UNIVERSITY OF OXFORD

Generic Generic Programming

José Pedro Magalhães
joint work with Andres Löh

Practical Aspects of Declarative Languages 2014

21 January 2014

Introduction

- Generic programming: an abstraction technique to reduce code duplication
  - Generic programs operate on "representation types"; a small set of types used to encode all other user-defined datatypes
  - Conversion functions mediate the isomorphism between a datatype and its generic representation

Introduction

UNIVERSITY OF OXFORD

- Generic programming: an abstraction technique to reduce code duplication
  - Generic programs operate on "representation types"; a small set of types used to encode all other user-defined datatypes
  - Conversion functions mediate the isomorphism between a datatype and its generic representation
- Several generic programming libraries, specialised for different tasks, supporting different datatypes

2008: regular

$$
\begin{array}{l}
\textbf{class}\ \mathit{GEq}\ \phi\ \textbf{where} \\
\quad \mathit{geq} :: (\alpha \to \alpha \to \mathit{Bool}) \to \phi\ \alpha \to \phi\ \alpha \to \mathit{Bool}
\end{array}
$$

$$
\begin{array}{llllll}
\textbf{instance} & & \mathit{GEq}\ \mathit{Unit} & \textbf{where}\ \mathit{geq}\ _ & \mathit{Unit}\quad \mathit{Unit} & = \mathit{True} \\
\textbf{instance} & \mathit{Eq}\ \alpha \Rightarrow & \mathit{GEq}\ (\mathit{Arg}\ \alpha) & \textbf{where}\ \mathit{geq}\ _ & (\mathit{Arg}\ x)\ (\mathit{Arg}\ y) & = x \equiv y \\
\textbf{instance} & & \mathit{GEq}\ \mathit{Rec} & \textbf{where}\ \mathit{geq}\ \mathit{eqf} & (\mathit{Rec}\ x)\ (\mathit{Rec}\ y) & = \mathit{eqf}\ x\ y
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{GEq}\ \phi, \mathit{GEq}\ \psi) \Rightarrow \mathit{GEq}\ (\phi :+: \psi)\ \textbf{where} \\
\quad \mathit{geq}\ \mathit{eqf}\ (\mathit{Left}\ \ x)\ (\mathit{Left}\ \ y) = \mathit{geq}\ \mathit{eqf}\ x\ y \\
\quad \mathit{geq}\ \mathit{eqf}\ (\mathit{Right}\ x)\ (\mathit{Right}\ y) = \mathit{geq}\ \mathit{eqf}\ x\ y \\
\quad \mathit{geq}\ _\quad _ \qquad\qquad _ \qquad\qquad = \mathit{False}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{GEq}\ \phi, \mathit{GEq}\ \psi) \Rightarrow \mathit{GEq}\ (\phi :\times: \psi)\ \textbf{where} \\
\quad \mathit{geq}\ \mathit{eqf}\ (x_1 :\times: y_1)\ (x_2 :\times: y_2) = \mathit{geq}\ \mathit{eqf}\ x_1\ x_2 \wedge \mathit{geq}\ \mathit{eqf}\ y_1\ y_2
\end{array}
$$

$$
\begin{array}{l}
\mathit{eq} :: (\mathit{Regular}\ \alpha, \mathit{GEq}\ (\mathit{Rep}\ \alpha)) \Rightarrow \alpha \to \alpha \to \mathit{Bool} \\
\mathit{eq}\ x\ y = \mathit{geq}\ \mathit{eq}\ (\mathit{from}\ x)\ (\mathit{from}\ y)
\end{array}
$$

