Generic Generic Programming

José Pedro Magalhães
(joint work with Andres Löh)

Department of Computer Science, University of Oxford

July 2013

2008: `regular`

```
class GEq φ where
  geq :: (α → α → Bool) → φ α → φ α → Bool

instance           GEq U      where geq _   U     U     = True
instance Eq α ⇒ GEq (K α)     where geq _   (K x) (K y) = x ≡ y
instance           GEq I      where geq eqf (I  x) (I  y) = eqf x y
instance (GEq φ, GEq ψ) ⇒ GEq (φ :+: ψ) where
  geq eqf (L x) (L y) = geq eqf x y
  geq eqf (R x) (R y) = geq eqf x y
  geq _   _     _     = False
instance (GEq φ, GEq ψ) ⇒ GEq (φ :×: ψ) where
  geq eqf (x1 :×: y1) (x2 :×: y2) = geq eqf x1 x2 ∧ geq eqf y1 y2

eq :: (Regular α, GEq (PF α)) ⇒ α → α → Bool
eq x y = geq eq (from x) (from y)
```

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 U       where geq _   _ U     U     = True
instance Eq α ⇒   GEq ρ (K α)   where geq _   _ (K x) (K y) = x ≡ y
instance El ρ ι ⇒ GEq ρ (I ι)   where geq eqf _ (I  x) (I  y) = eqf proof x y

instance (GEq ρ φ, GEq ρ ψ) ⇒ GEq ρ (φ :+: ψ) where
  geq eqf p (L x) (L y) = geq eqf p x y
  geq eqf p (R x) (R 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 (PF 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

```
class GEq φ where
  geq :: φ α → φ α → Bool

instance          GEq U1          where geq U1      U1      = True
instance Eq α ⇒ GEq (K1 ι α) where geq (K1 x) (K1 y) = geq x y

instance (GEq α, GEq β) ⇒ GEq (α :+: β) where
  geq (L1 x) (L1 y) = geq x y
  geq (R1 x) (R1 y) = geq x y
  geq _      _      = False

instance (GEq α, GEq β) ⇒ GEq (α :×: β) where
  geq (x1 :×: y1) (x2 :×: y2) = geq x1 x2 ∧ geq y1 y2

eq :: (Generic α, GEq (Rep α)) ⇒ α → α → Bool
eq x y = geq (from x) (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

- "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`:

```
instance Regular [α] where
  PF [α] = U1 :+: ((K α) :×: I)
  from []      = L U
  from (h : t) = R (K h :×: I t)
  to (L U)              = []
  to (R (K h :×: I t)) = h : t
```

Representing lists in `generic-deriving`:

```
instance Generic [α] where
  Rep [α] = U1 :+: ((Par0 α) :×: (Rec0 [α]))
  from []      = L1 U1
  from (h : t) = R1 (Par h :×: Rec t)
  to (L1 U1)                = []
  to (R1 (Par h :×: Rec t)) = h : t
```

Two instances in `generic-deriving`

```
instance Generic [α] where
  Rep [α] = U1 :+: ((Par0 α) :×: (Rec0 [α]))
  from []       = L1 U1
  from (h : t)  = R1 (Par h :×: Rec t)
  to (L1 U1)                  = []
  to (R1 (Par h :×: Rec t))   = h : t

instance Generic1 [] where
  Rep1 [] = U1 :+: (Par1 :×: (Rec1 []))
  from1 []       = L1 U1
  from1 (h : t)  = R1 (Par1 h :×: Rec1 t)
  to1 (L1 U1)                  = Nil
  to1 (R1 (Par1 h :×: Rec1 t)) = Cons h t
```

Two instances in `generic-deriving`

```
instance Generic [α] where
  Rep [α] = U1 :+: ((Par0 α) :×: (Rec0 [α]))
  from []       = L1 U1
  from (h : t)  = R1 (Par h :×: Rec t)
  to (L1 U1)                  = []
  to (R1 (Par h :×: Rec t))   = h : t

