Staged Sums of Products

Haskell Symposium 2020

Matthew Pickering, University of Bristol
Andres Löh, Well-Typed LLP
Nicolas Wu, Imperial College London

2020-08-28

Example: semigroup append

```haskell
class Semigroup a where
  (<>) :: a -> a -> a  -- supposed to be associative
```

Example: semigroup append

```
class Semigroup a where
  (<>) :: a -> a -> a  -- supposed to be associative
```

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

Example: semigroup append

```
class Semigroup a where
  (<>) :: a -> a -> a  -- supposed to be associative
```

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

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo (Foo is1 o1 t1) (Foo is2 o2 t2) =
  Foo (is1 <> is2) (o1 <> o2) (t1 <> t2)
```

Example: semigroup append

```
class Semigroup a where
  (<>) :: a -> a -> a  -- supposed to be associative
```

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

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo (Foo is1 o1 t1) (Foo is2 o2 t2) =
  Foo (is1 <> is2) (o1 <> o2) (t1 <> t2)
```

Example: semigroup append

```haskell
class Semigroup a where
  (<>) :: a -> a -> a   -- supposed to be associative
```

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

```haskell
sappendFoo :: Foo -> Foo -> Foo
sappendFoo (Foo is1 o1 t1) (Foo is2 o2 t2) =
  Foo (is1 <> is2) (o1 <> o2) (t1 <> t2)
```

```haskell
sappendFoo (Foo [1, 2] LT "has") (Foo [3, 4] EQ "kell")
```

Example: semigroup append

```haskell
class Semigroup a where
  (<>) :: a -> a -> a  -- supposed to be associative
```

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

```haskell
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo (Foo is1 o1 t1) (Foo is2 o2 t2) =
  Foo (is1 <> is2) (o1 <> o2) (t1 <> t2)
```

```haskell
Foo ([1, 2] <> [3, 4]) (LT <> EQ) ("has" <> "kell")
```

Example: semigroup append

```haskell
class Semigroup a where
  (<>) :: a -> a -> a  -- supposed to be associative
```

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

```haskell
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo (Foo is_1 o_1 t_1) (Foo is_2 o_2 t_2) =
  Foo (is_1 <> is_2) (o_1 <> o_2) (t_1 <> t_2)
```

```haskell
Foo [1, 2, 3, 4] LT "haskell"
```

Example: semigroup append

```haskell
class Semigroup a where
  (<>) :: a -> a -> a   -- supposed to be associative
```

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

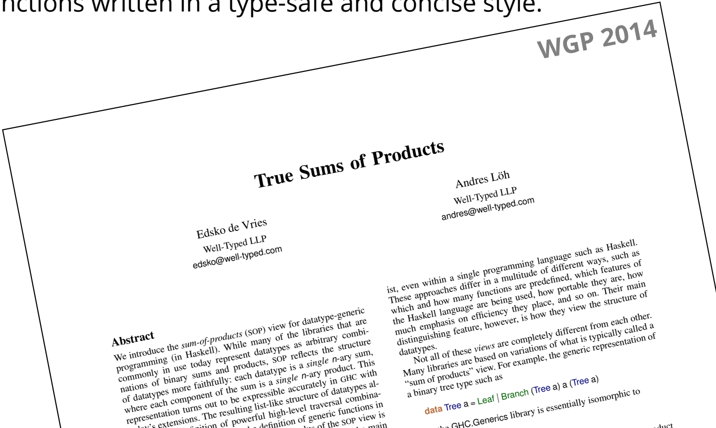
```haskell
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo (Foo is1 o1 t1) (Foo is2 o2 t2) =
  Foo (is1 <> is2) (o1 <> o2) (t1 <> t2)
```

```haskell
Foo [1, 2, 3, 4] LT "haskell"
```

Same idea applies to product types in general . . .

generics-sop

- Types are represented as n-ary **sums** (`NS`) and n-ary **products** (`NP`).
- Conversion functions and lots of combinators.
- Generic functions written in a type-safe and concise style.


