Generic Programming of All Kinds

Alejandro Serrano Mena, Victor Cacciari Miraldo

February 28, 2018

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Context and Motivation

$$
\begin{aligned}
&\textbf{data}\ \mathit{Exp} :: * \to * \textbf{where}\\
&\quad \mathit{Val} :: \mathit{Int} \to \mathit{Exp}\ \mathit{Int}\\
&\quad \mathit{Add} :: \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int}\\
&\quad \mathit{Eq} :: \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Bool}\\
&\quad \ldots\\
&\textbf{deriving instance}\ (\mathit{Serialize}\ a) \Rightarrow \mathit{Serialize}\ (\mathit{Exp}\ a)
\end{aligned}
$$

Context and Motivation

$$
\begin{array}{l}
\textbf{data}\ Exp :: * \to * \textbf{where} \\
\quad Val :: Int \to Exp\ Int \\
\quad Add :: Exp\ Int \to Exp\ Int \to Exp\ Int \\
\quad Eq :: Exp\ Int \to Exp\ Int \to Exp\ Bool \\
\quad \ldots \\
\textbf{deriving instance}\ (Serialize\ a) \Rightarrow Serialize\ (Exp\ a)
\end{array}
$$

We would like this feature!

Context and Motivation

$$\begin{array}{l}
\textbf{data}\ \mathit{Exp} :: * \to * \textbf{where} \\
\quad \mathit{Val} :: \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{Add} :: \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{Eq} :: \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \to \mathit{Exp}\ \mathit{Bool} \\
\quad \ldots \\
\textbf{deriving instance}\ (\mathit{Serialize}\ a) \Rightarrow \mathit{Serialize}\ (\mathit{Exp}\ a)
\end{array}$$

We would like this feature!

Implementing it in a general fashion requires some generic programming over GADTs and arbitrarily kinded types.

Context and Motivation

- GHC's modern extensions allow for more expressive generic programming.

Context and Motivation

- GHC's modern extensions allow for more expressive generic programming.

- Inability to currently handle arbitrarily kinded datatypes.

Context and Motivation

- GHC's modern extensions allow for more expressive generic programming.
- Inability to currently handle arbitrarily kinded datatypes.
- GADTs are becomming more common: **deriving** clauses would be handy.

Representing Datatypes (`generics-sop`)

Haskell datatypes come in sums-of-products shape:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf} \mid \mathit{Bin}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

Representing Datatypes (`generics-sop`)

Haskell datatypes come in sums-of-products shape:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf} \mid \mathit{Bin}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

Our codes will follow that structure:

$$\textbf{type family}\ \mathit{Code}\ (x :: *) :: {}'[{}'[*]]$$
$$\textbf{type instance}\ \mathit{Code}\ (\mathit{Tree}\ a) = {}'[{}'[], {}'[a, \mathit{Tree}\ a, \mathit{Tree}\ a]]$$

Representing Datatypes (`generics-sop`)

Haskell datatypes come in sums-of-products shape:

$$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Leaf} \mid \mathit{Bin}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

Our codes will follow that structure:

$$\begin{aligned}&\mathbf{type\ family}\ \mathit{Code}\ (x :: *) :: {'}[{'}[*]]\\ &\mathbf{type\ instance}\ \mathit{Code}\ (\mathit{Tree}\ a) = {'}[{'}[], {'}[a, \mathit{Tree}\ a, \mathit{Tree}\ a]]\end{aligned}$$

Given a map from ${'}[{'}[*]]$ into $*$, call it Rep, package:

$$\begin{aligned}&\mathbf{class}\ \mathit{Generic}\ a\ \mathbf{where}\\ &\quad \mathit{from} :: a \to \mathit{Rep}\ (\mathit{Code}\ a)\\ &\quad \mathit{to}\ \ \ :: \mathit{Rep}\ (\mathit{Code}\ a) \to a\end{aligned}$$

N-ary Sums and Products

$$NS\ p\ [x_1, \ldots, x_n] \approx Either\ (p\ x_1)\ (Either\ \ldots\ (p\ x_n))$$
$$NP\ p\ [x_1, \ldots, x_n] \approx (p\ x_1, \ldots, p\ x_n)$$

N-ary Sums and Products

$$NS\ p\ [x_1, \ldots, x_n] \approx Either\ (p\ x_1)\ (Either\ \ldots\ (p\ x_n))$$
$$NP\ p\ [x_1, \ldots, x_n] \approx (p\ x_1, \ldots, p\ x_n)$$

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

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

Interpreting Codes (`generics-sop`)

