Generic Programming with Multiple Parameters

José Pedro Magalhães

Department of Computer Science, University of Oxford, Oxford, UK
jpm@cs.ox.ac.uk

Abstract. Generic programming, a form of abstraction in programming languages that serves to reduce code duplication by exploiting the regular structure of algebraic datatypes, has been present in the Haskell language in different forms for many years. Lately, a library for generic deriving has been given native support in the compiler, allowing programmers to write functions such as *fmap* that abstract over one datatype parameter generically. The power of this approach is limited to dealing with one parameter per datatype, however. In this paper, we lift this restriction by providing a generalisation of generic deriving that supports multiple parameters, making essential use of datatype promotion and kind polymorphism. We show example encodings of datatypes, and how to define a map function that operates on multiple parameters simultaneously.

1 Introduction

Haskell [12], a pure, lazy, strongly-typed functional programming language, has been a research vehicle for generic programming almost since its inception [2]. Generic programming is a form of abstraction which exploits the structure of algebraic datatypes in order to define functionality that operates on many datatypes uniformely, reducing code duplication. With generic programming, certain functions (such as data serialisation and traversals) can be written once and for all, working for existing and future types of data.

Early approaches to generic programming in Haskell were separate extensions to the compiler or preprocessors [3], but recently all approaches are bundled as libraries [13], sometimes with direct compiler support [11]. The easier availability of generic programming functionality appears to increase its usage; in particular, offering native support for an approach in the compiler confers a sense of stability to a specific approach, allowing more programmers to use and benefit from generic programming.

This switch to library approaches simplifies the task of the generic programming library developer, since it is typically easier to maintain a library than a separate extension. It also presents the library user with fewer barriers to adoption, as it requires no external tools. However, it can compromise expressivity and usability, as library approaches are limited to the capabilities of the Haskell language itself. Performance and quality of error messages are common complaints of generic programming libraries, but also reduced support for certain datatypes can be a concern. In particular, the Generic Haskell compiler [6], an early, pre-processor approach, had full support for generic functions abstracting over multiple parameters. The same goes for the generics in the Clean language [1], which were implemented in the kind-indexed style of

Generic Haskell. However, to our knowledge no library approach to generic programming in Haskell has native support for abstraction over multiple datatype parameters. Concretely, consider the following example:

```
data WTree α ω = Leaf α
               | Fork (WTree α ω) (WTree α ω)
               | WithWeight (WTree α ω) ω
mapWTree :: (α → α′) → (ω → ω′) → WTree α ω → WTree α′ ω′
mapWTree f g (Leaf a)         = Leaf (f a)
mapWTree f g (Fork l r)       = Fork (mapWTree f g l) (mapWTree f g r)
mapWTree f g (WithWeight t w) = WithWeight (mapWTree f g t) (g w)
```

Clearly, functions like *mapWTree* follow the structure of the datatype, and should be defined generically, once and for all. In this paper we focus on the problem of defininig *functions* such as *mapWTree* generically; many (if not all) modern generic programming approaches support the *WTree datatype*, but not generic operations such as *mapWTree*. The only exception that we are aware of is a convoluted implementation using Scrap Your Boilerplate [5], which relies on runtime type comparison, casting, and (virtual) seralisation.[1] Our approach focuses instead on encoding the parameters adequately in the generic representation.

In this paper, we focus our attention on the `generic-deriving` approach, as implemented in the Glasgow Haskell Compiler (GHC) [8, Chapter 11]. To balance complexity and ease-of-use, `generic-deriving` was designed from the start with one compromise in mind: many generic functions (e.g. *fmap* and *traverse*) abstract over one datatype parameter (i.e. they operate on *type containers*, or types of kind $\star \to \star$), but few require abstraction over more than one parameter. As such, the design of `generic-deriving` could be kept relatively simple, at the loss of some generality. Recently, with the advent of kind polymorphism in GHC [16], many libraries are being generalised from kind-specific to kind-polymophic (e.g. *Typeable* [16]). In this paper we describe an elegant generalisation of `generic-deriving` that works with multiple parameters, lifting the one-parameter restriction without requiring the full-blown power (and complexity) of indexed functors [7]. Our solution is based on an earlier, failed attempt [9, Section 6.1], but uses new insights to overcome old challenges: we use heterogeneous collections [4] to encode a function with multiple arguments as a regular single-argument function that takes a list of arguments.

In the remainder of this paper, we first review one-parameter `generic-deriving` (Section 2). We proceed to describe our generalisation in Section 3, showing example datatype encodings (Section 3.2), and a generalised map function (Section 3.3). We list limitations and propose future work in Section 4, and conclude in Section 5.

Notation In order to avoid syntactic clutter and to help the reader, we adopt a liberal Haskell notation in this paper. Datatype promotion makes datatype definitions *also* define a kind (with the name of the datatype being defined), while the constructors *also* become the types that inhabit that kind; we assume the existence of a **kind** keyword, which allows us to define kinds that behave as if they had arisen from datatype promotion, ex-

[1] `http://okmij.org/ftp/Haskell/generics.html#gmap`

cept that they do not define a datatype and constructors. This helps us preventing name clashes. We omit the keywords **type family** and **type instance** entirely, making type-level functions look like their value-level counterparts. We colour constructors in *blue*, types in *red*, and kinds in *green*. Promoted lists are prefixed with a quote, to distinguish them from the list type. Additionally, we use Greek letters for type variables, apart from κ, which is reserved for kind variables. A grayscale version of this paper is available at `http://dreixel.net/research/pdf/gpmp_nocolour.pdf`. The syntactic sugar is only for presentation purposes. An executable version of the code, which compiles with GHC 7.6.2, is available at `http://dreixel.net/research/code/gpmp.zip`.

