Hierarchy in Generic Programming Libraries

José Pedro Magalhães[1] and Andres Löh[2]

[1] jpm@cs.ox.ac.uk. Department of Computer Science, University of Oxford
[2] andres@well-typed.com. Well-Typed LLP

Abstract. Generic programming (GP) is a form of abstraction in programming languages that serves to reduce code duplication by exploiting the regular structure of algebraic datatypes. Several different approaches to GP in Haskell have surfaced, giving rise to the problem of code duplication across GP libraries. Given the original goals of GP, this is a rather unfortunate turn of events. Fortunately, we can convert between the different representations of each approach, which allows us to "borrow" generic functions from different approaches, avoiding the need to reimplement every generic function in every single GP library.
In previous work we have shown how existing GP libraries relate to each other. In this paper we go one step further and advocate "hierarchical GP": through proper design of different GP approaches, each library can fit neatly in a hierarchy, greatly minimizing the amount of supporting infrastructure necessary for each approach, and allowing each library to be specific and concise, while eliminating code duplication overall. We introduce a new library for GP in Haskell intended to sit at the top of the "GP hierarchy". This library contains a lot of structural information, and is not intended to be used directly. Instead, it is a good starting point for generating generic representations for other libraries. This approach is also suitable for being the only library with native compiler support; all other approaches can be obtained from this one by simple conversion of representations in plain Haskell code.

1 Introduction

Generic programs are concise, abstract, and reusable. They allow one single definition to be used for many kinds of data, existing and to come. For example, parsing and pretty-printing, (de-)serialisation, test data generation, and traversals can all be implemented generically, freeing the programmer to implement only datatype-specific functionality.

Given its power, it's no surprise that GP approaches abound: including pre-processors, template-based approaches, language extensions, and libraries, there are well over 15 different approaches to GP in Haskell (Magalhães 2012, Chapter 8). This abundance is partly caused by the lack of a clearly superior approach; each approach has its strengths and weaknesses, uses different implementation mechanisms, a different generic view (Holdermans et al. 2006) (i.e. a different representation of datatypes), or focuses on solving a particular task. Their number and variety makes comparisons difficult, and can make prospective GP users struggle even before actually writing a generic program, since first they have to choose a library that is appropriate for their needs.

We have previously investigated how to model and formally relate some Haskell GP libraries using Agda (Magalhães and Löh 2012), and concluded that some approaches clearly subsume others. Afterwards, we have shown how to reduce code duplication in GP libraries in Haskell by converting between the representations of different approaches, in what we dubbed "generic generic programming" (Magalhães and Löh 2014).

To help understand the benefits of our work, it is important to distinguish three kinds of users of GP:

Compiler writer As far as GP goes, the compiler writer is concerned with which approach(es) are natively supported by the compiler. At the moment, in the main Haskell compiler GHC, both `syb` (Lämmel and Peyton Jones 2003, 2004) and `generic-deriving` (Magalhães et al. 2010) are natively supported. This means that the compiler can automatically generate the necessary generic representations to enable using these approaches. A quick analysis reveals that in GHC there are about 226 lines of code for supporting `syb`, and 884 for `generic-deriving`.

GP library author The library author maintains one or more GP libraries, and possibly creates new ones. Since most approaches are not natively supported, the library author has to deal with generating generic representations for their library (or accept that no end user will use this library, given the amount of boilerplate code they would have to write). Typically, the library author will rely on Template Haskell (TH, Sheard and Peyton Jones 2002) for this task. Given that TH handles code generation at the AST level (thus syntactic), this is not a pleasant task. Furthermore, the TH API changes as frequently as the compiler, so the library author currently has to update its supporting code frequently.

End users The end users know nothing about compiler support, and ideally not even about library-specific detail. They simply want to use a particular generic function on their data, with minimum overhead.

While Magalhães and Löh (2014) focused mostly on improving the life of the end user, the work we describe in this paper brings more advantages to the compiler writer and GP library author. We elaborate further on the idea of generic generic programming by highlighting the importance of hierarchy in GP approaches. Each new GP library should only be a piece of the puzzle, specialising in one task, or exploring a new generic representation, but obtaining most of its supporting infrastructure (such as example generic functions) from already existing approaches. To facilitate this, we introduce a new GP library, `structured`, which we use as a core to derive representations for other GP libraries. Defining a new library does not mean introducing a lot of new supporting code. In fact, we do not even think many generic functions will ever be defined in our new library, as its representation is verbose (albeit precise). Instead, we use it to guide conversion efforts, as a highly structured approach provides a good foundation to build upon.

From the compiler writer's perspective, this library would be the only one needing compiler support; support for other libraries follows automatically from conversions that are defined in plain Haskell, not through more compiler extensions. Since `structured` has only one representation, as opposed to `generic-deriving`'s two representations, we believe that supporting it in GHC would require fewer lines of code than the

existing support for `generic-deriving`. The code for supporting `generic-deriving` and `syb` could then be removed, as those representations can be obtained from `structured`.

Should we ever find that we need more information in `structured` to support converting to other libraries, we can extend it without changing any of the other libraries. This obviates the need to change the compiler for supporting or adapting GP approaches. It also simplifies the life of the GP library author, who no longer needs to rely on meta-programming tools.

Specifically, our contributions are the following:

- A new library for GP, `structured`, which properly encodes the nesting of the different structures within a datatype representation (Section 2). We propose this libary as a foundation for GP in Haskell, from which many other approaches can be derived. It is designed to be highly expressive and easily extensible, serving as a back-end for more stable and established GP libraries.
- We show how `structured` can provide GP library authors with different views of the nesting of constructors and fields (Section 3). Different generic functions prefer different balancings, which we provide through automatic conversion (instead of duplicated encodings differing only in the balancing).
- We position `structured` at the top of GP hierarchy by showing how to derive `generic-deriving` (Magalhães et al. 2010) representations from it (Section 4). This also shows how `structured` unifies the two generic representations of `generic-deriving`. Representations for other libraries (`regular` (Van Noort et al. 2008), `multirec` (Rodriguez Yakushev et al. 2009), and `syb` (Lämmel and Peyton Jones 2003, 2004)) can then be obtained from `generic-deriving`.

