Full Title

ANONYMOUS AUTHOR(S)

Many of the commonly used today libraries for datatype-generic programming offer fixed-point view on datatypes to express the recursive structure. 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. The representation allows for generic functions that do not need access to the recursive positions in the datatype structure, but it raises issues when it needs to deal with recursion. We present a technique that uses closed type families to allow us to handle recursive occurences. Moreover, we show, by giving an advanced example, that our approach allows the generics-sop's view for families of mutually recursive datatypes.

Additional Key Words and Phrases: Datatype-generic programming, Sums of products, Recursion, Closed type families, Zippers, Mutually recursive datatypes, Haskell

1 INTRODUCTION

The 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; 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* [Yorgey et al. 2012], *constraint kinds*, *kind polymorphism*, and *GADTs*. 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.

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* [Hinze et al. 2004; Huet 1997; Yakushev et al. 2009]. For handling recursion, several generic programming approachs 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, as we show in our paper, there is a problem, so it normally does not allow. This paper introduces a method that makes it possible to define such functions within the SOP view, though the definitions are still quite complicated.

1.1 The SOP view

We first explain the terminology we borrow from the SOP and use throughout the paper. We show the core definitions of the SOP universe in Fig. 1. The main idea is that a datatype is isomorphic to the sum-of-products of its code whose kind is a promoted list of lists of types `[[*]]` (here, the use of `DataKinds` extension enables datatype definitions to be promoted to kinds). This is specified using the `Code` type family. An n-ary sum and an n-ary product are therefore modelled as type-level

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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

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

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

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

Fig. 1. The SOP view on data.

heterogeneous lists: the inner list is an *n*-ary product that provides, depending on the constructor chosen, an appropriate sequence of constructor arguments; the outer list, an *n*-ary sum, corresponds to a choice between different constructors.

A type of promoted lists has no inhabitants, so the universe provides the definitions to operate on *n*-ary sums (NS) and on *n*-ary products (NP) as on terms. The datatypes NS and NP 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, that is, each element of the term list is given by applying a particular type constructor ("interpretation function") 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 interpretation function f maps k to $*$.

A common instantiation of f, which is enough for our goals, is the identity functor I

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

that is type-level equivalent of id function, for which the product becomes a heterogeneous list of types.

An example value of a product looks thus:

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

The sum constructors resemble Peano numbers, so the choice from 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 Zero contains the payload of type f x. The below is an example value of a sum.

```
S (S (Z (I 3))) :: NS I '[Char, Bool, Int, Bool]
```

1.2 Problem with handling recursion

Let us illustrate the problem by giving a short example. The QuickCheck, a library for automatic testing of Haskell program properties, using GHC.Generics, defines the 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 the 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 = []
```

REFERENCES

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

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

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, 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

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

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

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