Generic Programming for Mutually Recursive Families

Victor Cacciari Miraldo, Alejandro Serrano Mena

February 28, 2018

Motivation

Why another generic programming library?

Motivation

Why another generic programming library?

- No combinator-based GP library for mutually recursive families

Motivation

Why another generic programming library?

- No combinator-based GP library for mutually recursive families
- GHC novel features allows combination of sucessful ideas from previous libraries

Motivation

Why another generic programming library?

- No combinator-based GP library for mutually recursive families
- GHC novel features allows combination of sucessful ideas from previous libraries

Goal

Design and implement a “user-friendly” GP library for handling mutually recursive families

Generic Programming Primer

- Translate class of datatypes to uniform representation

$$T \xrightarrow{from} Rep\ T$$

Generic Programming Primer

- Translate class of datatypes to uniform representation
- Perform generic operation

$$T \xrightarrow{\mathit{from}} \mathit{Rep}\ T \xrightarrow{f} \mathit{Rep}\ U$$

Generic Programming Primer

- Translate class of datatypes to uniform representation
- Perform generic operation
- Translate back to original representation

$$T \xrightarrow{\mathit{from}} \mathit{Rep}\ T \xrightarrow{f} \mathit{Rep}\ U \xrightarrow{\mathit{to}} U$$

Generic Programming Primer

- Translate class of datatypes to uniform representation
- Perform generic operation
- Translate back to original representation

$$T \xrightarrow{\mathit{from}} \mathit{Rep}\ T \xrightarrow{f} \mathit{Rep}\ U \xrightarrow{\mathit{to}} U$$

$$\begin{array}{l} \textbf{class}\ \mathit{Generic}\ t\ \textbf{where} \\ \quad \mathit{from} :: t \to \mathit{Rep}\ t \\ \quad \mathit{to} \quad :: \mathit{Rep}\ t \to t \end{array}$$

The Design Space

- Class of representable datatypes
  - Regular, Nested, Mutually Recursive, ...

The Design Space

- Class of representable datatypes
  - Regular, Nested, Mutually Recursive, ...
- Representation of Recursion
  - Implicit versus Explicit

The Design Space

- Class of representable datatypes
  - Regular, Nested, Mutually Recursive, ...
- Representation of Recursion
  - Implicit versus Explicit
- Codes versus Pattern Functors

The Design Space

- Class of representable datatypes
  - Regular, Nested, Mutually Recursive, ...
- Representation of Recursion
  - Implicit versus Explicit
- Codes versus Pattern Functors

These choices determine the flavour of generic functions:

- Expressivity
- Ease of use

The Landscape

	Pattern Functors	Codes
No Explicit Recursion	`GHC.Generics`	`generics-sop`
Simple Recursion	`regular`	**`generics-mrsop`**
Mutual Recursion	`multirec`	

Pattern Functors (`GHC.Generics`)

Defines the representation of a datatype directly:

$$
\begin{aligned}
\textbf{data}\ \mathit{Bin}\ a &= \mathit{Leaf}\ a \\
&\mid\ \mathit{Fork}\ (\mathit{Bin}\ a)\ (\mathit{Bin}\ a)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Rep}\ (\mathit{Bin}\ a) &= \mathit{K1}\ a \\
&\mathrel{:+:} (\mathit{K1}\ (\mathit{Bin}\ a) \mathrel{:*:} \mathit{K1}\ (\mathit{Bin}\ a))
\end{aligned}
$$

Pattern Functors (GHC.Generics)