2009: multirec

```
class GEq ρ φ where
  geq :: (∀ι.ρ ι → τ ι → τ ι → Bool) → ρ ι → φ τ ι → φ τ ι → Bool

instance          GEq ρ Unit      where geq _ _ Unit     Unit     = True
instance Eq α ⇒  GEq ρ (Arg α)   where geq _ _ (Arg x) (Arg y)  = x ≡ y
instance El ρ ι ⇒ GEq ρ (Rec ι)  where
  geq eqf _ (Rec   x) (Rec   y) = eqf proof x y
instance (GEq ρ φ, GEq ρ ψ) ⇒ GEq ρ (φ :+: ψ) where
  geq eqf p (Left  x) (Left  y) = geq eqf p x y
  geq eqf p (Right x) (Right y) = geq eqf p x y
  geq _   _ _         _         = False
instance (GEq ρ φ, GEq ρ ψ) ⇒ GEq ρ (φ :×: ψ) where
  geq eqf p (x1 :×: y1) (x2 :×: y2) = geq eqf p x1 x2 ∧ geq eqf p y1 y2
instance GEq ρ φ ⇒ GEq ρ (φ :▷: ι) where
  geq eq p (Tag x) (Tag y) = geq eq p x y

eq :: (Fam ρ, GEq ρ (Rep ρ)) ⇒ ρ ι → ι → ι → Bool
eq p x y = geq (λp (I0 x) (I0 y) → eq p x y) p (from p x) (from p y)
```

2010: generic-deriving

$$\textbf{class } \mathit{GEq}\ \phi \textbf{ where}$$
$$\quad \mathit{geq} :: \phi\ \alpha \to \phi\ \alpha \to \mathit{Bool}$$

$$\textbf{instance } \mathit{GEq}\ \mathit{Unit} \textbf{ where } \mathit{geq}\ \mathit{Unit}\ \mathit{Unit} = \mathit{True}$$
$$\textbf{instance } \mathit{Eq}\ \alpha \Rightarrow \mathit{GEq}\ (\mathit{Arg}\ \iota\ \alpha) \textbf{ where } \mathit{geq}\ (\mathit{Arg}\ x)\ (\mathit{Arg}\ y) = \mathit{geq}\ x\ y$$

$$\textbf{instance } (\mathit{GEq}\ \alpha, \mathit{GEq}\ \beta) \Rightarrow \mathit{GEq}\ (\alpha :+: \beta) \textbf{ where}$$
$$\quad \mathit{geq}\ (\mathit{Left}\ x)\ (\mathit{Left}\ y) = \mathit{geq}\ x\ y$$
$$\quad \mathit{geq}\ (\mathit{Right}\ x)\ (\mathit{Right}\ y) = \mathit{geq}\ x\ y$$
$$\quad \mathit{geq}\ _\ _ = \mathit{False}$$

$$\textbf{instance } (\mathit{GEq}\ \alpha, \mathit{GEq}\ \beta) \Rightarrow \mathit{GEq}\ (\alpha :\times: \beta) \textbf{ where}$$
$$\quad \mathit{geq}\ (x_1 :\times: y_1)\ (x_2 :\times: y_2) = \mathit{geq}\ x_1\ x_2 \wedge \mathit{geq}\ y_1\ y_2$$

$$\mathit{eq} :: (\mathit{Generic}\ \alpha, \mathit{GEq}\ (\mathit{Rep}\ \alpha)) \Rightarrow \alpha \to \alpha \to \mathit{Bool}$$
$$\mathit{eq}\ x\ y = \mathit{geq}\ (\mathit{from}\ x)\ (\mathit{from}\ y)$$

This is generic programming

- "Generic programming (...) makes it possible to write programs that solve a class of problems *once and for all*, instead of writing new code over and over again (...)" [Backhouse et al., 1999]
- "Generic programming techniques aim to *eliminate boilerplate* code." [Lämmel and Peyton Jones, 2003]
- "Generic programming has developed as a technique for increasing the amount and scale of *reuse* in code (...)" [Jeuring et al., 2009]
- "Generic programming (...) *reduces code duplication* and makes code more robust to changes (...)" (Haskell Wiki, 2013)

This is generic programming

UNIVERSITY OF OXFORD

- "Generic programming (...) makes it possible to write programs that solve a class of problems *once and for all*, instead of writing new code over and over again (...)" [Backhouse et al., 1999]
- "Generic programming techniques aim to *eliminate boilerplate* code." [Lämmel and Peyton Jones, 2003]
- "Generic programming has developed as a technique for increasing the amount and scale of *reuse* in code (...)" [Jeuring et al., 2009]
- "Generic programming (...) *reduces code duplication* and makes code more robust to changes (...)" (Haskell Wiki, 2013)

. . . yet generic programmers keep writing boilerplate code!