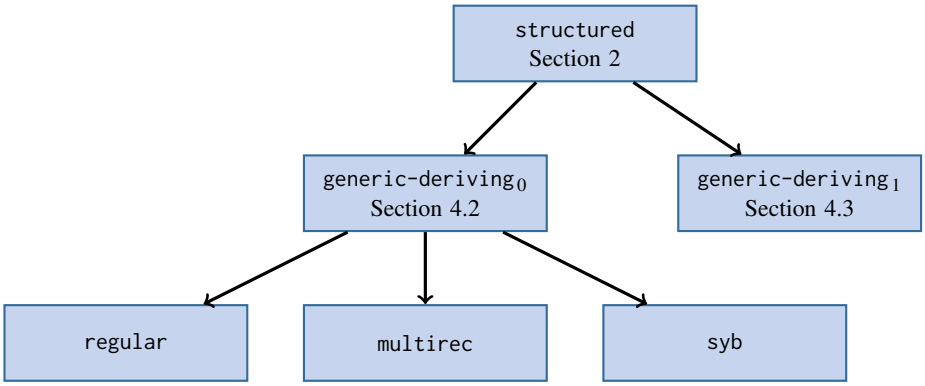

Fig. 1. Hierarchical relationship between GP approaches.

Figure 1 shows an overview of the hierarchical relationships between different libraries for GP in Haskell. In this paper, we introduce `structured` and its conversion to `generic-deriving`. We refer the reader to Magalhães and Löh (2014) for the conversions from `generic-deriving`.

1.1 Notation

In order to avoid syntactic clutter and to help the reader, we adopt a liberal Haskell notation in this paper. We assume the existence of a **kind** keyword, which allows us to define kinds directly. These kinds behave as if they had arisen from datatype promotion (Yorgey et al. 2012), except that they do not define a datatype and constructors. 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*. In case the colours cannot be seen, the "level" of an expression is clear from the context. Additionally, we use Greek letters for type variables, apart from κ, which is reserved for kind variables.

This syntactic sugar is only for presentation purposes. An executable version of the code, which compiles with GHC 7.8.3, is available at `http://dreixel.net/research/code/hgp.zip`. We rely on many GHC-specific extensions to Haskell, which are essential for our development. Due to space constraints we cannot explain them all in detail, but we try to point out relevant features as we use them.

1.2 Structure of the paper

The remainder of this paper is structured as follows. We first introduce the `structured` library (Section 2). We then see how to obtain views with different balancings of the constructors and constructor arguments (Section 3). Afterwards, we see how to obtain `generic-deriving` from `structured` (Section 4). We conclude with a discussion in Section 5.

2 A highly structured GP library

Our efforts of modularising a hierarchy of GP libraries stem from a `structured` library intended to sit at the top of the hierarchy. Our goal is to define a library that is highly expressive, without having to worry about convenience of use. Users requiring the level of detail given by `structured` can use it directly, but we expect most to prefer using any of the other, already existing GP libraries. Usability is not our concern here; expressiveness is. Stability is also not guaranteed; we might extend our library as needed to support converting to more approaches. Previous approaches had to find a careful balance between having too little information in the generic representation, resulting in a library with poor expressiveness, and having too much information, resulting in a verbose and hard to use approach. Given our modular approach, we are free from these concerns.

The design of `structured` we give here is preliminary; we plan to extend it in the future in order to support representing more datatypes. In fact, as the type language of GHC grows with new extensions, we expect to keep changing `structured` frequently. However, the simple fact that we introduce `structured`, and show how to use it for decoupling `generic-deriving` from the compiler, improves the current status quo. In particular, if `structured` is supported through automatic deriving in GHC, no more compiler support is required for the other libraries. Using this library also improves

modularity; it can be updated or extended more freely, since supporting the other libraries requires only updating the conversions, not the compiler itself (for the automatic derivation of instances).

In our previous work (Magalhães and Löh 2014) we have shown how to obtain the representation of many libraries from `generic-deriving`. Given that `structured`, at the time of writing, serves only to provide a conversion to `generic-deriving`, the reader might think that it is unnecessary. We have several reasons justifying `structured`, however:

- The `generic-deriving` library has been around for some time now, and lots of code using it has been written. Sticking to `generic-deriving` as a foundational approach would force us to break lots of code whenever we would need to update it in order to support new approaches, or to add functionality.
- By introducing `structured`, we can decouple most of `generic-deriving` from the compiler. In particular, the mechanism for deriving `generic-deriving` instances can be simplified, because `generic-deriving` has two representations (which need to be derived separately), while `structured` has only one (from which we can derive both `generic-deriving` representations).
- Being a new approach designed to be an internal representation, `structured` can be changed without worrying too much about breaking existing code; the only code that would need to be adapted is that for the conversion to `generic-deriving`. This is plain Haskell code, not compiler code or TH, so it's easier to update and maintain.

We now proceed to describe the representation types in `structured`, their interpretation as values, and the conversion between user datatypes and their generic representations, together with example encodings.

2.1 Universe

The structure used to encode datatypes in a GP approach is called its *universe* (Morris 2007). The universe of `structured`, for now, is similar to that of `generic-deriving` (Magalhães 2012, Chapter 11), as it supports abstraction over at most one datatype parameter. We choose to restrict this parameter to be the last of the datatype, and only if its kind is $\star$. This is a pragmatic decision: many generic functions, such as *map*, require abstraction over one parameter, but comparatively few require abstraction over more than one parameter. For example, in the type $[\alpha]$, the parameter is α, and in $\mathit{Either}\ \alpha\ \beta$, it is β. The differences to `generic-deriving` lay in the explicit hierarchy of data, constructor, and field, and the absence of two separate ways of encoding constructor arguments. It might seem unsatisfactory that we do not improve on the limitations of `generic-deriving` with regards to datatype parameters, but that is secondary to our goal in this paper (and it would be easy to implement support for multiple parameters in `structured` following the strategy of Magalhães (2014)). Furthermore, `structured` can easily be improved later, keeping the other libraries unchanged, and adapting only the conversions if necessary.

Datatypes are represented as types of *kind* Data. We define new kinds, whose types are not inhabited by values: only types of kind $\star$ are inhabited by values. These kinds

can be thought of as datatypes, but their "constructors" will be used as indices of a GADT (Schrijvers et al. 2009) to construct values with a specific structure.

Datatypes have some metadata, such as their name, and contain constructors. Constructors have their own metadata, and contain fields. Finally, each field can have metadata, and contain a value of some structure:

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

We use a binary leaf tree to encode the structure of the constructors in a datatype, and the fields in a constructor. Typically lists are used, but we will see in Section 3 that it is convenient to encode the structure as a tree, as we can change the way it is balanced for good effect.

