Generic Programming with Multiple Parameters

José Pedro Magalhães

Functional and Logic Programming 2014

5 June 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
- There are several generic programming libraries, with different functionality, ease of use, etc.
- This talk is about generalising one particular popular generic programming library in Haskell

Algebraic datatypes

What is the *shape* of algebraic datatypes?

Algebraic datatypes

What is the *shape* of algebraic datatypes?

$$\begin{array}{lllll} \textbf{data}\ \mathit{List}\ \alpha & = \mathit{Nil} & \mid \mathit{Cons}\ \alpha & (\mathit{List}\ \alpha) \\ \textbf{data}\ [\alpha] & = [\,] & \mid \alpha : & [\alpha] \end{array}$$

Algebraic datatypes

What is the *shape* of algebraic datatypes?

$$\textbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$
$$\textbf{data}\ [\alpha] = [\,] \mid \alpha : [\alpha]$$

$$\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha$$

Algebraic datatypes

What is the *shape* of algebraic datatypes?

$$\textbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$
$$\textbf{data}\ [\,\alpha\,] = [\,] \mid \alpha :\ [\,\alpha\,]$$

$$\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha$$

$$\textbf{data}\ (\alpha, \beta) = (\alpha, \beta)$$

Algebraic datatypes

What is the *shape* of algebraic datatypes?

$$\textbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$
$$\textbf{data}\ [\,\alpha\,] \quad = [\,] \quad \mid \alpha : \quad [\,\alpha\,]$$

$$\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha$$

$$\textbf{data}\ (\alpha, \beta) = (\alpha, \beta)$$

$$\textbf{data}\ \mathit{RTree}\ \alpha = \mathit{RTree}\ \alpha\ [\,\mathit{RTree}\ \alpha\,]$$

Algebraic datatypes

What is the *shape* of algebraic datatypes?

$$\textbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$
$$\textbf{data}\ [\alpha] = [\,] \mid \alpha : [\alpha]$$

$$\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha$$

$$\textbf{data}\ (\alpha, \beta) = (\alpha, \beta)$$

$$\textbf{data}\ \mathit{RTree}\ \alpha = \mathit{RTree}\ \alpha\ [\mathit{RTree}\ \alpha]$$

A value of an algebraic datatype is a *choice* between a *tuple* of *arguments*.

GHC Generics I

With generic programming, we model the shape of algebraic datatypes in a single representation:

$$
\begin{aligned}
&\textbf{kind}\ \mathit{Univ} = \\
&\qquad U \\
&\quad \mid P \\
&\quad \mid K\ \star \\
&\quad \mid R\ (\star \to \star) \\
&\quad \mid \mathit{Univ}\ {:}{+}{:}\ \mathit{Univ} \\
&\\
&\quad \mid \mathit{Univ}\ {:}{\times}{:}\ \mathit{Univ} \\
&\quad \mid \star \to \star\ {:}@{:}\ \mathit{Univ}
\end{aligned}
$$

GHC Generics I

With generic programming, we model the shape of algebraic datatypes in a single representation:

$$
\begin{array}{lll}
\textbf{kind}\ \mathit{Univ} = & & \\
\quad\ \ \mathit{U} & & \text{-- constructor with no arguments} \\
\quad |\ \mathit{P} & & \text{-- parameter} \\
\quad |\ \mathit{K}\ \star & & \text{-- base type (constant)} \\
\quad |\ \mathit{R}\ (\star \to \star) & & \text{-- recursion} \\
\quad |\ \mathit{Univ}\ {:}{+}{:}\ \mathit{Univ} & & \text{-- choice} \\
& & \\
\quad |\ \mathit{Univ}\ {:}{\times}{:}\ \mathit{Univ} & & \text{-- tuples} \\
\quad |\ \star \to \star\ {:}@{:}\ \mathit{Univ} & & \text{-- application}
\end{array}
$$

