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] Northeastern University, Boston, USA
artem@ccs.neu.edu

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 other approaches, especially the ones based on sums of products, avoid the fixed point encoding. They facilitate implementation of generic functions that do not require to look at the recursive knots in a datatype representation, but raise issues otherwise. A widely used and unwelcome solution to the problem resorts to overlapping instances. In this paper, 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 this idiom allows for generic programming with 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 [28, 5, 16]. De Vries and Löh [7] 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 [8] library and is based on several relatively recent extensions to the Haskell type system, such as *data kinds*, *kind polymorphism* [31] 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 [20] and *folds* [24]; a more advanced one is a *zipper* [12, 11, 1]. For handling recursion, several generic programming approaches express datatypes in the form of polynomial functors closed under fixed points [30, 13, 18]. However, the SOP view does not reflect recursion points in generic representation types. So it 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: the original SOP encoding and the modified one.

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 [25] extensively discuss the problems arising from overlapping instances. One of those problems is late error reporting, which is a consequence of the fact that GHC resolves overlapping instances at call sites. That is, if there are more than one most specific instances for particular types, GHC does not determine that before one's attempt to use the class function with those types. Another notable problem is lack of specification for overlapping instances, so their behavior depends on a concrete compiler. 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 [10].

The problem of using overlapping instances was first addressed by Kiselyov et al. [15]. 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. That variant and its later implementation with type families [14] fully eliminate OverlappingInstances. Despite this, 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. In the area of generic programming, the extension does not seem to be widely leveraged yet. Exploiting this observation, we 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 [27] package, is discussed in Section 3.2.

We believe that the idea presented is suitable for any sum-of-products approach that does not employ the fixed point view and thus subject to the problem. We choose the generics-sop library 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 [7, 17] 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 represents an n-ary product, isomorphic to a sequence of constructor arguments, while the outer list, representing 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* [11] 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

```
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.

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 constructors S and Z of NS allow to represent choices from a sum as indices in the form of Peano numbers. Firstly, the application of Z represents the choice of the first component of the sum. Secondly, S is the successor constructor. That is, $S\ .\ Z$ chooses the second component, and so on up to the length of the index list xs. Given a sum of type $NS\ f\ xs$, the i-th choice has type $f\ x$, where x is the i-th element in xs. For example, the following represents the third choice from a sum:

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

Putting this together, each term of a particular datatype can be represented as the choice from the sum of products, representing that datatype. The choice 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)
newtype SOP f a = SOP {unSOP :: NS (NP f) 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.

The *SListI* constraint in the *Generic* class definition is used in generics-sop to represent type-level index lists at the level of terms as singletons. We leave out discussion of this constraint, as it is 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 [6] 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 We outline one possible approach to SOP-based implementation of *subterms* below. It gives a clear idea of our intention, but so far, we miss the necessary toolbox (in particular, *typeOf* / *castEq*) to finish it.

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

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

subtermsNP :: ∀a xs. NP I xs → [a]
subtermsNP (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 the current element to the result list if its type is the same as of the original term, otherwise skipping the element. We use GHC's TypeApplications extension to pass that type.

Note that our implementation uses plain pattern matching for reasons of simplicity. However, in Section 3.1, we show a higher-level solution using one of the primary advantages of generics-sop—high-level traversal functions.

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 *subtermsNP*, 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 to release 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. [9]. 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
```

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

```
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 *subterms* revised

We now return to our running example from Section 2.2. With the type equality predicate, we can decide if $a \sim b$ 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 $:\sim:$ 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 dedicated 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 *castEq* (Section 2.2). The definition makes use of *castWith*, which performs a type-safe cast between two equal types, using the equality witness $b :\sim: a$:

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

Below we show two variants of completing the *subterms* implementation. The low-level version performs pattern matching on the structure of *NS* and *NP*, as we do in Section 2.2. For this variant, we reimplement *subtermsNP*. The different implementation employs powerful machinery of high-level combinators—one of the major advantages provided by generics-sop. Although, those high-level functions become one extra layer of complexity.

Low-level implementation The low-level definition of *subtermsNP* using the type cast resembles its outline 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 finish up, the *ProofEq* constraint must be added to the *subterms* and *subtermsNS* declarations as well.

High-level implementation By taking advantage of generics-sop's powerful functions for collapsing and mapping, one can define the functions *subtermsNS* and *subtermsNP* as follows:

```
subtermsNS :: ∀a xss. All (AllProofEq a) xss
           ⇒ NS (NP I) xss → [a]
subtermsNS = collapse_NS
           . cmap_NS (Proxy @(AllProofEq a)) subtermsNP

subtermsNP :: ∀a xs. AllProofEq a xs ⇒ NP I xs → K [a] xs
subtermsNP = K . catMaybes
               . collapse_NP
               . cmap_NP (Proxy @(ProofEq a)) (mapIK $ castEq)
```

This implementation requires also defining *AllProofEq* and properly changing the *subterms* type annotation.

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 the task of *subterms*. We give other examples showing how to adapt our idiom to different scenarios in the following subsections.

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 [27] based on the SOP view. We follow their implementation of *gshow* for the most part, but improve it in respect of handling recursion. The example shows, how the better implementation, consulting with recursive positions, affects the usability of the function, obviating boilerplate code.

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 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' = gshow
```

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 = caseShow' @(Equal a b) @a
```

The resulting function *gshow* is subject to 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 implementation 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)"
```

Our implementation of *gshow* obviates one drawback of its analogue from basic-sop. This drawback is that *gshow* from basic-sop does not treat recursive positions separately, and therefore requires the *Show* constraint for all knots in the datatype structure:

```
gshow :: ∀a. (Generic a, HasDatatypeInfo a, All2 Show (Code a))
      ⇒ a → String
```

As a consequence, basic-sop offers the following usage pattern for *gshow* and some datatype *T*, requiring an additional *Show* instance declaration for *T*:

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

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

As we have shown, the variant of the function *gshow*, distinguishing the recursion cases, can be used directly, without this extra declaration. This takes advantage of employing the polytypic instance of *CaseRecShow* for all datatypes.

3.3 Generic recursion schemes

Recursion schemes, such as *fold* [24, 30] and *compos* [3], are classical examples of generic functions that are usually treated by a fixed point view. In this subsection, we show how they can be defined in the SOP view, adopting our approach.

Generic *compos* The function *compos* is a traversal combinator for defining compositional functions. It applies a given function to all immediate children of a given term ot type *T*:

```
compos :: (T → T) → T → T
```

This can be used, for example, with a datatype *Expr* for defining the *renameVar* function, updating all variables in a given expression without affecting its other subexpressions:

```
data Expr = EAbs String Expr | EApp Expr Expr | EVar String

renameVar :: Expr → Expr
renameVar (EVar x) = EVar $ x ++ "_"
renameVar e        = compos renameVar e
```

The implementation of the generic function *gcompos* employs a class for resolving recursion cases, which now looks as the following:

```
class CaseCompos_Aux (eq :: Bool) (a :: ∗) (x :: ∗) (y :: ∗) where
  caseCompos_Aux :: (a → a) → I x → I y

instance CaseCompos_Aux 'False a x x where
  caseCompos_Aux _ = id
instance CaseCompos_Aux 'True  a a a where
  caseCompos_Aux f = I . f . unI
```

The function *caseCompos*, an abbreviation for $caseCompos_{Aux}$ @(*Equal a x*), is supposed to be applied to each component of the product, representing arguments of a chosen constructor. Given a component of type *I x*, it results in the component of different type *I y*. (The implementation below makes use of a higher-order function, whose type does not admit that *x* and *y* can be unified here.) The first case in this definition corresponds to mapping over a constant, which should not be changed. The second case is the application of the given function *f* to a recursive position.

The rest of the work is handled by generics-sop's function *trans_SOP*:

```
trans_SOP :: AllZip2 c xss yss
          ⇒ proxy c → (forall x y. c x y ⇒ f x → g y)
          → SOP f xss → SOP g yss
```

This function transforms the generic representation of a term, applying a given function to every component of a chosen product. *AllZip2* here is analogous to *All2*, zipping two lists of lists with the binary constraint *c*.