Defines the representation of a datatype directly:

$$\begin{aligned} \textbf{data}\ \mathit{Bin}\ a &= \mathit{Leaf}\ a \\ &\mid \mathit{Fork}\ (\mathit{Bin}\ a)\ (\mathit{Bin}\ a) \\ \mathit{Rep}\ (\mathit{Bin}\ a) &= \mathit{K1}\ a \\ &:\!+\!:\ (\mathit{K1}\ (\mathit{Bin}\ a)\ :\!*\!:\ \mathit{K1}\ (\mathit{Bin}\ a)) \end{aligned}$$

$$\begin{aligned} &\textbf{data}\ (f :\!+\!: g)\ x = \mathit{L1}\ (f\ x) \mid \mathit{R1}\ (g\ x) \\ &\textbf{data}\ (f :\!*\!: g)\ \ x = f\ x :\!*\!: g\ x \\ &\textbf{data}\ \mathit{K1}\ a\qquad x = \mathit{K1}\ x \end{aligned}$$

Pattern Functors (`GHC.Generics`)

Defines the representation of a datatype directly:

$$\begin{aligned}
\textbf{data}\ \mathit{Bin}\ a &= \mathit{Leaf}\ a \\
&\mid\ \mathit{Fork}\ (\mathit{Bin}\ a)\ (\mathit{Bin}\ a)
\end{aligned}$$

$$\begin{aligned}
\mathit{Rep}\ (\mathit{Bin}\ a) &= \mathit{K1}\ a \\
&{:}{+}{:}\ (\mathit{K1}\ (\mathit{Bin}\ a)\ {:}{*}{:}\ \mathit{K1}\ (\mathit{Bin}\ a))
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ (f\ {:}{+}{:}\ g)\ x = \mathit{L1}\ (f\ x) \mid \mathit{R1}\ (g\ x) \\
&\textbf{data}\ (f\ {:}{*}{:}\ g)\ \ x = f\ x\ {:}{*}{:}\ g\ x \\
&\textbf{data}\ \mathit{K1}\ a\qquad x = \mathit{K1}\ x
\end{aligned}$$

Note the absence of a pattern functor for handling recursion.

Pattern Functors (`regular`)

The `regular` and `multirec` have a pattern functor for representing recursion.

$$\mathbf{data}\ I\ x = I\ x$$

Now, $Rep\ (Bin\ a) = K1\ \ a\ {:+:}\ (I\ {:*:}\ I)$,

Pattern Functors (`regular`)

The `regular` and `multirec` have a pattern functor for representing recursion.

$$\mathbf{data}\ I\ x = I\ x$$

Now, $Rep\ (Bin\ a) = K1\ a\ {:+:}\ (I\ {:*:}\ I)$,
which allows for explicit least fixpoints:

$$Bin\ a\ \approx\ Rep\ (Bin\ a)\ (Bin\ a)$$

Pattern Functors (`regular`)

The `regular` and `multirec` have a pattern functor for representing recursion.

$$\textbf{data}\ I\ x = I\ x$$

Now, $Rep\ (Bin\ a) = K1\ a :+: (I :*: I)$,
which allows for explicit least fixpoints:

$$Bin\ a\ \approx\ Rep\ (Bin\ a)\ (Bin\ a)$$

Enabling generic recursion shemes:

$$cata :: (Rep\ f\ a \to a) \to f \to a$$

Pattern Functors

Regardless of recursion, class dispatch is used for generic functions:

$$
\begin{aligned}
&\textbf{class}\ \mathit{GSize}\ (\mathit{rep} :: * \to *)\ \textbf{where} \\
&\quad \mathit{gsize} :: \mathit{rep}\ x \to \mathit{Int} \\
&\textbf{instance}\ (\mathit{GSize}\ f, \mathit{GSize}\ g) \Rightarrow \mathit{GSize}\ (f :+: g)\ \textbf{where} \\
&\quad \mathit{gsize}\ (\mathit{L1}\ \ f) = \mathit{gsize}\ f \\
&\quad \mathit{gsize}\ (\mathit{R1}\ \ g) = \mathit{gsize}\ g \\
&\ldots \\
&\mathit{size} :: \mathit{Bin}\ a \to \mathit{Int} \\
&\mathit{size} = \mathit{gsize} \circ \mathit{from}
\end{aligned}
$$

Pattern Functors Drawbacks

- No guarantee about the form of Rep:

Pattern Functors Drawbacks

- No guarantee about the form of Rep: product-of-sums is valid

Pattern Functors Drawbacks

- No guarantee about the form of Rep: product-of-sums is valid
- No guarantee about combinators used in Rep:

Pattern Functors Drawbacks

- No guarantee about the form of Rep: product-of-sums is valid
- No guarantee about combinators used in Rep: $K1\ Int :+: Maybe$ breaks class-dispatch.

