“Scrap Your Boilerplate” Reloaded

Ralf Hinze[1] Andres Löh[1] Bruno C. d. S. Oliveira[2]

[1]Universität Bonn

[2]University of Oxford

April 24, 2006

Goals

- A new way to explain the Scrap Your Boilerplate approach to generic programming:
  - more obvious relation to other GP approaches such as PolyP or Generic Haskell
  - equally expressive as the original
- Long-term: structure and compare generic programming approaches.

Overview

Generic programming

In the context of Haskell (or functional programming):

- defining functions that are parameterized by type arguments and can access the structure of data types
- classic examples: structural equality, parsing, pretty-printing
- also known as: polytypic programming, structural polymorphism, datatype-generic programming

Generic functions are more than overloaded functions

Overloaded functions

- are parameterized by type arguments
- have an ad-hoc behaviour for each data type
- can be made to work for many data types, but for each data type a separate implementation is required

Generic functions are more than overloaded functions

Overloaded functions

- are parameterized by type arguments
- have an ad-hoc behaviour for each data type
- can be made to work for many data types, but for each data type a separate implementation is required

Generic functions

- can use the structure of data types
- therefore work for a large class of data types with only a few cases
- can even work for data types that the user is yet to define

Generic functions are more than overloaded functions

Overloaded functions

- are parameterized by type arguments
- have an ad-hoc behaviour for each data type
- can be made to work for many data types, but for each data type a separate implementation is required

Generic functions

- can use the structure of data types
- therefore work for a large class of data types with only a few cases
- can even work for data types that the user is yet to define

generic functions $\approx$ overloaded functions $+$ structure of data types

Generic functions are more than overloaded functions

Overloaded functions

- are parameterized by type arguments
- have an ad-hoc behaviour for each data type
- can be made to work for many data types, but for each data type a separate implementation is required

Generic functions

- can use the structure of data types
- therefore work for a large class of data types with only a few cases
- can even work for data types that the user is yet to define

generic functions $\approx$ overloaded functions $+$ generic view

Two orthogonal concepts for generic programming

- A mechanism to express overloaded functions.
- A generic view.

Current approaches to generic programming make different design decisions for both concepts.

Overloaded functions

Choices:

- Haskell type-classes.
- Dynamic types and run-time type casts.
- A data type of type representations.
- (Family of functions.)
- (Compile-time specialization.)

Overloaded functions

Choices:

- Haskell type-classes.
- Dynamic types and run-time type casts.
- A data type of type representations.
- (Family of functions.)
- (Compile-time specialization.)

Claim: Different mechanisms for overloaded functions can usually be interchanged. The generic view is the real essence of a GP approach.

Example: an overloaded sum function

$$
\begin{array}{llll}
\mathsf{sum} :: \mathsf{Type\ a} \to \mathsf{a} \to \mathsf{Int} \\
\mathsf{sum\ Int} & \mathsf{n} & = \mathsf{n} \\
\mathsf{sum\ Char} & \_ & = 0 \\
\mathsf{sum\ (List\ a)} & \mathsf{xs} & = \mathsf{foldr}\ (+)\ 0\ (\mathsf{map}\ (\mathsf{sum\ a})\ \mathsf{xs}) \\
\mathsf{sum\ (Pair\ a\ b)} & (\mathsf{x}, \mathsf{y}) & = \mathsf{sum\ a\ x} + \mathsf{sum\ b\ y} \\
\mathsf{sum\ (Tree\ a)} & \mathsf{t} & = \mathsf{sum\ (List\ a)\ (inorder\ t)}
\end{array}
$$

Example: an overloaded sum function

$$
\begin{array}{llll}
\mathsf{sum} :: \mathsf{Type\ a} \to \mathsf{a} \to \mathsf{Int} \\
\mathsf{sum\ Int} & \mathsf{n} & = \mathsf{n} \\
\mathsf{sum\ Char} & \_ & = 0 \\
\mathsf{sum\ (List\ a)} & \mathsf{xs} & = \mathsf{foldr\ (+)\ 0\ (map\ (sum\ a)\ xs)} \\
\mathsf{sum\ (Pair\ a\ b)} & \mathsf{(x, y)} & = \mathsf{sum\ a\ x + sum\ b\ y} \\
\mathsf{sum\ (Tree\ a)} & \mathsf{t} & = \mathsf{sum\ (List\ a)\ (inorder\ t)}
\end{array}
$$

