Generic Generic Programming

José Pedro Magalhães
Department of Computer Science, University of Oxford
jpm@cs.ox.ac.uk

Andres Löh
Well-Typed LLP
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 very similar, but have minor variations that make them particularly well-suited for one particular 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 introduce yet another library for GP in Haskell... from which we can automatically derive representations for the most popular other GP libraries. 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.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

Keywords datatype-generic programming, Haskell, SYB

1. Introduction

The abundance of generic programming approaches is not a new problem. Including pre-processors, template-based approaches, language extensions, and libraries, there are well over 15 different approaches to generic programming in Haskell (Magalhães 2012, 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 (Holdermans et al. 2006) (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 first they have to choose a library that is appropriate for their needs.

Some effort has been made in comparing different approaches to GP from a practical point of view (Hinze et al. 2007; Rodriguez Yakushev et al. 2008), or to classify approaches (Hinze and Löh 2009). 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. 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 in representation. A conversion between approaches provides the means to remove duplication of generic code.

In this paper we define a new GP library, `structured`, and use it to derive representations for many 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 our 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 (e.g. through the deriving mechanism); support for other libraries follows automatically from conversions that are defined in plain Haskell, not through more compiler extensions. 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.

We show how `structured` can handle multiple generic views with minimal encoding repetition, and then define a conversion to one of the standard modern GP libraries in Haskell, `generic-deriving` (Magalhães et al. 2010). From there we show conversions to other popular generic libraries: `regular` (Van Noort et al. 2008), `multirec` (Rodriguez Yakushev et al. 2009), and `syb` (Lämmel and Peyton Jones 2003, 2004).[1] Some of these libraries are remarkably different from each other, yet advanced type-level features in the Glasgow Haskell Compiler (GHC),[2] such as GADTs (Schrijvers et al. 2009), type functions (Schrijvers et al. 2008), and kind polymorphism (Yorgey et al. 2012), 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
import Generics.Regular.Functions.Fold as R
import Generics.SYB.Schemes            as S
import Data.Typeable
```

[1] We also have a conversion to `instant-generics` (Chakravarty et al. 2009) which we omit from the paper as it offers no new insights.

[2] `http://www.haskell.org/ghc/`

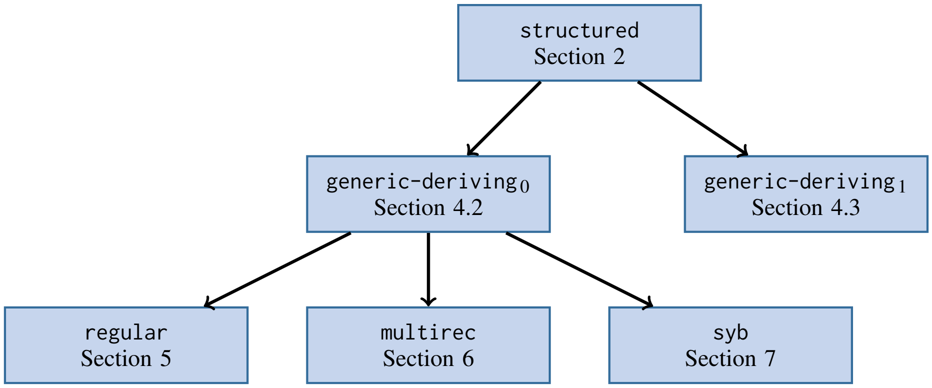

Figure 1. Conversions between the approaches.

```
import Conversions ()
data Logic = Logic :∧: Logic | Logic :∨: Logic
           | Not Logic | T | F
  deriving (Generic, Typeable)
test :: (Bool, Int)
test = (R.fold alg term, S.gsize term)
  where term = T :∨: F
        alg  = (∧) & (∨) & not & True & False
```

Here, the user defines a *Logic* datatype, and lets the compiler automatically derive a *Generic* representation for it. The *fold* function, from the `regular` library, and the *gsize* function, from `syb`, can then be used on *Logic* values, simply by importing the conversion instances defined in some module *Conversions*; there is no need to derive any generic representations for `regular` or `syb`.[3]

Generic library writers also see an improvement in their quality of life, as they no longer need to write Template Haskell (Sheard and Peyton Jones 2002) 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, 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 generic function writers 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 define conversions to multiple other GP libraries (Sections 4 to 7). We cover 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 update their definitions to make use of the latest GHC extensions (namely data kinds and kind polymorphism (Yorgey et al. 2012)). 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.[4]

Figure 1 shows a diagram with an overview of the conversions defined in this paper.

1.1 Notation

In order to avoid syntactic clutter and to help the reader, we adopt a liberal Haskell notation in this paper. We will 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 will 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 7.6.2, 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.

1.2 Structure of the paper

The remainder of this paper is structured as follows. We first introduce the `structured` library for GP (Section 2). We then see how how to obtain views with different balancings of the constructors and constructor arguments (Section 3). Afterwards, we see how to obtain many other libraries from `structured`; we start with `generic-deriving` (Section 4), one of the libraries currently bundled with GHC. From `generic-deriving` we see how to obtain `regular` (Section 5), `multirec` (Section 6), and `syb` (Section 7). We then conclude with a discussion in Section 8. No previous knowledge of any of the libraries is required, since we will understand them all in terms of `structured`. Along the way, we also

[3] We also derive *Typeable* because `syb` requires it. Note that the *Typeable* class only provides functionality related to runtime type comparison and casting; it is not a GP library, so it is not included in our conversions.

[4] While these libraries were always "type correct", our changes make them "more kind correct" as well.

show several examples of how our conversion enables seamless use of multiple approaches.

2. A highly structured library

We begin our efforts of homogenising GP libraries by defining a `structured` library intended to sit at the top of the hierarchy. Our goal is to define a library that is highly expressive, even if not entirely convenient to use. Users who require the level of detail given by `structured` are free to use it directly, but we expect most users to prefer using any of the other, already existing GP libraries. Usability is not our main 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.

This new approach is at the core of all other approaches, but users (and even generic function writers) need not be aware of that. In particular, if this library is supported by automatic deriving of representations in the compiler, 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).

2.1 Universe

The structure used to encode datatypes in a GP programming approach is called its *universe* (Morris 2007). The universe of `structured` 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 $Either\ \alpha\ \beta$, 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 on `generic-deriving` with regards to datatype parameters, but that is secondary to our goal in this paper. 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: in Haskell, only types of kind $\star$ are inhabited by values. These kinds can be thought of as datatypes, but its "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 7.6.2) 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 | 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 $\star$ ($K\ R$), some other type (such as an application of type variable, encoded with $K\ U$), a datatype of kind (at least) $\star \to \star$ (*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 annotations given by *KType* and *RecType* will prove essential when converting to approaches with a fixed-point view on data (Section 5 and Section 6), as there we need explicit knowledge about the recursive structure of data.

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