GHC Generics I

With generic programming, we model the shape of algebraic datatypes in a single representation:

$$
\begin{array}{l}
\textbf{kind}\ \mathit{Univ} = \\
\quad\ \ \mathit{U} \\
\quad \mid \mathit{P} \\
\quad \mid \mathit{K}\ \star \\
\quad \mid \mathit{R}\ (\star \to \star) \\
\quad \mid \mathit{Univ}\ {:}{+}{:}\ \mathit{Univ} \\
\\
\quad \mid \mathit{Univ}\ {:}{\times}{:}\ \mathit{Univ} \\
\quad \mid \star \to \star\ {:}@{:}\ \mathit{Univ}
\end{array}
\qquad
\begin{array}{llrll}
\multicolumn{5}{l}{\textbf{data}\ \mathit{In}\ (\upsilon :: \mathit{Univ})\ (\rho :: \star) :: \star\ \textbf{where}} \\
\mathit{U_1} & :: & & \mathit{In}\ U & \rho \\
\mathit{Par_1} & :: \rho & \to & \mathit{In}\ P & \rho \\
\mathit{K_1} & :: \alpha & \to & \mathit{In}\ (K\ \alpha) & \rho \\
\mathit{Rec_1} & :: \phi\ \rho & \to & \mathit{In}\ (R\ \phi) & \rho \\
\mathit{L_1} & :: \mathit{In}\ \phi\ \rho & \to & \mathit{In}\ (\phi\ {:}{+}{:}\ \psi) & \rho \\
\mathit{R_1} & :: \mathit{In}\ \psi\ \rho & \to & \mathit{In}\ (\phi\ {:}{+}{:}\ \psi) & \rho \\
({:}{\times}{:}) & :: \mathit{In}\ \phi\ \rho \to \mathit{In}\ \psi\ \rho & \to & \mathit{In}\ (\phi\ {:}{\times}{:}\ \psi) & \rho \\
\mathit{App_1} & :: \phi\ (\mathit{In}\ \psi\ \rho) & \to & \mathit{In}\ (\phi\ {:}@{:}\ \psi) & \rho
\end{array}
$$

GHC Generics II

The class *Generic* groups the types that can be handled generically:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Generic}\ (\alpha :: \star)\ \textbf{where} \\
\quad \mathit{Rep}\ \alpha :: \mathit{Univ} \\
\quad \mathit{Par}\ \ \alpha :: \star \\
\quad \mathit{from} :: \alpha \to \mathit{In}\ (\mathit{Rep}\ \alpha)\ (\mathit{Par}\ \alpha) \\
\quad \mathit{to}\ \ \ \ :: \mathit{In}\ (\mathit{Rep}\ \alpha)\ (\mathit{Par}\ \alpha) \to \alpha
\end{array}
$$

GHC Generics II

The class *Generic* groups the types that can be handled generically:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Generic}\ (\alpha :: \star)\ \textbf{where} \\
\quad \mathit{Rep}\ \alpha :: \mathit{Univ} \\
\quad \mathit{Par}\ \ \alpha :: \star \\
\quad \mathit{from} :: \alpha \rightarrow \mathit{In}\ (\mathit{Rep}\ \alpha)\ (\mathit{Par}\ \alpha) \\
\quad \mathit{to}\ \ \ \ :: \mathit{In}\ (\mathit{Rep}\ \alpha)\ (\mathit{Par}\ \alpha) \rightarrow \alpha
\end{array}
$$

As an example, we show the encoding of lists:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Generic}\ [\,\alpha\,]\ \textbf{where} \\
\quad \mathit{Rep}\ [\,\alpha\,] = U \mathrel{:+:} P \mathrel{:\times:} R\ [\,] \\
\quad \mathit{Par}\ \ [\,\alpha\,] = \alpha \\
\quad \mathit{from}\ [\,] \qquad\quad = L_1\ U_1 \\
\quad \mathit{from}\ (h : t) = R_1\ (\mathit{Par}_1\ h \mathrel{:\times:} \mathit{Rec}_1\ t) \\
\quad \mathit{to}\ (L_1\ U_1) \qquad\qquad\qquad\qquad = [\,] \\
\quad \mathit{to}\ (R_1\ (\mathit{Par}_1\ h \mathrel{:\times:} \mathit{Rec}_1\ t)) = h : t
\end{array}
$$

This code is derived automatically by the compiler.

