True Sums of Products

Edsko de Vries
Well-Typed LLP
edsko@well-typed.com

Andres Löh
Well-Typed LLP
andres@well-typed.com

Abstract

We introduce the *sum-of-products* (SOP) view for datatype-generic programming (in Haskell). While many of the libraries that are commonly in use today represent datatypes as arbitrary combinations of binary sums and products, SOP reflects the structure of datatypes more faithfully: each datatype is a *single* n-ary sum, where each component of the sum is a *single* n-ary product. This representation turns out to be expressible accurately in GHC with today's extensions. The resulting list-like structure of datatypes allows for the definition of powerful high-level traversal combinators, which in turn encourage the definition of generic functions in a compositional and concise style. A major plus of the SOP view is that it allows to separate function-specific metadata from the main structural representation and recombining this information later.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Data types and structures

Keywords datatype-generic programming; sums of products; universes; generic views; JSON; lenses; metadata

1. Introduction

The goal of datatype-generic programming is to make use of a common underlying structure of datatypes in order to define programs in such a way that they automatically work for a large class of datatypes. Using datatype-generic programs makes it easier to evolve and refactor programs, because when datatypes change, datatype-generic functions adapt. Typical examples of datatype-generic functions include structural equality, all sorts of conversion functions such as serialisation and deserialisation, and all kinds of traversals such as maps and folds.

The exact way in which a common structure of datatypes is established has a significant effect on which generic functions can be expressed easily or at all, the programming style they encourage, how easy they are to understand or adapt, and how efficient the generated code is.

Not every problem domain has exactly the same requirements. The combination of the general appeal of datatype-generic programming and the diversity of goals and scenarios in which it is employed make it unsurprising that many different approaches exist, even within a single programming language such as Haskell. These approaches differ in a multitude of different ways, such as which and how many functions are predefined, which features of the Haskell language are being used, how portable they are, how much emphasis on efficiency they place, and so on. Their main distinguishing feature, however, is how they view the structure of datatypes.

Not all of these *views* are completely different from each other. Many libraries are based on variations of what is typically called a "sum of products" view. For example, the generic representation of a binary tree type such as

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

using the GHC.Generics library is essentially isomorphic to

```
Either () (Tree a, (a, Tree a))
```

where Either is a binary sum type, and pairs (,) are a binary product type. The actual representation is more complicated, because it involves metadata such as type and constructor names etc., but let's focus on the pure structure for now.

Strictly speaking, the classification as a "sum of products" is not entirely accurate. Technically, we have not a sum of a product, but a sum of a product of products; and of course if we represent types using binary sums and products, then nothing in the types stops us from having products of sums, or sums of products of sums, etc. In practice the only nesting that is used is some stacking of sums of some stacking of products (usually to the right, sometimes balanced), but this is by implicit assumption only.

For some generic functions, such as equality, this does not matter. However, many generic functions care about the shape of the datatype. For instance, a function that constructs a default value might want to prefer a nullary constructor over other constructors (cf. Section 5.3). Similarly, when picking a random value for a datatype with multiple constructors we might want to vary the probability of picking a constructor depending on how many arguments it has (cf. Section 5.4). In general, defining operations that are not completely local, but need information about other constructors, or several constructor arguments at once, are surprisingly difficult to define using a binary view.

As an example, let us consider a function garity that counts the arities of all constructors of a datatype. Using GHC.Generics (Magalhães et al. 2010), a possible implementation is as follows:

```
class GArities (a :: * → *) where
  garities :: Proxy a → [Int]
instance GArities f ⇒ GArities (M1 i c f) where
  garities _ = garities (Proxy :: Proxy f)
instance GArities V1        where garities _ = []
instance GArities U1        where garities _ = [0]
instance GArities (K1 R a)  where garities _ = [1]
```


WGP '14, August 31 2014, Gothenburg, Sweden.

ACM 978-1-4503-3042-8/14/08…$15.00.
http://dx.doi.org/10.1145/2633628.2633634

```
instance (GArities f, GArities g) ⇒ GArities (f :*: g) where
  garities _ = let [x] = garities (Proxy :: Proxy f )
                   [y] = garities (Proxy :: Proxy g)
               in [x + y]
instance (GArities f, GArities g) ⇒ GArities (f :+: g) where
  garities _ = garities (Proxy :: Proxy f )
            ++ garities (Proxy :: Proxy g)
```

Let's ignore the details and just look at the high-level structure: The first instance skips over metadata; the next three instances are the case for empty datatypes, nullary constructors, and unary constructors, and the final two cases are for products and sums.

Let us focus on the case for binary products: at this point, we are traversing the structure of a particular constructor, so we can assume that if we call garities on the left or right component, exactly one arity will be returned. This function is safe only because we know how datatypes will in general be represented. If these implicit assumptions are violated for whatever reason, the code will break horribly, as the type system does not help us here.

While it is possible to reestablish type safety by using a different function $\mathsf{garities}_{\mathsf{Prod}}$ :: Proxy a → Int that just traverses the product structure and returns a single arity, this leads to new problems: calling $\mathsf{garities}_{\mathsf{Prod}}$ in the sum-case would not work, as some children of the binary sum are other sums, whereas others correspond to single constructors. Instead, we must observe that every individual constructor is guarded by a certain piece of metadata (another implicit assumption) and use the metadata-case to mitigate between $\mathsf{garities}_{\mathsf{Prod}}$ and garities. All this is possible, but it's tedious and error-prone. It's also in no way specific to GHC.Generics, which we've merely selected as a representative example.

We argue that binary sums and products that can arbitrarily be combined are simply not a good fit for many generic functions, because this kind of view does not represent the structure of the underlying Haskell datatypes very faithfully.

In this paper, we introduce a view (which we call SOP) which is based on a *single*, n-ary, sum, where each component of the sum is a *single*, n-ary, product. This sum of products is characterised by a list of list of types, and counting arities is a simple matter:

```
arities :: ∀a. SingI (Code a) ⇒ Proxy a → [Int]
arities _ = map length ∘ collapse_POP
        $ (pure_POP (K ()) :: POP (K ()) (Code a))
```

Intuitively, the $\mathsf{pure}_{\mathsf{POP}}$ creates a () value for every argument of every constructor in a nested, potentially heterogeneous list. As every element is in fact of the same type, we can $\mathsf{collapse}_{\mathsf{POP}}$ this structure into a normal homogeneous list of lists (of type [[()]]). Mapping with length then yields the desired result. The use of a list-like underlying structure facilitates a generic programming style based on high-level, powerful traversal combinators such as $\mathsf{pure}_{\mathsf{POP}}$ and $\mathsf{collapse}_{\mathsf{POP}}$ in this example, encouraging more concise and more high-level definitions that are based on composing several steps.

The SOP view also takes a very interesting approach to metadata. In most generic views, metadata is intertwined with the structural representation, which means that *every* generic function has to deal with it in some way—even if it is just ignored, as in the M1 case for garities shown above. Furthermore, metadata sometimes leads to additional implicit assumptions about the shape of the data.

In the SOP view, metadata is completely independent from the data representation. This means that functions which do not need it don't have to deal with it; conversely, it also means that we can easily define application-specific metadata, i.e., type-directed additional information that "configures" how a particular generic function should behave.

Contributions Our paper makes the following contributions:

- We present the SOP view for generic programming, a view that is more typed and therefore more faithfully represents the structure of Haskell datatype compared to binary sum-of-product approaches.
- We argue that a more precise, list-like structure, facilitates taking a high-level view on generic function definition by admitting a rich interface of powerful and reusable traversal combinators, while still admitting the flexibility of performing explicit induction on the type structure when needed.
- The separation of metadata from the data representation not only unclutters all the definitions that don't need any metadata—it also facilitates to define generic functions that use application-specific metadata in an entirely type-safe way.

Implementation The SOP view is made possible and feasible by several, partially quite recent, extensions to the GHC type system, such as data kinds, kind polymorphism, and GADTs.

The code in this paper itself is executable—the paper sources have been type-checked. There is also a separate implementation in the `generics-sop` library (available on Hackage). The library shares all the core ideas with this paper, but deviates in some details which we've adapted in this paper for reasons of space or presentation.

