Generic Generic Programming

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

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

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. Over the years, several different approaches to GP in Haskell have surfaced. These approaches are often similar, but certain differences make them particularly well-suited for one specific domain or application. As such, there is a lot of code duplication across GP libraries, which is rather unfortunate, given the original goals of GP.

To address this problem, we define conversions from one popular GP library representation to several others. Our work unifies many approaches to GP, and simplifies the life of both library writers and users. Library writers can define their approach as a conversion from our library, obviating the need for writing meta-programming code for generation of conversions to and from the generic representation. Users of GP, who often struggle to find "the right approach" to use, can now mix and match functionality from different libraries with ease, and need not worry about having multiple (potentially inefficient and large) code blocks for generic representations in different approaches.

1 Introduction

GP can be used to reduce code duplication, increase the level of abstraction in a program, and derive useful functionality "for free" from the structure of datatypes. Over the past few years, many approaches to GP have surfaced. Including pre-processors, template-based approaches, language extensions, and libraries, there are well over 15 different approaches to GP in Haskell [7, Chapter 8]. This abundance is caused by the lack of a clearly superior approach; each approach has its strengths and weaknesses, uses different implementation mechanisms, a different generic view [4] (i.e. a different structural 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 they first 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 [9], and concluded that some approaches clearly subsume others. The relevance of this fact extends above mere theoretical interest, since a comparison can also provide means for converting between approaches. Ironically, code duplication across generic programming libraries is evident: the same function can be nearly identical in different approaches, yet impossible to reuse, due to the underlying differences

M. Flatt and H.-F. Guo (Eds.): PADL 2014, LNCS 8324, pp. 216–231, 2014.

in representation. A conversion between approaches provides the means to remove duplication of generic code.

In this paper we show how to automatically derive representations for many popular GP libraries, all coming from one single compiler-supported approach. The base approach, `generic-deriving` [10], has been supported in the Glasgow Haskell Compiler (GHC), the main Haskell compiler, since version 7.2.1 (August 2011). From `generic-deriving` we define conversions to other popular generic libraries: `regular` [13], `multirec` [14], and `syb` [5, 6]. Some of these libraries are remarkably different from each other, yet advanced type-level features in GHC, such as GADTs [16], type functions [15], and kind polymorphism [18], allow us to perform these conversions.

Using the type class system, our conversions remain entirely under the hood for the end user, who need not worry anymore about which GP approach does what, and can simply use generic functions from any approach. As an example, the following combination of generic functionality is now possible:

```
import Generics.Deriving              as GD
import Generics.Regular.Rewriting     as R
import Generics.SYB.Schemes           as S
import Conversions ()

data Logic α = Var α | Logic α :∨: Logic α | Not (Logic α) | T | F
                 deriving (GD.Generic)

rewriting :: Logic Char
rewriting = let elim2Not = R.rule $ λx → Not (Not x) :~>: x
            in R.bottomUp (R.rewrite elim2Not) $ T :∨: Not (Not (Var 'p'))

size :: Int
size = S.everything (+) (const 1) $ Var 'p' :∨: Var 'q'

rename :: Logic String
rename = GD.gmap ('_':) $ T :∨: Var "p"
```

Here, the user defines a *Logic* datatype, and lets the compiler automatically derive a *Generic* representation for it (from `generic-deriving`). Three examples then show how functionality specific to three separate GP libraries can be used from this single representation:

- In *rewriting*, a rewrite rule is applied to a logical expression. The rewriting system requires a fixed-point view on data for encoding expressions extended with metavariables [13]. This fixed-point view is provided by the `regular` library. The term *rewriting* evaluates to *T* :∨: *Var* `'p'`.
- Expression *size* showcases the combinator approach to GP typical of `syb`, reducing all leaves to 1, and combining them with the (+) operator. The term *size* evaluates to 5.
- Expression *rename* uses a map on the *String* parameter of *Logic* to rename all the variables. This makes uses of the support for parameters of `generic-deriving`. The term *rename* evaluates to *T* :∨: *Var* `"_p"`.

All this functionality can be achieved using only the *Generic* representation of `generic-deriving`, and by importing the conversion instances defined in some module

Conversions (provided by us); there is no need to derive any generic representations for `regular` or `syb`. Previously, combining the functionality of these libraries would also require generic representations for `regular` and `syb`. This would bring a dependency on Template Haskell [17] for deriving `regular` representations, and added code bloat.

Generic library writers also see an improvement in their quality of life, as they no longer need to write Template Haskell code to derive representations for their libraries, and can instead rely on our conversion functions. Furthermore, many generic functions can now be recognised as truly duplicated across approaches, and can be deprecated appropriately. Defining new approaches to GP has never been easier; GP libraries can be kept small and specific, focusing on one particular aspect, as users can easily find and use other generic functionality in other approaches.

We say this work is about *generic generic programming* because it is generic over generic programming approaches. Specifically, we define conversions to multiple GP libraries (Sections 3 to 5), covering a wide range of approaches, including libraries with a fixed-point view on data (`regular` and `multirec`), and a library based on traversal combinators (`syb`). In defining our conversions to other libraries, we also update their definitions to make use of the latest GHC extensions (namely data kinds and kind polymorphism [18]). This is not essential for our conversions (i.e. we are not changing the libraries to make our conversion easier), but it improves the libraries (while these libraries were always type safe, our changes make them more kind safe).

Moreover, our work brings forward a new way of looking at GP, where new, special-purpose GP libraries can be easily defined, without needing to repeat lots of common infrastructure. Users of GP can now simply cherry-pick generic functions from different libraries, without having to worry about the overhead introduced by each GP approach.

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 [18], 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. When we use the same name for a constructor and a type, the "level" of the 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 7.6.2, is available at `http://dreixel.net/research/code/ggp.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.

Structure of the Paper. We first provide a brief introduction to the `generic-deriving` library for GP (Section 2). We then see how to obtain other libraries from `generic-deriving`: `regular` (Section 3), `multirec` (Section 4), and `syb` (Section 5). We then conclude with a discussion in Section 6. Along the way, we also show several examples of how our conversion enables seamless use of multiple approaches.