instance Generic1 [] where
  Rep1 [] = U1 :+: (Par1 :×: (Rec1 []))
  from1 []       = L1 U1
  from1 (h : t)  = R1 (Par1 h :×: Rec1 t)
  to1 (L1 U1)                  = Nil
  to1 (R1 (Par1 h :×: Rec1 t)) = Cons h t
```

The $Generic_1\ [\,]$ instance has more information than the $Generic\ [\alpha]$ instance. Why not derive $Generic$ instances from $Generic_1$ instances?

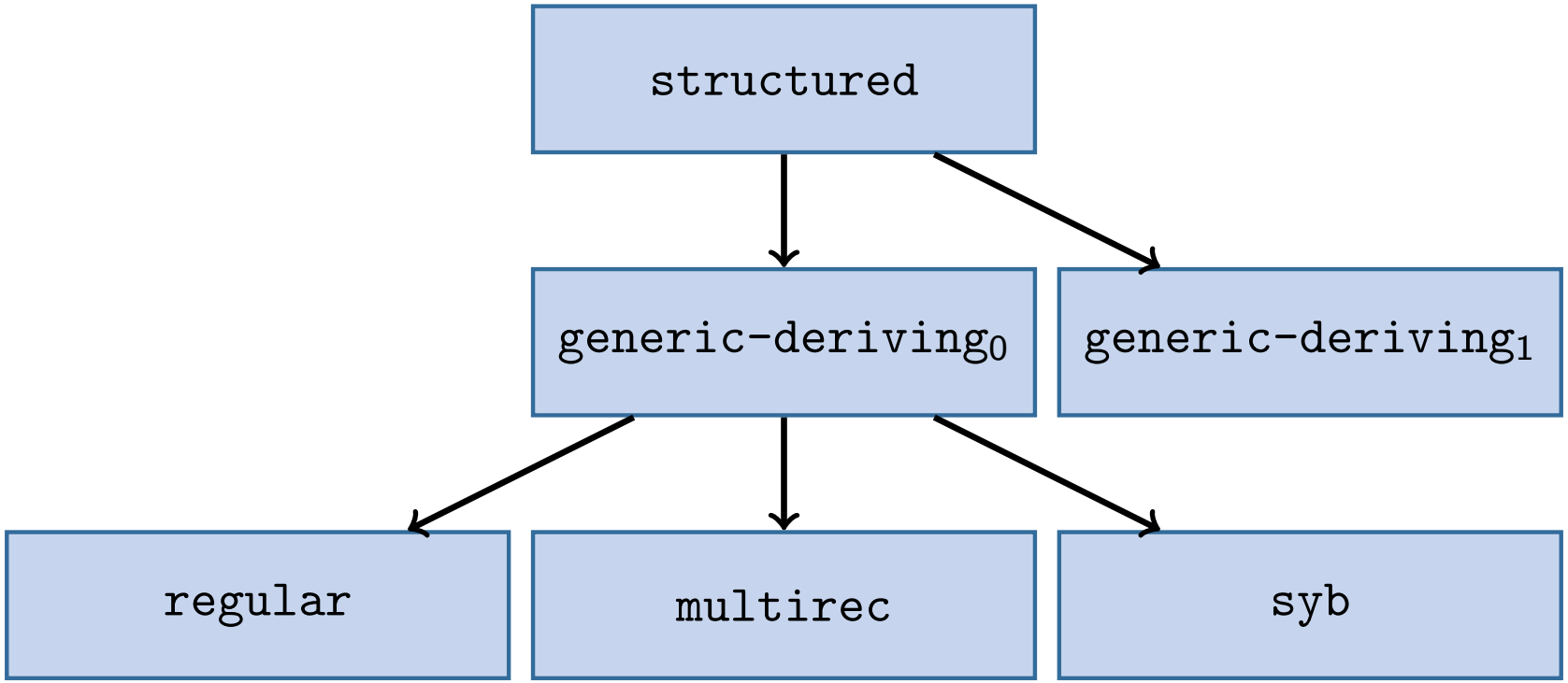
What should be possible

```
import Generics.Deriving
import Generics.Deriving.Functions.Monoid as D
import Generics.Regular.Functions.Fold    as R
import Generics.SYB.Schemes               as S
import Conversions ()

data Logic = Logic :∧: Logic | Logic :∨: Logic
           | Not Logic | T | F deriving Generic