Pattern Functors Drawbacks

- No guarantee about the form of Rep: product-of-sums is valid
- No guarantee about combinators used in Rep: $K1\ Int :+: Maybe$ breaks class-dispatch.
- Class-dispatch fragile and hard to extend.

Codes (`generics-sop`)

- Addresses the issues with pattern functors.

Codes (`generics-sop`)

- Addresses the issues with pattern functors.
- The language that representations are defined over.

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Bin}\ a & = \mathit{Leaf}\ a \\
 & \mid\ \mathit{Fork}\ (\mathit{Bin}\ a)\ (\mathit{Bin}\ a)
\end{array}
$$

$$
\begin{array}{lll}
\textbf{type family} & \mathit{Code}\ (a :: *) & :: '['[*]] \\
\textbf{type instance} & \mathit{Code}\ (\mathit{Bin}\ a) & = '['[a], '[\mathit{Bin}\ a, \mathit{Bin}\ a]]
\end{array}
$$

Interpreting Codes (`generics-sop`)

Start with n-ary sums and products:

$$
\begin{aligned}
&\textbf{data}\ \mathit{NS} :: (k \to *) \to [k] \to *\ \textbf{where}\\
&\quad \mathit{Here} \ :: f\ x \qquad\qquad \to \mathit{NS}\ f\ (x\ {}'{:}\ xs)\\
&\quad \mathit{There} :: \mathit{NS}\ f\ xs \to \mathit{NS}\ f\ (x\ {}'{:}\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data}\ \mathit{NP} :: (k \to *) \to [k] \to *\ \textbf{where}\\
&\quad \mathit{Nil} \quad :: \qquad\qquad\qquad\qquad\quad \mathit{NP}\ f\ {}'[]\\
&\quad \mathit{Cons} :: f\ x \to \mathit{NP}\ f\ xs \to \mathit{NP}\ f\ (x\ {}'{:}\ xs)
\end{aligned}
$$

$$\textbf{data}\ I\ x = I\ x$$

Interpreting Codes (`generics-sop`)

Start with n-ary sums and products:

$$
\begin{aligned}
&\textbf{data}\ \mathit{NS} :: (k \to *) \to [k] \to *\ \textbf{where}\\
&\quad \mathit{Here}\ \ :: f\ x \qquad\quad\ \to \mathit{NS}\ f\ (x\ {}^{\prime}{:}\ xs)\\
&\quad \mathit{There} :: \mathit{NS}\ f\ xs \to \mathit{NS}\ f\ (x\ {}^{\prime}{:}\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data}\ \mathit{NP} :: (k \to *) \to [k] \to *\ \textbf{where}\\
&\quad \mathit{Nil}\ \ \ :: \qquad\qquad\qquad\qquad\ \mathit{NP}\ f\ {}^{\prime}[\,]\\
&\quad \mathit{Cons} :: f\ x \to \mathit{NP}\ f\ xs \to \mathit{NP}\ f\ (x\ {}^{\prime}{:}\ xs)
\end{aligned}
$$

$$\textbf{data}\ I\ x = I\ x$$

Define the representation:

$$\textbf{type}\ \mathit{Rep} = \mathit{NS}\ (\mathit{NP}\ I) :: {}^{\prime}[{}^{\prime}[*]] \to *$$