The metadata we store is unsurprising:

```
kind MetaData = MD Symbol          -- datatype name
                   Symbol          -- datatype module name
                   Bool            -- is it a newtype?
kind MetaCon  = MC Symbol          -- constructor name
                   Fixity          -- constructor fixity
                   Bool            -- does it use record syntax?
kind MetaField = MF (Maybe Symbol) -- field name

kind Fixity = Prefix | Infix Associativity Nat
kind Associativity = LeftAssociative | RightAssociative | NotAssociative
kind Nat = Ze | Su Nat
kind Symbol   -- internal
```

It is important to note that this metadata is encoded at the type level. In particular, we have type-level strings and natural numbers. We make use of the current (in GHC 7.8.3) implementation of type-level strings, whose kind is *Symbol*.

Finally, *Arg* describes the structure of constructor arguments:

```
kind Arg = K KType ⋆
         | Rec RecType (⋆ → ⋆)
         | Par
         | (⋆ → ⋆) :○: Arg
kind KType   = P | R RecType | U
kind RecType = S | O
```

A field can either be a datatype parameter other than the last (*K P*), an occurrence of a different datatype of kind ⋆ (*K* (*R O*)), some other type (such as an application of a type variable, encoded with *K U*), a datatype of kind (at least) ⋆ → ⋆ (*Rec*), which can be

either the same type we're encoding (S) or a different one (O), the (last) parameter of the datatype (Par), or a composition of a type constructor with another argument ($:\circ:$).

The representation is best understood in terms of an example. Consider the following datatype:

$$\textbf{data}\ D\ \phi\ \alpha\ \beta = D_1\ Int\ (\phi\ \alpha)\ |\ D_2\ [D\ \phi\ \alpha\ \beta]\ \beta$$

We first show the encoding of each of the four constructor arguments: Int is a datatype of kind $\star$, so it's encoded with $K\ (R\ O)\ Int$; $\phi\ \alpha$ depends on the instantiation of ϕ, so it's encoded with $K\ U\ (\phi\ \alpha)$; $[D\ \phi\ \alpha\ \beta]$ is a composition between the list functor and the datatype we're defining, so it's encoded with $[]\ :\circ:\ Rec\ S\ (D\ \phi\ \alpha)$; finally, β is the parameter we abstract over, so it's encoded with Par:

$$\begin{aligned}
A_{11} &= K\ (R\ O)\ Int\\
A_{12} &= K\ U\ (\phi\ \alpha)\\
A_{21} &= []\ :\circ:\ Rec\ S\ (D\ \phi\ \alpha)\\
A_{22} &= Par
\end{aligned}$$

The entire representation consists of wrapping of appropriate meta-data around the representation for constructor arguments:

$$\begin{aligned}
&Rep_D\ \phi\ \alpha\ \beta =\\
&\quad Data\ (MD\ \texttt{"D"}\ \texttt{"Module"}\ False)\\
&\qquad (Bin\ (Leaf\ (Con\ (MC\ \texttt{"D1"}\ Prefix\ False)\\
&\qquad\qquad\qquad (Bin\ (Leaf\ (Field\ (MF\ Nothing)\ A_{11}))\\
&\qquad\qquad\qquad\qquad (Leaf\ (Field\ (MF\ Nothing)\ A_{12})))))\\
&\qquad\qquad (Leaf\ (Con\ (MC\ \texttt{"D2"}\ Prefix\ False)\\
&\qquad\qquad\qquad (Bin\ (Leaf\ (Field\ (MF\ Nothing)\ A_{21}))\\
&\qquad\qquad\qquad\qquad (Leaf\ (Field\ (MF\ Nothing)\ A_{22}))))))
\end{aligned}$$

2.2 Interpretation

The interpretation of the universe defines the structure of the values that inhabit the datatype representation. Datatype representations will be types of kind $Data$. We use a data family (Schrijvers et al. 2008) $[\![\_]\!]$ to encode the interpretation of the universe of `structured`:

$$\textbf{data family}\ [\![\_]\!] :: \kappa \to \star \to \star$$

Its first argument is written infix, and the second postfix. Its kind, $\kappa \to \star \to \star$, is overly general in κ; we will only instantiate κ to the types of the universe shown before, and prevent further instantiation by not exporting the family $[\![\_]\!]$ (effectively making it a closed data family). The second argument of $[\![\_]\!]$, of kind $\star$, is the parameter of the datatype which we abstract over.

The top-level inhabitant of a datatype representation is a constructor D_1, which serves only as a proxy to store the datatype metadata in its type:

```
data instance [[ υ :: Data ]] ρ where
  D1 :: [[ α ]] ρ → [[ Data ι α ]] ρ
```

Constructors, on the other hand, are part of a *Tree* structure, so they can be on the left (L_1) or right (R_1) side of a branch, or be a leaf. As a leaf, they contain the meta-information for the constructor that follows (C_1):

```
data instance [[ υ :: Tree Con ]] ρ where
  C1 :: [[ α ]] ρ → [[ Leaf (Con ι α) ]] ρ
  L1 :: [[ α ]] ρ → [[ Bin α β ]] ρ
  R1 :: [[ β ]] ρ → [[ Bin α β ]] ρ
```

Constructor fields are similar, except that they might be empty (U_1, as some constructors have no arguments), leaves contain fields (S_1), and branches are inhabited by the arguments of both sides ($:\times:$):

```
data instance [[ υ :: Tree Field ]] ρ where
  U1    ::                          [[ Empty ]] ρ
  S1    :: [[ α ]] ρ                → [[ Leaf (Field ι α) ]] ρ
  (:×:) :: [[ α ]] ρ → [[ β ]] ρ → [[ Bin α β ]] ρ
```

We're left with constructor arguments. We encode base types with K, datatype occurrences with *Rec*, the parameter with *Par*, and composition with *Comp*:

```
data instance [[ υ :: Arg ]] ρ where
  K    :: { unK1   :: α   }          → [[ K ι α ]]   ρ
  Rec  :: { unRec  :: ϕ ρ }          → [[ Rec ι ϕ ]] ρ
  Par  :: { unPar  :: ρ   }          → [[ Par ]]     ρ
  Comp :: { unComp :: σ ([[ ϕ ]] ρ) } → [[ σ :○: ϕ ]] ρ
```

2.3 Conversion to and from user datatypes

Having seen the generic universe and its interpretation, we need to provide a mechanism to mediate between user datatypes and our generic representation. We use a type class for this purpose:

```
class Generic (α :: ⋆) where
  Rep     α :: Data
  ThePar α :: ⋆
  ThePar α = NoPar
  from    :: α → [[ Rep ϕ ]] (ThePar α)
  to      :: [[ Rep ϕ ]] (ThePar α) → α

data NoPar    -- empty
```

In the *Generic* class, the type family *Rep* encodes the generic representation associated with user datatype α, and *ThePar* extracts the last parameter from the datatype. In case

the datatype is of kind $\star$, we use *NoPar*; a type family default allows us to leave the type instance empty for types of kind $\star$. The conversion functions *from* and *to* perform the conversion between the user datatype values and the interpretation of its generic representation.