We make use of a type of type representations:

$$
\begin{array}{ll}
\mathbf{data}\ \mathsf{Type} :: * \to *\ \mathbf{where} \\
\quad \mathsf{List} & :: \forall \mathsf{a}. \mathsf{Type\ a} \to \mathsf{Type\ [a]} \\
\quad \mathsf{Tree} & :: \forall \mathsf{a}. \mathsf{Type\ a} \to \mathsf{Type\ (Tree\ a)} \\
\quad \mathsf{Pair} & :: \forall \mathsf{a\ b}. \mathsf{Type\ a} \to \mathsf{Type\ b} \to \mathsf{Type\ (a, b)} \\
\quad \mathsf{Int} & :: \mathsf{Type\ Int} \\
\quad \ldots
\end{array}
$$

Generic views

Choices:

- Data types are fixed points of regular functors, as in PolyP.
- Data types are sums of products, as in Generic Haskell.
- What about SYB?

SYB is a **combinator-based** approach to generic programming.

Generic views

Choices:

- Data types are fixed points of regular functors, as in PolyP.
- Data types are sums of products, as in Generic Haskell.
- What about SYB?

SYB is a **combinator-based** approach to generic programming.

Now: Identify the generic view at the basis of SYB.

Overview

What do Haskell values have in common?

The key to the Scrap Your Boilerplate view are not the data types, but the values.

```
()
Just 'c'
Left 17
Node Empty True Empty
(:) 1 ((:) 2 ((:) 3 []))
(,,,) False 'a' 3 Nothing
```

What do Haskell values have in common?

The key to the Scrap Your Boilerplate view are not the data types, but the values.

```
()
Just 'c'
Left 17
Node Empty True Empty
(:) 1 ((:) 2 ((:) 3 []))
(,,,) False 'a' 3 Nothing
```

They are all data constructors that are applied to other values.

What do Haskell values have in common?

The key to the Scrap Your Boilerplate view are not the data types, but the values.

$\mathsf{Constr}\ ()$

$\mathsf{Constr\ Just} \diamond \texttt{'c'}$

$\mathsf{Constr\ Left} \diamond 17$

$\mathsf{Constr\ Node} \diamond \mathsf{Empty} \diamond \mathsf{True} \diamond \mathsf{Empty}$

$\mathsf{Constr}\ (:) \diamond 1 \diamond ((:)\ 2\ ((:)\ 3\ [\,]))$

$\mathsf{Constr}\ (,,,) \diamond \mathsf{False} \diamond \texttt{'a'} \diamond 3 \diamond \mathsf{Nothing}$

They are all data constructors that are applied to other values.

What do Haskell values have in common?

The key to the Scrap Your Boilerplate view are not the data types, but the values.

$\mathsf{Constr}\ ()$

$\mathsf{Constr\ Just} \diamond \mathtt{'c'}$

$\mathsf{Constr\ Left} \diamond 17$

$\mathsf{Constr\ Node} \diamond \mathsf{Empty} \diamond \mathsf{True} \diamond \mathsf{Empty}$

$\mathsf{Constr}\ (:) \diamond 1 \diamond ((:)\ 2\ ((:)\ 3\ [\,]))$

$\mathsf{Constr}\ (,,,) \diamond \mathsf{False} \diamond \mathtt{'a'} \diamond 3 \diamond \mathsf{Nothing}$

They are all data constructors that are applied to other values.

$$\begin{array}{lll} \mathsf{Constr} & :: \forall a.a \to & a \\ (\diamond) & :: \forall a\ b.\ (a \to b) \to a \to & b \end{array}$$

What do Haskell values have in common?

The key to the Scrap Your Boilerplate view are not the data types, but the values.