Interpreting Codes (generics-sop)

$$\textbf{type}\ Rep = NS\ (NP\ I) :: '['[*]] \rightarrow *$$

$$\begin{aligned}\textbf{data}\ Bin\ a &= Leaf\ a \\ &\mid\ Fork\ (Bin\ a)\ (Bin\ a)\end{aligned}$$

Interpreting Codes (`generics-sop`)

$$\textbf{type}\ \mathit{Rep} = \mathit{NS}\ (\mathit{NP}\ I) :: {}'[{}'[*]] \rightarrow *$$

$$\begin{aligned}\textbf{data}\ \mathit{Bin}\ a &= \mathit{Leaf}\ a \\ &\mid \mathit{Fork}\ (\mathit{Bin}\ a)\ (\mathit{Bin}\ a)\end{aligned}$$

Recall the Tree example:

$$\textbf{type instance}\ \mathit{Code}\ (\mathit{Bin}\ a) = {}'[{}'[a], {}'[\mathit{Bin}\ a, \mathit{Bin}\ a]]$$

$$\begin{aligned}&\mathit{leaf} &&:: a \rightarrow \mathit{Rep}\ (\mathit{Code}\ (\mathit{Tree}\ a)) \\ &\mathit{leaf}\ e &&= \mathit{Here}\ (\mathit{Cons}\ e\ \mathit{Nil})\end{aligned}$$

$$\begin{aligned}&\mathit{bin} &&:: \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Rep}\ (\mathit{Code}\ (\mathit{Tree}\ a)) \\ &\mathit{bin}\ l\ r &&= \mathit{There}\ (\mathit{Here}\ (\mathit{Cons}\ l\ (\mathit{Cons}\ r\ \mathit{Nil})))\end{aligned}$$

Generic Functionality (`generics-sop`)

Codes allow for combinators instead of class-dispatch:

$$elimNP :: (\forall\ k\ .\ f\ k \to a) \to NP\ f\ xs \to [a]$$
$$elimNS :: (\forall\ k\ .\ f\ k \to a) \to NS\ f\ xs \to a$$

Generic Functionality (`generics-sop`)

Codes allow for combinators instead of class-dispatch:

$$elimNP :: (\forall\ k\ .\ f\ k \to a) \to NP\ f\ xs \to [a]$$
$$elimNS :: (\forall\ k\ .\ f\ k \to a) \to NS\ f\ xs \to a$$

class $Size\ a$ **where**
$\quad size :: a \to Int$

$$gsize :: (Generic\ a, All2\ Size\ (Code\ a)) \Rightarrow a \to Int$$
$$gsize = succ \circ sum \circ elimNS\ (elimNP\ (size \circ unI)) \circ from$$
$\quad$ **where** $unI\ (I\ x) = x$

Generic Functionality (`generics-sop`)

Codes allow for combinators instead of class-dispatch:

$$elimNP :: (\forall\ k\ .\ f\ k \to a) \to NP\ f\ xs \to [a]$$
$$elimNS :: (\forall\ k\ .\ f\ k \to a) \to NS\ f\ xs \to a$$

class $Size\ a$ **where**
  $size :: a \to Int$

$gsize :: (Generic\ a, All2\ Size\ (Code\ a)) \Rightarrow a \to Int$
$gsize = succ \circ sum \circ elimNS\ (elimNP\ (size \circ unI)) \circ from$
  where $unI\ (I\ x) = x$

Still: no explicit recursion: typeclass and complicated constraints.

