# 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 a recursive structure. Some approaches based on sums of products, however, do not use a fixed point. Those views therefore allow for generic functions that do not require to look at the recursive knots in a datatype representation, but raise issues when it needs to deal with recursion. A known and unwelcome solution is the use of overlapping instances. We present an approach that uses closed type families to eliminate the need of overlap for handling recursion. Moreover, we show that our idiom allows for families of mutually recursive datatypes.

**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 both are 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"—this is implemented in the generics-sop [7] 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 for constructing sums and products, collapsing to homogeneous structures, and application, which are an expressive instrument for defining generic functions in a more succinct and high-level style compared to the classical binary sum-of-products views.

There are many generic functions that deal with the recursive knots, when traverse the structure of datatypes. Some of the most general examples are *maps* [18] and *folds* [21], more advanced one is a *zipper* [11, 10, 1]. For handling recursion, several generic programming approaches express datatypes in the form of polynomial functors closed under fixed points [27, 12, 16]. The SOP view naturally supports definitions of functions that do not require a knowledge about recursive occurrences, but otherwise it yields.

One possible solution to the aforementioned shortcoming of SOP is to modify the SOP core by explicitly encoding recursive positions using the fixed-point approach. 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` for GHC does not reflect unsafe overlaps and marks the module as safe [9].

We feel both existing approaches unsatisfactory 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 4).
- 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 [6] package, is discussed in Section 3.2.

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

```
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.

## 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* [10] 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 tiny abstraction of a usual definition of a heterogeneous list The definitions of `NS` and `NP` are kind polymorphic. The index list allows 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 of types, while `K a` turns it into homogeneous one. Here is an example value of `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 3))) :: 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—it does not recursively translate the constructor arguments.

We are leaving out discussion of the `All SListI` constraint in the `Generic` class definition as irrelevant to our work. Although, we do use `All` constraint combinator in the following. Therefore, it is worth noting that `All` applies 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 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. We reimplement this function using the SOP view:

```
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)
  | typeEq @a y  = witnessEq 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` that merely traverses the sum until it reaches the product. Once it gets the product, it passes that 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 the given term's, otherwise skipping the element. We use GHC's TypeAppications extension to pass that type.

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 and 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 _ = []
```

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. That would avoid the complexity ovehead introduced by the approach, as we have noted in the introduction.

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

## 3 Handling recursion with closed type families

In the previous section, we have shown a solution to the problem of handling recursion, which makes use of 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. [8]. The main idea of the extension being 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 a x = '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 `subterms` revised

We now return to our running example from Section 2.2. With the type equality, we can witness the coercion between the equal types by defining a type class:

```
class Proof (eq :: Bool) (a :: *) (b :: *) where
  witnessEq :: b → Maybe a

instance Proof 'False a b where
  witnessEq _ = Nothing
instance Proof 'True  a a where
  witnessEq   = Just
```

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 heavy-weighted interface of `Proof`:

```
class    Proof (Equal a b) a b ⇒ ProofEq a b
instance Proof (Equal a b) a b ⇒ ProofEq 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 witnessEq @(Equal a x) y of
      Just t  → t : subtermsNP ys
      Nothing → subtermsNP ys
subtermsNP _ Nil = []
```

As a side note, we make use of 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 [6] 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 the 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 `GShow` synonym:

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

gshow :: ∀a. GShow a ⇒ a → String
```

The function `gshow` employs metadata 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 [6] 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 The generic Zipper

The zipper is a data structure that enables efficient editing and navigation within the tree-like structure of a datatype by representing a current location in that structure. The location is a focus, which can be one of recursive nodes in the tree, along with its context that consists of surroundings of the focal subtree.

The classical zipper described by Huet [11] can be generically produced for regular datatypes [10], which are a subset of datatypes that can be viewed as a least fixed point of some polynomial expression on types. Yakushev et al. [27] generalize the definition of the generic zipper for an arbitrary family of mutually recursive datatypes. Those known solutions require a datatype to be expressed using forms of a fixed-point operator, since the zipper operates on recursion points. Using closed type families, it is possible to define the generic zipper using a representation that does not exploit a fixed point. In this section, we implement the generic zipper interface that includes functions for manipulating locations for mutually recursive datatypes using the SOP view.

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

**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.

### 4.1 Interface and using

Having the location of the zipper, which holds the current focus on one of recursive knots in the structure—for the zipper for mutually recursive datatypes, this means recursion points of the total structure of a family—and its context, we may produce a new location by moving that focus up, down, left, or right. We are first going to show the generic zipper interface and an example of how it can be used, and then proceed to implement that interface. The interface we provide for using the generic zipper is displayed in Figure 2. This 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. Movement may fail, as specified by the `Maybe` monad, if we cannot go further in a chosen direction.