$\mathsf{Constr}\ ()$

$\mathsf{Constr}\ \mathsf{Just} \diamond \texttt{'c'}$

$\mathsf{Constr}\ \mathsf{Left} \diamond 17$

$\mathsf{Constr}\ \mathsf{Node} \diamond \mathsf{Empty} \diamond \mathsf{True} \diamond \mathsf{Empty}$

$\mathsf{Constr}\ (:) \diamond 1 \diamond ((:)\ 2\ ((:)\ 3\ [\,]))$

$\mathsf{Constr}\ (,,,) \diamond \mathsf{False} \diamond \texttt{'a'} \diamond 3 \diamond \mathsf{Nothing}$

They are all data constructors that are applied to other values.

$$
\begin{aligned}
\mathsf{Constr} &:: \forall a.a \to f\ a \\
(\diamond) &:: \forall a\ b.f\ (a \to b) \to a \to f\ b
\end{aligned}
$$

What do Haskell values have in common?

The key to the Scrap Your Boilerplate view are not the data types, but the values.

```
Constr ()
Constr Just ◊ 'c'
Constr Left ◊ 17
Constr Node ◊ Empty ◊ True ◊ Empty
Constr (:) ◊ 1 ◊ ((:) 2 ((:) 3 []))
Constr (,,,) ◊ False ◊ 'a' ◊ 3 ◊ Nothing
```

They are all data constructors that are applied to other values.

```
data Spine :: * → * where
    Constr :: ∀a.a → Spine a
    (◊)    :: ∀a b.Spine (a → b) → a → Spine b
```

This talk

Generic programming using the Spine data type.

$$\begin{array}{l}\textbf{data}\ \mathsf{Spine} :: * \to * \ \textbf{where}\\ \quad \mathsf{Constr} :: \forall a.a \to \mathsf{Spine}\ a\\ \quad (\Diamond) \quad\ \ :: \forall a\ b.\mathsf{Spine}\ (a \to b) \to a \to \mathsf{Spine}\ b\end{array}$$

This talk

Generic programming using the Spine data type.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Spine} :: * \to * \ \textbf{where}\\
&\quad \mathsf{Constr} :: \forall a.a \to \mathsf{Spine}\ a\\
&\quad (\diamond) \quad\ :: \forall a\ b.\mathsf{Spine}\ (a \to b) \to a \to \mathsf{Spine}\ b
\end{aligned}
$$

In classic Haskell syntax:

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Spine}\ a =\\
&\qquad\quad \mathsf{Constr}\ a\\
&\quad\ \mid \forall b.\mathsf{Spine}\ (b \to a) \diamond b
\end{aligned}
$$

Overview

From spines to values

$$
\begin{aligned}
&\mathsf{fromSpine} :: \forall a.\mathsf{Spine}\ a \rightarrow a\\
&\mathsf{fromSpine}\ (\mathsf{Constr}\ c) = c\\
&\mathsf{fromSpine}\ (f \diamond x) \quad\ \ = \mathsf{fromSpine}\ f\ x
\end{aligned}
$$

From values to spines

$$
\begin{aligned}
&\mathsf{toSpine}_{\mathsf{Tree}} :: \forall a.\mathsf{Tree}\ a \rightarrow \mathsf{Spine}\ (\mathsf{Tree}\ a)\\
&\mathsf{toSpine}_{\mathsf{Tree}}\ \mathsf{Empty} = \mathsf{Constr}\ \mathsf{Empty}\\
&\mathsf{toSpine}_{\mathsf{Tree}}\ (\mathsf{Node}\ l\ x\ r) = \mathsf{Constr}\ \mathsf{Node} \diamond l \diamond x \diamond r\\
&\mathsf{toSpine}_{(,)} :: \forall a\ b.(a, b) \rightarrow \mathsf{Spine}\ (a, b)\\
&\mathsf{toSpine}_{(,)}\ (x, y) = \mathsf{Constr}\ (,) \diamond x \diamond y\\
&\mathsf{toSpine}_{\mathsf{Int}} :: \mathsf{Int} \rightarrow \mathsf{Spine}\ \mathsf{Int}\\
&\mathsf{toSpine}_{\mathsf{Int}}\ n = \mathsf{Constr}\ n
\end{aligned}
$$