2 Introduction to `generic-deriving`

We begin our efforts of homogenising GP libraries by introducing `generic-deriving`, the library from which we derive the other representations.

$$
\begin{array}{ll}
\textbf{kind}\ \mathit{Un}_D = V_D \mid U_D \mid K_D\ \mathit{KType}\ \star & \textbf{data}\ [\![\alpha :: \mathit{Un}_D]\!]_D :: \star\ \textbf{where} \\
\qquad\qquad \mid M_D\ \mathit{Meta}_D\ \mathit{Un}_D & \quad U_{1D} \ :: [\![U_D]\!]_D \\
\qquad\qquad \mid \mathit{Un}_D\ :+:_D\ \mathit{Un}_D & \quad M_{1D} :: [\![\alpha]\!]_D \to [\![M_D\ \iota\ \alpha]\!]_D \\
\qquad\qquad \mid \mathit{Un}_D\ :\times:_D\ \mathit{Un}_D & \quad K_{1D} \ :: \alpha \to [\![K_D\ \iota\ \alpha]\!]_D \\
\textbf{kind}\ \mathit{Meta}_D \ = D_D\ \mathit{MetaData} & \quad L_{1D} \ :: [\![\phi]\!]_D \to [\![\phi\ :+:_D\ \psi]\!]_D \\
\qquad\qquad\quad \mid C_D\ \mathit{MetaCon} & \quad R_{1D} \ :: [\![\psi]\!]_D \to [\![\phi\ :+:_D\ \psi]\!]_D \\
\qquad\qquad\quad \mid F_D\ \mathit{MetaField} & \quad :\times:_D :: [\![\phi]\!]_D \to [\![\psi]\!]_D \to [\![\phi\ :\times:_D\ \psi]\!]_D \\
\textbf{kind}\ \mathit{KType} \ = P \mid R\ \mathit{RecType} \mid U & \\
\textbf{kind}\ \mathit{RecType} = S \mid O &
\end{array}
$$

Fig. 1. Universe and interpretation of `generic-deriving`

Universe. The structure used to encode datatypes in a GP approach is called its *universe* [12]. The universe of `generic-deriving` can be seen on the left in Figure 1. It represents datatypes as a sum of products, additionally keeping track of meta-information. Since GP approaches often use the same names for similar representation types, we use the "D" subscript for `generic-deriving` names.

Datatypes are sums (choices between constructors, encoded with $:+:_D$) of products (constructors with several arguments, encoded with $:\times:_D$). The sum can be nullary (V_D), in case the datatype has no constructors, and so can each of the products (U_D), in case the constructor takes no arguments. Constructor arguments (encoded with K_D) can either be the (last) parameter of the datatype ($K_D\ P$), an occurrence of a datatype, which can be the same as the one we are defining ($K_D\ (R\ S)$) or some other datatype ($K_D\ (R\ O)$), or something else (such as an application of a type variable, encoded with $K_D\ U$). The annotations given by *KType* and *RecType* will prove essential when converting to approaches with a fixed-point view on data (Section 3 and Section 4), as there we need explicit knowledge about the recursive structure of data.

Interpretation. The interpretation of the universe defines the structure of the values that inhabit the datatype representation. Datatype representations are types of kind Un_D. We use a GADT [16] $[\![\_]\!]_D$ to encode the interpretation of the universe of `generic-deriving`, which can be seen on the right in Figure 1. The top-level inhabitant of a datatype representation is always a constructor M_{1D} (with type $[\![M_D\ (D_D\ \iota)\ \alpha]\!]_D$), which serves only as a proxy to store the datatype metadata on its type. An M_{1D} appears also around each constructor (but then with type $[\![M_D\ (C_D\ \iota)\ \alpha]\!]_D$, and each constructor field (but then with type $[\![M_D\ (F_D\ \iota)\ \alpha]\!]_D$). Constructors can be on the left (L_{1D}) or right (R_{1D}) side of a sum. Constructor arguments are encoded in a product structure ($:\times:_D$), or can be empty (U_{1D}). Constructor fields are all encoded with K_{1D}, which is used with different types to encode the meta-information of the field in question (similarly to M_{1D}). We encode the last parameter of the datatype with $K_{1D} :: K_D\ P\ \alpha$, datatype occurrences with $K_{1D} :: K_D\ (R\ \iota)\ \alpha$, with ι being S if the datatype is the same we are encoding and O otherwise, and anything else with $K_{1D} :: K_D\ U\ \alpha$.

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:

$$\begin{aligned}
&\textbf{class}\ Generic_D\ (\alpha :: \star)\ \textbf{where} \\
&\quad Rep_D\ \alpha :: Un_D \\
&\quad from_D :: \alpha \to [\![Rep_D\ \alpha]\!]_D \\
&\quad to_D \quad :: [\![Rep_D\ \alpha]\!]_D \to \alpha
\end{aligned}$$

In the $Generic_D$ class, the type family Rep_D encodes the generic representation associated with user datatype α. The class methods *from* and *to* perform the conversion between the user datatype values and the interpretation of the generic representation. From here on, we shall omit the to_D direction, as it is always entirely symmetrical to $from_D$.

Example Encoding: Lists. We now show an example of how a user datatype is encoded in `generic-deriving`. (Users never have to define the encodings manually; GHC can automatically derive $Generic_D$ instances.) We omit the encoding of metadata in the datatype, constructors, and selectors, as these are not relevant to our developments in the rest of the paper. The simplified instance looks as follows:

$$\begin{aligned}
&\textbf{instance}\ Generic\ [\alpha]\ \textbf{where} \\
&\quad Rep\ [\alpha] = U_D\ :+:_D\ ((K_D\ P\ \alpha)\ :\times:_D\ (K_D\ (R\ S)\ [\alpha])) \\
&\quad from\ [\,] \qquad = L_{1D}\ U_{1D} \\
&\quad from\ (h : t) = R_{1D}\ (K_{1D}\ h\ :\times:_D\ (K_{1D}\ t))
\end{aligned}$$

The first argument of the $(:)$ constructor is tagged as being the parameter (with P), and the second as being a recursive occurrence of the datatype being defined ($R\ S$).

3 From `generic-deriving` to `regular`

In this section we show how to obtain `regular` representations from `generic-deriving`. The `regular` library, first described in the context of generic rewriting [13], encodes datatypes using a "fixed-point view". As such, it abstracts over the recursive position of the datatype, allowing for the definition of recursive morphisms such as cata- and anamorphisms. It was previously thought that a fixed-point view was a requirement for defining recursive morphisms generally, or that it would be very hard or messy in other views. Here we show that this need not be the case, as our conversion to `regular` comes from a non-fixed point view, and is rather simple.