```
data D φ α β = D1 Int (φ α) | D2 [D φ α β] β
```

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*:

```
A11 = K (R O) Int
A12 = K U (φ α)
A21 = [] :◦: Rec S (D φ α)
A22 = Par
```

The entire representation consists of wrapping of appropriate metadata around the representation for constructor arguments:

```
RepD φ α β =
  Data (MD "D" "Module" False)
    (Bin (Leaf (Con (MC "D1" Prefix False)
                 (Bin (Leaf (Field (MF Nothing) A11))
                      (Leaf (Field (MF Nothing) A12)))))
```

```
(Leaf (Con (MC "D2" Prefix False)
      (Bin (Leaf (Field (MF Nothing) A21))
           (Leaf (Field (MF Nothing) A22))))))
```

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`:

```
data family ⟦_⟧ :: κ → ⋆ → ⋆
```

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 on 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 (:×:):

```
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
  Par_g α :: ⋆
  Par_g α = NoPar
  from :: α → ⟦Rep φ⟧ (Par_g α)
  to   :: ⟦Rep φ⟧ (Par_g α) → α

data NoPar    -- empty
```

In the *Generic* class, the type family *Rep* encodes the generic representation associated with user datatype α, and Par_g[5] 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 incorporate in the compiler, allowing automatic derivation of *Generic* instances.)

2.4.1 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 Par_g. 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.

2.4.2 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))))
  Par_g [α] = α
  from []     = D1 (L1 (C1 U1))
```

[5] The subscript g is only to distinguish Par_g from the universe type *Par*.

$$from\ (h : t) = D_1\ (R_1\ (C_1\ (S_1\ (Par\ h) :\times: S_1\ (Rec\ t))))$$
$$to\ (D_1\ (L_1\ (C_1\ U_1))) = [\,]$$
$$to\ (D_1\ (R_1\ (C_1\ (S_1\ (Par\ h) :\times: S_1\ (Rec\ t))))) = h : t$$

The type function Par_g 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. We use a type family to adapt the representation, and a type-class to adapt the values.

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:

$$RotR\ (\alpha :: Tree\ \kappa) :: Tree\ \kappa$$
$$RotR\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma) = Bin\ \alpha\ (Bin\ \beta\ \gamma)$$
$$RotR\ (Bin\ (Leaf\ \alpha)\ \gamma) = Bin\ (Leaf\ \alpha)\ \gamma$$

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:

$$S_{\rightarrow}SR_d\ (\alpha :: Data) :: Data$$
$$S_{\rightarrow}SR_d\ (Data\ \iota\ \alpha) = Data\ \iota\ (S_{\rightarrow}SR_{cs}\ \alpha)$$

$$S_{\rightarrow}SR_{cs}\ (\alpha :: Tree\ Con) :: Tree\ Con$$
$$S_{\rightarrow}SR_{cs}\ Empty = Empty$$
$$S_{\rightarrow}SR_{cs}\ (Leaf\ (Con\ \iota\ \gamma)) = Leaf\ (Con\ \iota\ (S_{\rightarrow}SR_{fs}\ \gamma))$$
$$S_{\rightarrow}SR_{cs}\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma) = S_{\rightarrow}SR_{cs}\ (RotR\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma))$$
$$S_{\rightarrow}SR_{cs}\ (Bin\ (Leaf\ \alpha)\ \gamma) = Bin\ (S_{\rightarrow}SR_{cs}\ (Leaf\ \alpha))\ (S_{\rightarrow}SR_{cs}\ \gamma)$$

$$S_{\rightarrow}SR_{fs}\ (\alpha :: Tree\ Field) :: Tree\ Field$$
$$S_{\rightarrow}SR_{fs}\ Empty = Empty$$
$$S_{\rightarrow}SR_{fs}\ (Leaf\ \gamma) = Leaf\ \gamma$$
$$S_{\rightarrow}SR_{fs}\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma) = S_{\rightarrow}SR_{fs}\ (RotR\ (Bin\ (Bin\ \alpha\ \beta)\ \gamma))$$
$$S_{\rightarrow}SR_{fs}\ (Bin\ (Leaf\ \alpha)\ \gamma) = Bin\ (Leaf\ \alpha)\ (S_{\rightarrow}SR_{fs}\ \gamma)$$