From values to spines

$$
\begin{aligned}
&\mathsf{toSpine}_{\mathsf{Tree}} :: \forall a. \mathsf{Tree}\ a \to \mathsf{Spine}\ (\mathsf{Tree}\ a)\\
&\mathsf{toSpine}_{\mathsf{Tree}}\ \mathsf{Empty} = \mathsf{Constr}\ \mathsf{Empty}\\
&\mathsf{toSpine}_{\mathsf{Tree}}\ (\mathsf{Node}\ l\ x\ r) = \mathsf{Constr}\ \mathsf{Node} \diamond l \diamond x \diamond r\\
&\mathsf{toSpine}_{(,)} :: \forall a\ b. (a, b) \to \mathsf{Spine}\ (a, b)\\
&\mathsf{toSpine}_{(,)}\ (x, y) = \mathsf{Constr}\ (,) \diamond x \diamond y\\
&\mathsf{toSpine}_{\mathsf{Int}} :: \mathsf{Int} \to \mathsf{Spine}\ \mathsf{Int}\\
&\mathsf{toSpine}_{\mathsf{Int}}\ n = \mathsf{Constr}\ n
\end{aligned}
$$

The function toSpine is not parametrically polymorphic, but can be made into an **overloaded** function.

From values to spines – continued

$$
\begin{aligned}
&\mathsf{toSpine} :: \forall a.\mathsf{Type}\ a \to a \to \mathsf{Spine}\ a\\
&\mathsf{toSpine}\ (\mathsf{Tree}\ a) \quad = \mathsf{toSpine}_{\mathsf{Tree}}\\
&\mathsf{toSpine}\ (\mathsf{Pair}\ a\ b) = \mathsf{toSpine}_{(,)}\\
&\mathsf{toSpine}\ \mathsf{Int} \qquad\quad = \mathsf{toSpine}_{\mathsf{Int}}\\
&\mathsf{toSpine} \ldots
\end{aligned}
$$

Using spines to program generically

Let us program a simple toString function:

$$\begin{array}{l}
\mathsf{toString} :: \mathsf{Type\ a} \to \mathsf{a} \to \mathsf{String} \\
\mathsf{toString\ t\ x} = \mathsf{toString}_{-}\ (\mathsf{toSpine\ t\ x}) \\
\mathsf{toString}_{-} :: \mathsf{Spine\ a} \to \mathsf{String} \\
\mathsf{toString}_{-}\ (\mathsf{Constr\ c}) = \mathbf{???} \\
\mathsf{toString}_{-}\ (\mathsf{f} \diamond \mathsf{x}) = \texttt{"("} +\!\!+\ \mathsf{toString}_{-}\ \mathsf{f} +\!\!+ \texttt{" "} +\!\!+\ \mathsf{toString}\ \mathbf{???}\ \mathsf{x} +\!\!+ \texttt{")"}
\end{array}$$

Using spines to program generically

Let us program a simple toString function:

```
toString :: Type a → a → String
toString t x = toString_ (toSpine t x)
toString_ :: Spine a → String
toString_ (Constr c)  = ???
toString_ (f ◇ x)     = "(" ++ toString_ f ++ " " ++ toString ??? x ++ ")"
```

We lack information:

- constructor name (and possibly other info about the constructor)
- type information about the spine contents

Using spines to program generically

Let us program a simple toString function:

$$\mathsf{toString} :: \mathsf{Type}\ a \to a \to \mathsf{String}$$
$$\mathsf{toString}\ t\ x = \mathsf{toString}_{\_}\ (\mathsf{toSpine}\ t\ x)$$

$$\mathsf{toString}_{\_} :: \mathsf{Spine}\ a \to \mathsf{String}$$
$$\mathsf{toString}_{\_}\ (c\ \text{`}\mathsf{As}\text{`}\ n) = n$$
$$\mathsf{toString}_{\_}\ (f \diamond (x : t)) = \texttt{"("} \mathbin{+\!\!+} \mathsf{toString}_{\_}\ f \mathbin{+\!\!+} \texttt{" "} \mathbin{+\!\!+} \mathsf{toString}\ t\ x \mathbin{+\!\!+} \texttt{")"}$$