Encoding `regular`. We show a simplified encoding of the universe of `regular` (subscript "R"), omitting the constructor meta-information:

$$\textbf{kind}\ Un_R = U_R \mid I_R \mid K_R \star \mid Un_R\ :+:_R\ Un_R \mid Un_R\ :\times:_R\ Un_R$$

As before, we have a type for encoding unitary constructors (U_R) and a type for constants (K_R). However, we also have a type I_R to encode recursion. The `regular` library supports abstracting over single recursive datatypes only, so I_R need not store the index of what type it encodes. Sums and products behave as in `generic-deriving`.

The interpretation of this universe is parametrised over the type of recursive positions τ, which is used in the I_R case:

$$\begin{array}{ll}
\textbf{data}\ [\![\, \alpha :: Un_R \,]\!]_R\ (\tau :: \star)\ \textbf{where} & \\
\quad U_R & :: [\![\, U_R \,]\!]_R\ \tau \\
\quad I_R & :: \tau \rightarrow [\![\, I_R \,]\!]_R\ \tau \\
\quad K_R & :: \alpha \rightarrow [\![\, K_R\ \alpha \,]\!]_R\ \tau \\
\quad L_R & :: [\![\, \alpha \,]\!]_R\ \tau \rightarrow [\![\, \alpha\ :+:_R\ \beta \,]\!]_R\ \tau \\
\quad R_R & :: [\![\, \beta \,]\!]_R\ \tau \rightarrow [\![\, \alpha\ :+:_R\ \beta \,]\!]_R\ \tau \\
\quad (:\times:_R) & :: [\![\, \alpha \,]\!]_R\ \tau \rightarrow [\![\, \beta \,]\!]_R\ \tau \rightarrow [\![\, \alpha\ :\times:_R\ \beta \,]\!]_R\ \tau
\end{array}$$

The *Regular* class witnesses the conversion between user-defined datatypes and their representation in `regular`. Note how the τ parameter of $[\![\, \alpha \,]\!]_R$ is set to α itself:

$$\begin{array}{l}
\textbf{class}\ \mathit{Regular}\ (\alpha :: \star)\ \textbf{where} \\
\quad PF\ \alpha :: Un_R \\
\quad \mathit{from}_R :: \alpha \rightarrow [\![\, PF\ \alpha \,]\!]_R\ \alpha
\end{array}$$

This means that `regular` encodes a one-layer generic representation, where the recursive positions are values of the original user datatype, not generic representations.

Type Conversion. We now show the first conversion in this paper, which serves as an introduction to the structure of our conversions. We use a type family to adapt the representation, and a type-class to adapt the values. The first step is then to convert the representation types of `generic-deriving` into representation types of `regular` using a type family:

$$D_{\rightarrow}R\ (\alpha :: Un_D) :: Un_R$$

For units, meta-information, sums, and products, the conversion is straightforward:

$$\begin{array}{ll}
D_{\rightarrow}R\ U_D & = U_R \\
D_{\rightarrow}R\ (M_D\ \iota\ \alpha) & = D_{\rightarrow}R\ \alpha \\
D_{\rightarrow}R\ (\alpha\ :+:_D\ \beta) & = D_{\rightarrow}R\ \alpha\ :+:_R\ D_{\rightarrow}R\ \beta \\
D_{\rightarrow}R\ (\alpha\ :\times:_D\ \beta) & = D_{\rightarrow}R\ \alpha\ :\times:_R\ D_{\rightarrow}R\ \beta
\end{array}$$

The interesting case is that for constructor arguments, as we have to treat recursion into the same datatype differently:

$$\begin{array}{ll}
D_{\rightarrow}R\ (K_D\ (R\ S)\ \ \tau) & = I_R \\
D_{\rightarrow}R\ (K_D\ (R\ O)\ \alpha) & = K_R\ \alpha \\
D_{\rightarrow}R\ (K_D\ P\ \qquad \alpha) & = K_R\ \alpha \\
D_{\rightarrow}R\ (K_D\ U\ \qquad \alpha) & = K_R\ \alpha
\end{array}$$

One might wonder what would happen if the `generic-deriving` representation had an inconsistent use of $K_D\ (R\ S)\ \tau$ where τ is not the type being represented. This would lead to a type error, as we explain in the next section.

Value Conversion. Having performed the type-level conversion, we have to convert the values in a type-directed fashion. The conversion of the values is witnessed by the $Convert_{D \to R}$ type class:

$$\textbf{class}\ Convert_{D\to R}\ (\alpha :: Un_D)\ \tau\ \textbf{where}$$
$$d_{\to}r :: [\![\, \alpha \,]\!]_D \to [\![\, D_{\to}R\ \alpha \,]\!]_R\ \tau$$

(We omit the $r_{\to}d$ direction, as it is entirely symmetrical.) This is a multiparameter type class because we need to enforce the restriction that the recursive occurrence under $K_D\ (R\ S)\ \tau$ has to be of the expected type τ:

$$\textbf{instance}\ Convert_{D\to R}\ (K_D\ (R\ S)\ \tau)\ \tau\ \textbf{where}\ d_{\to}r\ (K_{ID}\ x) = I_R\ x$$

The tag $R\ S$ expresses this restriction informally only; the formal guarantee is given by the type-checker, since this instance requires type equality, encoded in the repeated appearance of the variable τ in the instance head. We omit the remaining instances as they are unsurprising.

To finish the conversion, we provide a *Regular* instance for all $Generic_D$ types. It is here that we set the second parameter of $Convert_{D\to R}$ to the type being converted (α):

$$\textbf{instance}\ (Generic_D\ \alpha, Convert_{D\to R}\ (Rep_D\ \alpha)\ \alpha) \Rightarrow Regular\ \alpha\ \textbf{where}$$
$$PF\ \alpha = D_{\to}R\ (Rep_D\ \alpha)$$
$$from_R\ x = d_{\to}r\ (from_D\ x)$$

With this instance, functions defined in the `regular` library are now available to all `generic-deriving` supported datatypes. This is remarkable; in particular, functions that require a fixed-point view on data, such as the generic catamorphism, can be used on `generic-deriving` types without having to provide an explicit *Regular* instance. From the generic library developer point of view there are other advantages. When defining a new generic function that fits the fixed-point view naturally, a developer could implement this function easily in `regular`, but would then require the users of this function to use `regular`, and manually write *Regular* instances for their datatypes, or use the provided Template Haskell code to derive these automatically. Alternatively, the developer could try to define the same function in `generic-deriving`, but this would probably require more effort; the advantage would be that users wouldn't need an external library to use this function, and could rely solely on GHC.

