Staged Sums of Products

Matthew Pickering
Department of Computer Science
University of Bristol
United Kingdom
matthew.pickering@bristol.ac.uk

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

Nicolas Wu
Department of Computing
Imperial College London
United Kingdom
n.wu@imperial.ac.uk

Abstract

Generic programming libraries have historically traded efficiency in return for convenience, and the generics-sop library is no exception. It offers a simple, uniform, representation of all datatypes precisely as a sum of products, making it easy to write generic functions. We show how to finally make generics-sop fast through the use of staging with Typed Template Haskell.

CCS Concepts: • **Software and its engineering → Functional languages**.

Keywords: generic programming, staging

ACM Reference Format:
Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. Staged Sums of Products. In *Proceedings of the 13th ACM SIGPLAN International Haskell Symposium (Haskell '20), August 27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3406088.3409021

1 Introduction

The generics-sop library [de Vries and Löh 2014] organises datatypes into a uniform and structured way: the choice of a constructor is represented as an n-ary sum type, and each choice contains an n-ary product representing the constructor arguments. As with other generics libraries, the representation is used to define functions that work on a large number of datatypes by exploiting the uniform structure. Unfortunately, like all its generic library siblings, the performance of generated code suffers unless measures are taken to minimize the abstraction overhead. This paper shows how we can remove the abstraction overhead using *staging*.

Consider a product type such as

```
data Foo = Foo [Int] Ordering Text
```


Haskell '20, August 27, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8050-8/20/08…$15.00
https://doi.org/10.1145/3406088.3409021

We can provide a Semigroup instance for such a type, relying on the existing Semigroup instances for its components. The semigroup operation for Foo can be defined as

```
sappend_Foo :: Foo → Foo → Foo
sappend_Foo (Foo is1 o1 t1) (Foo is2 o2 t2) =
  Foo (is1 ⋄ is2) (o1 ⋄ o2) (t1 ⋄ t2)
```

This is a typical generic programming pattern: we match on the sole constructor of a datatype, apply the semigroup append operation (⋄) pointwise to its components, and apply the constructor again. None of this is specific to Foo; it all works whenever we have a single-constructor datatype where all components have the necessary Semigroup instances.

Using generics-sop, we can therefore define

```
gsappend :: (IsProductType a xs, All Semigroup xs) ⇒ a → a → a
gsappend a1 a2 = productTypeTo
  (czipWith_NP (Proxy @Semigroup) (map_III (⋄))
    (productTypeFrom a1) (productTypeFrom a2))
```

which captures exactly the pattern described above. The constraints state that the type a must be a single-constructor datatype and all its components must be an instance of Semigroup. The functions productTypeFrom and productTypeTo match on and apply the sole constructor of the datatype, respectively. The function czipWith_NP zips together the components pointwise, using the (⋄) function.

In order to make a type such as Foo satisfy the constraints of the gsappend function, it must be an instance of the Generic class, i.e., it must be representable in the sum-of-products style of generics-sop. Assuming such an instance exists, we can then simply write

```
sappend'_Foo :: Foo → Foo → Foo
sappend'_Foo = gsappend
```

The function gsappend can be instantiated to any single-constructor datatype that is an instance of Generic. Defining functions generically makes code substantially more concise and reduces the potential for errors. Furthermore, operations expressed generically are more robust to change: for example, adding or removing a field from Foo does not require any change to the code of sappend'_Foo.

However, before we start using generic programming all over the place, we should ask: Is sappend'_Foo equally fast as sappend_Foo, or do we incur an overhead for using the generic machinery? A simple benchmark that uses both functions

to append a simple value reveals the unfortunate truth: the generic version is roughly six times slower!

This is because productTypeFrom and productTypeTo work by converting between the original type Foo and an isomorphic, yet different structural representation. Functions like $\text{czipWith}_{\text{NP}}$ then traverse this structure. GHC is apparently incapable of optimising away these transformations.

This paper explains how to reduce these overheads using a variant of generics-sop that employs *staging* (via *Typed* Template Haskell[1]). This allows gsappend to be rewritten as:

```
sgsappend :: (IsProductType a xs, All (Quoted Semigroup) xs) ⇒
  Code a → Code a → Code a
sgsappend c1 c2 =
  productTypeFrom c1 $ λa1 → productTypeFrom c2 $ λa2 →
    productTypeTo (czipWithNP (Proxy @(Quoted Semigroup))
      (mapCCC [[(⋄)]]) a1 a2)
```

The code now has traces of the different stages in both types and terms, as can be observed by the presence of Code, Quoted and quotations (written $[\![\cdot]\!]$). We will discuss these constructs in detail later, in Sections 3 and 4. The semantics now guarantee that the conversion to the structural representation will all happen during compilation, whereas the generated code will not mention the structural representation at all.

To instantiate this function to Foo, we write

```
sappend''Foo :: Foo → Foo → Foo
sappend''Foo foo1 foo2 = $$(sgsappend [[foo1]] [[foo2]])
```

We use a *top-level splice* (written $\$\$(\cdot)$) to trigger the generation of specialised sgsappend code for Foo at compile time.

The spliced version, $\text{sappend}''_{\text{Foo}}$, is now just as fast as the hand-written version $\text{sappend}_{\text{Foo}}$. In fact, GHC generates the same code for both versions of function, as can be verified using inspection-testing [Breitner 2018].

The most common approach to optimising generic programs in Haskell is to rely on GHC's simplifier and hope that extensive use of inlining will remove the intermediate representations. There are two disadvantages of this approach: first, it is fragile, as the simplifier employs heuristics and thresholds, and it difficult to predict whether it will be successful in eliminating all overhead in a particular use case; second, it is costly, as inlining is not very targeted and can easily lead to huge intermediate code or repeatedly performed work at compile time. In contrast, with staging we can semantically guarantee good code, and the compile-time cost is also predictable, making it a much more practical choice.

In the rest of this paper, we first summarise necessary ideas of generics-sop (Section 2) and Typed Template Haskell (Section 3). Then, the contributions of this paper are that:

- We introduce staged-sop, a staged variant of generics-sop, and show it is a great fit for staging. The result is a library of high-level combinators which can be used to create generic functions with the guarantee of eliminating the generic representation from the result. (Section 4)
- We show how to use recently proposed extensions to Typed Template Haskell that are needed to properly deal with class constraints [Pickering et al. 2020]. (Section 5)
- We demonstrate several examples of generic functions in staged style, and that they typically very close to the structure of their unstaged counterparts. (Section 6)
- We compare the performance of the unstaged and staged versions of generics-sop. (Section 7)

Finally, we also discuss related work (Section 8) before concluding (Section 9). The work in this paper is implemented as a library called staged-sop.[2] The required changes to Typed Template Haskell are implemented in a branch of GHC 8.11.

2 Programming with Sums of Products

This section describes the original generics-sop library by de Vries and Löh [2014], as available on Hackage.

2.1 Structurally Representable Types

The approach is based on a class Generic for structurally representable types, two datatypes NS and NP for n-ary sums and products, respectively, and a number of powerful combinators implementing operations on such sums and products.

The class Generic looks as follows:

```
class Generic a where
  type Description a :: [[Type]]
  from :: a → Rep a
  to   :: Rep a → a
```

It associates with every type a that is an instance a *description*[3] which is a type-level list of lists of types, and functions from and to that convert between a value of type a and its isomorphic *representation* of type Rep a.

