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

\[
data \text{ Tree } a = \text{ Leaf } | \text{ Branch } (\text{ Tree } a) (\text{ Tree } a)
\]

using the \texttt{GHC.Generics} library is essentially isomorphic to

\[
\text{Either } () (\text{ Tree } a, (a, \text{ Tree } a))
\]

where \texttt{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 \texttt{garity} that counts the arities of all constructors of a datatype. Using \texttt{GHC.Generics} (Magalhães et al. 2010), a possible implementation is as follows:

\[
\text{class GArities } (a :: \ast \rightarrow \ast) \text{ where} \\
garities :: \text{Proxy } a \rightarrow [\text{Int}]
\]

\[
\text{instance GArities } f \Rightarrow \text{GArities } (\text{M1 } i c f) \text{ where} \\
garities _ = \text{garities } (\text{Proxy } :\text{Proxy } f)
\]

\[
\text{instance GArities } V1 \text{ where garities } _ = [1]
\]

\[
\text{instance GArities } U1 \text{ where garities } _ = [0]
\]

\[
\text{instance GArities } (K1 R a) \text{ where garities } _ = [1]
\]
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

\[
\begin{align*}
\text{id} &:: a \rightarrow a \\
\text{const} &:: a \rightarrow b \rightarrow a
\end{align*}
\]

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

\[
\begin{align*}
\text{newtype} \; l \; (a :+: b) &= \{ \text{unl} :: a \} \\
\text{newtype} \; K \; (a :+: k) &= K \{ \text{unk} :: a \}
\end{align*}
\]

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.

Polyorphic 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 \((k :': k)\) 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 `Show a` in
```
show :: Show a ⇒ a → String
```
Under the ConstraintKinds extension, `Show` is a type like another, albeit of a special kind:
```
Show :: * → 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 `* → 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 \([\star \star]\). The goal of the universe is to provide descriptions of Haskell datatypes of (kind `\star`). We use a type family to map a datatype to its code:
```
type family Code (a :: \star) :: [\star \star]
```
A type of kind \([\star \star]\) 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 description that we can operate on. Consider the following simple:
```
data Expr = Num Int | Add (left :: Expr, right :: Expr)
```
The code for `Expr` looks as follows:
```
type instance Code Expr = \('[\star Int], '[Expr, Expr]\)
```
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 \([\star \star]\), 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 f (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 f (Code a)
class Singl (Code a) ⇒ Generic (a :: \star) 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 `Singl` 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 :: \(\star\) → \(\star\) where  -- preliminary
    Nil :: NP \('\)\n    (\(\cdot\)) :: \(\cdot\) → NP \(x\) → NP (\(x\); \(x\))
```
For example, the nested pair
```
(True, ('\x\', 3)) :: (Bool, (Char, Int))
```
corresponds to
```
True :: '\x\' \$ 3 :: Nil :: NP ['\Char\', \'Int\']
```
(assuming that `\(\cdot\)` is right-associative, just like ordinary \(\cdot\) for lists).

We will however often need a product
```
f \(T_1 \times \cdots \times T_n\)
```
for some functor \(f\); so we choose to define `NP` with functor application “built-in”:
```
data NP :: \(k \rightarrow \star\) → \(k\) → \(\star\) where
    Nil :: NP \('\)
    (\(\cdot\)) :: \(\cdot\) → NP \(f\) \(x\) → NP \(f\) \(x\); \(x\)
```
The nested pair from above can still be expressed by choosing the identity functor \(I\) (cf. Section 2) for \(f\):
```
I True :: '\x\' \$ 13 :: Nil :: NP I ['\Char\', \'Int\']
```
We can define \(n\)-ary sums in a similar manner:
```
data NS :: \(k \rightarrow \star\) → \(k\) → \(\star\) where
    Z :: f \(x\) → NS \(f\) \(x\); \(x\)
    S :: \(f\) \(x\); \(x\) → NS \(f\) \(x\); \(x\)
```
The constructor names are reminiscent of Peano naturals. The constructor `Z` injects into the first component of a sum (with at least one component), `S` injects into the second component of a sum (with at least two components), and so on:
```
Z :: f \(x\) → NS \(f\) \(x\); \(x\)
S :: \(f\) \(x\); \(x\) → NS \(f\) \(x\); \(y\); \(x\)
S ∘ S :: \(f\) \(x\); \(y\); \(z\) → NS \(f\) \(x\); \(y\); \(z\); \(x\)
```
By nesting NS and NP applications, we can define both SOP and POP:

\[
\text{type SOP \( f :: k \rightarrow * \) (xss :: [:k]) = NS (NP \( f \) \) xss} \\
\text{type POP \( f :: k \rightarrow * \) (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 \( k \rightarrow * \), the first argument of the outer application has kind \( [k] \rightarrow * \).