Duplicated representations across different libraries

Representing lists in `regular`:

$$\textbf{instance}\ \mathit{Regular}\ [\alpha]\ \textbf{where}$$
$$\quad \mathit{Rep}\ [\alpha] = \mathit{Unit}\ {:+:}\ ((\mathit{Arg}\ \alpha)\ {:\times:}\ \mathit{Rec})$$
$$\quad \mathit{from}\ [\,] \qquad = \mathit{Left}\ \mathit{Unit}$$
$$\quad \mathit{from}\ (h : t) = \mathit{Right}\ (\mathit{Arg}\ h\ {:\times:}\ \mathit{Rec}\ t)$$
$$\quad \mathit{to}\ (\mathit{Left}\ \mathit{Unit}) \qquad\qquad\quad = [\,]$$
$$\quad \mathit{to}\ (\mathit{Right}\ (\mathit{Arg}\ h\ {:\times:}\ \mathit{Rec}\ t)) = h : t$$

Representing lists in `generic-deriving`:

$$\textbf{instance}\ \mathit{Generic}\ [\alpha]\ \textbf{where}$$
$$\quad \mathit{Rep}\ [\alpha] = \mathit{Unit}\ {:+:}\ ((\mathit{Arg}\ P\ \alpha)\ {:\times:}\ (\mathit{Arg}\ (R\ S)\ [\alpha]))$$
$$\quad \mathit{from}\ [\,] \qquad = \mathit{Left}\ \ \mathit{Unit}$$
$$\quad \mathit{from}\ (h : t) = \mathit{Right}\ (\mathit{Arg}\ h\ {:\times:}\ \mathit{Arg}\ t)$$
$$\quad \mathit{to}\ (\mathit{Left}\ \ \mathit{Unit}) \qquad\qquad\quad = [\,]$$
$$\quad \mathit{to}\ (\mathit{Right}\ (\mathit{Arg}\ h\ {:\times:}\ \mathit{Arg}\ t)) = h : t$$

What should be possible

```
import Generics.Deriving          as GD
import Generics.Regular.Rewriting as R
import Generics.SYB.Schemes       as S
import Conversions ()

data Logic α = Var α | Logic α :∨: Logic α | Not (Logic α) | T | F
                 deriving (GD.Generic)
```

What should be possible

```
import Generics.Deriving          as GD
import Generics.Regular.Rewriting as R
import Generics.SYB.Schemes       as S
import Conversions ()

data Logic α = Var α | Logic α :∨: Logic α | Not (Logic α) | T | F
                 deriving (GD.Generic)

rewriting :: Logic Char
rewriting =
  let elim2Not = R.rule $ λx → Not (Not x) :⇝: x
  in R.bottomUp (R.rewrite elim2Not) $ T :∨: Not (Not (Var 'p'))
```

What should be possible

```
import Generics.Deriving         as GD
import Generics.Regular.Rewriting as R
import Generics.SYB.Schemes       as S
import Conversions ()

data Logic α = Var α | Logic α :∨: Logic α | Not (Logic α) | T | F
                 deriving (GD.Generic)

rewriting :: Logic Char
rewriting =
  let elim2Not = R.rule $ λx → Not (Not x) :⇝: x
  in R.bottomUp (R.rewrite elim2Not) $ T :∨: Not (Not (Var 'p'))

size :: Int
size = S.everything (+) (const 1) $ Var 'p' :∨: Var 'q'
```

