The Brave New World of Haskell Type Classes

František Farka

June 3, 2015

University of Dundee, and
University of St Andrews

What are we going to talk about

There are certain shortcomings of Haskell type class system

What are we going to talk about

There are certain shortcomings of Haskell type class system and there are also means to overcome these:

- Default superclass instances
- Class aliases

What are we going to talk about

There are certain shortcomings of Haskell type class system and there are also means to overcome these:

- Default superclass instances
- Class aliases

and simplify the work with type classes.

Default Superclass Instances

```
min :: (Ord a) ⇒ a → a → a
min a b = if a < b then a else b
```

Default Superclass Instances

```
min :: (Ord a) ⇒ a → a → a
min a b = if a < b then a else b
```

```
-- | Linear order

class             Ord a where
    (<)  ::  a → a → Bool
    (≤)  ::  a → a → Bool
```

Default Superclass Instances

```
min ::  (Ord a) ⇒ a → a → a
min a b = if a < b then a else b
```

```
-- | Linear order

class           Ord a where
    (<) ::  a → a → Bool
    (≤) ::  a → a → Bool
```

```
instance Ord OurData where
    (<) = ...
    (≤) = ...
```

Default Superclass Instances

```
min :: (Ord a) ⇒ a → a → a
min a b = if a < b then a else b

-- | Equality
class Eq a where
    (≡) :: a → a → Bool

-- | Linear order

class (Eq a) ⇒ Ord a where
    (<) :: a → a → Bool
    (≤) :: a → a → Bool



instance Ord OurData where
    (<) = ...
    (≤) = ...
```

Default Superclass Instances

```
min ::  (Ord a) ⇒ a → a → a
min a b = if a < b then a else b

-- | Equality
class Eq a where
    (≡) ::  a → a → Bool

-- | Linear order
-- * a ≡ b = ¬(a < b || b < a)
class (Eq a) ⇒ Ord a where
    (<) ::  a → a → Bool
    (≤) ::  a → a → Bool



instance Ord OurData where
    (<) = ...
    (≤) = ...
```

```
min ::  (Ord a) ⇒ a → a → a
min a b = if a < b then a else b

-- | Equality
class Eq a where
    (≡) ::  a → a → Bool

-- | Linear order
-- * a ≡ b = ¬(a < b || b < a)
class (Eq a) ⇒ Ord a where
    (<) ::  a → a → Bool
    (≤) ::  a → a → Bool
    default instance Eq a where
        a ≡ b = ¬(a < b || b < a)

instance Ord OurData where
    (<) = ...
    (≤) = ...
```

Class Aliases

```
class Additive r where
    (+) :: r → r → r
class Multiplicative r where
    (*) :: r → r → r
```

```
class Additive r where
    (+) :: r → r → r
class Multiplicative r where
    (∗) :: r → r → r

class (Additive r, Multiplicative r)
    ⇒ Num r where
```

Class Aliases

```
class Additive r where
    (+) :: r → r → r
class Multiplicative r where
    (∗) :: r → r → r

class (Additive r, Multiplicative r)
    ⇒ Num r where

instance Applicative OurData where
     a + b = ...
instance Multiplicative OurData where
     a ∗ b = ...
```

Class Aliases'

class Additive r **where**
    $(+) :: r \to r \to r$
class Multiplicative r **where**
    $(*) :: r \to r \to r$

Class Aliases'

```
class Additive r where
    (+) :: r → r → r
class Multiplicative r where
    (*) :: r → r → r

class (Additive r, Multiplicative r)
    ⇒ Num r where
    add :: r → r → r


    mul :: r → r → r
```

Class Aliases'

```
class Additive r where
    (+) :: r → r → r
class Multiplicative r where
    (∗) :: r → r → r

class (Additive r, Multiplicative r)
    ⇒ Num r where
    add :: r → r → r
    default instance Additive a where
        a + b = add a b
    mul :: r → r → r
    default instance Multiplicative a where
        a ∗ b = mul a b

instance Num OurData where
     add a b = ...
     mul a b = ...
```

Class Aliases"

```
class Additive r where
    (+) :: r → r → r
class Multiplicative r where
    (*) :: r → r → r
```

Class Aliases"

```
class Additive r where
    (+) :: r → r → r
class Multiplicative r where
    (∗) :: r → r → r

class alias Num r = (Additive r, Multiplicative r)
```

Class Aliases”

```
class Additive r where
    (+) :: r → r → r
class Multiplicative r where
    (*) :: r → r → r

class alias Num r = (Additive r, Multiplicative r)

instance Num OurData where
    a + b = ...
    a * b = ...
```

What are these good for?

We have the two extensions to the Haskell type class system.
What are the applications?

Monoid

```
class Monoid a where
    mappend :: a → a → a
    mempty :: a
```

Monoid

```
class                   Monoid a where
    mappend :: a → a → a
    mempty :: a
```

```
– Lift a Semigroup into Maybe forming a Monoid.
instance Monoid a                  → Monoid (Maybe a)
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Monoid

```
class Semigroup a where
    (<>) :: a

class Semigroup a ⇒ Monoid a where
    mappend :: a → a → a
    mempty :: a
```

```
– Lift a Semigroup into Maybe forming a Monoid.
instance            Semigroup a → Monoid (Maybe a)
    mempty = Nothing
    Nothing ‘mappend‘ m = m
    m ‘mappend‘ Nothing = m
    Just m1 ‘mappend‘ Just m2 = Just (m1 ‘mappend‘ m2)
```

Monoid

```
class Semigroup a where
    (<>) :: a

class Semigroup a ⇒ Monoid a where
    mappend :: a → a → a
    mempty :: a
    default instance Semigroup a where
        (<>) = mappend

– Lift a Semigroup into Maybe forming a Monoid.
instance        Semigroup a → Monoid (Maybe a)
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Pointed and Bind

```haskell
class Functor f where
  fmap :: (a → b) → f a → f b
```

```haskell
class Pointed p where
  pure :: a → p a
```

```haskell
class Functor f ⇒ Apply f where
  (<*>) :: f (a → b) → f a → f b
```

```haskell
class (Apply f, Pointed f)
  ⇒ Applicative f where
```

```haskell
class Apply m ⇒ Bind m where
  (>>=) :: m a → (a → m b) → m b
```

```haskell
class Monad m where
```

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

```
class Pointed p where
  pure :: a → p a
```

```
class Functor f ⇒ Apply f where
  (<*>) :: f (a → b) → f a → f b
```

```
class alias Applicative =
  (Apply f, Pointed f)
```

```
class Apply m ⇒ Bind m where
  (>>=) :: m a → (a → m b) → m b
```

```
class alias Monad m =
  (Applicative, Bind)
```

Pointed and Bind - aliases

$$\text{fmap } f \ x = \text{pure } f <\!*\!> x \tag{1}$$

$$\text{pure } x = \text{return } x \tag{2}$$

$$\text{fmap } f \ x = x \gg\!= \text{return} \circ f \tag{3}$$

Pointed and Bind - aliases

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

```
class Pointed p where
  pure, return :: a → p a
```

```
class Functor f ⇒ Apply f where
  (<*>) :: f (a → b) → f a → f b
```

```
class alias Applicative = (Apply f,
  Pointed f, Applicative_)
```

```
class Apply m ⇒ Bind m where
  (>>=) :: m a → (a → m b) → m b
```

```
class alias Monad m = (Applicative m,
  Bind m, Monad_ m)
```

Pointed and Bind - aliases

```
class (Apply f, Pointed f) ⇒ Applicative_ f where
    default instance Functor f where
        fmap f x = pure f <*> x

class (Bind m, Applicative m) ⇒ Monad_ m where
    default instance Functor f where
        fmap f x = x >>= return ∘ f
    default instance Apply f where
        pf <*> px = px >>= λ x → pf >>= λ f →
                        return (f x)
```

Conclusion

We described a novel approach to class hierarchy modifications in Haskell. With the *Default Superclass Instances* and *Class Aliases* it is possible to:

- change type class hierarchies in a backward compatible way;
- provide some instances automatically, this holds for general type classes, no per-class compiler support is necessary; and
- it is no longer problem to provide fine-grained type class hierarchies—simpler interface to hierarchy can be provided.

Future work

- Extend the *Superclass Default Instances* to *Multi-Parameter Type Classes*
- Explore the interaction of *Superclass Default Instances* and *Class Aliases* with *Qualified Contexts*
- More remote: Explore the composability of type class instances in general. Improve type class instances resolution for co-inductive data structures.

Thank you. Questions?