Finally, the definition of *gcompos*, using *trans_SOP* with *caseCompos*, is following, with *CaseCompos* abbreviating the use of $CaseCompos_{Aux}$:

```
type GCompos a = (Generic a, AllZip2 (CaseCompos a) (Code a) (Code a))

gcompos :: ∀a. GCompos a ⇒ (a → a) → a → a
gcompos f = to . composSOP . from
  where
    composSOP = trans_SOP (Proxy @(CaseCompos a)) $ caseCompos f
```

Generic *fold* A fold is a higher-order function that recursively deconstructs a term, using a combining operation on its structure to aggregate the result. For example, consider a fold for a datatype of arithmetic expressions *AExpr*:

```
data AExpr = EConst Int | EAdd AExpr AExpr | EMul AExpr AExpr

fold_AExpr :: (Int → r, r → r → r, r → r → r) → AExpr → r
fold_AExpr f@(f1, f2, f3) e = case e of
  EConst x → f1 x
  EAdd l r → f2 (fold_AExpr f l) (fold_AExpr f r)
  EMul l r → f3 (fold_AExpr f l) (fold_AExpr f r)
```

Given *f*—called an *algebra*—it proceeds through the recursive structure of a given term *e*, applying its components f_1, f_2, and f_3 to elements of *e* and combining the results. This can be used to evaluate expressions. Consider:

```
example = EAdd (EMul (EConst 3) (EConst 2))
               (EAdd (EMul (EConst 2) (EConst 2))
                     (EConst 5))
```

The call

```
fold_AExpr (id, (+), (*)) example
```

yields 15.

In the example above, $(Int \to r,\ r \to r \to r,\ r \to r \to r)$ is a type of algebras for the datatype *AExpr* to a result type *r*. This type represents the underlying recursive structure of *AExpr*. In particular, the constructors of *AExpr* form an algebra for expressions, which is isomorphic to *AExpr*:

```
(EConst, EAdd, EMul) :: (Int → AExpr,
                         AExpr → AExpr → AExpr,
                         AExpr → AExpr → AExpr)
```

A type of algebras for a particular datatype a can be computed generically from its representation code. The result type r should occur in this type whenever a (a recursive position) occurs in the code. In the following, the type families *AlgebraS* and *AlgebraP* are employed to compute a type of algebras for a given datatype a and a result type r.

```
type family AlgebraS (code :: [[*]]) a r :: [*] where
  AlgebraS (xs ': xss) a r = (AlgebraP xs a r ': AlgebraS xss a r)
  AlgebraS '[]           _ _ = '[]

type family AlgebraP (xs :: [*]) a r where
  AlgebraP (a ': xs) a r = r -> AlgebraP xs a r
  AlgebraP (x ': xs) a r = x -> AlgebraP xs a r
  AlgebraP '[]       _ r = r

type Algebra a r = NP I (AlgebraS (Code a) a r)
```

The calculated type of algebras *Algebra a r* is meant to be an *NP* of its components, which are constructed by *AlgebraP* from each product in the code. All the components, being applied to the elements of a datatype structure, return a result of type r, as handled by the last case in the *AlgebraP* definition.

We turn to defining a generic fold. First, define a class $CaseFold_{Aux}$ with a function $caseFold_{Aux}$, which perform the main work and serve for managing the two recursion cases. As before, we will use the synonyms *CaseFold* and *caseFold* to abbreviate their use with the application of *Equal a x*:

```
class CaseFold_Aux (eq :: Bool)
                   (a :: *) (b :: *) (x :: *) (y :: *) where
  caseFold_Aux :: Algebra a b -> I x -> I y

instance CaseFold_Aux 'False a b x x where
  caseFold_Aux _ = id
instance GFold a b => CaseFold_Aux 'True a b a b where
  caseFold_Aux f = I . applyAlgebra @a f . foldSOP . from . unI
    where
      foldSOP = trans_SOP (Proxy @(CaseFold a b)) $
                          caseFold @a @b f
```

The function $caseFold_{Aux}$ is meant to operate on the elements of a product constituting a part of a datatype's representation. For constants, there are no recursive calls, so an operation for that case is trivially the identity function. Whereas in the case of the recursion point, this turns the element to its representation, then recursively processes that, and applies the given algebra to the result. This employs the function *applyAlgebra* that applies the algebra to the representation gained as a result of processing:

```
applyAlgebra :: ∀a b. ApplyAlgebra a b
             => Algebra a b -> SOP I (AlgCode a b) -> b
```

That representation has the code, calculated from the origin code of a by replacing its recursive occurences therein with the algebra's result type b. *AlgCode a b* here is a type synonym for the result of this computation. We omit the implementation of *applyAlgebra*, as well as of the type-level machinery of *AlgCode* and *ApplyAlgebra*, for the sake of space.