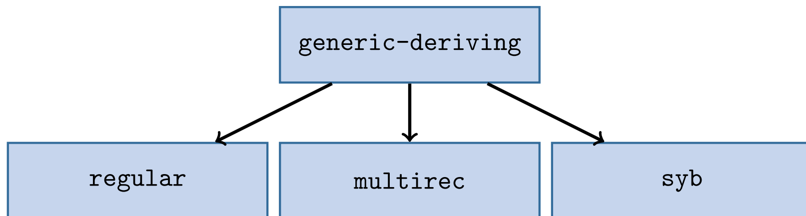
What should be possible

```
import Generics.Deriving         as GD
import Generics.Regular.Rewriting as R
import Generics.SYB.Schemes      as S
import Conversions ()

data Logic α = Var α | Logic α :∨: Logic α | Not (Logic α) | T | F
               deriving (GD.Generic)

rewriting :: Logic Char
rewriting =
  let elim2Not = R.rule $ λx → Not (Not x) :⇝: x
  in R.bottomUp (R.rewrite elim2Not) $ T :∨: Not (Not (Var 'p'))

size :: Int
size = S.everything (+) (const 1) $ Var 'p' :∨: Var 'q'

rename :: Logic String
rename = GD.gmap ('_':) $ T :∨: Var "p"
```

Why?

- Reduce code duplication across generic programming libraries - generic generic programming!
- Reduce code size for generic representations
- Libraries can be specialised, and need not provide general functionality
- No need to rely on Template Haskell for automatically generating representations

How?



We use `generic-deriving` (which has built-in GHC support) to derive conversions to the other libraries.

A closer look at generic-deriving: Universe

$$
\begin{array}{lll}
\textbf{kind}\ \mathit{Univ}_D = & \mathit{Univ}_D\ :+:_D\ \mathit{Univ}_D & \text{-- choice between constructors} \\
 \mid & \mathit{Univ}_D\ :\times:_D\ \mathit{Univ}_D & \text{-- multiple arguments} \\
 \mid & \mathit{Arg}_D\ \mathit{ArgType}\ \star & \text{-- arguments to constructors} \\
 \mid & \mathit{Unit}_D & \text{-- datatypes without constructors}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{kind}\ \mathit{ArgType} = P \mid R\ \mathit{RecType} \mid U & \text{-- parameter, recursion, or unknown} \\
\textbf{kind}\ \mathit{RecType} = S \mid O & \text{-- self or other}
\end{array}
$$

A closer look at generic-deriving: Interpretation

$$
\begin{array}{l}
\textbf{data}\ [\![\, \alpha :: \mathit{Univ}_D \,]\!]_D :: \star\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Unit}_D & :: & [\![\, \mathit{Unit}_D \,]\!]_D \\
\mathit{Arg}_D & :: \alpha & \rightarrow [\![\, \mathit{Arg}_D\ \iota\ \alpha \,]\!]_D \\
\mathit{Left}_D & :: [\![\, \phi \,]\!]_D & \rightarrow [\![\, \phi\ :\!+\!:_D\ \psi \,]\!]_D \\
\mathit{Right}_D & :: [\![\, \psi \,]\!]_D & \rightarrow [\![\, \phi\ :\!+\!:_D\ \psi \,]\!]_D \\
:\!\times\!:_D & :: [\![\, \phi \,]\!]_D \rightarrow [\![\, \psi \,]\!]_D & \rightarrow [\![\, \phi\ :\!\times\!:_D\ \psi \,]\!]_D
\end{array}
\end{array}
$$

A closer look at `generic-deriving`: Datatype instantiation

$$
\begin{array}{l}
\textbf{class}\ \mathit{Generic}_D\ (\alpha :: \star)\ \textbf{where} \\
\quad \mathit{Rep}_D\ \alpha :: \mathit{Univ}_D \\
\quad \mathit{from}_D :: \alpha \to [\![\, \mathit{Rep}_D\ \alpha \,]\!]_D \\
\quad \mathit{to}_D \quad :: [\![\, \mathit{Rep}_D\ \alpha \,]\!]_D \to \alpha
\end{array}
$$