The conversion for constructors ($S_{\rightarrow}SR_{cs}$) and selectors ($S_{\rightarrow}SR_{fs}$) 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:

$$\textbf{class}\ Convert_{S\rightarrow SR}\ (\alpha :: Data)\ \textbf{where}$$
$$\quad s_{\rightarrow}rs :: [\![\alpha]\!]\ \rho \rightarrow [\![S_{\rightarrow}SR_d\ \alpha]\!]\ \rho$$
$$\quad s_{\leftarrow}rs :: [\![S_{\rightarrow}SR_d\ \alpha]\!]\ \rho \rightarrow [\![\alpha]\!]\ \rho$$

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:

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

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

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

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

$$testCountSums :: ([Int],[Int])$$
$$testCountSums = (map\ countSumsBal\ [A,B,C,D]$$
$$\qquad ,map\ countSumsR\ \ [A,B,C,D])$$

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`

So far we have only seen a conversion within the `structured` approach. 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.[6] The type representation is similar to a collapsed version of `structured`, where all types inhabit a single kind Un_D:

$$\textbf{kind}\ Un_D = V_D \mid U_D \mid Par_D$$
$$\quad \mid K_D\quad KType\ \star$$
$$\quad \mid Rec_D\ RecType\ (\star \rightarrow \star)$$
$$\quad \mid M_D\ Meta_D\ Un_D$$
$$\quad \mid Un_D\ :+:_D\ Un_D$$
$$\quad \mid Un_D\ :\times:_D\ Un_D$$
$$\quad \mid (\star \rightarrow \star)\ :\circ:_D\ Un_D$$

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

$$\textbf{kind}\ Meta_D = D_D\ MetaData \mid C_D\ MetaCon \mid F_D\ MetaField$$

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:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \llbracket \alpha :: Un_D \rrbracket_D\ (\rho :: \star) :: \star\ \textbf{where}} \\
\quad U_{1D} & :: \llbracket U_D \rrbracket_D\ \rho & \\
\quad M_{1D} & :: \llbracket \alpha \rrbracket_D\ \rho \to \llbracket M_D\ \iota\ \alpha \rrbracket_D\ \rho & \\
\quad Par_{1D} & :: \rho & \to \llbracket Par_D \rrbracket_D\ \rho \\
\quad K_{1D} & :: \alpha & \to \llbracket K_D\ \iota\ \alpha \rrbracket_D\ \rho \\
\quad Rec_{1D} & :: \phi\ \rho & \to \llbracket Rec_D\ \iota\ \phi \rrbracket_D\ \rho \\
\quad Comp_{1D} & :: \phi\ (\llbracket \alpha \rrbracket_D\ \rho) & \to \llbracket \phi\ :\circ:_D\ \alpha \rrbracket_D\ \rho \\
\quad L_{1D} & :: \llbracket \phi \rrbracket_D\ \rho \to \llbracket \phi\ :+:_D\ \psi \rrbracket_D\ \rho & \\
\quad R_{1D} & :: \llbracket \psi \rrbracket_D\ \rho \to \llbracket \phi\ :+:_D\ \psi \rrbracket_D\ \rho & \\
\quad :\times:_D & :: \llbracket \phi \rrbracket_D\ \rho \to \llbracket \psi \rrbracket_D\ \rho \to \llbracket \phi\ :\times:_D\ \psi \rrbracket_D\ \rho &
\end{array}$$

The significant difference from `structured` is the lack of structure. The types (and kinds) do not prevent an L_{1D} from showing up under a $:\times:_D$, for example. It is clear that `structured` contains more information than `generic-deriving`, so the conversion should be simple.

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

$$\begin{array}{l}
\textbf{class}\ Generic_D\ (\alpha :: \star)\ \textbf{where} \\
\quad Rep_D\ \alpha :: Un_D \\
\quad Par_D\ \alpha :: \star \\
\quad Par_D = NoPar \\
\quad from_D\ :: \alpha \to \llbracket Rep_D\ \alpha \rrbracket_D\ (Par_D\ \alpha) \\
\quad to_D\ \ \ \ :: \llbracket Rep_D\ \alpha \rrbracket_D\ (Par_D\ \alpha) \to \alpha
\end{array}$$

$$\begin{array}{l}
\textbf{class}\ Generic_{1D}\ (\phi :: \star)\ \textbf{where} \\
\quad Rep_{1D}\ \phi :: Un_D \\
\quad from_{1D} :: \phi\ \rho \to \llbracket Rep_{1D}\ \phi \rrbracket_D\ \rho \\
\quad to_{1D}\ \ \ :: \llbracket Rep_{1D}\ \phi \rrbracket_D\ \rho \to \phi\ \rho
\end{array}$$

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 Par_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:

$$\begin{array}{l}
\textbf{instance}\ (Generic\ \alpha, Converts_{S\to D_0}\ (Rep\ \alpha)) \\
\qquad\qquad \Rightarrow Generic_D\ \alpha\ \textbf{where} \\
\quad Rep_0\ \alpha = S_{\to}G_0\ (Rep\ \alpha)\ (Par_g\ \alpha) \\
\quad Par_0\ \ \alpha = Par_g\ \alpha \\
\quad from_0 = s_{\to}g_0 \circ from \\
\quad to_0\ \ \ \ = to \circ s_{\leftarrow}g_0
\end{array}$$

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 $Converts_{S\to D_0}$, whose methods $s_{\to}g_0$ and $s_{\leftarrow}g_0$ perform the value-level conversion.

