Deriving Via

IFIP WG 2.1 meeting #77, Brandenburg

Andres Löh, joint work with Ryan Scott and Baldur Blöndal

2018-07-02
Can streams be treated as numbers?

```haskell
data Stream a = a -> Stream a
```
Can streams be treated as numbers?

```haskell
data Stream a = a => Stream a

class Num a where
  (+) :: a => a => a
  (\-) :: a => a => a
  (*) :: a => a => a
  negate :: a => a
  abs :: a => a
  signum :: a => a
  fromInteger :: Integer => a
```
instance Num a => Num (Stream a) where
  (x :> xs) + (y :> ys) = x + y :> xs + ys
...
  negate (x :> xs) = negate x :> negate xs
...
  fromInteger i = fromInteger i :> fromInteger i
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

instance Functor Stream where
  fmap f (x :> xs) = f x :> fmap f xs

instance Applicative Stream where
  pure x = x :> pure x
  (f :> fs) <*> (x :> xs) = f x :> (fs <*> xs)
```

Well-Typed
```haskell
instance Num a => Num (Stream a) where
  xs + ys = pure (+) <*> xs <*> ys
  xs - ys = pure (-) <*> xs <*> ys
  xs * ys = pure (*) <*> xs <*> ys
  negate xs = pure negate <*> xs
  abs xs = pure abs <*> xs
  signum xs = pure signum <*> xs
  fromInteger i = pure (fromInteger i)
```
instance Num a => Num (Maybe a) where
  mx + my = pure (+) <*> mx <*> my
  mx - my = pure (-) <*> mx <*> my
  mx * my = pure (*) <*> mx <*> my
  negate mx = pure negate <*> mx
  abs mx = pure abs <*> mx
  signum mx = pure signum <*> mx
  fromInteger i = pure (fromInteger i)
data Stream a = a :> Stream a
  deriving Num via (LiftApplicative Stream a)

data Maybe a = Nothing | Just a
  deriving Num via (LiftApplicative Maybe a)
Goal of **deriving** via

```haskell
data Stream a = a :> Stream a
  deriving Num via (LiftApplicative Stream a)
```

```haskell
data Maybe a = Nothing | Just a
  deriving Num via (LiftApplicative Maybe a)
```

- Allow defining and naming **instance rules** such as `LiftApplicative`.
- Allow instantiating rules in **deriving** clauses.
Instance rules
An approach that does not work

```haskell
instance (Num a, Applicative f) => Num (f a) where
  x + y = pure (+) <*> x <*> y
  x - y = pure (-) <*> x <*> y
  x * y = pure (*) <*> x <*> y
  negate x = pure negate <*> x
  abs x = pure abs <*> x
  signum x = pure signum <*> x
  fromInteger i = pure (fromInteger i)
```

▶ Allows defining, but not naming the rule.
▶ Overlaps with too many instances.
An approach that does not work

```haskell
instance (Num a, Applicative f) => Num (f a) where
    x + y = pure (+) <*> x <*> y
    x - y = pure (-) <*> x <*> y
    x * y = pure (*) <*> x <*> y
    negate x = pure negate <*> x
    abs x = pure abs <*> x
    signum x = pure signum <*> x
    fromInteger i = pure (fromInteger i)
```

- Allows **defining**, but not **naming** the rule.
- Overlaps with too many instances.
Use newtypes

newtype LiftApplicative f a = LA (f a)
Use newtypes

newtype LiftApplicative f a = LA (f a)

instance (Num a, Applicative f) => Num (LiftApplicative f a) where
  LA x + LA y = LA (pure (+) <*> x <*> y)
  LA x - LA y = LA (pure (-) <*> x <*> y)
  LA x * LA y = LA (pure (*) <*> x <*> y)
  negate (LA x) = LA (pure negate <*> x)
  abs (LA x)   = LA (pure abs <*> x)
  signum (LA x) = LA (pure signum <*> x)
  fromInteger i = LA (pure (fromInteger i))

Defines and names the rule.
data Stream a = a => Stream a
  deriving Num via (LiftApplicative Stream a)

data Maybe a = Nothing | Just a
  deriving Num via (LiftApplicative Maybe a)

Explicitly instantiates the rule.
Instantiating the rule