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:

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

The class instance looks for Expr as follows:

\[
\text{instance Generic Expr where} \\
\text{from (Num n) = Z} \ (\{ n :*: Nil \}) \\
\text{from (Add e f) = S} \ (Z \ (I \ e :*: I f :*: Nil)) \\
\text{to (Z (\{ n :*: Nil \}) ) = Num n} \\
\text{to (S (Z (I e :*: I f :*: Nil))) = Add e f}
\]

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:

\[
\text{type instance Code [a] = \{'[\text{Int}], '[a,[a]]\}}
\]

\[
\text{instance Generic [a] where} \\
\text{from [ ] = Z Nil} \\
\text{from (x : xs) = S (Z (I x :*: I xs :*: Nil))} \\
\text{to (Z Nil) = [ ]} \\
\text{to (S (Z (I x :*: I xs :*: Nil))) = x : xs}
\]

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 straightforward 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. 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

\[
pure_{\text{np}} :: (\forall a. f a) \rightarrow \text{NP} f \text{ xs} \quad -- \text{preliminary}
\]

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.\footnote{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.}

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 runtime. So even in the presence of a SingI \((a :: *)\) constraint, parametricity still holds. We can use singletons in the definition of \( \text{pure}_{\text{np}} \) as follows:

\[
pure_{\text{np}} :: \forall f. \text{SingI} f \text{ xs} \rightarrow (\forall a. f a) \rightarrow \text{NP} f \text{ xs} \\
pure_{\text{np}} f = \text{case sing :: SingI} f \text{ xs of} \\
\text{SNil} \rightarrow \text{Nil} \\
\text{SCons} \rightarrow f :\text{pure}_{\text{np}} f
\]

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.

\footnote{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:

\[
\text{pure}_{\text{pop}} :: \forall f\,xs. \text{Singl}\,xs \Rightarrow (\forall a. f\,a) \Rightarrow \text{POP}\,f\,xs
\]

\[
\text{pure}_{\text{pop}}\,f = \text{case}\,\text{sing} :: \text{Sing}\,\text{xs}\,\text{of}
\]

\[
\text{SNil} \Rightarrow \text{Nil}
\]

\[
\text{SCons} \to \text{pure}_{\text{pop}}\,f :: \text{pure}_{\text{pop}}\,f
\]

In practice, however, the types of \text{pure}_{\text{pop}} and \text{pure}_{\text{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 \text{pure}_{\text{pop}}:

\[
\text{cpure}_{\text{pop}} :: \forall c\,f\,xs. (\forall c\,xs, \text{Singl}\,xs) \Rightarrow (\forall a. c\,a \Rightarrow f\,a) \Rightarrow \text{NP}\,f\,xs
\]

\[
\text{cpure}_{\text{pop}}\,p\,f = \text{case}\,\text{sing} :: \text{Sing}\,\text{xs}\,\text{of}
\]

\[
\text{SNil} \Rightarrow \text{Nil}
\]

\[
\text{SCons} \to f :: \text{cpure}_{\text{pop}}\,p\,f
\]

The constraint \(\text{All}\,c\) requires all the types in \(\text{xs}\) to satisfy \(c\); we can define it as follows:

\[
\begin{align*}
\text{type family} & \quad \text{All} \ (c :: k \to \text{Constraint}) \ (xs :: [k]) :: \text{Constraint} \\
\text{type instance} & \quad \text{All} \ c \ '() = () \\
\text{type instance} & \quad \text{All} \ c \ (x :: xs) = (c\,x, \text{All}\,c\,xs)
\end{align*}
\]

For example, the application \(\text{All}\,\text{Eq}\,\text{[Bool, Char, Int]}\) expands to the constraint

\[
(\text{Eq}\,\text{Bool}, (\text{Eq}\,\text{Char}, (\text{Eq}\,\text{Int}, ())))
\]

which is equivalent to

\[
(\text{Eq}\,\text{Bool}, (\text{Eq}\,\text{Char}, \text{Eq}\,\text{Int}))
\]

The \(\text{Proxy}\,c\) 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 \(\text{Proxy}\):

\[
\text{data Proxy} \ (a :: k) = \text{Proxy}
\]