Structure of the paper The rest of the paper is structured as follows: In Section 2, we discuss some basic concepts and some GHC extensions that we make use of throughout the paper. In Section 3, we introduce the SOP view. Based on this view, we then develop a library of high-level traversal combinators (Section 4). With the help of these, we discuss several examples (Section 5). In Section 6, we focus on how metadata is treated using the SOP view and discuss further examples. In Section 7, we compare our approach to related work, before we discuss future possibilities and conclude in Section 8.

2. Preliminaries

In this section we briefly describe some simple datatypes that we will use throughout the paper, as well as some of the more recent GHC extensions that we rely on.

Every Haskell programmer is intimately familiar with

```
id    :: a → a
const :: a → b → a
```

In this paper we will be doing a lot of type-level programming, and will hence need the type level equivalents of these functions:

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

These are similar to their definitions in the standard libraries (called Identity and Constant, respectively), but the definition of K makes use of GHC's `PolyKinds` extension and is *kind polymorphic*; we will need this generality.

Polymorphic kinds become particularly important in the presence of *data kinds* (Yorgey et al. 2012), another recent GHC feature that we will use. With the `DataKinds` extension enabled, datatype definitions are automatically *promoted* to kind definitions. This includes several built-in datatypes such as lists: given a kind k, we can construct the a type level list of kind ′[k], with ′[] being the empty list of types and (k ′: ks) denoting type-level "cons". The tick marks are used to explicitly indicate that we mean the promoted type or term constructors in situations where the syntax is otherwise ambiguous. In this paper we will make extensive use of type-level lists (but no other promoted datatypes).

We furthermore rely on *constraint kinds*. Constraints, sometimes also known as *qualifiers*, are things such as $\mathsf{Show}\ \mathsf{a}$ in

$$\mathsf{show} :: \mathsf{Show}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{String}$$

Under the `ConstraintKinds` extension, Show is a type like another, albeit of a special kind:

$$\mathsf{Show} :: * \to \mathsf{Constraint}$$

This means that we can, and will, use constraints in type synonyms, type families, etc., and indeed we can, and will, *quantify* over constraints (i.e., use type variables of kind $* \to \mathsf{Constraint}$).

3. The SOP universe

In this section we introduce the basic idea and the implementation of the sum-of-products (SOP) approach to datatype-generic programming.

In dependently typed languages, a *universe* consists of a type of codes together with an interpretation function mapping codes to types (Martin-Löf 1984). The codes serve as an abstract representation of the types. Functions can be defined inductively over the codes and are then generic over all types described by the universe.

In Haskell, we cannot map values to types, but in the presence of `DataKinds`, we can lift everything up by one level: We use a *kind* (rather than a type) of codes. An interpretation function becomes a type-level function (in general, a type family, data family or a GADT) parameterised over an argument that has the kind of codes.

3.1 Codes and interpretations

The fundamental idea of the SOP universe is that the kind of *codes* is a (promoted) list of list of types, written $[[*]]$. The goal of the universe is to provide descriptions of Haskell datatypes (of kind $*$). We use a type family to map a datatype to its code:

```
type family Code (a :: *) :: [[*]]
```

A type of kind $[[*]]$ has no inhabitants—it is merely an abstract description that we can operate on. Consider the following simple datatype of arithmetic expressions with just integer constants and addition as an example:

```
data Expr = Num Int | Add {left :: Expr, right :: Expr}
```

The code for Expr looks as follows:

```
type instance Code Expr = '[ '[Int], '[Expr, Expr]]
```

The outer list has one element per constructor. For each constructor, the corresponding inner list contains the types of the constructor arguments.

Based on the SOP codes of kind $[[*]]$, we can now consider *interpretations*.

The most important interpretation is called SOP f ("sum of products"). It views the outer list as an *n*-ary sum, representing the choice between the constructors, and the inner lists as *n*-ary products, representing the constructor arguments. In addition, the functor f is applied to each of the elements. An important property is that SOP I (Code a) (where I is the identity functor) is isomorphic to the original datatype a. This isomorphism is captured by the type class Generic:

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

class SingI (Code a) ⇒ Generic (a :: *) where
  from :: a → Rep a
  to   :: Rep a → a
```

The functions from and to witness the isomorphism and are supposed to be mutual inverses. The SingI constraint will be explained in Section 4.1; for now, it suffices to say that it will always be satisfied. The purpose of the class Generic is as follows: if we manage to define a function that works for all (or a certain, well-specified subset of) codes, then we can turn that function into a datatype-generic function by making it work on all suitable instances of class Generic, applying the isomorphisms to translate as needed between the original datatype and its structural representation.

We will provide example instances of class Generic as soon as we have defined SOP in Section 3.3. As we shall see, they are straight-forward to define, and can easily be derived by Template Haskell (or by GHC itself, when appropriately extended).

There is another interesting interpretation of our codes, called POP f (for "product of products"). It views both the outer and the inner type-level lists as *n*-ary products, and once again applies f to all the elements. A POP f represents a table of information that is available at each component of the original datatype. A POP structure is excellent for storing information we need or want for all the components of all constructors. For example, the arities function from Section 1 made use of POP.

Let us now discuss how to define SOP and POP, before we go on to provide concrete examples for instances of the Generic class and then move on to build a library on top of our basic universe.

3.2 Sums and Products

In order to translate Haskell values into the SOP universe, we need support for *n*-ary sums and products. As a first attempt, the following datatype is isomorphic to arbitrary right-nested pairs (or heterogeneous lists):

```
data NP :: [*] → * where    -- preliminary
  Nil  :: NP '[]
  (:*) :: x → NP xs → NP (x ': xs)
```

For example, the nested pair

```
(True, ('x', 3)) :: (Bool, (Char, Int))
```

corresponds to

```
True :* 'x' :* 3 :* Nil :: NP '[Bool, Char, Int]
```

(assuming that (:*) is right-associative, just like ordinary (:) for lists).

We will however often need a product

$$\mathsf{f}\ \mathsf{T}_1 \times \mathsf{f}\ \mathsf{T}_2 \times \cdots \times \mathsf{f}\ \mathsf{T}_n$$

for some functor f; so we choose to define NP with functor application "built-in":

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

The nested pair from above can still be expressed by choosing the identity functor I (cf. Section 2) for f:

```
I True :* I 'x' :* I 3 :* Nil :: NP I '[Bool, Char, Int]
```

We can define *n*-ary sums in a similar manner:

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

The constructor names are reminiscent of Peano naturals. The constructor Z injects into the first component of a sum (with a least one component), S ∘ Z into the second component of a sum (with at least two components), and so on:

```
Z         :: f x → NS f (x ': xs)
S ∘ Z     :: f y → NS f (x ': y ': xs)
S ∘ S ∘ Z :: f z → NS f (x ': y ': z ': xs)
```

By nesting NS and NP applications, we can define both SOP and POP[1]:

```
type SOP (f :: k → ∗) (xss :: [[k]]) = NS (NP f) xss
type POP (f :: k → ∗) (xss :: [[k]]) = NP (NP f) xss
```

The definition of POP relies on the kind polymorphism of NP: the first argument of the inner application has kind k → ∗, the first argument of the outer application has kind [k] → ∗.

3.3 Examples

Having discussed the definition of SOP, we are finally equipped to give a concrete example instance of the Generic class.

Let's return to our example type of arithmetic expressions, for which we had already defined:

```
data Expr = Num Int | Add {left :: Expr, right :: Expr}
type instance Code Expr = '[ '[Int], '[Expr, Expr]]
```

The class instance looks for Expr as follows:

```
instance Generic Expr where
  from (Num n)  = Z     (I n :* Nil        )
  from (Add e f) = S (Z (I e :* I f :* Nil))
  to (Z     (I n :* Nil        ))  = Num n
  to (S (Z (I e :* I f :* Nil))) = Add e f
```

The recursive occurrences of Expr are not translated. This shallow transformation between a datatype and its structural representation is rather common for datatype-generic programming. It has the advantage that from and to are not recursive. We will come back to this point in Section 5.1.

As another example, let us look at a Generic instance for a parameterised datatype such as lists:

```
type instance Code [a] = '[ '[], '[a, [a]]]
instance Generic [a] where
  from []       = Z       Nil
  from (x : xs) = S (Z (I x :* I xs :* Nil))
  to (Z       Nil)                  = []
  to (S (Z (I x :* I xs :* Nil))) = x : xs
```

Again, we perform a shallow translation, not touching any of the components. This means that we can define Generic [a] without having to require Generic a.

As we can see from these two examples, Generic instances are rather straight-forward to define. Nevertheless, to make generic functions defined in the SOP view applicable to a datatype, a Generic instance has to be provided, and this is tedious.

In practice, we therefore prefer to let the compiler generate the instance for us. The `generics-sop` library contains Template Haskell (Sheard and Peyton Jones 2002) code to do so. There, we can e.g. write

```
deriveGeneric ''Expr
```

to have the above Generic instance of Expr derived for us. It's also possible to extend GHC (similar to the `DeriveGeneric` extension that already exists for GHC.Generics) to have built-in support for this class, or to use techniques as described by Magalhães and Löh (2014) to automatically translate between a GHC-internal representation and the SOP universe.

4. Traversal Combinators

In principle, we have all the ingredients now and could start defining generic functions, by induction over SOP values. However, the list-like structure we have available invites to build higher-level traversal operators that can be reused in the definition of several generic functions.

We argue that the very structured SOP view makes it easier to approach generic programming with higher-order functions: the product and sum structure are clearly separate from each other, which encourages to traverse them separately with dedicated combinators and compose the different phases.

In this section, we therefore try to reveal a bit more structure in the four types we mostly deal with: NP, NS, SOP and POP[2]. With the combinators we build in this section (the final list is shown in Figure 2), we can then implement actual application-specific generic functions in a very concise fashion. We will provide examples in Sections 5 and 6.

4.1 Constructing products

We will equip NP with what looks like an applicative interface; the analogy with Applicative is not perfect, but we will use the same nomenclature as an aid to the reader to make it easier to remember the names.

The first thing we need is an equivalent of pure for NP. We might try to define

```
pure_NP :: (∀a. f a) → NP f xs    -- preliminary
```

which creates an NP by repeating the given element as many times as there are elements in xs. However, there is a problem: in order to define this function, we need to perform induction over xs, and there is no way to perform pattern matching on a parametrically polymorphic type variable such as xs in Haskell. In a dependently typed language, xs would be a term-level list we could match on. If we want to simulate the situation in Haskell, we need to use either a type class, or a term-level value that reflects the structure of xs on the term level—so-called *singleton types* (Eisenberg and Weirich 2012).

In this paper, we will only need singletons for type-level lists as summarised in Figure 1. We define a data family Sing and a type class SingI that are mutually recursive. A Sing a is an explicit representation of type a on the term-level, in such a way that we can pattern-match on it. Since there is at most one such value for any given type, we use class SingI to infer that value automatically whenever possible.[3]

For type-level lists, we introduce SNil and SCons to distinguish between the two possible cases. For types of kind ∗, we introduce a "dummy" singleton SStar that does not actually allow us to distinguish different types of kind ∗ at run-time. So even in the presence of a SingI (a :: ∗) constraint, parametricity still holds. We can use singletons in the definition of $pure_{NP}$ as follows:

```
pure_NP :: ∀f xs. SingI xs ⇒ (∀a. f a) → NP f xs
pure_NP f = case sing :: Sing xs of
  SNil  → Nil
  SCons → f :* pure_NP f
```

Note that we use the `ScopedTypeVariables` extension here, so the call to sing produces a singleton of the same xs that is also the index of the resulting NP.

[1] In the `generics-sop` package, SOP and POP are defined via **newtype**, so that type class instances can be defined for them.

[2] In the `generics-sop` library, most of the combinators we define in this section are defined via type classes, so that names can be reused. As only the four instances for NP, NS, SOP, and POP are relevant, we do not introduce the classes here, but rather list the explicit types, and add indices to the function names to distinguish the different instances.

[3] Our singletons deviate slightly from Eisenberg and Weirich (2012), where Sing and SingI are not mutually recursive. However, using the original approach, every function defined using singletons needs in principle two versions, one using Sing and one using SingI. We can avoid that here. The change is not essential for our development.

We can provide a similar function for products of products:

```
pure_POP :: ∀f xss. SingI xss ⇒ (∀a. f a) → POP f xss
pure_POP f = case sing :: Sing xss of
  SNil   → Nil
  SCons  → pure_NP f :* pure_POP f
```

In practice, however, the types of $\mathrm{pure}_{\mathrm{NP}}$ and $\mathrm{pure}_{\mathrm{POP}}$ are often too restrictive. They require a value of type $\forall a. f\ a$, i.e., a value that is parametrically polymorphic in a. Often we want to use a value that relies on a type class constraint c, of type $\forall a. c\ a \Rightarrow f\ a$. We therefore define the following variant of $\mathrm{pure}_{\mathrm{NP}}$:

```
cpure_NP :: ∀c xs f. (All c xs, SingI xs)
         ⇒ Proxy c → (∀a. c a ⇒ f a) → NP f xs
cpure_NP p f = case sing :: Sing xs of
  SNil   → Nil
  SCons  → f :* cpure_NP p f
```

The constraint All c requires all the types in xs to satisfy c; we can define it as follows:

```
type family All (c :: k → Constraint) (xs :: [k]) :: Constraint

type instance All c '[]       = ()
type instance All c (x ': xs) = (c x, All c xs)
```

For example, the application All Eq '[Bool, Char, Int] expands to the constraint

```
(Eq Bool, (Eq Char, (Eq Int, ())))
```

which is equivalent to

```
(Eq Bool, Eq Char, Eq Int)
```

The Proxy c argument is necessary because GHC's type inferencer generally refuses to guess the value of constraint variables such as c unless they appear as an argument to a datatype such as Proxy:

```
data Proxy (a :: k) = Proxy
```

As datatypes are by definition injective, phantom arguments such as that of Proxy are a common technique to provide explicit instantiations of type variables to GHC. For example,

```
cpure_NP (Proxy :: Proxy Eq)
 :: (All Eq xs, SingI xs) ⇒ (∀a. Eq a ⇒ f a) → NP f xs
```

Following the ideas developed for $\mathrm{cpure}_{\mathrm{NP}}$, we try to generalise $\mathrm{pure}_{\mathrm{POP}}$ to $\mathrm{cpure}_{\mathrm{POP}}$ in a similar manner. Unfortunately, however, we cannot use

```
cpure_POP :: (All (All c) xss, SingI xss)    -- preliminary
          ⇒ Proxy c → (∀a. c a ⇒ f a) → POP f xss
```

since type family applications (just like type synonyms) must be fully saturated (Sulzmann et al. 2007, Section 3.6), and All c is not (it only partially applies All). Instead we define All^2

```
type family All² (c :: k → Constraint) (xs :: [[k]]) :: Constraint

type instance All² c '[]       = ()
type instance All² c (x ': xs) = (All c x, All² c xs)
```

and can then define

```
cpure_POP :: ∀c f xss. (All² c xss, SingI xss)
          ⇒ Proxy c → (∀a. c a ⇒ f a) → POP f xss
```

We will come back to the problem of a partial application of All in Section 4.5.

4.2 Application

Having defined the analogue of pure, we need to define the analogue of $\langle * \rangle$, which we will call ap. For NP this amounts to applying a product of functions to a product of arguments. The only complication here is that we need to define a lifted function space:

```
newtype (f -.-> g) a = Fn {apFn :: f a → g a}
```

For convenience, we define a few auxiliary constructors for lifted functions with several arguments:

```
fn_2 :: (f a → f' a → f'' a) → (f -.-> f' -.-> f'') a
fn_3 :: (f a → f' a → f'' a → f''' a) → (f -.-> f' -.-> f'' -.-> f''') a
```

Using (-.->), it is easy to define $\mathrm{ap}_{\mathrm{NP}}$:

```
ap_NP :: NP (f -.-> g) xs → NP f xs → NP g xs
ap_NP Nil          Nil       = Nil
ap_NP (Fn f :* fs) (x :* xs) = f x :* ap_NP fs xs
```

While we cannot apply a sum of functions to a sum of arguments, we can apply a *product* of functions to a sum of arguments:

```
ap_NS :: NP (f -.-> g) xs → NS f xs → NS g xs
ap_NS (Fn f :* _) (Z x)  = Z (f x)
ap_NS (_    :* fs) (S xs) = S (ap_NS fs xs)
```

We provide similar functions for product of products and sums of products:

```
ap_POP :: POP (f -.-> g) xs → POP f xs → POP g xs
ap_SOP :: POP (f -.-> g) xs → SOP f xs → SOP g xs
```

Armed with pure and ap, we can define a host of derived functions; for example, we can define various variations on liftA (which is like map), such as

```
liftA_NP :: SingI xs ⇒ (∀a. f a → g a) → NP f xs → NP g xs
liftA_NP f xs = pure_NP (Fn f) `ap_NP` xs
```

as well as various variations on liftA2 (which is like zipWith), such as

```
cliftA2_NP :: (All c xs, SingI xs) ⇒ Proxy c
           → (∀a. c a ⇒ f a → g a → h a)
           → NP f xs → NP g xs → NP h xs
cliftA2_NP p f xs ys = cpure_NP p (fn_2 f) `ap_NP` xs `ap_NP` ys
```

Figure 2 provides an overview.

4.3 Collapsing to homogeneous structures

If we instantiate our *n*-ary products with the constant functor K (cf. Section 2) we get a homogeneous product that we can collapse to a list.

```
data family Sing (a :: k) :: *

data instance Sing (a :: [k]) where
  SNil  :: Sing '[]
  SCons :: (SingI x, SingI xs) ⇒ Sing (x ': xs)

data instance Sing (a :: *) where
  SStar :: Sing (a :: *)

class SingI (a :: k) where
  sing :: Sing a

instance SingI (a :: *) where
  sing = SStar

instance SingI '[] where
  sing = SNil

instance (SingI k, SingI ks) ⇒ SingI (k ': ks) where
  sing = SCons
```

Figure 1. Singletons

```
liftA_NP   :: Singl xs  ⇒ (∀a. f a → g a) → NP  f xs  → NP  g xs
liftA_NS   :: Singl xs  ⇒ (∀a. f a → g a) → NS  f xs  → NS  g xs
liftA_POP  :: Singl xss ⇒ (∀a. f a → g a) → POP f xss → POP g xss
liftA_SOP  :: Singl xss ⇒ (∀a. f a → g a) → SOP f xss → SOP g xss

cliftA_NP  :: (All  c xs,  Singl xs)  ⇒ Proxy c → (∀a. c a ⇒ f a → g a) → NP  f xs  → NP  g xs
cliftA_NS  :: (All  c xs,  Singl xs)  ⇒ Proxy c → (∀a. c a ⇒ f a → g a) → NS  f xs  → NS  g xs
cliftA_POP :: (All² c xss, Singl xss) ⇒ Proxy c → (∀a. c a ⇒ f a → g a) → POP f xss → POP g xss
cliftA_SOP :: (All² c xss, Singl xss) ⇒ Proxy c → (∀a. c a ⇒ f a → g a) → SOP f xss → SOP g xss

liftA2_NP  :: Singl xs  ⇒ (∀a. f a → g a → h a) → NP  f xs  → NP  g xs  → NP  h xs
liftA2_NS  :: Singl xs  ⇒ (∀a. f a → g a → h a) → NP  f xs  → NS  g xs  → NS  h xs
liftA2_POP :: Singl xss ⇒ (∀a. f a → g a → h a) → POP f xss → POP g xss → POP h xss
liftA2_SOP :: Singl xss ⇒ (∀a. f a → g a → h a) → POP f xss → SOP g xss → SOP h xss

cliftA2_NP  :: (All  c xs,  Singl xs)  ⇒ Proxy c → (∀a. c a ⇒ f a → g a → h a) → NP  f xs  → NP  g xs  → NP h xs
cliftA2_NS  :: (All  c xs,  Singl xs)  ⇒ Proxy c → (∀a. c a ⇒ f a → g a → h a) → NP  f xs  → NS  g xs  → NS h xs
cliftA2_POP :: (All² c xss, Singl xss) ⇒ Proxy c → (∀a. c a ⇒ f a → g a → h a) → POP f xss → POP g xss → POP h xss
cliftA2_SOP :: (All² c xss, Singl xss) ⇒ Proxy c → (∀a. c a ⇒ f a → g a → h a) → POP f xss → SOP g xss → SOP h xss

cliftA2′_NP :: (All² c xss, Singl xss) ⇒ Proxy c → (∀xs. All c xs ⇒ f xs → g xs → h xs) → NP f xss → NP g xss → NP h xss
cliftA2′_NS :: (All² c xss, Singl xss) ⇒ Proxy c → (∀xs. All c xs ⇒ f xs → g xs → h xs) → NP f xss → NS g xss → NS h xss

collapse_NP  ::              NP  (K a) xs  → [a]
collapse_NS  ::              NS  (K a) xs  →  a
collapse_SOP :: Singl xss ⇒ SOP (K a) xss → [a]
collapse_POP :: Singl xss ⇒ POP (K a) xss → [[a]]

fromList :: (Alternative f, Singl xs) ⇒ [a] → f (NP (K a) xs)

sequence_NP  :: (Singl xs,  Applicative f) ⇒ NP  f xs  → f (NP  I xs)
sequence_NS  :: (Singl xs,  Applicative f) ⇒ NS  f xs  → f (NS  I xs)
sequence_SOP :: (Singl xss, Applicative f) ⇒ SOP f xss → f (SOP I xss)
sequence_POP :: (Singl xss, Applicative f) ⇒ POP f xss → f (POP I xss)
```

Figure 2. Useful combinators

```
collapse_NP :: NP (K a) xs → [a]
collapse_NP Nil          = [ ]
collapse_NP (K x :* xs) = x : collapse_NP xs
```

If we do the same for an n-ary sum, we have a homogeneous sum, from which we can extract a single element:

```
collapse_NS :: NS (K a) xs → a
collapse_NS (Z (K x)) = x
collapse_NS (S xs)    = collapse_NS xs
```

There are similar functions for POP and SOP that produce a [[a]] and a [a], respectively.

4.4 Constructing sums

The functions

```
Left  :: a → Either a b
Right :: b → Either a b
```

for Haskell's binary sum type Either are called *injections*. For our n-ary sums, we can define the type of injections as

```
type Injection (f :: k → ∗) (xs :: [k]) = f -.-> K (NS f xs)
```

For any n-ary sum we can then construct an n-ary product containing all injections into the sum[4]:

```
injections :: ∀xs f. Singl xs ⇒ NP (Injection f xs) xs
injections = case sing :: Sing xs of
  SNil  → Nil
  SCons → Fn (K ∘ Z) :* liftA_NP shift injections

shift :: Injection f xs a → Injection f (x ′: xs) a
shift (Fn f) = Fn (K ∘ S ∘ unK ∘ f)
```

[4] Although it might seem that injections is somewhat ad-hoc, categorically speaking it in fact corresponds rather precisely with pure_{NP}.

This is typically used in generic producers by applying a product of injections to a product of arguments to produce a value of a sum type:

```
apInjs_NP :: Singl xs ⇒ NP f xs → [NS f xs]
apInjs_NP = collapse_NP ∘ ap_NP injections
```

which has as special case

```
apInjs_POP :: Singl xss ⇒ POP f xss → [SOP f xss]
apInjs_POP = apInjs_NP
```

We will see examples in Sections 5.4 and 6.2.

4.5 Other combinators

Figure 2 shows a list of combinators for the SOP datatypes. We discussed the various liftA and liftA2 functions in Section 4.2, and collapse in Section 4.3.

The function fromList is dual to $\mathsf{collapse}_{NP}$ in that it creates a (homogeneous) product from a list, and fails if the list has the wrong length. Finally, $\mathsf{sequence}_{NP}$ and co are the analogue of