```hs
data Stream a = a :> Stream a
  deriving Num via (LiftApplicative Stream a)
```

```hs
data Maybe a = Nothing | Just a
  deriving Num via (LiftApplicative Maybe a)
```

Explicitly instantiates the rule.

Still need **Functor** and **Applicative** instances defined somewhere.
How does it work?
A **newtype** is a datatype with

- exactly one constructor
- of exactly one argument.
A **newtype** is a datatype with

- exactly one constructor
- of exactly one argument.

Wrapped and wrapper types are guaranteed to be **representationally equal**.
The Coercible constraint is built-in. "Instances" are provided (only) by the compiler.

\[
\text{coerce} :: \text{Coercible } a \; b \Rightarrow a \to b
\]
Witnessing representational equality

\[
\text{coerce} :: \text{Coercible } a \ b \Rightarrow a \rightarrow b
\]

The \text{Coercible} constraint is built-in.

“Instances” are provided (only) by the compiler.
newtype Amount = MkAmount Rational
newtype Amount = MkAmount Rational

Compiler knows:

Coercible Rational Amount
Coercible Amount Rational
Newtypes and type constructors

Compiler also knows:

\[
\text{Coercible } a \ b \Rightarrow \text{Coercible } [a] \ [b] \\
\text{Coercible } a \ b \Rightarrow \text{Coercible } (\text{IO } a) \ (\text{IO } b) \\
(\text{Coercible } a \ c, \text{Coercible } b \ d) \\
\Rightarrow \text{Coercible } (a, b) \ (c, d) \\
(\text{Coercible } a \ c, \text{Coercible } b \ d) \\
\Rightarrow \text{Coercible } (a \rightarrow b) \ (c \rightarrow d)
\]

Consequence: We can lift coerce through most types.
Compiler also knows:

\[ \text{Coercible } a \ b \Rightarrow \text{Coercible } [a] \ [b] \]
\[ \text{Coercible } a \ b \Rightarrow \text{Coercible } (\text{IO } a) \ (\text{IO } b) \]
\[ (\text{Coercible } a \ c, \text{Coercible } b \ d) \Rightarrow \text{Coercible } (a, b) \ (c, d) \]
\[ (\text{Coercible } a \ c, \text{Coercible } b \ d) \Rightarrow \text{Coercible } (a \rightarrow b) \ (c \rightarrow d) \]

Consequence: We can lift coerce through most types.
The situation

data Stream a = a => Stream a
  deriving Num via (LiftApplicative Stream a)
The situation

```haskell
data Stream a = a -> Stream a
  deriving Num via (LiftApplicative Stream a)

We have:

instance Num a => Num (LiftApplicative Stream a)
```
The situation

```haskell
data Stream a = a :> Stream a
  deriving Num via (LiftApplicative Stream a)
```

We have:

```haskell
instance Num a => Num (LiftApplicative Stream a)
```

We want:

```haskell
instance Num a => Num (Stream a)
```
The situation

\[
\text{data Stream } a = a :> \text{ Stream } a \\
\text{ deriving Num via } (\text{LiftApplicative Stream } a)
\]

We have:

\[
\text{instance Num } a \Rightarrow \text{ Num } (\text{LiftApplicative Stream } a)
\]

We want:

\[
\text{instance Num } a \Rightarrow \text{ Num } (\text{Stream } a)
\]

We know:

\[
\text{Coercible } (\text{LiftApplicative Stream } a) (\text{Stream } a)
\]
instance Num a => Num (Stream a) where

(+) =
    coerce (+) @(LiftApplicative Stream a))

(-) =
    coerce (-) @(LiftApplicative Stream a))

(*) =
    coerce (*) @(LiftApplicative Stream a))

negate =
    coerce (negate @(LiftApplicative Stream a))

abs =
    coerce (abs @(LiftApplicative Stream a))

signum =
    coerce (signum @(LiftApplicative Stream a))

fromInteger =
    coerce (fromInteger @(LiftApplicative Stream a))
Library writer is encouraged to write down and name instance rules.

- End user can use existing rules to avoid boilerplate.
- Very lightweight addition to GHC. Implemented in GHC 8.6.
Generalisations and interactions
newtype Amount = MkAmount Rational

deriving (Num, Fractional, Eq, Enum, Ord, Show)

via Rational
Monads are applicative functors

```haskell
newtype FromMonad m a = FM (m a)

instance Monad m => Functor (FromMonad m) where
  fmap f (FM m) = FM (m >>= return . f)

instance Monad m => Applicative (FromMonad m) where
  pure a = FM (return a)
  FM f <*> FM x = FM (f >>= \rf -> x >>= \rx -> return (rf rx))
```
Defining monads becomes almost as easy as pre-AMP

data Maybe a = Nothing | Just a
    deriving (Functor, Applicative) via (FromMonad Maybe)

instance Monad Maybe where
    return     = Just
    Just m    >>= k = k m
    Nothing    >>= _ = Nothing
newtype FromOrd a = FO a

instance Ord a => Eq (FromOrd a) where
  FO x == FO y =
    case compare x y of
      EQ -> True
      _   -> False
newtype FromOrd a = FO a

instance Ord a => Eq (FromOrd a) where
  FO x == FO y =
    case compare x y of
    EQ -> True
    _   -> False

data TaggedWith a b = TW {tag :: a, item :: b}
  deriving Eq via (FromOrd (TaggedWith a b))

instance Ord b => Ord (TaggedWith a b) where
  compare x y = compare (item x) (item y)
newtype Parser a = P (String -> [(a, String)])
newtype Parser a = P (String -> [(a, String)])
deriving (Functor, Applicative, Monad, MonadState String)
via (StateT String []

Well-Typed
newtype Parser a = P (String -> [(a, String)])
  deriving (Functor, Applicative, Monad, MonadState String)
  via (StateT String [])

newtype StateT s m a = 
  StateT {runStateT :: s -> m (a, s)}

instance Functor m => Functor (StateT s m)
instance (Functor m, Monad m) => Applicative (StateT s m)
instance Monad m => Monad (StateT s m)
Custom enumeration types
data Weekday =
  Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
| Sunday
deriving (GHC.Generic, SOP.Generic)
class IsEnumType a => CustomEnum a where
names :: NP (K String) (Code a)
class IsEnumType a => CustomEnum a where
  names :: NP (K String) (Code a)

instance CustomEnum Weekday where
  names =
    K "Mo"
    :* K "Di"
    :* K "Mi"
    :* K "Do"
    :* K "Fr"
    :* K "Sa"
    :* K "So"
    :* Nil
newtype FromCustomEnum a = FCE a

instance CustomEnum a => Read (FromCustomEnum a)
instance CustomEnum a => Show (FromCustomEnum a)
instance CustomEnum a => Enum (FromCustomEnum a)
data Weekday =
    Monday
    | Tuesday
    | Wednesday
    | Thursday
    | Friday
    | Saturday
    | Sunday

deriving (GHC.Generic, SOP.Generic)
deriving (Read, Show, Enum) via (FromCustomEnum Weekday)
Example: getters
Getting one type from another

```haskell
newtype Getting a b = G b

class HasGetting a b where
    getting :: b -> a
```

Well-Typed
newtype Getting a b = G b

class HasGetting a b where
  getting :: b -> a

instance (HasGetting a b, Eq a) => Eq (Getting a b) where
  G x == G y = getting @a x == getting @a y

instance (HasGetting a b, Ord a) => Ord (Getting a b) where
  compare (G x) (G y) =
    compare (getting @a x) (getting @a y)

instance (HasGetting a b, Show a) => Show (Getting a b) where
  showsPrec prec (G x) =
    showsPrec prec (getting @a x)
Example

```haskell
data TaggedWith a b = TW {tag :: a, item :: b}
deriving (Eq, Ord) via (Getting b (TaggedWith a b))
instance HasGetting b (TaggedWith a b) where
    getting = item
```
Examples in the paper

- QuickCheck modifiers
- representable functors
- default signatures
- same / similar generic representation
Conclusions

- Simple extension
- Generalises many features
- Compositional and configurable
- Encourages code reuse and more high-level programming