We lack information:

- constructor name (and possibly other info about the constructor)
- type information about the spine contents

Using spines to program generically

Let us program a simple toString function:

$$\begin{aligned}
&\mathsf{toString} :: \mathsf{Type}\ a \rightarrow a \rightarrow \mathsf{String}\\
&\mathsf{toString}\ t\ x = \mathsf{toString}_{-}\ (\mathsf{toSpine}\ t\ x)\\
&\mathsf{toString}_{-} :: \mathsf{Spine}\ a \rightarrow \mathsf{String}\\
&\mathsf{toString}_{-}\ (c\ \text{`As`}\ n) = n\\
&\mathsf{toString}_{-}\ (f \diamond (x : t)) = \texttt{"("} +\!\!+\ \mathsf{toString}_{-}\ f +\!\!+ \texttt{" "} +\!\!+\ \mathsf{toString}\ t\ x +\!\!+ \texttt{")"}
\end{aligned}$$

We lack information:

- constructor name (and possibly other info about the constructor)
- type information about the spine contents

```
SYB⟩ toString (Tree Bool) (Node Empty False Empty)
"(((Node Empty) False) Empty)"
```

The full "spine view"

```
type ConDescr = String
data Spine :: * → * where
   As  :: ∀a.a → ConDescr → Spine a
   (◇) :: ∀a b.Spine (a → b) → Typed a → Spine b
data Typed :: * → * where
   (:)  :: ∀a.a → Type a → Typed a
```

The full "spine view"

```
type ConDescr = String
data Spine :: * → * where
  As  :: ∀a.a → ConDescr → Spine a
  (◊) :: ∀a b.Spine (a → b) → Typed a → Spine b
data Typed :: * → * where
  (:)  :: ∀a.a → Type a → Typed a
```

Of course, fromSpine and toSpine must be adapted:

```
toSpine :: ∀a.Type a → a → Spine a
toSpine (Tree a) Empty        = Empty `As` "Empty"
toSpine (Tree a) (Node l x r) = Node `As` "Node"
                                  ◊ (l : Tree a) ◊ (x : a) ◊ (r : Tree a)
```

Sum all integers, generically

$$
\begin{aligned}
&\mathsf{sum} :: \forall a.\mathsf{Type}\ a \rightarrow a \rightarrow \mathsf{Int}\\
&\mathsf{sum}\ \mathsf{Int}\ n = n\\
&\mathsf{sum}\ t\quad x = \mathsf{sum}_{-}\ (\mathsf{toSpine}\ t\ x)\\
&\mathsf{sum}_{-} :: \forall a.\mathsf{Spine}\ a \rightarrow \mathsf{Int}\\
&\mathsf{sum}_{-}\ (c\ `\mathsf{As}`\ n)\quad = 0\\
&\mathsf{sum}_{-}\ (f \diamond (x : t)) = \mathsf{sum}_{-}\ f + \mathsf{sum}\ t\ x
\end{aligned}
$$

Sum all integers, generically

$$
\begin{aligned}
&\mathsf{sum} :: \forall a.\mathsf{Type}\ a \to a \to \mathsf{Int}\\
&\mathsf{sum}\ \mathsf{Int}\ n = n\\
&\mathsf{sum}\ t\quad x = \mathsf{sum}_{\text{\_}}\ (\mathsf{toSpine}\ t\ x)\\
&\mathsf{sum}_{\text{\_}} :: \forall a.\mathsf{Spine}\ a \to \mathsf{Int}\\
&\mathsf{sum}_{\text{\_}}\ (c\ \text{`}\mathsf{As}\text{`}\ n)\quad = 0\\
&\mathsf{sum}_{\text{\_}}\ (f \diamond (x : t)) = \mathsf{sum}_{\text{\_}}\ f + \mathsf{sum}\ t\ x
\end{aligned}
$$

