# 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.

Today:

- Recap and cover the missing parts of Monday's lecture.
- Introduce *n*-ary sums and the generics-sop view.
- Representing datatypes using generics-sop.
- More combinators and simple applications.

Friday:

- Metadata.
- More applications.

# Recap: NP

# Environments, NP

```
data NP (f :: k -> *) (xs :: [k]) where
  Nil  :: NP f '[]
  (:*) :: f x -> NP f xs -> NP f (x ': xs)
infixr 5 :*
```

# Building and combining environments

```
hpure :: SListI xs => (forall a . f a) -> NP f xs
```

# Building and combining environments

```
hpure :: SListI xs => (forall a . f a) -> NP f xs
```

```
hap :: NP (f -.-> g) xs -> NP f xs -> NP g xs
newtype (f -.-> g) a = Fn {apFn :: f a -> g a}
```

# Mapping and zipping environments

```
hmap :: SListI xs
     => (forall a . f a -> g a)
     -> NP f xs -> NP g xs
hmap f xs = hpure (Fn f) `hap` xs
```

```
hzipWith :: SListI xs
         => (forall a . f a -> g a -> h a)
         -> NP f xs -> NP g xs -> NP h xs
hzipWith f xs ys = hpure (fn_2 f) `hap` xs `hap` ys
```

# Collapsing environments

```
hcollapse :: NP (K a) xs -> [a]
```

```
newtype I a   = I a
newtype K a b = K a
```

# Abstracting from classes, type functions

# Mapping constrained functions?

```
hmap (K . show . unI) group'
```

fails, because

```
K . show . unI ::  forall x . Show x => I x -> K String x
```

does not match

```
                   forall x .             f x -> g        x
```

# Constraints are types of kind `Constraint`

```
GHCi> :kind Eq
Eq :: * -> Constraint
GHCi> :kind Functor
Functor :: (* -> *) -> Constraint
GHCi> :kind MonadReader
MonadReader :: * -> (* -> *) -> Constraint
```

# Constraints are types of kind `Constraint`

```
GHCi> :kind Eq
Eq :: * -> Constraint
GHCi> :kind Functor
Functor :: (* -> *) -> Constraint
GHCi> :kind MonadReader
MonadReader :: * -> (* -> *) -> Constraint
```

Overloaded tuple syntax:

```
type NoConstraint          = (() :: Constraint)
type SomeConstraints a     = (Eq a, Show a)
type MoreConstraints f a = (Monad f, SomeConstraints a)
```

# The `All` type family

```
type family All (c :: k -> Constraint)
                (xs :: [k])
                 :: Constraint where
  All c '[]       = ()
  All c (x ': xs) = (c x, All c xs)
```

# The `All` type family

```
type family All (c :: k -> Constraint)
                (xs :: [k])
                 :: Constraint where
  All c '[]        = ()
  All c (x ': xs) = (c x, All c xs)
```

Example:

```
GHCi> :kind! All Eq '[Int, Bool]
All Eq '[Int, Bool] :: Constraint
 = (Eq Int, (Eq Bool, ()))
```

(Constraints are flattened.)

# Trying to define hcpure

We want:

```
hpure  :: SListI xs
       => (forall a .       f a) -> NP f xs
hcpure :: (SListI xs, All c xs)
       => (forall a . c a => f a) -> NP f xs
```

Then:

```
hcmap :: (SListI xs, All c xs)
      => (forall a . c a => f a -> g a)
      -> NP f xs -> NP g xs
hcmap f xs = hcpure (Fn f) `hap` xs
```

However, this does not work.

# Limitations in GHC's type inference

Assume:

```
hcpure :: (SListI xs, All c xs)
       => (forall a . c a => f a) -> NP f xs
hcpure = undefined
```

Then

```
minBound   :: Bounded a => a
I minBound :: Bounded a => I a
```

```
GHCi> hcpure (I minBound) :: NP I '[Char, Bool]
```

is a type error.

Well-Typed

# Proxies

```haskell
data Proxy (a :: k) = Proxy
```

Examples:

```haskell
pBounded :: Proxy Bounded
pBounded = Proxy

pShow :: Proxy Show
pShow = Proxy
```

# Using proxies to define `hcpure`

```
hcpure :: forall c f xs . (SListI xs, All c xs)
       => Proxy c -> (forall a . c a => f a) -> NP f xs
hcpure p x = case sList :: SList xs of
  SNil  -> Nil
  SCons -> x :* hcpure p x
```

# Using proxies to define `hcpure`

```
hcpure :: forall c f xs . (SListI xs, All c xs)
       => Proxy c -> (forall a . c a => f a) -> NP f xs
hcpure p x = case sList :: SList xs of
  SNil  -> Nil
  SCons -> x :* hcpure p x
```

Example:

```
GHCi> hcpure pBounded (I minBound) :: NP I '[Char, Bool]
 I '\NUL' :* (I False :* Nil)
GHCi> hcpure pShow (Fn (K . show . unI)) `hap` group'
 K "'x'" :* (K "False" :* (K "3" :* Nil))
```

# Constrained `hmap`

```
hcmap :: (SListI xs, All c xs)
      => Proxy c
      -> (forall a . c a => f a -> g a)
      -> NP f xs -> NP g xs
hcmap p f xs = hcpure p (Fn f) `hap` xs
```

# Generalizing choice

# Choosing from a list

```
data LChoice a = LCZero a | LCSuc (LChoice a)
```

An index into a list paired with the element at that position.

# Choosing from a list

```
data LChoice a = LCZero a | LCSuc (LChoice a)
```

An index into a list paired with the element at that position.

Equivalently in GADT syntax:

```
data LChoice (a :: *) where
  LCZero :: a -> LChoice a
  LCSuc  :: LChoice a -> LChoice a
```

# Choosing from a vector

```
data LChoice (a :: *) where
  LCZero :: a -> LChoice a
  LCSuc  :: LChoice a -> LChoice a
```

# Choosing from a heterogeneous list

```
data VChoice (a :: *) (n :: Nat) where
  VCZero :: a -> VChoice a (Suc n)
  VCSuc  :: VChoice a n -> VChoice a (Suc n)
```

```
data HChoice (xs :: [*]) where
  HCZero :: x -> HChoice (x ': xs)
  HCSuc  :: HChoice xs -> HChoice (x ': xs)
```

# Choosing from an environment

```
data VChoice (a :: *) (n :: Nat) where
  VCZero :: a -> VChoice a (Suc n)
  VCSuc  :: VChoice a n -> VChoice a (Suc n)

data HChoice (xs :: [*]) where
  HCZero :: x -> HChoice (x ': xs)
  HCSuc  :: HChoice xs -> HChoice (x ': xs)
```

```
data NS (f :: k -> *) (xs :: [k]) where
  Z :: f x -> NS f (x ': xs)
  S :: NS f xs -> NS f (x ': xs)
```

# An example

```
data NS (f :: k -> *) (xs :: [k]) where
  Z :: f x -> NS f (x ': xs)
  S :: NS f xs -> NS f (x ': xs)
```

```
type ExampleChoice = NS I '[Char, Bool, Int]
```

```
c0, c1, c2 :: ExampleChoice
c0 = Z (I 'x')
c1 = S (Z (I True))
c2 = S (S (Z (I 3)))
```

# Representing types as sums of products

# Representing expressions

```
data Expr = NumL Int
          | BoolL Bool
          | Add Expr Expr
          | If Expr Expr Expr
```

Values are of form:

$C_i \; x_0 \ldots x_{n_i - 1}$

# Representing expressions – contd.

```haskell
data Expr = NumL Int
          | BoolL Bool
          | Add Expr Expr
          | If Expr Expr Expr
```

```haskell
type RepExpr = NS (NP I) (’[ ’[Int]
                           , ’[Bool]
                           , ’[Expr, Expr]
                           , ’[Expr, Expr, Expr]
                           ])
```

# Representing expressions – example

```
exampleExpr :: Expr
exampleExpr = If (BoolL True) (NumL 1) (NumL 0)
```

```
exampleRepExpr :: RepExpr
exampleRepExpr =
  S (S (S (Z (   I (BoolL True)
              :* I (NumL 1)
              :* I (NumL 0)
              :* Nil))))
```

# Representing expressions – example

```
exampleExpr :: Expr
exampleExpr = If (BoolL True) (NumL 1) (NumL 0)
```

```
exampleRepExpr :: RepExpr
exampleRepExpr =
  S (S (S (Z (   I (BoolL True)
              :* I (NumL 1)
              :* I (NumL 0)
              :* Nil))))
```

Beautified syntax:

```
exampleRepExpr :: RepExpr
exampleRepExpr =
  C₃ [I (BoolL True), I (NumL 1), I (NumL 0)]
```

Well-Typed

# Converting between `Expr` and `RepExpr`

```
fromExpr :: Expr -> RepExpr
fromExpr (NumL n)      =
  Z          (I n :* Nil)
fromExpr (BoolL b)     =
  S (Z       (I b :* Nil))
fromExpr (Add e1 e2)   =
  S (S (Z    (I e1 :* I e2 :* Nil)))
fromExpr (If e1 e2 e3) =
  S (S (S (Z (I e1 :* I e2 :* I e3 :* Nil))))
```

Similarly `toExpr`.

# The `Generic` class

Monday:

```
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

# The Generic class

Monday:

```
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

Wednesday:

```
class (SListI (Code a), All SListI (Code a))
      => Generic (a :: *) where
  type Code a :: [[*]]
  from :: a -> Rep a
  to   :: Rep a -> a

newtype SOP f a = SOP (NS (NP f) a)
type Rep a = SOP I (Code a)
```

# Instance for expressions

```
instance Generic Expr where
  type Code Expr = ' [ '[Int]
                     , '[Bool]
                     , '[Expr, Expr]
                     , '[Expr, Expr, Expr]
                     ]
  from x     = SOP (fromExpr x)
  to (SOP x) = toExpr x
```

# Instance for lists

```
instance Generic [a] where
  type Code [a] = '['[], '[a, [a]]]
  from []        = SOP (Z    Nil)
  from (x : xs)  = SOP (S (Z (I x :* I xs :* Nil)))
  to (SOP (Z    Nil))                    = []
  to (SOP (S (Z (I x :* I xs :* Nil)))) = x : xs
```

Shows how parameters are handled.

# On defining `Generic` instances

# The role of `Generic`

If a type is an instance of `Generic`, then lots of generic functions will be available for that type.

However, the `Generic` instance must still be written.

# Options

- Define by hand.
- Use Template Haskell.
- Extend the compiler (GHC).
- Use “Generic Generic Programming”.

# Using Template Haskell

```haskell
data Expr = ...
deriveGeneric ''Expr
```

This is implemented.

# Direct compiler support

```
data Expr = ...
  deriving (..., Generic)
```

This is not implemented (and unlikely to be).

# On compiler support

GHC has first-class support for (at least) two approaches:

- Scrap your boilerplate (SYB) via the `Data` class
- Generic deriving via (another) `Generic` class

In essence, `Data` and `Generic` are different structural representations of datatypes, encoding similar or even the same information as we are trying to.

# On compiler support

GHC has first-class support for (at least) two approaches:

- Scrap your boilerplate (SYB) via the `Data` class
- Generic deriving via (another) `Generic` class

In essence, `Data` and `Generic` are different structural representations of datatypes, encoding similar or even the same information as we are trying to.

What if we could translate one representation into the other?

# Generic Generic Programming

- Use Haskell libraries to translate between different representations.
- Built-in representation can aim to be as informative as possible; no need for it to be (directly) practical.
- Makes it easier to use several approaches in a single program.
- Encourages specialized representations for specific domains, rather than trying to find the “one true generic programming approach”.

# Generic Generic Programming in practice

```haskell
import qualified GHC.Generics as GHC

data Expr = ...
  deriving (..., GHC.Generic)

instance Generic Expr
```

This is implemented.

# Generic Generic Programming in practice

```
import qualified GHC.Generics as GHC

data Expr = ...
  deriving (..., GHC.Generic)

instance Generic Expr
```

This is implemented.

In the future:

```
data Expr = ...
  deriving (..., GHC.Generic, Generic)
```

# Generic equality

# Equality for products

```
geqNP :: All Eq xs => NP I xs -> NP I xs -> Bool
geqNP Nil            Nil            = True
geqNP (I x1 :* xs1) (I x2 :* xs2) = x1 == x2 && geqNP xs1 x
```

We are going to fall back on the `Eq` class for the components of the product.

# Equality for sums of products

```
geqNS :: ... => NS (NP I) xss -> NS (NP I) xss -> Bool
geqNS (Z np1) (Z np2) = geqNP np1 np2
geqNS (S ns1) (S ns2) = geqNS ns1 ns2
geqNS _       _       = False
```

# Equality for sums of products

```
geqNS :: ... => NS (NP I) xss -> NS (NP I) xss -> Bool
geqNS (Z np1) (Z np2) = geqNP np1 np2
geqNS (S ns1) (S ns2) = geqNS ns1 ns2
geqNS _       _       = False
```

We need `All (All Eq) xss` – but we can't do that.

# From `All` to `All2`

To avoid partial applications, we have to “copy” the `All` family for two-dimensional structures:

```
type family All (c :: k -> Constraint)
                (xs :: [k])
                 :: Constraint where
  All c '[]       = ()
  All c (x ': xs) = (c x, All c xs)
```

```
type family All2 (f :: k -> Constraint)
                 (xss :: [[k]])
                  :: Constraint where
  All2 f '[]          = ()
  All2 f (xs ': xss) = (All f xs, All2 f xss)
```

# Equality for sums of products – contd.

```
geqNS :: All2 Eq xss
      => NS (NP I) xss -> NS (NP I) xss -> Bool
geqNS (Z np1) (Z np2) = geqNP np1 np2
geqNS (S ns1) (S ns2) = geqNS ns1 ns2
geqNS _       _       = False
```

# Completing the definition

```
geqSOP :: All2 Eq xss => SOP I xss -> SOP I xss -> Bool
geqSOP (SOP sop1) (SOP sop2) = geqNS sop1 sop2
```

# Completing the definition

```haskell
geqSOP :: All2 Eq xss => SOP I xss -> SOP I xss -> Bool
geqSOP (SOP sop1) (SOP sop2) = geqNS sop1 sop2
```

```haskell
geq :: (Generic a, All2 Eq (Code a)) => a -> a -> Bool
geq x y = geqSOP (from x) (from y)
```

# Using generic equality

```
instance Eq Expr where
  (==) = geq
```

# Using generic equality

```
instance Eq Expr where
  (==) = geq
```

Example:

```
GHCi> geq (Add (NumL 3) (NumL 5)) (Add (NumL 3) (NumL 5))
 True
GHCi> geq (Add (NumL 3) (NumL 5)) (Add (NumL 3) (NumL 6))
 False
```

# Another look at the product case

```
geqNP :: All Eq xs => NP I xs -> NP I xs -> Bool
geqNP Nil              Nil              = True
geqNP (I x1 :* xs1) (I x2 :* xs2) = x1 == x2 && geqNP xs1 x
```

Well-Typed

# Another look at the product case

```
geqNP :: All Eq xs => NP I xs -> NP I xs -> Bool
geqNP Nil           Nil           = True
geqNP (I x1 :* xs1) (I x2 :* xs2) = x1 == x2 && geqNP xs1 x
```

It's a `hczipWith`!

# Redefining the product case

```haskell
geqNP' :: (SListI xs, All Eq xs)
       => NP I xs -> NP I xs -> Bool
geqNP' xs ys = and (hcollapse (hczipWith p aux xs ys))
  where
    aux (I x) (I y) = K (x == y)
    p = Proxy :: Proxy Eq
```

# Generic producers

# Default values

Provided by the data-default package:

```
class Default a where
  def :: a
```

# Default values

Provided by the data-default package:

```haskell
class Default a where
  def :: a
```

We will try to define:

```haskell
gdef :: (Generic a,
         Code a ~ (xs ': xss),
         All Default xs) => a
gdef = to (SOP gdefNS)
```

Note the rather precise type.

# Completing the definition

```
gdefNS :: forall xs xss . (SListI xs, All Default xs)
       => NS (NP I) (xs ': xss)
gdefNS = case sList :: SList xs of
  SCons -> Z (hcpure (Proxy :: Proxy Default) (I def))
```

Will generate

```
C₀ [def, ...]
```

# Using the function

```
instance Default Int where
  def = 0
instance Default Bool where
  def = False
```

```
GHCi> gdef :: Expr
 NumL 0
```

# Using default signatures

```
class Default a where
  def :: a
  default def :: (Generic a,
                  Code a ~ (xs ': xss),
                  All Default xs) => a
  def = gdef
```

# Using default signatures

```
class Default a where
  def :: a
  default def :: (Generic a,
                  Code a ~ (xs ': xss),
                  All Default xs) => a
  def = gdef
```

Then:

```
instance Default Expr
```

Or in the near future:

```
data Expr = ...
  deriving (..., Default)
```

# Generating one value for each constructor

```
gdefAll :: (Generic a, All2 Default (Code a)) => [a]
gdefAll = map (to . SOP) gdefAllNS
```

Well-Typed

# Generating one value for each constructor

```
gdefAll :: (Generic a, All2 Default (Code a)) => [a]
gdefAll = map (to . SOP) gdefAllNS
```

```
gdefAllNS :: forall xss .
             (SListI xss, All SListI xss,
              All2 Default xss)
          => [NS (NP I) xss]
gdefAllNS = case sList :: SList xss of
  SNil  -> []
  SCons -> Z (hcpure (Proxy :: Proxy Default) (I def))
         : map S gdefAllNS
```

