# Handling Recursion in Generic Programming Using Closed Type Families

Anna Bolotina[1] and Artem Pelenitsyn[2]

[1] Southern Federal University, Rostov-on-Don, Russia
bolotina@sfedu.ru
[2] Czech Technical University in Prague, Prague, Czech Republic
pelenart@fit.cvut.cz

**Abstract.** Many of the extensively used libraries for datatype-generic programming offer a fixed-point view on datatypes to express their recursive structure. However, some approaches, based on sums of products, do not use a fixed point. They allow for generic functions that do not require to look at the recursive knots in a datatype representation, but raise issues otherwise. A known and unwelcome solution is the use of overlapping instances. We present an alternative approach that uses closed type families to eliminate the need of overlap for handling recursion in datatypes. Moreover, we show that our idiom allows for families of mutually recursive datatypes.
**Category:** full research paper.
**Studentship:** both authors are students.

**Keywords:** Datatype-generic programming · Sums of products · Recursion · Overlapping instances · Closed type families · Zipper · Mutually recursive datatypes · Haskell.

## 1 Introduction

A classical way to generically express a datatype is to represent its constructors as the chains of nested *binary* sums, and turn constructor arguments into the chains of nested *binary* products [25, 3, 14]. De Vries and Löh [5] describe a different sum-of-products approach to representing data using $n$-ary sums and products that are both lists of types; a sum of products is thus a list of lists of types. They call their view SOP which stands for a "sum of products". It is implemented in the generics-sop [6] library and is based on several relatively recent extensions to the Haskell type system, such as *data kinds*, *kind polymorphism* [28] and *constraint kinds*. Using these Haskell features, the library provides the generic view and equips it with a rich collection of high-level combinators, such as ones for constructing sums and products, collapsing to homogeneous structures, and others. They form an expressive instrument for defining generic functions in a more succinct and high-level style as compared to the classical binary sum-of-products views.

There are many generic functions that deal with the recursive knots when traversing the structure of datatypes. Some of the most general examples are *maps* [18] and *folds* [21]; more advanced one is a *zipper* [10, 9, 1]. For handling recursion, several generic programming approaches express datatypes in the form of polynomial functors closed under fixed points [27, 11, 16]. The SOP view naturally supports definitions of functions that do not require a knowledge about recursive occurrences, but otherwise becomes unhandy.

One possible solution to the aforementioned shortcoming of SOP is to modify its core by explicitly encoding recursive positions using the fixed-point approach. However, this may complicate the whole framework significantly. Besides, such a decision may lead to extra conversions between the generic views.

Another known solution uses overlapping instances. This, usually unwelcome, Haskell extension complicates reasoning about the semantics of code. In particular, the program behavior becomes unstable, for it can be affected by any module defining more specific instances. Morris and Jones [22] extensively discuss the problems arising from overlapping instances. The overlap problem also strikes in the security setting when code is compiled as `-XSafe` because GHC does not reflect unsafe overlaps and marks the module as safe [8].

The problem of using overlapping instances was first addressed by Kiselyov et al. [13]. Their technique for avoiding overlap relies on a Haskell 98 extension for functional dependencies. The solution proposes two variants of defining a type-level equality predicate, a type class, and then systematically localizes overlap by circumventing it with that predicate. The first version of type equality maps types to unique type representations and compares them. Its later implementation with type families [12] fully eliminates OverlappingInstances. Although, each type needs a representation instance to be derived—by means of Template Haskell or GHC. The most generic solution for type equality, the second version, again makes use of overlapping instances, however.

Closed type families, today's Haskell extension, has been proposed primarily to obviate the need for overlapping instances. But it does not seem to be well-known that they have useful application in the generic programming area.

We follow up the overlap-avoiding idea from the existing approach and make the following contributions.

- We describe the problem with the current approach of SOP in detail (Section 2).
- We introduce an idiom that overcomes the problem. The approach avoids both, the use of overlapping instances and changing a generic representation (Section 3).
- We evaluate our approach through the development of a larger-scale use case—the generic zipper. The zipper is meant to be easily and flexibly used with families of mutually recursive datatypes (Section 5).
- We note, that our approach can contribute to the generics-sop's one eliminating some boilerplate instance declarations, which necessarily arise in practice as a consequence of absence of information about recursion points. An example of that, taken from the basic-sop [24] package, is discussed in Section 3.2.

We believe that our idea is suitable for any sum-of-products approach that do not exploit the fixed point view and thus subject to the problem. We choose

```
data NP (f :: k → *) (xs :: [k]) where
  Nil  :: NP f '[]
  (:*) :: f x → NP f xs → NP f (x ': xs)

data NS (f :: k → *) (xs :: [k]) where
  Z :: f x → NS f (x ': xs)
  S :: NS f xs → NS f (x ': xs)
```

**Fig. 1.** Datatypes for $n$-ary sums and products.

the generics-sop approach as a case study because it appears to be a widely applicable library and builds on powerful language extensions implemented in GHC.

## 2 The SOP universe and the problem

In this section, we first review the SOP view on data, describing its basic concepts to introduce the terminology we are using. Then we discuss the problem with handling recursion by generic functions and illustrate it with a short example.

### 2.1 The SOP view

We first explain the terminology we adopt from SOP [5, 15] and use throughout the paper. The main idea of the SOP view is to use $n$-ary sums and products to represent a datatype as an isomorphic code whose kind is a list of lists of types. The SOP approach expresses the code using the DataKinds extension, with a type family:

```
type family Code (a :: *) :: [[*]]
```

An $n$-ary sum and an $n$-ary product are therefore modelled as type-level heterogeneous lists: the inner list is an $n$-ary product that represents a sequence of constructor arguments, while the outer list, an $n$-ary sum, corresponds to a choice of a particular constructor.

Consider, for instance, a datatype of binary trees:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

This datatype is isomorphic to the following code:

```
type instance Code (Tree a) = '[ '[a], '[Tree a, Tree a]]
```

As shown in Figure 1, the datatypes `NS` for an $n$-ary sum and `NP` for an $n$-ary product are defined as GADTs and are *indexed* [9] by a promoted list of types. The encoding also holds an auxiliary type constructor `f` (typically, a functor) which is meant to be applied to every element of the index list. Therefore, `NP` is a modest abstraction over a heterogeneous list.