4.2.1 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:

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

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:

$$S_{\to}G_0\ (\phi\ :\circ:\ \alpha)\ \rho = K_D\ U\ (\phi\ (S_{\to}G_{0_{comp}}\ \alpha\ \rho))$$

$$\begin{array}{ll}
\multicolumn{2}{l}{S_{\to}G_{0_{comp}}\ (\alpha :: Arg)\ (\rho :: \star) :: \star} \\
S_{\to}G_{0_{comp}}\ Par & \rho = \rho \\
S_{\to}G_{0_{comp}}\ (K\ \alpha) & \rho = \alpha \\
S_{\to}G_{0_{comp}}\ (Rec\ \iota\ \phi) & \rho = \phi\ \rho \\
S_{\to}G_{0_{comp}}\ (\phi\ :\circ:\ \alpha) & \rho = \phi\ (S_{\to}G_{0_{comp}}\ \alpha\ \rho)
\end{array}$$

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

4.2.2 Value conversion

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

$$\begin{array}{l}
\textbf{class}\ Converts_{S\to D_0}\ (\alpha :: \kappa)\ \textbf{where} \\
\quad s_{\to}g_0 :: \llbracket \alpha \rrbracket\ \rho \to \llbracket S_{\to}G_0\ \alpha\ \rho \rrbracket\ \rho \\
\quad s_{\leftarrow}g_0 :: \llbracket S_{\to}G_0\ \alpha\ \rho \rrbracket\ \rho \to \llbracket \alpha \rrbracket\ \rho
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ (Converts_{S\to D_0}\ \alpha) \Rightarrow Converts_{S\to D_0}\ (Data\ \iota\ \alpha)\ \textbf{where} \\
\quad s_{\to}g_0\ (D_1\ x) = M_{1D}\ (s_{\to}g_0\ x) \\
\quad s_{\leftarrow}g_0\ (D_1\ x) = M_{1D}\ (s_{\leftarrow}g_0\ x)
\end{array}$$

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:

$$\textbf{instance}\ (Convert_{S\rightarrow D_0}\ \alpha) \Rightarrow Convert_{S\rightarrow D_0}\ (Leaf\ (Con\ \iota\ \alpha))\ \textbf{where}$$
$$s_{\rightarrow}g_0\ (C_1\ x) = M_{1D}\ (s_{\rightarrow}g_0\ x)$$

$$\textbf{instance}\ (Convert_{S\rightarrow D_0}\ \alpha, Convert_{S\rightarrow D_0}\ \beta) \Rightarrow Convert_{S\rightarrow D_0}\ (Bin\ \alpha\ \beta)\ \textbf{where}$$
$$s_{\rightarrow}g_0\ (L_1\ x) = L_{1D}\ (s_{\rightarrow}g_0\ x)$$
$$s_{\rightarrow}g_0\ (R_1\ x) = R_{1D}\ (s_{\rightarrow}g_0\ x)$$

$$\textbf{instance}\ Convert_{S\rightarrow D_0}\ Empty\ \textbf{where}\ s_{\rightarrow}g_0\ U_1 = U_{1D}$$

$$\textbf{instance}\ (Convert_{S\rightarrow D_0}\ \alpha) \Rightarrow Convert_{S\rightarrow D_0}\ (Leaf\ (Field\ \iota\ \alpha))\ \textbf{where}$$
$$s_{\rightarrow}g_0\ (S_1\ x) = M_{1D}\ (s_{\rightarrow}g_0\ x)$$

$$\textbf{instance}\ (Convert_{S\rightarrow D_0}\ \alpha, Convert_{S\rightarrow D_0}\ \beta) \Rightarrow Convert_{S\rightarrow D_0}\ (Bin\ \alpha\ \beta)\ \textbf{where}$$
$$s_{\rightarrow}g_0\ (x :\times: y) = s_{\rightarrow}g_0\ x\ :\times:_D\ s_{\rightarrow}g_0\ y$$

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

$$\textbf{instance}\ Convert_{S\rightarrow D_0}\ (K\ \iota\ \alpha)\ \textbf{where}\ s_{\rightarrow}g_0\ (K\ x) = K_{1D}\ x$$
$$\textbf{instance}\ Convert_{S\rightarrow D_0}\ (Rec\ \iota\ \phi)\ \textbf{where}\ s_{\rightarrow}g_0\ (Rec\ x) = K_{1D}\ x$$
$$\textbf{instance}\ Convert_{S\rightarrow D_0}\ Par\ \textbf{where}\ s_{\rightarrow}g_0\ (Par\ x) = K_{1D}\ x$$

$$\textbf{instance}\ (Functor\ \phi, Convert_{comp}\ \alpha) \Rightarrow Convert_{S\rightarrow D_0}\ (\phi :\circ: \alpha)\ \textbf{where}$$
$$s_{\rightarrow}g_0\ (Comp\ x) = K_{1D}\ (g_{\rightarrow}g_{0_{comp}}\ x)$$