Example:

$$
\begin{aligned}
&\mathsf{SYB}\rangle\ \mathsf{sum}\ (\mathsf{List}\ (\mathsf{Either}\ \mathsf{Bool}\ \mathsf{Int}))\ [\mathsf{Right}\ 15, \mathsf{Left}\ \mathsf{False}, \mathsf{Right}\ 27]\\
&42
\end{aligned}
$$

Overview

Queries

type $\text{Query } r = \forall a.\text{Type } a \to a \to r$

Queries

$$\textbf{type}\ \mathsf{Query}\ r = \forall a.\mathsf{Type}\ a \to a \to r$$

Both toString and sum are queries:

$$\begin{aligned} &\mathsf{toString} :: \mathsf{Query\ String} \\ &\mathsf{sum} \quad\quad\ \, :: \mathsf{Query\ Int} \end{aligned}$$

Query combinators

The function mapQ applies a query to all immediate children of a value:

$$
\begin{aligned}
&\mathsf{mapQ} :: \forall r.\mathsf{Query}\ r \to \mathsf{Query}\ [r] \\
&\mathsf{mapQ}\ q\ t = \mathsf{mapQ\_}\ q \circ \mathsf{toSpine}\ t \\
&\mathsf{mapQ\_} :: \forall r.\mathsf{Query}\ r \to (\forall a.\mathsf{Spine}\ a \to [r]) \\
&\mathsf{mapQ\_}\ q\ (c\ \text{`}\mathsf{As}\text{`}\ n) \quad = [\,] \\
&\mathsf{mapQ\_}\ q\ (f \diamond (x : t)) = \mathsf{mapQ\_}\ q\ f \mathbin{+\!\!+} [q\ t\ x]
\end{aligned}
$$

Query combinators

The function mapQ applies a query to all immediate children of a value:

$$\mathrm{mapQ} :: \forall r.\mathrm{Query}\ r \to \mathrm{Query}\ [r]$$
$$\mathrm{mapQ}\ q\ t = \mathrm{mapQ}_{\_}\ q \circ \mathrm{toSpine}\ t$$
$$\mathrm{mapQ}_{\_} :: \forall r.\mathrm{Query}\ r \to (\forall a.\mathrm{Spine}\ a \to [r])$$
$$\mathrm{mapQ}_{\_}\ q\ (c\ \text{`As`}\ n) \quad = [\,]$$
$$\mathrm{mapQ}_{\_}\ q\ (f \diamond (x : t)) = \mathrm{mapQ}_{\_}\ q\ f \mathbin{+\!\!+} [q\ t\ x]$$

The function everything applies a query recursively and combines the results:

$$\mathrm{everything} :: \forall r.(r \to r \to r) \to \mathrm{Query}\ r \to \mathrm{Query}\ r$$
$$\mathrm{everything}\ op\ q\ t\ x = \mathrm{foldl1}\ op\ ([q\ t\ x] \mathbin{+\!\!+} \mathrm{mapQ}\ (\mathrm{everything}\ op\ q)\ t\ x)$$

Using query combinators

Rewriting sum as a query:

$$
\begin{aligned}
&\mathsf{sumQ} :: \mathsf{Query\ Int} \\
&\mathsf{sumQ\ Int\ n} = \mathsf{n} \\
&\mathsf{sumQ\ t} \quad \mathsf{x} = 0 \\
&\mathsf{sum} :: \mathsf{Query\ Int} \\
&\mathsf{sum} = \mathsf{everything}\ (+)\ \mathsf{sumQ}
\end{aligned}
$$

Similarly, we can define SYB traversals

$$\textbf{type}\ \mathsf{Traversal} = \forall \mathsf{a}.\mathsf{Type\ a} \to \mathsf{a} \to \mathsf{a}$$

and traversal combinators.