The definitions of `NS` and `NP` are kind polymorphic. The index list is allowed to contain types of arbitrary kind `k`, since `k` turns to `*` by applying the type constructor `f`. Basic instantiations of type parameter `f` found in SOP are identity functor `I`, that is, a type-level equivalent for `id` function, and a constant functor `K`, an analogue of `const`:

```
newtype I (a :: *)          = I {unI :: a}
newtype K (a :: *) (b :: k) = K {unK :: a}
```

If instantiated with `I`, `NP` is a plain heterogeneous list, while `K a` turns it into a homogeneous one, isomorphic to `[a]`. Here is an example value of type `NP I`:

```
I 5 :* I True :* I 'x' :* Nil :: NP I '[Int, Bool, Char]
```

We turn to the sum definition now. The constructor `S` of `NS`, given an index in $n$-element list, results in an index in a list with $n+1$ elements, skipping the first one, while `Z` stores the payload of type `f x`. For example, the following chooses the third element of a sum:

```
S (S (Z (I 5))) :: NS I '[Char, Bool, Int, Bool]
```

The sum constructors are similar to Peano numbers, so the choice from a sum of products of a datatype matches the index of its particular constructor in the index list and stores the product representing arguments of that constructor.

With the `NS` and `NP` machinery at hand, SOP defines the `Generic` class with conversion functions `from` and `to` witnessing the isomorphism between a datatype and its generic representation:

```
type Rep a = SOP I (Code a)

class All SListI (Code a) ⇒ Generic (a :: *) where
  type Code a :: [[*]]
  from :: a → Rep a
  to   :: Rep a → a
```

The sum of products type, `SOP f`, is a newtype-wrapper for `NS (NP f)`, and the structural representation `Rep` of a datatype `a` is a type synonym for a `SOP I` of `a`'s code. The functions, `from` and `to`, perform a shallow conversion of the datatype topmost layer—they do not recursively translate the constructor arguments.

We leave out discussion of the `SListI` constraint in the `Generic` class definition as irrelevant to our work. Although, we do use `All` constraint combinator (as in `All SListI`) in the following. Therefore, it is worth noting that `All` applies a particular constraint (e.g. `SListI` above) to each member of a list of types. The usage of constraints as type arguments is allowed due to the ConstraintKinds language extension introducing a dedicated kind `Constraint`.

We have introduced generic representation employed by the SOP library and are ready to describe the problem of handling recursion points, stemming from the representation.

### 2.2 Problem with handling recursion

We illustrate the problem through a short example. The QuickCheck library [4] for automatic testing of Haskell code defines a helper function `subterms` that takes a term and obtains a list of all its immediate subterms that are of the same type as the given term, that is, all the recursive positions in the term structure. In the following, we reimplement this function using the SOP view. But first we give a sketch of solution to introduce the idea.

**Non-implementation** The SOP-based `subterms` definition is outlined below:

```
subterms :: Generic a ⇒ a → [a]
subterms t = subtermsNS $ unSOP $ from t

subtermsNS :: NS (NP I) xss → [a]
subtermsNS (S ns) = subtermsNS ns
subtermsNS (Z np) = subtermsNP np

subtermsNP :: ∀a xs. NP I xs → [a]
subtermsNP p (I y :* ys)
  | typeOf @a y = castEq y : subtermsNP ys
  | otherwise   = subtermsNP ys
subtermsNP _ Nil = []
```

The function `subterms` translates the term to its representation, unwrapping the sum of products from `SOP`, and passes that to the auxiliary function `subtermsNS`. The latter merely traverses the sum and, once reaches the product, passes it further to `subtermsNP`.

The algorithm of `subtermsNP` is straightforward—it traverses the product, appending current element to the result list if its type is the same as of the term `t`, otherwise skipping the element. We use GHC's TypeAppications extension to pass that type.

**Overlap-based implementation** Now, we need a way to check type equality and, in the case of equal types, to witness that the element is of the desired type. There is no clear path to this at the moment. Therefore, we step back (until Section 3.1) and, to implement `subtermNP`, follow the QuickCheck's example[3], using overlapping instances of a dedicated class instead.

```
class Subterms a (xs :: [*]) where
  subtermsNP :: NP I xs → [a]

instance Subterms a xs ⇒ Subterms a (x ': xs) where
  subtermsNP (_   :* xs) = subtermsNP xs
instance {-# OVERLAPS #-}
         Subterms a xs ⇒ Subterms a (a ': xs) where
  subtermsNP (I x :* xs) = x : subtermsNP xs
instance Subterms a '[] where
  subtermsNP _ = []
```

[3] The QuickCheck library applies another approach to generic programming, namely GHC.Generics.

To make the whole solution work, we need to propagate the constraints all the way through `subtermsNS` and `subterms` signatures:

```
subterms   :: (Generic a, All (Subterms a) (Code a))
           ⇒ a → [a]
subtermsNS :: All (Subterms a) xss
           ⇒ NS (NP I) xss → [a]
```

Although the approach works, as exemplified by a number of the packages on Hackage, we aim for release of generic programs from overlap. This would remove the complexity overhead introduced by the approach, as we have mentioned in the introduction.

## 3 Handling recursion with closed type families

In the previous section, we have shown a solution to the problem of handling recursion, which uses overlapping instances. We are going to improve the solution and remove overlap now.

Closed type families are the Haskell language extension introduced by Eisenberg et al. [7]. The main idea of the extension is that the equations for a *closed type family* are disallowed outside its declaration. Under the extension, we can give the following definition of type-level equality:

```
type family Equal a x :: Bool where
  Equal a a = 'True
  Equal _ _ = 'False
```

The equations in a closed type family are matched in a top-to-bottom order. Since the order is fixed, the overlapping equations here cannot be used to define unsound type-level equations.