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

$$\textbf{class}\ Convert_{comp}\ (\alpha :: Arg)\ \textbf{where}$$
$$g_{\rightarrow}g_{0_{comp}} :: Functor\ \phi \Rightarrow \phi\ (\llbracket \alpha \rrbracket\ \rho) \rightarrow \phi\ (S_{\rightarrow}G_{0_{comp}}\ \alpha\ \rho)$$

$$\textbf{instance}\ Convert_{comp}\ Par\ \textbf{where}\ g_{\rightarrow}g_{0_{comp}} = fmap\ unPar$$
$$\textbf{instance}\ Convert_{comp}\ (K\ \iota\ \alpha)\ \textbf{where}\ g_{\rightarrow}g_{0_{comp}} = fmap\ unK_1$$
$$\textbf{instance}\ Convert_{comp}\ (Rec\ \iota\ \phi)\ \textbf{where}\ g_{\rightarrow}g_{0_{comp}} = fmap\ unRec$$

$$\textbf{instance}\ (Functor\ \phi, Convert_{comp}\ \alpha) \Rightarrow Convert_{comp}\ (\phi :\circ: \alpha)\ \textbf{where}$$
$$g_{\rightarrow}g_{0_{comp}} = fmap\ (g_{\rightarrow}g_{0_{comp}} \circ unComp)$$

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 $[\alpha]$.

4.3 To $Generic_{1D}$

Similarly, the conversion to $Generic_{1D}$ has two components.

4.3.1 Type conversion

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

$$S_{\rightarrow}G_1\ (\alpha :: \kappa) :: Un_D$$

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

$$S_{\rightarrow}G_1\ (K\ \iota\ \alpha) = K_D\ \iota\ \alpha$$
$$S_{\rightarrow}G_1\ (Rec\ \iota\ \alpha) = Rec_D\ \iota\ \alpha$$
$$S_{\rightarrow}G_1\ Par = Par_D$$
$$S_{\rightarrow}G_1\ (\phi :\circ: \alpha) = \phi\ :\circ:_D\ S_{\rightarrow}G_1\ \alpha$$

4.3.2 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:

$$\textbf{class}\ Convert_{S\rightarrow D_1}\ (\alpha :: \kappa)\ \textbf{where}$$
$$s_{\rightarrow}g_1 :: \llbracket \alpha \rrbracket\ \rho \rightarrow \llbracket S_{\rightarrow}G_1\ \alpha \rrbracket_D\ \rho$$

Again, we only give instances of $Convert_{S\rightarrow D_1}$ 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 2.4.2) can be transported to `generic-deriving` with the following instance:

$$\textbf{instance}\ Generic_{1D}\ [\,]\ \textbf{where}$$
$$Rep_{1D}\ [\,] = S_{\rightarrow}G_1\ (Rep\ [NoPar])$$
$$from_{1D}\ x = s_{\rightarrow}g_1\ (from\ x)$$

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 4.2.2, 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. From `generic-deriving` to `regular`

The conversion of the previous section was rather trivial because the two libraries involved are very similar. We now turn our attention to a conversion to a more unrelated approach, namely `regular`. The `regular` library, first described in the context of generic rewriting (Van Noort et al. 2008), 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.

5.1 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:

$$\textbf{data}\ \llbracket \alpha :: Un_R \rrbracket_R\ (\tau :: \star)\ \textbf{where}$$
$$U_R :: \llbracket U_R \rrbracket_R\ \tau$$

$$
\begin{array}{ll}
I_R & :: \tau \to [\![I_R]\!]_R\ \tau \\
K_R & :: \alpha \to [\![K_R\ \alpha]\!]_R\ \tau \\
L_R & :: [\![\alpha]\!]_R\ \tau \to [\![\alpha\ :+:_R\ \beta]\!]_R\ \tau \\
R_R & :: [\![\beta]\!]_R\ \tau \to [\![\alpha\ :+:_R\ \beta]\!]_R\ \tau \\
(:\times:_R) & :: [\![\alpha]\!]_R\ \tau \to [\![\beta]\!]_R\ \tau \to [\![\alpha\ :\times:_R\ \beta]\!]_R\ \tau
\end{array}
$$

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:

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

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

5.2 Type conversion

We convert to `regular` from `generic-deriving`, as we do not need the added complexity of `structured`. Naturally, `structured` representations can be converted into `regular` by first converting them to `generic-deriving`.

The conversion type family takes a `generic-deriving` representation and returns a `regular` representation:

$$
D_{\to}R\ (\alpha :: \mathit{Un}_D) :: \mathit{Un}_R
$$

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

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

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

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

One might wonder what would happen if the `generic-deriving` representation would have 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.