test :: (Logic, Bool, Int)
test = (D.gmempty, R.fold alg term, S.gsize term)
  where term = T :∨: F
        alg  = (∧) & (∨) & not & True & False
```

Why?

- Reduce code duplication across generic programming libraries
- 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 introduce a new library, `structured`, with built-in GHC support, and derive conversions from there to the other libraries.

`structured`: representation types

$$
\begin{aligned}
&\textbf{kind } \mathit{Data} \;\; = \mathit{Data}\ \mathit{MetaData}\ (\mathit{Tree}\ \mathit{Con}) \\
&\textbf{kind } \mathit{Con} \;\; = \mathit{Con}\ \;\mathit{MetaCon}\ \;(\mathit{Tree}\ \mathit{Field}) \\
&\textbf{kind } \mathit{Field} \;\; = \mathit{Field}\ \mathit{MetaField}\ \mathit{Arg} \\
&\textbf{kind } \mathit{Tree}\ \kappa = \mathit{Empty} \mid \mathit{Leaf}\ \ \kappa \mid \mathit{Bin}\ (\mathit{Tree}\ \kappa)\ (\mathit{Tree}\ \kappa)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{kind } \mathit{Arg} = \mathit{K}\ \mathit{KType}\ \star \\
&\qquad\qquad \mid \ \mathit{Rec}\ \mathit{RecType}\ (\star \to \star) \\
&\qquad\qquad \mid \ \mathit{Par} \\
&\qquad\qquad \mid \ (\star \to \star)\ :\!\circ\!:\ \mathit{Arg} \\
&\textbf{kind } \mathit{KType} \;\; = \mathit{P} \mid \mathit{R}\ \mathit{RecType} \mid \mathit{U} \\
&\textbf{kind } \mathit{RecType} = \mathit{S} \mid \mathit{O}
\end{aligned}
$$

structured: meta data

```
kind MetaData = MD Symbol     -- datatype name
              | Symbol        -- datatype module name
              | Bool          -- is it a newtype?
kind MetaCon  = MC Symbol     -- constructor name
              | Fixity        -- constructor fixity
              | Bool          -- does it use record syntax?
kind MetaField = MF (Maybe Symbol)   -- field name
kind Fixity        = Prefix | Infix Associativity Nat
kind Associativity = LeftAssociative
                   | RightAssociative
                   | NotAssociative
kind Nat = Ze | Su Nat
kind Symbol   -- internal
```

structured: representation values

$$
\begin{aligned}
&\textbf{data family } [\![\ \_\ ]\!] :: \kappa \to \star \to \star \\
&\textbf{data instance } [\![\ \upsilon :: \mathit{Data}\ ]\!]\ \rho \textbf{ where} \\
&\quad \mathit{D} \quad :: [\![\ \alpha\ ]\!]\ \rho \to [\![\ \mathit{Data}\ \iota\ \alpha\ ]\!]\ \rho \\
&\textbf{data instance } [\![\ \upsilon :: \mathit{Tree}\ \mathit{Con}\ ]\!]\ \rho \textbf{ where} \\
&\quad \mathit{C} \quad :: [\![\ \alpha\ ]\!]\ \rho \to [\![\ \mathit{Leaf}\ (\mathit{Con}\ \iota\ \alpha)\ ]\!]\ \rho \\
&\quad \mathit{L} \quad :: [\![\ \alpha\ ]\!]\ \rho \to [\![\ \mathit{Bin}\ \alpha\ \beta\ ]\!]\ \rho \\
&\quad \mathit{R} \quad :: [\![\ \beta\ ]\!]\ \rho \to [\![\ \mathit{Bin}\ \alpha\ \beta\ ]\!]\ \rho \\
&\textbf{data instance } [\![\ \upsilon :: \mathit{Tree}\ \mathit{Field}\ ]\!]\ \rho \textbf{ where} \\
&\quad \mathit{U} \quad :: [\![\ \mathit{Empty}\ ]\!]\ \rho \\
&\quad \mathit{S} \quad :: [\![\ \alpha\ ]\!]\ \rho \to [\![\ \mathit{Leaf}\ (\mathit{Field}\ \iota\ \alpha)\ ]\!]\ \rho \\
&\quad (\times) :: [\![\ \alpha\ ]\!]\ \rho \to [\![\ \beta\ ]\!]\ \rho \to [\![\ \mathit{Bin}\ \alpha\ \beta\ ]\!]\ \rho \\
&\textbf{data instance } [\![\ \upsilon :: \mathit{Arg}\ ]\!]\ \rho \textbf{ where} \\
&\quad \mathit{K} \quad :: \{\mathit{unK} :: \alpha\} \to [\![\ \mathit{K}\ \iota\ \alpha\ ]\!]\ \rho \\
&\quad \mathit{Rec} :: \{\mathit{unRec} :: \phi\ \rho\} \to [\![\ \mathit{Rec}\ \iota\ \phi\ ]\!]\ \rho \\
&\quad \mathit{Par} :: \{\mathit{unPar} :: \rho\} \to [\![\ \mathit{Par}\ ]\!]\ \rho \\
&\quad \mathit{Comp} :: \{\mathit{unComp} :: \sigma\ ([\![\ \phi\ ]\!]\ \rho)\} \to [\![\ \sigma :\circ: \phi\ ]\!]\ \rho
\end{aligned}
$$

`structured`: embedding user datatypes