The representation of a type is defined by interpreting the elements of the outer list in the description as choices in an n-ary sum NS (corresponding to the constructors of the datatype), and the elements of the inner lists in the description as the components of an n-ary product NP (corresponding to the fields of the constructors).

As an example, let us consider binary trees:

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

The description of binary trees is

[1] An extension of Template Haskell [Sheard and Peyton Jones 2002] that adds types in the style of MetaML [Taha and Sheard 2000]. It was implemented in GHC in 2013 by Geoffrey Mainland under a proposal by Simon Peyton Jones.

[2] https://github.com/well-typed/generics-sop/tree/staged-sop

[3] There is a naming conflict between the fields of datatype-generic programming and staging. A description of a type that is then interpreted is often called a *code*. Likewise, staged fragments of the syntax tree are often referred to as *code*. We keep Code to be a type that denotes a term to be generated for use in a future stage, and rename the generics-sop use of Code to Description in this paper.

```
type Description (Tree a) = '[ '[a], '[Tree a, Tree a] ]
```

The outer list has two elements, one per constructor. The first of the inner lists contains only a, for the argument of Leaf; the other inner list contains two occurrences of Tree a, for the two subtrees in Node.

The representation Rep is defined as follows:

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

newtype I a = I a
newtype SOP f xss = SOP (NS (NP f) xss)

data NS :: (a → Type) → [a] → Type where
  Z :: f x → NS f (x : xs)
  S :: NS f xs → NS f (x : xs)

data NP :: (a → Type) → [a] → Type where
  Nil  :: NP f '[]
  (:*) :: f x → NP f xs → NP f (x : xs)

infixr 5 :*
```

An NS f is a GADT denoting an n-ary sum (i.e., a choice), where every component is wrapped in an application of f, so NS f '$[x_1, \ldots, x_n]$ is equivalent to $f\,x_1 + \ldots + f\,x_n$.

An NP f is a GADT denoting an n-ary product (i.e., a sequence) where again every component is wrapped in an application of f, so NP f '$[x_1, \ldots, x_n]$ is equivalent to $f\,x_1 \times \ldots \times f\,x_n$.

An SOP f is a newtype-wrapper around a sum of products, and I is the identity functor. Thus Rep is just a sum of products. The functions from and to are the conversion functions witnessing the isomorphism. For binary trees, from is

```
from :: Tree a → Rep (Tree a)
from (Leaf a)       = SOP (Z (I a :* Nil))
from (Node t1 t2) = SOP (S (Z (I t1 :* I t2 :* Nil)))
```

and to is similar. It is worth noting that the translation is shallow: while Tree is a recursive type, the recursive occurrences appear in Rep (Tree a) in their original form. We only change the top-level layer of the type. This means that from and to are never recursive, and an instance for Generic (Tree a) can be defined without requiring an instance Generic a.

Users do not have to write instances of Generic by hand. Such instances can be derived automatically via (untyped) Template Haskell or via type-level programming in terms of the GHC-derivable GHC.Generics representation [Magalhães and Löh 2014].

2.2 Product Types

In the Semigroup-based example from Section 1, we have been limiting ourselves to product types, i.e., types with a single constructor. It is a strength of generics-sop that such a requirement can be easily captured as a constraint on the Description. We can define

```
type IsProductType a xs = (Generic a, Description a ~ '[xs])
```

to express that in order to be a product type, a has to be an instance of Generic, and its description must have a single entry xs, which grants us access to the types of the fields of the sole constructor.

It is also useful to define variants of the conversion functions from and to for product types that additionally perform the (un)wrapping of the constructor:

```
productTypeFrom :: IsProductType a xs ⇒ a → NP I xs
productTypeFrom a = case from a of SOP (Z xs) → xs

productTypeTo :: IsProductType a xs ⇒ NP I xs → a
productTypeTo xs = to (SOP (Z xs))
```

Note that IsProductType a xs ensures that Description a has a single element, so there is no way to create a valid sum via S, and Z is the only case.

2.3 Operations on Sums and Products

Most generic functions implemented using generics-sop do not have to make use of the constructors of NP and NS directly, but can instead be implemented in terms of various high-level combinators provided by the library. An example of such a combinator is $\mathrm{czipWith}_{\mathrm{NP}}$ that we used in gsappend in order to zip together two product types in a pointwise fashion. Let us first look at $\mathrm{zipWith}_{\mathrm{NP}}$, a slightly simpler variant, that can be defined on two NPs as follows:

```
zipWithNP :: (∀ x . f x → g x → h x) →
  NP f xs → NP g xs → NP h xs
zipWithNP _  Nil       Nil       = Nil
zipWithNP op (f :* fs) (g :* gs) = op f g :* zipWithNP op fs gs
```

We need a rank-2-polymorphic type here, since the operator op must work for any choice of x. This way, we can guarantee that it in particular works for all the elements of xs.

In many situations, it is useful to be able to make additional assumptions about the elements of x, such as that they are all instances of a class. This is what $\mathrm{czipWith}_{\mathrm{NP}}$ allows: it additionally takes a class argument. This class argument has to be explicitly instantiated, as GHC's type inference is otherwise not clever enough to infer it. These days, that could be done by using a plain explicit type application [Eisenberg et al. 2016], but generics-sop chooses to instead use a proxy argument, highlighting that explicit instantiation is required:

```
czipWithNP :: All c xs ⇒ proxy c →
  (∀ x . c x ⇒ f x → g x → h x) → NP f xs → NP g xs → NP h xs
czipWithNP _ _  Nil       Nil       = Nil
czipWithNP p op (f :* fs) (g :* gs) = op f g :* czipWithNP p op fs gs
```

The body of the function is nearly the same. However, we need to express as a constraint that all elements in xs satisfy the constraint c. This is the purpose of All, which can be defined as a type family:[4]

[4] In the actual library, All is defined as a mutually recursive type family and type class. This allows partial parameterisation of All to just the constraint, improves type inference, and allows attaching an induction principle for type-level lists to the class.

```
type family
  All (f :: a → Constraint) (xs :: [a]) :: Constraint where
  All f '[]       = ()
  All f (x : xs) = (f x, All f xs)
```

If we define a trivial constraint Top as

```
class Top a
instance Top a
```

then $\mathsf{czipWith_{NP}}$ (Proxy @Top) is almost the same as the unconstrained variant; it just has the extra All Top xs constraint.[5]

The $\mathsf{zipWith_{NP}}$ and $\mathsf{czipWith_{NP}}$ functions are just two of many functions that generics-sop provides. In general, NP is a higher-order applicative and traversable functor [Johann and Ghani 2007, 2009], and NS exposes a similarly rich structure.

We can now understand gsappend from Section 1:

```
gsappend :: (IsProductType a xs, All Semigroup xs) ⇒ a → a → a
gsappend a1 a2 = productTypeTo
  (czipWithNP (Proxy @Semigroup) (mapIII (⋄))
    (productTypeFrom a1) (productTypeFrom a2))
```

The arguments a_1 and a_2 are converted into products of type NP I xs via productTypeFrom. These products are zipped using $\mathsf{czipWith_{NP}}$, combining their elements with (⋄). The function

