Maintainable type classes for Haskell

František Farka

March 25, 2015

State of affairs ...

- *GHC 7.8.4* (latest) released Dec 23, 2014

State of affairs ...

- *GHC 7.8.4* (latest) released Dec 23, 2014
- *GHC 7.10.1* final release scheduled for Mar 20, 2015

State of affairs ...

- *GHC 7.8.4* (latest) released Dec 23, 2014
- *GHC 7.10.1* final release scheduled for Mar 20, 2015
    - Functor–Applicative–Monad proposal[5, 1, 6]

State of affairs ...

- *GHC 7.8.4* (latest) released Dec 23, 2014
- *GHC 7.10.1* final release scheduled for Mar 20, 2015
  - Functor–Applicative–Monad proposal[5, 1, 6]
  - Prelude 7.10 - Plan FTP [2]

Motivation

Library.hs

```
module Library where




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

Motivation

Library.hs

```
module Library where



class          Ord’ a where
  (≤′) ::  a → a → Bool
```

Client.hs

```
import Library

data Foo = ...



instance Ord’ Foo where
  (≤′) = ...
```

Motivation

Library.hs

```
module Library where

class Eq' a where
  (≡') ::  a → a → Bool

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

Client.hs

```
import Library

data Foo = ...




instance Ord' Foo where
  (≤') = ...
```

Motivation

Library.hs

```
module Library where

class Eq’ a where
  (≡′) ::  a → a → Bool

class Eq’ a ⇒ Ord’ a where
  (≤′) ::  a → a → Bool
```

Client.hs

```
import Library

data Foo = ...




instance Ord’ Foo where
  (≤′) = ...
```

Motivation

Library.hs

```
module Library where

class Eq' a where
  (≡') ::  a → a → Bool

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

Client.hs

```
import Library

data Foo = ...

instance Ord' Foo where
  (≤') = ...
```

```
[1 of 1] Compiling Client ...
No instance for (Eq' Foo)
   arising from the superclasses of an instance declaration
In the instance declaration for ‘Ord' ClientData‘
```

Motivation

Library.hs

```
module Library where

class Eq’ a where
  (≡′) ::  a → a → Bool

class Eq’ a ⇒ Ord’ a where
  (≤′) ::  a → a → Bool
```

Client.hs

```
import Library

data Foo = ...

instance Eq’ Foo where
  (≡′) = ...

instance Ord’ Foo where
  (≤′) = ...
```

The Problem

It is not generally possible to alter type class hierarchy and maintain backward compatibility.

The Problem

It is not generally possible to alter type class hierarchy and maintain backward compatibility.

Some changes are not viable in principle – e. g. removing a class that is beiing used – and can be solved by DEPRECATED pragma.

The Problem

It is not generally possible to alter type class hierarchy and maintain backward compatibility.

Some changes are not viable in principle – e. g. removing a class that is beiing used – and can be solved by DEPRECATED pragma.

On the other hand there is no way to add a superclass into the class context – existing code does not provide instances.

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class

```
class Foo a where

class         Bar a where

class Bar a ⇒ Baz a where
```

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class

~~**class** Foo a **where**~~

class Bar a **where**

class Bar a ⇒ Baz a **where**

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint

~~**class** `Foo a` **where**~~

class `Eq a` ⇒ `Bar a` **where**

class `Bar a` ⇒ `Baz a` **where**

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint
4. Remove a superclass constraint

~~**class** `Foo a` **where**~~

class `Eq a ⇒ Bar a` **where**

class ~~`Bar a ⇒`~~ `Baz a` **where**

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint
4. Remove a superclass constraint
5. Add a new method

~~**class** Foo a **where**~~

class Eq a ⇒ Bar a **where**

class ~~Bar a ⇒~~ Baz a **where**
  qux :: a → ...
  xyz :: a → ...

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint
4. Remove a superclass constraint
5. Add a new method
6. Remove an existing method

~~**class** Foo a **where**~~

class Eq a ⇒ Bar a **where**

class ~~Bar a ⇒~~ Baz a **where**
  ~~qux :: a → ...~~
  xyz :: a → ...

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint
4. Remove a superclass constraint
5. Add a new method
6. Remove an existing method

~~**class** Foo a **where**~~

class Eq a ⇒ Bar a **where**

class ~~Bar a ⇒~~ Baz a **where**
  ~~qux :: a → ...~~
  xyz :: a → ...

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint
4. Remove a superclass constraint
5. Add a new method
6. Remove an existing method