A closer look at `generic-deriving`: Datatype instantiation

$$
\begin{aligned}
&\textbf{class}\ \mathit{Generic}_D\ (\alpha :: \star)\ \textbf{where}\\
&\quad \mathit{Rep}_D\ \alpha :: \mathit{Univ}_D\\
&\quad \mathit{from}_D :: \alpha \to [\![\, \mathit{Rep}_D\ \alpha \,]\!]_D\\
&\quad \mathit{to}_D\ \ :: [\![\, \mathit{Rep}_D\ \alpha \,]\!]_D \to \alpha
\end{aligned}
$$

Example instance (lists):

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Generic}\ [\,\alpha\,]\ \textbf{where}\\
&\quad \mathit{Rep}\ [\,\alpha\,] = \mathit{Unit}_D\ :\!+\!:_D\ ((\mathit{Arg}_D\ P\ \alpha)\ :\!\times\!:_D\ (\mathit{Arg}_D\ (R\ S)\ [\,\alpha\,]))\\
&\quad \mathit{from}\ [\,]\qquad = \mathit{Left}_D\ \mathit{Unit}_D\\
&\quad \mathit{from}\ (h : t) = \mathit{Right}_D\ (\mathit{Arg}_D\ h\ :\!\times\!:_D\ (\mathit{Arg}_D\ t))\\
&\quad \mathit{to}\ (\mathit{Left}_D\ \mathit{Unit}_D) = [\,]\\
&\quad \mathit{to}\ (\mathit{Right}_D\ (\mathit{Arg}_D\ h\ :\!\times\!:_D\ (\mathit{Arg}_D\ t))) = h : t
\end{aligned}
$$

Comparison with regular

$$\textbf{kind}\ \mathit{Univ}_R = \mathit{Unit}_R \mid \mathit{Rec}_R \mid \mathit{Arg}_R\ \star \\ \mid \mathit{Univ}_R\ :\!+\!:_R\ \mathit{Univ}_R \mid \mathit{Univ}_R\ :\!\times\!:_R\ \mathit{Univ}_R$$

Comparison with regular

$$\textbf{kind}\ \mathit{Univ}_R = \mathit{Unit}_R \mid \mathit{Rec}_R \mid \mathit{Arg}_R\ \star$$
$$\mid\ \mathit{Univ}_R\ :\!+\!:_R\ \mathit{Univ}_R \mid \mathit{Univ}_R\ :\!\times\!:_R\ \mathit{Univ}_R$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ [\![\,\alpha :: \mathit{Univ}_R\,]\!]_R\ (\tau :: \star)\ \textbf{where}} \\
\quad \mathit{Unit}_R & :: & [\![\,\mathit{Unit}_R\,]\!]_R\ \tau \\
\quad \mathit{Rec}_R & :: \tau & \to [\![\,\mathit{Rec}_R\,]\!]_R\ \tau \\
\quad \mathit{Arg}_R & :: \alpha & \to [\![\,\mathit{Arg}_R\ \alpha\,]\!]_R\ \tau \\
\quad \mathit{Left}_R & :: [\![\,\alpha\,]\!]_R\ \tau & \to [\![\,\alpha\ :\!+\!:_R\ \beta\,]\!]_R\ \tau \\
\quad \mathit{Right}_R & :: [\![\,\beta\,]\!]_R\ \tau & \to [\![\,\alpha\ :\!+\!:_R\ \beta\,]\!]_R\ \tau \\
\quad (:\!\times\!:_R) & :: [\![\,\alpha\,]\!]_R\ \tau \to [\![\,\beta\,]\!]_R\ \tau & \to [\![\,\alpha\ :\!\times\!:_R\ \beta\,]\!]_R\ \tau
\end{array}
$$