# Example use of `gdefAll`

```
GHCi> gdefAll :: [Expr]
[NumL 0,
 BoolL False,
 Add (NumL 0) (NumL 0),
 If (NumL 0) (NumL 0) (NumL 0)]
```

# Products of products, injections

# Another idea

A table of recursive calls:

```
[[def], [def], [def, def], [def, def, def]]
```

# Another idea

A table of recursive calls:

```
[[def], [def], [def, def], [def, def, def]]
```

A list of constructor functions:

$$[C_0, C_1, C_2, C_3]$$

# Another idea

A table of recursive calls:

```
[[def], [def], [def, def], [def, def, def]]
```

A list of constructor functions:

$[C_0, C_1, C_2, C_3]$

Zip:

$[C_0\ [\mathrm{def}], C_1\ [\mathrm{def}], C_2\ [\mathrm{def}, \mathrm{def}], C_3\ [\mathrm{def}, \mathrm{def}, \mathrm{def}]]$

# Products of products

```
newtype POP f a = POP {unPOP :: NP (NP f) a}
```

# An `hcpure` for `POP`

```
hcpure_POP :: forall c f xss .
                (SListI xss, All SListI xss, All2 c xss)
             => Proxy c -> (forall a . c a => f a)
             -> POP f xss
hcpure_POP p x = POP (case sList :: SList xss of
  SNil  -> Nil
  SCons -> hcpure p x :* unPOP (hcpure_POP p x))
```

Unfortunately, because

```
All2 c xss /= All (All c) xss
```

we cannot just call `hcpure` twice.

# Using `hcpure_POP` to generate the table

```
hcpure_POP (Proxy :: Proxy Default) (I def)
   :: POP I (Code Expr)
```

yields

```
[ [I def]
, [I def]
, [I def, I def]
, [I def, I def, I def]
]
```

# Injections for `Either`

```
data Either a b = Left a | Right b
```

```
Left  :: a -> Either a b
Right :: b -> Either a b
```

# Injections for `Either`

```
data Either a b = Left a | Right b
```

```
Left  :: a -> Either a b
Right :: b -> Either a b
```

```
eitherInjections :: NP (I -.-> K (Either a b)) '[a, b]
eitherInjections  =  Fn (K . Left  . unI)
                  :* Fn (K . Right . unI)
                  :* Nil
```

# Injections for `NS`

Assume `xs = '[x, y, z, ...]`:

```
Z         :: f x -> NS f xs
S . Z     :: f y -> NS f xs
S . S . Z :: f z -> NS f xs
```

# Injections for NS

Assume `xs = '[x, y, z, ...]`:

```
Z         :: f x -> NS f xs
S . Z     :: f y -> NS f xs
S . S . Z :: f z -> NS f xs
```

```
injections ::  forall xs f . SListI xs
           => NP (f -.-> K (NS f xs)) xs
injections = case sList :: SList xs of
  SNil  -> Nil
  SCons -> Fn (K . Z)
        :* hmap (Fn . ((K . S . unK) .) . apFn) injections
```

Well-Typed

# Putting things together

```
apInjs :: SListI xs => NP f xs -> [NS f xs]
apInjs np = hcollapse (injections `hap` np)
```

```
apInjs_POP :: SListI xs => POP f xs -> [SOP f xs]
apInjs_POP (POP pop) = map SOP (apInjs pop)
```

# Putting things together

```
apInjs :: SListI xs => NP f xs -> [NS f xs]
apInjs np = hcollapse (injections ‘hap‘ np)
```

```
apInjs_POP :: SListI xs => POP f xs -> [SOP f xs]
apInjs_POP (POP pop) = map SOP (apInjs pop)
```

```
gdefAll’ :: (Generic a, All2 Default (Code a)) => [a]
gdefAll’ =
  map to (apInjs_POP (hcpure_POP p (I def)))
  where
    p = Proxy :: Proxy Default
```

# Summary

- Type `a` is represented as `NS (NP I) (Code a)`.
- Generic functions can be defined by pattern matching on `NS` and `NP`, but also by combining general combinators.

Well-Typed

# Exercises

1. Generalize equality to comparison.
2. Define a variant of generic equality (or comparison) that ignores some constructor arguments.
3. Define generic enumeration.