

Generic Generic Programming

José Pedro Magalhães
(joint work with Andres Löh)

Department of Computer Science, University of Oxford

July 2013

class *GEq* ϕ where

geq :: ($\alpha \rightarrow \alpha \rightarrow \text{Bool}$) $\rightarrow \phi \alpha \rightarrow \phi \alpha \rightarrow \text{Bool}$

instance *GEq* *U* where *geq* _ *U* *U* = *True*

instance *Eq* $\alpha \Rightarrow$ *GEq* (*K* α) where *geq* _ (*K* *x*) (*K* *y*) = $x \equiv y$

instance *GEq* *I* where *geq* *eqf* (*I* *x*) (*I* *y*) = *eqf* *x* *y*

instance (*GEq* ϕ , *GEq* ψ) \Rightarrow *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) \Rightarrow *GEq* (ϕ $:\times$ ψ) where

geq *eqf* ($x_1 \times y_1$) ($x_2 \times y_2$) = *geq* *eqf* x_1 $x_2 \wedge$ *geq* *eqf* y_1 y_2

eq :: (*Regular* α , *GEq* (*PF* α)) $\Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Bool}$

eq *x* *y* = *geq* *eq* (*from* *x*) (*from* *y*)

class **GEq** ρ ϕ where

$geq :: (\forall \iota. \rho \iota \rightarrow \tau \iota \rightarrow \tau \iota \rightarrow Bool) \rightarrow \rho \iota \rightarrow \phi \tau \iota \rightarrow \phi \tau \iota \rightarrow Bool$

instance **GEq** ρ **U** where geq _ _ **U** **U** = **True**

instance **Eq** $\alpha \Rightarrow$ **GEq** ρ (**K** α) where geq _ _ (**K** x) (**K** y) = $x \equiv y$

instance **El** $\rho \iota \Rightarrow$ **GEq** ρ (**I** ι) where geq eqf _ (**I** x) (**I** y) = eqf $proof$ x y

instance (**GEq** ρ ϕ , **GEq** ρ ψ) \Rightarrow **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) \Rightarrow **GEq** ρ (ϕ **:x:** ψ) where

geq eqf p (x_1 **:x:** y_1) (x_2 **:x:** y_2) = geq eqf p x_1 $x_2 \wedge geq$ eqf p y_1 y_2

instance **GEq** ρ $\phi \Rightarrow$ **GEq** ρ (ϕ **:v:** ι) where

geq eq p (**Tag** x) (**Tag** y) = geq eq p x y

$eq :: (Fam$ ρ , **GEq** ρ (**PF** ρ)) \Rightarrow $\rho \iota \rightarrow \iota \rightarrow \iota \rightarrow Bool$

eq p x y = geq (λp (**I** x) (**I** y) $\rightarrow eq$ p x y) p ($from$ p x) ($from$ p y)

class *GEq* ϕ **where**

geq :: $\phi \alpha \rightarrow \phi \alpha \rightarrow \text{Bool}$

instance *GEq* U_1 **where** *geq* U_1 U_1 = *True*

instance *Eq* $\alpha \Rightarrow \text{GEq} (K_1 \iota \alpha)$ **where** *geq* (K_1 x) (K_1 y) = *geq* x y

instance (*GEq* α , *GEq* β) \Rightarrow *GEq* (α $:+$ β) **where**

geq (L_1 x) (L_1 y) = *geq* x y

geq (R_1 x) (R_1 y) = *geq* x y

geq - - = *False*

instance (*GEq* α , *GEq* β) \Rightarrow *GEq* (α $:\times$ β) **where**

geq (x_1 $:\times$ y_1) (x_2 $:\times$ y_2) = *geq* x_1 x_2 \wedge *geq* y_1 y_2

eq :: (*Generic* α , *GEq* (*Rep* α)) \Rightarrow $\alpha \rightarrow \alpha \rightarrow \text{Bool}$