GHC Generics III

A slightly more complicated encoding is that of rose (or multiway) trees:

$$\textbf{data}\ \mathit{RTree}\ \alpha = \mathit{RTree}\ \alpha\ [\,\mathit{RTree}\ \alpha\,]$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Generic}\ (\mathit{RTree}\ \alpha)\ \textbf{where} \\
\quad \mathit{Rep}\ (\mathit{RTree}\ \alpha) = \mathit{P}\ {:}{\times}{:}\ ([\,]\ {:}@{:}\ \mathit{R}\ \mathit{RTree}) \\
\quad \mathit{Par}\ \ (\mathit{RTree}\ \alpha) = \alpha \\
\quad \mathit{from}\ (\mathit{RTree}\ x\ \mathit{xs}) = \mathit{Par}_1\ x\ {:}{\times}{:}\ \mathit{App}_1\ (\mathit{fmap}\ \mathit{Rec}_1\ \mathit{xs}) \\
\quad \mathit{to} \ldots
\end{array}$$

Generic map (one parameter) I

The class *Functor* implements mapping over container types in Haskell:

$$\textbf{class}\ \mathit{Functor}\ (\phi :: \star \to \star)\ \textbf{where}$$
$$\quad \mathit{fmap} :: (\alpha \to \beta) \to \phi\ \alpha \to \phi\ \beta$$

Generic map (one parameter) I

The class *Functor* implements mapping over container types in Haskell:

$$\textbf{class}\ \mathit{Functor}\ (\phi :: \star \to \star)\ \textbf{where}$$
$$\quad \mathit{fmap} :: (\alpha \to \beta) \to \phi\ \alpha \to \phi\ \beta$$

We could now define an instance for lists:

$$\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}$$
$$\quad \mathit{fmap}\ f\ [\,] \qquad = [\,]$$
$$\quad \mathit{fmap}\ f\ (h : t) = f\ h : \mathit{fmap}\ f\ t$$

Generic map (one parameter) I

The class *Functor* implements mapping over container types in Haskell:

$$\textbf{class}\ \mathit{Functor}\ (\phi :: \star \to \star)\ \textbf{where}$$
$$\quad \mathit{fmap} :: (\alpha \to \beta) \to \phi\ \alpha \to \phi\ \beta$$

We could now define an instance for lists:

$$\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}$$
$$\quad \mathit{fmap}\ f\ [\,] \qquad = [\,]$$
$$\quad \mathit{fmap}\ f\ (h : t) = f\ h : \mathit{fmap}\ f\ t$$

And another one for *RTree*...

$$\textbf{instance}\ \mathit{Functor}\ \mathit{RTree}\ \textbf{where}$$
$$\quad \mathit{fmap}\ f\ (\mathit{RTree}\ a\ ts) = \mathit{RTree}\ (f\ a)\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ ts)$$

Generic map (one parameter) II

... but fortunately we don't have to. Map is a generic function, so we can give a single definition that will operate on all *Generic* types.

For that we need to define how to map over the representation types. We use a type class for this:

$$\textbf{class}\ \mathit{Functor}_R\ (\upsilon :: \mathit{Univ})\ \textbf{where}$$
$$\quad \mathit{fmap}_R :: (\alpha \rightarrow \beta) \rightarrow \mathit{In}\ \upsilon\ \alpha \rightarrow \mathit{In}\ \upsilon\ \beta$$

Generic map (one parameter) III

And now we give instances for each of the representation types:

$$
\begin{array}{l}
\textbf{instance } \mathit{Functor}_R\ U \textbf{ where} \\
\quad \mathit{fmap}_R\ \_\ U_1 = U_1 \\
\textbf{instance } \mathit{Functor}_R\ (K\ \alpha) \textbf{ where} \\
\quad \mathit{fmap}_R\ \_\ (K_1\ x) = K_1\ x \\
\textbf{instance } (\mathit{Functor}_R\ \phi, \mathit{Functor}_R\ \psi) \Rightarrow \mathit{Functor}_R\ (\phi :+: \psi) \textbf{ where} \\
\quad \mathit{fmap}_R\ f\ (L_1\ x) = L_1\ (\mathit{fmap}_R\ f\ x) \\
\quad \mathit{fmap}_R\ f\ (R_1\ x) = R_1\ (\mathit{fmap}_R\ f\ x) \\
\textbf{instance } (\mathit{Functor}_R\ \phi, \mathit{Functor}_R\ \psi) \Rightarrow \mathit{Functor}_R\ (\phi :\times: \psi) \textbf{ where} \\
\quad \mathit{fmap}_R\ f\ (x :\times: y) = \mathit{fmap}_R\ f\ x :\times: \mathit{fmap}_R\ f\ y
\end{array}
$$

Generic map (one parameter) IV

These are the most interesting cases:

$$\textbf{instance}\ \mathit{Functor}_R\ P\ \textbf{where}$$
$$\quad \mathit{fmap}_R\ f\ (\mathit{Par}_1\ x) = \mathit{Par}_1\ (f\ x)$$
$$\textbf{instance}\ (\mathit{Functor}\ \phi) \Rightarrow \mathit{Functor}_R\ (R\ \phi)\ \textbf{where}$$
$$\quad \mathit{fmap}_R\ f\ (\mathit{Rec}_1\ x) = \mathit{Rec}_1\ (\mathit{fmap}\ f\ x)$$
$$\textbf{instance}\ (\mathit{Functor}\ \phi, \mathit{Functor}_R\ \upsilon) \Rightarrow \mathit{Functor}_R\ (\phi :@: \upsilon)\ \textbf{where}$$
$$\quad \mathit{fmap}_R\ f\ (\mathit{App}_1\ x) = \mathit{App}_1\ (\mathit{fmap}\ (\mathit{fmap}_R\ f)\ x)$$

Generic map (one parameter) IV

These are the most interesting cases:

$$\textbf{instance}\ \mathit{Functor_R}\ P\ \textbf{where}$$
$$\quad \mathit{fmap_R}\ f\ (\mathit{Par_1}\ x)\ = \mathit{Par_1}\ (f\ x)$$
$$\textbf{instance}\ (\mathit{Functor}\ \phi) \Rightarrow \mathit{Functor_R}\ (R\ \phi)\ \textbf{where}$$
$$\quad \mathit{fmap_R}\ f\ (\mathit{Rec_1}\ x) = \mathit{Rec_1}\ (\mathit{fmap}\ f\ x)$$
$$\textbf{instance}\ (\mathit{Functor}\ \phi, \mathit{Functor_R}\ \upsilon) \Rightarrow \mathit{Functor_R}\ (\phi :@: \upsilon)\ \textbf{where}$$
$$\quad \mathit{fmap_R}\ f\ (\mathit{App_1}\ x) = \mathit{App_1}\ (\mathit{fmap}\ (\mathit{fmap_R}\ f)\ x)$$

Defining instances for *Generic* types is now very easy:

$$\textbf{instance}\ \mathit{Functor}\ [\,]\qquad \textbf{where}\ \mathit{fmap}\ f = \mathit{to} \circ \mathit{fmap_R}\ f \circ \mathit{from}$$
$$\textbf{instance}\ \mathit{Functor}\ \mathit{RTree}\ \textbf{where}\ \mathit{fmap}\ f = \mathit{to} \circ \mathit{fmap_R}\ f \circ \mathit{from}$$

Generic map (one parameter) IV

These are the most interesting cases:

$$
\begin{array}{l}
\textbf{instance } \mathit{Functor_R}\ P \textbf{ where} \\
\quad \mathit{fmap_R}\ f\ (\mathit{Par_1}\ x) = \mathit{Par_1}\ (f\ x) \\
\textbf{instance } (\mathit{Functor}\ \phi) \Rightarrow \mathit{Functor_R}\ (R\ \phi) \textbf{ where} \\
\quad \mathit{fmap_R}\ f\ (\mathit{Rec_1}\ x) = \mathit{Rec_1}\ (\mathit{fmap}\ f\ x) \\
\textbf{instance } (\mathit{Functor}\ \phi, \mathit{Functor_R}\ \upsilon) \Rightarrow \mathit{Functor_R}\ (\phi\ {:}@{:}\ \upsilon) \textbf{ where} \\
\quad \mathit{fmap_R}\ f\ (\mathit{App_1}\ x) = \mathit{App_1}\ (\mathit{fmap}\ (\mathit{fmap_R}\ f)\ x)
\end{array}
$$