Comparison with regular

$$
\begin{aligned}
&\textbf{kind}\ \mathit{Univ}_R = \mathit{Unit}_R \mid \mathit{Rec}_R \mid \mathit{Arg}_R\ \star \\
&\qquad\qquad \mid \mathit{Univ}_R\ :\!+\!:_R\ \mathit{Univ}_R \mid \mathit{Univ}_R\ :\!\times\!:_R\ \mathit{Univ}_R
\end{aligned}
$$

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\textbf{data}\ [\![\, \alpha :: \mathit{Univ}_R \,]\!]_R\ (\tau :: \star)\ \textbf{where}} \\
\quad \mathit{Unit}_R & :: & & & [\![\, \mathit{Unit}_R \,]\!]_R\ \tau \\
\quad \mathit{Rec}_R & :: & \tau & \rightarrow & [\![\, \mathit{Rec}_R \,]\!]_R\ \tau \\
\quad \mathit{Arg}_R & :: & \alpha & \rightarrow & [\![\, \mathit{Arg}_R\ \alpha \,]\!]_R\ \tau \\
\quad \mathit{Left}_R & :: & [\![\, \alpha \,]\!]_R\ \tau & \rightarrow & [\![\, \alpha\ :\!+\!:_R\ \beta \,]\!]_R\ \tau \\
\quad \mathit{Right}_R & :: & [\![\, \beta \,]\!]_R\ \tau & \rightarrow & [\![\, \alpha\ :\!+\!:_R\ \beta \,]\!]_R\ \tau \\
\quad (:\!\times\!:_R) & :: & [\![\, \alpha \,]\!]_R\ \tau \rightarrow [\![\, \beta \,]\!]_R\ \tau & \rightarrow & [\![\, \alpha\ :\!\times\!:_R\ \beta \,]\!]_R\ \tau
\end{array}
$$

$$
\begin{aligned}
&\textbf{class}\ \mathit{Regular}\ (\alpha :: \star)\ \textbf{where} \\
&\quad \mathit{Rep}\ \alpha :: \mathit{Univ}_R \\
&\quad \mathit{from}_R :: \alpha \rightarrow [\![\, \mathit{Rep}\ \alpha \,]\!]_R\ \alpha
\end{aligned}
$$

From `generic-deriving` to `regular`: types

Our conversion from `generic-deriving` to `regular` consists of two steps:

- A type function *ToRegular* to convert the type representations
- A value-level function *toRegular* to convert the value accordingly

From `generic-deriving` to `regular`: types

Our conversion from `generic-deriving` to `regular` consists of two steps:

- A type function *ToRegular* to convert the type representations
- A value-level function *toRegular* to convert the value accordingly

$$
\begin{array}{lcl}
\mathit{ToRegular}\ (\alpha :: \mathit{Univ}_D) :: \mathit{Univ}_R \\
\mathit{ToRegular}\ \mathit{Unit}_D & = & \mathit{Unit}_R \\
\mathit{ToRegular}\ (\alpha\ :+:_D\ \beta) & = & \mathit{ToRegular}\ \alpha\ :+:_R\ \mathit{ToRegular}\ \beta \\
\mathit{ToRegular}\ (\alpha\ :\times:_D\ \beta) & = & \mathit{ToRegular}\ \alpha\ :\times:_R\ \mathit{ToRegular}\ \beta \\
\mathit{ToRegular}\ (\mathit{Arg}_D\ (R\ S)\ \tau) & = & \mathit{Rec}_R \\
\mathit{ToRegular}\ (\mathit{Arg}_D\ (R\ O)\ \alpha) & = & \mathit{Arg}_R\ \alpha \\
\mathit{ToRegular}\ (\mathit{Arg}_D\ P\ \alpha) & = & \mathit{Arg}_R\ \alpha \\
\mathit{ToRegular}\ (\mathit{Arg}_D\ U\ \alpha) & = & \mathit{Arg}_R\ \alpha
\end{array}
$$

