Applying Type-Level and Generic Programming in Haskell

Summer School on Generic and Effectful Programming

Andres Löh

6–10 July 2015



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
```



A class



Different constraints

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

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

```
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]
```



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



More general sequencing

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



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.



Testing

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

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



Testing

```
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))))
       fregs
         injections
        'hap' unPOP (hcpure pArbitrary arbitrary))
```

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



A better attempt

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



Computing constructor arities

```
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 \mid n \le 0 = filter ((\le 1) . fst)
                 | otherwise = id
```



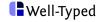
Metadata

Codes cover only structure, not names



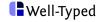
The DatatypeInfo type

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



Constructor information

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



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

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



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.