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* $[\alpha]$ **where**

$PF [\alpha] = U_1 \text{ :+} : ((K \alpha) \text{ :}\times\text{ :} I)$

from $[] = L U$

from $(h : t) = R (K h \text{ :}\times\text{ :} I t)$

to $(L U) = []$

to $(R (K h \text{ :}\times\text{ :} I t)) = h : t$

Representing lists in generic-deriving:

instance *Generic* $[\alpha]$ **where**

$Rep [\alpha] = U_1 \text{ :+} : ((Par_0 \alpha) \text{ :}\times\text{ :} (Rec_0 [\alpha]))$

from $[] = L_1 U_1$

from $(h : t) = R_1 (Par h \text{ :}\times\text{ :} Rec t)$

to $(L_1 U_1) = []$

to $(R_1 (Par h \text{ :}\times\text{ :} Rec t)) = h : t$

Two instances in generic-deriving



instance *Generic* [α] **where**

Rep [α] = U_1 :+: ((*Par*₀ α) :×: (*Rec*₀ [α]))

from [] = $L_1 U_1$

from ($h : t$) = R_1 (*Par* h :×: *Rec* t)

to ($L_1 U_1$) = []

to (R_1 (*Par* h :×: *Rec* t)) = $h : t$

instance *Generic*₁ [] **where**

*Rep*₁ [] = U_1 :+: (*Par*₁ :×: (*Rec*₁ []))

*from*₁ [] = $L_1 U_1$

*from*₁ ($h : t$) = R_1 (*Par*₁ h :×: *Rec*₁ t)

*to*₁ ($L_1 U_1$) = *Nil*

*to*₁ (R_1 (*Par*₁ h :×: *Rec*₁ t)) = *Cons* h t

Two instances in generic-deriving



instance *Generic* [α] **where**

Rep [α] = U_1 *:+:* ((*Par*₀ α) *:×*: (*Rec*₀ [α]))

from [] = L_1 U_1

from (h : t) = R_1 (*Par* h *:×*: *Rec* t)

to (L_1 U_1) = []

to (R_1 (*Par* h *:×*: *Rec* t)) = h : t

instance *Generic*₁ [] **where**

*Rep*₁ [] = U_1 *:+:* (*Par*₁ *:×*: (*Rec*₁ []))

*from*₁ [] = L_1 U_1

*from*₁ (h : t) = R_1 (*Par*₁ h *:×*: *Rec*₁ t)

*to*₁ (L_1 U_1) = *Nil*

*to*₁ (R_1 (*Par*₁ h *:×*: *Rec*₁ t)) = *Cons* h t

The *Generic*₁ [] instance has more information than the *Generic* [α] 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 : $\wedge$ : Logic | Logic : $\vee$ : Logic
           | Not Logic | T | F deriving Generic

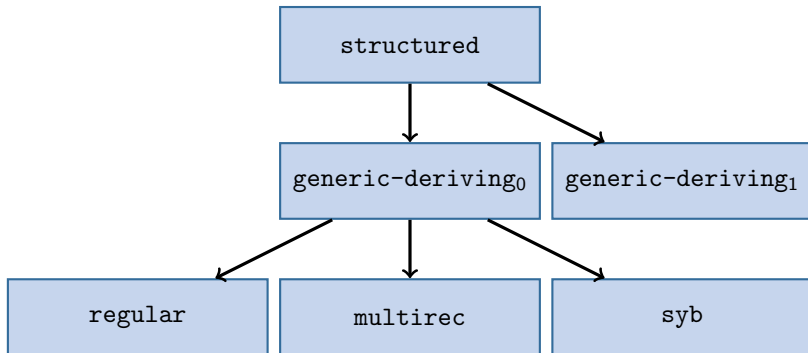
test :: (Logic, Bool, Int)
test = (D.gmempty, R.fold alg term, S.gsize term)
  where term = T : $\vee$ : F
        alg  = ( $\wedge$ ) & ( $\vee$ ) & not & True & False
```

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



kind *Data* = *Data* *MetaData* (*Tree* *Con*)
kind *Con* = *Con* *MetaCon* (*Tree* *Field*)
kind *Field* = *Field* *MetaField* *Arg*
kind *Tree* κ = *Empty* | *Leaf* κ | *Bin* (*Tree* κ) (*Tree* κ)

kind *Arg* = *K* *KType* \star
| *Rec* *RecType* ($\star \rightarrow \star$)
| *Par*
| ($\star \rightarrow \star$) *o*: *Arg*

kind *KType* = *P* | *R* *RecType* | *U*
kind *RecType* = *S* | *O*