Defining instances for *Generic* types is now very easy:

$$
\begin{array}{l}
\textbf{instance } \mathit{Functor}\ [\,] \\
\textbf{instance } \mathit{Functor}\ \mathit{RTree}
\end{array}
$$

(And even easier if we use `-XDefaultSignatures`.)

Map over multiple parameters

All is good so far, but what if I want to define the following map?

$$
\begin{aligned}
\textbf{data}\ \mathit{WTree}\ \alpha\ \omega = {} & \mathit{Leaf}\ \alpha \\
\mid {} & \mathit{Fork}\ (\mathit{WTree}\ \alpha\ \omega)\ (\mathit{WTree}\ \alpha\ \omega) \\
\mid {} & \mathit{WithWeight}\ (\mathit{WTree}\ \alpha\ \omega)\ \omega
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{mapWTree} :: (\alpha \to \alpha') \to (\omega \to \omega') \to \mathit{WTree}\ \alpha\ \omega \to \mathit{WTree}\ \alpha'\ \omega' \\
& \mathit{mapWTree}\ f\ g\ (\mathit{Leaf}\ a) && = \mathit{Leaf}\ (f\ a) \\
& \mathit{mapWTree}\ f\ g\ (\mathit{Fork}\ l\ r) && = \mathit{Fork}\ (\mathit{mapWTree}\ f\ g\ l) \\
& && \qquad\quad (\mathit{mapWTree}\ f\ g\ r) \\
& \mathit{mapWTree}\ f\ g\ (\mathit{WithWeight}\ t\ w) && = \mathit{WithWeight}\ (\mathit{mapWTree}\ f\ g\ t) \\
& && \qquad\qquad\qquad\quad (g\ w)
\end{aligned}
$$

Map over multiple parameters

All is good so far, but what if I want to define the following map?

$$
\begin{aligned}
\textbf{data}\ \mathit{WTree}\ \alpha\ \omega = &\ \mathit{Leaf}\ \alpha \\
\mid &\ \mathit{Fork}\ (\mathit{WTree}\ \alpha\ \omega)\ (\mathit{WTree}\ \alpha\ \omega) \\
\mid &\ \mathit{WithWeight}\ (\mathit{WTree}\ \alpha\ \omega)\ \omega
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mapWTree} :: (\alpha \to \alpha') \to (\omega \to \omega') \to \mathit{WTree}\ \alpha\ \omega \to \mathit{WTree}\ \alpha'\ \omega' \\
&\mathit{mapWTree}\ f\ g\ (\mathit{Leaf}\ a) &&= \mathit{Leaf}\ (f\ a) \\
&\mathit{mapWTree}\ f\ g\ (\mathit{Fork}\ l\ r) &&= \mathit{Fork}\ (\mathit{mapWTree}\ f\ g\ l) \\
& && \qquad\quad (\mathit{mapWTree}\ f\ g\ r) \\
&\mathit{mapWTree}\ f\ g\ (\mathit{WithWeight}\ t\ w) &&= \mathit{WithWeight}\ (\mathit{mapWTree}\ f\ g\ t) \\
& && \qquad\qquad\qquad\quad (g\ w)
\end{aligned}
$$

With GHC generics, all we can get is a map over the ω parameter.

Generic map over multiple parameters I

The focus of this work is to generalise generics in GHC to support generic functions over multiple parameters.

Generic map over multiple parameters I

The focus of this work is to generalise generics in GHC to support generic functions over multiple parameters.

With this generalisation, we can write a generic map *gmap* over multiple parameters:

$$\textbf{instance}\ \mathit{GMap}\ \mathit{WTree}\ `[\alpha \to \alpha', \omega \to \omega']$$
$$\mathit{mapWTree}\ f\ g \simeq \mathit{gmap}\ (\mathit{HCons}\ f\ (\mathit{HCons}\ g\ \mathit{HNil}))$$

Generic map over multiple parameters I

The focus of this work is to generalise generics in GHC to support generic functions over multiple parameters.

With this generalisation, we can write a generic map *gmap* over multiple parameters:

$$\textbf{instance}\ \mathit{GMap}\ \mathit{WTree}\ `[\alpha \to \alpha', \omega \to \omega']$$
$$\mathit{mapWTree}\ f\ g \simeq \mathit{gmap}\ (\mathit{HCons}\ f\ (\mathit{HCons}\ g\ \mathit{HNil}))$$

$$\textbf{instance}\ \mathit{GMap}\ (,)\ `[\alpha \to \alpha', \beta \to \beta']$$
$$\mathit{example} :: (\mathit{Int}, \mathit{Float})$$
$$\mathit{example} = \mathit{gmap}\ (\mathit{HCons}\ (+1)\ (\mathit{HCons}\ (+1.1)\ \mathit{HNil}))\ (0, 0.0)$$

Generalising GHC Generics I

The first step is to generalise the universe to include support for multiple parameters:

$$
\begin{array}{l}
\textbf{kind}\ \mathit{Univ} = \\
\quad \mathit{U} \\
\quad \mid \mathit{F}\ \mathit{Field} \\
\quad \mid \mathit{Univ}\ {:+:}\ \mathit{Univ} \\
\\
\quad \mid \mathit{Univ}\ {:\times:}\ \mathit{Univ}
\end{array}
\qquad
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathit{In}\ (\upsilon :: \mathit{Univ})\ (\rho :: [\star]) :: \star\ \textbf{where}} \\
\quad \mathit{U} & :: & & \mathit{In}\ \mathit{U}\ \rho \\
\quad \mathit{F} & :: & \mathit{InField}\ \upsilon\ \rho & \to \mathit{In}\ (\mathit{F}\ \upsilon)\ \rho \\
\quad \mathit{L} & :: & \mathit{In}\ \alpha\ \rho & \to \mathit{In}\ (\alpha\ {:+:}\ \beta)\ \rho \\
\quad \mathit{R} & :: & \mathit{In}\ \beta\ \rho & \to \mathit{In}\ (\alpha\ {:+:}\ \beta)\ \rho \\
\quad ({:\times:}) & :: & \mathit{In}\ \alpha\ \rho \to \mathit{In}\ \beta\ \rho & \to \mathit{In}\ (\alpha\ {:\times:}\ \beta)\ \rho
\end{array}
$$

$$
\begin{array}{l}
\textbf{kind}\ \mathit{Field} = \\
\quad \mathit{K}\ \star \\
\quad \mid \mathit{P}\ \mathit{Nat} \\
\quad \mid \forall \kappa.\kappa\ {:@:}\ [\,\mathit{Field}\,]
\end{array}
\qquad
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathit{InField}\ (\upsilon :: \mathit{Field})\ (\rho :: [\star]) :: \star\ \textbf{where}} \\
\quad \mathit{K} & :: & \alpha & \to \mathit{InField}\ (\mathit{K}\ \alpha)\ \rho \\
\quad \mathit{P} & :: & \rho\ {:!:}\ \nu & \to \mathit{InField}\ (\mathit{P}\ \nu)\ \rho \\
\quad \mathit{A} & :: & \mathit{AppFields}\ \sigma\ \chi\ \rho & \to \mathit{InField}\ (\sigma\ {:@:}\ \chi)\ \rho
\end{array}
$$

Generalising GHC Generics II

Some auxiliary type-level computations:

$$\textbf{kind}\ \mathit{Nat} = \mathit{Ze} \mid \mathit{Su}\ \mathit{Nat}$$

$$\begin{aligned}
&(\rho :: [\star])\ \mathbin{:!:}\ (\nu :: \mathit{Nat}) :: \star \\
&(\alpha\ \text{`}{:}\ \rho)\ \mathbin{:!:}\ \mathit{Ze} \qquad\ = \alpha \\
&(\alpha\ \text{`}{:}\ \rho)\ \mathbin{:!:}\ (\mathit{Su}\ \nu) = \rho\ \mathbin{:!:}\ \nu
\end{aligned}$$