```
sequenceA :: Applicative f ⇒ t (f a) → f (t a)
```

The implementations of all these functions is straight-forward, and we omit them. There are two functions in the list, however, whose implementation is non-trivial: $\mathsf{cliftA2}'_{NP}$ and $\mathsf{cliftA2}'_{NS}$. These functions are useful if we have a sum of products (or product of products), and we want to process each inner product as a whole, rather than mapping a function individually over all the leaves. (We will see an example using cliftA2′ as well as fromList and sequence in Section 6.2).

At first glance it might seem that $\mathsf{cliftA2}'_{NP}$ is simply an alias for $\mathsf{cliftA2}_{NP}$, instantiating c at All c. However, as we saw earlier, type family applications must always be fully saturated; hence, that is not possible.

One solution to this problem is to use defunctionalisation (Eisenberg 2013), but this affects the entire development and introduces significant complications. Fortunately, there is another solution. We can *reify* All c as an explicit dictionary AllDict c[5]

```
data AllDict (c :: k → Constraint) (xs :: [k]) where
  AllDict :: All c xs ⇒ AllDict c xs
```

Crucially, we can construct a product of these dictionaries, provided that we know that the constraint holds at the leaves:

```
allDict_NP :: ∀(c :: k → Constraint) (xss :: [[k]]).
              (All² c xss, SingI xss)
           ⇒ Proxy c → NP (AllDict c) xss
allDict_NP p = case sing :: Sing xss of
  SNil  → Nil
  SCons → AllDict :* allDict_NP p
```

We can then use $\mathsf{allDict}_{\mathsf{NP}}$ to implement $\mathsf{cliftA2}'_{\mathsf{NP}}$

```
cliftA2'_NP :: (All² c xss, SingI xss) ⇒ Proxy c
            → (∀xs. All c xs ⇒ f xs → g xs → h xs)
            → NP f xss → NP g xss → NP h xss
cliftA2'_NP p f xs ys =
  pure_NP (fn_3 $ λAllDict → f)
    ‘ap_NP‘ allDict_NP p ‘ap_NP‘ xs ‘ap_NP‘ ys
```

We construct the product of dictionaries, and then provide that dictionary as an additional argument; by opening the dictionary we bring the original type class constraint back into scope. The implementation of $\mathsf{cliftA2}'_{\mathsf{NS}}$ is analogous.

5. Generic functions in SOP

In this section we discuss some example generic functions. We describe two consumers (reduction to normal form and comparing for equality), two producers (constructing a default value and producing Arbitrary values), and the generic computation of lenses into a record type. The latter is purely defined in terms of the code of datatype, and cannot easily be classified as either consumer or producer.

The functions in this section do not need any metadata about the datatypes they are working with. In the SOP universe we have described so far metadata is not present. The advantage is that functions that do not require metadata do not have to deal with metadata at all.

In Section 6 we will consider further examples of generic functions (e.g. (de-)serialization of JSON) that *do* make use of metadata.

5.1 Reduction to normal form

Haskell's NFData class captures types that can be fully evaluated:

```
class NFData a where rnf :: a → ()
```

The idea is that x will be evaluated to normal form when rnf x is demanded (evaluated to weak head normal form). For example, while

```
Add ⊥ ⊥ ‘seq‘ True
```

will happily evaluate to True,

```
rnf (Add ⊥ ⊥) ‘seq‘ True
```

evaluates to $\perp$. The generic instance for NFData illustrates nicely that generic functions in the SOP approach can be very concise

[5] In the `generics-sop` library, the AllDict constructor has an additional SingI xs constraint, which is sometimes useful and doesn't affect the definition of $\mathsf{allDict}_{\mathsf{NP}}$. We omit it here because we won't need it in the paper.

```
grnf :: (Generic a, All² NFData (Code a)) ⇒ a → ()
grnf = rnf ∘ collapse_SOP ∘ cliftA_SOP p (K ∘ rnf ∘ unI) ∘ from
  where
    p = Proxy :: Proxy NFData
```

We can understand this function by tracking the types. First we use from to translate from a to the generic representation SOP I (Code a). We then map rnf (modulo newtype wrapping and unwrapping) across this sum of products to get a value of type SOP (K ()) (Code a), which we can collapse to a list of type [()]. Finally, we can reduce that list to a single unit value through one more application of rnf.

We use All^2 in the type of grnf to require that the types of the leaves must all satisfy NFData. Typically grnf will be used to define class instances:

```
instance NFData Expr where
  rnf = grnf
```

It is also possible to provide *default signatures* for type classes to use a generic definition such as grnf as default implementation for a type class (this requires the `DefaultSignatures` extension). Then you can even provide empty instance declarations.

Using both the generic function and the type class in connection is standard, and it means that the behaviour of a generic function can be specialised for specific datatypes, and in particular for abstract datatypes—even if these datatypes are deeply nested. This is made possible by the fact that the generic conversion functions from and to are shallow and only translate one layer of the datatype.

5.2 Equality

The definition of grnf showed that the combinators from Section 4 give us powerful means to define functions very succinctly. However, nothing is stopping us from traversing the sum of products structure more directly if that is more convenient—or indeed use a combination of both.

```
geq :: (Generic a, All² Eq (Code a)) ⇒ a → a → Bool
geq a b = go sing (from a) (from b)
  where
    go :: ∀xss. (All² Eq xss)
       ⇒ Sing xss → SOP I xss → SOP I xss → Bool
    go SCons (Z xs)  (Z ys)  = and (collapse_NP
                                 (cliftA2_NP p aux xs ys))
    go SCons (S xss) (S yss) = go sing xss yss
    go _     _       _       = False

    aux :: Eq a ⇒ I a → I a → K Bool a
    aux (I x) (I y) = K (x == y)

    p = Proxy :: Proxy Eq
```

In this definition of generic equality we pattern match on the constructors of the sum directly to check that both values are an application of the same constructor, but then compare the products for equality using the $\mathsf{cliftA2}_{\mathsf{NP}}$ combinator from Section 4.2. As a minor complication, we need to pass a singleton in the helper function go, because it's required by $\mathsf{cliftA2}_{\mathsf{NP}}$.

5.3 Producing default values

The Default type class from the `data-default` package describes types that have a default value.

```
class Default a where def :: a
```

We can produce default values generically for any type with at least one constructor, provided that we can provide default values for each of the arguments of that constructor:

```
gdef :: ∀a xs xss.
    (Generic a, Code a ∼ (xs ': xss), All Default xs) ⇒ a
gdef = case sing :: Sing (Code a) of
    SCons → to ∘ Z $ cpure_NP p (I def)
  where
    p = Proxy :: Proxy Default
```

The pattern match on sing is necessary to bring a SingI xs into scope. We use cpure$_{NP}$ to create a product of default values for the arguments of the first constructor, selected by applying Z.

We can use this function to create a Default instance for Expr:

```
instance Default Expr where def = gdef
```

so that def :: Expr will evaluate to .

In Section 1, we have claimed that SOP makes it easy to define functions that operate on the shape of the underlying datatypes in non-local ways. Let us therefore explore a variant of gdef that does not simply choose the first constructor, but instead chooses a constructor with the minimal number of arguments—in particular, it would choose a nullary constructor if one is present.

To achieve this, we first define an auxiliary function gdef′ that produces a list of default values, one of each constructor, paired with the arity of that constructor:

```
gdef' :: (Generic a, All² Default (Code a)) ⇒ [(Int, a)]
gdef' = collapse_NP $ liftA2_NP aux (cpure_POP p (I def)) injections
  where
    aux np (Fn inj) = K (length_NP np, to (unK (inj np)))
    p = Proxy :: Proxy Default
```

This function is similar to apInjs$_{NP}$ from Section 4.4. We create a product of all injections into the target type using injections. We wish to apply each of the injections to suitable default arguments generated using cpure$_{POP}$ p (I def). We combine the two products using liftA2$_{NP}$ with the function aux that not only performs the application and converts to the target type to, but also determines the arity of each constructor, which is the length of the product of default values.

We need the trivial additional function

```
length_NP :: NP f xs → Int
```

to compute the length of an *n*-ary product.

Finding the desired default value from the list of candidates is now simply a matter of applying standard list functions:

```
gdef :: ∀a xs xss. (Generic a,
    Code a ∼ (xs ': xss), All² Default (Code a)) ⇒ a
gdef = snd ∘ head $ sortBy (comparing fst) gdef'
```

5.4 Producing arbitrary values

The Arbitrary type class from the QuickCheck package (Claessen and Hughes 2000) is similar to Default, but captures the production of a *random* value instead[6]:

```
class Arbitrary a where arbitrary :: Gen a
```

The monad Gen is defined in the QuickCheck package for producing random values; one combinator predefined in QuickCheck that we will need is

```
elements :: [a] → Gen a
```

which picks a random element from a list. We will use elements to pick a constructor:

```
garbitrary :: ∀a. (Generic a, All² Arbitrary (Code a)) ⇒ Gen a
garbitrary = liftM to $ do
    branch ← elements (apInjs_POP (cpure_POP p arbitrary))
    sequence_SOP branch
  where
    p = Proxy :: Proxy Arbitrary
```

We first use cpure$_{POP}$ to create a product of product of generators; i.e., a generator for each argument of each constructor. We then use apInjs$_{POP}$ to pairwise apply the injections of the sum to each inner product; each element in the list corresponds to one of the constructors of the datatype. We use elements to pick one of these, which has type SOP Gen (Code a), and finally use sequence$_{SOP}$ to run all the generators for this constructor to obtain random values for each of the constructor arguments.

The naive implementation of garbitrary given above provides no control over the size of structures that are generated. In fact, depending on the datatypes it is called, it is quite possible that it will effectively generate infinite values. As already indicated in Section 1, for a production-grade implementation of garbitrary, you want to do a significant amount of additional work, such as making use of the size parameter provided by Gen and decreasing that before descending into substructures, ensuring that only "small" constructors are chosen if the desired size is small, and tweaking the relative probabilities of the constructors. In fact, even then you might want to take application-specific knowledge into account and make garbitrary configurable by providing extra metadata that tweaks the generation process. Generating high-quality random test cases is by no means trivial, and beyond the scope of this paper.

5.5 Lenses

A lens (Foster et al. 2007), in its simplest guise, is a combination of a setter and a getter:

```
data Lens a b = Lens (a → b) (b → a → a)
```

Lenses are a useful abstraction because they compose: it is easy to define

```
instance Category Lens where
```

which gives us

```
id  :: Lens a a
(∘) :: Lens b c → Lens a b → Lens a c
```

We can define some simple lenses for our SOP datatypes:

```
lens_Rep :: Generic a ⇒ Lens a (Rep a)
lens_SOP :: SingI xs ⇒ Lens (SOP f '[xs]) (NP f xs)
lens_hd  :: Lens (NP f (x ': xs)) (f x)
lens_tl  :: Lens (NP f (x ': xs)) (NP f xs)
lens_I   :: Lens (I a) a
```

More interestingly, given a product, we can define a product of projection lenses:

```
lens_NP :: ∀xs. SingI xs ⇒ NP (Lens (NP I xs)) xs
lens_NP = case sing :: Sing xs of
    SNil  → Nil
    SCons → lens_I ∘ lens_hd :* liftA_NP (∘lens_tl) lens_NP
```

This is useful because we can now define a generic function that computes a product of lenses for a record type:

```
glenses :: ∀a xs. (Generic a, Code a ∼ '[xs]) ⇒ NP (Lens a) xs
glenses = case sing :: Sing (Code a) of
    SCons → liftA_NP (λl → l ∘ lens_SOP ∘ lens_Rep) lens_NP
```

The type equality constraint (Code a ∼ '[xs]) on glenses states that we can only compute lenses for single-constructor types.

For example, given the datatype

```
data Point = Point { _x :: Double, _y :: Double }
```

[6] We ignore shrink for the sake of simplicity.

with a Generic instance, we can define lenses into Point using

```
x, y :: Lens Point Double
(x, y) = extract glenses
  where
    extract :: NP f '[x, y] → (f x, f y)
    extract (x :* y :* Nil) = (x, y)
```

Lenses are usually computed through Template Haskell, but we can give a fully typed alternative in the SOP universe.

6. Metadata

Many generic functions, though by no means all, need metadata about the type they are working with: the name of the type, names of constructors, names of record fields, etc. Traditionally this information is included directly in the generic universe, but this has two disadvantages. The definition of generic functions which are independent of the metadata is obscured by having to deal with it. Moreover, it means it is difficult to change the metadata, or extend the universe with additional, application-specific metadata.

6.1 Traditional metadata in SOP

With GADTs and the availability of the code of a datatype available as a first-class entity, we can define metadata completely separate from the universe proper. For instance, we might define metadata that records the names of types, constructors and record field names as follows[7]:

```
type Name = Text

data TypeInfo :: [[∗]] → ∗ where
  ADT :: Name → NP ConInfo xss → TypeInfo xss
  New :: Name → ConInfo '[x] → TypeInfo '[ '[x]]

data ConInfo :: [∗] → ∗ where
  Con :: SingI xs ⇒ Name → ConInfo xs
  Rec :: SingI xs ⇒ Name → NP (K Name) xs → ConInfo xs

class HasTypeInfo a where
  typeInfo :: Proxy a → TypeInfo (Code a)
```

For example:

```
instance HasTypeInfo Expr where
  typeInfo _ = ADT "Expr" $
       Con "Num"
    :* Rec "Add" (K "left" :* K "right" :* Nil)
    :* Nil
```

The TypeInfo datatype is yet another interpretation of SOP codes: it is indexed over types of kind [[∗]]. The New constructor, used to indicate that something is a newtype, is only applicable to types with a single constructor with a single field, since it is only applicable to types whose code is [[x]]. Thus, pattern matching on the metadata may reveal something about the shape of the datatype.

For constructors, we distinguish between ordinary constructors and record constructors, and only for the latter do we get a list of field names, precisely one for each field in the record.

In a universe with an arbitrary nesting of binary sums and products, it is more difficult to give such a clean definition of metadata. For instance, if we have binary products, where do we attach the information about record field names? In GHC.Generics, this information is distributed throughout the generic representation of a type, with various implicit conventions such as "if one argument to a constructor has a record field name, then they must all have a record field name". Not only is that unsatisfactory from a typing perspective, it also makes it more difficult to write generic functions.

6.2 Generic JSON encoder and decoder

As a somewhat more elaborate example of a generic function (that happens to make use of the metadata we just described) we will define a generic JSON encoder and decoder, based on the type classes defined in the `aeson` package:

```
class ToJSON   a where toJSON    :: a → Value
class FromJSON a where parseJSON :: Value → Parser a
```

The datatype Value is `aeson`'s representations of JSON values; all we need to know about Parser is that it satisfies MonadPlus. We encode a normal constructor as a tag and a list of values, aided by

```
con :: Text → [Value] → Value
```

and a record constructor as a tag and an object:

```
rec :: Text → [(Text, Value)] → Value
```

For example, we encode Add (Num 1) (Num 2) as

```
{"Add" : {"left" : {"Num" : [1]}, "right" : {"Num" : [2]}}}
```

For the decoder we rely on two additional auxiliary functions, which are essentially inverses to con and rec:

```
unCon :: Text → Value → Parser [Value]
unRec :: Text → Text → Value → Parser Value
```

The function unCon verifies the tag and that the payload is a list, and returns that list, whereas unRec verifies the tag and that the payload is an object, and looks up a field in that object.

Both the encoder and decoder are shown in Figure 3. The encoder is relatively straightforward. We translate the value to be encoded to its generic representation in gtoJSON and combine this with the metadata in gtoJSON′. For a regular constructor encCon calls toJSON on each argument (modulo some wrapping and unwrapping of newtypes), translates the resulting product to a list and then calls con. For record constructors. we pair the field names with the encoded arguments and then call rec.

The decoder is more interesting. For a normal constructor, decCon uses unCon to get a list of values, passes that to fromList to get a product of values, maps parseJSON to get a product of parsers, and finally uses $sequence_{NP}$ to get a parser of a product. For record constructors we do something similar, except that we look up every value of the record in the JSON object. Since unCon and unRec fail if the tag does not match, these parsers will fail if the encoded value does not correspond to this particular constructor. Then in gparseJSON′ we use injections to lift the result of these parsers into the sum of products, and finally choose the right parser (if any) using msum. Ultimately, gparseJSON lifts the result of the final parser out of the representation type.

We can use gtoJSON and gparseJSON to give ToJSON and FromJSON instances:

```
instance ToJSON   Expr where toJSON    = gtoJSON
instance FromJSON Expr where parseJSON = gparseJSON
```

We have kept the decoder and encoder simple for the sake of presentation; the `generics-sop` contains a "higher quality" version that avoids unnecessary tags, produces better error messages, checks for unexpected fields in objects, and more. The `generics-sop` version also performs a pre-processing step on the metadata, transforming the generic metadata into a shape that contains precisely what is needed for the JSON encoder and decoder. Having the metadata available separately makes such a transformation step very natural to write.

[7] In `generics-sop`, metadata is provided that is very similar to what we discuss here, but contains slightly more information. When using Template Haskell to derive a Generic instance for a particular datatype, an instance for the class corresponding to HasTypeInfo will be generated as well.

