Full Title

Anonymous Author(s)

Abstract

Many of the commonly used today libraries for datatype-generic programming offer a fixed-point view on datatypes to express the recursive structure. Some approachs 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 the datatype representation, but raise issues when it needs to deal with recursion. A known unwelcome solution is use of overlapping instances. We present a technique that uses closed type families to allow us to eliminate overlapping for handling recursion. Moreover, we show that our idiom allows for families of mutually recursive datatypes.

The generics-sop library is an approach to representing data using *n*-ary sums and products that both are list-like structures, which is different from the classical view where datatypes are expressed by combinations of separate binary sums and products, and it does not encode recursive knots explicitly as with the fixed-point view. We choose this approach as a case study to demonstrate our solution.

Keywords Datatype-generic programming, Sums of products, Recursion, Overlapping instances, Closed type families, Zippers, Mutually recursive datatypes, Haskell

1 Introduction

A classical way to generically view data is to represent constructors as chains of nested binary sums, and constructor arguments as chains of nested binary products [Cheney and Hinze 2002; Löh 2004; Magalhães et al. 2010; Van Noort et al. 2008; Yakushev et al. 2009]. De Vries and Löh [2014] 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 that stands for a "sum of products"—this is implemented in the generics-sop library and is based on several relatively recent extensions to the Haskell type system, such as *data kinds* and *kind polymorphism* [Yorgey et al. 2012], *constraint kinds*, and *GADTs* [Schrijvers et al. 2009]. Using these Haskell features, the library provides the generic view and equips it with a rich collection of high-level traversal 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.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

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* [Magalhães et al. 2010] and *folds* [Meijer et al. 1991; Yakushev et al. 2009], more advanced one is a *zipper* [Adams 2010; Hinze et al. 2004; Huet 1997; Yakushev et al. 2009]. For handling recursion, several generic programming approachs use different forms of a fixed-point operator and express datatypes in the form of polynomial functors closed under fixed points [Jansson and Jeuring 1997; Löh and Magalhães 2011; Van Noort et al. 2008; Yakushev et al. 2009]. The SOP view allows to easily define functions that do not need to specially process recursive occurences. But for those that do need it, it normally does not allow.

One possible solution is to construct a more specific universe modifying the SOP core by explicitly encoding recursive positions using a fixed point. The fixed-point approach can be combined with SOP [De Vries and Löh 2014], but doing so makes that more complicated. Besides, such a decision may lead to need for additional conversions between the views. Our method makes it possible to define functions that need access to recursive occurences within the SOP view without modifying it anyhow. However, the definitions are still quite verbose.

Another known solution is use of overlapping instances. This is unwelcome, as involving overlapping instances complicates reasoning about the program semantics, makes it unstable because code with overlapping instances can indirectly change the behavior, if one adds more specific instances into scope by importing a new module. Furthermore, GHC does not reject ambiguous overlaps where neither of instance declarations is more specific than the other and does not even warn about ambiguity before attempting to use it at a type class function call site. This is also crucial in the security setting when code is compiled as safe, as GHC does not capture unsafe overlaps by detecting the presence of overlapping declarations and marking the module unsafe [GHC 2015]. Hence, the undesirable effect of overlapping instances becomes a problem in the setting of Safe Haskell. Kiselyov et al. [2004] show how to localize overlapping for access operations in a systematic way. Using *closed type families* [Eisenberg et al. 2014], we introduce an idiom that allows us to completely eliminate overlapping for handling recursion.

We believe that our idea is suitable for a number of different sum-of-products approachs that do not exploit a fixed point—so there the described problem appears. We choose the generics-sop approach as a case study because we think it is a widely applicable library based on the interesting ideas

for generic programming, and it relies on GHC's recent extensions.

To demonstrate our approach, we give several examples. Another interesting result of our work, which contributes to generics-sop, is an interface we provide for generic functions: the type class instances with them do not need to be manually declared, as we define the instances on the generic representation of datatypes.

Contributions This paper makes the following contributions:

- We introduce an idiom that allows to exclude use of overlapping instances for handling recursion in a number of sum-of-products approachs to datatype-generic programming that do not express recursive positions through a fixed point.
- We present our idea by giving several examples of generic functions, particularly the generic zipper for mutually recursive datatypes, using the generics-sop view.
- We provide a convenient interface for generic functions: there is no need to manually declare type class instances for using them.

