Staged generics-sop

IFIP WG 2.1 meeting #79, Otterlo

Matthew Pickering, Andres Löh

2020-01-07

Well-Typed
The Haskell Consultants

generics-sop

Sums and products

```
Sum     :: (a -> Type) -> [a] -> Type
Product :: (a -> Type) -> [a] -> Type
```

Sums and products

```
Sum     :: (a -> Type) -> [a] -> Type
Product :: (a -> Type) -> [a] -> Type
```

$$\texttt{Sum}\ \ \ \ \ \texttt{f}\ [x_1, x_2, x_3] \approx \texttt{f}\ x_1 + \texttt{f}\ x_2 + \texttt{f}\ x_3$$

$$\texttt{Product}\ \texttt{f}\ [x_1, x_2, x_3] \approx \texttt{f}\ x_1 \times \texttt{f}\ x_2 \times \texttt{f}\ x_3$$

Sums and products

```
Sum     :: (a -> Type) -> [a] -> Type
Product :: (a -> Type) -> [a] -> Type
```

$$\texttt{Sum}\ \ \ \ \ f\ [x_1, x_2, x_3] \approx f\ x_1 + f\ x_2 + f\ x_3$$

$$\texttt{Product}\ f\ [x_1, x_2, x_3] \approx f\ x_1 \times f\ x_2 \times f\ x_3$$

$$\texttt{Sum (Product f)}\ ([x_1, x_2], [], [x_3, x_4, x_5]) \approx (f\ x_1 \times f\ x_2) + 1 + (f\ x_3 \times f\ x_4 \times f\ x_5)$$

Example datatype

```haskell
data Animal =
    HoppingAnimal String Double
  | WalkingAnimal String Int
```

Example datatype

```
data Animal =
    HoppingAnimal String Double
  | WalkingAnimal String Int
```

```
Sum (Product I) [[String, Double], [String, Int]]
  ≈ (I String × I Double) + (I String × I Int)
  ≈ (String × Double) + (String × Int)
```

Example datatype

```haskell
data Animal =
    HoppingAnimal String Double
  | WalkingAnimal String Int
```

```
Sum (Product I) [[String, Double], [String, Int]]
  ≈ (I String × I Double) + (I String × I Int)
  ≈ (String × Double) + (String × Int)
```

```haskell
Description Animal = [[String, Double], [String, Int]]

from :: Animal -> Sum (Product I) (Description Animal)
to   :: Sum (Product I) (Description Animal) -> Animal
```

A class for representable types

```haskell
class (All (All Top) (Description a)) => Generic a where
  type Description a :: [[Type]]
  from :: a -> Sum (Product I) (Description a)
  to   :: Sum (Product I) (Description a) -> a
```

Operations on sums and products

```
mapSum     ::
      All Top xs
   => (∀ x . f x -> g x) -> Sum     f xs -> Sum     g xs

mapProduct ::
      All Top xs
   => (∀ x . f x -> g x) -> Product f xs -> Product g xs
```

Well-Typed

Operations on sums and products

```
cmap_Sum     ::
               All c xs
   => (∀ x . c x => f x -> g x)
   -> Sum     f xs -> Sum     g xs

cmap_Product ::
               All c xs
   => (∀ x . c x => f x -> g x)
   -> Product f xs -> Product g xs

cmap_SoP     ::
               All (All c) xs
   => (∀ x . c x => f x -> g x)
   -> Sum (Product f) xs -> Sum (Product g) xs
```

Operations on sums and products

```
cpure_Product ::
  All c xs
   => (∀ x . c x => f x)
   -> Product f xs
```

Operations on sums and products

```
collapse_Sum     :: All Top xs => Sum     (K a) xs -> a
collapse_Product :: All Top xs => Product (K a) xs -> [a]
```

Well-Typed

Operations on sums and products

```
zipWith_Product ::
      All Top xs
   => (∀ x . f x -> g x -> h x)
   -> Product f xs -> Product g xs -> Product h xs
```

Well-Typed

Operations on sums and products

```
zipWith_Product ::
      All Top xs
   => (∀ x . f x -> g x -> h x)
   -> Product f xs -> Product g xs -> Product h xs
```

```
zipWith_Sum     ::
      All Top xs
   => (∀ x . f x -> g x -> h x)
   -> Product f xs -> Sum     g xs -> Sum     h xs
```

Operations on sums and products

```
anaProduct ::
      All Top xs
   => (∀ y ys . s (y : ys) -> (f y, s ys))
   -> s xs -> Product f xs
```

