Applying Type-Level and Generic Programming in Haskell

Summer School on Generic and Effectful Programming

Andres Löh

6–10 July 2015

Well-Typed
The Haskell Consultants

Plan for the week

Monday:

- Learn about *n*-ary products.
- Along the way, discuss nearly everything we need in terms of Haskell type-level programming features.

Wednesday:

- Introduce *n*-ary sums and the generics-sop view.
- Representing datatypes using generics-sop.
- More combinators and simple applications.

Today:

- More applications.
- A few more useful combinators.
- Metadata and user-defined metadata.

Overloading common combinators

Four types, common functions

```
NP
NS
POP
SOP
```

```
hcpure_POP :: (SListI xss, All SListI xss, All2 c xss)
           => Proxy c -> (forall a . c a => f a) -> POP f x
hcpure_NP  :: (SListI xs, All c xs)
           => Proxy c -> (forall a . c a => f a) -> NP  f x
```

```
hpure_POP :: (SListI xss, All SListI xss) => (forall a . f
hpure_NP  :: (SListI xs)                  => (forall a . f
```

A class

```
class HPure (h :: (k -> *) -> (l -> *)) where
  hcpure :: (SListIMap h x, AllMap h c x)
         => Proxy c -> (forall a . c a => f a) -> h f x
  hpure  :: (SListIMap h x)
         =>            (forall a . f a) -> h f x
```

Different constraints

```
type family SListIMap (h :: (k -> *) -> (l -> *))
                      (xs :: l)
                       :: Constraint
type instance SListIMap NP  xs  = SListI  xs
type instance SListIMap POP xss = SListI2 xss
```

```
class (SListI xss, All SListI xss) => SListI2 xss
instance (SListI xss, All SListI xss) => SListI2 xss
```

```
type family AllMap (h :: (k -> *) -> (l -> *))
                   (c :: k -> Constraint)
                   (xs :: l)
                    :: Constraint
type instance AllMap NP  c xs  = All  c xs
type instance AllMap POP c xss = All2 c xss
```

Instances

```
instance HPure NP  where
  hcpure = hcpure_NP
  hpure  = hpure_NP
instance HPure POP where
  hcpure = hcpure_POP
  hpure  = hpure_POP
```

We don’t show all the definitions.

Generalizing hap

A clas HAp such that:

```
hap :: (...) => NP  (f -.-> g) xs  -> NP  f xs  -> NP  g xs
hap :: (...) => NP  (f -.-> g) xs  -> NS  f xs  -> NS  g xs
hap :: (...) => POP (f -.-> g) xss -> POP f xss -> POP g xss
hap :: (...) => POP (f -.-> g) xss -> SOP f xss -> SOP g xss
```

Mapping and zipping

```
hmap :: (...) => (forall a . f a -> g a)
              -> NP  f xs  -> NP  g xs
hmap :: (...) => (forall a . f a -> g a)
              -> NS  f xs  -> NS  g xs
hmap :: (...) => (forall a . f a -> g a)
              -> POP f xss -> POP g xss
hmap :: (...) => (forall a . f a -> g a)
              -> SOP f xss -> SOP g xss
```

Similarly for `hcmap`, `hzipWith`, `hczipWith`.

Collapsing

```
hcollapse :: (...) => NP  (K a) xs  -> [a]
hcollapse :: (...) => NS  (K a) xs  -> a
hcollapse :: (...) => POP (K a) xss -> [[a]]
hcollapse :: (...) => SOP (K a) xss -> [a]
```

Generating test data

QuickCheck

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

Let’s focus on `arbitrary`.

Generators

```haskell
data Gen a

instance Functor Gen
instance Applicative Gen
instance Monad Gen

elements  :: [a] -> Gen a
oneof     :: [Gen a] -> Gen a
frequency :: [(Int, Gen a)] -> Gen a
sized     :: (Int -> Gen a) -> Gen a
resize    :: Int -> Gen a -> Gen a
```

Dealing with effects

Like `gdefAll'`, build table of recursive calls and inject:

```
apInjs_POP
  (hcpure (Proxy :: Proxy Arbitrary) arbitrary)
   :: (SListI xss, All SListI xss, All2 Arbitrary xss)
   => [SOP Gen xss]
```

Well-Typed

Dealing with effects

Like `gdefAll'`, build table of recursive calls and inject:

```
apInjs_POP
  (hcpure (Proxy :: Proxy Arbitrary) arbitrary)
   :: (SListI xss, All SListI xss, All2 Arbitrary xss)
   => [SOP Gen xss]
```

How to go from `SOP Gen xss` to `Gen (SOP I xss)`?