With the instance above, however, the developer can implement the function in `regular`, and the users can use it through the **deriving** $Generic_D$ extension of GHC. In fact, `regular` can be simplified by removing the Template Haskell code for generating *Regular* instances altogether. Given that this code often requires updating due to new releases of GHC changing Template Haskell, this is a clear improvement, and helps reduce clutter from the GP libraries themselves.

4 From `generic-deriving` to `multirec`

Having seen how to convert from `generic-deriving` to a fixed-point view for a single datatype, we are ready to tackle the challenge of converting to `multirec`, a library with a fixed-point view over *families* of datatypes [14].

Encoding `multirec`**.** The universe of `multirec` is similar to that of `regular`, only I_M is parametrised over an index (since we now support recursion into several datatypes), and we have a new code $:\triangleright:_M$ for tagging a part of the representation with a concrete index:

$$\begin{aligned}\textbf{data}\ \mathit{Un}_M\ \kappa = {} & U_M \mid I_M\ \kappa \mid K_M\ \star \mid \mathit{Un}_M\ \kappa\ :\triangleright:_M\ \kappa \\ \mid {} & \mathit{Un}_M\ \kappa\ :+:_M\ \mathit{Un}_M\ \kappa \mid \mathit{Un}_M\ \kappa\ :\times:_M\ \mathit{Un}_M\ \kappa\end{aligned}$$

Tagging is used to differentiate between different datatypes within a single representation. As an example, we show a family of two mutually-recursive datatypes together with the type-level representation in `multirec`:

$$\begin{aligned}&\textbf{data}\ \mathit{Zig}\ = \mathit{Zig}\ \mathit{Zag} \mid \mathit{ZigEnd}\\ &\textbf{data}\ \mathit{Zag} = \mathit{Zag}\ \mathit{Zig}\\ &\mathit{ZigZagRep} = \quad ((I_M\ \mathit{Zag}\ :+:_M\ U)\ :\triangleright:_M\ \mathit{Zig})\\ &\qquad :+:_M\ ((I_M\ \mathit{Zig})\qquad :\triangleright:_M\ \mathit{Zag})\end{aligned}$$

The `multirec` library encodes indices by using the datatype itself as an index. As such, in our example above, the index κ is $\star$. This turns out to be convenient for our conversion, so we will always use Un_M instantiated to kind $\star$.

The interpretation of the `multirec` universe is parametrised not only by the representation type α, but also by a type constructor τ that converts indices into their concrete representation, and a particular index type ι:

$$\begin{aligned}&\textbf{data}\ [\![\, \alpha :: \mathit{Un}_M\ \kappa]\!]_M\ (\tau :: \kappa \to \star)\ (\iota :: \kappa)\ \textbf{where}\\ &\quad U_M \quad :: [\![\, U]\!]_M\ \tau\ \iota\\ &\quad I_M \quad :: \tau\ o \to [\![\, I_M\ o]\!]_M\ \ \tau\ \iota\\ &\quad K_M \quad :: \alpha\ \ \to [\![\, K_M\ \alpha]\!]_M\ \tau\ \iota\\ &\quad \mathit{Tag}_M :: [\![\, \alpha]\!]_M\ \tau\ \iota \to [\![\, \alpha\ :\triangleright:_M\ \iota]\!]_M\ \tau\ \iota\\ &\quad L_M \quad :: [\![\, \alpha]\!]_M\ \tau\ \iota \to [\![\, \alpha\ :+:_M\ \beta]\!]_M\ \tau\ \iota\\ &\quad R_M \quad :: [\![\, \beta]\!]_M\ \tau\ \iota \to [\![\, \alpha\ :+:_M\ \beta]\!]_M\ \tau\ \iota\\ &\quad :\times:_M :: [\![\, \alpha]\!]_M\ \tau\ \iota \to [\![\, \beta]\!]_M\ \tau\ \iota \to [\![\, \alpha\ :\times:_M\ \beta]\!]\ \tau\ \iota\end{aligned}$$

In other words, the interpretation $[\![\, \alpha]\!]_M\ \tau\ \iota$ can be seen as a family of datatypes, one for each particular index ι. The Tag_M constructor introduces a type equality constraint on the tagged index; this is how the interpretation is restricted to a particular index.

Finally, user datatypes are converted to the `multirec` representation using two type classes, Fam_M and El_M:

$$\begin{aligned}&\textbf{newtype}\ I_{0M}\ \alpha = I_{0M}\ \alpha\\ &\textbf{class}\ \mathit{Fam}_M\ (\phi :: \star \to \star)\ \textbf{where}\\ &\quad \mathit{PF}_M\ \phi :: \mathit{Un}_M\ \star\\ &\quad \mathit{from}_M :: \phi\ \iota \to \iota \to [\![\, \mathit{PF}_M\ \phi]\!]_M\ I_{0M}\ \iota\\ &\textbf{class}\ \mathit{El}_M\ (\phi :: \kappa \to \star)\ (\iota :: \kappa)\ \textbf{where}\\ &\quad \mathit{proof}_M :: \phi\ \iota\end{aligned}$$

The class Fam_M takes as argument a *family* type ϕ. Here we instantiate the τ in $[\![\, \_]\!]_M$ to an identity type I_{0M}; other applications in `multirec`, such as the generalised catamorphism, make use of the generality of τ. The El_M class associates each index type ι with its family ϕ.

This is all best understood through an example, so we show the encoding for the family of datatypes *Zig* and *Zag* shown before. The first step is to define a GADT to represent the family. This datatype contains elements of either type *Zig* or *Zag*:

$$\begin{array}{l}\textbf{data}\ ZigZag\ \iota\ \textbf{where}\\ \quad ZigZag_{Zig} :: ZigZag\ Zig\\ \quad ZigZag_{Zag} :: ZigZag\ Zag\end{array}$$

The type *ZigZag* now describes our family, by providing two indices $ZigZag_{Zig}$ and $ZigZag_{Zag}$. This is made concrete by the following instances:

$$\begin{array}{l}\textbf{instance}\ Fam_M\ ZigZag\ \textbf{where}\\ \quad PF_M\ ZigZag = ZigZagRep\\ \quad from_M\ ZigZag_{Zig}\ (Zig\ z)\ = L_M\ (Tag_M\ (L_M\ (I_M\ (I_{0M}\ z))))\\ \quad from_M\ ZigZag_{Zig}\ ZigEnd = L_M\ (Tag_M\ (R_M\ U_M))\\ \quad from_M\ ZigZag_{Zag}\ (Zag\ z) = R_M\ (Tag_M\ (I_M\ (I_{0M}\ z)))\end{array}$$

$$\begin{array}{l}\textbf{instance}\ El_M\ ZigZag\ Zig\ \ \textbf{where}\ proof_M = ZigZag_{Zig}\\ \textbf{instance}\ El_M\ ZigZag\ Zag\ \textbf{where}\ proof_M = ZigZag_{Zag}\end{array}$$

Type Conversion. The first step in converting a family of datatypes representable in `generic-deriving` to `multirec` is to convert a single datatype. This is the task of the $D_{\rightarrow}M$ type family:

$$\begin{array}{ll}D_{\rightarrow}M\ (\alpha :: Un_D) :: Un_M\ \star & \\ D_{\rightarrow}M\ U_D & = U_M\\ D_{\rightarrow}M\ (M_D\ \iota\ \alpha) & = D_{\rightarrow}M\ \alpha\\ D_{\rightarrow}M\ (\alpha\ :+:_D\ \beta) & = D_{\rightarrow}M\ \alpha\ :+:_M\ D_{\rightarrow}M\ \beta\\ D_{\rightarrow}M\ (\alpha\ :\times:_D\ \beta) & = D_{\rightarrow}M\ \alpha\ :\times:_M\ D_{\rightarrow}M\ \beta\end{array}$$

The most interesting case is that for constants, which we now need either to turn into indices, or to keep as constants. We turn recursive occurrences into indices, and leave the rest as constants:

$$\begin{array}{l}D_{\rightarrow}M\ (K_D\ (R\ \iota)\ \tau) = I_M\ \ \tau\\ D_{\rightarrow}M\ (K_D\ U\ \quad\ \alpha) = K_M\ \alpha\\ D_{\rightarrow}M\ (K_D\ P\ \quad\ \alpha) = K_M\ \alpha\end{array}$$

Having defined $D_{\rightarrow}M$ to convert one datatype, we are left with the task of converting a *family* of datatypes. We encode a family as a type-level list of datatypes, and define $D_{\rightarrow}M_{Fam}$ parametrised over such a list:

$$\begin{array}{ll}\textbf{data}\ \bot & \\ D_{\rightarrow}M_{Fam}\ (\alpha :: [\star]) :: Un_M\ \star & \\ D_{\rightarrow}M_{Fam}\ [\,] & = K_M\ \bot\\ D_{\rightarrow}M_{Fam}\ (\alpha : \beta) & = (D_{\rightarrow}M\ (Rep_D\ \alpha))\ :\triangleright:_M\ \alpha)\ :+:_M\ D_{\rightarrow}M_{Fam}\ \beta\end{array}$$

We convert a list of datatypes by taking each element, looking up its representation in `generic-deriving` using Rep_D, converting it to a `multirec` representation using $D_{\rightarrow}M$, and tagging that with the original datatype. The base case is the empty list, which we encode with an empty representation (since `multirec` has no empty representation type, we define an empty datatype $\perp$ and use it as a constant).

Value Conversion. Converting a value of a single type is done in exactly the same way as for the `regular` conversion:

$$\begin{array}{l}\textbf{class}\ \mathit{Convert}_{D\rightarrow M}\ (\alpha :: \mathit{Un}_D)\ \textbf{where} \\ \quad d_{\rightarrow}m :: [\![\,\alpha\,]\!]_D \rightarrow [\![D_{\rightarrow}M\ \alpha\,]\!]_M\ I_{0M}\ \iota\end{array}$$

As before, we omit the instances, as they are without surprises.

We're left with dealing with the encapsulation of values within a family. We represent families as lists of types, but a value of a family is still of a single, concrete type. We use a GADT to encode the notion of a value within a family:

$$\begin{array}{l}\textbf{data}\ (\alpha :: [\star])\ {:}@{:}\ (\beta :: \star)\ \textbf{where} \\ \quad \mathit{This} :: \qquad\qquad\qquad (\alpha : \beta)\ {:}@{:}\ \alpha \\ \quad \mathit{That} :: \beta\ {:}@{:}\ \alpha \rightarrow (\gamma : \beta)\ {:}@{:}\ \alpha\end{array}$$

For example, the value *This ZigEnd* has the type $[\mathit{Zig},\mathit{Zag}]\ {:}@{:}\ \mathit{Zig}$, and the value $\mathit{That}\ (\mathit{This}\ (\mathit{Zag}\ \mathit{ZigEnd}))$ has the type $[\mathit{Zig},\mathit{Zag}]\ {:}@{:}\ \mathit{Zag}$.

The application of ${:}@{:}$ to a single argument is of kind $\star \rightarrow \star$, and it encodes precisely the notion of a `multirec` family. We make this explicit by providing El_M instances stating that a type α is either at the head of the list, and can be accessed with *This*, or it might be deeper within the list, in which case we have to continue indexing with *That*:

$$\begin{array}{l}\textbf{instance}\qquad\qquad\qquad\qquad\quad El_M\ ((\alpha : \beta)\ {:}@{:})\ \alpha\ \textbf{where}\ \mathit{proof}_M = \mathit{This} \\ \textbf{instance}\ (El_M\ (\beta\ {:}@{:})\ \alpha) \Rightarrow El_M\ ((\gamma : \beta)\ {:}@{:})\ \alpha\ \textbf{where}\ \mathit{proof}_M = \mathit{That}\ \mathit{proof}_M\end{array}$$

Converting a value within a family requires producing the appropriate injection into the right element of the family, plus the tag (with Tag_M). We use our ${:}@{:}$ GADT for this (which results in a right-biased encoding of the family):

$$\begin{array}{l}\textbf{instance}\ (\mathit{FamConstrs}\ \alpha) \Rightarrow \mathit{Fam}_M\ (\alpha\ {:}@{:})\ \textbf{where} \\ \quad PF_M\ (\alpha\ {:}@{:}) = D_{\rightarrow}M_{\mathit{Fam}}\ \alpha \\ \quad \mathit{from}_M\ \mathit{This}\qquad x = L_M\ (\mathit{Tag}_M\ (d_{\rightarrow}m\ (\mathit{from}_D\ x))) \\ \quad \mathit{from}_M\ (\mathit{That}\ k)\ x = R_M\ (\mathit{from}_M\ k\ x)\end{array}$$