```
mapIII :: (a → b → c) → I a → I b → I c
mapIII op (I a) (I b) = I (op a b)
```

is a utility to help with the (un)wrapping of the I constructors.

3 Typed Template Haskell

In this section, we have a look at the typed fragment of Template Haskell, which is much less widely known than its untyped counterpart.

Despite the similarity in name, Typed Template Haskell is rather different to Template Haskell and concerns itself only with expressions. We cannot manipulate arbitrary syntax trees, and cannot write code that generates datatypes, classes, or other top-level declarations. We can, however, write typed code that is *staged*, i.e., that can in particular be used to generate code at compile time, and we can guarantee in advance that every type-correct instantiation of a code-generating function will produce type-correct code.

In this section, we present Typed Template Haskell essentially as it is implemented in GHC as of version 8.10. There are extensions necessary to the GHC implementation for all the work described in this paper to function, and in fact, the work on staging generics-sop has helped identify the current shortcomings. We describe these aspects later, in Section 5.

3.1 Quotations and Splices

The "Hello world" example of staging is a function that computes the n-th power of an integer:

```
power :: Int → Int → Int
power n k | n ⩽ 0     = 1
          | otherwise = k ∗ power (n − 1) k
```

If n is statically known, such as n = 5, we might wish to unroll the loop to produce k ∗ k ∗ k ∗ k ∗ k ∗ 1.[6]

We can do so by writing a staged version of power:

```
spower :: Int → Code Int → Code Int
spower n k | n ⩽ 0     = ⟦1⟧
           | otherwise = ⟦$$(k) ∗ $$(spower (n − 1) k)⟧
```

A Code Int is a fragment of code of type Int that we are constructing.[7]

If we have a Haskell expression e of type a, then ⟦e⟧ denotes the syntax tree of e, which has type Code a.

We see such quotations on the right hand sides of the two cases of the spower function. For example, if n ⩽ 0, we construct the code that is the literal 1. In the other case, we want to build a larger code fragment out of already existing code fragments. We can do so by *splicing* a value x of type Code a into a quotation by writing $$(x), where it then behaves as an expression of type a again. Here, we build the multiplication of the (statically unknown) argument k with the code resulting from the recursive invocation of spower.

3.2 Top-Level Splices

If we want to actually integrate a piece of code into a program, we use a special form of splice that occurs outside of a quotation. This is called a top-level splice and causes the code to be generated at compilation time and inserted at that point in the program.

For example, here the first argument of spower is 5:

```
power5 :: Int → Int
power5 k = $$(spower 5 ⟦k⟧)
```

Once compiled, this produces code equivalent to

```
power5 :: Int → Int
power5 k = k ∗ (k ∗ (k ∗ (k ∗ 1))))
```

This code is still subject to potential further optimisation of the compiler, as if it had been written by hand.

There is nothing else we can do with Code values but to splice them. In general we cannot know at compilation time what the value denoted by a piece of code is, so a Code Int cannot be treated as an Int. If we want to do something depending on the Int denoted by a Code Int, we have to generate a program that performs a case distinction on the unknown value.

[5] The library defines SListI as a synonym for All Top, but we consistently use All in this paper.

[6] Many variants of this example go further by trying to minimise the number of multiplications needed. This does not add to the explanation of the staging concepts, so we use the simpler example.

[7] In Template Haskell, Code currently has to be defined as a type synonym

```
type Code a = Q (TExp a)
```

3.3 Stages

The presence of quotations and splices segregates expressions into different stages, where quotations increase the stage counter by one, and splices decrease it by one.

Conceptually, stages are running at different times, and not all information may be present at any stage. However, the top-level of the program is special in that it is part of the program at any stage. This means that a top-level binding can be used in a quotation without restrictions. A definition being available at all stages is often called *cross-stage persistence*.

For bindings made at other places, there are more restrictions in place, and a program may be *stage-incorrect* despite it being type-correct. As an example, consider

```
⟦λx → $$(x)⟧
```

We could give this fragment the type `Code (Code a → a)`, but what should it denote? We have to produce code for a function that itself takes a code fragment and then contains that code fragment as its body. But we do not know the argument of the function at the point we have to construct its code. This is because `x` is bound at a later stage than it is used. This is generally forbidden and produces a *stage error*.

Let us discuss the situation when using a variable that is bound at an earlier stage, as in

```
λx → ⟦x⟧
```

We can look at this as a function of type `a → Code a` that can be applied to `x` at compile time and then has to produce code representing `x`. How difficult it is to generate an abstract syntax tree corresponding to a value of an arbitrary type depends very much on the implementation. The situation is reminiscent of serialising a value. We need to turn the value into some form of reusable representation. Just like serialisation, Haskell makes the decision that this is not possible for all types, but governed by a type class called `Lift`:

```
class Lift a where
  liftTyped :: a → Code a
```

The task of `liftTyped` is to map a value to code representing that value. If we have a concrete type, we can usually provide an implementation relatively easily. For example, consider binary trees as in Section 2:

```
instance Lift a ⇒ Lift (Tree a) where
  liftTyped (Leaf a)   = ⟦Leaf $$(liftTyped a)⟧
  liftTyped (Node l r) = ⟦Node $$(liftTyped l) $$(liftTyped r)⟧
```

Given the systematic nature of the code for `liftTyped`, it is perhaps not surprising that the `Lift` type class can itself be generically defined (similar to `Show` for standard human-readable serialisation), and GHC already has functionality to derive instances via the `DeriveLift` language extension. Just as with the `Show` type class, there are limits: functions as well as abstract types such as `IO` have no `Lift` instances.

Using `Lift`, staged code that uses variables at a later stage than their binding occurrence is implicitly rewritten to use `liftTyped`. For example, `λx → ⟦x⟧` becomes `λx → ⟦$$(liftTyped x)⟧` where the use of `x` is at the same stage as its binding due to the splice and quotation cancelling each other out.

3.4 Staging and Constraints

The `spower` example used quotations and splices on values of monorphic types. We have not yet discussed interactions between staging and class constraints. As we will see, that is not entirely trivial, but we will defer the discussion until Section 5 when we have seen examples of such cases naturally arising in the context of staging generics-sop.

4 Staging Generic Programs

Let us now see how we can combine staging as discussed in Section 3 with generics-sop as described in Section 2. For this, we will revisit the example of the staged semigroup append operation from Section 1.

4.1 Unstaged Starting Point

In the standard example of `spower`, the staged variant is just an annotated version of the unstaged function `power` – dropping all the staging annotations and identifying the types `Code a` and `a` yields back the original program.

It is always true that this transformation of dropping the staging annotations should recover a valid unstaged program [Inoue and Taha 2016]. It is not always the case that we can just drop staging annotations in the original code to end up with a satisfactory staged version. However, the unstaged function is almost always a good starting point. So let us start by looking at `gsappend` from Section 1 again:

```
gsappend :: (IsProductType a xs, All Semigroup xs) ⇒ a → a → a
gsappend a₁ a₂ = productTypeTo
  (czipWith_NP (Proxy @Semigroup) (map_III (⋄))
    (productTypeFrom a₁) (productTypeFrom a₂))
```

Note that the type of this function, modulo constraints, is `a → a → a`. In particular, there is no trace on the outside of the fact that we are using the `NS` and `NP` types internally.

A simple guiding principle for the design of the staged version of generics-sop is that the structural representation should only occur at compile time, and we will ensure this by staging as follows:

- In the compile time stage, the generic functions can be instantiated to a particular type and will then generate code specific to that type which will be executed in a runtime stage.
- In the code fragments we generate (by means of quotation), we will *never* mention the structural representation, but only the original types we operate on.

While this simple principle in itself does not guarantee optimal code, it still goes a long way to make sure that we do not incur overhead from the transformations between a type

and its structural representation. Also, the principle has the advantage of being easy to check.

4.2 Conversion Functions

A key issue in the design is how we change the conversion functions from and to, as well as their product-type variants.

At first glance, the from conversion already seems to leave us with an unsurpassable problem – expanding the Rep type synonym, we have:

```
from :: Generic a ⇒ a → SOP I (Description a)
```

Clearly, we cannot impose the argument a to be known statically; otherwise generic functions could only ever operate on data that is known at compilation time. But once we have

```
from :: Generic a ⇒ Code a → ...
```

it is unclear how to continue. According to our guiding principle, we do not want to embed the SOP type in Code, ever. But the choice of constructor is unknown until runtime, so how would we be able to recover that statically?

Indeed, we have to deviate from the original type here somewhat. The choice of constructor is indeed unknown, but the number of constructors is statically known (because the type information is static), and if we are prepared to deal with any choice of constructor, we can perform a dynamic pattern match, and subsequently continue assuming we statically know the choice of constructor.

So the idea is to have from generate code for a pattern match and to have continuations available for each constructor which pretend that the constructor is statically known. Performing a pattern match in order to exploit static information discovered in each branch is a common technique in partial evaluation [Jones et al. 1993]. For us, the combination with an advanced type system also means that the additional information is visible in the types of the continuations.

In Section 2, we have given a definition of from for binary trees as follows:

```
fromTree :: Tree a → SOP I (Description (Tree a))
fromTree (Leaf a)      = SOP (Z (I a :* Nil))
fromTree (Node t1 t2) = SOP (S (Z (I t1 :* I t2 :* Nil)))
```

Let us now rewrite this definition in continuation passing style, also making the pattern match explicit by using **case**, to prepare for staging:

```
from'Tree :: Tree a → (SOP I (Description (Tree a)) → r) → r
from'Tree t k =
  case t of
    Leaf a      → k (SOP (Z (I a :* Nil)))
    Node t1 t2 → k (SOP (S (Z (I t1 :* I t2 :* Nil))))
```

Note how within the branches of the **case**, on the right hand side, the entire sum of products structure of the value is now statically known locally, and only the arguments to the constructors are still dynamic information.

Now, when staging from'Tree, we benefit from the fact that our sum and product types are already parameterised over a type constructor. Where we used I before, we can now use a **newtype**-wrapped form of Code:

```
newtype C a = C (Code a)

sfromTree :: LiftT r ⇒ Code (Tree a) →
  (SOP C (Description (Tree a)) → Code r) → Code r
sfromTree t k =
  [|| case $$(t) of
    Leaf a      → $$(k (SOP (Z (C [|a|] :* Nil))))
    Node t1 t2 → $$(k (SOP (S (Z (C [|t1|] :* C [|t2|] :* Nil))))) ||]
```

The role of the LiftT constraint will be discussed in Section 5, so let us focus on the implementation for now: according to the plan, sfrom generates code for a pattern match on the original type. Every branch has locally revealed the sum-of-products structure of the argument, and can apply a continuation that has that information available statically.

The conversion in the other direction is simpler. Given an SOP C (Description a), we do not have to reveal more structure and can recover Code a by applying the original constructors:

```
stoTree :: SOP C (Description (Tree a)) → Code (Tree a)
stoTree (SOP (Z (C a :* Nil)))                = [|Leaf $$(a)|]
stoTree (SOP (S (Z (C t1 :* C t2 :* Nil)))) = [|Node $$(t1) $$(t2)|]
```

This motivates us to redefine the Generic class for staged generic functions as follows:[8]

```
class Generic a where
  type Description a :: [[Type]]
  from :: LiftT r ⇒ Code a → (Rep a → Code r) → Code r
  to   :: Rep a → Code a

type Rep a = SOP C (Description a)
```

Perhaps surprisingly, this already concludes the hard part of our work. It turns out to be a blessing that our core infrastructure, the NP, NS and SOP types, are all parameterised over a type constructor, and all we have done is to change that type constructor in the representation from I to C.

The operations the library provides on these types, however, are still available, and are able to cope with the slightly changed representations just as well. One remaining issue we still have to discuss, however, is how exactly staging interacts with polymorphism and overloading – several of the combinators make use of abstraction over constraints, such as czipWithNP in gsappend, and it is crucial that these can be used in quotations.

5 Typed Template Haskell and Constraints

This section discusses the interaction of constraints, polymorphism, and staging in detail. As a starting point, let us focus on a fragment that occurs in the staged function sgsappend in

[8] We re-use the old names from now on, e.g., we say from for the staged version rather than sfrom.

Section 1, namely the invocation of the semigroup operation in a quotation, ⟦(⋄)⟧. What is the type of this expression?

5.1 Staging in GHC 8.10

Given that the type of (⋄) is

```
(⋄) :: Semigroup a ⇒ a → a → a
```

one might consider the following two candidate types:

```
⟦(⋄)⟧ :: Code (Semigroup a ⇒ a → a → a)
⟦(⋄)⟧ :: Semigroup a ⇒ Code (a → a → a)
```

The first of these seems innocent. However, it is an *impredicative type*: it requires instantiating the type argument of Code to a qualified type, and GHC does not allow this.

The second is the type that GHC 8.10 infers for this code. However, this turns out to be problematic. In order to understand why, we have to consider the implementation of class constraints in GHC. Classes are translated into *dictionaries*. A class definition can be seen as defining an internal record type, with the class methods being the components. Each instance definition for the class defines a value of this record type, or, if the instance has preconditions, it defines a function taking other dictionaries and turning them into a dictionary for the class in question.

Every overloaded function is then passed a dictionary value at run-time, and invoking an overloaded function is translated into selecting a component from such a dictionary.

But if class constraints are translated into abstracting from and applying to dictionary arguments, then it means these dictionaries should be subject to the stage restrictions previously discussed in Section 3.3!

Trying to make the dictionary passing explicit, we arrive at the following ill-staged code

```
wrong :: Dict (Semigroup a) → Code (a → a → a)
wrong dict = ⟦(⋄) dict⟧
```

Here, dict is used at a later stage than it is bound. As discussed in Section 3.3, this situation is not completely hopeless. If we could lift the dictionary, we would be fine. However, dictionaries are nearly always comprised of functions, and functions cannot be lifted in GHC.

The actual GHC implementation in 8.10 is internally untyped, so it will just generate the code for (⋄) and then retypecheck it at the splice site, reinferring the dictionary. But this can lead to various unintended effects because the instances available at the splice site can be very different from the quotation site, as described by Pickering et al. [2019].

So using a constrained method inside a quotation can have unintended consequences, but at least it works in most situations. On the other hand, generating constrained polymorphic programs is currently impossible in GHC 8.10. Considering the running example, if we instantiate a to Maybe b, then we would want to splice this definition at the top-level to create a polymorphic function:

```
fails :: Semigroup b ⇒ Maybe b → Maybe b → Maybe b
fails = $$(⟦(⋄)⟧)
```

This program is rejected by GHC because the ⟦(⋄)⟧ function has a Semigroup b constraint that it cannot solve. It is *not* solved by the local Semigroup b constraint because using the evidence from the local constraint would result in a variable being used at a level before it was bound:

```
fails :: Dict (Semigroup b) → Maybe b → Maybe b → Maybe
fails dict = $$(⟦(⋄)⟧ dict)
```

Therefore, as a sound but conservative restriction, all local contexts are ignored when type checking a top-level splice. This avoids the situation where a variable is used before it is bound, but unfortunately precludes the generation of any programs with constraints.

The ability to be precise about which stage a dictionary is bound is necessary to relax this restriction and permit the sound generation of constrained programs.

5.2 Quoting and Splicing Dictionaries

When the dictionaries are explicit as in wrong, the solution to the problem with staging is obvious: dictionaries must also be allowed to be quoted and spliced. Instead of wrong, we can instead write

```
correct :: Code (Dict (Semigroup a)) → Code (a → a → a)
correct cdict = ⟦(⋄) $$(cdict)⟧
```

which is stage-correct as well as type-correct.

However, it now means we have to be able to talk about the *code* of dictionaries, or, translated back to the standard implicit setting, about the code of constraints.

Therefore, we introduce a constraint form

```
CodeC :: Constraint → Constraint
```

While Semigroup a denotes that we must have a Semigroup a dictionary now, the constraint CodeC (Semigroup a) denotes that we must have a Semigroup a constraint at the next stage.

For our initial example, we then obtain the type

```
⟦(⋄)⟧ :: CodeC (Semigroup a) ⇒ Code (a → a → a)
```

from which we can clearly see that the Semigroup constraint is required within a quotation.

The CodeC constraint is formally specified and explained in much more detail by Pickering et al. [2020].

5.3 Lifting Type Variables

Now that we have talked about constraints, let us consider that GHC uses a core language that is explicitly typed, and makes type abstractions and type applications explicit by passing types.

So whenever we use polymorphic code in quotations, we get into similar issues with types as we have just discussed for constraints. All named types are defined at the top-level and cross-stage persistent, but type variables are not.

Therefore, we introduce another constraint form

```
LiftT :: k → Constraint
```

which is kind-polymorphic and arises whenever the translation of a quotation to the core language mentions a type variable. The payload of the LiftT class is a type representation for a specific type and the logic to solve it is built into the compiler. Similar to the Typeable [Peyton Jones et al. 2016] class, the purpose of the constraint is to turn an irrelevant type argument into a relevant argument which can be manipulated. In our case, the representation is only suitable to be inserted into a quotation by splicing and cannot be inspected or manipulated further by users.

Our running example is polymorphic at type variable a, and therefore the final type signature for this example is

```
⟦(⋄)⟧ :: (CodeC (Semigroup a), LiftT a) ⇒ Code (a → a → a)
```

The extension of GHC with LiftT and CodeC constraints is useful independently of the application to generic programming, but particularly crucial for our application, because we want to splice polymorphic and overloaded functions.

The theoretical details of these extensions are beyond the scope of this paper and described elsewhere [Pickering et al. 2020, 2019]. Note that while CodeC constraints are fundamental, LiftT constraints are an artifact of the implementation and can perhaps be removed in the future.

5.4 What If We Had Impredicativity?

In the beginning of this Section 5, we briefly considered the impredicative type

```
⟦(⋄)⟧ :: Code (Semigroup a ⇒ a → a → a)
```

GHC is finally about to get a type system extension that allows this [Serrano et al. 2020]. For our purposes, having such types available is certainly useful. However, it cannot replace the need for CodeC, which has the advantage of being a first-class constraint form. It can appear in an argument position to another type constructor, and be used in class contexts or any place where we abstract over a constraint. We will see some examples of this in Section 6.

6 Examples of Staged Generic Functions

We now have a library for writing staged generic functions at our disposal: we have the modified Generic class and conversion functions introduced in Section 4, and we understand the issues concerning class constraints that may arise. All we have to do now is to use it.

6.1 Staged Semigroup Append

Let us first revisit the example from the introduction once again, and see how the staged version works:

```
sgsappend :: (IsProductType a xs, All (Quoted Semigroup) xs) ⇒
  Code a → Code a → Code a
sgsappend c1 c2 =
  productTypeFrom c1 $ λa1 → productTypeFrom c2 $ λa2 →
    productTypeTo (czipWithNP (Proxy @(Quoted Semigroup))
      (mapCCC ⟦(⋄)⟧) a1 a2)
```

We are using the staged version of productTypeFrom to pattern match on the sole constructor, and a_1 and a_2 are then of type NP C xs, products containing code values. We use the original $\text{czipWith}_{\text{NP}}$ function to zip together these products.

The new helper function

```
mapCCC :: Code (a → b → c) → C a → C b → C c
mapCCC op (C a) (C b) = C ⟦($$(op) $$(a) $$(b))⟧
```

makes it easier to apply functions to code values without having to deal with the newtype constructors. The function we are applying is ⟦(⋄)⟧, which we have discussed in Section 5, and which we now know in our system to be of type

```
⟦(⋄)⟧ :: (CodeC (Semigroup a), LiftT a) ⇒ Code (a → a → a)
```

We thus need both CodeC (Semigroup a) and LiftT a to hold for all types a that are in the type-level list xs.

Since the combination of CodeC and LiftT is very common in the staged generic programming context we are in, we introduce an abbreviation:

```
class (CodeC (c a), LiftT a) ⇒ Quoted c a
instance (CodeC (c a), LiftT a) ⇒ Quoted c a
```

Consequently, we use Quoted Semigroup to instantiate the $\text{czipWith}_{\text{NP}}$ function, and therefore the constraint also appears in the type signature of the function.

6.2 Instantiating and Observing the Generated Code

If we want to instantiate sgsappend at a concrete type such as Foo, we use a top-level splice:

```
sappend''Foo :: Foo → Foo → Foo
sappend''Foo foo1 foo2 = $$(sgsappend ⟦foo1⟧ ⟦foo2⟧)
```

The constraints arising from sgsappend are easily discharged: the All (Quoted Semigroup) xs constraint from the quotation means the Semigroup constraints for the component types of Foo at the splice site must be satisfied – and they are. And LiftT constraints are *always* satisfiable when splicing.

We can instruct GHC to show us the generated code by passing the `-ddump-splices` or `-ddump-simpl` flags. In our extended version, GHC always shows generated code in the Core language, but abstracting from that, the code shown is equivalent to

```
case foo1 of {Foo x1 x2 x3 →
  case foo2 of {Foo y1 y2 y3 →
    ((Foo ((⋄) x1 y1)) ((⋄) x2 y2)) ((⋄) x3 y3)}}
```

This is exactly what we would expect and hope for. All the conversion to structural types happens at compile time; the only code we construct are the applications of (⋄) to their arguments (explicitly via the map_{CCC} call), the application of Foo (via productTypeTo) and the pattern matches on foo_1 and foo_2 (via productTypeFrom).

A hand-written version of semigroup append for Foo would result in the same code, as can be observed by writing it manually and applying inspection-testing to compare the generated Core code automatically.

Reflecting on this, it should be obvious from the definition we have given, because as we promised when we set out to stage generic functions in Section 4.1, we can easily see that no traces of the intermediate representation exist at runtime. All library functions adhere to this guideline, and the only user-supplied quotation we use creates applications of ($\diamond$).

6.3 Equality

Next to the Semigroup example we have already seen, we can easily extend staging to Monoid or apply it to other standard examples for generic programming, such as NFData (to completely force a value), or GHC's stock derivable classes, for example Bounded, Enum, Eq or Ord.

As one representative, let us look at equality. The unstaged variant is simple to write, shifting the bulk of the work to $\text{ccompare}_{\text{SOP}}$, another of the various utility functions provided by generics-sop:

```
geq :: (Generic a, All (All Eq) (Description a)) ⇒ a → a → Bool
geq a1 a2 = ccompareSOP (Proxy @Eq)
  False
  (λxs1 xs2 → and (collapseNP
    (czipWithNP (Proxy @Eq) (mapIIK (==)) xs1 xs2)))
  False (from a1) (from a2)
```

If the constructors do not match in either direction, we return False; otherwise, the constructor arguments are type-compatible and can be compared pointwise using (==).

It is easy enough to come up with a staged variant based on the above, following exactly the same strategy the we have seen so far:

```
sgeq :: (Generic a, All (All (Quoted Eq)) (Description a)) ⇒
  Code a → Code a → Code Bool
sgeq c1 c2 = from c1 $ λa1 → from c2 $ λa2 →
  ccompareSOP (Proxy @(Quoted Eq))
    ⟦False⟧
    (λxs1 xs2 → sand (collapseNP
      (czipWithNP (Proxy @(Quoted Eq))
        (mapCCK ⟦(==)⟧) xs1 xs2)))
    ⟦False⟧ a1 a2

sand :: [Code Bool] → Code Bool
sand = foldr (λx r → ⟦$$(x) && $$(r)⟧) ⟦True⟧
```

We see the expected changes: switching to the continuation-passing style for the from functions, parameterising over Quoted Eq rather than Eq, and quoting False and (==).

At the point where we apply and in the original function, we are operating on a static list of Code Bool values, which are processed using a staged version of and that inserts the code for && between the elements. Strictly speaking, this will not exactly generate the code we would have written by hand, because we would not add an additional conjunction with True when reaching the end of the list except if the list is empty to start with. This could easily be fixed by using a slightly different traversal than foldr or the more advanced technique of *partially-static* datatype [Yallop et al. 2018], but it is also quite clearly in the realm of GHC's optimising capabilities, so in this case, it is not worth worrying about.

A more substantial problem is that the code size of the generated code is quadratic. The nested use of from on the two arguments creates a nested **case** that has cases for all possible combinations of constructors. On the other hand, a manually written equality function can do with a linear number of cases, for example:

```
eqTree :: (Eq a, Eq (Tree a)) ⇒ Tree a → Tree a → Bool
eqTree (Leaf a1)     (Leaf a2)     = a1 == a2
eqTree (Node l1 r1) (Node l2 r2) = l1 == l2 && r1 == r2
eqTree _             _             = False
```

In practice, GHC's optimiser will also do a good job here in collapsing the unnecessary cases again, but we still have quadratic code size at an intermediate stage, which might become worrying for large datatypes.

A proper solution for this is possible, but we leave the details to future work and only sketch it here: a weakness of our current library is that we cannot ever generate partial pattern matches on the source type, because our only function for matching is from, and it will always consider all cases. Since Typed Template Haskell is still based on normal (untyped) Template Haskell, we can use that to implement new typed "primitives", including one that would allow us to select which cases we want to generate (and would need a default branch in case none of the selected cases match). We could use this to define a generalised version of from that could take its local knowledge into account to decide on which of the constructors of the source datatype it wants to match. Such a generalised from could then be used to improve staged equality further.

6.4 Metadata and Customisation

Another common application area of generic programming are serialisation and deserialisation functions, among them classes such as Read and Show, that use *metadata* from the datatype, such as constructor or record field names, in order to come up with a human-readable representation.

The generics-sop library makes metadata available via a separate type class HasDatatypeInfo and exposes the metadata using sums and products, thereby ensuring that its shape matches that of the datatype itself. This just transfers to the staged setting without any changes.

A nice side effect of this approach is that we can apply non-standard customisation data to a generic function using sums and products, and can still guaranteee that they have the correct shape. Here is an example of a simple show-like function that only works on enumeration types (types where

none of the constructors have any arguments), and uses user-supplied strings instead of the normal constructor names:

```
gshowEnum ::
  IsEnumType a ⇒ NP (K String) (Description a) → a → String
gshowEnum names a =
  collapse_NS (selectWith_NS const names (enumTypeFrom a))
```

Using enumTypeFrom, we convert the original value into a sum. We use this sum to select one of the provided String values from the product of names – the types ensure that the product has the correct number of components. The function selectWith$_{NS}$ performs this selection and hands us back another sum, with the payload being just the name (because the combining function is const). Finally, collapse$_{NS}$ extracts the payload from the sum.

However, both standard and user-supplied metadata often have it in common that they are actually statically known. Nevertheless, in gshowEnum, we traverse this data at runtime, so we are not just paying the overhead of the generic representation, but additionally the overhead of having to traverse statically known data dynamically.

This phenomenon is common in the wild for various generic programming approaches. For example, the aeson package has a genericToEncoding function that takes a record type Options as an argument, which in most cases will contain statically known information.

One mitigation strategy that is sometimes employed is to supply such customisation data at the type-level.[9] In the case of gshowEnum, this would mean parameterising the function with a type-level list, such as

```
gshowEnumT ::
  IsEnumType a ⇒ Proxy (symbols :: [Symbol]) → a → String
```

However, we now have to use type-level constructs to propagate the type-level symbols to the right place, and cannot use our existing powerful term-level combinators. Supplying more complicated information at the type level, and possibly even performing additional computation on that information statically, quickly becomes very difficult [Kiss et al. 2018]. The term-level language in Haskell is still much more powerful and versatile than the type-level language, and staging turns out to be a much better solution for making the distinction between static and dynamic information explicit.

In a manually written instantiation of gshowEnum for the Ordering type, where for example we want to use symbolic names for the constructors, we would do a direct mapping:

```
showEnum_Ordering :: Ordering → String
showEnum_Ordering LT = "<"
showEnum_Ordering EQ = "="
showEnum_Ordering GT = ">"
```

[9] In fact, both GHC.Generics and generics-sop make the metadata additionally available also on the type-level for this reason.

We would like to get the same result with the staged generic version. When staging, we can easily indicate that the product is supposed to be available at compile-time:

```
sgshowEnum :: IsEnumType a ⇒
  NP (K String) (Description a) → Code a → Code String
sgshowEnum names c = enumTypeFrom c $ λa →
  liftTyped (collapse_NS (selectWith_NS const names a))
```

The argument names is known statically as it is a value of type NP rather than a value of type Code. Therefore we can use it at compile-time in order to generate the program which will produce a String. Apart from the pattern match introduced by enumTypeFrom, the only code generated by this function is by means of liftTyped, which creates a literal String value for the selected name on the right-hand side of the **case**. Therefore, we can easily match the hand-written version.

If we instantiate this function via

```
showEnum'_Ordering :: Ordering → String
showEnum'_Ordering o =
  $$(sgshowEnum (K "<" :* K "=" :* K ">" :* Nil) [|o|])
```

we get code equivalent to showEnum$_{Ordering}$. We still maintain the invariant that neither sums nor products appear in generated code. And if we wanted to use the standard constructor names after all, we could easily do that by constructing the argument product using the metadata functions from generics-sop.

6.5 Parsing

Let us consider a final example from the sqlite-simple package (a straight-forward database binding) in which there is a FromRow class that governs the marshalling from a row of database query results to a Haskell record type.