Sequencing

```
hsequence :: (..., Applicative f)
          => NP  f xs  -> f (NP  I xs)
hsequence :: (..., Applicative f)
          => NS  f xs  -> f (NS  I xs)
hsequence :: (..., Applicative f)
          => POP f xss -> f (POP I xss)
hsequence :: (..., Applicative f)
          => SOP f xss -> f (SOP I xss)
```

More general sequencing

```
newtype (f :.: g) x = Comp {unComp :: f (g x)}
```

```
hsequence' :: (..., Applicative f)
           => NP  (f :.: g) xs  -> f (NP  g xs)
hsequence' :: (..., Applicative f)
           => NS  (f :.: g) xs  -> f (NS  g xs)
hsequence' :: (..., Applicative f)
           => POP (f :.: g) xss -> f (POP g xss)
hsequence' :: (..., Applicative f)
           => SOP (f :.: g) xss -> f (SOP g xss)
```

The NP case

```
hsequence_NP' :: (Applicative f)
              => NP (f :.: g) xs -> f (NP g xs)
hsequence_NP' Nil               =  pure Nil
hsequence_NP' (Comp x :* xs)    =  (:*)
                               <$> x
                               <*> hsequence_NP' xs
```

```
hsequence_NP :: (SListI xs, Applicative f)
             => NP f xs -> f (NP I xs)
hsequence_NP = hsequence_NP' . hmap (Comp . fmap I)
```

Completing `garbitrary`

```
apInjs_POP
  (hcpure (Proxy :: Proxy Arbitrary) arbitrary)
   :: (SListI xss, All SListI xss, All2 Arbitrary xss)
   => [SOP Gen xss]
```

Completing `garbitrary`

```
apInjs_POP
  (hcpure (Proxy :: Proxy Arbitrary) arbitrary)
   :: (SListI xss, All SListI xss, All2 Arbitrary xss)
   => [SOP Gen xss]
```

```
map (fmap to . hsequence) (apInjs_POP (...))
   :: (Generic a, All2 Arbitrary (Code a)) => [Gen a]
```

Now we can apply

```
oneof :: [Gen a] -> Gen a
```

Completing `garbitrary` – contd.

```
garbitrary :: (Generic a, All2 Arbitrary (Code a))
           => Gen a
garbitrary = oneof
           $ map (fmap to . hsequence)
           $ apInjs_POP
           $ hcpure (Proxy :: Proxy Arbitrary)
           $ arbitrary
```

Testing

```
instance Arbitrary Expr where
  arbitrary = garbitrary
```

```
GHCi> sample (arbitrary :: Gen Expr)
```

Testing

```haskell
instance Arbitrary Expr where
  arbitrary = garbitrary
```

```
GHCi> sample (arbitrary :: Gen Expr)
```

Most likely does not terminate.

Configuring generic functions

Using `frequency`

```
oneof :: [Gen a]                  -> Gen a
frequency :: [(Int, Gen a)] -> Gen a
```

We need the frequencies of the constructors as input.

Using `frequency`

```
oneof :: [Gen a]                -> Gen a
frequency :: [(Int, Gen a)] -> Gen a
```

We need the frequencies of the constructors as input.

```
NP (K Int) (Code a)
```

guaranteed to contain the right number of weights.

More flexible `garbitrary`

```
garbitraryWithFreqs
   :: (Generic a, All2 Arbitrary (Code a))
   => NP (K Int) (Code a) -> Gen a
garbitraryWithFreqs freqs =
      frequency
    $ hcollapse
    $ hzipWith
        (\(K x) (K y)
           -> K (x, fmap to (hsequence (SOP y))))
        freqs
        (     injections
         `hap` unPOP (hcpure pArbitrary arbitrary))
```

```
pArbitrary :: Proxy Arbitrary
pArbitrary = Proxy :: Proxy Arbitrary
```

A better attempt

```haskell
instance Arbitrary Expr where
  arbitrary =
    garbitraryWithFreqs
      (K 5 :* K 5 :* K 2 :* K 1 :* Nil)
```