Arities of each constructor

```
constructorArities ::
  Generic a => Product (K Word) (Description a)
constructorArities =
  cpure_Product @(All Top) go
  where
    go :: ∀ xs . All Top xs => K Word xs
    go = K (fromIntegral (length_SList @xs))
```

Arities of each constructor

```
constructorArities ::
  Generic a => Product (K Word) (Description a)
constructorArities =
  cpure_Product @(All Top) go
  where
    go :: ∀ xs . All Top xs => K Word xs
    go = K (fromIntegral (length_SList @xs))
```

Example:

```
data Animal =
     HoppingAnimal String Double
   | WalkingAnimal String Int
```

```
constructorArities @Animal
   = K 2 :* K 2 :* Nil
```

Well-Typed

Numbering each constructor

```
constructorNumbers ::
  Generic a => Product (K Word) (Description a)
constructorNumbers =
  ana_Product
    (\ (K i) -> (K i, K (i + 1)))
    (K 0)
```

Well-Typed

Numbering each constructor

```
constructorNumbers ::
  Generic a => Product (K Word) (Description a)
constructorNumbers =
  anaProduct
    (\ (K i) -> (K i, K (i + 1)))
    (K 0)
```

Example:

```
data Animal =
     HoppingAnimal String Double
   | WalkingAnimal String Int
```

```
constructorNumbers @Animal
   = K 0 :* K 1 :* Nil
```

Encoding, generically

```haskell
gencode ::
      ∀ a . (Generic a, All (All Serialise) (Description a))
   => a -> Encoding
gencode x =
  collapse_Sum
    (czipWith3_Sum @(All Top)
      (\ (K a) (K i) encs ->
        K   (  encodeListLen (a + 1)
            <> encodeWord i
            <> mconcat (collapse_Product encs)
             )
      )
      (constructorArities @a)
      (constructorNumbers @a)
      (cmap_SoP @Serialise (\ (I y) -> K (encode y)) (from x))
    )
```

Well-Typed

Staging using Typed Template Haskell

Quotes and splices

```
type Code a = Q (TExp a)
newtype Code' a = Code {unCode :: Code a}
```

Quotes and splices

```
type Code a = Q (TExp a)
newtype Code' a = Code {unCode :: Code a}
```

```
ex1 :: Code Int
ex1 = [|| 1 + 2 + 3 ||]
```

Quotes and splices

```
type Code a = Q (TExp a)
newtype Code' a = Code {unCode :: Code a}
```

```
ex1 :: Code Int
ex1 = [|| 1 + 2 + 3 ||]
```

```
ex2 :: Code Int
ex2 = [|| $$ex1 * $$ex1 ||]
```

AST: `(1 + 2 + 3) * (1 + 2 + 3)`

Lifting

```
f :: Int -> Code Int
f x = [|| x + x ||]
```

```
ex₃ :: Code Int
ex₃ = [|| $$(f (1 + 2 + 3)) ||]
```

AST: `6 + 6`

Lifting

```
f :: Int -> Code Int
f x = [|| x + x ||]
```

```
ex₃ :: Code Int
ex₃ = [|| $$(f (1 + 2 + 3)) ||]
```

AST: `6 + 6`

```
g :: Lift a => [a] -> Code [a]
g xs = [|| reverse xs ||]
```

```
ex₄ :: Code [Int]
ex₄ = [|| $$(g (replicate 3 1)) ||]
```

AST: `reverse [1, 1, 1]`

Using variables before they are defined

```
f :: Int -> Code Int
f x = [|| x + x ||]
```

```
ex5 :: Code (Int -> Int)
ex5 = [|| \ x -> $$(f x) ||]  -- not ok
```

Stage error: ‘`x`’ is bound at stage 2 but used at stage 1

Using variables before they are defined

```
f :: Int -> Code Int
f x = [|| x + x ||]
```

```
ex5 :: Code (Int -> Int)
ex5 = [|| \ x -> $$(f x) ||]  -- not ok
```

Stage error: ‘ `x` ’ is bound at stage 2 but used at stage 1

But this is ok:

```
h :: Code Int -> Code Int
h x = [|| $$x + $$x ||]
```

```
ex6 :: Code (Int -> Int)
ex6 = [|| \ x -> $$(h [|| x ||]) ||]
```