The constraints on this instance are not trivial, as each type in the family needs to have a $\mathit{Generic}_D$ instance and be convertible through $\mathit{Convert}_{D\rightarrow M}$. The *FamConstrs* constraint family expresses these requirements:

$$\begin{array}{l}\mathit{FamConstrs}\ (\alpha :: [\star]) :: \mathit{Constraint} \\ \mathit{FamConstrs}\ [\,]\qquad\ = () \\ \mathit{FamConstrs}\ (\alpha : \beta) = (\ \mathit{Generic}_D\ \alpha, \mathit{Convert}_{D\rightarrow M}\ (\mathit{Rep}_D\ \alpha) \\ \qquad\qquad\qquad\qquad\quad ,\ \mathit{Fam}_M\ (\beta\ {:}@{:}), \mathit{FamConstrs}\ \beta)\end{array}$$

Example. To test this conversion, assume we have some generic function $size_M$ defined in `multirec` which computes the size of a term. Assume we also have $Generic_D$ instances for the *Zig* and *Zag* types in `generic-deriving` (derived by the compiler). These give rise to a Fam_M ($[Zig, Zag]$:@:) instance (this section). As such, we can call $size_M$ directly on a value of type *Zig*:

$$size_M :: (Fam_M\ \phi, \ldots) \Rightarrow \phi\ \iota \rightarrow \iota \rightarrow Int$$
$$size_M = \ldots$$
$$zigZag :: Zig$$
$$zigZag = Zig\ (Zag\ (Zig\ (Zag\ ZigEnd)))$$
$$test_{d \rightarrow m} :: Int$$
$$test_{d \rightarrow m} = size_M\ (proof :: [Zag, Zig]\ {:}@{:}\ Zig)\ zigZag$$

Our test value $test_{d \rightarrow m}$ evaluates to 4 as expected. The use of :@: makes `multirec` easier to use than before; unlike in our example in Section 4, it is not necessary to define a family type; we can simply use :@:. The index (first argument to $size_M$) is automatically computed from the type signature of *proof*, so there is no need to explicitly use *This* and *That*. Finally, families can be easily extended: the code for $test_{d \rightarrow m}$ works equally well if we supply *proof* as having type $[Zag, Zig, Int]$:@: *Zig*, for instance.

5 From `generic-deriving` to `syb`

The `syb` library, unlike the others we have seen so far, does not encode the structure of user datatypes at the type level. Instead, it views data as successive applications of terms; generic functions then operate on this applicative structure. The interface presented to the user hides this view, and is instead based on various traversal operators. In this section we show how to obtain `syb` representations of data from `generic-deriving`. We use the `syb` encoding of Hinze et al. [3] as the basis of our development instead of the "official" encoding shipped with GHC, but this does not make our conversion any less applicable or general.

Encoding `syb`**.** The basis of `syb` is the *Spine* datatype, which defines a view on data as a sequence of applications. A value of type *Spine* is either a constructor, or an application of a *Spine* with functional type to an argument:

$$\textbf{data}\ Spine :: \star \rightarrow \star\ \textbf{where}$$
$$Con :: \alpha \rightarrow Spine\ \alpha$$
$$(:\diamond:) :: (Data\ \alpha) \Rightarrow Spine\ (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow Spine\ \beta$$

The *Data* constraint will be explained later.

The *Spine* datatype is both *Functor*ial and *Applicative*, and we can also *fold* it:

$$\textbf{instance}\ Functor\ Spine\ \textbf{where}$$
$$fmap\ f\ (Con\ x) = Con\ (f\ x)$$
$$fmap\ f\ (c :\diamond: x) = fmap\ (f \circ)\ c :\diamond: x$$
$$\textbf{instance}\ Applicative\ Spine\ \textbf{where}$$
$$pure = Con$$

$$\begin{aligned}
&Con\ f \quad <\!*\!> x \qquad\quad = fmap\ f\ x\\
&(c :\diamond: x) <\!*\!> Con\ y \quad = fmap\ (\lambda f\ x \to f\ x\ y)\ c :\diamond: x\\
&(c :\diamond: x) <\!*\!> (d :\diamond: y) = (fmap\ (\lambda f\ d\ y \to f\ (d\ y))\ (c :\diamond: x) <\!*\!> d) :\diamond: y
\end{aligned}$$

$$\begin{aligned}
&foldSpine :: (\forall \alpha\ \beta . Data\ \alpha \Rightarrow \phi\ (\alpha \to \beta) \to \alpha \to \phi\ \beta)\\
&\qquad\qquad \to (\forall \alpha . \alpha \to \phi\ \alpha) \to Spine\ \alpha \to \phi\ \alpha\\
&foldSpine\ f\ z\ (Con\ c) = z\ c\\
&foldSpine\ f\ z\ (c :\diamond: x) = foldSpine\ f\ z\ c\ `f`\ x
\end{aligned}$$

Although the type of *foldSpine* might look intimidating at first, its first argument is simply the replacement for the $:\diamond:$ constructor, and the second is the replacement for *Con*.

The *Data* class is used to embed conversions between user datatypes and the *Spine* generic view:

$$\begin{aligned}
&\textbf{class}\ (Typeable\ \alpha) \Rightarrow Data\ \alpha\ \textbf{where}\\
&\quad spine \ :: \ \alpha \to Spine\ \alpha\\
&\quad gfoldl :: (\forall \gamma\ \beta . Data\ \gamma \Rightarrow \phi\ (\gamma \to \beta) \to \gamma \to \phi\ \beta)\\
&\qquad\qquad \to (\forall \beta . \beta \to \phi\ \beta) \to \alpha \to \phi\ \alpha\\
&\quad gfoldl\ f\ z = foldSpine\ f\ z \circ spine
\end{aligned}$$

The *Data* class has *Typeable* as a superclass for convenience, because many generic functions in `syb` make use of type-safe runtime cast. The *gfoldl* method is the basis of all generic consumer functions in `syb`, and we see that it is just a variant of *foldSpine*.

The way `syb` is implemented in GHC, *gfoldl* is a primitive, and its definition is automatically generated by the compiler for user datatypes using the **deriving** mechanism. In our presentation, the *spine* method is the primitive, from which *gfoldl* follows.