Well-Typed

Computing constructor arities

```haskell
garities :: forall a . (Generic a) => Proxy a -> NP (K Int)
garities _ =
    hcmap (Proxy :: Proxy SListI)
          (K . length . hcollapse)
  $ unPOP $ hpure (K ())
```

```
GHCi> garities (Proxy :: Proxy Expr)
 K 1 :* (K 1 :* (K 2 :* (K 3 :* Nil)))
```

Sizing the generator

```
garbitrarySized ::
  forall a . (Generic a, All2 Arbitrary (Code a)) => Gen a
garbitrarySized = sized go
  where
    go n = oneof (map snd (filtered table))
      where
        table :: [(Int, Gen a)]
        table = hcollapse
              $ hzipWith aux
                  (garities (Proxy :: Proxy a))
                  (    injections
                   `hap` unPOP (hcpure pArbitrary arbitrary))
        aux ::   forall x . K Int x -> K (NS (NP Gen) (Code a)) x
            ->  K (Int, Gen a) x
        aux (K arity) (K gen) =
          K (arity, resize (n `div` arity)
                  (fmap to (hsequence (SOP gen))))
        filtered | n <= 0     = filter ((<= 1) . fst)
                 | otherwise = id
```

Metadata

Codes cover only structure, not names

```
gcoerce :: (Generic a, Generic b, Code a ~ Code b) => a ->
gcoerce = to . from
```

The `DatatypeInfo` type

```
data DatatypeInfo (code :: [[*]]) where
  ADT      :: ModuleName -> DatatypeName
           -> NP ConstructorInfo xss
           -> DatatypeInfo xss
  Newtype :: ModuleName -> DatatypeName
           -> ConstructorInfo '[x]
           -> DatatypeInfo '['[x]]
```

```
type ModuleName      = String
type DatatypeName    = String
type ConstructorName = String
type FieldName       = String
```

Constructor information

```
data ConstructorInfo (xs :: [*]) where
  Constructor :: ConstructorName
              -> ConstructorInfo xs
  Infix       :: ConstructorName
              -> Associativity -> Fixity
              -> ConstructorInfo '[x, y]
  Record      :: ConstructorName
              -> NP FieldInfo xs
              -> ConstructorInfo xs
```

```
data Associativity =
  LeftAssociative | RightAssociative | NotAssociative
type Fixity = Int
```

Well-Typed

Record field information

```
data FieldInfo (x :: *) = FieldInfo FieldName
```

An extension of the `Generic` class

```
class Generic a => HasDatatypeInfo a where
  datatypeInfo :: Proxy a -> DatatypeInfo (Code a)
```

An extension of the `Generic` class

```
class Generic a => HasDatatypeInfo a where
  datatypeInfo :: Proxy a -> DatatypeInfo (Code a)
```

This can also be generated via Template Haskell or “Generic Generic Programming”.

Example

```
exprInfo :: DatatypeInfo (Code Expr)
exprInfo =
  ADT "LectureNotes" "Expr"
     $  Constructor "NumLit"
     :* Constructor "BoolLit"
     :* Constructor "Add"
     :* Constructor "If"
     :* Nil
instance HasDatatypeInfo Expr where
  datatypeInfo _ = exprInfo
```

Record example

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

```haskell
personInfo :: DatatypeInfo (Code Person)
personInfo =
  ADT "LectureNotes" "Person"
     $  Record "Person"   (  FieldInfo "name"
                          :* FieldInfo "age"
                          :* FieldInfo "address"
                          :* Nil
                          )
    :* Nil
instance HasDatatypeInfo Person where
  datatypeInfo _ = personInfo
```

Useful helper functions

```
constructorInfo :: DatatypeInfo xss -> NP ConstructorInfo xss
constructorInfo (ADT      _ _ i) = i
constructorInfo (Newtype _ _ i) = i :* Nil

constructorName :: ConstructorInfo xs -> ConstructorName
constructorName (Constructor n) = n
constructorName (Infix n _ _)   = n
constructorName (Record n _)    = n

constructorNames :: SListI xss
                 => DatatypeInfo xss -> NP (K ConstructorName) xss
constructorNames = hmap (K . constructorName) . constructorInfo
```

Conclusions

Other applications

Have a look at the existing packages:

- generics-sop
- basic-sop
- pretty-sop
- lens-sop
- json-sop
- postgresql-simple-sop

Summary

- Application of many type-level programming features in GHC.
- Encourages a high-level compositional style.
- Functions are easy to parameterize by type-specific metadata.
- The view provides precise types to all relevant concepts.