2 Generic programming with one parameter

In this section we review the `generic-deriving` library (in the slightly revised version of Magalhães and Löh [10]), which supports abstraction over (at most) one datatype parameter. We omit meta-information, as it is not relevant to our development.

2.1 Universe

Using datatype promotion and kind polymorphism, we can keep the “realm” of generic representations of user datatypes separate from the realm of user datatypes by defining a new kind. Like types classify values, kinds classify types. User datatypes always have kind $\star$. For the generic representation, we define a kind *Univ* that aggregates the types used to represent user datatypes in `generic-deriving`. *U* encodes constructors without arguments, *P* an occurence of the parameter, *K* a type that does not contain the parameter, *R* a type that contains the parameter, $(:+:)$ the choice between two constructors, $(:\times:)$ is used for constructors with multiple arguments, and $(:@:)$ encodes the application of a functor to a type. The interpretation datatype $In\ \upsilon\ \rho$ encodes the values associated with a representation type υ and parameter ρ:

```
kind Univ =                     data In (υ :: Univ) (ρ :: ⋆) :: ⋆ where
    U                             U_1    ::                    In U         ρ
  | P                             Par_1  :: ρ                → In P         ρ
  | K ⋆                           K_1    :: α                → In (K α)     ρ
  | R (⋆ → ⋆)                     Rec_1  :: φ ρ              → In (R φ)     ρ
  | Univ :+: Univ                 L_1    :: In φ ρ           → In (φ :+: ψ) ρ
                                  R_1    :: In ψ ρ           → In (φ :+: ψ) ρ
  | Univ :×: Univ                 (:×:)  :: In φ ρ → In ψ ρ  → In (φ :×: ψ) ρ
  | ⋆ → ⋆ :@: Univ                App_1  :: φ (In ψ ρ)       → In (φ :@: ψ) ρ
```

There is some redundacy in the universe codes; for example, $R\ \phi$ is essentially a shortcut for $\phi\ :@:\ P$, but the latter has a more complicated representation ($\phi\ (In\ P\ \rho)$ instead of just $\phi\ \rho$). Our new encoding in Section 3 also solves this issue.

The *Generic* type class mediates between user datatypes and their generic representation. A *Rep* type function is used to encode the generic representation of some user datatype α, while *Par* identifies the parameter. Conversion functions *from* and *to* witness the isomorphism between α and $In\ (Rep\ \alpha)\ (Par\ \alpha)$:

$$\begin{aligned}
&\mathbf{class}\ \mathit{Generic}\ (\alpha :: \star)\ \mathbf{where}\\
&\quad \mathit{Rep}\ \alpha :: \mathit{Univ}\\
&\quad \mathit{Par}\ \ \alpha :: \star\\
&\quad \mathit{from} :: \alpha \to \mathit{In}\ (\mathit{Rep}\ \alpha)\ (\mathit{Par}\ \alpha)\\
&\quad \mathit{to}\quad\ :: \mathit{In}\ (\mathit{Rep}\ \alpha)\ (\mathit{Par}\ \alpha) \to \alpha
\end{aligned}$$

Instances of this class are trivial, but tedious to write, and are necessary for each datatype intended to be used generically. Fortunately, these instances are automatically derivable by GHC, therefore making `generic-deriving` a "built-in" generic programming approach.

2.2 Datatype encodings

The universe of `generic-deriving` is best understood through sample datatype instantiations, which we provide in this section. We start with the encoding of lists. A list is a choice ($(:+:)$) between an empty constructor (U) or two ($(:\times:)$) arguments—the parameter (P) and another list ($R\ [\,]$):

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

From this point forward we shall omit the *to* function, as it is always entirely symmetric to *from*.

A slightly more complicated encoding is that of rose (or multiway) trees. Since we use application of lists in the representation, the conversion functions need to be able to map over these lists. This is achieved using the *fmap* function, a generic map that can be defined for all *Generic* types (Section 2.3):

$$\begin{aligned}
&\mathbf{data}\ \mathit{RTree}\ \alpha = \mathit{RTree}\ \alpha\ [\mathit{RTree}\ \alpha]\\
&\mathbf{instance}\ \mathit{Generic}\ (\mathit{RTree}\ \alpha)\ \mathbf{where}\\
&\quad \mathit{Rep}\ (\mathit{RTree}\ \alpha) = P :\times: ([\,] :@: R\ \mathit{RTree})\\
&\quad \mathit{Par}\ \ (\mathit{RTree}\ \alpha) = \alpha\\
&\quad \mathit{from}\ (\mathit{RTree}\ x\ \mathit{xs}) = \mathit{Par}_1\ x :\times: \mathit{App}_1\ (\mathit{fmap}\ \mathit{Rec}_1\ \mathit{xs})
\end{aligned}$$

2.3 Mapping

As `generic-deriving` supports abstraction over one parameter, we can define the standard function *fmap* generically. The user-facing class for this function abstracts over type containers ϕ of kind $\star \to \star$:

$$\begin{aligned}
&\mathbf{class}\ \mathit{Functor}\ (\phi :: \star \to \star)\ \mathbf{where}\\
&\quad \mathit{fmap} :: (\alpha \to \beta) \to \phi\ \alpha \to \phi\ \beta
\end{aligned}$$