The type applications $@a$ and $@b$ appear in the lines above, because they are needed to calculate the *Algebra a b* type. Also, the constraint *GFold a b* on the second instance of $CaseFold_{Aux}$ is necessary to use *applyAlgebra*, *caseFold*, and *from*:

```
type GFold a b = (Generic a
                 , AllZip2 (CaseFold a b) (Code a) (AlgCode a b)
                 , ApplyAlgebra a b)
```

The generic fold function *gfold* is defined using *caseFold*:

```
gfold :: ∀a b. GFold a b ⇒ Algebra a b → a → b
gfold f = unI . caseFold @a @b f . I
```

This can be conveniently used with various algebras that are now of type *NP*, if we define an infix operator & for constructing products:

```
infixr 1 &
x & ys = I x :* ys
```

The call of *gfold* with *example*, defined above in this subsection,

```
gfold (id & (+) & (*) & Nil) example
```

again yields 15.

3.4 Abstract description of the design pattern

We can continue along the lines of the previous examples to consider the approach separately from concrete ones. One can use the technique, following the abstracted pattern below:

```
class DispatchRec_Aux (p :: Bool)
                      (a_1 :: k_1) ... (a_n :: k_n) (b :: *) where
  dispatchRec_Aux :: X b

instance C_1 ... ⇒ DispatchRec_Aux 'False a_1 ... a_n b where
  dispatchRec_Aux = f_1
instance C_2 ... ⇒ DispatchRec_Aux 'True  a_1 ... a_n b where
  dispatchRec_Aux = f_2

class     DispatchRec_Aux (P a_1 ... a_k b) a_1 ... a_k ... a_n b
  ⇒ DispatchRec a_1 ... a_n b
instance DispatchRec_Aux (P a_1 ... a_k b) a_1 ... a_k ... a_n b
  ⇒ DispatchRec a_1 ... a_n b

dispatchRec :: ∀a_1 ... a_n b. DispatchRec a_1 ... a_n b ⇒ X b
```

$$dispatchRec = dispatchRec_{Aux}\ @(P\ a_1\ \ldots\ a_k\ b)\ @a_1\ \ldots\ @a_k\ \ldots\ @a_n$$

This machinery employs the abstract multi-place predicate P on types, supposed to be a closed typed family, to resolve the overlap of two dispatch branches, as captured by $DispatchRec_{Aux}$. The predicate P generalizes *Equal* to an arbitrary relation on types, so that one can model, e.g., subtyping for some domain-specific language. The introduction of the associated type class *DispatchRec* provides a handy interface for dispatch. The dispatch function *dispatchRec* decides on one of the functions f_1 and f_2—depending on whether the predicate P is true for types $a_1, \ldots, a_k, b$. Here, the type b is supposed to be an instance of *Generic*.

The type $X\ b$, associated with b, depends of the kind of the generic function. All generic functions may be classified into three categories [26]: *consumers*, *transformers*, and *producers*. Given type b, an instance of *Generic*, they turn b into a constant type, b into b with a changed value, and a constant type into b, respectively. For consumer functions, such as *gshow* (Section 3.2), the type $X\ b$ typically is instantiated with $b \to T$ for some datatype T. For transformers, such as *gcompos* (Section 3.3), $X\ b$ typically takes the form $S \to I\ a_i \to I\ a_j$. Here, a_i, a_j for $i \neq j$ are from $a_1\ \ldots\ a_n$, and are supposed to be equal to b at recursion points, and S is some type. This suggests that the transformer function changes b, employing values of type S. The pattern fits equally well for defining producers, such as *garbitrary* from the basic-sop package, where $X\ b$ would be instantiated with $Gen\ b$.