```
class FromRow a where
  fromRow :: RowParser a
```

All we need to know about RowParser is that it is an instance of Applicative, and that there is a function

```
field :: FromField a ⇒ RowParser a
```

that we can use to construct a RowParser for the component of a datatype as long as it is an instance of FromField.

For a concrete datatype such as

```
data Person = Person {personId :: Int, name :: Text, date :: Day}
```

we can then define a FromRow instance manually as

```
instance FromRow Person where
  fromRow = Person <$> field <*> field <*> field
```

by calling field on each of the components, and applying the constructor in an applicative context.

This is easy to convert into a generic definition:

```
gfromRow :: (IsProductType a xs, All FromField xs) ⇒
  RowParser a
```

```
gfromRow = productTypeTo
  <$> sequenceNP (cpureNP (Proxy @FromField) field)
```

The difference to the other generic functions we have discussed is that we are operating in an applicative context. The call to cpure_{NP} constructs a product with field invocations at every position. The result type of cpure_{NP} is of type NP RowParser xs. Since RowParser is applicative, we can combine the effects of the product components in sequence and extract the context, by applying

```
sequenceNP :: Applicative f ⇒ NP f xs → f (NP I xs)
```

As a final step, we can then map the conversion function productTypeTo over the resulting RowParser.

Unfortunately, the presence of the RowParser context affects staging. We can still call cpure_{NP}, and create code for the invocations of field for the components:

```
cpureNP (Proxy @(Quoted FromField)) (Comp (C [|field|]))
  :: NP (C :.: RowParser) xs
```

The expression [|field|] is of type

```
Quoted FromField x ⇒ Code (RowParser x)
```

To make this fit the structure of NP, where a single type constructor is applied to each x, we use functor composition:

```
newtype (f :.: g) x = Comp (f (g x))
```

The next step is a challenge. We cannot apply $\text{sequence}_{\text{NP}}$, because Code and C are not applicative, and neither is C :.: RowParser. Furthermore, we cannot expect to draw RowParser out of the NP statically, because the wiring of the parser is part of the generated code. Neither can we extract C out of the NP, because then we would have Code producing an NP, and we promised never to have NPs at run-time.

Therefore, instead of trying to stage $\text{sequence}_{\text{NP}}$, directly, we will look at the common combination of $\text{sequence}_{\text{NP}}$ followed by a mapping of productTypeTo – which can be seen as performing the transformation from the structural representation to a user type in an applicative context – and try to produce a staged version of that:

```
sproductTypeToA :: (IsProductType a xs, Quoted Applicative f) ⇒
  NP (C :.: f) xs → Code (f a)
```

Assuming this function is part of our staged generics library, we can complete the staged version of gfromRow:

```
sgfromRow :: (IsProductType a xs, All (Quoted FromField) xs) ⇒
  Code (RowParser a)
sgfromRow = sproductTypeToA
  (cpureNP (Proxy @(Quoted FromField)) (Comp (C [|field|])))
```

Now we can focus on defining sproductTypeToA. What kind of code do we have to generate? We start from a product

$$\text{Comp}\ (\text{C}\ \text{fx}_1) :* \ldots :* \text{Comp}\ (\text{C}\ \text{fx}_n) :* \text{Nil}$$

where each fx_i is code for an f-structure. We have to construct

$$[|\text{pure Con} <*> \$\$(\text{fx}_1) <*> \ldots <*> \$\$(\text{fx}_n)|]$$

where Con is the constructor function of our original Haskell datatype. Let us try to obtain this constructor generically. We have productTypeTo which applies the contructor, but it expects its argument as a product. We have to curry that function, turning the product into a chain of function arguments. To this end, we define a type family Curry that computes a curried function type from an NP:

```
type family Curry r xs where
  Curry r '[]       = r
  Curry r (x : xs) = x → Curry r xs
```

Given Curry, we define a staged currying function that takes a function expecting an NP C (just as productTypeTo does) and produces code for the curried function:

```
scurryNP :: ∀ r xs . (All LiftT xs, AllTails (LiftTCurry r) xs) ⇒
  (NP C xs → Code r) → Code (Curry r xs)
scurryNP f =
  case sList :: SList xs of
    SNil  → f Nil
    SCons → [|λx → $$(scurryNP (λxs → f (C [|x|] :* xs)))|]
```

The function sList from generics-sop gives us a way to pattern-match on the type-level list xs by means of a singleton type SList. An unfortunate aspect of $\text{scurry}_{\text{NP}}$ are the constraints we require. Not only do we need to know that all elements of xs are in LiftT, we also have to know that for all suffixes of xs, the constraint LiftT (Curry r xs) is satisfied. While intuitively, it is clear this follows from All LiftT xs and LiftT r, GHC cannot perform this reasoning. Therefore we encode the constraint via additional type classes and families AllTails and LiftTCurry. It is even more unfortunate that this AllTails constraint now propagates to all functions using $\text{scurry}_{\text{NP}}$ directly or indirectly. For the time being, we have decided to hide it in the IsProductType constraint, but having to add a constraint like this for a specific function is a flaw we would like to improve on in the future.

We can now complete sproductTypeToA by performing a foldl-like traversal over the NP, using the curried constructor function as the initial value of the accumulator:

```
sproductTypeToA ::
  ∀ a f xs . (IsProductType a xs, CodeC (Applicative f)) ⇒
  NP (C :.: f) xs → Code (f a)
sproductTypeToA = go [|pure $$(scurryNP (productTypeTo @a))|]
  where
    go :: ∀ ys . Code (f (Curry a ys)) →
      NP (C :.: f) ys → Code (f a)
    go acc Nil                  = acc
    go acc (Comp (C fx) :* fxs) = go [|$$(acc) <*> $$(fx)|] fxs
```

The strategy that we just applied to obtain a staged conversion function for product types that operates in an applicative context can be generalised to general sum-of-product types, to obtain a function

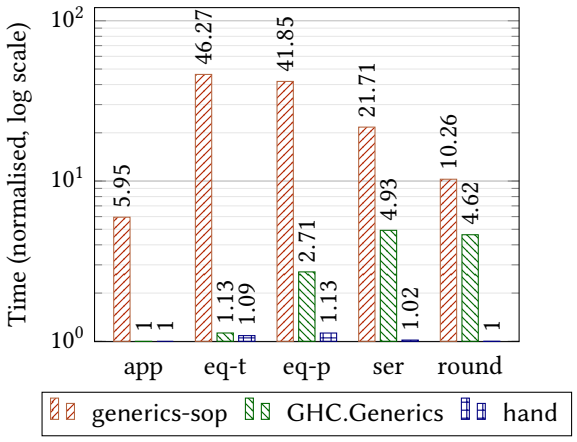

Figure 1. Benchmarks, time relative to staged-sop.