The generic version, fmap_R, on the other hand, operates on the generic representation. We need to give an instance of fmap_R for each representation type; we use another type class for this purpose:

$$\begin{array}{l}
\textbf{class}\ \mathit{Functor}_R\ (\upsilon :: \mathit{Univ})\ \textbf{where}\\
\quad \mathit{fmap}_R :: (\alpha \to \beta) \to \mathit{In}\ \upsilon\ \alpha \to \mathit{In}\ \upsilon\ \beta
\end{array}$$

Sums, products, units, and constants are trivial:

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

More interesting are the instances for: the parameter, where we apply the mapping function; recursion into type containers, where we recurse using *fmap*; and application of type containers, where we *fmap* on the outer container, and fmap_R on the inner representation:

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

Providing instances for types with a *Generic* instance, such as $[\alpha]$, requires only using fmap_R and converting *from*/*to* the original datatype:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ [\,]\ \textbf{where}\\
\quad \mathit{fmap}\ f = \mathit{to} \circ \mathit{fmap}_R\ f \circ \mathit{from}
\end{array}$$

We now have an easy way to define a generic *fmap* for any user datatype with a *Generic* instance.

3 Generic programming with multiple parameters

Having seen the current implementation of `generic-deriving`, we are ready to explore the changes that are necessary to make it support abstraction over multiple parameters.

3.1 Universe

The main change to the universe involves the introduction of a list of parameters, and the separation of fields into a kind of their own. A field can be one of three things.

Constant types (*K*) are unchanged. Parameters (*P* ν) now take an argument ν :: *Nat* to indicate which of the parameters it is. The kind *Nat* encodes Peano-style natural numbers; we will write *0* for *Ze*, *1* for *Su Ze*, etc. A generalised form of application ((:@:)) replaces *R* and the old (:@:), encoding the application of a type of kind κ to a type-level list of fields. As we only deal with parameters of kind $\star$ (see Section 4.1), the first argument to :@: will always have a kind of the form $\star \to \ldots \to \star$, and the second argument will have as many elements as necessary to fully saturate the first argument. The interpretation now takes a list of parameters ρ instead of a single parameter. A separate *InField* υ ρ datatype interprets a field representation type υ with parameters ρ. Constants are interpreted as before. Parameters are looked-up in the parameter list with a type-level lookup operator (:!:) akin to the value-level (!) operator. An application σ :@: χ is interpreted by applying σ to each of the arguments in χ, after transforming these (with *ExpandField*) into types of kind $\star$:

```
kind Univ =                    data In (υ :: Univ) (ρ :: [⋆]) :: ⋆ where
    U                            U   ::                          In U         ρ
  | F Field                      F   :: InField υ ρ          → In (F υ)     ρ
  | Univ :+: Univ                L   :: In α ρ               → In (α :+: β) ρ
                                 R   :: In β ρ               → In (α :+: β) ρ
  | Univ :×: Univ                :×: :: In α ρ → In β ρ      → In (α :×: β) ρ
kind Field =                   data InField (υ :: Field) (ρ :: [⋆]) :: ⋆ where
    K ⋆                          K :: α                  → InField (K α)       ρ
  | P Nat                        P :: ρ :!: ν            → InField (P ν)       ρ
  | ∀κ.κ :@: [Field]             A :: AppFields σ χ ρ    → InField (σ :@: χ)   ρ

kind Nat = Ze | Su Nat
(ρ :: [⋆]) :!: (ν :: Nat) :: ⋆
(α ': ρ) :!: Ze      = α
(α ': ρ) :!: (Su ν)  = ρ :!: ν
AppFields σ χ ρ = σ :$: ExpandField χ ρ
(σ :: κ) :$: (ρ :: [⋆]) :: ⋆
σ :$: '[]            = σ
σ :$: (α ': β)       = (σ α) :$: β
ExpandField (χ :: [Field]) (ρ :: [⋆]) :: [⋆]
ExpandField '[]                 ρ = '[]
ExpandField ((K α) ':        χ) ρ = α ':                          ExpandField χ ρ
ExpandField ((P ν) ':        χ) ρ = (ρ :!: ν) ':                  ExpandField χ ρ
ExpandField ((σ :@: ω) ': χ) ρ = (σ :$: ExpandField ω ρ) ': ExpandField χ ρ
```

The *ExpandField* type function converts a list of *Field*s into user-defined types (of kind $\star$). Constants are represented by the type in question, parameters are looked up in the parameter list, and applications are expanded into a fully applied type.

The *Generic* class to mediate between user datatypes and their representation now has a type function *Pars* which lists the parameters of the datatype:

```
class Generic (α :: ⋆) where
  Rep  α :: Univ
  Pars α :: [⋆]
  from :: α → In (Rep α) (Pars α)
  to   :: In (Rep α) (Pars α) → α
```

3.2 Datatype encodings

To better understand the universe of our generalised `generic-deriving`, we show several example datatype encodings in this section. We begin with lists; their encoding is similar to that of Section 2.2, only now we use (*:@:*) instead of *R*:

```
instance Generic [α] where
  Rep  [α] = U :+: F (P 0) :×: F ([] :@: '[P 0])
  Pars [α] = '[α]
  from []      = L U
  from (h : t) = R (F (P h) :×: F (A t))
```

The *RTree* type of Section 2.2 can also still be encoded. In fact, since we use the type family *ExpandField* to transform fields *υ* of kind *Field* into ⋆, instead of using the interpretation *InField υ* recursively, we no longer need to use *fmap*; the arguments can be used directly, simplifying the implementation of *from* and *to*:

```
instance Generic (RTree α) where
  Rep  (RTree α) = F (P 0) :×: F ([] :@: '[RTree :@: '[P 0]])
  Pars (RTree α) = '[α]
  from (RTree x xs) = F (P x) :×: F (A xs)
```