From generic-deriving to regular: values

$$
\begin{array}{l}
\textbf{class } \mathit{Convert}_R\ (\alpha :: \mathit{Univ}_D)\ \tau \textbf{ where}\\
\quad \mathit{toRegular} :: [\![\, \alpha \,]\!]_D\ \rho \to [\![\, \mathit{ToRegular}\ \alpha \,]\!]_R\ \tau\\
\\
\textbf{instance } \mathit{Convert}_R\ (\mathit{Arg}_D\ (R\ S)\ \tau)\ \tau \textbf{ where}\\
\quad \mathit{toRegular}\ (\mathit{Arg}_D\ x) = \mathit{Rec}_R\ x\\
\textbf{instance } \mathit{Convert}_R\ (\mathit{Arg}_D\ \gamma\ \tau)\ \rho \textbf{ where}\\
\quad \mathit{toRegular}\ (\mathit{Arg}_D\ x) = \mathit{Arg}_R\ x\\
\textbf{instance } (\mathit{Convert}_R\ \alpha\ \tau, \mathit{Convert}_R\ \beta\ \tau)\\
\qquad\qquad \Rightarrow \mathit{Convert}_R\ (\alpha :+:_D \beta)\ \tau \textbf{ where}\\
\quad \mathit{toRegular}\ (\mathit{Left}_D\ x) = \mathit{Left}_R\ (\mathit{toRegular}\ x)\\
\quad \mathit{toRegular}\ (\mathit{Right}_D\ x) = \mathit{Right}_R\ (\mathit{toRegular}\ x)\\
\textbf{instance } (\mathit{Convert}_R\ \alpha\ \tau, \mathit{Convert}_R\ \beta\ \tau)\\
\qquad\qquad \Rightarrow \mathit{Convert}_R\ (\alpha :\times:_D \beta)\ \tau \textbf{ where}\\
\quad \mathit{toRegular}\ (x :\times:_D y) = \mathit{toRegular}\ x :\times:_R \mathit{toRegular}\ y\\
\ldots
\end{array}
$$

From `generic-deriving` to `regular`: datatypes

It is here that we set the second parameter of $\mathit{Convert}_R$ to the type being converted (α):

$$\textbf{instance}\ (\mathit{Generic}_D\ \alpha, \mathit{Convert}_R\ (\mathit{Rep}_D\ \alpha)\ \alpha) \Rightarrow \mathit{Regular}\ \alpha\ \textbf{where}$$
$$\mathit{Rep}\ \alpha = \mathit{ToRegular}\ (\mathit{Rep}_D\ \alpha)$$
$$\mathit{from}_R\ x = \mathit{toRegular}\ (\mathit{from}_D\ x)$$

With this instance, functions defined in the `regular` library are now available to all `generic-deriving` supported datatypes.

The Spine view (syb)

Encoding syb via the Spine view:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Spine} :: \star \rightarrow \star\ \textbf{where} \\
\quad \mathit{Con} :: \alpha \rightarrow \mathit{Spine}\ \alpha \\
\quad (:\diamond:) :: (\mathit{Data}\ \alpha) \Rightarrow \mathit{Spine}\ (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \mathit{Spine}\ \beta
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Typeable}\ \alpha) \Rightarrow \mathit{Data}\ \alpha\ \textbf{where} \\
\quad \mathit{spine} :: \alpha \rightarrow \mathit{Spine}\ \alpha
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Data}\ \alpha) \Rightarrow \mathit{Data}\ [\,\alpha\,]\ \textbf{where} \\
\quad \mathit{spine}\ [\,] \qquad\;\; = \mathit{Con}\ [\,] \\
\quad \mathit{spine}\ (h : t) = \mathit{Con}\ (:)\ :\diamond:\ h\ :\diamond:\ t
\end{array}
$$