```
gtoJSON :: ∀a. (Generic a, HasTypeInfo a, All² ToJSON (Code a)) ⇒ a → Value
gtoJSON = gtoJSON′ (typeInfo (Proxy :: Proxy a)) ∘ from

gtoJSON′ :: (All² ToJSON xss, SingI xss) ⇒ TypeInfo xss → SOP I xss → Value
gtoJSON′ (ADT _ cs) = collapse_NS ∘ cliftA2′_NS pt encCon cs
gtoJSON′ (New _ c)  = collapse_NS ∘ cliftA2′_NS pt encCon (c :* Nil)

encCon :: All ToJSON xs ⇒ ConInfo xs → NP I xs → K Value xs
encCon (Con n)    = K ∘ con n ∘ collapse_NP ∘ cliftA_NP  pt (λ     (I a) → K (  toJSON a))
encCon (Rec n fs) = K ∘ rec n ∘ collapse_NP ∘ cliftA2_NP pt (λ(K f) (I a) → K (f, toJSON a)) fs

pt = Proxy :: Proxy ToJSON
```

```
gparseJSON :: ∀a. (Generic a, HasTypeInfo a, All² FromJSON (Code a)) ⇒ Value → Parser a
gparseJSON = liftM to ∘ gparseJSON′ (typeInfo (Proxy :: Proxy a))

gparseJSON′ :: ∀xss. (All² FromJSON xss, SingI xss) ⇒ TypeInfo xss → Value → Parser (SOP I xss)
gparseJSON′ (ADT _ cs) v = let injs = injections :: NP (Injection (NP I) xss) xss
                           in msum ∘ collapse_NP $ cliftA2′_NP pf (λ(Fn inj) → K ∘ liftM (unK ∘ inj) ∘ decCon v) injs cs
gparseJSON′ (New _ c)  v = Z ‘liftM‘ decCon v c

decCon :: All FromJSON xs ⇒ Value → ConInfo xs → Parser (NP I xs)
decCon v (Con n)    = sequence_NP ∘ cliftA_NP pf (parseJSON ∘ unK) =<< fromList =<< unCon n v
decCon v (Rec n fs) = sequence_NP $ cliftA_NP pf (λ(K n′) → parseJSON =<< unRec n n′ v) fs

pf = Proxy :: Proxy FromJSON
```

Figure 3. Generic JSON encoder and decoder

6.3 Application specific metadata

Suppose we have a system with a large number of record types, such as

```
data Person = Person {name :: String, age :: Int}
```

and suppose further that we wanted to write a generic validation function for all these records. This means that we will need more metadata, specifying what it means for particular components of particular datatype to be valid. Since we have set things up so that we can define metadata independent from the generic universe, we can also define application-specific metadata. For this concrete example, we might define

```
class ValidationRules a where
  validationRules :: Proxy a → POP (I -.-> K Bool) (Code a)
```

In words, in order to validate something, we need a validation function from a → Bool for each constructor argument of type a, modulo some newtype wrapping. For Person, we might define

```
validName :: (I -.-> K Bool) String
validName = Fn $ λ(I n) → K (not (null n))

validAge :: (I -.-> K Bool) Int
validAge = Fn $ λ(I n) → K (n ⩾ 0)

instance ValidationRules Person where
  validationRules _ = (validName :* validAge :* Nil) :* Nil
```

We can now define a generic validator:

```
validate :: ∀a. (Generic a, ValidationRules a) ⇒ a → Bool
validate = and ∘ collapse_SOP ∘ ap_SOP rules ∘ from
  where
    rules = validationRules (Proxy :: Proxy a)
```

The validator is very simple: we use $\mathsf{ap}_{\mathsf{SOP}}$ to apply each validation function to each argument, $\mathsf{collapse}_{\mathsf{SOP}}$ the result to a list of Bools, and finally take their conjunction.

As another example of domain specific metadata, one might consider a domain specific permission language, perhaps for a database server, with rules for each of the fields of the records in the database. Any such example, with metadata associated with each of the fields of a datatype, is easy to express once we have access to the codes of the universe.

7. Related work

In the following, we make a selection of a number of generic programming approaches that we believe to be related to the SOP view and compare them to our work.

7.1 Sum-of-products approaches

As mentioned in various places throughout this article, there is no shortage of approaches that makes use of a binary sum-of-products view where sums and products can be nested without restriction. This includes (for Haskell) the built-in GHC.Generics (Magalhães et al. 2010) and the `generic-deriving` package that builds on it, but also `instant-generics` (Chakravarty et al. 2009), `regular` (Van Noort et al. 2010), `multirec` (Rodriguez Yakushev et al. 2009), `compdata` (Bahr and Hvitved 2011), and older preprocessor or non-Haskell approaches such as PolyP (Jansson and Jeuring 1997), Generic Haskell (Hinze 2002; Löh 2004), or Generic Clean (Alimarine and Plasmeijer 2001).

While these approaches differ in many details, most importantly their treatment of recursion, they all share that the sum and product layer are not clearly separated on the type-level, encouraging a programming style that predominantly makes use of explicit induction on the structural representation where function make frequent use of implicit assumptions. Metadata, if handled at all, is mixed into the representation. On the other hand, some of these approaches have a good story on handling parameterised datatypes, whereas this remains future work for SOP (cf. Section 8).

7.2 Traversal-based approaches

There is also a large class of libraries that focus on traversals over data structures such as Scrap your Boilerplate (syb) (Lämmel and Peyton Jones 2003), `uniplate` (Mitchell and Runciman 2007), `multiplate` (O'Connor 2011), or `kure` (Sculthorpe et al. 2014). They share with SOP the desire to define generic functions by combining high-level combinators. They are mostly based on a structural representation of concrete *values*, often providing a list-like interface to the value structure known as the Spine view (Hinze et al. 2006), which reflects the product structure, but omits the sum structure (as a value has always been created by one particular constructor application).

These approaches make the definition of consumers very easy and appealing, but often fail to provide an equally simple way to deal with producers or functions that are just based on the structure of the type—without a concrete value to traverse in hand. Metadata is usually available separately, but without connection to the structural representation and therefore without type-level guarantees that it is being used correctly.

7.3 Template Haskell

Template Haskell (TH) (Sheard and Peyton Jones 2002) is a meta-programming solution for Haskell that gives the programmer access to an abstract syntax tree of datatype definitions. This abstract syntax represents the datatype faithfully. One can define meta-programs based on this structure and splice them back into Haskell programs as first-class definitions. The expressive power of TH is therefore unsurpassed. One has nearly as much information and possibilities as the compiler itself. But the power comes at a price: There are no advance checks that meta-programs are guaranteed to produce valid code; checking is for the most part performed only when a template is instantiated. Furthermore, access to all the details means that there is a lot of information in the abstract syntax that is not directly relevant to the definition of generic functions and that must be filtered out manually.

But these disadvantages do not prevent TH from being useful as a basis of other generic programming approaches: many approaches, including our own SOP, use TH for the generation of the structural representations of datatypes. Some approaches such as e.g. `uniplate` and `syb` have variants that make use of TH as a back-end for the generation of efficient code (Augustsson 2011; Adams and DuBuisson 2012).

7.4 Haskell approaches with a more precise representation

Holdermans et al. (2006) introduce *generic views* for a language variant of Haskell called Generic Haskell. One of the views discussed makes use of lists to represent sums and products, but this is not explored in much detail. Also, generic functions in Generic Haskell are not first-class, and using higher-order combinators to define generic functions would be awkward if not impossible.

The `RepLib` library (Weirich 2006) makes use of type representation that separate the sum and the product structure from each other, use list-like structures to represent each of them, and clearly nest them. However, the representation is less uniform than in SOP and has metadata mixed into the representation. Also, the approach predates several of the more recent GHC extensions. As a result, the library seems to encourage functions defined by induction rather than using combinators, and the overall look and feel is somewhat more complex.

In the `gdiff` package for datatype-generic diff (Lempsink et al. 2009), an application-specific universe is used that employs a list-like view of the product structure of the datatype (but pre-dating data kinds), but only marginally reflects the list-like sum structure.

Magalhães (2012) explores the use of data kinds, kind polymorphism, and other recent GHC extensions to refine various approaches to generic programming, trying to make the underlying universes more precisely typed. However, he sticks to the basic choices of the universes he bases his refinements on, which means arbitrarily nested binary sums for some, and product-only value representations for others.

Magalhães and Löh (2013) (in an unpublished draft version) discuss a universe called `structured` which is supposed to be very faithful, almost TH-like yet typed representation of Haskell datatypes. It makes use of a properly nested sum of product structure, where sums and products can even be type-level trees rather than lists. However, the universe has metadata mixed in and is in general very complex. In the article, it is considered only as a base universe for defining transformations into other, simpler universes.

7.5 Dependently typed programming

In the context of dependently typed programming, types are more precise, so it is not surprising that more precise and essentially list-like representations of sums of products have a somewhat longer history there than in Haskell. Nevertheless, binary sums of products are common also in the dependently typed setting (Altenkirch et al. 2007, for example).

Benke et al. (2003) discuss various universes that can be used to define generic programs and proofs in a dependently typed calculus. Some make use of list-like sums and products. The focus is on how expressive the universes are, actual programming is not discussed in any detail. On the other hand, many other universes presented even there go far beyond the types that SOP can represent.

A more recent example is (Chapman et al. 2010), which aims at using codes as the primary way to represent and define datatypes. It is therefore mandatory that their codes are as precise as it gets. They make use of a type of codes which is itself dependently typed.

7.6 Handling metadata

Most generic programming approaches—if they deal with metadata at all—mix the metadata into the structural representation. This sometimes (as in GHC.Generics) leads to extra complexity in all functions. Sometimes (such as e.g. in Generic Clean (Alimarine and Plasmeijer 2001)), this complexity is hidden from the user, by automatically generating good defaults when the cases are not needed. Another option is to actually use different universes, both with and without metadata.

The approach of SOP to define metadata separately, but use the type system to ensure that it aligns with the structure of the datatype, is—to our knowledge—new. It is the only approach that makes it easy to define domain-specific metadata. The separation is similar in style to the idea of ornaments (McBride 2010) that more generally describe additional structure that can be attached to an existing datatype.

8. Future work and conclusions

We have presented the SOP library for generic programming with a precisely typed universe of n-ary sums of products. We believe that the library is easy to use for a wide range of applications—in particular those that require more global information about the shape of the underlying datatype, and those that need additional application-specific metadata. We are using the library ourselves. In particular, the difficulty of expressing more advanced versions of the lens computation and JSON translation functions using GHC.Generics have driven us to explore this avenue.

The library as we have described it in this paper suits our current needs perfectly, as indeed we believe it will suit many other applications. However, there are some areas in which it could be extended.

Representing higher-kinded types In our version of SOP we can represent Maybe Int but not Maybe itself. A representation of parameterised types is helpful if one wants to define generic functions that range over type constructors, such as fmap or traverse. Some libraries offer separate representations for $* \rightarrow *$ datatypes, and some approaches are yet more flexible.

Fixed points Several approaches to generic programming treat recursion by identifying fixed points and representing the underlying functor. We do not. For functions that do not require specific treatment of recursion, this only makes things simpler. Once again, there is no fundamental problem in combining a more precise universe using list-like sums of products with the fixed point approach.

Representing existentials and GADTs Not even all (concrete) types of kind $*$ can be represented by SOP. We fail for types that involve existentially quantified type variables, embedded class constraints, or for GADTs (which have embedded equality constraints).

None of the ideas above are fundamentally incompatible with SOP. For some of these there are well-known techniques in other approaches that can most likely be easily combined with SOP. However, in doing so, we may have to extend or modify the underlying representation and may lose some of the simplicity that makes the SOP view so appealing.

Regardless of the concrete design decisions we made for SOP, we think that the following lessons are relevant to all generic programmers: Design a universe that *precisely* reflects the structure of the underlying types, not allowing flexibility that is not used and encourages making implicit assumptions. Strive for the development of a library of high-level combinators that can be reused that allow a compositional approach to defining generic functions. And separate metadata, again using the type system to maintain shape constraints, so that it becomes easy to work with metadata as needed, and tweak it to application-specific needs.

Acknowledgements

We thank Francis Nevard and Nicolas Wu. They asked Well-Typed to work on the project that sparked the development and use of the `generics-sop` library. We are also grateful to Paolo Capriotti for answering several questions about category theory, and to the anonymous reviewers for their helpful suggestions.

References

Michael D. Adams and Thomas M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Haskell '12*, pages 13–24. ACM, 2012.

Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In *IFL*, volume 2312 of *LNCS*, pages 168–185. Springer, 2001.

Thorsten Altenkirch, Conor McBride, and Peter Morris. Generic programming with dependent types. In *SSDGP '06*, pages 209–257. Springer, 2007.

Lennart Augustsson. geniplate: Use Template Haskell to generate Uniplate-like functions, 2011. URL `http://hackage.haskell.org/package/geniplate`.

Patrick Bahr and Tom Hvitved. Compositional data types. In *WGP '11*, pages 83–94. ACM, 2011.

Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, December 2003.

Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy, 2009. URL `http://www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *ICFP '10*, pages 3–14. ACM, 2010.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00*, pages 268–279. ACM, 2000.

Richard A. Eisenberg. Defunctionalization for the win, 2013. URL `http://typesandkinds.wordpress.com/2013/04/01/defunctionalization-for-the-win/`.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Haskell '12*, pages 117–130. ACM, 2012.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007. ISSN 0164-0925.

Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In *FLOPS*, volume 3945 of *LNCS*, pages 13–29. Springer, 2006.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez Yakushev. Generic views on data types. In *MPC*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006.

Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *POPL '97*, pages 470–482. ACM, 1997.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03*, pages 26–37. ACM, 2003.

Eelco Lempsink, Sean Leather, and Andres Löh. Type-safe diff for families of datatypes. In *WGP '09*, pages 61–72. ACM, 2009.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Universiteit Utrecht, 2004.

José Pedro Magalhães. The right kind of generic programming. In *WGP '12*, pages 13–24. ACM, 2012.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Haskell '10*, pages 37–48. ACM, 2010.

José Pedro Magalhães and Andres Löh. Generic generic programming, 2013. URL `http://www.andres-loeh.de/GenericGenericProgramming/`. Unpublished original draft of Magalhães and Löh (2014).

José Pedro Magalhães and Andres Löh. Generic generic programming. In *PADL*, volume 8324 of *LNCS*, pages 216–231. Springer, 2014.

Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

Conor McBride. Ornamental algebras, algebraic ornaments. 2010. Submitted to Journal of Functional Programming.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Haskell '07*, pages 49–60. ACM, 2007.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren, and José Pedro Magalhães. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(Special Issue 3–4):375–413, 2010.

Russell O'Connor. Functor is to lens as applicative is to biplate: Introducing multiplate. *CoRR*, abs/1103.2841, 2011.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP '09*, pages 233–244. ACM, 2009.

Neil Sculthorpe, Nicolas Frisby, and Andy Gill. The Kansas University Rewrite Engine: A Haskell-embedded strategic programming language with custom closed universes. Submitted to the Journal of Functional Programming, 2014. URL `http://www.cs.swan.ac.uk/~csnas/papers_and_talks/kure.pdf`.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02*, pages 1–16. ACM, December 2002.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07*, pages 53–66. ACM, 2007.

Stephanie Weirich. RepLib: a library for derivable type classes. In *Haskell '06*, pages 1–12. ACM, 2006.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *TLDI '12*, pages 53–66. ACM, 2012.