2.4 Example datatype encodings

We now show two complete examples of how user datatypes are encoded in `structured`. Naturally, users should never have to define these manually; a release version of `structured` would be incorporated in the compiler, allowing automatic derivation of *Generic* instances.

Choice The first datatype we encode represents a choice between four options:

```
data Choice = A | B | C | D
```

Choice is a datatype of kind $\star$, so we do not need to provide a type instance for *ThePar*. The encoding, albeit verbose, is straightforward:

```
instance Generic Choice where
  Rep Choice =
    Data (MD "Choice" "Module" False)
      (Bin (Bin (Leaf (Con (MC "A" Prefix False) Empty))
                (Leaf (Con (MC "B" Prefix False) Empty)))
           (Bin (Leaf (Con (MC "C" Prefix False) Empty))
                (Leaf (Con (MC "D" Prefix False) Empty))))
  from A = D1 (L1 (L1 (C1 U1)))
  from B = D1 (L1 (R1 (C1 U1)))
  from C = D1 (R1 (L1 (C1 U1)))
  from D = D1 (R1 (R1 (C1 U1)))
  to (D1 (L1 (L1 (C1 U1)))) = A
  ...
```

We use a balanced tree structure for the constructors; in Section 3 we will see how this can be changed without any user effort.

Lists Standard Haskell lists are a type of kind $\star \to \star$. We break down its type representation into smaller fragments using type synonyms, to ease comprehension. The encoding of the metadata of each constructor and the two arguments to (:) follows:

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

The encoding of the first argument to $(:)$, H, states that there is no record selector, and that the argument is the parameter Par. The encoding of the second argument, T, is a recursive occurrence of the same datatype being defined ($Rec\ S\ []$).

With these synonyms in place, we can show the complete $Generic$ instance for lists:

```
instance Generic [α] where
  Rep [α] = Data (MD "[]" "Prelude" False)
                 (Bin (Leaf (Con MC_Nil Empty))
                      (Leaf (Con MC_Cons (Bin H T))))
  ThePar [α] = α
  from []      = D1 (L1 (C1 U1))
  from (h : t) = D1 (R1 (C1 (S1 (Par h) :×: S1 (Rec t))))
  to (D1 (L1 (C1 U1)))                          = []
  to (D1 (R1 (C1 (S1 (Par h) :×: S1 (Rec t))))) = h : t
```

The type function $ThePar$ extracts the parameter α from $[\alpha]$; the *from* and *to* conversion functions are unsurprising.

3 Left- and right-biased encodings

The `structured` library uses trees to store the constructors inside a datatype, as well as the fields inside a constructor. So far we have kept these trees balanced, but other choices would be acceptable too. In fact, the balancing choice determines a generic view (Holdermans et al. 2006). Different balancings might be more convenient for certain generic functions. For example, if we are defining a binary encoding function, it is convenient to use the balanced encoding, as then we can easily minimise the number of bits used to encode a constructor. On the other hand, if we are defining a generic function that extracts the first argument to a constructor (if it exists), we would prefer using a right-nested view, as then we can simply pick the first argument on the left. Fortunately, we do not have to provide multiple representations to support this; we can automatically convert between different balancings. As an example, we see in this section how to convert from the (default) balanced encoding to a right-nested one.

This is the first conversion shown in this paper, and as such serves as an introduction to our conversions. Following the style of Magalhães and Löh (2014), we use a type family to adapt the representation, and a type-class to adapt the values. Since this conversion works at the top of the hierarchy (on `structured`), the new balancing persists in future conversions, so a generic function in `generic-deriving` could make use of a right-biased encoding.

3.1 Type conversion

The essential part of the type conversion is a type function that performs one rotation to the right on a tree:

```
RotR (α :: Tree κ) :: Tree κ
RotR (Bin (Bin α β) γ) = Bin α (Bin β γ)
RotR (Bin (Leaf α)  γ) = Bin (Leaf α) γ
```

We then apply this rotation repeatedly at the top level until the tree contains a *Leaf* on the left subtree, and then proceed to rotate the right subtree:

$$
\begin{aligned}
&S_{\to}SR_d\ (\alpha :: Data) :: Data\\
&S_{\to}SR_d\ (Data\ \iota\ \alpha) = Data\ \iota\ (S_{\to}SR_{cs}\ \alpha)\\
&S_{\to}SR_{cs}\ (\alpha :: Tree\ Con) :: Tree\ Con\\
&S_{\to}SR_{cs}\ Empty = Empty\\
&S_{\to}SR_{cs}\ (Leaf\ (Con\ \iota\ \ \gamma)) = Leaf\ (Con\ \iota\ (S_{\to}SR_{fs}\ \gamma))\\
&S_{\to}SR_{cs}\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma) = S_{\to}SR_{cs}\ (RotR\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma))\\
&S_{\to}SR_{cs}\ (Bin\ (Leaf\ \alpha)\ \ \gamma) = Bin\ (S_{\to}SR_{cs}\ (Leaf\ \alpha))\ (S_{\to}SR_{cs}\ \gamma)\\
&S_{\to}SR_{fs}\ (\alpha :: Tree\ Field) :: Tree\ Field\\
&S_{\to}SR_{fs}\ Empty = Empty\\
&S_{\to}SR_{fs}\ (Leaf\ \gamma) = Leaf\ \gamma\\
&S_{\to}SR_{fs}\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma) = S_{\to}SR_{fs}\ (RotR\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma))\\
&S_{\to}SR_{fs}\ (Bin\ (Leaf\ \alpha)\ \ \gamma) = Bin\ (Leaf\ \alpha)\ (S_{\to}SR_{fs}\ \gamma)
\end{aligned}
$$

The conversion for constructors ($S_{\to}SR_{cs}$) and selectors ($S_{\to}SR_{fs}$) differs only in the treatment for leaves, as the leaf of a selector is the stopping point of this transformation.