Overview

Comparison with the original SYB approach

```
data Spine :: * → * where
  As  :: ∀a.a → ConDescr → Spine a
  (◊) :: ∀a b.Spine (a → b) → Typed a → Spine b
```

Comparison with the original SYB approach

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Spine} :: * \to * \ \textbf{where} \\
\quad \mathsf{As}\ \ :: \forall a.a \to \mathsf{ConDescr} \to \mathsf{Spine}\ a \\
\quad (\diamond) :: \forall a\ b.\mathsf{Spine}\ (a \to b) \to \mathsf{Typed}\ a \to \mathsf{Spine}\ b \\
\mathsf{foldSpine} :: \forall a\ r. \\
\qquad\qquad (\forall a.a \to r\ a) \to \\
\qquad\qquad (\forall a\ b.r\ (a \to b) \to \mathsf{Typed}\ a \to r\ b) \to \\
\qquad\qquad \mathsf{Spine}\ a \to r\ a \\
\mathsf{foldSpine}\ \mathsf{constr}\ (\blacklozenge)\ (c\ `\mathsf{As}`\ n) \quad = \mathsf{constr}\ c \\
\mathsf{foldSpine}\ \mathsf{constr}\ (\blacklozenge)\ (f \diamond (x : t)) = (\mathsf{foldSpine}\ \mathsf{constr}\ (\blacklozenge)\ f) \blacklozenge (x : t)
\end{array}
$$

Comparison with the original SYB approach

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Spine} :: * \to * \ \textbf{where} \\
\quad \mathsf{As}\ \ :: \forall a.a \to \mathsf{ConDescr} \to \mathsf{Spine}\ a \\
\quad (\diamond) :: \forall a\ b.\mathsf{Spine}\ (a \to b) \to \mathsf{Typed}\ a \to \mathsf{Spine}\ b \\
\mathsf{foldSpine} :: \forall a\ r. \\
\qquad\qquad (\forall a.a \to r\ a) \to \\
\qquad\qquad (\forall a\ b.r\ (a \to b) \to \mathsf{Typed}\ a \to r\ b) \to \\
\qquad\qquad \mathsf{Spine}\ a \to r\ a \\
\mathsf{foldSpine}\ \mathsf{constr}\ (\blacklozenge)\ (c\ \text{`}\mathsf{As}\text{`}\ n) \quad = \mathsf{constr}\ c \\
\mathsf{foldSpine}\ \mathsf{constr}\ (\blacklozenge)\ (f \diamond (x : t)) = (\mathsf{foldSpine}\ \mathsf{constr}\ (\blacklozenge)\ f) \blacklozenge (x : t)
\end{array}
$$

Compare this to SYB's gfoldl:

$$
\begin{array}{l}
\mathsf{gfoldl} :: \forall a\ r.\mathsf{Data}\ a \Rightarrow \\
\qquad\quad (\forall a\ b.\mathsf{Data}\ a \Rightarrow r\ (a \to b) \to a \to r\ b) \to \\
\qquad\quad (\forall a.a \to r\ a) \to \\
\qquad\quad a \to r\ a
\end{array}
$$

Spine view vs. gfoldl

- All original SYB combinators are defined in terms of gfoldl.
- Using Spine, we can define generic functions in a more direct way.
- The function gfoldl is the catamorphism on Spine composed with toSpine.
- One can build the spine representation using gfoldl; therefore, both approaches are equally expressive.

Consumers vs. producers

$\mathsf{consume} :: \forall a.\mathsf{Type}\ a \to a \to t$
$\mathsf{consume} \ldots$
$\mathsf{consume}\ t\ x = \mathsf{consume}_{-}\ (\mathsf{toSpine}\ t\ x)$

$\mathsf{consume}_{-} :: \forall a.\mathsf{Spine}\ a \to t$
$\mathsf{consume}_{-} \ldots$

Consumers vs. producers

$$\begin{array}{l}
\mathsf{consume} :: \forall a. \mathsf{Type}\ a \to a \to t \\
\mathsf{consume} \ldots \\
\mathsf{consume}\ t\ x = \mathsf{consume}_{_}\ (\mathsf{toSpine}\ t\ x) \\
\mathsf{consume}_{_} :: \forall a. \mathsf{Spine}\ a \to t \\
\mathsf{consume}_{_} \ldots
\end{array}$$

$$\begin{array}{l}
\mathsf{produce} :: \forall a. \mathsf{Type}\ a \to t \to a \\
\mathsf{produce} \ldots \\
\mathsf{produce}\ t\ x = \mathsf{fromSpine}\ (\mathsf{produce}_{_}\ x) \\
\mathsf{produce}_{_} :: \forall a. t \to \mathsf{Spine}\ a \\
\mathsf{produce}_{_} \ldots
\end{array}$$

Cannot exhibit type-specific behaviour!