```
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
```

data family $\llbracket _ \rrbracket :: \kappa \rightarrow \star \rightarrow \star$

data instance $\llbracket v :: Data \rrbracket \rho$ where

$D \quad :: \llbracket \alpha \rrbracket \rho \rightarrow \llbracket Data \iota \alpha \rrbracket \rho$

data instance $\llbracket v :: Tree Con \rrbracket \rho$ where

$C \quad :: \llbracket \alpha \rrbracket \rho \quad \rightarrow \llbracket Leaf (Con \iota \alpha) \rrbracket \rho$

$L \quad :: \llbracket \alpha \rrbracket \rho \quad \rightarrow \llbracket Bin \alpha \beta \rrbracket \rho$

$R \quad :: \llbracket \beta \rrbracket \rho \quad \rightarrow \llbracket Bin \alpha \beta \rrbracket \rho$

data instance $\llbracket v :: Tree Field \rrbracket \rho$ where

$U \quad :: \llbracket Empty \rrbracket \rho$

$S \quad :: \llbracket \alpha \rrbracket \rho \quad \rightarrow \llbracket Leaf (Field \iota \alpha) \rrbracket \rho$

$(\times) \quad :: \llbracket \alpha \rrbracket \rho \rightarrow \llbracket \beta \rrbracket \rho \rightarrow \llbracket Bin \alpha \beta \rrbracket \rho$

data instance $\llbracket v :: Arg \rrbracket \rho$ where

$K \quad :: \{ unK \quad :: \alpha \} \quad \rightarrow \llbracket K \iota \alpha \rrbracket \rho$

$Rec \quad :: \{ unRec \quad :: \phi \rho \} \quad \rightarrow \llbracket Rec \iota \phi \rrbracket \rho$

$Par \quad :: \{ unPar \quad :: \rho \} \quad \rightarrow \llbracket Par \rrbracket \rho$

$Comp \quad :: \{ unComp \quad :: \sigma (\llbracket \phi \rrbracket \rho) \} \rightarrow \llbracket \sigma \circ \iota \phi \rrbracket \rho$

```
class Generic ( $\alpha :: \star$ ) where  
  Rep  $\alpha :: \text{Data}$   
  Parg  $\alpha :: \star$   
  Parg  $\alpha = \text{NoPar}$   
  
  from  $:: \alpha \rightarrow \llbracket \text{Rep } \phi \rrbracket (\text{Par}_g \alpha)$   
  to  $:: \llbracket \text{Rep } \phi \rrbracket (\text{Par}_g \alpha) \rightarrow \alpha$   
  
data NoPar -- empty
```

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


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

instance *Generic* [α] **where**

```
Rep [ $\alpha$ ] = Data (MD "[]" "Prelude" False)
             (Bin (Leaf (Con MCNil Empty))
                  (Leaf (Con MCCons (Bin H T))))
```

```
Parg [ $\alpha$ ] =  $\alpha$ 
```

```
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

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



kind $Un_R = U_R \mid I_R \mid K_R \star \mid Un_R \text{ :+ :}_R Un_R \mid Un_R \text{ :x :}_R Un_R$

Recall regular



kind $Un_R = U_R \mid I_R \mid K_R \star \mid Un_R \text{ :+}_R Un_R \mid Un_R \text{ :X}_R Un_R$

data $[\alpha :: Un_R]_R (\tau :: \star)$ where

$U_R \quad :: [U_R]_R \tau$

$I_R \quad :: \tau \rightarrow [I_R]_R \tau$

$K_R \quad :: \alpha \rightarrow [K_R \alpha]_R \tau$

$L_R \quad :: [\alpha]_R \tau \rightarrow [\alpha \text{ :+}_R \beta]_R \tau$

$R_R \quad :: [\beta]_R \tau \rightarrow [\alpha \text{ :+}_R \beta]_R \tau$

$(\text{:X}_R) :: [\alpha]_R \tau \rightarrow [\beta]_R \tau \rightarrow [\alpha \text{ :X}_R \beta]_R \tau$