3.2 Value conversion

The value-level conversion is witnessed by a type class:

$$
\begin{aligned}
&\textbf{class}\ Convert_{S_{\to}SR}\ (\alpha :: Data)\ \textbf{where}\\
&\quad s_{\to}rs :: [\![\alpha]\!]\ \rho \to [\![S_{\to}SR_d\ \alpha]\!]\ \rho\\
&\quad s_{\leftarrow}rs :: [\![S_{\to}SR_d\ \alpha]\!]\ \rho \to [\![\alpha]\!]\ \rho
\end{aligned}
$$

We skip the definition of the instances, as they are mostly unsurprising and can be found in our code bundle.

3.3 Example

To test the conversion, we define a generic function that computes the depth of the encoding of a constructor:

$$
\begin{aligned}
&\textbf{class}\ CountSums_r\ \alpha\ \textbf{where}\\
&\quad countSums_r :: [\![\alpha]\!]\ \rho \to Int\\
&\textbf{instance}\ (CountSums_r\ \alpha) \Rightarrow CountSums_r\ (Data\ \iota\ \alpha)\ \textbf{where}\\
&\quad countSums_r\ (D_1\ x) = countSums_r\ x\\
&\textbf{instance}\ CountSums_r\ Empty\ \ \textbf{where}\ countSums_r\ \_ = 0\\
&\textbf{instance}\ CountSums_r\ (Leaf\ \alpha)\ \textbf{where}\ countSums_r\ \_ = 0\\
&\textbf{instance}\ (CountSums_r\ \alpha, CountSums_r\ \alpha)\\
&\qquad \Rightarrow CountSums_r\ (Bin\ \alpha\ \beta :: Tree\ Con)\ \textbf{where}\\
&\quad countSums_r\ (L_1\ x) = 1 + countSums_r\ x\\
&\quad countSums_r\ (R_1\ x) = 1 + countSums_r\ x
\end{aligned}
$$

We now have two ways of calling this function; one using the standard encoding, and other using the right-nested encoding obtained using $Convert_{S\to SR}$:

$$\begin{aligned}
&countSumsBal :: (Generic\ \alpha, CountSums_r\ (Rep\ \alpha)) \Rightarrow \alpha \to Int\\
&countSumsBal = countSums_r \circ from\\
&countSumsR :: (Generic\ \alpha, Convert_{S\to SR}\ (Rep\ \alpha)\\
&\qquad\qquad , CountSums_r\ (S_\to SR_d\ (Rep\ \alpha))) \Rightarrow \alpha \to Int\\
&countSumsR = countSums_r \circ s_\to rs \circ from
\end{aligned}$$

Applying these two functions to the constructors of the *Choice* datatype should give different results:

$$\begin{aligned}
&testCountSums :: ([Int], [Int])\\
&testCountSums = (map\ countSumsBal\ [A,B,C,D]\\
&\qquad\qquad , map\ countSumsR\quad [A,B,C,D])
\end{aligned}$$

Indeed, *testCountSums* evaluates to $([2,2,2,2],[1,2,3,3])$ as expected. As we've seen, not only can we obtain a different balancing without having to duplicate the representation, but we can also effortlessly apply the same generic function to differently-balanced encodings. Furthermore, the conversions shown in the coming sections automatically "inherit" the balancing chosen in `structured`, allowing us to provide representations with different balancings to the other GP libraries as well.

4 From `structured` to `generic-deriving`

In this section we show how to obtain `generic-deriving` representations from `structured`.

4.1 Encoding `generic-deriving`

The first step is to define `generic-deriving`. We could use its definition as implemented in the `GHC.Generics` module, but it seems more appropriate to at least make use of proper kinds. We thus redefine `generic-deriving` in this paper to bring it up to date with the most recent compiler functionality.[3] This is not essential for our conversions, and should be seen only as a small improvement. The type representation is similar to a collapsed version of `structured`, where all types inhabit a single kind Un_D:

$$\begin{aligned}
&\textbf{kind}\ Un_D = V_D\\
&\qquad\qquad \mid U_D\\
&\qquad\qquad \mid Par_D\\
&\qquad\qquad \mid K_D \quad KType \quad \star\\
&\qquad\qquad \mid Rec_D\ RecType\ (\star \to \star)
\end{aligned}$$

[3] Along the lines of its proposed kind-polymorphic overhaul described in `http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/GenericDeriving#Kindpolymorphicoverhaul`.

$$
\begin{array}{l}
\quad \mid M_D\ Meta_D\ Un_D \\
\quad \mid Un_D\ :{+}:_D\ Un_D \\
\quad \mid Un_D\ :{\times}:_D\ Un_D \\
\quad \mid (\star \to \star)\ :{\circ}:_D\ Un_D \\
\textbf{kind}\ Meta_D = D_D\ MetaData \mid C_D\ MetaCon \mid F_D\ MetaField
\end{array}
$$

Since many names are the same as those in `structured`, we use the "D" subscript for `generic-deriving` names. V_D, U_D, Par_D, K_D, Rec_D, and $(:{\circ}:_D)$ behave very much like the `structured` *Empty*, *Leaf*, *Par*, *K*, *Rec*, and $(:{\circ}:)$, respectively. The binary operators $(:{+}:_D)$ and $(:{\times}:_D)$ are equivalent to *Bin*, and M_D encompasses `structured`'s *Data*, *Con*, and *Field*.