Organization The paper is organized as follows. In Section 2, we introduce the SOP view concepts and demonstrate the problem by a short example. In Section 3, we show our solution using closed type families and accompany it with one more example of a generic function adapting the idea. In Section 4, we present an advanced example of the generic zipper for mutually recursive datatypes. In Section 5, we review related work, and we conclude in Section 6.

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. After this introduction, 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 borrow from SOP and use throughout the paper. The main idea of the SOP view is that a datatype is isomorphic to the sum of products of its code whose kind is a list of lists of types—it makes use of the DataKinds extension that enables datatype definitions to be promoted to kinds. SOP expresses this using a `Code` 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 provides a sequence of constructor arguments

```
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)
```

Figure 1. Datatypes for n-ary sums and products.

depending on the constructor chosen; the outer list, an n-ary sum, corresponds to a choice between different constructors.

For example, the code for the datatype of binary trees

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

is expected to be as follows:

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

A type of promoted lists has no inhabitants, so the universe provides datatypes to operate on n-ary sums and products as on terms. 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 [Hinze et al. 2004] by a promoted datatype of lists. The universe defines these with built-in functor application, where the functor is some type constructor. Thus, each component of `NS` and `NP` is given by mapping a corresponding type in the type-level list. The definitions of `NS` and `NP` both are kind polymorphic. The index list allows to contain types of arbitrary kind `k`, as applying the functor `f` turns `k` to `*`.

As basic instantiation types of `f`, SOP defines an identity functor `I` that is type-level equivalent for `id`, and a constant functor `K`, for `const`:

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

Whereas instantiated with `I`, the product is a direct heterogeneous list of types, with `K a`, it is essentially homogeneous. An example value of a product for `I` looks thus:

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

A clear idea underlies the sum definition. The constructor `S` of a sum skips the first element of an n-element index list giving an index in a list that has $n + 1$ elements, 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 datatype's sum of products matches the index

of its particular constructor in the index list and gives the product representing arguments of that constructor.

With the representation datatypes NS and NP, SOP defines a Generic class with conversion functions from and to to witness the isomorphism:

```
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 SOP f is a newtype for NS (NP f), indexed by a type-level list of lists, and the structural representation Rep of a datatype is a type synonym for a SOP I of its code. The functions from and to perform a shallow conversion of the datatype topmost layer—it does not recursively translate the constructor arguments.

The All SListI constraint in the Generic class definition is supposed to be satisfied, and an understanding of this detail is not necessary to understand the paper, but we will next use an understanding of the constraint All specifying that a particular constraint holds for each element in a list of types. Using constraints as types of a special kind Constraint, hence they can appear as type parameters, is allowed by the ConstraintKinds language extension.

The Generic class definition in the generics-sop library uses a *generic generic programming* technique [Magalhães and Löh 2014] to create the instance for a particular datatype automatically using the internal GHC.Generics representation. Alternatively, generics-sop allows to produce the instances using Template Haskell [Sheard and Jones 2002].

2.2 Problem with handling recursion

Let us illustrate the problem by giving a short example. The QuickCheck [Claessen and Hughes 2011], a library for automatic testing of Haskell code, using the GHC.Generics view, 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. To implement such a function using the SOP view, we have to say something like this:

```
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 definition of subtermsNP shows the clear idea: it traverses the product adding the element to the result list if its type is the same as a given term's, otherwise skipping that. We use GHC's TypeAppications extension, which allows *visible type applications*, here to fix that type. Note that using this with the type variables appearing in the function signature requires explicit universal quantification.

Now, we need a way to check type equality, and in the case of equal types, to witness that the element is of the appropriate type that admits adding that to the result list of subterms. There is a solution: we can follow QuickCheck's example and implement this using overlapping instances as it does with GHC.Generics.

The definition of subtermsNP we have shown serves as a clear illustration of the idea and as a template for a more satisfactory solution that we present in Section 3. But right now, the simplest way to implement this is to rewrite subtermsNP as follows:

```
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 _ = []
```

This uses a type class with the overloaded function. It needs also to declare that all the products in the datatype code are instances of Subterms a by adding a constraint to the subterms and subtermsNS signatures:

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

Though this will work, and there are a number of the Hackage packages that do it in such a manner, we want to write programs free of overlapping because of complexity it introduces into code as we have discussed in Section 1. Fortunately, we can do without overlapping, and we will demonstrate the solution in the next section.