### 3.1 Solution to `subtermsNP` revised

We now return to our running example from Section 2.2. With the type equality predicate, we can decide if `a ∼ b` by defining a type class:

```
class DecideEq (eq :: Bool) (a :: *) (b :: *) where
  decideEq :: Maybe (b :∼: a)

instance a ∼ b ⇒ DecideEq True  a b where
  decideEq = Just Refl
instance          DecideEq False a b where
  decideEq = Nothing
```

If the types `a` and `b` are the same, the `:∼:` type from `Data.Type.Equality` witnesses the equality.

For every element in a list of all direct subterms of a term we shall provide a proof object witnessing its type (in)equality to the type of the term. This can be done by means of the `All` combinator and partially applied auxiliary type class `ProofEq`, which abbreviates the heavy-weighted interface of `DecideEq`:

```
class    DecideEq (Equal a b) a b ⇒ ProofEq a b
instance DecideEq (Equal a b) a b ⇒ ProofEq a b
```

The equality proof can then be employed to provide a type-safe cast between two equal types:

```
castEq :: ∀a b. ProofEq a b ⇒ b → Maybe a
castEq t = (\d → castWith d t) <$> decideEq @(Equal a b)
```

Resulting implementation of `subtermsNP` resembles our first definition given in the previous section:

```
subtermsNP :: ∀a xs. All (ProofEq a) xs ⇒ NP I xs → [a]
subtermsNP (I (y :: x) :* ys)
  = case castEq y of
      Just t  → t : subtermsNP ys
      Nothing → subtermsNP  ys
subtermsNP _ Nil = []
```

As a side note, we make use of the ScopedTypeVariables extension in the definition above, as the type of the element being matched does not appear in the function signature, since it may match an empty list.

To complete the solution of the problem, the `ProofEq` constraint must be added to the `subterms` and `subtermsNS` declarations as well.

In summary, we claim that any generic function accessing recursive knots in the underlying datatype structure can be defined in the way described above for `subterms`' task. We give another example showing how to adapt our idiom to different scenarios in the following subsection.

### 3.2 Generic show

The function `show` is a common example of useful functions that traverse a datatype's recursive structure. It is known that this function can be defined in a generic way for an arbitrary datatype. De Vries and Löh define generic function `gshow` in the basic-sop package [24] based on the SOP view. We follow their implementation of `gshow` for the most part, but improve it in respect of handling recursion. The original `gshow` yields to the standard `show` generated through `deriving Show`, because it does not consult with recursion points to place parentheses. We eliminate this drawback.

The following exploits the idea of *pattern matching*. As before, we consider two cases. In the first case, when the position we are matching on is not recursive, we only require it to be an instance of `Show`, and invoke its `show` function. Whereas in the case of the recursive position, we surround it with parentheses and apply our generic function `gshow`. Thus, by means of the type family for equality, we model a form of pattern matching on the types again:

```
class CaseShow (eq :: Bool) (a :: *) (b :: *) where
  caseShow' :: b → String

instance Show  b ⇒ CaseShow 'False a b where
  caseShow'   = show
instance GShow a ⇒ CaseShow 'True  a a where
  caseShow' t = "(" ++ gshow t ++ ")"
```

We provide a synonym for the `CaseShow (Equal a b) a b` instance, which we call `CaseRecShow`, as before with `ProofEq`; likewise a synonym for the matching function:

```
caseShow :: ∀a b. CaseRecShow a b ⇒ b → String
caseShow t = caseShow' @(Equal a b) @a t
```

The resulting function `gshow` is a subject of a number of constraints abbreviated by a `GShow` synonym:

```
type GShow a = (Generic a, HasDatatypeInfo a,
                All2 (CaseRecShow a) (Code a))

gshow :: ∀a. GShow a ⇒ a → String
```

The function `gshow` employs meta-information provided by generics-sop's class `HasDatatypeInfo` to show the names of a datatype constructor and its record fields. The generics-sop library is able to derive this metadata automatically. The function is also constrained by `CaseRecShow` with the `All2` combinator that is an analogue of `All` for a list of lists of types.

We define `gshow` mutually recursive with `caseShow`. The full implemenentation of the function `gshow` is left for the extended version of the paper in Technical Report[4].

The function `gshow` can now be used to generically show data—for example, a value of type `Tree Bool`; note that `Tree a` from Section 2.1 is now assumed to be an instance of `Generic` and `HasDatatypeInfo`.

```
*Main> let tree = Node (Leaf True) (Leaf False)
*Main> gshow tree

"Node (Leaf True) (Leaf False)"
```

Here is a benefit of our implementation: it can be used directly, without any additional instance declarations, whereas basic-sop [24] offers the following usage pattern for `gshow` and some datatype `T`:

```
instance Show T where
  show = gshow
```

This is a consequence of `gshow` from basic-sop not treating recursive positions separately, and therefore requiring the `Show` constraint for all knots in the datatype structure.

[4] https://users.fit.cvut.cz/~pelenart/2018-generic-zipper-tr.pdf

## 4 Polymorphic recursion

The solution, described in the previous section, is applicable to a range of datatypes that are *monomorphically recursive*. Any of those datatypes has the same type parameters in the left-hand side of its definition and at its recursion points (e.g. `Tree a` from Section 2.1). We can go further and proceed with a solution for generic functions, which covers some datatypes whose type parameters in each recursive knot may differ from those in its parent. It turns out, as we will show below, that the solution allows for datatypes with a "simple" form of *polymorphic recursion*, but fails to work for nested datatypes.