Having seen the interpretation of `structured`, the interpretation of the `generic-deriving` universe is unsurprising:

$$
\begin{array}{l}
\textbf{data}\ [\![\, \alpha :: Un_D \,]\!]_D\ (\rho :: \star) :: \star\ \textbf{where} \\
\quad U_{1D} \quad :: [\![\, U_D \,]\!]_D\ \rho \\
\quad M_{1D} \quad :: [\![\, \alpha \,]\!]_D\ \rho \to [\![\, M_D\ \iota\ \alpha \,]\!]_D\ \rho \\
\quad Par_{1D} \quad :: \rho \to [\![\, Par_D \,]\!]_D\ \rho \\
\quad K_{1D} \quad :: \alpha \to [\![\, K_D\ \iota\ \alpha \,]\!]_D\ \rho \\
\quad Rec_{1D} \quad :: \phi\ \rho \to [\![\, Rec_D\ \iota\ \phi \,]\!]_D\ \rho \\
\quad Comp_{1D} :: \phi\ ([\![\, \alpha \,]\!]_D\ \rho) \to [\![\, \phi\ :{\circ}:_D\ \alpha \,]\!]_D\ \rho \\
\quad L_{1D} \quad :: [\![\, \phi \,]\!]_D\ \rho \to [\![\, \phi\ :{+}:_D\ \psi \,]\!]_D\ \rho \\
\quad R_{1D} \quad :: [\![\, \psi \,]\!]_D\ \rho \to [\![\, \phi\ :{+}:_D\ \psi \,]\!]_D\ \rho \\
\quad :{\times}:_D \quad :: [\![\, \phi \,]\!]_D\ \rho \to [\![\, \psi \,]\!]_D\ \rho \to [\![\, \phi\ :{\times}:_D\ \psi \,]\!]_D\ \rho
\end{array}
$$

The significant difference from `structured` is the relative lack of structure. The types (and kinds) do not prevent an L_{1D} from showing up under a $:{\times}:_D$, for example.

User datatypes are converted to the generic representation using two type classes:

$$
\begin{array}{l}
\textbf{class}\ Generic_D\ (\alpha :: \star)\ \textbf{where} \\
\quad Rep_D \quad \alpha :: Un_D \\
\quad ThePar_D\ \alpha :: \star \\
\quad ThePar_D = NoPar \\
\quad from_D \ :: \alpha \to [\![\, Rep_D\ \alpha \,]\!]_D\ (ThePar_D\ \alpha) \\
\quad to_D \quad :: [\![\, Rep_D\ \alpha \,]\!]_D\ (ThePar_D\ \alpha) \to \alpha \\
\\
\textbf{class}\ Generic_{1D}\ (\phi :: \star \to \star)\ \textbf{where} \\
\quad Rep_{1D}\ \phi :: Un_D \\
\quad from_{1D} :: \phi\ \rho \to [\![\, Rep_{1D}\ \phi \,]\!]_D\ \rho \\
\quad to_{1D} \quad :: [\![\, Rep_{1D}\ \phi \,]\!]_D\ \rho \to \phi\ \rho
\end{array}
$$

Class $Generic_D$ is used for all supported datatypes, and encodes a simple view on the constructor arguments. For datatypes that abstract over (at least) one type parameter, an instance for $Generic_{1D}$ is also required. The type representation in this instance encodes the more general view of constructor arguments (i.e. using Par_D, Rec_D, and $:{\circ}:_D$). Note that $Generic_D$ doesn't currently have $ThePar_D$ in GHC, but we think this is a (minor)

improvement. Furthermore, the presence of a type family default makes it backwards-compatible.

Since these two classes represent essentially two different universes in `generic-deriving`, we need to define two distinct conversions from `structured` to `generic-deriving`.

4.2 To $Generic_D$

The universe of `structured` has a detailed encoding of constructor arguments. However, many generic functions do not need such detailed information, and are simpler to write by giving a single case for constructor arguments (imagine, for example, a function that counts the number of arguments). For this purpose, `generic-deriving` states that representations from $Generic_D$ contain only the K_D type at the arguments (so no Par_D, Rec_D, and $:\circ:_D$).

To derive $Generic_D$ instances from $Generic$, we use the following instance:

$$\begin{aligned}
&\textbf{instance}\ (Generic\ \alpha, Convert_{S_\to D_0}\ (Rep\ \alpha)) \Rightarrow Generic_D\ \alpha\ \textbf{where}\\
&\quad Rep_D\ \ \alpha = S_\to G_0\ (Rep\ \alpha)\ (ThePar\ \alpha)\\
&\quad ThePar_D\ \alpha = ThePar\ \alpha\\
&\quad from_D\ = s_\to g_0 \circ from\\
&\quad to_D\ \ = to \circ s_\leftarrow g_0
\end{aligned}$$

In the remainder of this section, we explain the definition of $S_\to G_0$, a type family that converts a representation of `structured` into one of `generic-deriving`, and the class $Convert_{S_\to D_0}$, whose methods $s_\to g_0$ and $s_\leftarrow g_0$ perform the value-level conversion.

Type representation conversion To convert between the type representations, we use a type family:

$$S_\to G_0\ (\alpha :: \kappa)\ (\rho :: \star) :: Un_D$$

The kind of $S_\to G_0$ is overly polymorphic; its input is not any κ, but only the kinds that make up the `structured` universe. We could encode this by using multiple type families, one at each "level". For simplicity, however, we use a single type family, which we instantiate only for the `structured` representation types.

The encoding of datatype meta-information is left unchanged:

$$S_\to G_0\ (Data\ \iota\ \alpha)\ \rho = M_D\ (D_D\ \iota)\ (S_\to G_0\ \alpha\ \rho)$$

We then proceed with the conversion of the constructors:

$$\begin{aligned}
&S_\to G_0\ Empty && \rho = V_D\\
&S_\to G_0\ (Leaf\ (Con\ \iota\ \alpha)) && \rho = M_D\ (C_D\ \iota)\ (S_\to G_0\ \alpha\ \rho)\\
&S_\to G_0\ (Bin\ \alpha\ \beta) && \rho = (S_\to G_0\ \alpha\ \rho)\ :+:_D\ (S_\to G_0\ \beta\ \rho)
\end{aligned}$$

Again, the structure of the constructors and their meta-information is left unchanged. We proceed similarly for constructor fields:

$$
\begin{array}{ll}
S_{\to} G_0\ Empty & \rho = U_D \\
S_{\to} G_0\ (Leaf\ (Field\ \iota\ \alpha))\ & \rho = M_D\ (F_D\ \iota)\ (S_{\to} G_0\ \alpha\ \rho) \\
S_{\to} G_0\ (Bin\ \alpha\ \beta) & \rho = (S_{\to} G_0\ \alpha\ \rho)\ :\times:_D\ (S_{\to} G_0\ \beta\ \rho)
\end{array}
$$

Finally, we arrive at individual fields, where the interesting part of the conversion takes place:

$$
\begin{array}{lll}
S_{\to} G_0\ (K\ \iota\ \alpha) & \rho = K_D\ \iota & \alpha \\
S_{\to} G_0\ (Rec\ \iota\ \phi) & \rho = K_D\ (R\ \iota)\ (\phi\ \rho) & \\
S_{\to} G_0\ Par & \rho = K_D\ P & \rho
\end{array}
$$