5.3 Value conversion

The conversion of the values is witnessed by the $\mathit{Convert}_{D_{\to}R}$ type class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Convert}_{D_{\to}R}\ (\alpha :: \mathit{Un}_D)\ \tau\ \textbf{where} \\
\quad d_{\to}r :: [\![\alpha]\!]_D\ \rho \to [\![D_{\to}R\ \alpha]\!]_R\ \tau
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Convert}_{D_{\to}R}\ (K_D\ (R\ S)\ \tau)\ \tau\ \textbf{where} \\
\quad d_{\to}r\ (K_{1D}\ x) = I_R\ x
\end{array}
$$

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 value conversion, we provide a *Regular* instance for all $\mathit{Generic}_D$ types. It is here that we set the second parameter of $\mathit{Convert}_{D_{\to}R}$ to the type being converted (α):

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Generic}_D\ \alpha, \mathit{Convert}_{D_{\to}R}\ (\mathit{Rep}_D\ \alpha)\ \alpha) \\
\qquad \Rightarrow \mathit{Regular}\ \alpha\ \textbf{where} \\
\quad \mathit{PF}\ \alpha = D_{\to}R\ (\mathit{Rep}_D\ \alpha)\ \alpha \\
\quad \mathit{from}_R\ x = d_{\to}r\ (\mathit{from}_D\ x)
\end{array}
$$

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** $\mathit{Generic}_D$ 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.

6. 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 (Rodriguez Yakushev et al. 2009).

6.1 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:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Un}_M\ \kappa = U_M \mid I_M\ \kappa \mid K_M\ \star \mid \mathit{Un}_M\ \kappa\ :\triangleright:_M\ \kappa \\
\qquad\qquad\quad \mid \mathit{Un}_M\ \kappa\ :+:_M\ \mathit{Un}_M\ \kappa \mid \mathit{Un}_M\ \kappa\ :\times:_M\ \mathit{Un}_M\ \kappa
\end{array}
$$

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`:

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

In this example, the index κ is $\star$. This is how the original `multirec` library encodes indices (by using the datatype itself as an index), and this turns out to be convenient for our conversion, so we will always use Un_M instantiated with 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:

$$
\begin{array}{l}
\textbf{data}\ [\![\alpha :: \mathit{Un}_M\ \kappa]\!]_M\ (\tau :: \kappa \to \star)\ (\iota :: \kappa)\ \textbf{where} \\
\quad U_M \quad :: [\![U]\!]_M\ \tau\ \iota \\
\quad I_M \quad\ :: \tau\ o \to [\![I_M\ o]\!]_M\ \ \tau\ \iota
\end{array}
$$

$$
\begin{array}{ll}
K_M & :: \alpha \rightarrow [\![K_M\ \alpha]\!]_M\ \tau\ \iota \\
Tag_M & :: [\![\alpha]\!]_M\ \tau\ \iota \rightarrow [\![\alpha\ :\triangleright:_M\ \iota]\!]_M\ \tau\ \iota \\
L_M & :: [\![\alpha]\!]_M\ \tau\ \iota \rightarrow [\![\alpha\ :+:_M\ \beta]\!]_M\ \tau\ \iota \\
R_M & :: [\![\beta]\!]_M\ \tau\ \iota \rightarrow [\![\alpha\ :+:_M\ \beta]\!]_M\ \tau\ \iota \\
:\times:_M & :: [\![\alpha]\!]_M\ \tau\ \iota \rightarrow [\![\beta]\!]_M\ \tau\ \iota \rightarrow [\![\alpha\ :\times:_M\ \beta]\!]\ \tau\ \iota
\end{array}
$$

In other words, the interpretation $[\![\alpha]\!]_M\ \tau\ \iota$ can be seen as a family of datatypes, one for each particular index ι. Note how 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:

$$
\begin{array}{l}
\textbf{newtype}\ I_{0M}\ \ \alpha = I_{0M}\ \alpha \\
\textbf{class}\ Fam_M\ (\phi :: \star \rightarrow \star)\ \textbf{where} \\
\quad PF_M\ \phi :: Un_M\ \star \\
\quad from_M :: \phi\ \iota \rightarrow \iota \rightarrow [\![PF_M\ \phi]\!]_M\ \ I_{0M}\ \ \iota \\
\\
\textbf{class}\ El_M\ (\phi :: \kappa \rightarrow \star)\ (\iota :: \kappa)\ \textbf{where} \\
\quad proof_M :: \phi\ \iota
\end{array}
$$

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 can contain elements of type Zig and 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:

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

6.2 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:

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

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:

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

Note that the tag on the K_D type determines whether a particular constructor argument becomes a family index or not. The R tag in `generic-deriving` is used for occurrences of datatypes; this means that a `multirec` family generated by our conversion will include all such types as part of the family. This might sometimes give rise to a family that is larger than desired; for instance, for the datatype D of Section 2.1, the family is composed of both D and Int. However, it is preferrable to have a larger family and ignore some indices, than to have a smaller family which is missing important indices. We take a conservative approach, and generate large families, including base types such as Int.[7]

Having defined $D_{\rightarrow}M$ to convert one datatype, we're 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:

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

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 $\bot$ and use it as a constant).

6.3 Value conversion

Converting a value of a single type is done in exactly the same way as for the other conversions:

$$
\begin{array}{l}
\textbf{class}\ Convert_{D_{\rightarrow}M}\ (\alpha :: Un_D)\ \textbf{where} \\
\quad d_{\rightarrow}m :: [\![\alpha]\!]_D\ \rho \rightarrow [\![D_{\rightarrow}M\ \alpha]\!]_M\ \ I_{0M}\ \ \sigma
\end{array}
$$

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:

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

For example, $This\ ZigEnd$ is a value of type $[Zig, Zag]\ :@:\ Zig$, and $That\ (This\ (Zag\ ZigEnd))$ is a value of type $[Zig, Zag]\ :@:\ Zag$.