Assume we have a polymorphically recursive datatype `PolyRec a` defined in terms of a type family `Poly`:

```
data PolyRec a = Tail a | Rec a (PolyRec (Poly a))

type family Poly a where
  Poly Bool = Char
  Poly Char = Bool
  Poly a    = a
```

For managing polymorphic recursion in this datatype, we can write an analogue of the `Equal` type family from Section 3, which ignores type parameters when checking two polymorphic types. Since any datatype with type parameters `f a b c ...` has kind `* → (* → (* → ...))`, a `PolyEq` type family can be defined thus:

```
type family PolyEq (a :: k) (x :: k) :: Bool where
  PolyEq (f a) (g b) = PolyEq f g
  PolyEq a     a     = 'True
  PolyEq _     _     = 'False
```

The function `gshow` from Section 3.2 can be reimplemented by using this type family instead of `Equal`. The only piece of its definition must be changed as well in order to recursively invoke the function each time with a new type (with the proper change in the `CaseRecShow` definition):

```
instance Show  a ⇒ CaseShow 'False a where
  caseShow'   = show
instance GShow a ⇒ CaseShow 'True  a where
  caseShow' t = "(" ++ gshow t ++ ")"
```

Unfortunately, this approach, albeit working well for datatypes defined like `PolyRec a`, becomes unsuitable for nested datatypes, such as one below:

```
data Nested a = Epsilon | Nest a (Nested [a])
```

The culprit is the constraint `All2 (CaseRecShow a) (Code a)` that now turns out to be a root of nonterminating computation of the constraint, because of the `GShow b` instance context above.

## 5 The generic zipper

The zipper is a data structure that enables efficient navigation and editing within the tree-like structure of a datatype. It represents a current location in that structure, storing a tree node, a *focus*, along with its context. Having a zipper focused on a recursive knot in a structure, we may produce a new location by moving the focus up, down, left, or right. On the way, we can update the nodes. Entering and leaving the navigation usually need a special care.

The classical zipper described by Huet [10] can be generically calculated for regular datatypes [9]—all datatypes expressible as polynomial expressions on types. Yakushev et al. [27] generalize the definition of the generic zipper for an arbitrary family of mutually recursive datatypes. All mentioned solutions require a datatype to be expressed using forms of a fixed-point operator, since the zipper operates on recursion points.

In this section, we describe our approach allowing one to define the generic zipper out of a representation that does not exploit a fixed point. We start with the generic zipper interface and an example of how it can be used (Section 5.1). Then, we turn to the type-level machinery employed to define locations inside mutually recursive datatypes using the SOP view (Section 5.2). Finally, we discuss the implementation of the generic zipper interface — the functions for manipulating locations (Section 5.3).

### 5.1 Interface and usage

The interface we provide for the generic zipper is shown on Figure 2. It comprises the functions for *movement*, *starting* and *ending navigation*, and *updating* the focus, which are defined over the location structure.

The functions `goUp`, `goDown`, `goLeft`, and `goRight` produce a location with the focus moved *up* to the parent of the focal subtree, *down* to its leftmost child, *left* and *right* to the left and right sibling, respectively, if it is possible. A movement may fail, as specified by the `Maybe` monad, if we cannot go further in a chosen direction. Navigation in a tree starts at the root, and the type variable `a` refers to the root type that remains the same during the navigation, while the type in the focus of the location may vary and is one of the types in a type list `fam`.

The function signature of `enter` specifies the constraints necessary to begin navigation in a structure. Firstly, a datatype of the structure needs to have the `Generic` representation. Secondly, the `In` constraint checks if type `a` is a member of a type family `fam`. Thirdly, the `Zipper` constraint collects specific constraints that refer to the implementation of movement operations. Note that the universal quantifier here sets the instantiation order of the type variables for type applications that will be a part of our usage pattern for the zipper.

The `leave` function ends navigation moving up to the root and returns its modified value.

The `update` function modifies the focal subtree with a given constrained function. The type in focus is existentially quantified inside `Loc` and should

**Movement functions**

```
goUp    :: Loc a fam c → Maybe (Loc a fam c)
goDown  :: Loc a fam c → Maybe (Loc a fam c)
goLeft  :: Loc a fam c → Maybe (Loc a fam c)
goRight :: Loc a fam c → Maybe (Loc a fam c)
```

**Starting navigation**

```
enter   :: ∀fam c a. (Generic a, In a fam, Zipper a fam c)
        ⇒ a → Loc a fam c
```

**Ending navigation**

```
leave   :: Loc a fam c → a
```

**Updating**

```
update  :: (∀b. c b ⇒ b → b) → Loc a fam c → Loc a fam c
```

**Fig. 2.** Generic zipper interface.

satisfy the constraint `c`. The structure of `Loc` (shown in Section 5.2) guarantees that the constraint holds for all types in the family `fam` and, therefore, for all recursive nodes that can be in focus, hence `update` can always be applied.

Consider the following example of usage of the interface. Define a pair of mutually recursive datatypes for a rose tree and a forest, where the forest is a list of trees, and the tree is defined as a value in a node and a forest of its children:

```
data RoseTree a = RTree a (Forest a)

data Forest   a = Empty | Forest (RoseTree a) (Forest a)
```

Updating the trees can be done through a class:

```
class UpdateTree a b where
  replaceBy :: RoseTree a → b → b
  replaceBy _   = id
instance UpdateTree a (RoseTree a) where
  replaceBy t _ = t
instance UpdateTree a (Forest a)
```

This replaces a tree node with a given tree, and, for the forests, this leaves the nodes untouched.

For chaining moves and edits, we can follow Yakushev et al. [27] and employ the flipped function composition ⋙ and Kleisli composition >=>. The latter is instantiated with the `Maybe` monad that wraps the result type of the movement functions.

```
(⋙) :: (a → b) → (b → c) → (a → c)
```

```
(>=>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
```

The type family we need to run the example is defined as follows:

```
type TreeFam a = '[RoseTree a, Forest a]
```

Finally, we can use zipper operations with our updating function to traverse and replace a part of a forest:

```
*Main> let forest
          = Forest (RTree 'a' $ Forest (RTree 'b' Empty) Empty)
                   (Forest (RTree 'x' Empty) Empty)

*Main> let t = RoseTree 'c' Empty

*Main> enter @(TreeFam Char) @(UpdateTree Char)
          >>> goDown >=> goRight >=> goDown
          >=> update (replaceBy t)
          >>> leave >>> return $ forest
```

This yields the following result:

```
Forest (RTree 'a' $ Forest (RTree 'b' Empty) Empty)
       (Forest (RTree 'c' Empty) Empty)
```

Our zipper applies to regular datatypes as well. In that case, `fam` list shall contain a single element. Generally, the interface is flexible enough to allow us to check in any collection of types we are interested in during traversal. However, we demand an updating operation to be a type class function to distinguish the types of the nodes.