```
class Generic (α :: ⋆) where
  Rep  α :: Data
  Par_g α :: ⋆
  Par_g α = NoPar
  from :: α → [[ Rep φ ]] (Par_g α)
  to   :: [[ Rep φ ]] (Par_g α) → α

data NoPar   -- empty
```

The Par_g trick allows us to have a single class, instead of two like `generic-deriving`.

Representing lists in `structured`

```
MC_Nil  = MC "[]" Prefix                          False
MC_Cons = MC ":"  (Infix RightAssociative 5)      False
H       = Leaf (Field (MF Nothing) Par)
T       = Leaf (Field (MF Nothing) (Rec S []))

instance Generic [α] where
  Rep [α] = Data (MD "[]" "Prelude" False)
                 (Bin (Leaf (Con MC_Nil Empty))
                      (Leaf (Con MC_Cons (Bin H T))))
  Par_g [α] = α
  from []      = D (L (C U))
  from (h : t) = D (R (C (S (Par h) :×: S (Rec t))))
  to (D (L (C U)))                                = []
  to (D (R (C (S (Par h) :×: S (Rec t)))))) = h : t
```

What do we gain with `structured`?

At first sight, not much:

- More hierarchy in the representation types
- Trees, instead of sums and products
- One single instance for each user datatype, instead of two
- Still supporting at most one datatype parameter
- No GADTs, higher-kinded arguments, etc.

What do we gain with `structured`?

At first sight, not much:

- More hierarchy in the representation types
- Trees, instead of sums and products
- One single instance for each user datatype, instead of two
- Still supporting at most one datatype parameter
- No GADTs, higher-kinded arguments, etc.

But, in fact, we gain a lot:

- We're not changing `generic-deriving`, so existing programs keep working
- We decouple `generic-deriving` from GHC
- We can evolve `structured` easily in the future

On to conversions

A conversion from one library to another consists normally of two things:

- A type function to convert the type representations
- A value-level function to convert the value accordingly

Recall `regular`

$$\textbf{kind}\ Un_R = U_R \mid I_R \mid K_R\ \star \mid Un_R\ :+:_R\ Un_R \mid Un_R\ :\times:_R\ Un_R$$

Recall `regular`

$$\textbf{kind}\ Un_R = U_R \mid I_R \mid K_R\ \star \mid Un_R\ \ :+:_R\ \ Un_R \mid Un_R\ \ :\times:_R\ \ Un_R$$

$$\begin{aligned}
&\textbf{data}\ [\![\, \alpha :: Un_R \,]\!]_R\ (\tau :: \star)\ \textbf{where} \\
&\quad U_R \quad\ \ :: [\![\, U_R \,]\!]_R\ \tau \\
&\quad I_R \quad\ \ :: \tau \to [\![\, I_R \,]\!]_R\ \tau \\
&\quad K_R \quad\ \ :: \alpha \to [\![\, K_R\ \alpha \,]\!]_R\ \tau \\
&\quad L_R \quad\ \ :: [\![\, \alpha \,]\!]_R\ \tau \to [\![\, \alpha\ \ :+:_R\ \ \beta \,]\!]_R\ \tau \\
&\quad R_R \quad\ \ :: [\![\, \beta \,]\!]_R\ \tau \to [\![\, \alpha\ \ :+:_R\ \ \beta \,]\!]_R\ \tau \\
&\quad (:\times:_R) :: [\![\, \alpha \,]\!]_R\ \tau \to [\![\, \beta \,]\!]_R\ \tau \to [\![\, \alpha\ \ :\times:_R\ \ \beta \,]\!]_R\ \tau
\end{aligned}$$

Recall `regular`

$$\textbf{kind}\ Un_R = U_R \mid I_R \mid K_R \star \mid Un_R\ :+:_R\ Un_R \mid Un_R\ :\times:_R\ Un_R$$

$$\begin{aligned}
&\textbf{data}\ [\![\, \alpha :: Un_R \,]\!]_R\ (\tau :: \star)\ \textbf{where} \\
&\quad U_R \quad :: [\![\, U_R \,]\!]_R\ \tau \\
&\quad I_R \quad :: \tau \to [\![\, I_R \,]\!]_R\ \tau \\
&\quad K_R \quad :: \alpha \to [\![\, K_R\ \alpha \,]\!]_R\ \tau \\