Recall regular



kind $Un_R = U_R \mid I_R \mid K_R \star \mid Un_R \text{ :+}_R Un_R \mid Un_R \text{ :x}_R Un_R$

data $[[\alpha :: Un_R]]_R (\tau :: \star)$ **where**

$U_R \quad :: [[U_R]]_R \tau$

$I_R \quad :: \tau \rightarrow [[I_R]]_R \tau$

$K_R \quad :: \alpha \rightarrow [[K_R \alpha]]_R \tau$

$L_R \quad :: [[\alpha]]_R \tau \rightarrow [[\alpha \text{ :+}_R \beta]]_R \tau$

$R_R \quad :: [[\beta]]_R \tau \rightarrow [[\alpha \text{ :+}_R \beta]]_R \tau$

$(\text{:x}_R) :: [[\alpha]]_R \tau \rightarrow [[\beta]]_R \tau \rightarrow [[\alpha \text{ :x}_R \beta]]_R \tau$

class *Regular* $(\alpha :: \star)$ **where**

$PF \alpha :: Un_R$

$from_R :: \alpha \rightarrow [[PF \alpha]]_R \alpha$

From generic-deriving to regular: types



$D_{\rightarrow R} (\alpha :: Un_D) :: Un_R$

$D_{\rightarrow R} U_D = U_R$

$D_{\rightarrow R} (M_D \iota \alpha) = D_{\rightarrow R} \alpha$

$D_{\rightarrow R} (\alpha :+:_D \beta) = D_{\rightarrow R} \alpha :+:_R D_{\rightarrow R} \beta$

$D_{\rightarrow R} (\alpha :\times:_D \beta) = D_{\rightarrow R} \alpha :\times:_R D_{\rightarrow R} \beta$

$D_{\rightarrow R} (K_D (R S) \tau) = I_R$

$D_{\rightarrow R} (K_D (R O) \alpha) = K_R \alpha$

$D_{\rightarrow R} (K_D P \alpha) = K_R \alpha$

$D_{\rightarrow R} (K_D U \alpha) = K_R \alpha$

From generic-deriving to regular: values



class $Convert_{D \rightarrow R} (\alpha :: Und) \tau$ **where**

$d_{\rightarrow r} :: [\alpha]_D \rho \rightarrow [D \rightarrow R \alpha]_R \tau$

instance $Convert_{D \rightarrow R} (K_D (R S) \tau) \tau$ **where**

$d_{\rightarrow r} (K_{ID} x) = I_R x$

instance $Convert_{D \rightarrow R} (K_D \gamma \quad \tau) \rho$ **where**

$d_{\rightarrow r} (K_{ID} x) = K_R x$

instance ($Convert_{D \rightarrow R} \alpha \tau, Convert_{D \rightarrow R} \beta \tau$)
 $\Rightarrow Convert_{D \rightarrow R} (\alpha :+:_D \beta) \tau$ **where**

$d_{\rightarrow r} (L_{ID} x) = L_R (d_{\rightarrow r} x)$

$d_{\rightarrow r} (R_{ID} x) = R_R (d_{\rightarrow r} x)$

instance ($Convert_{D \rightarrow R} \alpha \tau, Convert_{D \rightarrow R} \beta \tau$)
 $\Rightarrow Convert_{D \rightarrow R} (\alpha :X:_D \beta) \tau$ **where**

$d_{\rightarrow r} (x :X:_D y) = d_{\rightarrow r} x :X:_R d_{\rightarrow r} y$

...

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

```
instance (GenericD α, ConvertD→R (RepD α) α)  
    ⇒ Regular α where  
    PF α = D→R (RepD α)  
    fromR x = d→r (fromD x)
```

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

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>.

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.