From generic-deriving to syb

Representing generic-deriving types as a Spine:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Converts}_S\ (\alpha :: \mathit{Univ}_D)\ \textbf{where} \\
\quad \mathit{toSpine} :: [\![\, \alpha \,]\!]_D \rightarrow \mathit{Spine}\ ([\![\, \alpha \,]\!]_D)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Converts}_S\ \mathit{Unit}_D\ \textbf{where}\ \mathit{toSpine}\ \mathit{Unit}_D = \mathit{Con}\ \mathit{Unit}_D \\
\textbf{instance}\ (\mathit{Converts}_S\ \alpha, \mathit{Converts}_S\ \beta) \Rightarrow \mathit{Converts}_S\ (\alpha\ :+:_D\ \beta)\ \textbf{where} \\
\quad \mathit{toSpine}\ (\mathit{Left}_D\ \ x) = \mathit{fmap}\ \mathit{Left}_D\ \ (\mathit{toSpine}\ x) \\
\quad \mathit{toSpine}\ (\mathit{Right}_D\ x) = \mathit{fmap}\ \mathit{Right}_D\ (\mathit{toSpine}\ x) \\
\textbf{instance}\ (\mathit{Data}\ \alpha) \Rightarrow \mathit{Converts}_S\ (\mathit{Arg}_D\ \iota\ \alpha)\ \textbf{where} \\
\quad \mathit{toSpine}\ (\mathit{Arg}_D\ x) = \mathit{Con}\ \mathit{Arg}_D\ :\diamond:\ x
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Generic}_D\ \alpha, \mathit{Converts}_S\ (\mathit{Rep}_D\ \alpha), \mathit{Typeable}\ \alpha) \\
\quad \Rightarrow \mathit{Data}\ \alpha\ \textbf{where} \\
\quad \mathit{spine} = \mathit{fmap}\ \mathit{to}_D \circ \mathit{toSpine} \circ \mathit{from}_D
\end{array}
$$

Further reading

In the paper we also have a conversion from `generic-deriving` to `multirec`.

Full code (compiling with GHC 7.6.2) available at `http://dreixel.net/research/code/ggp.zip`.

Future work

- What is the performance penalty imposed by the conversions?
- Can we optimise it using "known techniques"?
- Which other libraries can we convert to?

Conclusion

UNIVERSITY OF OXFORD

We have:

- Recognised the existence of code duplication across generic programming libraries
- Eliminated this duplication by defining conversions between approaches
- Proposed a new approach to the development of generic programming, libraries, focusing on how to relate them to existing approaches, rather than reimplementing everything from scratch

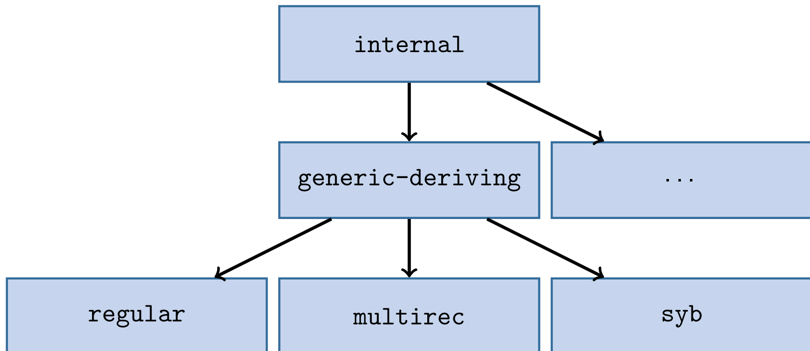
Conclusion

We have:

- Recognised the existence of code duplication across generic programming libraries
- Eliminated this duplication by defining conversions between approaches
- Proposed a new approach to the development of generic programming, libraries, focusing on how to relate them to existing approaches, rather than reimplementing everything from scratch

Thank you for your time!

How?



Bibliography

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In Doaitse S. Swierstra, Pedro R. Henriques, and José Nuno Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608, pages 28–115. Springer-Verlag, 1999. doi:10.1007/10704973_2.

Johan Jeuring, Sean Leather, José Pedro Magalhães, and Alexey Rodriguez Yakushev. Libraries for generic programming in Haskell. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 165–229. Springer, 2009. URL `http://dreixel.net/research/pdf/lgph.pdf`. ISBN-13 978-3-642-04651-3.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37. ACM, 2003. doi:10.1145/604174.604179.