Mutual Recursion (`generics-mrsop`)

Start with Rep as before:

$$\textbf{data}\ I\ x = I\ x$$

$$\begin{aligned}&\textbf{type}\ Rep\ (f :: '['[*]])\\ &\quad = NS\ (NP\ I)\ f\end{aligned}$$

Mutual Recursion (`generics-mrsop`)

Add codes to handle a single recursive position:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Atom} = I \mid \mathit{KInt} \mid \ldots \\
&\textbf{data}\ \mathit{NA} :: * \to \mathit{Atom} \to *\ \textbf{where} \\
&\quad \mathit{NA\_I}\ \ :: x \quad\ \to \mathit{NA}\ x\ I \\
&\quad \mathit{NA\_K} :: \mathit{Int} \to \mathit{NA}\ x\ \mathit{KInt}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ \mathit{Rep}\ (x :: *)\ (f :: '['[\mathit{Atom}]]) \\
&\quad = \mathit{NS}\ (\mathit{NP}\ (\mathit{NA}\ x))\ f
\end{aligned}
$$

Mutual Recursion (`generics-mrsop`)

Augment codes to have n recursive positions:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Atom} = I\ \mathit{Nat} \mid K \mathit{Int} \mid \ldots \\
\textbf{data}\ \mathit{NA} :: (\mathit{Nat} \to *) \to \mathit{Atom} \to *\ \textbf{where} \\
\quad \mathit{NA\_I}\ \ :: x\ n \to \mathit{NA}\ x\ (I\ n) \\
\quad \mathit{NA\_K} :: \mathit{Int} \to \mathit{NA}\ x\ K\mathit{Int}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{Rep}\ (x :: \mathit{Nat} \to *)\ (f :: '['[\mathit{Atom}]]) \\
\quad = \mathit{NS}\ (\mathit{NP}\ (\mathit{NA}\ x))\ f
\end{array}
$$

Example (generics-mrsop)

$$\textbf{data}\ RTree\ a = RTree\ a\ (Forest\ a)$$
$$\textbf{data}\ Forest\ a = Nil \mid Cons\ (RTree\ a)\ (Forest\ a)$$

$$\textbf{type}\ Fam = '[RTree\ Int, Forest\ Int]$$

Example (generics-mrsop)

$$\textbf{data}\ RTree\ a = RTree\ a\ (Forest\ a)$$
$$\textbf{data}\ Forest\ a = Nil \mid Cons\ (RTree\ a)\ (Forest\ a)$$

$$\textbf{type}\ Fam = '[RTree\ Int, Forest\ Int]$$

$$\textbf{type}\ CodeRTree = '['[KInt, I\ 1]]$$
$$\textbf{type}\ CodeForest = '['[], '[I\ 0, I\ 1]]$$
$$\textbf{type}\ Codes = '[CodeRTree, CodeForest]$$

Example (generics-mrsop)

$$\textbf{data}\ RTree\ a\ =\ RTree\ a\ (Forest\ a)$$
$$\textbf{data}\ Forest\ a = Nil \mid Cons\ (RTree\ a)\ (Forest\ a)$$

$$\textbf{type}\ Fam = '[RTree\ Int, Forest\ Int]$$

$$\textbf{type}\ CodeRTree\ = '['[KInt, I\ 1]]$$
$$\textbf{type}\ CodeForest = '['[], '[I\ 0, I\ 1]]$$
$$\textbf{type}\ Codes = '[CodeRTree, CodeForest]$$

$$\textbf{instance}\ Family\ Fam\ Codes\ \textbf{where}$$
$$\quad \ldots$$

Closing the Recursive Knot (`generics-mrsop`)

- Define a family: $fam :: '[*]$

Closing the Recursive Knot (`generics-mrsop`)

- Define a family: $fam :: '[*]$
- Define its codes: $codes :: '['['[Atom]]]$

Closing the Recursive Knot (`generics-mrsop`)

- Define a family: $fam :: {}'[*]$
- Define its codes: $codes :: {}'['['[Atom]]]$
- Define lookup:

$$
\begin{array}{l}
\textbf{type family}\ Lkup\ (ls :: {}'[k])\ (n :: Nat) :: k\ \textbf{where} \\
\quad Lkup\ {}'[]\qquad\quad \_ \qquad = TypeError\ \text{“}Out\ of\ bounds\text{”} \\
\quad Lkup\ (x\ {}': xs)\ Z \qquad = x \\
\quad Lkup\ (x\ {}': xs)\ (S\ n) = Lkup\ xs\ n
\end{array}
$$