### 5.2 Locations

The location structure consists of a focal subtree, which is one of the mutually recursive nodes of the whole structure of the family of datatypes, and its surrounding context:

```
data Loc (r :: *) (fam :: [*]) (c :: * → Constraint) where
  Loc :: Focus r a fam c → Contexts r a fam c
      → Loc r fam c
```

The type parameters `r`, `fam`, and `c` in `Loc` correspond to the root type of the tree, the list of types of nodes to visit, and a constraint imposing restrictions on the types in the list, respectively. Also, the single constructor is existentially quantified over one more type variable, `a`, for we need to store a type of the focus' parent to be able to move up successively in a tree-like structure. We discuss both term parameters of the constructor of `Loc` in detail below.

**Focus** The subtree in focus is wrapped by the `Focus` datatype. The wrapper incapsulates the proofs about a number of important properties of a focus.

```
class ProofFocus (inFam :: Bool) (r :: *) (a :: *) (b :: *)
                 (fam :: [*]) (c :: * → Constraint) where
  witness :: b → Maybe (Focus r a fam c)

instance ProofFocus 'False r a b fam c where
  witness _ = Nothing
instance (Generic b, In b fam, ZipperI r a b fam c)
    ⇒ ProofFocus 'True r a b fam c where
  witness   = Just . Focus

class ProofFocus (InFam b fam) r a b fam c
    ⇒ ProofIn r a b fam c
instance ProofFocus (InFam b fam) r a b fam c
    ⇒ ProofIn r a b fam c
```

**Fig. 3.** Proof of membership of a family of datatypes.

```
data Focus (r :: *) (a :: *) (fam :: [*])
           (c :: * → Constraint) where
 Focus :: (Generic b, In b fam, ZipperI r a b fam c)
       ⇒ b → Focus r a fam c
```

Existential type variable `b` represents the type of a focus. We apply a number of predicates to `b`, hence we can implement the steps of the navigation not knowing the actual type of a focus. Firstly, the type of a focus should have the `Generic` representation. Secondly, it should live `In` the list of types we are going to visit. Lastly, it ought to satisfy the set of constraints for the whole zipper interface captured by the `ZipperI` predicate. In particular, the predicate ensures that `a` is the type of the parent for the focus in the structure under consideration.

We implement the `In` constraint by means of a type family `InFam` exactly along the lines of the `Equal` type family defined in the beginning of Section 3.

```
type In a fam = InFam a fam ∼ 'True
```

The definition of `InFam` is omitted as a boring one.

One last missing piece for managing focuses is the class `ProofIn`. It provides a proof of membership of a focus type to a family. Again, this generalizes the proof of type equality from Section 3. The definitions of `ProofIn` and an auxiliary class `ProofFocus` are given in Figure 3. In the setting of families, where there are no analogues of `:∼:`, we can define only a weak form of proof, which is more flexible.

**Contexts** A focus on a particular node, augmented with a surrounding context of that node, is enough to reconstruct the entire structure. Therefore, the context of a location has the shape of the original structure but with one hole at the place of its focus. This is sometimes called a *one-hole context*.

The context can be expressed as a stack, called `Contexts`, and each frame, `Context`, corresponds to the particular node with a hole. The stack ascends from the focal node keeping its siblings, the siblings of its parent, etc., until it reaches the root node. So the stack of contexts, essentially, reflects the track of the movement inside the structure.

```
data Contexts (r :: *) (a :: *) (fam :: [*])
              (c :: * → Constraint) where
 CNil :: Contexts a a fam c
 Ctxs :: (Generic a, In a fam, ZipperI r x a fam c)
      ⇒ Context fam a → Contexts r x fam c
      → Contexts r a fam c
```

The type parameters have the same meaning as for the `Loc` datatype. The `ZipperI` constraint with the type `x` of the previous context frame indicates that the constraint for the zipper holds after plugging the focus in the hole. Therefore, all the properties can be proved by induction for the focus type when it moves down in the tree adding new contexts onto the stack. The `CNil` constructor for an empty context, with the `r` and `a` types being equal, forms the inductive basis in that kind of proof.

Note that the type of the current focus is not reflected in the `Contexts` datatype.

**Type-level Differentiation** McBride [20] studies a relation between the one-hole context definition and *partial differentiation* from calculus: he shows that the type of the context for an arbitrary (regular) type can be derived mechanically from that type by means of a list of differentiation *rules* that serve as formulaic instructions for computing the type in type-level programming. Yakushev et al. [27] then demonstrate that the method can be generalized for mutually recursive datatypes. We adapt that technique to generics-sop, and now need a few auxiliary type-level functions to implement the computation of the context type. Those functions, defined recursively via type families, provide algebraic operations for lists of types (which we regard as sums and products of types): addition and multiplication. Specifically, we define addition `.++` of two sums of products (SOP) of types, multiplication `.*` of a SOP by a single type, and multiplication `.**` of a SOP by a product of types, as shown in Figure 4.

The addition operation just appends two type-level lists of lists (sums of products), multiplication by a type adds the type to the head of each inner list of the sum (here we see multiplication of a product and the distributive property of multiplication over addition, just as in arithmetic of numbers), and multiplication by a product appends the list to the head of each inner product of the sum. Again, kind `[*]` here denotes products, and `[[*]]` denotes sums (of products), so the relation with arithmetic of numbers in these definitions becomes more clear if one realizes that an empty sum `'[] :: [[*]]` corresponds to 1, and an empty product `'[] :: [*]` corresponds to 0. We also specify, through the `infixr` declaration, that multiplication has a higher priority than addition.