Basically, all the information kept about the field is condensed into the first argument of K_D. Composition requires special care, but gets similarly collapsed into a K_D:

$$
\begin{array}{ll}
S_{\to} G_0\ (\phi\ :\circ:\ \alpha)\ \rho = K_D\ U\ (\phi\ (S_{\to} G_{0_{comp}}\ \alpha\ \rho)) & \\
S_{\to} G_{0_{comp}}\ (\alpha :: Arg)\ (\rho :: \star) :: \star & \\
S_{\to} G_{0_{comp}}\ Par & \rho = \rho \\
S_{\to} G_{0_{comp}}\ (K\ \alpha) & \rho = \alpha \\
S_{\to} G_{0_{comp}}\ (Rec\ \iota\ \phi)\ \rho = \phi\ \rho & \\
S_{\to} G_{0_{comp}}\ (\phi\ :\circ:\ \alpha)\ \rho = \phi\ (S_{\to} G_{0_{comp}}\ \alpha\ \rho) &
\end{array}
$$

The auxiliary type family $S_{\to} G_{0_{comp}}$ takes care of unwrapping the composition, and re-applying the type to its arguments.

Value conversion Having performed the type-level conversion, we have to convert the values in an equally type-directed fashion. We begin with datatypes:

$$
\begin{array}{l}
\textbf{class}\ Convert_{S_{\to} D_0}\ (\alpha :: \kappa)\ \textbf{where} \\
\quad s_{\to} g_0 :: [\![\, \alpha \,]\!]\ \rho \to [\![\, S_{\to} G_0\ \alpha\ \rho \,]\!]\ \rho \\
\quad s_{\leftarrow} g_0 :: [\![\, S_{\to} G_0\ \alpha\ \rho \,]\!]\ \rho \to [\![\, \alpha \,]\!]\ \rho \\
\textbf{instance}\ (Convert_{S_{\to} D_0}\ \alpha) \Rightarrow Convert_{S_{\to} D_0}\ (Data\ \iota\ \alpha)\ \textbf{where} \\
\quad s_{\to} g_0\ (D_1\ x) = M_{1D}\ (s_{\to} g_0\ x) \\
\quad s_{\leftarrow} g_0\ (D_1\ x) = M_{1D}\ (s_{\leftarrow} g_0\ x)
\end{array}
$$

As in the type conversion, we simply traverse the representation, and convert the constructors with another function. From here on, we omit the $s_{\leftarrow} g_0$ direction, as it is entirely symmetrical.

Constructors and selectors simply traverse the meta-information:

$$
\begin{array}{l}
\textbf{instance}\ (Convert_{S_{\to} D_0}\ \alpha) \Rightarrow Convert_{S_{\to} D_0}\ (Leaf\ (Con\ \iota\ \alpha))\ \textbf{where} \\
\quad s_{\to} g_0\ (C_1\ x) = M_{1D}\ (s_{\to} g_0\ x) \\
\textbf{instance}\ (Convert_{S_{\to} D_0}\ \alpha, Convert_{S_{\to} D_0}\ \beta) \Rightarrow Convert_{S_{\to} D_0}\ (Bin\ \alpha\ \beta)\ \textbf{where} \\
\quad s_{\to} g_0\ (L_1\ x) = L_{1D}\ (s_{\to} g_0\ x) \\
\quad s_{\to} g_0\ (R_1\ x) = R_{1D}\ (s_{\to} g_0\ x) \\
\\
\textbf{instance}\ Convert_{S_{\to} D_0}\ Empty\ \textbf{where}
\end{array}
$$

```
  s→g0 U1 = U1D
instance (Convert_{S→D0} α) ⇒ Convert_{S→D0} (Leaf (Field ι α)) where
  s→g0 (S1 x)    = M1D (s→g0 x)
instance (Convert_{S→D0} α, Convert_{S→D0} β) ⇒ Convert_{S→D0} (Bin α β) where
  s→g0 (x :×: y) = s→g0 x :×:D s→g0 y
```

Finally, at the argument level, we collapse everything into K_{1D}:

```
instance Convert_{S→D0} (K ι α)    where s→g0 (K x)    = K1D x
instance Convert_{S→D0} (Rec ι φ)  where s→g0 (Rec x)  = K1D x
instance Convert_{S→D0} Par        where s→g0 (Par x)  = K1D x
instance (Functor φ, Convert_comp α) ⇒ Convert_{S→D0} (φ :∘: α) where
  s→g0 (Comp x) = K1D (g→g0_comp x)
```

Again, for composition we need to unwrap the representation, removing all representation types within:

```
class Convert_comp (α :: Arg) where
  g→g0_comp :: Functor φ ⇒ φ (⟦α⟧ ρ) → φ (S→G0_comp α ρ)
instance Convert_comp Par        where g→g0_comp = fmap unPar
instance Convert_comp (K ι α)    where g→g0_comp = fmap unK1
instance Convert_comp (Rec ι φ)  where g→g0_comp = fmap unRec
instance (Functor φ, Convert_comp α) ⇒ Convert_comp (φ :∘: α) where
  g→g0_comp = fmap (g→g0_comp ∘ unComp)
```

With all these instances in place, the $Generic\ \alpha \Rightarrow Generic_D\ \alpha$ shown at the beginning of this section takes care of converting to the simpler representation of `generic-deriving` without syntactic overhead. In particular, all generic functions defined over the $Generic_D$ class, such as *gshow* and *genum* from the `generic-deriving` package, are now available to all types in `structured`, such as *Choice* and $[\alpha]$.

Example: length To test the conversion, we define a generic function in `generic-deriving` that computes the number of elements in a structure:

```
class GLength_r (α :: Un_D) where
  gLength_r :: ⟦α⟧_D ρ → Int
```

We omit the instances of $GLength_r$ as they are unsurprising: we traverse the representation until we reach the arguments, which are recursively counted and added.