AST: `\ x -> x + x`

Hello world of staging

```haskell
square :: Int -> Int
square x = x * x
```

Hello world of staging

```
square :: Int -> Int
square x = x * x
```

```
power :: Int -> Int -> Int
power n x
  | n == 0     = 1
  | even n     = square (power (n `div` 2) x)
  | otherwise = x * power (n - 1) x
```

Hello world of staging

```
square :: Int -> Int
square x = x * x
```

```
power :: Int -> Int -> Int
power n x
   | n == 0    = 1
   | even n    = square (power (n `div` 2) x)
   | otherwise = x * power (n - 1) x
```

```
spower :: Int -> Code Int -> Code Int
spower n x
   | n == 0    = [|| 1 ||]
   | even n    = [|| square $$(spower (n `div` 2) x) ||]
   | otherwise = [|| $$x * $$(spower (n - 1) x) ||]
```

Staging generics-sop

Basic idea

Structure is statically known, so rather than

```
Sum (Product I) (Description a)
```

let us use

```
Sum (Product Code') (Description a)
```

Staged conversions

```haskell
class Generic a => SGeneric a where
  sfrom ::
        Code a
     -> (Sum (Product Code') (Description a) -> Code r)
     -> Code r
  sto   ::
        Sum (Product Code') (Description a)
     -> Code a
```

Well-Typed

Example

The function `sfrom` introduces case analysis and passes the representation to the continuation:

```
instance SGeneric Animal where
  sfrom x k =
    [||
      case $$x of
        HoppingAnimal n d ->
          $$(k (Z (Code [|| n ||]:* Code [|| d ||]:* Nil)))
        WalkingAnimal n i ->
          $$(k (S (Z (Code [|| n ||]:* Code [|| i ||]:* Nil))))
     ||]
  sto x = ...
```

Staged generic encode

```
gencode ::
      ∀ a . (Generic a, All (All Serialise) (Description a))
   => a -> Encoding
gencode x =
  collapse_Sum
    (czipWith3_Sum @(All Top)
      (\ (K a) (K i) encs ->
        K   (  encodeListLen (a + 1)
            <> encodeWord i
            <> mconcat (collapse_Product encs)
            )
      )
      (constructorArities @a)
      (constructorNumbers @a)
      (cmap_SoP @Serialise (\ (I y) -> K (encode y)) (from x))
    )
```

Staged generic encode

```
sgencode ::
      ∀ a . (SGeneric a, All (All Serialise) (Description a))
   => Code (a -> Encoding)
sgencode =
  [|| \ x -> $$(sfrom [|| x ||] $ \ x' ->
    collapse_Sum
      (czipWith3_Sum @(All Top)
        (\ (K a) (K i) encs -> let a' = a + 1 in
          K [||  encodeListLen a'
              <>  encodeWord i
              <>  $$(smconcat (collapse_Product encs))
              ||]
        )
        (constructorArities @a)
        (constructorNumbers @a)
        (cmap_SoP @Serialise
          (\ (Code y) -> K [|| encode $$y ||]) x')
      )
  ) ||]
```

Missing function

```
smconcat :: Monoid a => [Code a] -> Code a
smconcat []       = [|| mempty ||]
smconcat [x]      = x
smconcat (x : xs) = [|| $$x <> $$(smconcat xs) ||]
```

Well-Typed

Status

What (also) works

Implementing other staged generic functions:

- Deriving lenses (getters and setters).
- Generic equality and comparison.
- ...

Current limitations

```haskell
data IntList = IntCons Int IntList | IntNil

instance Serialise IntList where
  encode = $$(sgencode @IntList)
```

GHC stage restriction: instance for ‘`Serialise IntList`’ is used in a top-level splice […] and must be imported, not defined locally

Current limitations

```haskell
data IntList = IntCons Int IntList | IntNil
```

```haskell
instance Serialise IntList where
  encode = $$(sgencode @IntList)
```

GHC stage restriction: instance for ‘`Serialise IntList`’ is used in a top-level splice [. . .] and must be imported, not defined locally

```haskell
data Option a = None | Some a
```

```haskell
instance Serialise (Option a) where
  encode = $$(sgencode @(Option a))
```

No instance for `(Serialise a)` arising from a use of ‘`sgencode`’

Open questions

Are the conversion functions `sfrom` and `sto` sufficient?

E.g. quadratic code size for generic equality.

Open questions

Are the conversion functions `sfrom` and `sto` sufficient?

E.g. quadratic code size for generic equality.

Can we transfer all the other known techniques from staging SYB (Yallop)?