The application of $:@:$ to a single element 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 $That$:

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

Converting a value within a family requires producing the appropriate injection into the right element of the family, plus the tag

[7] It is also possible to parameterise the conversion of a single datatype $D_{\rightarrow}M$ by a type-level list containing the elements of the family we desire, like we do for the family conversion $D_{\rightarrow}M_{Fam}$. In this way we would not need to rely on the tags from `generic-deriving`.

(with Tag_M). We use our $:@:$ GADT for this (which results in a right-biased encoding of the family):

$$\begin{array}{l} \textbf{instance}\ (FamConstrs\ \alpha) \Rightarrow Fam_M\ ((:@:)\ \alpha)\ \textbf{where} \\ \quad PF_M\ ((:@:)\ \alpha) = D_{\to}M_{Fam}\ \alpha \\ \quad from_M\ This\ \quad x = L_M\ (Tag_M\ (d_{\to}m\ (from_D\ x))) \\ \quad from_M\ (That\ k)\ x = R_M\ (from_M\ k\ x) \end{array}$$

The constraints on this instance are not trivial, as each type in the family needs to have a $Generic_D$ instance and be convertible through $Convert_{D_{\to}M}$. The $FamConstrs$ constraint family expresses these requirements:

$$\begin{array}{l} FamConstrs\ (\alpha :: [\star]) :: Constraint \\ FamConstrs\ [\,] \qquad = () \\ FamConstrs\ (\alpha : \beta) = (\,Generic_D\ \alpha, Convert_{D_{\to}M}\ (Rep_D\ \alpha) \\ \qquad\qquad\qquad\qquad , Fam_M\ ((:@:)\ \beta), FamConstrs\ \beta) \end{array}$$

6.4 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 $Generic$ instances for the Zig and Zag types in `structured`. These give rise to $Generic_D$ instances (Section 4), which 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:

$$\begin{array}{l} size_M :: (Fam_M\ \phi, \ldots) \Rightarrow \phi\ \iota \to \iota \to Int \\ size_M = \ldots \\ \textbf{instance}\ Generic\ Zig\ \ \textbf{where} \ldots \\ \textbf{instance}\ Generic\ Zag\ \textbf{where} \ldots \\ zigZag :: Zig \\ zigZag = Zig\ (Zag\ (Zig\ (Zag\ ZigEnd))) \\ test_{d_{\to}m} :: Int \\ test_{d_{\to}m} = size_M\ (proof :: [Zag, Zig]\ {:@:}\ Zig)\ zigZag \end{array}$$

Our test value $test_{d_{\to}m}$ evaluates to 4 as expected. Note that this makes `multirec` even easier to use than before; unlike in our example in Section 6.1, it is not necessary to define a family type, since we can 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 $test_{d_{\to}m}$ works equally well if we supply $proof$ as having type $[Zag, Zig, Int]\ {:@:}\ Zig$, for instance.

7. 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. (2006) 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.

7.1 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:

$$\begin{array}{l} \textbf{data}\ Spine :: \star \to \star\ \textbf{where} \\ \quad Con :: \alpha \to Spine\ \alpha \\ \quad ({:\diamond:}) :: (Data\ \alpha) \Rightarrow Spine\ (\alpha \to \beta) \to \alpha \to Spine\ \beta \end{array}$$

The $Data$ constraint will be explained later.

The $Spine$ datatype is both $Functor$ial and $Applicative$:

$$\begin{array}{l} \textbf{instance}\ Functor\ Spine\ \textbf{where} \\ \quad fmap\ f\ (Con\ x) = Con\ (f\ x) \\ \quad fmap\ f\ (c\ {:\diamond:}\ x) = fmap\ (f \circ)\ c\ {:\diamond:}\ x \\ \textbf{instance}\ Applicative\ Spine\ \textbf{where} \\ \quad pure = Con \\ \quad Con\ f \quad <\!\!*\!\!>\ x \qquad = fmap\ f\ x \\ \quad (c\ {:\diamond:}\ x)\ <\!\!*\!\!>\ Con\ y \quad = fmap\ (\lambda f\ x \to f\ x\ y)\ c\ {:\diamond:}\ x \\ \quad (c\ {:\diamond:}\ x)\ <\!\!*\!\!>\ (d\ {:\diamond:}\ y) = (fmap\ (\lambda f\ d\ y \to f\ (d\ y))\ (c\ {:\diamond:}\ x) \\ \qquad\qquad\qquad\qquad\qquad <\!\!*\!\!>\ d)\ {:\diamond:}\ y \end{array}$$

We can also define a fold on $Spine$:

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

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:

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

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:

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

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. (2006) describe how to support meta-information in the $Spine$ view, and Hinze and Löh (2006) describe how to define $gunfold$.