While the $GLength_r$ class works on the generic representation, a user-facing class *GLength* takes care of handling user-defined datatypes. We define a generic default which implements *gLength* generically:

```
class GLength (α :: ⋆) where
  gLength :: α → Int
```

$$\begin{aligned}&\textbf{default}\ gLength :: (Generic_D\ \alpha, GLength_r\ (Rep_D\ \alpha)) \Rightarrow \alpha \to Int\\ &gLength = gLength_r \circ from_D\end{aligned}$$

Because of the generic default, instantiating $GLength$ to datatypes with a $Generic_D$ instance is very simple:

$$\begin{aligned}&\textbf{instance}\ GLength\ [\alpha]\\ &\textbf{instance}\ GLength\ Choice\end{aligned}$$

Recall, however, that in Section 2.4 we have given only $Generic$ instances for $Choice$ and $[\alpha]$, not $Generic_D$. However, due to the $(Generic\ \alpha, Convert_{S\to D_0}\ (Rep\ \alpha)) \Rightarrow Generic_D\ \alpha$ instance of the beginning of this section, $Choice$ and $[\alpha]$ automatically get a $Generic_D$ instance, which is being used here.

We can test that this function behaves as expected: $gLength\ [0,1,2,3]$ returns 4, and $gLength\ D$ returns 0. And further: using our previous work (Magalhães and Löh 2014), we also gain all the functionality from other libraries, such as `syb` traversals or a zipper, for example.

4.3 To $Generic_{1D}$

Converting to $Generic_{1D}$ is very similar, only that we preserve more structure. The conversion is similarly performed by two components.

Type representation conversion We define a type family to perform the conversion of the type representation:

$$S_\to G_1\ (\alpha :: \kappa) :: Un_D$$

The type instances for the datatype, constructors, and fields behave exactly like in $S_\to G_0$, so we skip straight to the constructor arguments, which are simple to handle because they are in one-to-one correspondence:

$$\begin{aligned}S_\to G_1\ (K\ \iota\ \alpha) &= K_D\ \iota\ \alpha\\ S_\to G_1\ (Rec\ \iota\ \alpha) &= Rec_D\ \iota\ \alpha\\ S_\to G_1\ Par &= Par_D\\ S_\to G_1\ (\phi\ :\circ:\ \alpha) &= \phi\ \ :\circ:_D\ \ S_\to G_1\ \alpha\end{aligned}$$

Value conversion The value-level conversion is as trivial as the type-level conversion, so we omit it from the paper. It is witnessed by a poly-kinded type class:

$$\begin{aligned}&\textbf{class}\ Convert_{S\to D_1}\ (\alpha :: \kappa)\ \textbf{where}\\ &\quad s_\to g_1 :: [\![\alpha]\!]\ \rho \to [\![S_\to G_1\ \alpha]\!]_D\ \rho\end{aligned}$$

Again, we only give instances of $Convert_{S\to D_1}$ for the representation types of `structured`.

Using this class we can give instances for each user datatype that we want to convert. For example, the list datatype (instantiated in `structured` in Section 2.4) can be transported to `generic-deriving` with the following instance:

```
instance Generic1D [] where
  Rep1D [] = S→G1 (Rep [NoPar])
  from1D x = s→g1 (from x)
```

We use $Rep\ [NoPar]$ because we need to instantiate the list with some parameter. Any parameter will do, because we know that $\forall \phi\ \alpha\ \beta . Rep\ (\phi\ \alpha) \sim Rep\ (\phi\ \beta)$. However, this means that, unlike in Section 4.2, we cannot give a single instance of the form $Generic\ (\phi\ \rho) \Rightarrow Generic_{1D}\ \phi$. The reason for this is the disparity between the kinds of the two classes involved; $Generic_{1D}$ only mentions the parameter ρ in the signature of its methods, where it's impossible to state that said ρ is the same as in the instance head ($Generic\ (\phi\ \rho)$).

This is not a major issue, however, because $Generic_{1D}$ instances are currently derived by the compiler. If these instances were to be replaced by conversions from $Generic$, the behaviour of **deriving** $Generic_{1D}$ would change to mean "derive $Generic$, and define a trivial $Generic_{1D}$ instance".

With the instance above, functionality defined in the `generic-deriving` package over the $Generic_{1D}$ class, such as *gmap*, is now available to $[\alpha]$.

5 Conclusion

Following the lines of generic generic programming, we've shown how to add another level of hierarchy to the current landscape of GP libraries in Haskell. Introducing `structured` allows us to unify the two generic views in `generic-deriving`, and brings the possibility of using different nestings in the constructor and constructor arguments encoding. These developments can help in simplifying the implementation of GP in the compiler, as less code has to be part of the compiler itself (only that for generating `structured` instances), and more code can be moved into the user domain. GP library writers also see their life simplified, by gaining access to multiple generic views without needing to duplicate code.

Should `structured` turn out to be not informative enough to cover a particular approach, then it can always be refined or extended. Since we do not advocate to use `structured` directly, this means that only the direct conversions from `structured` have to be extended, and everything else will just keep working. Our hierarchical approach facilitates a future where GP libraries themselves are as modular and duplication-free as the code they enable end users to write.

Acknowledgements

The first author is funded by EPSRC grant number EP/J010995/1. We thank the anonymous reviewers for the helpful feedback.

Bibliography

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez Yakushev. Generic views on data types. In *Proceedings of the 8th International Conference on Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. Springer, 2006. doi:10.1007/11783596_14.

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.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*, pages 244–255. ACM, 2004. doi:10.1145/1016850.1016883.

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

José Pedro Magalhães. Generic programming with multiple parameters. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 136–151. Springer International Publishing, 2014. doi:10.1007/978-3-319-07151-0_9.

José Pedro Magalhães and Andres Löh. A formal comparison of approaches to datatype-generic programming. In James Chapman and Paul Blain Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–67. Open Publishing Association, 2012. doi:10.4204/EPTCS.76.6.

José Pedro Magalhães and Andres Löh. Generic generic programming. In Matthew Flatt and Hai-Feng Guo, editors, *Practical Aspects of Declarative Languages*, volume 8324 of *Lecture Notes in Computer Science*, pages 216–231. Springer International Publishing, 2014. doi:10.1007/978-3-319-04132-2_15.

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

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, The University of Nottingham, November 2007.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 13–24. ACM, 2008. doi:10.1145/1411318.1411321.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 233–244. ACM, 2009. doi:10.1145/1596550.1596585.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 51–62. ACM, 2008. doi:10.1145/1411204.1411215.

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 341–352. ACM, 2009. doi:10.1145/1596550.1596599.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, volume 37, pages 1–16. ACM, December 2002. doi:10.1145/581690.581691.

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.