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

\[
\text{cpure}_{\text{pop}} \ (\text{Proxy} :: \text{Proxy}\,\text{Eq})
\]

\[
:: (\text{All}\,\text{Eq}\,xs, \text{Singl}\,xs) \Rightarrow (\forall a. \text{Eq}\,a \Rightarrow f\,a) \Rightarrow \text{NP}\,f\,xs
\]

Following the ideas developed for \text{cpure}_{\text{pop}}, we try to generalise \text{cpure}_{\text{pop}} to \text{cpure}_{\text{pop}} in a similar manner. Unfortunately, however, we cannot use

\[
\text{cpure}_{\text{pop}} \ (\text{All} \ (\text{All} \ (c :: k)\,xs, \text{Singl}\,xs))
\]

\[
:: (\forall c\,f\,xs. (\forall c\,xs, \text{Singl}\,xs) \Rightarrow (\forall a. c\,a \Rightarrow f\,a) \Rightarrow \text{POP}\,f\,xs
\]

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

\[
\begin{align*}
\text{type family} & \quad \text{All}^2 \ (c :: k \to \text{Constraint}) \ (xs :: [[k]]) :: \text{Constraint} \\
\text{type instance} & \quad \text{All}^2 \ c \ '[] = () \\
\text{type instance} & \quad \text{All}^2 \ c \ (x :: xs) = (\text{All}\,c\,x, \text{All}^2\,c\,xs)
\end{align*}
\]

and then define

\[
\text{cpure}_{\text{pop}} :: \forall c\,f\,xs. (\text{All}^2\,c\,xs, \text{Singl}\,xs) \Rightarrow (\forall a. c\,a \Rightarrow f\,a) \Rightarrow \text{POP}\,f\,xs
\]

We will come back to the problem of a partial application of \(\text{All}\) in Section 4.5.

### 4.2 Application

Having defined the analogue of \text{pure}, we need to define the analogue of \text{ap}. For \text{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:

\[
\text{newtype} \ (f \to g)\,a = \text{Fn} \ (\text{ap}\,f :: f\,a \to g\,a)
\]

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

\[
\begin{align*}
\text{fn}_2 :: (f\,a \to f'\,a \to f''\,a) \to (f \to f' \to f'')\,a \\
\text{fn}_3 :: (f\,a \to f'\,a \to f''\,a \to f'''\,a) \to (f \to f' \to f'' \to f''')\,a
\end{align*}
\]

Using \((\_\_\_\_)\), it is easy to define \text{ap}_{\text{pop}}:

\[
\begin{align*}
\text{ap}_{\text{pop}} :: \text{NP} \ (f \to g)\,xs \Rightarrow \text{NP}\,f\,xs \Rightarrow \text{NP}\,g\,xs \\
\text{ap}_{\text{pop}}\,\text{Nil} \text{ Nil} = \text{Nil} \\
\text{ap}_{\text{pop}}\,(\text{Fn}\,f \times f)\,(x :: xs) = f\,x :: \text{ap}_{\text{ap}}\,fs\,xs
\end{align*}
\]

While we cannot apply a sum of functions to a sum of arguments, we can apply a \text{product} of functions to a sum of arguments:

\[
\begin{align*}
\text{ap}_{\text{pop}} :: \text{NP} \ (f \to g)\,xs \Rightarrow \text{NS}\,f\,xs \Rightarrow \text{NS}\,g\,xs \\
\text{ap}_{\text{pop}}\,(\text{Fn}\,f \times)\,(Z\,x) = Z\,(f\,x) \\
\text{ap}_{\text{pop}}\,(\_ \_ :: f)\,(S\,xs) = S\,(\text{ap}_{\text{pop}}\,fs\,xs)
\end{align*}
\]

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

\[
\begin{align*}
\text{ap}_{\text{pop}} :: \text{POP} \ (f \to g)\,xs \Rightarrow \text{POP}\,f\,xs \Rightarrow \text{POP}\,g\,xs \\
\text{ap}_{\text{pop}} :: \text{POP} \ (f \to g)\,xs \Rightarrow \text{SOP}\,f\,xs \Rightarrow \text{SOP}\,g\,xs
\end{align*}
\]