The encoding of user datatypes in `syb` using *Spine* is very simple. As an example, here is the encoding of lists:

$$\begin{aligned}
&\textbf{instance}\ (Data\ \alpha) \Rightarrow Data\ [\alpha]\ \textbf{where}\\
&\quad spine\ [\,] \qquad = Con\ [\,]\\
&\quad spine\ (h : t) = Con\ (:) :\diamond: h :\diamond: t
\end{aligned}$$

Base types are encoded trivially:

$$\textbf{instance}\ Data\ Int\ \textbf{where}\ spine = Con$$

We show a simplified version of `syb`, in particular omitting meta-information and the *gunfold* function. These are cosmetic simplifications only; Hinze et al. [3] describe how to support meta-information in the *Spine* view, and Hinze and Löh [2] describe how to define *gunfold*.

Value Conversion. To convert the generic representation of `generic-deriving` into that of `syb` we only need to convert values, as `syb` has no type-level representation. As such, we require only a type class:

$$\begin{aligned}
&\textbf{class}\ Convert_{D \to S}\ (\alpha :: Un_D)\ \textbf{where}\\
&\quad d_{\to} s :: [\![\alpha]\!]_D \to Spine\ ([\![\alpha]\!]_D)
\end{aligned}$$

The idea is to first build a representation of type *Spine* $(\llbracket \alpha \rrbracket_D)$, and later transform this into *Spine* α. The instances are unsurprising, and follow the functorial nature of *Spine*:

$$
\begin{array}{l}
\textbf{instance } \mathit{Convert}_{D\to S}\ U_D \textbf{ where } d_{\to}s\ U_{1D} = \mathit{Con}\ U_{1D} \\
\textbf{instance } (\mathit{Convert}_{D\to S}\ \alpha, \mathit{Convert}_{D\to S}\ \beta) \Rightarrow \mathit{Convert}_{D\to S}\ (\alpha\ :+:_D\ \beta) \textbf{ where} \\
\quad d_{\to}s\ (L_{1D}\ x) = \mathit{fmap}\ L_{1D}\ (d_{\to}s\ x) \\
\quad d_{\to}s\ (R_{1D}\ x) = \mathit{fmap}\ R_{1D}\ (d_{\to}s\ x) \\
\textbf{instance } (\mathit{Convert}_{D\to S}\ \alpha, \mathit{Convert}_{D\to S}\ \beta) \Rightarrow \mathit{Convert}_{D\to S}\ (\alpha\ :\times:_D\ \beta) \textbf{ where} \\
\quad d_{\to}s\ (x\ :\times:_D\ y) = \mathit{pure}\ (:\times:_D)\ <\!\!*\!\!>\ d_{\to}s\ x\ <\!\!*\!\!>\ d_{\to}s\ y \\
\textbf{instance } (\mathit{Data}\ \alpha) \Rightarrow \mathit{Convert}_{D\to S}\ (K_D\ \iota\ \alpha) \textbf{ where} \\
\quad d_{\to}s\ (K_{1D}\ x) = \mathit{Con}\ K_{1D}\ :\diamond:\ x \\
\textbf{instance } (\mathit{Convert}_{D\to S}\ \alpha) \Rightarrow \mathit{Convert}_{D\to S}\ (M_D\ \iota\ \alpha) \textbf{ where} \\
\quad d_{\to}s\ (M_{1D}\ x) = \mathit{fmap}\ M_{1D}\ (d_{\to}s\ x)
\end{array}
$$

With these instances in place, we can define a *Data* instance for all $\mathit{Generic}_D$ types:

$$
\begin{array}{l}
\textbf{instance } (\mathit{Generic}_D\ \alpha, \mathit{Convert}_{D\to S}\ (\mathit{Rep}_D\ \alpha), \mathit{Typeable}\ \alpha) \Rightarrow \mathit{Data}\ \alpha \textbf{ where} \\
\quad \mathit{spine} = \mathit{fmap}\ \mathit{to}_D \circ d_{\to}s \circ \mathit{from}_D
\end{array}
$$

We first convert the user type to its `generic-deriving` representation with from_D, then build a *Spine* representation using $d_{\to}s$, and finally adapt this representation with $\mathit{fmap}\ \mathit{to}_D$.

To test our conversion, assume that we had *not* given the $\mathit{Data}\ [\alpha]$ instance earlier in this section; the $\mathit{Generic}_D\ [\alpha]$ instance of Section 2 would then cascade down into a $\mathit{Data}\ [\alpha]$ instance using the conversion defined in this section. Assuming also generic functions *everywhere* (to apply a transformation to all subterms) and *mkT* (to transform a type-specific query into a generic query), as defined in `syb`, the expression $\mathit{everywhere}\ (\mathit{mkT}\ (\lambda n \to n+1 :: \mathit{Int}))\ [1,2,3 :: \mathit{Int}]$ evaluates to $[2,3,4]$, as expected, *without* ever having to derive *Data* instances directly.

The code defined in this section, albeit straightforward, allows GHC developers to scrap the current code for deriving *Data* instances, as these can be obtained automatically from $\mathit{Generic}_D$ instances (which are currently derivable in GHC). Furthermore, it brings the combinator-style approach to GP of `syb` within immediate reach of the other approaches. It is also worth nothing that `uniplate`, another GP library, can derive its encodings from `syb` [11, Section 5.3]; therefore, by transitivity, we can also provide `uniplate` encodings from `generic-deriving`.

6 Discussion and Conclusion

We conclude this paper with a review of related work, and a discussion of concerns regarding the pratical implementation of the conversions as shown in the paper.

Related Work. We have defined conversions between GP approaches before, in Agda [9]. Those conversions were of a more theoretical nature, as the intention was to formally compare approaches. Furthermore, `generic-deriving` was not involved. Our work can be seen as providing conversions between views. In particular, while the

Generic Haskell compiler had generic views defined internally, whose adaptation required changing the compiler itself [4, Section 5], our work allows new views to be defined simply by writing a new universe and interpretation together with a conversion (as in Section 3).

Other approaches to providing functionality mixing different views have been attempted. Chakravarty et al. [1] mention support for multiple views, but do this through duplication of the universe, interpretation, and datatype representations. The Hackage pages `instant-zipper` and `generic-deriving-extras` provide functionality usually associated with a fixed-point view on a library without such a view, respectively, a zipper in `instant-generics`, and a fold in `generic-deriving`. This is achieved by extending the non fixed-point view libraries, rather than by converting between representations, as we do.