**SOP addition**

```
type family (.++) (xs :: [[*]]) (ys :: [[*]]) :: [[*]] where
  (x ': xs) .++ ys = x ': (xs .++ ys)
  '[]       .++ ys = ys
infixr 6 .++
```

**SOP-by-type multiplication**

```
type family (.*) (x :: *) (ys :: [[*]]) :: [[*]] where
  x .* (ys ': yss) = (x ': ys) ': (x .* yss)
  _ .* '[]         = '[]
infixr 7 .*
```

**SOP-by-product multiplication**

```
type family (.**) (xs :: [*]) (ys :: [[*]]) :: [[*]] where
  (x ': xs) .** yss = x .* (xs .** yss)
  '[]       .** yss = yss
infixr 7 .**
```

**Fig. 4.** Algebraic operations on type-level sums and products.

**Context Frame** At this point, we can implement differentiation of a product of types and, therefore, the computation of a context frame type.

The definition of differentiation, shown in Figure 5, resembles its analogue from calculus, but it is now generalized for the setting of families of datatypes: the differentiation of the single type reflected by a one-element list here results in 0 (reflected by `'[]`) if that is not in the family and hence is regarded as a constant, otherwise it results in 1. When differentiation gives 1, it is actually the hole, which we express by defining the unit type `Hole`. Reflecting this type in the context is helpful when we traverse the context representation to plug the hole. We use the empty type `End` to distinguish the case when the hole is found at the end of the list, in order not to add that into the result twice. The sum `'[ '[]]` represents a unit type that is exactly 1. We also use type-level `If` that returns its second argument for `'True`, and the third one otherwise. We do not give its definition here, as it is straightforward.

The following completes the computation of the context type:

```
data ConsN = F | N ConsN | None
  deriving Eq

type family ToContext (n :: ConsN) (fam :: [*])
                      (code :: [[*]]) :: [[*]] where
  ToContext _ _ '[] = '[]
  ToContext n fam (xs ': xss)
    = Proxy n .* DiffProd fam xs .++ ToContext ('N n) fam xss
```

```
data Hole = Hole
data End

type family DiffProd (fam :: [*]) (xs :: [*]) :: [[*]] where
  DiffProd _   '[]       = '[]
  DiffProd fam '[x]      = If (InFam x fam) '[ '[Hole]] '[]
  DiffProd fam '[End, x] = If (InFam x fam) '[ '[]]     '[]
  DiffProd fam (x ': xs)
    = Hole .* xs .** DiffProd fam '[End, x] .++
              x  .*  DiffProd fam xs
```

**Fig. 5.** Differentiation of a product of types.

The type family `ToContext` derives the type of the context of a datatype performing differentiation of a sum on its code. Since each product from the code matches a sum of multiple products in the context, it is helpful for each product of the context representation, to keep the index of its matching constructor of the datatype. We store the index in the datatype `ConsN` adding that to the head of each product but wrapped by `Proxy` because we use `ConsN` promoted to a kind in the type family, while the products contain types of kind `*`. The constructors `F` and `N` denote "first" and "next", respectively, and the special constructor `None` will be used further to indicate failure of matching indices.

Finally, a context frame has two representations: as a type synonym and as a newtype wrapper:

```
type    CtxCode fam a = ToContext 'F fam (Code a)

newtype Context fam a = Ctx {ctx :: SOP I (CtxCode fam a)}
```

The wrapper allows GHC to perform type inference where it is doomed to fail with a plain type synonym. On the other hand, the `CtxCode` type comes handy in constraints applied to the generic zipper interface functions.

### 5.3 Implementing the zipper interface

We now can implement the interface functions of the zipper, which we have previously described. We only demonstrate the implementation of the `goDown` function here. This shows the idea of how we can use our idiom for defining the zipper functions, and the source code with the full implementation of the zipper interface is available at our GitHub repository[5].

To move focus down to the leftmost child of the current focal node in the tree, we should analyze the focal subtree's representation to find its first immediate child, and compute its respective context. The following definition of `goDown` uses two auxiliary functions: `toFirst` and `toFirstCtx`.

```
goDown :: Loc a fam c → Maybe (Loc a fam c)
```

[5] https://github.com/Maryann13/Zipper

```
toFirst :: ∀fam c r a. (Generic a, ToFirst r a fam c)
        ⇒ a → Maybe (Focus r a fam c)
toFirst t = appToNP @AllProof toFirstNP $ unSOP $ from t
```

**Proof**

```
class    All (ProofIn r a fam c) xs ⇒ AllProof r a fam c xs
instance All (ProofIn r a fam c) xs ⇒ AllProof r a fam c xs

type ToFirst r a fam c = All (AllProof r a fam c) (Code a)
```

**Processing products**

```
toFirstNP :: ∀fam c r a xs. All (ProofIn r a fam c) xs
          ⇒ NP I xs → Maybe (Focus r a fam c)
toFirstNP (I (x :: b) :* xs)
  = witness @(InFam b fam) x ‘mplus‘ toFirstNP xs
toFirstNP Nil = Nothing
```

**Fig. 6.** Implementation of `toFirst`.

```
goDown (Loc (Focus t) cs)
  = case toFirst t of
      Just t’ → Just $ Loc t’ (Ctxs (toFirstCtx t) cs)
      _       → Nothing
```

The function `toFirst` returns its result in the `Maybe` monad, and may return `Nothing`, if the focal node has no children, that is, we currently focus on the leaf node and cannot go down. The function `toFirstCtx` should not fail: if we can move, it computes the context that matches the leftmost subtree selected from the focus' children.

**toFirst** We first implement the function `toFirst`. Its full definition is displayed in Figure 6. The `toFirst` function uses the higher-oredered function `appToNP` that unwraps the product from `NS` and applies the given function to that product. The function `toFirstNP` is defined recursively using the proof we have defined for families: it traverses the product until it finds the first recursive node by means of `witness` that for the node `x` of unknown type `b`, witnesses its membership of the family, or else returns `Nothing`. To provide the proof for the representation code of a datatype, we define the proof for all products in a sum, and pass this proof through explicit type application to `appToNP` which takes a constrained function. The `appToNP` function is defined similarly to `subtermsNS` from Section 2.2, and we omit its definition here.

**toFirstCtx** The definition of `toFirstCtx` is more complicated, as it performs the computation of the context. The implementation comprises several steps including type- and term-level programming. In the following, we systematically construct the context representation from the given generic representation of a datatype.