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

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

Interpreting Codes (`generics-sop`)

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

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

Recall the $Tree$ example:

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

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

Writing Generic Functions

Package it in a class

$$\textbf{class}\ \mathit{Size}\ a\ \textbf{where}$$
$$\quad \mathit{size} :: a \rightarrow \mathit{Int}$$

Writing Generic Functions

Package it in a class

$$
\begin{array}{l}
\textbf{class}\ \mathit{Size}\ a\ \textbf{where} \\
\quad \mathit{size} :: a \to \mathit{Int}
\end{array}
$$

Then write the generic infrastructure:

$$
\begin{array}{l}
\mathit{gsize} :: (\mathit{Generic}\ x, \mathit{All2}\ \mathit{Size}\ (\mathit{Code}\ x)) \\
\qquad \Rightarrow x \to \mathit{Int} \\
\mathit{gsize} = \mathit{goS} \circ \mathit{from} \\
\quad \textbf{where} \\
\qquad \mathit{goS}\ (\mathit{Here}\ \ x) = \mathit{goP}\ x \\
\qquad \mathit{goS}\ (\mathit{There}\ x) = \mathit{goS}\ x \\
\qquad \mathit{goP}\ \mathit{Nil} \qquad\quad\ = 0 \\
\qquad \mathit{goP}\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{size}\ x + \mathit{goP}\ \mathit{xs}
\end{array}
$$

Generics of All Kinds

- So far, only handle types of kind $*$ with no parameters.

Generics of All Kinds

- So far, only handle types of kind $*$ with no parameters.
- Consequence of little structure on $Code$s.

Generics of All Kinds

- So far, only handle types of kind $*$ with no parameters.
- Consequence of little structure on $Code$s.
- Solution: Augment the language of codes!

$$\textbf{type}\ DataType\ (\zeta :: Kind) = '['[Atom\ \zeta\ (*)]]$$

Generics of All Kinds

- So far, only handle types of kind $*$ with no parameters.
- Consequence of little structure on $Code$s.
- Solution: Augment the language of codes!

$$\textbf{type}\ DataType\ (\zeta :: Kind) = {}'[{}'[Atom\ \zeta\ (*)]]$$

- $Atom$ is the applicative fragment of the λ-calculus with de Bruijn indices.

Generics of All Kinds

$$
\begin{array}{l}
\textbf{data}\ \mathit{Atom}\ (\zeta :: \mathit{Kind})\ (k :: \mathit{Kind}) :: (*)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Var} & :: \mathit{TyVar}\ \zeta\ k & \rightarrow \mathit{Atom}\ \zeta\ k \\
\mathit{Kon} & :: k & \rightarrow \mathit{Atom}\ \zeta\ k \\
(:@:) & :: \mathit{Atom}\ \zeta\ (l \rightarrow k) \rightarrow \mathit{Atom}\ \zeta\ l & \rightarrow \mathit{Atom}\ \zeta\ k
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{TyVar}\ (\zeta :: \mathit{Kind})\ (k :: \mathit{Kind}) :: (*)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{VZ} & :: & \mathit{TyVar}\ (x \rightarrow xs)\ x \\
\mathit{VS} & :: \mathit{TyVar}\ xs\ k \rightarrow & \mathit{TyVar}\ (x \rightarrow xs)\ k
\end{array}
\end{array}
$$

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Generics of All Kinds

$$
\begin{aligned}
&\textbf{data}\ \mathit{Atom}\ (\zeta :: \mathit{Kind})\ (k :: \mathit{Kind}) :: (*)\ \textbf{where}\\
&\quad \mathit{Var}\ \ :: \mathit{TyVar}\ \zeta\ k \qquad\qquad\qquad\qquad\quad \to \mathit{Atom}\ \zeta\ k\\
&\quad \mathit{Kon}\ :: k \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \mathit{Atom}\ \zeta\ k\\
&\quad (:@:) :: \mathit{Atom}\ \zeta\ (l \to k) \to \mathit{Atom}\ \zeta\ l \to \mathit{Atom}\ \zeta\ k
\end{aligned}
$$

Going back to our *Tree* example:

$$
\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf} \mid \mathit{Bin}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)
$$

$$
\begin{aligned}
&\textbf{type}\ \mathit{V0} = \mathit{Var}\ \mathit{VZ}\\
&\textbf{type}\ \mathit{TreeCode}\\
&\quad = '['[], '[\mathit{V0}, \mathit{Kon}\ \mathit{Tree}\ {:@:}\ \mathit{V0}, \mathit{Kon}\ \mathit{Tree}\ {:@:}\ \mathit{V0}]]\\
&\quad :: '['[\mathit{Atom}\ (* \to *)\ *]]
\end{aligned}
$$