Armored with \text{pure} and \text{ap}, we can define a host of derived functions; for example, we can define various variations on \text{lift}\text{A} (which is like \text{map}), such as

\[
\begin{align*}
\text{lift}\text{A}_{\text{ap}} :: \text{Singl}\,xs \Rightarrow (\forall a. f\,a \to g\,a) \Rightarrow \text{NP}\,f\,xs \Rightarrow \text{NP}\,g\,xs \\
\text{lift}\text{A}_{\text{ap}}\,f\,xs = \text{pure}_{\text{ap}}\,(\text{Fn}\,f)\,'\text{ap}_{\text{ap}}\,'\text{xs}
\end{align*}
\]

as well as various variations on \text{lift}\text{A}2 (which is like \text{zipWith}), such as

\[
\begin{align*}
\text{lift}\text{A}_{\text{ap}} :: (\forall a. c\,a \Rightarrow f\,a \Rightarrow g\,a \Rightarrow h\,a) \\
\text{lift}\text{A}_{\text{ap}}\,f\,xs \Rightarrow \text{NP}\,f\,xs \Rightarrow \text{NP}\,g\,xs \Rightarrow \text{NP}\,h\,xs \\
\text{lift}\text{A}_{\text{ap}}\,f\,ys = \text{cpure}_{\text{ap}}\,p\,(\text{fn}_2\,f)\,'\text{ap}_{\text{ap}}\,'\text{xs}'\text{ap}_{\text{ap}}\,'\text{ys}
\end{align*}
\]

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.

---

**Figure 1.** Singlets

**Figure 2.** Collapsing to homogeneous structures
lifting it in fact corresponds rather precisely with pureNP.

Homogeneous sums create one point from which we can extract a single element: POP

For any \( n \)-ary sum, we have a homogeneous sum, which has as special case POP. 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:

\[
\text{apInjs}_\text{pop} :: \text{Singl} \times f \Rightarrow \text{NP} \ f \ x s \to \text{NP} \ f \ x s
\]

which has as special case

\[
\text{apInjs}_\text{pop} = \text{apInjs}_\text{pop} \circ \text{apInjs}_\text{pop} \text{ injections}
\]

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 collapseNP in that it creates a (homogeneous) product from a list, and fails if the list has the wrong length. Finally, sequenceNP and co are the analogue of sequenceA::Applicative f => t (f a) -> t (f a)

The implementations of all these functions is straightforward, and we omit them. There are two functions in the list, however, whose implementation is non-trivial: cliftA2NP and cliftA2NP. 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 cliftA2NP is simply an alias for cliftA2NP, instantiating c at All c. However, as we saw earlier, type family applications must always be fully saturated; hence, that is not possible.

---

4.4 Constructing sums

The functions

\[
\begin{align*}
\text{collapsesNP} &:: \text{NP} \ (K a) \ x s \to [a] \\
\text{collapsesNil} &:: \text{Nil} \\
\text{collapsesZ} &:: (K x :\!\!:: x s) = x :\!\!:: \text{collapsesNP} \ x s
\end{align*}
\]

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

\[
\begin{align*}
\text{collapsesNP} &:: \text{NP} \ (K a) \ x s \to [a] \\
\text{collapsesNil} &:: \text{Nil} \\
\text{collapsesZ} &:: (K x :\!\!:: x s) = x :\!\!:: \text{collapsesNP} \ x s
\end{align*}
\]

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

\[
\text{data AllDict (c :: k \rightarrow \text{Constraint}) (xs :: [[k]]) where}
\]
\[
\text{AllDict :: All c xs } \Rightarrow \text{AllDict c xs}
\]

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

\[
\text{AllDict}_{\text{op}} :: \forall (c :: k \rightarrow \text{Constraint}) (xs :: [[k]]) .
\]
\[
\quad (\text{All c} \ xss, \text{Singl} \ xss) \Rightarrow \text{Proxy} \ c \rightarrow \text{NP} \ (\text{AllDict} \ c) \ xss
\]

\[
\text{allDict}_{\text{op}} \ p = \text{case sing :: Sing xss of}
\]
\[
\quad \text{SNil } \rightarrow \text{Nil}
\]
\[
\quad \text{SCons } \rightarrow \text{AllDict} :: \text{allDict}_{\text{op}} \ p
\]

We can then use \text{allDict}_{\text{op}} to implement \text{cliftA2}_{\text{op}}.

\[
\text{cliftA2}_{\text{op}} :: (\text{All} \ c \ xss, \text{Singl} \ xss) \Rightarrow \text{Proxy} \ c
\]
\[
\rightarrow \forall \text{xss}. \text{All c} \ xss \Rightarrow f \ xss \rightarrow g \ xss \rightarrow h \ xss)
\]
\[
\rightarrow \text{NP} f \ xss \rightarrow \text{NP} g \ xss \rightarrow \text{NP} h \ xss
\]
\[
\text{cliftA2}_{\text{op}} \ p \ f \ xss \ yss =
\]
\[
\text{p} \ 'apn' \ 'apn' \ 'apn' \ xss \ 'apn' \ yss
\]