Performance. One aspect that we have not addressed in this paper is the potential performance penalty that the conversions might bring. We find it very likely that such an overhead exists, given that the conversions are not trivial. However, we also believe that this overhead should be fully removable by the compiler, using techniques similar to those described by Magalhães [8]. Performance concerns are relevant, as these are crucial for user adoption of our conversions. However, optimisation concerns often result in cumbersome code where the original idea is obscured. As such, we preferred to focus on presenting the conversions and their potential applications, and defer performance concerns to future work.

Practical Implementation. Performance concerns are just one of the aspects to consider when deciding how to best integrate our conversions with the existing GP libraries. While we have tried to remain faithful to the original libraries in our encoding, a few modifications to the way `generic-deriving` handles the tags in K_D and Rec_D were necessary to support the conversion to `multirec`. These changes, besides being minor, actually improve `generic-deriving`, as the current implementation is rather ill-defined with respect to which tag is used when. Furthermore, we know of no generic function currently relying on these tags; our conversion in Section 4 might be the first example.

We have used datatype promotion in all approaches, and encode meta-information at the type level, instead of using type classes. These changes are not backwards compatible because the current implementation of datatype promotion requires choosing different names for a representation type (e.g. U_R) and its interpretation (also named U_R), while these are often the same in the current implementations of the libraries. While the implementation of datatype promotion might change to allow avoiding name clashes,it might be preferable to have a new release for each library that breaks backwards compatibility, requires GHC $\geqslant$ 7.6, but homogenises naming conventions and meta-data representation across libraries, for instance. Alternatively, we could introduce a new library, intended to sit at the top of the hierarchy, from which all other conversions could be derived. This library would not be intended for direct use, allowing it to be easily adapted to support new libraries. This would further enhance the new approach to GP in Haskell that we advocate: a particular library is just a particular way to *view* data, and all libraries interplay seamlessly because they all share a common root.

Conclusion. In the past, there was a lot of apparent competition between different approaches to GP. While it is reasonably easy to use Template Haskell to derive the encodings of the datatypes needed to use a particular library, most users seemed to prefer the libraries that had direct support within GHC, such as `syb` or `generic-deriving`. On the other hand, users had a difficult decision to make, operating under the assumption that they have to pick a single library among the many that are available, perhaps afraid to make the wrong choice and to then stumble upon a programming problem that cannot easily be solved using the chosen library.

Those times are over. GP library authors no longer have to feel embarrassed if they present a new library suitable only for a specific class of GP programming problems. All they need to do is to define a conversion, and their library will be integrated better than ever before, without any need for Template Haskell. Users should no longer worry that they have to make a particular choice. All GP libraries interact nicely, and they can simply pick the one that offers the functionality they need right now—we have arrived in the era of truly generic generic programming!

Acknowledgements. The first author is supported by the EP/J010995/1 EPSRC grant. We thank Nicolas Wu for suggesting the title of this paper, and Sean Leather for encouraging us to include `multirec` in our considerations. Jeremy Gibbons, Ralf Hinze, Johan Jeuring, Sean Leather, the members of the reading club at Utrecht University, and anonymous reviewers provided helpful feedback on an earlier draft of this paper.

References

[1] Chakravarty, M.M.T., Ditu, G.C., Leshchinskiy, R.: Instant generics: Fast and easy (2009), http://www.cse.unsw.edu.au/~chak/papers/CDL09.html

[2] Hinze, R., Löh, A.: "Scrap Your Boilerplate" revolutions. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 180–208. Springer, Heidelberg (2006), doi:10.1007/11783596_13

[3] Hinze, R., Löh, A., Oliveira, B.C.d.S.: "Scrap Your Boilerplate" reloaded. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 13–29. Springer, Heidelberg (2006), doi:10.1007/11737414_3

[4] Holdermans, S., Jeuring, J., Löh, A., Rodriguez Yakushev, A.: Generic views on data types. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 209–234. Springer, Heidelberg (2006), doi:10.1007/11783596_14

[5] Lämmel, R., Peyton Jones, S.: 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, pp. 26–37. ACM (2003), doi:10.1145/604174.604179

[6] Lämmel, R., Peyton Jones, S.: Scrap more boilerplate: reflection, zips, and generalised casts. In: Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming, pp. 244–255. ACM (2004), doi:10.1145/1016850.1016883

[7] Magalhães, J.P.: Less Is More: Generic Programming Theory and Practice. PhD thesis, Universiteit Utrecht (2012)

[8] Magalhães, J.P.: Optimisation of generic programs through inlining. Accepted for publication at the 24th Symposium on Implementation and Application of Functional Languages, IFL 2012 (2013)

[9] Magalhães, J.P., Löh, A.: A formal comparison of approaches to datatype-generic programming. In: Chapman, J., Levy, P.B. (eds.) Proceedings Fourth Workshop on Mathematically Structured Functional Programming. Electronic Proceedings in Theoretical Computer Science, vol. 76, pp. 50–67. Open Publishing Association (2012), doi:10.4204/EPTCS.76.6

[10] Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for Haskell. In: Proceedings of the 3rd ACM Haskell Symposium on Haskell, pp. 37–48. ACM (2010), doi:10.1145/1863523.1863529

[11] Mitchell, N., Runciman, C.: Uniform boilerplate and list processing. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 49–60. ACM (2007), doi:10.1145/1291201.1291208

[12] Morris, P.: Constructing Universes for Generic Programming. PhD thesis, The University of Nottingham (November 2007)

[13] Van Noort, T., Rodriguez Yakushev, A., Holdermans, S., Jeuring, J., Heeren, B.: A lightweight approach to datatype-generic rewriting. In: Proceedings of the ACM SIGPLAN Workshop on Generic Programming, pp. 13–24. ACM (2008), doi:10.1145/1411318.1411321

[14] Rodriguez Yakushev, A., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, pp. 233–244. ACM (2009), doi:10.1145/1596550.1596585

[15] Schrijvers, T., Peyton Jones, S., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 51–62. ACM (2008), doi:10.1145/1411204.1411215

[16] Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, pp. 341–352. ACM (2009), doi:10.1145/1596550.1596599

[17] Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, vol. 37, pp. 1–16. ACM (December 2002), doi:10.1145/581690.581691

[18] Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a promotion. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, pp. 53–66. ACM (2012), doi:10.1145/2103786.2103795