Closing the Recursive Knot (`generics-mrsop`)

- Define a family: $fam :: '[*]$
- Define its codes: $codes :: '['['[Atom]]]$
- Define lookup:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{type family } Lkup\ (ls :: '[k])\ (n :: Nat) :: k \textbf{ where}} \\
\quad Lkup\ '[] & \_ & = & TypeError\ \text{“}Out\ of\ bounds\text{”} \\
\quad Lkup\ (x\ ': xs) & Z & = & x \\
\quad Lkup\ (x\ ': xs) & (S\ n) & = & Lkup\ xs\ n
\end{array}
$$

Then, finally, the i-th type is represented by:

$$Rep\ (Lkup\ fam)\ (Lkup\ i\ codes)$$

Closing the Recursive Knot (`generics-mrsop`)

- Define a family: $fam :: {}'[*]$
- Define its codes: $codes :: {}'[{}'[{}'[Atom]]]$
- Define lookup:

$$
\begin{array}{l}
\textbf{type family } Lkup\ (ls :: {}'[k])\ (n :: Nat) :: k\ \textbf{where} \\
\quad Lkup\ {}'[]\ \qquad\ \_ \qquad = TypeError\ \text{"Out of bounds"} \\
\quad Lkup\ (x\ {}': xs)\ Z \qquad = x \\
\quad Lkup\ (x\ {}': xs)\ (S\ n) = Lkup\ xs\ n
\end{array}
$$

Then, finally, the i-th type is represented by:

$$Rep\ (Lkup\ fam)\ (Lkup\ i\ codes)$$

$Lkup$ can't be partially applied though.

Wrapping it up (`generics-mrsop`)

Create an El type to be able to partially apply it and wrap it all in a typeclass:

Wrapping it up (`generics-mrsop`)

Create an *El* type to be able to partially apply it and wrap it all in a typeclass:

$$
\begin{array}{l}
\textbf{data}\ El :: '[*] \to Nat \to * \ \textbf{where} \\
\quad El :: Lkup\ fam\ ix \to El\ fam\ ix
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ Family\ (fam :: '[*])\ (codes :: '['['[Atom]]])\ \textbf{where}} \\
\quad from & :: & SNat\ ix \\
 & \to & El\ fam\ ix \\
 & \to & Rep\ (El\ fam)\ (Lkup\ codes\ ix) \\
\quad to & :: & SNat\ ix \\
 & \to & Rep\ (El\ fam)\ (Lkup\ codes\ ix) \\
 & \to & El\ fam\ ix
\end{array}
$$

Well formed Representations Only

- The **data** $Atom = I\ Nat \mid \ldots$ type might seem too permissive

Well formed Representations Only

- The **data** $Atom = I\ Nat \mid \ldots$ type might seem too permissive
- One solution: **data** $Atom\ n = I\ (Fin\ n) \mid \ldots$
  Too complicated in Haskell.

Well formed Representations Only

- The **data** $Atom = I\ Nat \mid \ldots$ type might seem too permissive
- One solution: **data** $Atom\ n = I\ (Fin\ n) \mid \ldots$
  Too complicated in Haskell.
- In fact, there is no problem: one could define:
  type $CodeRTree = '['[KInt, I\ 42]]$, the instance would be impossible to write.

Well formed Representations Only

- The **data** $Atom = I\ Nat \mid \ldots$ type might seem too permissive
- One solution: **data** $Atom\ n = I\ (Fin\ n) \mid \ldots$
  Too complicated in Haskell.
- In fact, there is no problem: one could define:
  type $CodeRTree = '[' [KInt, I\ 42]]$, the instance would be impossible to write.
- Malformed codes $\Rightarrow$ uninhabitable representations.
- Errors are caught at compile time.