Generalising GHC Generics II

Some auxiliary type-level computations:

$$\textbf{kind}\ \mathit{Nat} = \mathit{Ze} \mid \mathit{Su}\ \mathit{Nat}$$

$$\begin{array}{l}
(\rho :: [\star])\ {:!:}\ (\nu :: \mathit{Nat}) :: \star \\
(\alpha\ {`:}\ \rho)\ {:!:}\ \mathit{Ze} \qquad\ = \alpha \\
(\alpha\ {`:}\ \rho)\ {:!:}\ (\mathit{Su}\ \nu) = \rho\ {:!:}\ \nu
\end{array}$$

$$\mathit{AppFields}\ \sigma\ \chi\ \rho = \sigma\ {:\$:}\ \mathit{ExpFld}\ \chi\ \rho$$

$$\begin{array}{l}
(\sigma :: \kappa)\ {:\$:}\ (\rho :: [\star]) :: \star \\
\sigma\ {:\$:}\ `[] \qquad\quad = \sigma \\
\sigma\ {:\$:}\ (\alpha\ {`:}\ \beta) = (\sigma\ \alpha)\ {:\$:}\ \beta
\end{array}$$

$$\begin{array}{l}
\mathit{ExpFld}\ (\chi :: [\mathit{Field}])\ (\rho :: [\star]) :: [\star] \\
\mathit{ExpFld}\ `[] \qquad\qquad\qquad\quad \rho = `[] \\
\mathit{ExpFld}\ ((K\ \alpha)\ {`:} \quad\ \chi)\ \rho = \alpha\ {`:} \qquad\qquad\qquad\quad \mathit{ExpFld}\ \chi\ \rho \\
\mathit{ExpFld}\ ((P\ \nu)\ {`:} \quad\ \chi)\ \rho = (\rho\ {:!:}\ \nu)\ {`:} \qquad\qquad \mathit{ExpFld}\ \chi\ \rho \\
\mathit{ExpFld}\ ((\sigma\ {:@:}\ \omega)\ {`:}\ \chi)\ \rho = (\sigma\ {:\$:}\ \mathit{ExpFld}\ \omega\ \rho)\ {`:}\ \mathit{ExpFld}\ \chi\ \rho
\end{array}$$

Generalising GHC Generics III

We adapt the *Generic* class to encode the parameters as a type-level list:

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

Generalising GHC Generics III

We adapt the *Generic* class to encode the parameters as a type-level list:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Generic}\ (\alpha :: \star)\ \mathbf{where} \\
\quad \mathit{Rep}\ \ \alpha :: \mathit{Univ} \\
\quad \mathit{Pars}\ \alpha :: [\star] \\
\quad \mathit{from} :: \alpha \rightarrow \mathit{In}\ (\mathit{Rep}\ \alpha)\ (\mathit{Pars}\ \alpha) \\
\quad \mathit{to} \quad\ :: \mathit{In}\ (\mathit{Rep}\ \alpha)\ (\mathit{Pars}\ \alpha) \rightarrow \alpha
\end{array}
$$

Here is the instance for lists:

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Generic}\ [\alpha]\ \mathbf{where} \\
\quad \mathit{Rep}\ \ [\alpha] = \mathit{U}\ {:}{+}{:}\ \mathit{F}\ (\mathit{P}\ \mathit{0})\ {:}{\times}{:}\ \mathit{F}\ ([\,]\ {:}@{:}\ `[\mathit{P}\ \mathit{0}]) \\
\quad \mathit{Pars}\ [\alpha] = `[\alpha] \\
\quad \mathit{from}\ [\,] \qquad\ = \mathit{L}\ \mathit{U} \\
\quad \mathit{from}\ (h : t) = \mathit{R}\ (\mathit{F}\ (\mathit{P}\ h)\ {:}{\times}{:}\ \mathit{F}\ (\mathit{A}\ t))
\end{array}
$$