Having support for multiple parameters, we can now deal with pairs properly, for example. These are simply a product between two fields with a parameter each:

```
instance Generic (α, β) where
  Rep  (α, β) = F (P 0) :×: F (P 1)
  Pars (α, β) = '[α, β]
  from (a, b) = F (P a) :×: F (P b)
```

A more complicated example shows how we can mix datatypes with a different number of parameters, and partially instantiated datatypes:

```
data D α β γ = D β [(α, Int)] (RTree [γ])
instance Generic (D α β γ) where
  Rep  (D α β γ) =
    F (P 1) :×: F ([] :@: '[(,) :@: '[P 0, K Int]]) :×: F (RTree :@: '[[] :@: '[P 2]])
  Pars (D α β γ) = '[α, β, γ]
  from (D a b c) = F (P a) :×: F (A b) :×: F (A c)
```

Even nested datatypes can be encoded, as our application supports the notion of recursion with parameters instantiated to other applications:

```
data Perfect α = Perfect (Perfect (α, α)) | End α
instance Generic (Perfect α) where
  Rep  (Perfect α) = F (Perfect :@: '[(,) :@: '[P 0, P 0]]) :+: F (P 0)
  Pars (Perfect α) = '[α]
  from (Perfect x) = L (F (A x))
  from (End x)     = R (F (P x))
```

As we have seen, our generalisation of `generic-deriving` supports all the previous datatypes, plus many new ones, involving any number of parameters of kind ⋆.

3.3 Mapping

While designing a new or improved generic programming library, defining the universe and showing example datatype encodings is not enough; it is, in fact, easy to define a simple and encompassing universe that is then found not to be suitable for defining generic functions. As such, we show that our universe allows defining a generalised map function, which maps n different functions over n datatype parameters.

Preliminaries As the generalised map takes n functions, one per datatype parameter, we need a value-level counterpart to our type-level lists of parameters. We use a strongly typed heterogenous list [4] for this purpose:

```
data HList (ρ :: [⋆]) where
  HNil  ::                 HList '[]
  HCons :: α → HList β → HList (α ': β)
```

We also need a way to pick the n-th element from such a list; we define a *Lookup* type class for this purpose, as the function is defined by induction on the type-level natural numbers. We use a *Proxy* to carry type information at the value-level (the index being looked up):

```
data Proxy (σ :: κ) = Proxy
class Lookup (ρ :: [⋆]) (ν :: Nat) where
  lookup :: Proxy ν → HList ρ → ρ :!: ν
instance Lookup ρ Ze where
  lookup _ (HCons f _) = f
instance (Lookup β ν) ⇒ Lookup (α ': β) (Su ν) where
  lookup _ (HCons _ fs) = lookup (Proxy :: Proxy ν) fs
```

User-facing type class We are now ready to see the class that generalises *Functor* of Section 2.3. This is a multi-parameter type class, taking a parameter σ for the unsaturated type we are mapping over, and a list of function types τ encoding the types of the functions we will be mapping. As σ will often be ambiguous in the code for the generic definition of map, we provide a version $gmap_P$ with an extra argument that serves to identify the σ. A version without the proxy (*gmap*) is provided with a default for convenient usage:

```
class GMap (σ :: κ) (τ :: [⋆]) | τ → κ where
  gmap ::                HList τ → σ :$: Doms τ → σ :$: Codoms τ
  gmap = gmap_P (Proxy :: Proxy σ)

  gmap_P :: Proxy σ → HList τ → σ :$: Doms τ → σ :$: Codoms τ
```

Our generalised map, *gmap*, takes two arguments. The first is an *HList* of functions to map. The second is the type σ applied to the domains of the functions we are mapping. Its return type is again σ, but now applied to the codomains of the same functions. For example, for lists, σ is $[\,]$, and τ is $`[\alpha \to \beta]$. In this case, *gmap* gets the expected type $\mathit{HList}\ `[\alpha \to \beta] \to [\alpha] \to [\beta]$. The functional dependency $\tau \to \kappa$ is essential to prevent ambiguity when using *gmap*—and indeed the kind of σ is uniquely determined by the (length of the) list τ. The type functions *Doms* and *Codoms* compute the domains and codomains of the list of functions τ:

```
Doms (τ :: [⋆]) :: [⋆]
Doms `[]              = `[]
Doms ((α → β) `: τ) = α `: Doms τ

Codoms (τ :: [⋆]) :: [⋆]
Codoms `[]              = `[]
Codoms ((α → β) `: τ) = β `: Codoms τ
```

Mapping on the generic representation We have seen *GMap*, the user-facing type class for the generalised map. Its counterpart for representation types is GMap_R, which operates on the interpretation of a representation:

```
class GMap_R (υ :: Univ) (τ :: [⋆]) where
  gmap_R :: HList τ → In υ (Doms τ) → In υ (Codoms τ)
```

We now go through the instances of GMap_R for each representation type. There is nothing to map over in units, and sums and products are simply traversed through:

```
instance GMap_R U τ where
  gmap_R _ U        = U
instance (GMap_R α τ, GMap_R β τ) ⇒ GMap_R (α :+: β) τ where
  gmap_R fs (L x)    = L (gmap_R fs x)
  gmap_R fs (R x)    = R (gmap_R fs x)
instance (GMap_R α τ, GMap_R β τ) ⇒ GMap_R (α :×: β) τ where
  gmap_R fs (x :×: y) = gmap_R fs x :×: gmap_R fs y
```