4 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 [12] can be generically calculated for regular datatypes [11]—all datatypes expressible as polynomial expressions on types. Yakushev et al. [30] 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 4.1). Then, we turn to the type-level machinery employed to define locations inside mutually recursive datatypes using the SOP view (Section 4.2).

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.

In a technical report, we discuss the implementation of the generic zipper interface—the functions for manipulating locations. The source code with the full implementation of the zipper is available at our GitHub repository[5].

4.1 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.

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

The *update* function modifies the focal subtree with a given constrained function. The type in focus is existentially quantified inside *Loc* and should satisfy the constraint *c*. The structure of *Loc* (shown in Section 4.2) 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. [30] 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 rely on a type class function to distinguish the types of the nodes.

4.2 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 the term parameters of the constructor of *Loc* in detail below.

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

```
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 without 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. Also, it guarantees that *b* fulfils the constraint *c*.

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 it is a boring one.

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

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

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

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

One last missing piece for managing focuses is the class *ProofFocus*. It provides a proof of membership of a focus type to a family. Again, this generalizes (in a weak form) the proof of type equality from Section 3.1. The definitions of *ProofFocus* and an auxiliary class $ProofFocus_{Aux}$ are given in Figure 3. In the setting of families, where there are no analogues of :∼:, we can define only a weak kind of proof, which is more flexible. We can then make it handy by defining a function *castFocus* using $castFocus_{Aux}$, as with *castEq* before.

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 [23] 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. [30] 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 introduce 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. The definition of the latter uses $.*$.

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. We define multiplication by a type below. The definitions for the other type-level operations are similar.

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

Again, kind $[*]$ denotes products, and $[[*]]$ denotes sums (of products), so the relation with arithmetic of numbers becomes more clear if one realizes that an empty sum $'[] :: [[*]]$ corresponds to 0, and an empty product $'[] :: [*]$ corresponds to 1.

Context Frame At this point, we can implement differentiation of a product of types and, therefore, the computation of a context frame type[6].

The definition of differentiation resembles its analogue from calculus, but it is now generalized for the setting of families of datatypes:

```
type family DiffProd (fam :: [*]) (xs :: [*]) :: [[*]] where
  DiffProd _   '[]      = '[]
  DiffProd fam '[x]     = If (InFam x fam) '[ '[]] '[]
```

[6] The actual definitions slightly differ from ones, presented in the following. We omit some implementation details for simplicity.

```
DiffProd fam (x ': xs) =
  xs .** DiffProd fam '[x] .++ '[x] .** DiffProd fam xs
```

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 represented by the sum `'[ '[]]`. When differentiation gives 1, it is actually the hole. 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:

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

The type family *ToContext* derives the type of the context of a datatype performing differentiation of a sum on its code.

Finally, the type of a context frame representation is a type synonym:

```
type Context fam a = SOP I (ToContext fam (Code a))
```

5 Generic generic programming

In the previous sections, we have shown that many generic functions that treat recursion can be defined, following one pattern, in the SOP view, which does not employ a fixed point. Despite this, some generic functions can be defined in a more elegant way, when using a fixed point view. Furthermore, there are such ones, for which a fixed point view is essential. A prominent example of this is generic pattern-matching [28]. It involves generic definitions of patterns for datatypes, which extend their nodes with a metavariable environment. A pattern type is defined as a sum of a datatype's representation functor and a type of variables, closed under a fixed point. This, therefore, cannot be builded out of the SOP generic representation of a datatype.

Still, one can efficiently employ these views together in one program, taking advantages of the both. Combining the use of different generic encodings to write programs is a known technique in generic programming. Magalhães and Löh [22] investigate optimization of interaction between various generic views in their work on *generic generic programming*. It proposes to automatically derive generic representations of datatypes for numerous generic views, defining them via conversions from one particular view. Specifically, these are translations between the analogues of the *Generic* class in different libraries. This obviates the need of writing multiple blocks of code for their instances for one datatype, thus optimizing generic programs.

regular [28] is a generic programming library, designed for generic rewriting, that represents regular datatypes, using a fixed point view. Converting to this

view requires detecting recursion points in the datatype structure, so it can adopt our approach. In this section, we define a conversion from the generics-sop view to the regular one.

Encoding regular The regular library represents datatypes as sums of products by means of the following single combinators (we omit the metadata combinator, for simplicity):

