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 need access to the recursive positions in the datatype structure, 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, by giving an advanced example, 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 represented as combinations of binary sums and products, and it does not encode recursive knots explicitly as with the fixed-point view. We have chosen 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 by nested binary sum types while constructor arguments are represented by nested binary product types [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's features, the library provides the generic view as well as a rich interface of high-level traversal combinators, such as for constructing sums and products, collapsing to homogeneous structures, and application, which together encourage generic function definition in more concise and high-level style compared to the classical binary sum-of-products views.

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

Many generic functions require access to the recursive knots in 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 approaches use different forms of fixed-point operator and represent the underlying polynomial functor [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 treat recursive occurences. But for those that do need it, it normally does not allow.

One possible solution is to construct a more more specific universe modifying the SOP core by explicitly encoding recursive positions using a fixed point. The fixed-point approach is not fundamentally incompatible 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 alter the behavior, when resolving instance selection, if one adds more specific instances into scope by importing a new module. Furthermore, GHC does not reject ambiguous overlaps where neither of instances is more specific than the other and does not even report an error 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 declaration of overlapping instances in a module and marking it 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 have chosen 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 the recent GHC's extensions.

To demonstrate our approach, we give several examples. Another interesting result of our work 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 describe the problem. In Section 3, we demonstrate our solution using closed type families and accompany it with several examples of generic functions. 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 depending on the constructor chosen; the outer list, an n-ary sum, corresponds to a choice between different constructors.

```
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.

For example, the code for the datatype of binary trees

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

is supposed 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. Each element of NS and NP is given by applying a functor to a corresponding type in the type-level index list. The definitions of NS and NP both are kind polymorphic. The index list allows to be a list of arbitrary types of kind k, as the functor f maps k to *.

As a basic instantiation of f, SOP defines type-level equivalent of id, an identity functor I:

```
newtype I (a :: *) = I {unI :: a}
```

When instantiated with I, the product becomes a direct heterogeneous list of types.

An example value of a product looks thus:

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

The sum constructors are similar to Peano numbers, so the choice from a datatype's sum of products matches the index of a particular constructor in the index list and gives the constructor being chosen. The constructor S skips the first element of an n-element index list producing an index into a list that has $n + 1$ elements, while Z contains 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]
```

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

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

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

The SOP is a newtype for an NS of an NP, 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 All SlistI constraint in the Generic class definition is supposed to be always satisfied and an understanding of this detail is not necessary to understand the paper, but we will next use an understanding of the type family All specifying that a constraint holds for all elements of a list of types. Using constraints as types of a special kind Constraint, hence they can appear in type families, 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 derive the instance for a particular datatype automatically using the internal GHC.Generics representation. Alternatively, the generics-sop library allows to generate the Generic 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 program properties, using the GHC.Generics view, defines a helper function subterms that obtains all the immediate subterms of a term that are of the same type as the term itself, 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 (Proxy :: Proxy a)
                        (unSOP $ from t)

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

subtermsNP :: Proxy a → NP I xs → [a]
subtermsNP p (I x :* xs)
  | typeEq p x = witness p x : subtermsNP p xs
  | otherwise  = subtermsNP p xs
subtermsNP _ Nil = []
```

3 Handling recursion with closed type families

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

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