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 \text{cliftA2}_{\text{op}} 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 the 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:

\[
\text{class NFData a where rnf :: a } \rightarrow \{\}
\]

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

\[
\text{Add} \ \perp \ \perp \ \text{seq} \ True
\]

will happily evaluate to \(True\),
\[
\text{rnf (Add} \ \perp \ \perp \ \text{seq} \ True
\]

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

\[
\text{grnf} :: \text{NFData (Code a)} \Rightarrow a \rightarrow \{\}
\]
\[
\text{grnf} = \text{rnf} \circ \text{collapse}_{\text{op}} \circ \text{cliftA}_{\text{op}} \ p \ \text{from}
\]
\[
\text{where}
\]
\[
\quad p = \text{Proxy} :: \text{Proxy NFData}
\]

We can understand this function by tracking the types. First we use \text{from} to translate from \(a\) to the generic representation SOP I (Code \(a\)). We then map \text{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 \text{rnf}.

We use All\(\text{c}\) in the type of \text{grnf} to require that the types of the leaves must all satisfy NFData. Typically \text{grnf} will be used to define class instances:

\[
\text{instance NFData Expr where}
\]
\[
\text{rnf} = \text{grnf}
\]

It is also possible to provide default signatures for type classes to use a generic definition such as \text{grnf} as default implementation for a type class (this requires the Default1Signature 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 \text{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.

\[
\text{geq} :: \text{Eq (Code a)} \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]
\[
\text{geq a b} = \text{go sing (from a) (from b)}
\]
\[
\text{where}
\]
\[
\quad \text{go} :: \forall \text{xss}. \text{Eq} \ xss
\]
\[
\rightarrow \text{Sing} \ xss \rightarrow \text{SOP} \ I \ xss \rightarrow \text{SOP} \ I \ xss \rightarrow \text{Bool}
\]
\[
\text{go SCons (Z xs)} \ (Z ys) = \text{true} \ (\text{collapse}_{\text{op}} \ p \ \text{aux} \ \text{xs} \ \text{ys})
\]
\[
\text{aux :: Eq} \ a \rightarrow \text{I} \ a \rightarrow \text{K} \ \text{Bool} \ a
\]
\[
\text{aux} \ (I x) \ (I y) = \text{K} \ (x :: y)
\]
\[
\text{p} = \text{Proxy} :: \text{Proxy Eq}
\]

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 \text{cliftA}_{\text{op}} combinator from Section 4.2. As a minor complication, we need to pass a singleton in the helper function \text{go}, because it’s required by \text{cliftA}_{\text{op}}.

5.3 Producing default values

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

\[
\text{class Default a where def :: a}
\]

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:
We ignore \( \text{sing} \) which picks a random element from a list. We will use \( \text{sing} \)ing random values; one combinator predefined in Quickcheck now simply a matter of applying standard list functions:

\[
\text{sing} = \text{sing} \,
\]

We first use \( \text{pure} \) to create a product of default values; i.e., a generator for each argument of each constructor. We then use \( \text{apInjs} \) 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 \( \text{elements} \) to pick one of these, which has type \( \text{SOP} \text{Gen} \), and finally use \( \text{sequenceSOP} \) to run all the generators for this constructor to obtain random values for each of the constructor arguments.

The naive implementation of \( \text{arbitrary} \) 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 \( \text{arbitrary} \), you want to do a significant amount of additional work, such as making use of the size parameter provided by \( \text{Gen} \) 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 \( \text{arbitrary} \) 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:

\[
\text{data Lens a b} = \text{Lens} \, (a \rightarrow b) \, (b \rightarrow a \rightarrow a)
\]

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

\[
\text{instance Category Lens where}
\]

which gives us

\[
\text{id} :: \text{Lens a a}
\]

\[
(\circ) :: \text{Lens b c} \rightarrow \text{Lens a b} \rightarrow \text{Lens a c}
\]

We can define some simple lenses for our SOP datatypes:

\[
\begin{align*}
\text{lensSOP} & :: \text{Generic a} \Rightarrow \text{Lens a (Rep a)} \\
\text{lensSOP} & :: \text{Singly f xs} \Rightarrow \text{Lens SOP f (xs')} (\text{NP f xs}) \\
\text{lensSOP} & :: \text{NP f (xs')} \Rightarrow \text{Lens (NP f xs')} \\
\text{lensSOP} & :: \text{Lens (NP f xs')} \\
\text{lensSOP} & :: \text{Lens (NP f xs)}
\end{align*}
\]

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

\[
\begin{align*}
\text{lensSOP} & :: \text{Vxs. Singly xs} \Rightarrow \text{NP (NP (NP l xs)) xs} \\
\text{lensSOP} & :: \text{case sing :: Sing xs of} \\
\text{Nil} & \rightarrow \text{Nil} \\
\text{SCons} & \rightarrow \text{lensSOP (\text{lensSOP (\text{lensSOP (\text{SCons})}))}}
\end{align*}
\]

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

\[
\begin{align*}
\text{g lenses} & :: \text{Vxs. (Generic a, Code a ~ [xs]) \Rightarrow (NP (NP (NP l xs)) x)} \\
\text{g lenses} & :: \text{case sing :: Sing (Code a) of} \\
\text{SCons} & \rightarrow \text{lensSOP (\text{lensSOP (\text{lensSOP (\text{SCons})}))}}
\end{align*}
\]

The type equality constraint \( \text{Code a ~ [xs]} \) on \( \text{g lenses} \) states that we can only compute lenses for single-constructor types. For example, given the datatype

\[
\text{data Point} = \text{Point} \, \{ \_x :: \text{Double}, \_y :: \text{Double} \}
\]
with a `Generic` instance, we can define lenses into `Point` using

```haskell
x, y :: Lens Point Double
(x, y) = extract lenses
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:

```haskell
type Name = Text

data TypeInfo :: [[*]] → * where
  ADT :: Name → NP ConInfo xs → TypeInfo xs
  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:

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

The `ConInfo` 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 for the latter 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:

```haskell
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

```haskell
con :: Text → [Value] → Value
and a record constructor as a tag and an object:

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

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

```haskell
{ "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`:

```haskell
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 `sequenceA` 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:

```haskell
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.
6.3 Application specific metadata

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

\[
\text{data Person} = \text{Person} \{ \text{name} :: \text{String}, \text{age} :: \text{Int} \}
\]

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

\[
\text{class ValidationRules a where}
\]

\[
\text{validationRules} :: \text{Proxy a} \rightarrow \text{POP} (\text{I} \rightarrow \text{K} \text{Bool}) (\text{Code a})
\]

In words, in order to validate something, we need a validation function from \(a \rightarrow \text{Bool}\) for each constructor argument of type \(a\), modulo some newtype wrapping. For \(\text{Person}\), we might define

\[
\text{validName} :: (\text{I} \rightarrow \text{K} \text{Bool}) \text{String}
\]

\[
\text{validAge} :: (\text{I} \rightarrow \text{K} \text{Bool}) \text{Int}
\]

\[
\text{instance ValidationRules Person where}
\]

\[
\text{validationRules} _\_ = (\text{validName} :: \text{validAge} :: \text{Nil}) :: \text{Nil}
\]

We can now define a generic validator:

\[
\text{validate} :: \forall a. (\text{Generic a, ValidationRules a}) \Rightarrow a \rightarrow \text{Bool}
\]

\[
\text{validate} = \text{and} \circ \text{collapse}_\text{ToJSON} \circ \text{ap}_\text{ToJSON} \text{fules} \circ \text{from}
\]

\[
\text{where}
\]

\[
\text{rules} = \text{validationRules} (\text{Proxy :: Proxy a})
\]

The validator is very simple; we use \(\text{ap}_\text{ToJSON}\) to apply each validation function to each argument, \(\text{collapse}_\text{ToJSON}\), the result to a list of \(\text{Bools}\), 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 metadata 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), the 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 (Syp) (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 \textit{values}, often providing a list-like interface to the value structure known as the Spine view (Hinze 1997), Generic Haskell (Hinze 2002; Löh 2004), or Generic Clean (Alimarine and Plasmeijer 2001).
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 backend for the generation of efficient code (Augustsson 2011; Adams and DuBuisson 2012).

7.4 Haskell approaches with a more precise representation
Holderness 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 ReplLib library (Weirich 2006) makes use of type representation that separates 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 (Lempink 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 coinductive types 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 actually to 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-kindred 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 map or traverse. Some libraries offer separate representations for *→* datatypes, and such 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


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