```
newtype K_R a        r = K_R a
newtype I_R          r = I_R r
data U_R             r = U_R
data (f :+:_R g)     r = L_R (f r) | R_R (g r)
data (f :×:_R g)     r = f r :×:_R g r

infixr 6 :+:_R
infixr 7 :×:_R
```

It employs the type $:+:_R$ of binary sums and the type $:\times:_R$ of binary products. By nesting the sums, it chains constructors, and by nesting the products, it chains fields within one constructor. The types K_R, I_R, and U_R represent fields of a constant type, recursive positions, and constructors without fields (called "units"), respectively.

Using these representation types, regular encodes the underlying recursive structure of datatypes as polynomial functors. For example, recall the type *AExpr*:

```
data AExpr = EConst Int | EAdd AExpr AExpr | EMul AExpr AExpr
```

The corresponding polynomial functor is

```
type PF_AExpr = K_R Int :+:_R I_R :×:_R I_R :+:_R I_R :×:_R I_R
```

Datatypes can then be represented as their polynomial functors, closed under a fixed point operator. Although, for practical tasks, it is sufficient to convert only one layer of the datatype structure. regular provides a *Regular* class of representable datatypes, with an associated type *PF* for a polynomial functor and shallow conversion functions $from_R$ and to_R:

```
class Regular a where
  type PF a :: * → * → *
  from_R :: a → PF a a
  to_R   :: PF a a → a
```

The functions perform conversion between the datatype a and its generic representation $PF\ a\ a$ that stores elements of type a in the recursive positions.

Now, we turn to defining translation from generics-sop to regular. All the conversion work is divided into two steps. First, we perform type-level translation of the SOP codes into the regular representation types. Then, we convert representations at the level of terms.

Type-level conversion The type-level conversion is done by type families:

```
type family RegS t (xss :: [[*]]) :: * → * where
  RegS t '[a]      = RegP t a
  RegS t (a ': b) = RegP t a :+:R RegS t b

type family RegP t (xs :: [*]) :: * → * where
  RegP t '[a]      = RegEl t a
  RegP t (a ': b) = RegEl t a :×:R RegP t b
  RegP _ '[]       = UR

type family RegEl t a :: * → * where
  RegEl t t = IR
  RegEl _ a = KR a
```

This machinery systematically turns n-ary sums of products to corresponding nested binary sums of binary products. Empty products are being turned into units U_R, and recursive positions and constants are being wrapped into the combinators I_R and K_R, respectively.

The regular representation type for the datatype a is then defined as

```
type Reg a = RegS a (Code a) a
```

Term-level conversion The value conversion of representations is handled by the type classes *ConvS*, *ConvP*, and *ConvEl*:

```
class ConvS (s :: Bool) a (xss :: [[*]]) where
  toRegS :: NS (NP I) xss → RegS a xss a

class ConvP (s :: Bool) a (xs :: [*]) where
  toRegP :: NP I xs → RegP a xs a

class ConvEl (eq :: Bool) a x where
  toRegEl :: x → RegEl a x a
```

The logical type parameter s in the first two declarations is used to distinguish the cases, when the lists xss and xs consist of a single element.

We only show the *ConvEl* instances that do work on managing recursion:

```
instance ConvEl 'True t t where
  toRegEl x = IR x
instance RegEl t a ~ KR a ⇒ ConvEl 'False t a where
  toRegEl x = KR x
```

The second case requires the context, witnessing that *RegEl t a* results in $K_R\,a$, because *RegEl t a* is less specific than *RegEl t t*, hence may be overlapped.

The conversion function from SOP to regular is defined as

```
toReg :: ConvS a (Code a) ⇒ SOP I (Code a) → Reg a
toReg = toRegS . unSOP
```

Finally, we use this function to give an instance of *Regular* for all instances of SOP's *Generic*:

```
instance (Generic a, ConwS a (Code a)) ⇒ Regular a where
  type PF a = RegS a (Code a)
  from_R = toReg . from
```

6 Discussion

In this section, we give a discussion of some concerns relating to the scope and user-friendliness of the introduced no-overlap technique for handling recursion.

Polymorphic recursion The approach, described in this paper, 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* [2].

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 $* \to (* \to (* \to \ldots))$, 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' = gshow
```

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 *GShow* constraint for $[a]$, $[[a]]$, and so on ad infinitum. Similarly, the approach fails for functions that are nonrecursive itself, but meant to be used recursively, such as *gcompos* from Section 3.3, as it induces an infinite constraint at a call site of the generic function.

Still, one can think of problems where a generic operation uses the recursive structure of data, but has no recursive calls—and, therefore, can tolerate nested datatypes. An example of this kind of problems is "generic generic programming" (see Section 5). In our setting, it is possible to define a translation from generics-sop to another generic view that explicitly encodes recursive positions and supports polymorphic recursion—for example, generic-deriving [20]. This needs only to detect recursion points, because this library provides shallow conversion functions.

Usability and error-reporting The shown no-overlap technique may involve a lot of multiple constraints on generic functions. We tend to abbreviate them with type aliases. Although convenient when reading and writing code, one may encounter a leak in abstraction by making a mistake in client code. In this case, GHC sometimes produces an embarrassingly long error message where multiple underlying constraints are displayed. We currently investigate the *TypeError* recent mechanism of GHC which allows for user-defined error messages, specifically targeted for type level-heavy computations.

7 Related work

There are many works that contribute to the datatype-generic programming. Rodriguez et al. [26] and Magalhães and Löh [21] 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 [28, 30, 13, 18]. And there are a number of approaches that do not make use of fixed points [4, 20], but explicitly encode recursion in the datatype representation. The SOP view [7], 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. [15] 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 [25] 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 [9] were recently introduced in Haskell to solve the overlap problem.

Several works show how to define the Zipper [12] generically for regular [11, 23] and mutually recursive [30] 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.

8 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 thankful to Andres Löh for his helpful recommendations and comments on the paper. We address some insightful questions and suggestions from participants of the TFP symposium, to whom we are deeply grateful. We thank Julia Belyakova, who helped to proof-read certain parts of the paper, and 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. Bird, R., Meertens, L.: Nested datatypes. In: Jeuring, J. (ed.) Mathematics of Program Construction. pp. 52–67. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
3. Bringert, B., Ranta, A.: A pattern for almost compositional functions. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming. pp. 216–226. ICFP '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1159803.1159834
4. 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
5. 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
6. 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
7. 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
8. De Vries, E., Löh, A.: generics-sop: Generic programming using true sums of products (2018), http://hackage.haskell.org/package/generics-sop
9. 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
10. Safe haskell & overlapping instances—ghc (2015), https://ghc.haskell.org/trac/ghc/wiki/SafeHaskell/NewOverlappingInstances
11. 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)
12. Huet, G.: The zipper. Journal of Functional Programming **7**(5), 549–554 (1997)
13. 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
14. Kiselyov, O.: Type equalities, disequalities and obsoleting of overlapping instances (2012), http://okmij.org/ftp/Haskell/typeEQ.html
15. 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
16. Löh, A.: Exploring Generic Haskell. Ph.D. thesis, Utrecht University (2004)
17. 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)

18. 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
19. Magalhães, J.P.: Less Is More: Generic Programming Theory and Practice. Ph.D. thesis, Utrecht University (2012)
20. 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
21. 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
22. Magalhães, J.P., Löh, A.: Generic generic programming. In: Flatt, M., Guo, H.F. (eds.) Practical Aspects of Declarative Languages. pp. 216–231. Springer International Publishing, Cham (2014)
23. McBride, C.: The derivative of a regular type is its type of one-hole contexts (2001), http://strictlypositive.org/diff.pdf, unpublished manuscript
24. 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)
25. 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
26. 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
27. basic-sop: Basic examples and functions for generics-sop (2017), https://hackage.haskell.org/package/basic-sop
28. 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
29. 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
30. 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
31. 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