Interpreting `Atoms`

Interpreting atoms needs environment.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{\Gamma}\ (\zeta :: \mathit{Kind})\ \textbf{where} & & \\
\quad \epsilon & :: & \mathit{\Gamma}\ (*) \\
\quad (:\&:) & :: k \to \mathit{\Gamma}\ ks \to & \mathit{\Gamma}\ (k \to ks)
\end{array}
$$

Interpreting `Atoms`

Interpreting atoms needs environment.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{\Gamma}\ (\zeta :: \mathit{Kind})\ \textbf{where} & & \\
\quad \epsilon & :: & \mathit{\Gamma}\ (*) \\
\quad (:\&:) & :: & k \to \mathit{\Gamma}\ ks \to \mathit{\Gamma}\ (k \to ks)
\end{array}
$$

For example,

$$\mathit{Int}\ :\&:\ \mathit{Maybe}\ :\&:\ \mathit{Char}\ :\&:\ \epsilon$$

Is a well-formed enviroment of kind

$$\mathit{\Gamma}\ (* \to (* \to *) \to * \to *)$$

Interpreting Atoms

Interpreting atoms needs environment.

$$
\begin{array}{l}
\textbf{data}\ \Gamma\ (\zeta :: \mathit{Kind})\ \textbf{where} \\
\quad \epsilon \quad\ :: \qquad\qquad\qquad\quad \Gamma\ (*) \\
\quad (:\&:) :: k \to \Gamma\ ks \to \Gamma\ (k \to ks)
\end{array}
$$

$$
\begin{array}{l}
\textbf{type family}\ \mathit{Ty}\ \zeta\ (\mathit{tys} :: \Gamma\ \zeta)\ (t :: \mathit{Atom}\ \zeta\ k) :: k\ \textbf{where} \\
\quad \mathit{Ty}\ (k \to ks)\ (t :\&: ts)\ (\mathit{Var}\ \mathit{VZ}) \qquad = t \\
\quad \mathit{Ty}\ (k \to ks)\ (t :\&: ts)\ (\mathit{Var}\ (\mathit{VS}\ v)) = \mathit{Ty}\ ks\ ts\ (\mathit{Var}\ v) \\
\quad \mathit{Ty}\ \zeta\ ts\ (\mathit{Kon}\ t)\ = t \\
\quad \mathit{Ty}\ \zeta\ ts\ (f :@: x) = (\mathit{Ty}\ \zeta\ ts\ f)\ (\mathit{Ty}\ \zeta\ ts\ x)
\end{array}
$$

Interpreting Codes

We are now ready to map a code, of kind $\mathit{DataType}\ \zeta$, into $*$.

First, package Ty into a GADT:

$$
\begin{array}{l}
\textbf{data}\ \mathit{NA}\ (\zeta :: \mathit{Kind}) :: \Gamma\ \zeta \to \mathit{Atom}\ \zeta\ (*) \to *\ \textbf{where} \\
\quad T :: \forall\ \zeta\ t\ a\ .\ \mathit{Ty}\ \zeta\ a\ t \to \mathit{NA}\ \zeta\ a\ t
\end{array}
$$

Interpreting Codes

We are now ready to map a code, of kind $\mathit{DataType}\ \zeta$, into $*$.

First, package Ty into a GADT:

$$\begin{array}{l}\textbf{data}\ \mathit{NA}\ (\zeta :: \mathit{Kind}) :: \Gamma\ \zeta \to \mathit{Atom}\ \zeta\ (*) \to *\ \textbf{where} \\ \quad T :: \forall\ \zeta\ t\ a\ .\ \mathit{Ty}\ \zeta\ a\ t \to \mathit{NA}\ \zeta\ a\ t\end{array}$$

Then, assemble NS, NP and NA:

$$\begin{array}{l}\textbf{type}\ \mathit{Rep}\ (\zeta :: \mathit{Kind})\ (c :: \mathit{DataType}\ \zeta)\ (a :: \Gamma\ \zeta) \\ \quad = \mathit{NS}\ (\mathit{NP}\ (\mathit{NA}\ \zeta\ a))\ c\end{array}$$

Approaching a Unified API

Usually, GP libraries provide a class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Generic}\ f\ \textbf{where} \\
\quad \textbf{type}\ \mathit{Code}\ f :: \mathit{CodeKind} \\
\quad \mathit{from} :: f \rightarrow \mathit{Rep}\ (\mathit{Code}\ f) \\
\quad \mathit{to}\ \ \ \ :: \mathit{Rep}\ (\mathit{Code}\ f)
\end{array}
$$