SYB and producers

- The original SYB (without extensions) as well as the "spine view" cannot handle producers (such as a generic parsing function).
- Simon Peyton Jones and Ralf Lämmel define gunfold in a successor to the original SYB paper.
- Unfortunately, gunfold doesn't have a direct connection to the Spine data type.
- However, one can play the same trick and define a data type which has gunfold as its catamorphism.
- SYB also cannot handle generic functions on type constructors (such as a generic map).

Applicability of toSpine

The Spine type is based on the structure of concrete Haskell values. Therefore, it is very widely applicable:

$$\textbf{data}\ \mathsf{Dynamic} :: * \ \textbf{where}$$
$$\quad \mathsf{Dyn} :: \forall t . t \to \mathsf{Type}\ t \to \mathsf{Dynamic}$$

Applicability of toSpine

The Spine type is based on the structure of concrete Haskell values. Therefore, it is very widely applicable:

$$\begin{array}{l}
\mathbf{data}\ \mathsf{Dynamic} :: * \ \mathbf{where} \\
\quad \mathsf{Dyn} :: \forall t.t \rightarrow \mathsf{Type}\ t \rightarrow \mathsf{Dynamic} \\
\mathbf{data}\ \mathsf{Type} :: * \rightarrow * \ \mathbf{where} \\
\quad \ldots \\
\quad \mathsf{Type} \quad\quad\ :: \forall a.\mathsf{Type}\ a \rightarrow \mathsf{Type}\ (\mathsf{Type}\ a) \\
\quad \mathsf{Dynamic} :: \mathsf{Type}\ \mathsf{Dynamic} \\
\quad \ldots
\end{array}$$

Applicability of toSpine

The Spine type is based on the structure of concrete Haskell values. Therefore, it is very widely applicable:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Dynamic} :: * \ \textbf{where} \\
\quad \mathsf{Dyn} :: \forall t.t \to \mathsf{Type}\ t \to \mathsf{Dynamic} \\
\textbf{data}\ \mathsf{Type} :: * \to * \ \textbf{where} \\
\quad \ldots \\
\quad \mathsf{Type} \quad\ :: \forall a.\mathsf{Type}\ a \to \mathsf{Type}\ (\mathsf{Type}\ a) \\
\quad \mathsf{Dynamic} :: \mathsf{Type}\ \mathsf{Dynamic} \\
\quad \ldots \\
\mathsf{toSpine}\ (\mathsf{Type}\ a')\ (\mathsf{Type}\ a)\ \ = \mathsf{Type}\ \text{`}\mathsf{As}\text{`}\ \texttt{"Type"} \diamond (a : \mathsf{Type}\ a) \\
\mathsf{toSpine}\ \mathsf{Dynamic}\ \ (\mathsf{Dyn}\ x\ t) = \mathsf{Dyn}\ \text{`}\mathsf{As}\text{`}\ \texttt{"Dyn"} \diamond (x : t) \diamond (t : \mathsf{Type}\ t)
\end{array}
$$

The Spine view covers GADTs and typed existentials in addition to regular and nested data types.

Overview

Conclusions

- Using the Spine data type, we can reimplement SYB.
- Using this framework, the relation to PolyP and Generic Haskell (and other approaches) becomes more obvious.
- SYB without extensions can express relatively few generic functions.
- SYB is applicable to a very large class of data types, including GADTs.
- Our approach as shown here is not suitable as a library, because the data type Type is not extensible. Alternatives:
  - Add a form of extensible data types to the language.
  - Use a different mechanism to express overloaded functions, but keep the Spine view.
- We can extend our approach to handle generic producers and generic functions on type constructors.