```
~~class Foo a where~~

class Eq a ⇒ Bar a where

class ~~Bar a ⇒~~ Baz a where
  ~~qux :: a → ...~~
  xyz :: a → ...
```

Actions 1,4, and 5 are non-breaking

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint
4. Remove a superclass constraint
5. Add a new method
6. Remove an existing method

~~**class** Foo a **where**~~

class Eq a ⇒ Bar a **where**

class ~~Bar a ⇒~~ Baz a **where**
  ~~qux :: a → ...~~
  xyz :: a → ...

Actions 1,4, and 5 are non-breaking
Actions 2 and 6 can be dealt with by a deprecation cycle

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint
4. Remove a superclass constraint
5. Add a new method
6. Remove an existing method

~~**class** Foo a **where**~~

class Eq a ⇒ Bar a **where**

class ~~Bar a ⇒~~ Baz a **where**
  ~~qux :: a → ...~~
  xyz :: a → ...

Actions 1,4, and 5 are non-breaking
Actions 2 and 6 can be dealt with by a deprecation cycle
Action 4 is the only problem

Problem (cont.)

We can show that the problem can be decomposed to primitive operations

1. Add an empty class
2. Remove an empty class
3. Add a superclass constraint
4. Remove a superclass constraint
5. Add a new method
6. Remove an existing method

~~**class** Foo a **where**~~

class Eq a ⇒ Bar a **where**

class ~~Bar a ⇒~~ Baz a **where**
  ~~qux :: a → ...~~
  xyz :: a → ...

Actions 1,4, and 5 are non-breaking
Actions 2 and 6 can be dealt with by a deprecation cycle
Action 4 is the only problem

There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors

Previous solution attempts

There are some attempts to deal with the problem:

- Arbitrary change with the compiler support – Functor–Applicative–Monad proposal.

Previous solution attempts

There are some attempts to deal with the problem:

- Arbitrary change with the compiler support – Functor–Applicative–Monad proposal.
- Various language extension proposals, none of them implemented. Most of them lacks formal description.

Previous solution attempts

There are some attempts to deal with the problem:

- Arbitrary change with the compiler support – Functor–Applicative–Monad proposal.
- Various language extension proposals, none of them implemented. Most of them lacks formal description.
- The Strathclyde Haskell Enhancement[8]

Arbitrary change with compiler support

Three phase process:

Arbitrary change with compiler support

Three phase process:

- Phase 1: Introduce compiler warnings aka "Applicative/Monad proposal related warnings (AMP phase 1)"[1],

Arbitrary change with compiler support

Three phase process:

- Phase 1: Introduce compiler warnings aka “Applicative/Monad proposal related warnings (AMP phase 1)”[1],
- Phase 2: Prepare Hackage

Arbitrary change with compiler support

Three phase process:

- Phase 1: Introduce compiler warnings aka “Applicative/Monad proposal related warnings (AMP phase 1)”[1],

- Phase 2: Prepare Hackage
- Phase 3: Do the change aka “Implement Functor =>Applicative =>Monad Hierarchy (aka AMP phase 3)”[6],

Arbitrary change with compiler support

Three phase process:

- Phase 1: Introduce compiler warnings aka “Applicative/Monad proposal related warnings (AMP phase 1)”[1],

  ticket opened 20 and closed 17 months ago
- Phase 2: Prepare Hackage
- Phase 3: Do the change aka “Implement Functor =>Applicative =>Monad Hierarchy (aka AMP phase 3)”[6],

  ticket opened 4 years ago, yet to be closed

Various language extension proposals

- Dates back to 2006

Various language extension proposals

- Dates back to 2006
- None implemented, only incomplete specifications

Various language extension proposals

- Dates back to 2006
- None implemented, only incomplete specifications
- Three line of ideas:

Various language extension proposals

- Dates back to 2006
- None implemented, only incomplete specifications
- Three line of ideas:
  - Superclass default instances

Various language extension proposals

- Dates back to 2006
- None implemented, only incomplete specifications
- Three line of ideas:
  - Superclass default instances
  - Default methods

Various language extension proposals

- Dates back to 2006
- None implemented, only incomplete specifications
- Three line of ideas:
  - Superclass default instances
  - Default methods
  - Class aliases

The Strathclyde Haskell Enhancement

- by Connor McBride

The Strathclyde Haskell Enhancement

- by Connor McBride
- a language preprocessor

The Strathclyde Haskell Enhancement

- by Connor McBride
- a language preprocessor
- limited version of Default Superclass Instances proposal
  eg. instance visible only within one module

Our solution - Default Superclass Instances

The instance may be generated automatically:

Our solution - Default Superclass Instances

The instance may be generated automatically:

```
class         Ord' a where
  (<′) ::  a → a → Bool
  (>′) ::  a → a → Bool
```

Our solution - Default Superclass Instances

The instance may be generated automatically:

```
class Eq’ ⇒ Ord’ a where
  (<′) ::  a → a → Bool
  (>′) ::  a → a → Bool

  default instance Eq’ a where
    a ≡′ b = (a ≤′ b) && (b ≤′ a)
```

Our solution - Default Superclass Instances

The instance may be generated automatically:

```
class Eq' ⇒ Ord' a where
  (<′) ::  a → a → Bool
  (>′) ::  a → a → Bool

  default instance Eq' a where
    a ≡′ b = (a ≤′ b) && (b ≤′ a)
```

We provide the formal syntax and the semantics.

Applications

- *Functor–Applicative–Monad* classes

Applications

- *Functor–Applicative–Monad* classes
- Introduce *Bind* and *Pointed* classed

Applications

- *Functor–Applicative–Monad* classes
- Introduce *Bind* and *Pointed* classed
- Provide *Functor* and *Foldable* default superclasses instances of *Traversable*

Applications

- *Functor*–*Applicative*–*Monad* classes
- Introduce *Bind* and *Pointed* classed
- Provide *Functor* and *Foldable* default superclasses instances of *Traversable*

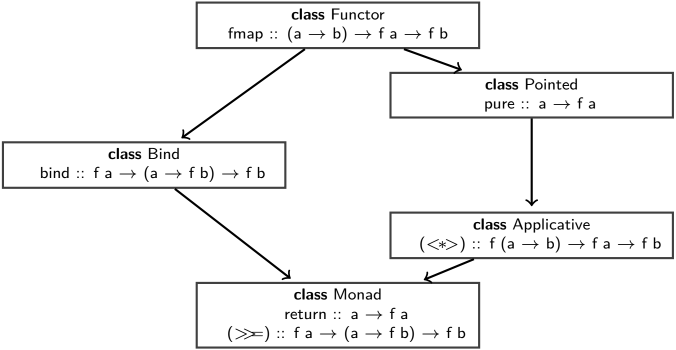
Applications

- *Functor–Applicative–Monad* classes
- Introduce *Bind* and *Pointed* classed
- Provide *Functor* and *Foldable* default superclasses instances of *Traversable*