At first, for a datatype's given constructor represented by NP, we build its matching constructor of the context. All constructors of the context have the same shape as the constructors of its respective datatype but with a hole at one of points of recursion—we are now computing the context with the first recursive node deleted. Assuming that we have the product type for the context computed, we can compute the product by matching on that type:

```
class FromFstRec (ys :: [*]) (xs :: [*]) where
  fromFstRec :: NP I xs → NP I ys

instance FromFstRec (Hole ': xs) (x ': xs) where
  fromFstRec (_ :* xs) = I Hole :* xs
instance FromFstRec ys xs
      ⇒ FromFstRec (x ': ys) (x ': xs) where
  fromFstRec (x :* xs) = x :* fromFstRec xs
instance FromFstRec '[] '[] where
  fromFstRec _ = Nil
```

Note that the first type parameter in the FromFstRec class is the index type list of the result NP—this order of type variables remains for type application through which we will supply the computed type.

Once we have constructed the product, we have to build the sum representing the choice of that product. If we have the index of the chosen constructor for the datatype, the one that we find for its context can be computed as follows:

```
type family CtxConsN (xss :: [[*]]) (n :: ConsN) :: ConsN where
  CtxConsN '[]                        _ = 'None
  CtxConsN ((Proxy n  ': _) ': _  ) n = 'F
  CtxConsN ((Proxy n' ': _) ': xss) n = 'N (CtxConsN xss n)
```

The list xss here is expected to be the context code, which this traverses until the first product storing the constructor index equal to the given one, i. e., the context's first constructor matching with the chosen constructor of the datatype.

To construct the sum with the computed index and product, we again need a proof to witness that the product is type-consistent with the constructor choice. To choose the constructor from the context code, we can adopt generics-sop's *injections*[6]:

```
injections :: ∀xs f. SListI xs ⇒ NP (Inj f xs) xs

newtype Inj f xs a = Inj {apInj :: f a → K (NS f xs) a}
```

For f instantiated to NP I and xs to be the sum of products reflecting a representation code, injections creates a product of all constructor choices that

[6] The actual definition of the type of injections in generics-sop slightly differs from this. We adapt that for presentation.

inject appropriate constructor arguments into each sum. We can use injections to choose the one that matches the obtained index.

We can witness the choice using a variant of the type-safe cast we have defined in Section 3, which takes the result of the equality predicate:

```
typeCastEq :: ∀eq a b. DecideEq eq a b ⇒ b → Maybe a
```

In the following definition, `f` is supposed to be instantiated to `Inj (NP I) xss` where `xss` reflects the code of the context[7]:

```
class    DecideEq (Equal a b) (f a) (f b) ⇒ ProofF f a b
instance DecideEq (Equal a b) (f a) (f b) ⇒ ProofF f a b

class ConsNInj (n :: ConsN) (ys :: [*]) where
  consNInj :: All (ProofF f ys) xss ⇒ NP f xss → f ys

instance ConsNInj n xs ⇒ ConsNInj ('N n) xs where
  consNInj (_ :* xss) = consNInj @n xss
  consNInj Nil        = impossible
instance ConsNInj 'F xs where
  consNInj ((xs :: f xs) :* _)
    = fromMaybe impossible $ typeCastEq @(Equal xs ys) xs
  consNInj Nil = impossible
instance ConsNInj 'None xs where
  consNInj _ = impossible

impossible :: a
impossible = error "impossible"
```

As long as the constructor index and product type are computed correctly, the proof should never fail (`impossible`). And when it passes, this ensures by type check that the constructor choice for the given product type is faithful.

Finally, we define a function that applies the injection and returns the constructed context for the given product, representing its constructor arguments, and index of that constructor. The type of `injections` below will be inferred from the return type of the context.

```
type AppInj n xs ctx = (ConsNInj n xs, SListI ctx,
                        All (ProofF (Inj (NP I) ctx) xs) ctx)

appInjCtx :: ∀n xs fam a. AppInj n xs (CtxCode fam a)
          ⇒ NP I xs → Context fam a
appInjCtx np
  = Ctx $ SOP $ unK $ apInj (consNInj @n injections) np
```

[7] This might be simplified by using a singleton type for `ConsN` instead of defining a type class function. However, when using a singleton, the recursive definition of `toFirstCtx` we give further requires a constraint on its function signature that leads to a nonterminating computation.

The definition of `toFirstCtx` can now be given using the defined functions `appInjCtx` and `fromFstRec`. The following makes use of a type-level function `FstRecToHole` to compute the product type by replacing the first recursive occurrence in the given datatype's product list with type `Hole`. Its definition is straightforward, and is omitted here.

```
toFirstCtx :: ∀fam c a. (Generic a, ToFirstCtx fam a)
           ⇒ a → Context fam a
toFirstCtx t = toFirstCtxNS @'F $ unSOP $ from t

toFirstCtxNS :: ∀n fam a xss. ToFirstCtx' fam a n xss
             ⇒ NS (NP I) xss → Context fam a
toFirstCtxNS (S ns) = toFirstCtxNS @('N n) ns
toFirstCtxNS (Z (np :: NP I xs))
  = appInjCtx @(CtxConsN (CtxCode fam a) n) $
        I (Proxy @n) :* fromFstRec @(FstRecToHole fam xs) np

type ToFirstCtx fam a = ToFirstCtx' fam a 'F (Code a)
```

The `ToFirstCtx'` constraint here is a complex constraint for use of `appInjCtx` and `fromFstRec`, which involves extra type classes and a bit of type-level programming to deduce those particular constraints. Its definition is given in the extended version of the paper in Technical Report. We omit this here for reasons of space.

We have established the mechanism of translation, which is cumbersome but verifying, via the proof, that the constructed context will have the correct type for any given datatype and family. The other functions in the zipper interface can be implemented according to this technique.