Deep versus Shallow

Deep encoding comes for free!

$$\textbf{newtype}\ \mathit{Fix}\ \mathit{codes}\ \mathit{ix}$$
$$\quad = \mathit{Fix}\ (\mathit{Rep}\ (\mathit{Fix}\ \mathit{codes})\ (\mathit{Lkup}\ \mathit{codes}\ \mathit{ix}))$$

Deep versus Shallow

Deep encoding comes for free!

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{Fix}\ \mathit{codes}\ \mathit{ix} \\
&\quad = \mathit{Fix}\ (\mathit{Rep}\ (\mathit{Fix}\ \mathit{codes})\ (\mathit{Lkup}\ \mathit{codes}\ \mathit{ix}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{deep} :: (\mathit{Family}\ \mathit{fam}\ \mathit{codes}) \\
&\qquad \Rightarrow \mathit{El}\ \mathit{fam}\ \mathit{ix} \to \mathit{Fix}\ \mathit{codes}\ \mathit{ix} \\
&\mathit{deep} = \mathit{Fix} \circ \mathit{mapRep}\ \mathit{deep} \circ \mathit{from}
\end{aligned}
$$

Deep versus Shallow

Deep encoding comes for free!

$$\textbf{newtype}\ \mathit{Fix}\ \mathit{codes}\ \mathit{ix} \\ \quad = \mathit{Fix}\ (\mathit{Rep}\ (\mathit{Fix}\ \mathit{codes})\ (\mathit{Lkup}\ \mathit{codes}\ \mathit{ix}))$$

$$\begin{aligned} &\mathit{deep} :: (\mathit{Family}\ \mathit{fam}\ \mathit{codes}) \\ &\qquad \Rightarrow \mathit{El}\ \mathit{fam}\ \mathit{ix} \rightarrow \mathit{Fix}\ \mathit{codes}\ \mathit{ix} \\ &\mathit{deep} = \mathit{Fix} \circ \mathit{mapRep}\ \mathit{deep} \circ \mathit{from} \end{aligned}$$

- provide recursion schemes (cata, ana, $\mathit{synthesize}$, etc)
- No need to carry constraints around

Deep versus Shallow

Deep encoding comes for free!

$$\textbf{newtype}\ \mathit{Fix}\ \mathit{codes}\ \mathit{ix}$$
$$\quad = \mathit{Fix}\ (\mathit{Rep}\ (\mathit{Fix}\ \mathit{codes})\ (\mathit{Lkup}\ \mathit{codes}\ \mathit{ix}))$$

$$\mathit{deep} :: (\mathit{Family}\ \mathit{fam}\ \mathit{codes})$$
$$\qquad \Rightarrow \mathit{El}\ \mathit{fam}\ \mathit{ix} \rightarrow \mathit{Fix}\ \mathit{codes}\ \mathit{ix}$$
$$\mathit{deep} = \mathit{Fix} \circ \mathit{mapRep}\ \mathit{deep} \circ \mathit{from}$$

- provide recursion schemes (cata, ana, $\mathit{synthesize}$, etc)
- No need to carry constraints around

$$\mathit{gsize} :: (\mathit{Family}\ \mathit{fam}\ \mathit{codes})$$
$$\qquad \Rightarrow \mathit{El}\ \mathit{fam}\ \mathit{ix} \rightarrow \mathit{Int}$$
$$\mathit{gsize} = \mathit{cata}\ (\mathit{succ} \circ \mathit{sum} \circ \mathit{elimNP}\ (\mathit{elimNA}\ \mathit{id})) \circ \mathit{deep}$$

Custom Opaque Types

Recall our definition of $Atom$:

$$
\begin{array}{l}
\textbf{data}\ Atom = I\ Nat \mid KInt \mid \ldots \\
\textbf{data}\ NA :: (Nat \to *) \to Atom \\
\qquad\qquad \to * \ \textbf{where} \\
\quad NA\_I \ \ :: x\ n \to NA\ x\ (I\ n) \\
\quad NA\_K :: Int \to NA\ x\ KInt
\end{array}
$$