```
class Applicative m => Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a

    default instance Applicative m where
        pure x = return x
        pf (<*>) px = px >>= \ x -> pf
            >>= \ f -> return (f x)

    default instance Functor m where
        fmap f x = pure f >>= \ g -> return (g x)
```

Applications

- *Functor–Applicative–Monad* classes
- Introduce *Bind* and *Pointed* classed
- Provide *Functor* and *Foldable* default superclasses instances of *Traversable*



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

class Pointed
pure :: a → f a

class Bind
bind :: f a → (a → f b) → f b

class Applicative
(<*>) :: f (a → b) → f a → f b

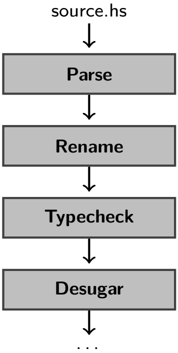
class Monad
return :: a → f a
(>>=) :: f a → (a → f b) → f b

Figure: Refactored class structure

Applications (cont.)

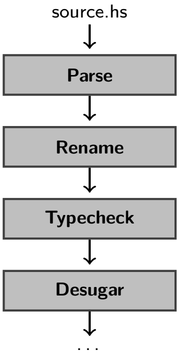
```
{-# LANGUAGE SuperclassDefaultInstance #-}
newtype Id a = Id { getId :: a }
newtype Const a = Const { getConst :: a }

instance Functor Identity where
  fmap f (Id x) = Id (f x)

instance Traversable (Const m) where
  traverse _ (Const m) = pure (Const m)

class (Functor t, Foldable t) => Traversable t where
  ...
  default instance Functor t where
    fmap f = getId .  traverse (Id . f)
  default instance Foldable t where
    foldMap f = getConst .  traverse (Const .  f)
```

GHC implementation



GHC implementation

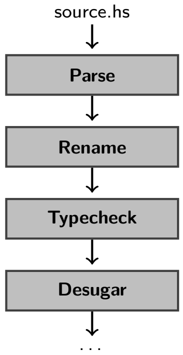
Proof-of-concept implementation of our proposal.



GHC implementation

Proof-of-concept implementation of our proposal.

Enables a new language extension *SuperclassDefaultInstances*



Summary

Summary

- What is the problem?

Summary

- What is the problem?
    - Maintainability of hierarchies

Summary

- What is the problem?
  - Maintainability of hierarchies
- Can we provide a solution?

Summary

- What is the problem?
    - Maintainability of hierarchies
- Can we provide a solution?
    - Yes, Superclass Default Instances

Summary

- What is the problem?
    - Maintainability of hierarchies
- Can we provide a solution?
    - Yes, Superclass Default Instances
- Is it a good solution?

Summary

- What is the problem?
  - Maintainability of hierarchies
- Can we provide a solution?
  - Yes, Superclass Default Instances
- Is it a good solution?
  - It is up to the community
  - We have an implementation to test it

Applicative/Monad proposal related warnings (AMP phase 1). Online. July 2014. URL: https://ghc.haskell.org/trac/ghc/ticket/8004.

Applicative/Monad proposal related warnings (AMP phase 1). Online. Feb. 2015. URL: https://ghc.haskell.org/trac/ghc/wiki/Prelude710.

Default superclass instances. Online. July 2014. URL: https://ghc.haskell.org/trac/ghc/wiki/DefaultSuperclassInstances?version=30.

Karl-Filip Faxén. "A static semantics for Haskell". In: *Journal of Functional Programming* 12 (2002), pp. 295 –357.

Functor–Applicative–Monad Proposal. Online. July 2014. URL: http://www.haskell.org/haskellwiki/index.php?title=Functor-Applicative-Monad_Proposal&oldid=58553.

- *Implement Functor => Applicative => Monad Hierarchy (aka AMP phase 3)*. Online. Feb. 2015. URL: https://ghc.haskell.org/trac/ghc/ticket/4834.
- Simon Marlow. *Haskell 2010 Language Report*. Tech. rep. June 2010. URL: http://www.haskell.org/onlinereport/haskell2010/.
- Connor McBride. *the Strathclyde Haskell Enhancement*. Online. July 2014. URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/.
- John Meacham. *Class Aliases*. Online. URL: http://repetae.net/recent/out/classalias.html.