Approaching a Unified API

Usually, GP libraries provide a class:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Generic}\ f\ \textbf{where} \\
&\quad \textbf{type}\ \mathit{Code}\ f :: \mathit{CodeKind} \\
&\quad \mathit{from} :: f \rightarrow \mathit{Rep}\ (\mathit{Code}\ f) \\
&\quad \mathit{to} \quad\ :: \mathit{Rep}\ (\mathit{Code}\ f)
\end{aligned}
$$

In our case, though, the number of arguments to f depend on it's kind!

$$
\begin{aligned}
&\mathit{from} :: f \qquad\ \rightarrow \mathit{Rep}\ (*)\ (\mathit{Code}\ f)\ \epsilon \\
&\mathit{from} :: f\ x \quad\ \rightarrow \mathit{Rep}\ (*)\ (\mathit{Code}\ f)\ (x :\&: \epsilon) \\
&\mathit{from} :: f\ x\ y \rightarrow \mathit{Rep}\ (*)\ (\mathit{Code}\ f)\ (x :\&: y :\&: \epsilon)
\end{aligned}
$$

Approaching a Unified API

Write a GADT:

$$
\begin{array}{l}
\textbf{data}\ \mathit{ApplyT}\ \zeta\ (f :: k)\ (\alpha :: \Gamma\ \zeta) :: *\ \textbf{where} \\
\quad \mathit{A0} :: f \qquad\qquad\qquad\qquad\quad \to \mathit{ApplyT}\ (*) \qquad\quad f\ \epsilon \\
\quad \mathit{AS} :: \mathit{ApplyT}\ ks\ (f\ t)\ ts \to \mathit{ApplyT}\ (k \to ks)\ f\ (t\ {:}\&{:}\ ts)
\end{array}
$$

Approaching a Unified API

Write a GADT:

$$\textbf{data}\ \mathit{ApplyT}\ \zeta\ (f :: k)\ (\alpha :: \Gamma\ \zeta) :: * \ \textbf{where}$$
$$\begin{aligned} &\mathit{A0} :: f && \to \mathit{ApplyT}\ (*) && f\ \epsilon \\ &\mathit{AS} :: \mathit{ApplyT}\ ks\ (f\ t)\ ts && \to \mathit{ApplyT}\ (k \to ks) && f\ (t :\&: ts) \end{aligned}$$

Which allows us to unify the interface:

$$\mathit{from} :: \mathit{ApplyT}\ \zeta\ f\ a \to \mathit{Rep}\ \zeta\ (\mathit{Code}\ f)\ a$$

Wait?! type-in-type?

- We require `-XTypeInType` to type check our code because we need to promote GADTs and work with kinds as types.

Wait?! type-in-type?

- We require `-XTypeInType` to type check our code because we need to promote GADTs and work with kinds as types.
- We do not require the `*:*` axiom

Wait?! type-in-type?

- We require `-XTypeInType` to type check our code because we need to promote GADTs and work with kinds as types.
- We do not require the $*{:}*$ axiom
- We provide an Agda model of our approach to prove so. Basic types live in Set_0, our codes inhabit Set_1 and the interpretations inhabit Set_2.

Representing Constraints

With small modifications, we can handle constraints.

Representing Constraints

With small modifications, we can handle constraints.

Add one layer on top of $Atom$:

$$\begin{aligned}
&\textbf{data}\ \mathit{Field}\ (\zeta :: \mathit{Kind})\ \textbf{where}\\
&\quad \mathit{Explicit} :: \mathit{Atom}\ \zeta\ (*) \qquad\qquad\quad \to \mathit{Field}\ \zeta\\
&\quad \mathit{Implicit} :: \mathit{Atom}\ \zeta\ \mathit{Constraint} \to \mathit{Field}\ \zeta\\
&\textbf{type}\ \mathit{DataType}\ \zeta = {}'[{}'[\mathit{Field}\ \zeta]]
\end{aligned}$$

Representing Constraints

With small modifications, we can handle constraints.