The function signature of `enter` specifies the constraints necessary for starting navigation in a structure. A datatype of the structure needs to have the generic representation; the `In` constraint is defined over a generalization of `Equal` from Section 3, which checks a type for membership of a family; the `Zipper` constraint collects specific constraints that refer to the implementation of movement operations. 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. Navigation in a tree starts at the root, and the type variable `a` refers to the root type that keeps fixed during the navigation, since it is necessary for leaving a tree and returning the root, while type in focus of the location may vary and is one of types in the type list `fam` (of kind `[*]`) specifying a family. The `leave` function ends navigation moving up to the root and returns its modified value.

The function `update` modifies the focal subtree with a constrained function, where the type in focus is existential in `Loc`, and the datatype of locations guarantees that the constraint `c` (actually, it has kind `* → Constraint`) holds for all datatypes in the family `fam` and therefore for all recursive nodes that can be in focus, whence the function can be applied.

Let us show how to use this with an example. Consider the following 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 the node and a forest of its children:

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

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

We can define a class that provides a function for updating the trees, for example:

```
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 the tree node with a given tree, and for the forests, this leaves the nodes without change. This is also scalable if we want to extend the family.

For chaining moves and edits, we can follow Yakushev et al. [27] and exploit the flipped function composition ⋙ and Kleisli composition >=>, using the monad interface of `Maybe` 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)
```

We now define the family of our datatypes simply as follows:

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

And the example below shows how we can use the zipper operations with our updating function to change a mutually recursive expression:

```
*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)
```

We also may use the zipper merely for regular datatypes, when we want to move only to recursive knots of a single datatype, just assigning a one-element type-level list to the family at the `enter` call. Or we may enumerate all types, whose nodes we want to visit, in that list. This is flexible, as this requires no boilerplate declarations. However, we demand an updating operation to be a type class function to distinguish types of the nodes, and enforce a constraint, which may be single or complex, on all types in the family, because the type of the focus is existential, as already said.

## 4.2 Locations

The location structure consists of the focal subtree, which is one of the mutually recursive nodes of total structure of the family of datatypes, and its surrounding context. We give the following definition of the datatype `Loc` for locations:

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

The subtree in focus is wrapped by the `Family` datatype that we use to prove its required properties, while we do not know the current type of the node in focus. For being able to move up in the tree, we keep the type that will be obtained by uniting the current focus and context (essentially, the type of the focal node's parent in the tree): that type is unknown outside the location, and is reflected in the type parameters of `Family` and the `Contexts` datatype of the context by the type variable `a` that is existential in the `Loc` datatype; the type of the focal subtree itself is existential in `Family`. The other type parameters `r`, `fam`, and `c` in `Loc` mean the root type of the tree, the list of types constituting the family, and the constraint constructor imposing restrictions on the family's types, respectively.

**Families** We define the `Family` datatype as a GADT, with embedded constraints on its constructor:

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

Though we do not know the type of the focus that is the existential type `b` in the family, we can obtain enough information about that type to provide the zipper functionality, which is captured by the restrictions above. We guarantee that the type `b` is a member of the family, and satisfies a set of constraints of

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

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

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

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

the whole zipper interface, collected in the `ZipperI` constraint: in particular, it ensures that `a` is the type of the parent node for the node of focus of type `b` in the tree.

We define the `In` constraint by means of a type family `InFam` that is a generalization of `Equal` defined in Section 3. The following checks a type for membership of a type-level list that we use to express a family:

```
type family InFam (a :: k) (fam :: [k]) :: Bool where
  InFam a (a ': fam) = 'True
  InFam a (x ': fam) = InFam a fam
  InFam a '[]        = 'False
```

The `In` constraint is defined as

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

Together with the datatype of families, we define a class `ProofIn` that provides a proof of membership of a family, which generalizes the proof of type equality from Section 3. The definition of the `ProofIn` class and its auxiliary class `ProofFam` is given in Figure 3. The constraints on the second `ProofFam` instance, which repeat the same ones from the `Family` constructor, constitute the proof.