7.2 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:

$$\begin{array}{l} \textbf{class}\ Convert_{D_{\to}S}\ (\alpha :: Un_D)\ \textbf{where} \\ \quad d_{\to}s :: [\![\,\alpha\,]\!]_D\ \rho \to Spine\ ([\![\,\alpha\,]\!]_D\ \rho) \end{array}$$

The idea is to first build a representation of type $Spine\ ([\![\,\alpha\,]\!]_D\ \rho)$, and later transform this into $Spine\ \alpha$. The instances are unsurprising, and follow the functorial nature of $Spine$:

$$\begin{array}{l} \textbf{instance}\ Convert_{D_{\to}S}\ U_D\ \textbf{where} \\ \quad d_{\to}s\ U_{1D} = Con\ U_{1D} \end{array}$$

$$\textbf{instance}\ (Convert_{D\to S}\ \alpha, Convert_{D\to S}\ \beta) \Rightarrow Convert_{D\to S}\ (\alpha :+:_D\ \beta)\ \textbf{where}$$
$$d_{\to}s\ (L_{1D}\ x) = fmap\ L_{1D}\ (d_{\to}s\ x)$$
$$d_{\to}s\ (R_{1D}\ x) = fmap\ R_{1D}\ (d_{\to}s\ x)$$

$$\textbf{instance}\ (Convert_{D\to S}\ \alpha, Convert_{D\to S}\ \beta) \Rightarrow Convert_{D\to S}\ (\alpha :\times:_D\ \beta)\ \textbf{where}$$
$$d_{\to}s\ (x :\times:_D\ y) = pure\ (:\times:_D) <\!\!*\!\!> d_{\to}s\ x <\!\!*\!\!> d_{\to}s\ y$$

$$\textbf{instance}\ (Data\ \alpha) \Rightarrow Convert_{D\to S}\ (K_D\ \iota\ \alpha)\ \textbf{where}$$
$$d_{\to}s\ (K_{1D}\ x) = Con\ K_{1D} :\diamond: x$$

$$\textbf{instance}\ (Convert_{D\to S}\ \alpha) \Rightarrow Convert_{D\to S}\ (M_D\ \iota\ \alpha)\ \textbf{where}$$
$$d_{\to}s\ (M_{1D}\ x) = fmap\ M_{1D}\ (d_{\to}s\ x)$$

With these instances in place, we are ready to define a *Data* instance for all $Generic_D$ types:

$$\textbf{instance}\ (Generic_D\ \alpha, Convert_{D\to S}\ (Rep_D\ \alpha), Typeable\ \alpha) \Rightarrow Data\ \alpha\ \textbf{where}$$
$$spine\ x = fmap\ to_D\ (d_{\to}s\ (from_D\ x))$$

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 $fmap\ to_D$.

To test our conversion, assume that we had *not* given the *Data* $[\alpha]$ instance in Section 7.1. The *Generic* $[\alpha]$ instance of Section 2.4.2 would cascade down into a *Data* $[\alpha]$ instance using the conversion defined in this section. Assuming also generic functions *everywhere* and *mkT* as defined in `syb`, the expression $everywhere\ (mkT\ (\lambda n \to n + 1 :: Int))\ [1,2,3 :: Int]$ evaluates to $[2,3,4]$, as expected.

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 $Generic_D$ 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` (Mitchell and Runciman 2007, Section 5.3); therefore, by transitivity, we can also provide `uniplate` encodings from `structured`.

8. 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.

8.1 Related work

We have defined conversions between GP approaches before, in Agda (Magalhães and Löh 2012). Those conversions were of a more theoretical nature, as the intention was to formally compare approaches. Furthermore, `generic-deriving` was not involved, nor was the idea of a `structured` library at the top of the hierarchy, decoupling the quest for an "ideal" generic representation from the quest of finding an easy-to-use GP library. 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 (Holdermans et al. 2006, Section 5), our work allows new views to be defined simply by writing a conversion (as in Section 3), or by writing a new universe and interpretation together with a conversion (as in Section 5).

Other approaches to providing functionality mixing different views have been attempted. Chakravarty et al. (2009) mention support for multiple views, but do this through duplication of the universe, interpretation, and datatype representations. The `instant-zipper` and `generic-deriving-extras` Hackage packages 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.

8.2 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 (2013). 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 application potential, and defer performance concerns to future work.

8.3 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 6.2 might be the first example that actually relies on proper tagging. The addition of Par_D to $Generic_D$ in Section 4.1 is entirely unproblematic.

We have used datatype promotion in all approaches, and encode meta-information at the type level, instead of using auxiliary type classes. These changes are not backwards compatible, in particular because the current implementation of datatype promotion requires choosing different names for a representation type (e.g. U_R) and its interpretation (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 clashing,[8] it might be preferrable 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. 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 (in this case, `structured`).

8.4 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 embarassed 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 path from `structured`, and their library will

[8] See `http://hackage.haskell.org/trac/ghc/ticket/6024`.

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.

Should `structured` turn out to be not informative enough to cover a particular approach, then `structured` (and with it, GHC support) 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—-we have arrived in the era of truly generic generic programming!

Acknowledgments

The first author is funded by EPSRC grant number EP/J010995/1. 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, Johan Jeuring, and Sean Leather provided helpful feedback on an earlier draft of this paper.

References

Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy, 2009. Available at `http://www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In Tarmo Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer, 2006. doi:10.1007/11783596_13.

Ralf Hinze and Andres Löh. Generic programming in 3D. *Science of Computer Programming*, 74:590–628, June 2009. doi:10.1016/j.scico.2007.10.006.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In *Proceedings of the 8th international conference on Functional and Logic Programming*, volume 3945, pages 13–29. Springer-Verlag, 2006. doi:10.1007/11737414_3.

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

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. Optimisation of generic programs through inlining. In *Accepted for publication at the 24th Symposium on Implementation and Application of Functional Languages (IFL'12)*, IFL '12, 2013.

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

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 49–60. ACM, 2007. doi:10.1145/1291201.1291208.

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

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 of *Haskell '02*, 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.