `Either` since, in the type family, we cannot distinguish overlapping instances.

Fold and Beyond

We presented a generic recursive pattern in a library that would not typically have it.

- We can also define many other (co-)recursive functions, including the generic zipper.
- The unfortunate aspect of `Alg` is that we must use `Either` since, in the type family, we cannot distinguish overlapping instances.
- We believe this can be fixed with the new ordered overlapping instances in GHC.

Conclusions

- We believe generic programming is easy to understand if looked at from the right perspective.

Conclusions

- We believe generic programming is easy to understand if looked at from the right perspective.
- We are still searching for that optimal view.

Conclusions

- We believe generic programming is easy to understand if looked at from the right perspective.
- We are still searching for that optimal view.
- The library presented here is quite simple.

Conclusions

- We believe generic programming is easy to understand if looked at from the right perspective.
- We are still searching for that optimal view.
- The library presented here is quite simple.
- Yet, with a few tricks, it is also quite powerful.

Conclusions

- We believe generic programming is easy to understand if looked at from the right perspective.
- We are still searching for that optimal view.
- The library presented here is quite simple.
- Yet, with a few tricks, it is also quite powerful.
- We have also done this work in the more complicated Generic Deriving library.

References

- 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`
- Generic Deriving Extras: `https://github.com/spl/generic-deriving-extras`