```
stoA :: (Generic a, Quoted Applicative f) ⇒
  SOP (C :.: f) (Description a) → Code (f a)
```

so that we can generally stage generic functions that require an applicative context.

7 Evaluation

In Figure 1, we summarise the results of five benchmarks[10] comparing our library staged-sop relative to generics-sop, an implementation using GHC.Generics and a hand-written version of the function. The benchmarks implement the generic append function (app), equality on trees (eq-t) and propositional formulae (eq-p), serialisation (ser) and serialisation followed by deserialisation (round).

Benchmarking the library is in some ways a vacuous exercise as with the use of inspection testing we are already confident in the optimality of our definitions. However, it is still interesting to see that for particular examples GHC.Generics is comparable to hand-written code, but for other bigger examples it reaches the limits of the optimiser. This confirms our hypothesis that the optimiser is not to be trusted and using a mechanism which guarantees the generic overhead is removed has been worthwhile.

One reason the original generics-sop performs so unfavourably is that most functions are written as compositions of recursive functions, and are therefore less amenable to inlining, whereas e.g. in GHC.Generics, generic functions are typically implemented via a single type class.

A final note is that we did not go to any special effort to write code generators which would produce optimal code. The generators are of the high-level variety described in this paper, without any further performance tweaks or modifications to standard GHC flags. The well-defined semantics of staging is enough to produce a program in the correct form which only requires the optimiser to do a small amount of work to clean up beta-reductions or nested case expressions.

[10]Conducted using gauge, a fork of criterion with fewer dependencies.

8 Related Work

The problem of making generic programming libraries more efficient is well known, and comparisons conducted between them [Hinze et al. 2007; Rodriguez et al. 2008] showed that performance degradation could be severe as implementations moved away from compiler preprocessors to libraries.

Both the generic-deriving library [Magalhães 2013] and generic-lens library [Kiss et al. 2018] are made more efficient through inlining. Inlining [Peyton Jones and Marlow 2002] works by providing annotations that give hints to the compiler about where it is desirable to inline code. While the technique is able to remove most of the overhead most of the time, it is unpredictable, because inlining rules are sensitive to the particular optimisation strategies that GHC is employing. It also requires significant annotations: inline pragmas are required for to and from, as well as each of the other generic functions, which is a tedious process.

In terms of overall strategy, the inlining method could not be used on the Scrap Your Boilerplate (SYB) generics library by Lämmel and Peyton Jones [2003], because SYB relies on runtime casts [Lämmel and Peyton Jones 2004] (although they are type safe). More recently, Yallop [2017] showed how this could be avoided in a version of SYB in OCaml where type equality tests are witnessed by GADTs. From there, the features of MetaOCaml [Kiselyov 2014] are used for staging.

Adams et al. [2014] use the HERMIT plugin [Farmer 2015] as a different approach to optimising SYB. This works by interacting with GHC during the compilation process using a combination of supercompilation [Turchin 1979] and partial evaluation [Jones et al. 1993]. They too use type information to determine whether an expression should be computed statically at compile time or dynamically at runtime.

Untyped Template Haskell has been used in order to directly generate generic traversal functions in an ad-hoc manner for both geniplate in the style of uniplate [Mitchell and Runciman 2007] and Template Your Boilerplate [Adams and DuBuisson 2012] in the style of SYB. These techniques are not compositional, as you are restricted to only deriving traversals for a specific type. We are the first to apply staging to a sums-of-products based approach to generic programming, and have demonstrated that using principled type-safe techniques using reusable components that we can also generate optimal code.

Aside from our current work, there are remarkably few users of Typed Template Haskell [Pickering et al. 2020]. Recently, Willis et al. [2020] have exploited it in order to remove the overhead of using parser combinators. Looking further afield there are many more examples of staged programming in different domains which have used MetaOCaml [Kiselyov 2014] or LMS [Rompf and Odersky 2010] where the ideas can be readily applied to Typed Template Haskell.

9 Conclusions

This paper has demonstrated the use of staged-sop to write high-level generic code generators for a selection of different examples. Of course, we need not have stopped here, any function which can be derived generically using generics-sop could have been staged in a similar manner. What is missing from the story is a seamless experience for users in order to use our staged functions. At the moment they must explicitly write the splice for each generic function they use. In the future we wish to extend the instance derivation mechanism in order to allow the derivation of type class instances using generators defined using Typed Template Haskell. That way the usage of Typed Template Haskell can simply be a powerful tool for library authors and an implementation detail for users.

In principle, the staging techniques in this paper could also be applied to GHC.Generics as well as other generics libraries, although the techniques we employ work particularly well with generics-sop and may be more invasive in other libraries. In particular the functor-parameterised style of generics-sop coordinates with the Code constructor in order to provide a familiar interface which needed only a few modifications in order to support the generation of a wide variety of common generic functions without any overhead.

The raison d'être of datatype generic programming is to use statically known information about a value in order to generate families of functions which work for many different datatypes. The folly has been that despite a clear separation between the static and dynamic information, the distinction has not been enforced in order to remove the generic overhead from the generation functions. This paper shows that staged programming provides the perfect tool for writing generic functions, at a high-level, with reusable combinators that guarantee that generic abstractions will be eliminated. Finally, after all these years, we can use the full power of generic programming without regret!

Acknowledgements

The authors would like to thank Ryan Scott and all the anonymous reviewers for their helpful and constructive comments on draft versions of this paper. This work has been supported by EPSRC grant number EP/S028129/1 on "SCOPE: Scoped Contextual Operations and Effects".

References

Michael D. Adams and Thomas M. DuBuisson. 2012. Template Your Boilerplate: Using Template Haskell for Efficient Generic Programming. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2364506.2364509

Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. 2014. Optimizing SYB is Easy!. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation* (San Diego, California, USA) *(PEPM '14)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/2543728.2543730

Joachim Breitner. 2018. A Promise Checked is a Promise Kept: Inspection Testing. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/3242744.3242748

Edsko de Vries and Andres Löh. 2014. True sums of products. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, José Pedro Magalhães and Tiark Rompf (Eds.). ACM, 83–94. https://doi.org/10.1145/2633628.2633634

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 229–254.

Andrew Farmer. 2015. *HERMIT: mechanized reasoning during compilation in the Glasgow Haskell Compiler*. Ph.D. Dissertation. University of Kansas.

Ralf Hinze, Johan Jeuring, and Andres Löh. 2007. Comparing Approaches to Generic Programming in Haskell. In *Datatype-Generic Programming*, Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–149.

Jun Inoue and Walid Taha. 2016. Reasoning about multi-stage programs. *Journal of Functional Programming* 26 (2016), e22. https://doi.org/10.1017/S0956796816000253

Patricia Johann and Neil Ghani. 2007. Initial Algebra Semantics Is Enough!. In *Typed Lambda Calculi and Applications*, Simona Ronchi Della Rocca (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–222.

Patricia Johann and Neil Ghani. 2009. A Principled Approach to Programming with Nested Types in Haskell. *Higher Order Symbol. Comput.* 22, 2 (June 2009), 155–189. https://doi.org/10.1007/s10990-009-9047-7

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., USA.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Csongor Kiss, Matthew Pickering, and Nicolas Wu. 2018. Generic Deriving of Generic Traversals. *Proc. ACM Program. Lang.* 2, ICFP, Article 85 (July 2018), 30 pages. https://doi.org/10.1145/3236780

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (New Orleans, Louisiana, USA) *(TLDI '03)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/604174.604179

Ralf Lämmel and Simon Peyton Jones. 2004. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 244–255. https://doi.org/10.1145/1016850.1016883

José Pedro Magalhães. 2013. Optimisation of Generic Programs Through Inlining. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 104–121.

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

Neil Mitchell and Colin Runciman. 2007. Uniform Boilerplate and List Processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany) *(Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/1291201.1291208

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.* 12, 5 (July 2002), 393–434. https://doi.org/10.1017/S0956796802004331

Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. *A Reflection on Types*. Springer International Publishing, Cham, 292–317. https://doi.org/10.1007/978-3-319-30936-1_16

Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. A Specification for Typed Template Haskell. (2020). https://mpickering.github.io/papers/specification-typed-th.pdf

Matthew Pickering, Nicolas Wu, and Csongor Kiss. 2019. Multi-Stage Programs in Context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/3331545.3342597

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

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408971

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

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

V. F. Turchin. 1979. A Supercompiler System Based on the Language REFAL. *SIGPLAN Not.* 14, 2 (Feb. 1979), 46–54. https://doi.org/10.1145/954063.954069

Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged Selective Parser Combinators. *Proc. ACM Program. Lang.* 4, ICFP, Article 120 (Aug. 2020), 30 pages. https://doi.org/10.1145/3409002

Jeremy Yallop. 2017. Staged Generic Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110273

Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially-Static Data as Free Extension of Algebras. *Proc. ACM Program. Lang.* 2, ICFP, Article 100 (July 2018), 30 pages. https://doi.org/10.1145/3236795