## 6 Related work

There are many works that contribute to the datatype-generic programming. Rodriguez et al. [23] and Magalhães and Löh [19] review a number of existing approaches and provide their detailed comparison in various aspects. There are several generic views that use certain forms of the fixed point operator to express recursion in a datatype structure [25, 27, 11, 16]. And there are a number of approaches that do not make use of fixed points [2, 3, 18, 26], but explicitly encode recursion in the datatype representation. The SOP view [5], which we use to demonstrate our technique, is an approach to generic programming that does not reflect recursive positions in the generic representation of a datatype. This approach uses heterogeneous lists of types to encode sums and products in the generic representation.

The idea similar to SOP has been proposed by Kiselyov et al. [13] in their HList library for strongly typed heterogeneous collections. In the paper, the authors also discuss problems connected with overlap, which they use for access operations. Another Haskell extension, functional dependencies, is applied to restrict overlap by introducing a class for type equality there, which resembles our solution.

Morris and Jones [22] introduce the type-class system ilab, based on the Haskell 98 class system, with a new feature called "instance chains". This enables one to control overlap by using an explicit syntax in instance declarations. The approach resembles if-else chains. But the use of instance chains and local use of overlap leave code error-prone as a consequence of type class openness. Closed type families [7] were recently introduced in Haskell to solve the overlap problem.

Several works show how to define the Zipper [10] generically for regular [9, 20] and mutually recursive [27] types using fixed-point generic views. Adams [1] defines a generic zipper for heterogeneous types: a different kind of zipper that can traverse knots of various types, so it does not use the recursive structure.

## 7 Conclusion

Defining generic functions, which consider recursion points, is easy within generic views that are explicit about recursion in the datatype representation. Not so much otherwise. Although, there are some approaches that address the problem by means of global or local overlaps. We have developed the technique that allows one to define generic functions that treat recursion without its explicit encoding and without overlap.

We have demonstrated that the method suits for advanced recursive schemes, such as the generic zipper interface. Also, it supports families of mutually recursive datatypes.

Arguably, it is still easier to treat recursion when "explicit" encoding is used. On the other hand, we believe, once the problem of handling recursion is shown to be manageable, new generic universes shall emerge, not worrying about the recursion support, but rather focusing on other generic programming problems.

## Acknowledgments

We are grateful to Julia Belyakova, who helped to proof-read certain parts of the paper, and to the participants of the Seminar on Programming Languages and Compilers at I.I. Vorovich Institute of Mathematics, Mechanics, and Computer Science (Southern Federal University, Russia), where we presented partial results of the work and got valuable feedback.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 695412).

## References

1. Adams, M.D.: Scrap your zippers: A generic zipper for heterogeneous types. In: Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming. pp. 13–24. WGP '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863495.1863499

2. 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
3. Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 90–104. Haskell '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/581690.581698
4. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of Haskell programs. SIGPLAN Not. **46**(4), 53–64 (May 2011). https://doi.org/10.1145/1988042.1988046
5. De Vries, E., Löh, A.: True sums of products. In: Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming. pp. 83–94. WGP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2633628.2633634
6. De Vries, E., Löh, A.: generics-sop: Generic programming using true sums of products (2018), http://hackage.haskell.org/package/generics-sop
7. Eisenberg, R.A., Vytiniotis, D., Peyton Jones, S., Weirich, S.: Closed type families with overlapping equations. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 671–683. POPL '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2535838.2535856
8. Safe Haskell & overlapping instances—GHC (2015), https://ghc.haskell.org/trac/ghc/wiki/SafeHaskell/NewOverlappingInstances
9. Hinze, R., Jeuring, J., Löh, A.: Type-indexed data types. Science of Computer Programming **51**(1), 117–151 (2004). https://doi.org/10.1016/j.scico.2003.07.001, Mathematics of Program Construction (MPC 2002)
10. Huet, G.: The zipper. Journal of Functional Programming **7**(5), 549–554 (1997)
11. Jansson, P., Jeuring, J.: Polyp—a polytypic programming language extension. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 470–482. POPL '97, ACM, New York, NY, USA (1997). https://doi.org/10.1145/263699.263763
12. Kiselyov, O.: Type equalities, disequalities and obsoleting of overlapping instances (2012), http://okmij.org/ftp/Haskell/typeEQ.html
13. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell. pp. 96–107. Haskell '04, ACM, New York, NY, USA (2004). https://doi.org/10.1145/1017472.1017488
14. Löh, A.: Exploring Generic Haskell. Ph.D. thesis, Utrecht University (2004)
15. Löh, A.: Applying type-level and generic programming in Haskell (2018), https://github.com/kosmikus/SSGEP/blob/master/LectureNotes.pdf, Summer School on Generic and Effectful Programming (SSGEP 2015)
16. Löh, A., Magalhães, J.P.: Generic programming with indexed functors. In: Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming. pp. 1–12. WGP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2036918.2036920
17. Magalhães, J.P.: Less Is More: Generic Programming Theory and Practice. Ph.D. thesis, Utrecht University (2012)
18. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for Haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell. pp. 37–48. Haskell '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863523.1863529
19. Magalhães, J.P., Löh, A.: A formal comparison of approaches to datatype-generic programming. In: Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012. pp. 50–67 (2012). https://doi.org/10.4204/EPTCS.76.6

20. McBride, C.: The derivative of a regular type is its type of one-hole contexts (2001), http://strictlypositive.org/diff.pdf, unpublished manuscript
21. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture. pp. 124–144. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
22. Morris, J.G., Jones, M.P.: Instance chains: Type class programming without overlapping instances. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. pp. 375–386. ICFP '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1863543.1863596
23. Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., Oliveira, B.C.d.S.: Comparing libraries for generic programming in Haskell. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell. pp. 111–122. Haskell '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1411286.1411301
24. basic-sop: Basic examples and functions for generics-sop (2017), https://hackage.haskell.org/package/basic-sop
25. Van Noort, T., Rodriguez, 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. WGP '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1411318.1411321
26. Weirich, S.: Replib: A library for derivable type classes. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell. pp. 1–12. Haskell '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1159842.1159844, http://doi.acm.org/10.1145/1159842.1159844
27. Yakushev, A.R., 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. ICFP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1596550.1596585
28. 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. TLDI '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2103786.2103795