Generalising GHC Generics IV

But now we can also encode datatypes with multiple parameters:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Generic}\ (\alpha,\beta)\ \textbf{where} \\
\quad \mathit{Rep}\ \ (\alpha,\beta) = F\ (P\ \mathit{0})\ {:}{\times}{:}\ F\ (P\ \mathit{1}) \\
\quad \mathit{Pars}\ (\alpha,\beta) = `[\alpha,\beta] \\
\quad \mathit{from}\ (a,b) = F\ (P\ a)\ {:}{\times}{:}\ F\ (P\ b)
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ D\ \alpha\ \beta = D\ \beta\ [\,(\alpha,\mathit{Int})\,] \\
\textbf{instance}\ \mathit{Generic}\ (D\ \alpha\ \beta)\ \textbf{where} \\
\quad \mathit{Rep}\ \ (D\ \alpha\ \beta) = F\ (P\ \mathit{1})\ {:}{\times}{:}\ F\ ([\,]\ {:}@{:}\ `[(,)\ {:}@{:}\ `[P\ \mathit{0}, K\ \mathit{Int}]]) \\
\quad \mathit{Pars}\ (D\ \alpha\ \beta) = `[\alpha,\beta] \\
\quad \mathit{from}\ (D\ a\ b) = F\ (P\ a)\ {:}{\times}{:}\ F\ (A\ b)
\end{array}
$$

Generic map over multiple parameters II

Since we can now map an arbitrary number of functions, we need arbitrary-length tuples (heterogenous collections):

$$
\begin{array}{l}
\textbf{data}\ \mathit{HList}\ (\rho :: [\star])\ \textbf{where} \\
\quad \mathit{HNil} \quad :: \qquad\qquad\qquad\quad \mathit{HList}\ \text{'}[\,] \\
\quad \mathit{HCons} :: \alpha \rightarrow \mathit{HList}\ \beta \rightarrow \mathit{HList}\ (\alpha\ \text{'}{:}\ \beta)
\end{array}
$$

We can then express the user-facing class for the generalised map:

$$
\begin{array}{l}
\textbf{class}\ \mathit{GMap}\ (\sigma :: \kappa)\ (\tau :: [\star])\ |\ \tau \rightarrow \kappa\ \textbf{where} \\
\quad \mathit{gmap} :: \mathit{HList}\ \tau \rightarrow \sigma\ {:}\$ {:}\ \mathit{Doms}\ \tau \rightarrow \sigma\ {:}\$ {:}\ \mathit{Codoms}\ \tau
\end{array}
$$

Limitation to kind $\star$ parameters

Consider the following datatype:

$$\mathbf{data}\ \mathit{GTree}_1\ \phi\ \alpha = \mathit{GTree}_1\ \alpha\ (\phi\ (\mathit{GTree}_1\ \phi\ \alpha))$$

We would like to obtain the following map for it:

$$\begin{aligned}
&\mathit{gmap1}\ ::\ (\alpha \to \beta) \to (\forall \alpha\ \beta.(\alpha \to \beta) \to \phi\ \alpha \to \psi\ \beta)\\
&\qquad \to \mathit{GTree}_1\ \phi\ \alpha \to \mathit{GTree}_1\ \psi\ \beta\\
&\mathit{gmap1}\ f\ g\ (\mathit{GTree}_1\ x\ y) = \mathit{GTree}_1\ (f\ x)\ (g\ (\mathit{gmap1}\ f\ g)\ y)
\end{aligned}$$

But this is not yet possible with our approach.

Conclusion

- We've seen how to encode a generic representation that supports abstraction over multiple parameters (of kind $\star$);
- We've defined a generalised map;
- Other generalised functions are now possible, e.g. folding, traversing, and zipping;
- We hope to implement support for multiple parameters in GHC soon.

Conclusion

- We've seen how to encode a generic representation that supports abstraction over multiple parameters (of kind $\star$);
- We've defined a generalised map;
- Other generalised functions are now possible, e.g. folding, traversing, and zipping;
- We hope to implement support for multiple parameters in GHC soon.

Thank you for your time!