**Contexts** When we focus on a particular node in a data structure, having the surrounding context of that node is enough to reconstruct the entire structure. The context for the tree's location has the shape of the original structure of that tree but with one hole at the place of its focus, whence it is sometimes called a *one-hole context*. The context can be expressed as a stack (`Contexts` below), where each frame `Context` corresponds to the particular node with a hole: it ascends from the focal node keeping its siblings, the siblings of its parent, and until it reaches the root node, through all levels in the tree. So the stack of contexts essentially reflects the track of the movement in 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
```

This has the shape similar to the previously defined `Loc` and `Family` datatypes: it keeps the types of the root and of the immediate parent of the focal node (of the union with the focal subtree when that plugs the hole)—as ensured by sharing that type with `Family` in the `Loc` constructor. The `ZipperI` constraint with the type `x` of the previous context frame here indicates that the constraint for the zipper holds after plugging the hole, so all the properties, conversely, can be proved by induction for the focus type when it moves down in the tree adding new contexts onto the stack—and the `CNil` constructor for an empty context with the `r` and `a` types equal is the inductive basis in that proof. Note that the type of the current focus is not reflected in the `Contexts` datatype, as we do not need that.

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 his 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.

Using the defined operations, we can implement differentiation of a product of types. 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:

**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)
  x .* '[]         = '[]
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.

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 Hole = Hole
data End

type family DiffProd (fam :: [*]) (xs :: [*]) :: [[*]] where
  DiffProd fam '[]       = '[]
  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.

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

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

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.

We finally define `Context` as a newtype wrapping the result of the computation, as it allows GHC to perform type inference for the context type, where it is possible, to avoid extra type applications for defining the zipper operations.

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

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

The `CtxCode` type is the computed type of the context for the given code of a datatype. We will use this type synonym further when defining constraints for functions providing the generic zipper interface.

## 4.3 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)
goDown (Loc (Family t) cs)
  = case toFirst t of
      Just t' → Just $ Loc t' (Ctxs (toFirstCtx t) cs)
      _       → Nothing
```

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

```
toFirst :: ∀fam c r a. (Generic a, ToFirst r a fam c)
        ⇒ a → Maybe (Family 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 (Family 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`.

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 '[]                           n = 'None
  CtxConsN ((Proxy n  ': xs) ': xss) n = 'F
  CtxConsN ((Proxy n' ': xs) ': 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 inject appropriate constructor arguments into each sum. We can use injections to choose the one that matches the obtained index.

We now can witness the choice using the proof of type equality we have defined in Section 3. In the following definition, `f` is supposed to be instantiated to `Inj (NP I) xss` where `xss` reflects the code of the context[7]:

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

[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

```
class    Proof (Equal a b) (f a) (f b) ⇒ ProofF f a b
instance Proof (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 $ witnessEq @(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
```

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
```

`toFirstCtx` we give further requires a constraint on its function signature that leads to a nonterminating computation.

```
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.

## 5 Related work

Over the years, there has been a lot of works that contribute to datatype-generic programming, and a lot of knowledge has been accumulated. Rodriguez et al. [23] and Magalhães and Löh [19] review a number of existing approaches and provide their detailed comparison in different aspects. There are several generic views that use forms of the fixed point operator to express recursion in a datatype structure [25, 27, 12, 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 their paper, they also discuss problems concerened with overlap which they use for access operations. They restrict overlap by introducing a type class using a functional dependency, which provides functionality of type equality that is similar to 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", that allows to control overlap by using an explicit syntax in instance declarations, which resembles if-else chains. But use of instance chains and local use of overlap leave code error-prone as a consequence of type class oppennes. Closed type families [8] were recently introduced in Haskell to solve the overlap problem.

Several works show how to define the Zipper [11] generically for regular [10, 20] and mutually recursive [27] types using fixed-point generic views. Adams [1] defines a generic zipper for heterogeneous types.

## 6 Conclusion

It was believed until today, that defining generic functions that consider recursion points when looking at the generic representation of a datatype structure is only possible within generic views that explicitly express recursion in the datatype representation, whereas the other commonly used approaches to generic programming was considered as unsuitable for this task. There are some approaches that address this problem by means of local overlaps, where it needs to access the points of recursion. We have developed the technique that allows us to define generic functions that treat recursion without its explicit encoding and overlap. This makes use of closed type families that are the recent Haskell extension.

We have demonstrated that the method suits for advanced recursive schemes, such as the generic zipper interface, and supports families of mutually recursive datatypes. Though it is still easier to treat recursion where the explicit encoding is used, once we have shown that the problem of handling recursion in datatype-generic approaches is not critical, it encourages to invent new generic universes not worrying about the recursion support, but rather focusing on other generic programming tasks.

## 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.: basic-sop: Basic examples and functions for generics-sop (2017), https://hackage.haskell.org/package/basic-sop
7. De Vries, E., Löh, A.: generics-sop: Generic programming using true sums of products (2018), http://hackage.haskell.org/package/generics-sop
8. 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
9. Safe Haskell & overlapping instances—GHC (2015), https://ghc.haskell.org/trac/ghc/wiki/SafeHaskell/NewOverlappingInstances

10. 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)
11. Huet, G.: The zipper. Journal of Functional Programming **7**(5), 549–554 (1997)
12. 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
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. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for gadts. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. pp. 341–352. ICFP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1596550.1596599
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