&\quad L_R \quad :: [\![\, \alpha \,]\!]_R\ \tau \to [\![\, \alpha\ :+:_R\ \beta \,]\!]_R\ \tau \\
&\quad R_R \quad :: [\![\, \beta \,]\!]_R\ \tau \to [\![\, \alpha\ :+:_R\ \beta \,]\!]_R\ \tau \\
&\quad (:\times:_R) :: [\![\, \alpha \,]\!]_R\ \tau \to [\![\, \beta \,]\!]_R\ \tau \to [\![\, \alpha\ :\times:_R\ \beta \,]\!]_R\ \tau
\end{aligned}$$

$$\begin{aligned}
&\textbf{class}\ Regular\ (\alpha :: \star)\ \textbf{where} \\
&\quad PF\ \alpha :: Un_R \\
&\quad from_R :: \alpha \to [\![\, PF\ \alpha \,]\!]_R\ \alpha
\end{aligned}$$

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

$$
\begin{aligned}
&D_{\to} R\ (\alpha :: Un_D) :: Un_R \\
&D_{\to} R\ U_D &&= U_R \\
&D_{\to} R\ (M_D\ \iota\ \alpha) &&= D_{\to} R\ \alpha \\
&D_{\to} R\ (\alpha\ :+:_D\ \beta) &&= D_{\to} R\ \alpha\ :+:_R\ D_{\to} R\ \beta \\
&D_{\to} R\ (\alpha\ :\times:_D\ \beta) &&= D_{\to} R\ \alpha\ :\times:_R\ D_{\to} R\ \beta \\
&D_{\to} R\ (K_D\ (R\ S)\ \tau) &&= I_R \\
&D_{\to} R\ (K_D\ (R\ O)\ \alpha) &&= K_R\ \alpha \\
&D_{\to} R\ (K_D\ P\ \ \ \alpha) &&= K_R\ \alpha \\
&D_{\to} R\ (K_D\ U\ \ \ \alpha) &&= K_R\ \alpha
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{class } \mathit{Convert}_{D\to R}\ (\alpha :: \mathit{Un}_D)\ \tau \textbf{ where}\\
&\quad d_{\to} r :: [\![\, \alpha \,]\!]_D\ \rho \to [\![\, D_{\to} R\ \alpha \,]\!]_R\ \tau\\
&\\
&\textbf{instance } \mathit{Convert}_{D\to R}\ (K_D\ (R\ S)\ \tau)\ \tau \textbf{ where}\\
&\quad d_{\to} r\ (K_{1D}\ x) = I_R\ x\\
&\textbf{instance } \mathit{Convert}_{D\to R}\ (K_D\ \gamma\ \ \ \ \tau)\ \rho \textbf{ where}\\
&\quad d_{\to} r\ (K_{1D}\ x) = K_R\ x\\
&\textbf{instance } (\mathit{Convert}_{D\to R}\ \alpha\ \tau, \mathit{Convert}_{D\to R}\ \beta\ \tau)\\
&\qquad\qquad \Rightarrow \mathit{Convert}_{D\to R}\ (\alpha\ :+:_D\ \beta)\ \tau \textbf{ where}\\
&\quad d_{\to} r\ (L_{1D}\ x) = L_R\ (d_{\to} r\ x)\\
&\quad d_{\to} r\ (R_{1D}\ x) = R_R\ (d_{\to} r\ x)\\
&\textbf{instance } (\mathit{Convert}_{D\to R}\ \alpha\ \tau, \mathit{Convert}_{D\to R}\ \beta\ \tau)\\
&\qquad\qquad \Rightarrow \mathit{Convert}_{D\to R}\ (\alpha\ :\times:_D\ \beta)\ \tau \textbf{ where}\\
&\quad d_{\to} r\ (x\ :\times:_D\ y) = d_{\to} r\ x\ :\times:_R\ d_{\to} r\ y\\
&\ldots
\end{aligned}
$$

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

It is here that we set the second parameter of $Convert_{D_\to R}$ to the type being converted (α):

$$
\begin{aligned}
&\textbf{instance}\ (Generic_D\ \alpha, Convert_{D_\to R}\ (Rep_D\ \alpha)\ \alpha) \\
&\qquad \Rightarrow Regular\ \alpha\ \textbf{where} \\
&\quad PF\ \alpha = D_\to R\ (Rep_D\ \alpha) \\
&\quad from_R\ \ x = d_\to r\ (from_D\ x)
\end{aligned}
$$

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

Further reading

In the paper (submitted to the Haskell Symposium) we have:

- Conversion from `structured` to `generic-deriving`
- Conversion from `generic-deriving` to `multirec`
- Conversion from `generic-deriving` to `syb`
- Conversion between different balancings of sums and products

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

To do

Food for thought:

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

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.