WGP 2014

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 al…

ist, even within a single programming language such as Haskell. These approaches differ in a multitude of different ways, such as which and how many functions are predefined, which features of the Haskell language are being used, how portable they are, how much emphasis on efficiency they place, and so on. Their main distinguishing feature, however, is how they view the structure of datatypes.

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

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

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

Generic semigroup append

```haskell
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a1 a2 =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a1) (productTypeFrom a2))
```

```haskell
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
sappend_Foo
  (Foo [1, 2] LT "has")
  (Foo [3, 4] EQ "kell")
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWithNP (Proxy @Semigroup) (mapIII (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappendFoo :: Foo -> Foo -> Foo
sappendFoo = gsappend
```

```
productTypeTo
  (czipWithNP (Proxy @Semigroup) (mapIII (<>))
    (productTypeFrom (Foo [1, 2] LT "has"))
    (productTypeFrom (Foo [3, 4] EQ "kell")))
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo
  (czipWith_NP (Proxy @Semigroup) (map_III (<>))
    (productTypeFrom (Foo [1, 2] LT "has"))
    (productTypeFrom (Foo [3, 4] EQ "kell")))
```

```
productTypeFrom :: Foo -> NP I '[[Int], Ordering, Text]
productTypeFrom (Foo is o t) = I is :* I o :* I t :* Nil
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo
  (czipWith_NP (Proxy @Semigroup) (map_III (<>))
    (productTypeFrom (Foo [1, 2] LT "has"))
    (productTypeFrom (Foo [3, 4] EQ "kell")))
```

```
productTypeFrom :: Foo -> NP I '[[Int], Ordering, Text]
productTypeFrom (Foo is o t) = I is :* I o :* I t :* Nil
                             ≈ I is × I o × I t
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo
  (czipWith_NP (Proxy @Semigroup) (map_III (<>))
    (productTypeFrom (Foo [1, 2] LT "has"))
    (productTypeFrom (Foo [3, 4] EQ "kell")))
```

```
productTypeFrom :: Foo -> NP I '[[Int], Ordering, Text]
productTypeFrom (Foo is o t) = I is :* I o :* I t :* Nil
                             ≈   is ×    o ×    t
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo
  (czipWith_NP (Proxy @Semigroup) (map_III (<>))
    (I [1, 2] :* I LT :* I "has"  :* Nil)
    (I [3, 4] :* I EQ :* I "kell" :* Nil))
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWithNP (Proxy @Semigroup) (mapIII (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappendFoo :: Foo -> Foo -> Foo
sappendFoo = gsappend
```

```
productTypeTo
  (czipWithNP (Proxy @Semigroup) (mapIII (<>))
    (I [1, 2] :* I LT :* I "has"  :* Nil)
    (I [3, 4] :* I EQ :* I "kell" :* Nil))
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a1 a2 =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a1) (productTypeFrom a2))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo
  (czipWith_NP (Proxy @Semigroup) (map_III (<>))
    (I [1, 2] :* I LT :* I "has"  :* Nil)
    (I [3, 4] :* I EQ :* I "kell" :* Nil))
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo
    (  (map_III (<>) (I [1, 2]) (I [3, 4]))
    :* (map_III (<>) (I LT)     (I EQ))
    :* (map_III (<>) (I "has")  (I "kell"))
    :* Nil
    )
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo
    (  (map_III (<>) (I [1, 2]) (I [3, 4]))
    :* (map_III (<>) (I LT)     (I EQ))
    :* (map_III (<>) (I "has")  (I "kell"))
    :* Nil
    )
```

```
map_III :: (a -> b -> c) -> I a -> I b -> I c
map_III op (I x) (I y) = I (op x y)
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo
    (  I ([1, 2] <> [3, 4])
    :* I (LT <> EQ)
    :* I ("has" <> "kell")
    :* Nil
    )
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
productTypeTo :: NP I '[[Int], Ordering, Text] -> Foo
productTypeTo (I is :* I o :* I t :* Nil) = Foo is o t
```

```
productTypeTo
    (  I ([1, 2] <> [3, 4])
    :* I (LT <> EQ)
    :* I ("has" <> "kell")
    :* Nil
    )
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWith_NP (Proxy @Semigroup) (map_III (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo = gsappend
```

```
Foo
        ([1, 2] <> [3, 4])
        (LT <> EQ)
        ("has" <> "kell")
```

Generic semigroup append

```
gsappend :: (IsProductType a xs, All Semigroup xs) => a -> a -> a
gsappend a₁ a₂ =
  productTypeTo
    (czipWithNP (Proxy @Semigroup) (mapIII (<>))
      (productTypeFrom a₁) (productTypeFrom a₂))
```

```
sappendFoo :: Foo -> Foo -> Foo
sappendFoo = gsappend
```

```
Foo [1, 2, 3, 4] LT "haskell"
```

What about efficiency?

What about efficiency?

Typed Template Haskell

- A **typed subset** of Template Haskell.
- Construct and use Haskell **expressions** at **compilation time**.

Haskell 2002

Template Meta-programming for Haskell

Tim Sheard
OGI School of Science & Engineering
Oregon Health & Science University
sheard@cse.ogi.edu

Simon Peyton Jones
Microsoft Research Ltd
simonpj@microsoft.com

Abstract

We propose a new extension to the purely functional programming language Haskell that supports *compile-time meta-programming*. The purpose of the system is to support the *algorithmic* construction of programs at compile-time.

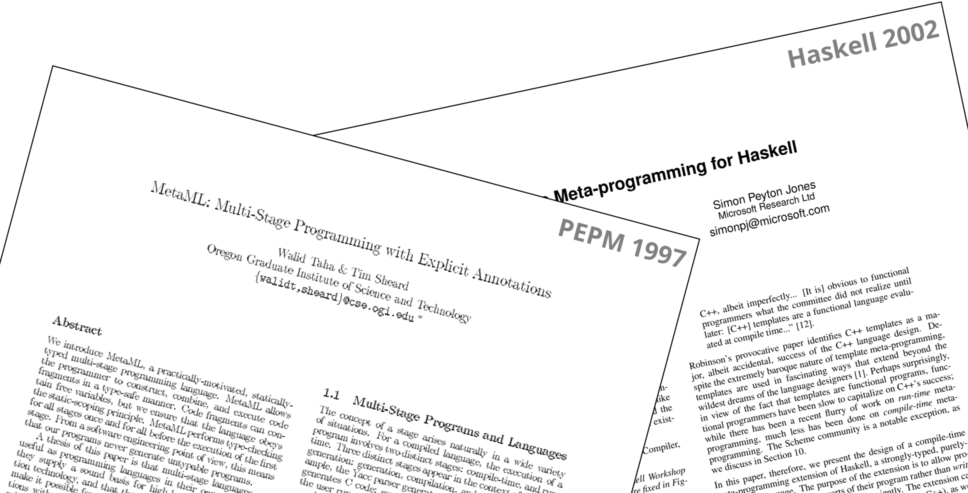
The ability to generate code at compile time allows the programmer to implement such features as polytypic programs, macro-like expansion, user directed optimization (such as inlining), and the generation of supporting data structures and functions from existing data structures and functions.

C++, albeit imperfectly... [It is] obvious to functional programmers what the committee did not realize until later: [C++] templates are a functional language evaluated at compile time..." [12].

Robinson's provocative paper identifies C++ templates as a major, albeit accidental, success of the C++ language design. Despite the extremely baroque nature of template meta-programming, templates are used in fascinating ways that extend beyond the wildest dreams of the language designers [1]. Perhaps surprisingly, in view of the fact that templates are functional programs, functional programmers have been slow to capitalize on C++'s success; while there has been a recent flurry of work on *run-time* meta-programming, much less has been done on *compile-time* meta-programming. The Scheme community is a notable exception, as we discuss in Section 10.

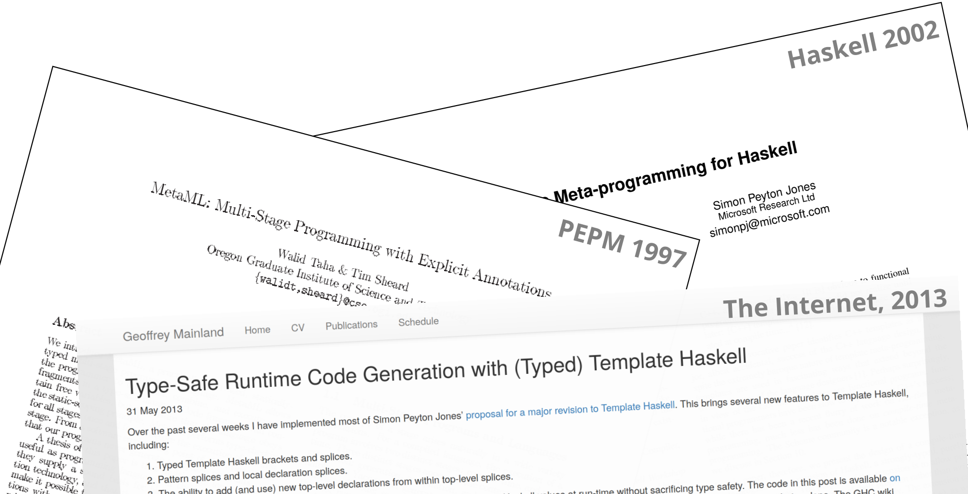
Typed Template Haskell

- A **typed subset** of Template Haskell.
- Construct and use Haskell **expressions** at **compilation time**.



Typed Template Haskell

- A **typed subset** of Template Haskell.
- Construct and use Haskell **expressions** at **compilation time**.



Staging constructs

Quotes

$$\frac{\texttt{e :: t}}{\texttt{[||e||] :: Code t}}$$

Prevent reduction, build an AST.

Staging constructs

Quotes

$$\frac{\texttt{e :: t}}{\texttt{[||e||] :: Code t}}$$

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

Prevent reduction, build an AST.

Staging constructs

Quotes

```
   e :: t
────────────────
[||e||] :: Code t
```

Prevent reduction, build an AST.

Splices

```
e :: Code t
───────────
 $$e :: t
```

Re-enable reduction, insert into an AST.

Staging constructs

Quotes

$$\frac{\texttt{e :: t}}{\texttt{[||e||] :: Code t}}$$

Prevent reduction, build an AST.

Splices

$$\frac{\texttt{e :: Code t}}{\texttt{\$\$e :: t}}$$

Re-enable reduction, insert into an AST.

Top-level splices insert into the current module.

Staging constructs

Quotes

$$\frac{\texttt{e :: t}}{\texttt{[||e||] :: Code t}}$$

Prevent reduction, build an AST.

Splices

$$\frac{\texttt{e :: Code t}}{\texttt{\$\$e :: t}}$$

Re-enable reduction, insert into an AST.

Top-level splices insert into the current module.

Splices and quotes cancel each other out: `$$([||e||]) ⇝ e` .

Staged semigroup append

```
sgsappend :: (IsProductType a xs, All (Quoted Semigroup) xs) =>
               Code a -> Code a -> Code a
sgsappend c1 c2 =
  sproductTypeFrom c1 $ \ a₁ -> sproductTypeFrom c2 $ \ a₂ ->
    sproductTypeTo (czipWith_NP (Proxy @(Quoted Semigroup))
      (map_CCC [||(<>)||]) a₁ a₂)
```

Staged semigroup append

```
sgsappend :: (IsProductType a xs, All (Quoted Semigroup) xs) =>
               Code a -> Code a -> Code a
sgsappend c1 c2 =
  sproductTypeFrom c1 $ \ a₁ -> sproductTypeFrom c2 $ \ a₂ ->
    sproductTypeTo (czipWith_NP (Proxy @(Quoted Semigroup))
      (map_CCC [|| (<>) ||]) a₁ a₂)
```

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo foo₁ foo₂ = $$(sgsappend [|| foo₁ ||] [|| foo₂ ||])
```

Staged semigroup append

```
sappend_Foo foo_1 foo_2
```

Staged semigroup append

```
$$(sgsappend [||foo₁||] [||foo₂||])
```

Staged semigroup append

```
$$(
  sproductTypeFrom [||foo₁||] $ \ a₁ ->
      sproductTypeFrom [||foo₂||] $ \ a₂ ->
          sproductTypeTo
            (czipWith_NP (Proxy @(Quoted Semigroup))
              (map_CCC [||(<>)||])
              a₁
              a₂)
)
```

Staged semigroup append

```
$$(
  sproductTypeFrom [||foo₁||] $ \ a₁ ->
      sproductTypeFrom [||foo₂||] $ \ a₂ ->
          sproductTypeTo
            (czipWithNP (Proxy @(Quoted Semigroup))
              (mapCCC [||(<>)||])
              a₁
              a₂)
)
```

```
sproductTypeFrom :: Code Foo -> (NP C '[[Int], Ordering, Text] -> Code r) -> Code r
sproductTypeFrom foo k =
  [||case $$foo of { Foo is o t ->
      $$(k (C [||is||] :* C [||o||] :* C [||t||] :* Nil)) }
  ||]
```

Staged semigroup append

```
$$(
  sproductTypeFrom [||foo₁||] $ \ a₁ ->
      sproductTypeFrom [||foo₂||] $ \ a₂ ->
          sproductTypeTo
            (czipWith_NP (Proxy @(Quoted Semigroup))
              (map_CCC [||(<>)||])
              a₁
              a₂)
)
```

```
sproductTypeFrom :: Code Foo -> (NP C '[[Int], Ordering, Text] -> Code r) -> Code r
sproductTypeFrom foo k =
  [||case $$foo of { Foo is o t ->
      $$(k (C [||is||] :* C [||o||] :* C [||t||] :* Nil)) }
  ||]

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

Staged semigroup append

```
$$(
  [||case foo₁ of { Foo is₁ o₁ t₁ ->
    $$(sproductTypeFrom [||foo₂||] $ \ a₂ ->
            sproductTypeTo
              (czipWithNP (Proxy @(Quoted Semigroup))
                (mapCCC [||(<>)||])
                (C [||is₁||] :* C [||o₁||] :* C [||t₁||] :* Nil)
                a₂)) }
  ||]
)
```

Staged semigroup append

```
$$(
  [||case foo₁ of { Foo is₁ o₁ t₁ ->
        case foo₂ of { Foo is₂ o₂ t₂ ->
          $$(sproductTypeTo
               (czipWith_NP (Proxy @(Quoted Semigroup))
                 (map_CCC [||(<>)||])
                 (C [||is₁||] :* C [||o₁||] :* C [||t₁||] :* Nil)
                 (C [||is₂||] :* C [||o₂||] :* C [||t₂||] :* Nil))) } }
  ||]
)
```

Staged semigroup append

```
$$(
  [||case foo₁ of { Foo is₁ o₁ t₁ ->
       case foo₂ of { Foo is₂ o₂ t₂ ->
         $$(sproductTypeTo
              (   map_CCC [||(<>)||] (C [||is₁||]) (C [||is₂||])
               :* map_CCC [||(<>)||] (C [||o₁||])  (C [||o₂||])
               :* map_CCC [||(<>)||] (C [||t₁||])  (C [||t₂||])
               :* Nil
              )) } }
  ||]
)
```

Staged semigroup append

```
$$(
  [||case foo₁ of { Foo is₁ o₁ t₁ ->
       case foo₂ of { Foo is₂ o₂ t₂ ->
         $$(sproductTypeTo
              (   map_CCC [||(<>)||] (C [||is₁||]) (C [||is₂||])
               :* map_CCC [||(<>)||] (C [||o₁||])  (C [||o₂||])
               :* map_CCC [||(<>)||] (C [||t₁||])  (C [||t₂||])
               :* Nil
              )) } }
  ||]
)
```

```
map_CCC :: Code (a -> b -> c) -> C a -> C b -> C c
map_CCC op (C x) (C y) = C [||$$op $$x $$y||]
```

Staged semigroup append

```
$$(
  [||case foo1 of { Foo is1 o1 t1 ->
       case foo2 of { Foo is2 o2 t2 ->
         $$(sproductTypeTo
              (   C [||(<>) is1 is2||]
               :* C [||(<>) o1 o2||]
               :* C [||(<>) t1 t2||]
               :* Nil
              )) } }
  ||]
)
```

Staged semigroup append

```
$$(
  [||case foo1 of { Foo is o t ->
       case foo2 of
         $$(sproductTypeTo
              (   C [||(<>) is1 is2||]
               :* C [||(<>) o1 o2||]
               :* C [||(<>) t1 t2||]
               :* Nil
              )) } }
  ||]
)
```

```
sproductTypeTo :: NP C '[[Int], Ordering, Text] -> Code Foo
sproductTypeTo (C is :* C o :* C t :* Nil) = [||Foo $$is $$o $$t||]
```

Staged semigroup append

```
$$(
  [||case foo1 of { Foo is1 o1 t1 ->
       case foo2 of { Foo is2 o2 t2 ->
         Foo
                     ((<>) is1 is2)
                     ((<>) o1 o2)
                     ((<>) t1 t2) } }
  ||]
)
```

Staged semigroup append

```
case foo1 of { Foo is1 o1 t1 ->
  case foo2 of { Foo is2 o2 t2 ->
    Foo ((<>) is1 is2) ((<>) o1 o2) ((<>) t1 t2) } }
```

Staged semigroup append

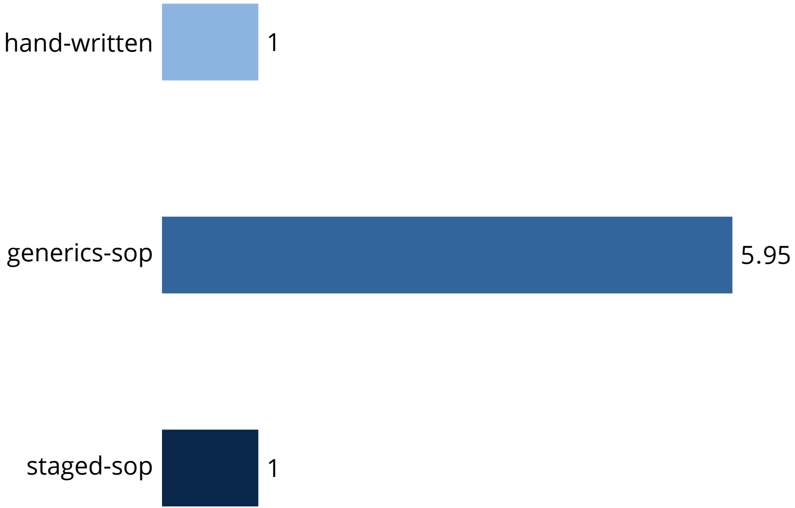
```
case foo₁ of { Foo is₁ o₁ t₁ ->
  case foo₂ of { Foo is₂ o₂ t₂ ->
    Foo ((<>) is₁ is₂) ((<>) o₁ o₂) ((<>) t₁ t₂) } }
```

This is **obviously equivalent** to the hand-written version:

```
sappend_Foo :: Foo -> Foo -> Foo
sappend_Foo (Foo is₁ o₁ t₁) (Foo is₂ o₂ t₂) =
  Foo (is₁ <> is₂) (o₁ <> o₂) (t₁ <> t₂)
```

What about efficiency?

What about efficiency?



staged-sop

- A variant of generics-sop.
- Can reuse the `NS` and `NP` types, because they are already parameterized over a type constructor.
- Conversion functions use `C` rather than `I` as the type constructor.
- Can reuse nearly all of the provided combinators for working with sums and products.
- Requires proper handling of class constraints in Typed Template Haskell.

Quotes and constraints

```
(<>) :: Semigroup a => a -> a -> a
-----------------------------------
        [||(<>)||] :: ...
```

Quotes and constraints

```
(<>) :: Semigroup a => a -> a -> a
----------------------------------------------
[||(<>)||] :: Semigroup a => Code (a -> a -> a)
```

GHC 8.10 ad-hoc answer, which is **wrong**.

Quotes and constraints

```
(<>) :: Semigroup a => a -> a -> a
--------------------------------------------------
[||(<>)||] :: Code (Semigroup a => a -> a -> a)
```

An option once impredicativity is available, but **not first class** (does not allow to decouple the constraint from the quote).

Quotes and constraints

```
(<>) :: Semigroup a => a -> a -> a
--------------------------------------------------------
[||(<>)||] :: Quoted Semigroup a => Code (a -> a -> a)
```

Our answer, which reflects that we need the constraint satisfied **when this fragment is spliced**, not when it is constructed.

Quotes and constraints

```
(<>) :: Semigroup a => a -> a -> a
```

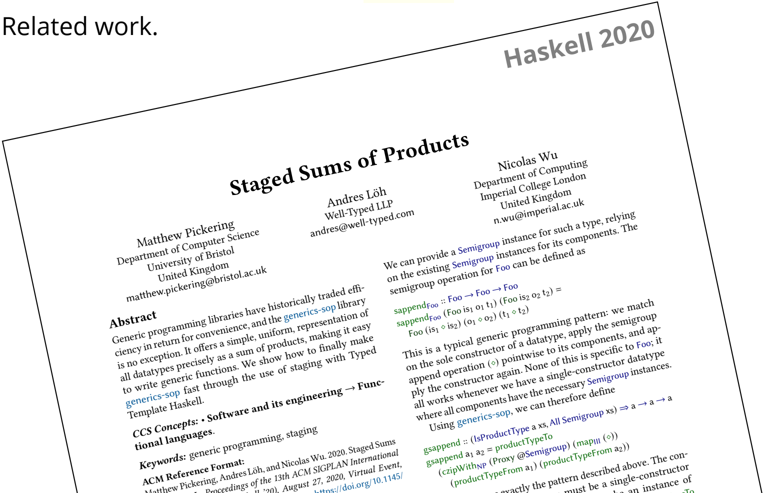
```
[||(<>)||] :: Quoted Semigroup a => Code (a -> a -> a)
```

Our answer, which reflects that we need the constraint satisfied **when this fragment is spliced**, not when it is constructed.

Implemented in a GHC branch; GHC proposal to follow.

In the paper

- More examples of staged generic functions.
- A more detailed explanation of `Quoted`.
- Related work.



Haskell 2020

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 ... *Proceedings of the 13th ACM SIGPLAN International ...* ('20), *August 27, 2020, Virtual Event,* ... https://doi.org/10.1145/

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))
```

... exactly the pattern described above. The con... must be a single-constructor ... be an instance of ...

Conclusions

- We can finally write datatype-generic programs at a **high level**, with **type safety** and **reliable performance**.
- The identified improvements to constraint handling in Typed Template Haskell are independently useful.
- It is wonderful that we can reuse so much of the original generics-sop library.
- Nevertheless, staging in this style is also applicable to other generic programming approaches such as `GHC.Generics` and SYB.

Try the prototype:

`https://github.com/well-typed/generics-sop/tree/staged-sop`

(README has instructions on how to build a suitably patched GHC branch.)