Fields require more attention, so we define a separate type class GMap_{RF} to handle them:

```
instance (GMap_RF υ τ) ⇒ GMap_R (F υ) τ where
  gmap_R fs (F x) = F (gmap_RF fs x)
class GMap_RF (υ :: Field) (τ :: [⋆]) where
  gmap_RF :: HList τ → InField υ (Doms τ) → InField υ (Codoms τ)
```

Constants, however, just like units, are returned unchanged:

```
instance GMap_RF (K α) τ where
  gmap_RF _ (K x) = K x
```

For a parameter $P\ \nu$, we need to lookup the right function to map over. We again use a separate type class, $GMap_{RP}$, and we traverse the input list of functions until we reach the ν-th function. We thus require the list of functions τ to have its elements in the same order as the datatype parameters:

```
instance (GMap_RP ν τ) ⇒ GMap_RF (P ν) τ where
  gmap_RF fs (P x) = P (gmap_RP (Proxy :: Proxy ν) fs x)

class GMap_RP (ν :: Nat) (τ :: [⋆]) where
  gmap_RP :: Proxy ν → HList τ → (Doms τ) :!: ν → (Codoms τ) :!: ν
instance GMap_RP Ze ((α → β) ‘: τ) where
  gmap_RP _ (HCons f _) x = f x
instance (GMap_RP ν τ) ⇒ GMap_RP (Su ν) ((α → β) ‘: τ) where
  gmap_RP _ (HCons _ fs) p = gmap_RP (Proxy :: Proxy ν) fs p
```

Handling application The only representation type we still have to deal with is $(:@:)$. This is also the most challenging case. As a running example, consider the second field of the D datatype of Section 3.2. D has three parameters, α, β, and γ, and the second field of its only constructor has type $[(\alpha, Int)]$, represented as $[]\ {:@:}\ `[(,)\ {:@:}\ `[P\ 0, K\ Int]]$. In this situation, we intend to transform $[(\alpha, Int)]$ into $[(\alpha', Int)]$, having available a function of type $\alpha \to \alpha'$. We do this by requiring the availability of *gmap* for this particular list type; that is, we require an instance $GMap\ []\ `[(\alpha, Int) \to (\alpha', Int)]$. Having such an instance, we can simply *gmap* over the argument. However, the functions we map need to be adapted to this new argument. That is the task of *AdaptFs*, which we explain below.

A type function *MakeFs* computes the types of the functions to be mapped inside the applied type. For example, $MakeFs\ `[(,)\ {:@:}\ `[P\ 0, K\ Int]]\ `[\alpha \to \alpha', \beta \to \beta', \gamma \to \gamma']$ is $`[(\alpha, Int) \to (\alpha', Int)]$. We use proxies to fix otherwise ambiguous types when invoking $gmap_P$ and *adaptFs*:

```
MakeFs (ρ :: [Field]) (τ :: [⋆]) :: [⋆]
MakeFs ‘[]                τ = ‘[]
MakeFs ((K α) ‘: ρ)       τ = (α → α) ‘: MakeFs ρ τ
MakeFs ((P ν) ‘: ρ)       τ = (τ :!: ν) ‘: MakeFs ρ τ
MakeFs ((σ :@: ω) ‘: ρ) τ =
  (AppFields σ ω (Doms τ) → AppFields σ ω (Codoms τ)) ‘: MakeFs ρ τ
instance (GMap σ (MakeFs ρ τ)
          ,AdaptFs ρ τ
          ,ExpandField ρ (Doms   τ) ∼ Doms   (MakeFs ρ τ)
          ,ExpandField ρ (Codoms τ) ∼ Codoms (MakeFs ρ τ)
          ) ⇒ GMap_RF (σ :@: ρ) τ where
  gmap_RF fs (A x) = A (gmap_P p_1 (adaptFs p_2 fs) x)
```

```
    where p1 = Proxy :: Proxy σ
          p2 = Proxy :: Proxy ρ
```

This instance has four constraints, two of them being equality constraints, introduced by the $\sim$ operator: a constraint of the form $\alpha \sim \beta$ indicates that the type α must be equal to the type β. The four constraints of this instance are: the ability to map over the argument type, the requirement to rearrange the functions we map over, and two coherence conditions on the behaviour of *ExpandField* and *MakeFs* over the list of functions. The latter are always true for valid ρ and τ.

We are left with the task of adapting the functions to be mapped over. In our running example, we have an *HList* '$[\alpha \to \alpha', \beta \to \beta', \gamma \to \gamma']$, and we have to produce an *HList* '$[(\alpha, Int) \to (\alpha', Int)]$. This is done by *adaptFs*, a method of the type class *AdaptFs* which is implemented by induction on the list of fields to be mapped over:

```
class AdaptFs (ρ :: [Field]) (τ :: [⋆]) where
  adaptFs :: Proxy ρ → HList τ → HList (MakeFs ρ τ)
```

For an empty list of an arguments, we return an empty list of functions. If the argument is a constant, we ignore it and proceed to the next argument. For a parameter $P\ \nu$, we use the ν-th function of the original list of functions, and proceed to the next argument:

```
instance AdaptFs '[] τ where
  adaptFs _ _ = HNil
instance (AdaptFs ρ τ) ⇒ AdaptFs ((K α) ': ρ) τ where
  adaptFs _ fs = HCons id (adaptFs (Proxy :: Proxy ρ) fs)
instance (AdaptFs ρ τ, Lookup τ ν) ⇒ AdaptFs ((P ν) ': ρ) τ where
  adaptFs _ fs = HCons (lookup p1 fs) (adaptFs p2 fs)
    where p1 = Proxy :: Proxy ν
          p2 = Proxy :: Proxy ρ
```

The most delicate case is, again, application. Back to our running example, this is where we have to produce a function of type $(\alpha, Int) \to (\alpha', Int)$. We do so by requiring an instance *GMap* $(,)$ '$[\alpha \to \alpha', Int \to Int]$, reusing *MakeFs* and *AdaptFs* in the process. We also need to proceed recursively for the rest of the arguments. Again, we have the same two coherence conditions on the behaviour of *ExpandField* and *MakeFs* over the list of functions, and use proxies to fix ambiguous types:

```
instance (GMap σ (MakeFs ω τ)
         ,AdaptFs ω τ
         ,AdaptFs ρ τ
         ,ExpandField ω (Doms   τ) ∼ Doms   (MakeFs ω τ)
         ,ExpandField ω (Codoms τ) ∼ Codoms (MakeFs ω τ)
         ) ⇒ AdaptFs ((σ :@: ω) ': ρ) τ where
  adaptFs _ fs = HCons (gmapP p1 (adaptFs p2 fs)) (adaptFs p3 fs)
    where p1 = Proxy :: Proxy σ
          p2 = Proxy :: Proxy ω
          p3 = Proxy :: Proxy ρ
```

The generic definition of the generalised map is thus complete, and ready to be used.

3.4 Example usage

Before we instantiate map to our example datatypes of Section 3.2, we first provide a generic default [11] to make instantiation simpler. This default, for the $GMap$ class, will allow us to then give empty instances of the class, which will automatically use the generic definition for the generalised map. The default version of $gmap_P$ converts a value into its generic representation, applies $gmap_R$, and then converts back to a user datatype. This requires several constraints (which would all be inferred if the function was defined at the top level). First, we introduce α and β as synonyms for the input and output type, respectively, for mere convenience. In the case of the $GMap\ [\,]\ `[\gamma \to \gamma']$ instance, for example, α is $[\gamma]$, and β is $[\gamma']$. We also require a $Generic\ [\gamma]$ instance (for *from*), and a $Generic\ [\gamma']$ instance (for *to*); these instances will coincide, and indeed we also require that the representation $Rep\ [\gamma]$ be the same as $Rep\ [\gamma']$ (which is the case). Furthermore, the parameters of $[\gamma]$ have to coincide with the domains of $`[\gamma \to \gamma']$, and the parameters of $[\gamma']$ have to coincide with the codomains of the same list. Finally, we require the ability to map generically over the representation of the input list:

$$
\begin{array}{ll}
\textbf{default}\ gmap_P :: & (\alpha \sim (\sigma\ {:}\$ {:}\ Doms\ \tau) \\
 & ,\beta \sim (\sigma\ {:}\$ {:}\ Codoms\ \tau) \\
 & ,Generic\ \alpha, Generic\ \beta \\
 & ,Rep\ \ \alpha \sim Rep\ \beta \\
 & ,Pars\ \alpha \sim Doms\ \tau \\
 & ,Pars\ \beta \sim Codoms\ \tau \\
 & ,GMap_R\ (Rep\ \alpha)\ \tau \\
 &) \Rightarrow Proxy\ \sigma \to HList\ \tau \to \alpha \to \beta \\
\multicolumn{2}{l}{gmap_P\ \_ fs = to \circ gmap_R\ fs \circ from}
\end{array}
$$

With this default in place, we are ready to instantiate our example datatypes:

$$
\begin{array}{ll}
\textbf{instance}\ GMap\ [\,] & `[\alpha \to \alpha'] \\
\textbf{instance}\ GMap\ RTree & `[\alpha \to \alpha'] \\
\textbf{instance}\ GMap\ (,) & `[\alpha \to \alpha', \beta \to \beta'] \\
\textbf{instance}\ GMap\ D & `[\alpha \to \alpha', \beta \to \beta', \gamma \to \gamma'] \\
\textbf{instance}\ GMap\ Perfect & `[\alpha \to \alpha']
\end{array}
$$

Using the generic default, instantiation is kept simple and concise. We can verify that our map works as expected in an example that should cover all the representation types:

$$
\begin{array}{l}
x :: D\ Int\ Float\ Char \\
x = D\ 0.2\ [(0,0),(1,1)]\ (RTree\ \texttt{"p"}\ [\,])
\end{array}
$$

$$
\begin{array}{l}
y :: D\ Int\ String\ Char \\
y = gmap\ (HCons\ (+1)\ (HCons\ show\ (HCons\ (const\ \texttt{'q'})\ HNil)))\ x
\end{array}
$$

Indeed, y evaluates to $D\ \texttt{"0.2"}\ [(1,0),(2,1)]\ (RTree\ \texttt{"q"}\ [\,])$ as expected.

4 Limitations and future work

In this section we discuss the limitations of our new `generic-deriving`, and propose future research directions.

4.1 Parameters of higher kinds

While our generalisation supports any number of parameters of kind $\star$, it is unable to deal with parameters of higher kinds. Consider the following two datatypes representing generalised trees:

$$\begin{array}{l}\textbf{data}\ \mathit{GTree_1}\ \phi\ \alpha = \mathit{GTree_1}\ \alpha\ (\phi\ (\mathit{GTree_1}\ \phi\ \alpha)) \\ \textbf{data}\ \mathit{GTree_2}\ \alpha\ \phi\ = \mathit{GTree_2}\ \alpha\ (\phi\ (\mathit{GTree_2}\ \alpha\ \phi))\end{array}$$

The most general mapping function for $\mathit{GTree_1}$ has the following type:

$$(\alpha \to \beta) \to (\forall \alpha\ \beta.(\alpha \to \beta) \to \phi\ \alpha \to \psi\ \beta) \to \mathit{GTree_1}\ \phi\ \alpha \to \mathit{GTree_1}\ \psi\ \beta$$

The generalisation of this paper provides a map of type $(\alpha \to \beta) \to \mathit{GTree_1}\ \phi\ \alpha \to \mathit{GTree_1}\ \phi\ \beta$, therefore ignoring the ϕ parameter of kind $\star \to \star$. For $\mathit{GTree_2}$, however, our approach cannot even provide that simple map; we cannot give the GMap instance, since the parameters of kind $\star$ of interest come before a parameter of kind $\star \to \star$, and the second parameter of GMap is a list of kind $[\star]$ (so all arguments must have kind $\star$).

As such, we support generic abstraction only over the parameters of kind $\star$ which come after any parameters of other kinds. Lifting this restriction is not trivial. Recall that the Pars type family has return kind $[\star]$. To generalise to multiple parameters, we should make this return kind be a (promoted) heterogeneous list. This is currently not possible, as heterogeneous lists are implemented as GADTs, which cannot be promoted. Foregoing giving Pars the correct kind and working with nested tuples instead gives rise to many kind ambiguities, which are cumbersome to solve. As such, we hope that the promotion mechanism is extended to allow promotion of GADTs soon [15], and defer generalising our approach to parameters of arbitrary kinds until then.

4.2 Integration with existing generic programming libraries

The introduction of our new `generic-deriving` raises the question of how to upgrade from the old version. Our changes are not backwards compatible, but since the new version encodes strictly more information than the previous one, we can provide a conversion that automatically transforms the new representation into the old one, therefore remaining compatible with old code [10]. The core of this conversion is a type-level function $\mathit{D_{n\to}D}$ that converts the new representation into the old one. We show a prototype implementation here, subscripting the new `generic-deriving` codes with an n to distinguish them from the old ones. Units, sums, and products are converted trivially. Fields are handled by a separate function $\mathit{D_{n\to}D_F}$. Constants are converted trivially. For a parameter, we return P if it is the last parameter of the datatype (the only one supported by the old version of `generic-deriving`), or a constant otherwise. We make use of type-level if-then-else, equality on Nat, and length. Applications of types of kind $\star \to \star$ are converted into compositions. For applications of types of higher arity, we first apply the type to all its arguments but the last:

$$\begin{array}{l}\mathit{IfThenElse}\ (\alpha :: \mathit{Bool})\ (\beta :: \kappa)\ (\gamma :: \kappa) :: \kappa \\ \mathit{IfThenElse\ True}\ \ \beta\ \gamma = \beta \\ \mathit{IfThenElse\ False}\ \beta\ \gamma = \gamma\end{array}$$

$$
\begin{array}{l}
(\alpha :: \mathit{Nat}) \equiv_{\mathit{Nat}} (\beta :: \mathit{Nat}) :: \mathit{Bool} \\
\mathit{Ze} \equiv_{\mathit{Nat}} \mathit{Ze} = \mathit{True} \\
\mathit{Su}\ \upsilon \equiv_{\mathit{Nat}} \mathit{Ze} = \mathit{False} \\
\mathit{Ze} \equiv_{\mathit{Nat}} \mathit{Su}\ \nu = \mathit{False} \\
\mathit{Su}\ \upsilon \equiv_{\mathit{Nat}} \mathit{Su}\ \nu = \upsilon \equiv_{\mathit{Nat}} \nu
\end{array}
$$

$$
\begin{array}{l}
\mathit{Length}\ (\rho :: [\,\kappa\,]) :: \mathit{Nat} \\
\mathit{Length}\ `[\,] = \mathit{Ze} \\
\mathit{Length}\ (\alpha\ `{:}\ \rho) = \mathit{Su}\ (\mathit{Length}\ \rho)
\end{array}
$$

$$
\begin{array}{l}
D_{n\to}D\ (\upsilon :: \mathit{Univ}_n)\ (\rho :: [\star]) :: \mathit{Univ} \\
D_{n\to}D\ U_n\ \rho = U \\
D_{n\to}D\ (\alpha :+_n: \beta)\ \rho = D_{n\to}D\ \alpha\ \rho :+: D_{n\to}D\ \beta\ \rho \\
D_{n\to}D\ (\alpha :\times_n: \beta)\ \rho = D_{n\to}D\ \alpha\ \rho :\times: D_{n\to}D\ \beta\ \rho \\
D_{n\to}D\ (F_n\ \alpha)\ \rho = D_{n\to}D_F\ \alpha\ \rho
\end{array}
$$

$$
\begin{array}{ll}
D_{n\to}D_F\ (\upsilon :: \mathit{Field}_n)\ (\rho :: [\star]) :: \mathit{Univ} & \\
D_{n\to}D_F\ (K_n\ \alpha) & \rho = K\ \alpha \\
D_{n\to}D_F\ (P_n\ \nu) & \rho = \mathit{IfThenElse}\ (\nu \equiv_{\mathit{Nat}} \mathit{Length}\ \rho)\ P\ (K\ (\rho :!: \nu)) \\
D_{n\to}D_F\ (\phi :@_n: `[\alpha]) & \rho = \phi :@: D_{n\to}D_F\ \alpha\ \rho \\
D_{n\to}D_F\ (\phi :@_n: (\alpha\ `{:}\ \beta\ `{:}\ \gamma)) & \rho = D_{n\to}D_F\ ((\mathit{AppFields}\ \phi\ `[\alpha]\ \rho) :@_n: (\beta\ `{:}\ \gamma))\ \rho
\end{array}
$$

The introduction of yet another generic programming library underscores the need for a single mechanism for type reflection baked into the compiler, from which other mechanisms, such as our new `generic-deriving`, or *Data* and *Typeable* [5], could then be derived.

4.3 Parameter genericity vs. arity genericity

The approach described in this paper allows us to define generic functions that operate over multiple datatype parameters. This is not the same as generic functions that operate at diverse arities [14]. Consider the following generic functions:

$$
\begin{array}{lll}
\mathit{map}^1_n & :: & \mathit{HList}\ `[\alpha^1_1, \ldots, \alpha^1_n] \to \phi\ (\alpha^1_1 \ldots \alpha^1_n) \\
\mathit{map}^2_n & :: & \mathit{HList}\ `[\alpha^1_1 \to \alpha^2_1, \ldots, \alpha^1_n \to \alpha^2_n] \to \phi\ (\alpha^1_1 \ldots \alpha^1_n) \to \phi\ (\alpha^2_1 \ldots \alpha^2_n) \\
\mathit{map}^3_n & :: & \mathit{HList}\ `[\alpha^1_1 \to \alpha^2_1 \to \alpha^3_1, \ldots, \alpha^1_n \to \alpha^2_n \to \alpha^3_n] \\
 & & \to \phi\ (\alpha^1_1 \ldots \alpha^1_n) \to \phi\ (\alpha^2_1 \ldots \alpha^2_n) \to \phi\ (\alpha^3_1 \ldots \alpha^3_n) \\
\mathit{map}^m_n & :: & \mathit{HList}\ `[\alpha^1_1 \to \ldots \to \alpha^m_1, \ldots, \alpha^1_n \to \ldots \to \alpha^m_n] \\
 & & \to \phi\ (\alpha^1_1 \ldots \alpha^1_n) \to \ldots \to \phi\ (\alpha^m_1 \ldots \alpha^m_n)
\end{array}
$$

The function map^1_n, or *repeat*, creates a ϕ-structure given elements for the parameter positions. The function map^2_n, equivalent to the generic *gmap* of Section 3.3, is the function that maps over a ϕ-structure, taking one function per parameter. The function map^3_n, or *zipWith*, is the function that takes two ϕ-structures and zips them when their shapes are compatible, using the provided functions to zip the parameters. The function map^m_n is the generalisation of the previous three, in the arity-generic sense. Our approach allows defining each of map^1_n, map^2_n, map^3_n, etc., individually, as separate generic functions. It does not allow defining map^m_n; that generalisation is described by Weirich and Casinghino [14], in the dependently-typed programming language Agda. It remains to see how to transfer the concept of arity-genericity to Haskell.

5 Conclusion

In this paper we have seen how we can use the promotion mechanism together with kind polymorphism to encode a generic representation of datatypes that supports abstraction over multiple parameters (of kind $\star$). This enables a whole new class of generic functionality: we have shown a generalised map, but also folding, traversing, and zipping, for example, are now possible. We plan to implement support for the new `generic-deriving` in GHC soon, so that users can take advantage of the new functionality without needing to write their own *Generic* instances.

Acknowledgements This work was supported by the EP/J010995/1 EPSRC grant. Jeremy Gibbons and anonymous referees provided valuable feedback on an early draft of this paper.

Bibliography

[1] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In *The 13th International Workshop on the Implementation of Functional Languages*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–185. Springer, 2001. doi:10.1007/3-540-46028-4_11.

[2] 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.

[3] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 72–149. Springer-Verlag, 2007. doi:10.1007/978-3-540-76786-2_2.

[4] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107. ACM, 2004. doi:10.1145/1017472.1017488.

[5] 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.

[6] Andres Löh. *Exploring Generic Haskell.* PhD thesis, Universiteit Utrecht, 2004. `http://igitur-archive.library.uu.nl/dissertations/2004-1130-111344`.

[7] Andres Löh and José Pedro Magalhães. Generic programming with indexed functors. In *Proceedings of the 7th ACM SIGPLAN Workshop on Generic Programming*, pages 1–12. ACM, 2011. doi:10.1145/2036918.2036920.

[8] José Pedro Magalhães. *Less Is More: Generic Programming Theory and Practice*. PhD thesis, Universiteit Utrecht, 2012.

[9] José Pedro Magalhães. The right kind of generic programming. In *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming*, WGP '12, pages 13–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1576-0. doi:10.1145/2364394.2364397.

[10] José Pedro Magalhães and Andres Löh. Generic generic programming, 2014. URL `http://dreixel.net/research/pdf/ggp.pdf`. Accepted for publication at Practical Aspects of Declarative Languages (PADL'14).

[11] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Proceedings of the 3rd ACM Haskell Symposium on Haskell*, pages 37–48. ACM, 2010. doi:10.1145/1863523.1863529.

[12] Simon Peyton Jones, editor. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. doi:10.1017/S0956796803000315. Journal of Functional Programming Special Issue 13(1).

[13] Alexey Rodriguez Yakushev, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C.d.S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 111–122. ACM, 2008. doi:10.1145/1411286.1411301.

[14] Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages meets Program Verification*, pages 15–26. ACM, 2010. doi:10.1145/1707790.1707799.

[15] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 275–286. ACM, 2013. doi:10.1145/2500365.2500599.

[16] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012. doi:10.1145/2103786.2103795.