3 Handling recursion with closed type families

In Section 2, we have shown a solution to the problem of handling recursion, which makes use of overlapping instances. We are now going to improve the solution and remove overlapping.

Closed type families are the Haskell language extension introduced by Eisenberg et al. [2014]. Equations for the *closed type family* are disallowed to de defined outside its declaration. Under this 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 always strictly defined, overlapping equations can be used to define sound type-level functions like this type equality.

We now come back to our running example from Section 2. With type equality, we can witness the coersion 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 convenience, we provide a synonym for this. As we need the constraint can be partially applied, we define this as a class with a superclass constraint, rather than as a type synonym:

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

The following implementation of `subtermsNP` using this proof resembles our first definition given in Section 2:

```
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 = []
```

We also make use of ScopedTypeVariables 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. A proper `ProofEq` constraint must be added to the `subterms` and `subtermsNS` declarations as well.

All generic functions accessing recursive knots in the underlying datatype structure can be defined in this way. We give another detailed example of one of such functions to show how to adapt our idiom to different scenarios.

3.1 Generic show

The function `show` is one of common examples 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. We follow the implementation of the generic function `gshow` in basic-sop [2017] for the most part, but improve it in respect of handling recursion. The original function `gshow` from basic-sop is not a substitute for the version of `show` that can be generated for a particular datatype through **`deriving`** `Show`, because it does not consider recursion to place parentheses at points of recursive calls—but we remove this lack of expressiveness.

The following exploits the idea of *pattern matching*. As before, we consider two cases. In the first case when the position we are matching is not recursive, we only require it to be the `Show` instance, and invoke its `show`, whereas in the case of the recursion point, we surround it with parentheses and recursively apply our generic function we define next. We again use the defined type equality to model a form of pattern matching on types:

```
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 ++ ")"

class CaseShow (Equal a b) a b ⇒
      CaseRecShow a b
instance CaseShow (Equal a b) a b ⇒
         CaseRecShow a b

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

As before, we provide the synonyms in order to reduce the constraints, as well as one for the matching function.

The function `gshow` involves meta-information to show a datatype constructor's and its record fields' names. It uses the functions that generics-sop provides for handling metadata which it defines separate from the universe, as follows:

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

gshow :: ∀a. GShow a ⇒ a → String
gshow t = gshow' @a (constructorInfo
                   $ datatypeInfo
                   $ Proxy @a) (from t)
```

The generics-sop library has features for deriving this meta-information automatically. We again define the synonym for the set of the constraints to appear in the function signature. The constraint `All2` is an analogue of `All` for a list of lists of types. The functions from generics-sop use a proxy to fix a type value, where we use TypeApplications—a later language extension than those the library relies on.

The auxiliary function `gshow'` that works on the metadata encoding and generic representation uses generics-sop's traversal combinators for collapsing and mapping:

```
gshow' :: ∀a xss.
          (All2 (CaseRecShow a) xss, SListI xss)
       ⇒ NP ConstructorInfo xss
       → SOP I xss → String
gshow' cs (SOP sop)
    = hcollapse
      $ hczipWith allp (goConstructor @a) cs sop
  where
    allp = Proxy @(All (CaseRecShow a))
```

When a sum structure actually is homogeneous (i. e., has type `NS (K a) xs`), it can be collapsed to a single component of type `a`. The respective instantiation of the function `hcollapse` that generalizes collapsing homogeneous structures is

```
hcollapse :: NS (K a) xs → a
```

The `hczipWith` function is generalized `zipWith` that operates with a constrained function on heterogeneous structures, where the proxy fixes the constraint:

```
hczipWith :: All c xs
          ⇒ proxy c
          → (∀a. c a ⇒ f a → g a → h a)
          → NP f xs → NS g xs → NS h xs
```

Finally, we show functions, which process constructors and record fields and make use of `caseShow` (Figure 2). The function `goConstructor` repeats in general the shape of `gshow'`, where `hcmap` generalizes `map` for `NP`, `hczipWith` on two products returns `NP`, and on two cases, `hcollapse` collapses the result `NP` to a list of strings.

The function `gshow` now can be used to generically show data, for example, a value of type `Tree Bool`, where `Tree a` (Section 2) is now assumed to be an instance of `Generic`, and of `HasDatatypeInfo`:

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

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

And here is a benefit of our implementation: it can be used directly, without any additional instance declarations, whereas basic-sop [2017] offers the following usage pattern for `gshow` and some datatype `T`:

```
goConstructor :: ∀a xs. All (CaseRecShow a) xs
              ⇒ ConstructorInfo xs → NP I xs
              → K String xs
goConstructor (Constructor n) args
    = K $ unwords (n : args')
  where
    args' :: [String]
    args' = hcollapse
            $ hcmap (p @a)
                    (K . caseShow @a . unI)
                    args
goConstructor (Record n ns) args
    = K $ n ++ " {" ++ intercalate ", " args'
          ++  "}"
  where
    args' :: [String]
    args' = hcollapse
            $ hczipWith (p @a)
                        (goField @a) ns args
goConstructor (Infix n _ _)
              (I arg1 :* I arg2 :* Nil)
    = K $    caseShow @a arg1
         ++ " " ++ n ++ " "
         ++ caseShow @a arg2

p :: Proxy (CaseRecShow a)
p = Proxy
```

```
goField :: ∀a x. (CaseRecShow a) x
        ⇒ FieldInfo x → I x → K String x
goField (FieldInfo field) (I y)
    = K $ field ++ " = " ++ caseShow @a y
```

Figure 2. Processing constructors and record fields.

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

This is a consequence of that in basic-sop, the `gshow` function does not treat recursive positions separately, and therefore requires the `Show` constraint for all knots in the datatype structure.

4 The generic Zipper

5 Related work

6 Conclusion

References

Michael D. Adams. 2010. Scrap Your Zippers: A Generic Zipper for Heterogeneous Types. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP '10)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1863495.1863499

basic-sop 2017. basic-sop: Basic examples and functions for generics-sop. (2017). https://hackage.haskell.org/package/basic-sop

James Cheney and Ralf Hinze. 2002. A Lightweight Implementation of Generics and Dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 90–104. https://doi.org/10.1145/581690.581698

Koen Claessen and John Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 46, 4 (May 2011), 53–64. https://doi.org/10.1145/1988042.1988046

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 671–683. https://doi.org/10.1145/2535838.2535856

GHC 2015. Safe Haskell & Overlapping Instances—GHC. (2015). https://ghc.haskell.org/trac/ghc/wiki/SafeHaskell/NewOverlappingInstances

Ralf Hinze, Johan Jeuring, and Andres Löh. 2004. Type-indexed data types. *Science of Computer Programming* 51, 1 (2004), 117–151. https://doi.org/10.1016/j.scico.2003.07.001 Mathematics of Program Construction (MPC 2002).

Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554.

Patrik Jansson and Johan Jeuring. 1997. PolyP—a Polytypic Programming Language Extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 470–482. https://doi.org/10.1145/263699.263763

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/1017472.1017488

Andres Löh. 2004. *Exploring Generic Haskell*. Ph.D. Dissertation. Utrecht University.

Andres Löh. 2015. Applying Type-Level and Generic Programming in Haskell. Summer School on Generic and Effectful Programming. (July 2015). https://github.com/kosmikus/SSGEP/blob/master/LectureNotes.pdf Lecture notes.

Andres Löh and José Pedro Magalhães. 2011. Generic Programming with Indexed Functors. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming (WGP '11)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2036918.2036920

José Pedro Magalhães. 2012. *Less Is More: Generic Programming Theory and Practice*. Ph.D. Dissertation. Utrecht University.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A Generic Deriving Mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1863523.1863529

José Pedro Magalhães and Andres Löh. 2012. 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*. 50–67. https://doi.org/10.4204/EPTCS.76.6

José Pedro Magalhães and Andres Löh. 2014. Generic Generic Programming. In *Practical Aspects of Declarative Languages*, Matthew Flatt and Hai-Feng Guo (Eds.). Springer International Publishing, Cham, 216–231.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, John Hughes (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–144.

Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. 2008. A Lightweight Approach to Datatype-generic Rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP '08)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1411318.1411321

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. 2008. Comparing Libraries for Generic Programming in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/1411286.1411301

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and Decidable Type Inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 341–352. https://doi.org/10.1145/1596550.1596599

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/2633628.2633634

Edsko de Vries and Andres Löh. 2018. generics-sop: Generic Programming using True Sums of Products. (2018). http://hackage.haskell.org/package/generics-sop

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. 2009. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1596550.1596585

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795