Add one layer on top of $Atom$:

$$\begin{aligned}
&\mathbf{data}\ Field\ (\zeta :: Kind)\ \mathbf{where} \\
&\quad Explicit :: Atom\ \zeta\ (*) \qquad\qquad \to Field\ \zeta \\
&\quad Implicit :: Atom\ \zeta\ Constraint \to Field\ \zeta \\
&\mathbf{type}\ DataType\ \zeta = {}'[{}'[Field\ \zeta]]
\end{aligned}$$

Adapt the interpretation of $Atom$ to work on top of $Field$:

$$\begin{aligned}
&\mathbf{data}\ NA\ (\zeta :: Kind) :: \Gamma\ \zeta \to Field\ \zeta \to *\ \mathbf{where} \\
&\quad E :: \forall\ \zeta\ t\ a\ .\ Ty\ \zeta\ a\ t \to NA\ \zeta\ a\ (Explicit\ t) \\
&\quad I\ \ :: \forall\ \zeta\ t\ a\ .\ Ty\ \zeta\ a\ t \Rightarrow NA\ \zeta\ a\ (Implicit\ t)
\end{aligned}$$

Example: Representing a GADT

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} :: * \to * \ \textbf{where} \\
\quad \mathit{Lit} \quad\quad :: a \to \mathit{Expr}\ a \\
\quad \mathit{IsZero} :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool} \\
\quad \mathit{If} \quad\quad\ :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ a \to \mathit{Expr}\ a \to \mathit{Expr}\ a
\end{array}
$$

Example: Representing a GADT

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} :: * \to * \ \textbf{where} \\
\quad \mathit{Lit} \quad\ \ :: a \to \mathit{Expr}\ a \\
\quad \mathit{IsZero} :: (a \sim \mathit{Bool}) \Rightarrow \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ a \\
\quad \mathit{If} \quad\ \ \ :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ a \to \mathit{Expr}\ a \to \mathit{Expr}\ a
\end{array}
$$

Example: Representing a GADT

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} :: * \to * \ \textbf{where} \\
\quad \mathit{Lit} \quad\;\; :: a \to \mathit{Expr}\ a \\
\quad \mathit{IsZero} :: (a \sim \mathit{Bool}) \Rightarrow \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ a \\
\quad \mathit{If} \quad\;\;\; :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ a \to \mathit{Expr}\ a \to \mathit{Expr}\ a
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{CodeExpr} \\
\quad = {}'[\ {}'[\ \mathit{Explicit}\ \mathit{V0}\ ] \\
\qquad ,\ {}'[\ \mathit{Implicit}\ (\mathit{Kon}\ (\sim)\ :@:\ \mathit{V0}\ :@:\ \mathit{Kon}\ \mathit{Bool}) \\
\qquad\quad ,\ \mathit{Explicit}\ (\mathit{Kon}\ \mathit{Expr}\ :@:\ \mathit{Kon}\ \mathit{Int})\ ] \\
\qquad ,\ \ldots \\
\qquad]
\end{array}
$$

Generic GADTs: Extensions Limitations

- On our paper we discuss how to handle existential types.

Generic GADTs: Extensions Limitations

- On our paper we discuss how to handle existential types. The resulting interface is not user-friendly and make the writing of generic combinators cumbersome.

Generic GADTs: Extensions Limitations

- On our paper we discuss how to handle existential types. The resulting interface is not user-friendly and make the writing of generic combinators cumbersome.

- Existential kinds pose a problem on the other hand. We can't represent telescopes like:

$$\textbf{data}\ \mathit{Problem} :: k \to * \ \textbf{where}$$
$$\quad \mathit{Constructor} :: \forall\ k\ (a :: k)\ .\ X\ a \to \mathit{Problem}\ a$$

Arity-generic fmap

We are able to generalize $Functor$ and $BiFunctor$ to $NFunctor$.

Arity-generic fmap

We are able to generalize $\mathit{Functor}$ and $\mathit{BiFunctor}$ to $\mathit{NFunctor}$.

That is, let f be of kind $* \to * \to \ldots \to *$, we can then write:

$$\begin{aligned} \mathit{fmapN} :: & (a_1 \to b_1) \\ & \to \ldots \\ & \to (a_n \to b_n) \\ & \to f\ a_1\ \ldots\ a_n \\ & \to f\ b_1\ \ldots\ b_n \end{aligned}$$

Discussion and Future Work

- We are able to represent a reasonable amount of GADTs generically.

Discussion and Future Work

- We are able to represent a reasonable amount of GADTs generically.
- Our approach also extend to mutually recursive types as long as we do not bring in explicit fixpoints.

Discussion and Future Work

- We are able to represent a reasonable amount of GADTs generically.
- Our approach also extend to mutually recursive types as long as we do not bring in explicit fixpoints.
- Fork `generics-mrsop` and package these ideas into a usable library.

Generic Programming of All Kinds

Alejandro Serrano Mena, Victor Cacciari Miraldo

February 28, 2018