Custom Opaque Types

Add another parameter to it:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Atom}\ \mathit{kon} = I\ \mathit{Nat} \mid K\ \mathit{kon}\\
&\textbf{data}\ \mathit{NA} :: (\mathit{kon} \to *) \to (\mathit{Nat} \to *) \to \mathit{Atom}\ \mathit{kon}\\
&\qquad\qquad \to *\ \textbf{where}\\
&\quad \mathit{NA\_I}\ \ :: x\ \ n \to \mathit{NA}\ \mathit{ki}\ x\ (I\ \ n)\\
&\quad \mathit{NA\_K} :: \mathit{ki}\ k \to \mathit{NA}\ \mathit{ki}\ x\ (K\ \ k)
\end{aligned}
$$

Custom Opaque Types

Add another parameter to it:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Atom}\ \mathit{kon} = \mathit{I}\ \mathit{Nat} \mid \mathit{K}\ \mathit{kon} \\
&\textbf{data}\ \mathit{NA} :: (\mathit{kon} \to *) \to (\mathit{Nat} \to *) \to \mathit{Atom}\ \mathit{kon} \\
&\qquad\qquad \to * \ \textbf{where} \\
&\quad \mathit{NA\_I} \ :: x\ \ n \to \mathit{NA}\ \mathit{ki}\ x\ (\mathit{I}\ n) \\
&\quad \mathit{NA\_K} :: \mathit{ki}\ k \to \mathit{NA}\ \mathit{ki}\ x\ (\mathit{K}\ k)
\end{aligned}
$$

Define a kind for opaque types and their interpretation:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Opaque} = \mathit{O\_Int} \mid \mathit{O\_Float} \\
&\textbf{data}\ \mathit{OpaqueSingl} :: \mathit{Opaque} \to * \ \textbf{where} \\
&\quad \mathit{OS\_Int} \quad :: \mathit{Int} \quad \to \mathit{OpaqueSingl}\ \mathit{O\_Int} \\
&\quad \mathit{OS\_Float} :: \mathit{Float} \to \mathit{OpaqueSingl}\ \mathit{O\_Float}
\end{aligned}
$$

Other Features from `generics-mrsop`

- Custom opaque types.

Other Features from `generics-mrsop`

- Custom opaque types.
- Zippers for mutually recursive families.

Other Features from `generics-mrsop`

- Custom opaque types.
- Zippers for mutually recursive families.
- Automatic *Family* generation with Template Haskell.

Other Features from `generics-mrsop`

- Custom opaque types.
- Zippers for mutually recursive families.
- Automatic *Family* generation with Template Haskell.
- Metadata support inspired by `generics-sop`.

Lessons and Discussion

- Found two bugs in GHC: #14987 and #15517 (closed)

Lessons and Discussion

- Found two bugs in GHC: #14987 and #15517 (closed)
- Working with deep representations is simpler:
  - Recursion schemes
  - No need to carry constraints around

Lessons and Discussion

- Found two bugs in GHC: #14987 and #15517 (closed)
- Working with deep representations is simpler:
  - Recursion schemes
  - No need to carry constraints around
- Very powerful tool to work with generic AST's

Lessons and Discussion

- Found two bugs in GHC: #14987 and #15517 (closed)
- Working with deep representations is simpler:
  - Recursion schemes
  - No need to carry constraints around
- Very powerful tool to work with generic AST's
- Curious about handling GADTs?
  Join Haskell Symposium tomorrow at 9h30!

Conclusions

Use it and Hack it!

`https://hackage.haskell.org/package/generics-mrsop`

Generic Programming for Mutually Recursive Families

Victor Cacciari Miraldo, Alejandro Serrano Mena

February 28, 2018

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences