Effect Systems in Haskell - Part I

Sanchayan Maity

Agenda

- Cover two papers on Effect Systems by Oleg Kiselyov

Agenda

- Cover two papers on Effect Systems by Oleg Kiselyov
  - Extensible Effects - An Alternative to Monad Transformers

Agenda

- Cover two papers on Effect Systems by Oleg Kiselyov
  - Extensible Effects - An Alternative to Monad Transformers
  - Freer Monads, More Extensible Effects

Agenda

- Cover two papers on Effect Systems by Oleg Kiselyov
  - Extensible Effects - An Alternative to Monad Transformers
  - Freer Monads, More Extensible Effects
- Related paper `Reflection Without Remorse`

Agenda

- Cover two papers on Effect Systems by Oleg Kiselyov
  - Extensible Effects - An Alternative to Monad Transformers
  - Freer Monads, More Extensible Effects
- Related paper `Reflection Without Remorse`
- Some sections today's discussion isn't going to cover

Agenda

- Cover two papers on Effect Systems by Oleg Kiselyov
  - Extensible Effects - An Alternative to Monad Transformers
  - Freer Monads, More Extensible Effects
- Related paper `Reflection Without Remorse`
- Some sections today's discussion isn't going to cover
  - Efficiency/Performance of the library or effect system itself

Agenda

- Cover two papers on Effect Systems by Oleg Kiselyov
  - Extensible Effects - An Alternative to Monad Transformers
  - Freer Monads, More Extensible Effects
- Related paper `Reflection Without Remorse`
- Some sections today's discussion isn't going to cover
  - Efficiency/Performance of the library or effect system itself
  - For the sake of time, focus more on the implementation

Agenda

- Cover two papers on Effect Systems by Oleg Kiselyov
  - Extensible Effects - An Alternative to Monad Transformers
  - Freer Monads, More Extensible Effects
- Related paper `Reflection Without Remorse`
- Some sections today's discussion isn't going to cover
  - Efficiency/Performance of the library or effect system itself
  - For the sake of time, focus more on the implementation
  - Comparison of effect system libraries or how to choose one

What's it all about

- **Separate syntax from semantics**

What's it all about

- **Separate syntax from semantics**
- **Interpret your abstract syntax tree in various ways**

What's it all about

- **Separate syntax from semantics**
- **Interpret your abstract syntax tree in various ways**
- **Not losing performance while having both**

Why effect systems

- Monads to model effects but monads don't compose[1]

[1] Composing Monads by Mark Jones and Luc Duponcheel

Why effect systems

- Monads to model effects but monads don't compose[1]
- transformers/mtl has limitations

```
class Monad m => MonadReader r m | m -> r
```

[1] Composing Monads by Mark Jones and Luc Duponcheel

Why effect systems

- Monads to model effects but monads don't compose[1]
- transformers/mtl has limitations
  - Monad transformer stacks are rigid

```
class Monad m => MonadReader r m | m -> r
```

[1] Composing Monads by Mark Jones and Luc Duponcheel

Why effect systems

- Monads to model effects but monads don't compose[1]
- transformers/mtl has limitations
  - Monad transformer stacks are rigid
  - Doesn't allow handling something like `Reader Int (Reader String)` due to functional dependencies

```
class Monad m => MonadReader r m | m -> r
```

[1] Composing Monads by Mark Jones and Luc Duponcheel

Why effect systems

- Monads to model effects but monads don't compose[1]
- transformers/mtl has limitations
  - Monad transformer stacks are rigid
  - Doesn't allow handling something like `Reader Int (Reader String)` due to functional dependencies

```
class Monad m => MonadReader r m | m -> r
```

  - More than a few effects in stack become unwieldy

[1] Composing Monads by Mark Jones and Luc Duponcheel

Why effect systems

- Monads to model effects but monads don't compose[1]
- transformers/mtl has limitations
  - Monad transformer stacks are rigid
  - Doesn't allow handling something like `Reader Int (Reader String)` due to functional dependencies

```
class Monad m => MonadReader r m | m -> r
```

  - More than a few effects in stack become unwieldy
  - n-square instances problem

[1] Composing Monads by Mark Jones and Luc Duponcheel

Effect system libraries

- `freer-simple` based on Extensible Effects and Freer Monads, More Extensible Effects by Oleg Kiselyov

Effect system libraries

- `freer-simple` based on Extensible Effects and Freer Monads, More Extensible Effects by Oleg Kiselyov
- `polysemy` based on Effect Handlers in Scope by Wu, Schrijvers et al

Effect system libraries

- `freer-simple` based on Extensible Effects and Freer Monads, More Extensible Effects by Oleg Kiselyov
- `polysemy` based on Effect Handlers in Scope by Wu, Schrijvers et al
- `fused-effects` based on Fusion for Free: Efficient Algebraic Effect Handlers by Wu, Schrijvers et al

Effect system libraries

- `freer-simple` based on Extensible Effects and Freer Monads, More Extensible Effects by Oleg Kiselyov
- `polysemy` based on Effect Handlers in Scope by Wu, Schrijvers et al
- `fused-effects` based on Fusion for Free: Efficient Algebraic Effect Handlers by Wu, Schrijvers et al
- `cleff` based on `ReaderT IO`

Effect system libraries

- `freer-simple` based on Extensible Effects and Freer Monads, More Extensible Effects by Oleg Kiselyov
- `polysemy` based on Effect Handlers in Scope by Wu, Schrijvers et al
- `fused-effects` based on Fusion for Free: Efficient Algebraic Effect Handlers by Wu, Schrijvers et al
- `cleff` based on `ReaderT IO`
- `effectful` based on `ReaderT IO`

Effect system libraries

- `freer-simple` based on Extensible Effects and Freer Monads, More Extensible Effects by Oleg Kiselyov
- `polysemy` based on Effect Handlers in Scope by Wu, Schrijvers et al
- `fused-effects` based on Fusion for Free: Efficient Algebraic Effect Handlers by Wu, Schrijvers et al
- `cleff` based on `ReaderT IO`
- `effectful` based on `ReaderT IO`
- others?

Free monads

Given a `Functor f`, `Free f` is a `Free` monad.

```
data Free f a = Pure a
              | Free (f (Free f a))
```

A Monad is something that "computes" when monadic context is collapsed by `join :: m (m a) -> m a` (recalling that >>= can be defined as `x >>= y = join (fmap y x)`). This is how Monads carry context through a sequential chain of computations: because at each point in the series, the context from the previous call is collapsed with the next.

A free monad satisfies all the Monad laws, but doesn't do any collapsing (that's the computation). It just builds up a nested series of contexts. The user who creates such a free monadic value is responsible for doing something with those nested contexts, so that the meaning of such a composition can be deferred until after the monadic value has been created.[2]

[2] John Wiegley on Stack Overflow.

Huh, what did that mean

- Define a monad in terms of `return`, `fmap` and `join`, rather than `return` and `(>>=)`.

```
m >>= f = join (fmap f m)
```

```
join :: Functor f => Free f (Free f a) -> Free f a
join (Pure a) = a
join (Free as) = Free (fmap join as)
```

Huh, what did that mean

- Define a monad in terms of `return`, `fmap` and `join`, rather than `return` and `(>>=)`.

```
m >>= f = join (fmap f m)
```

- fmap is performing substitution and join is dealing with any re-normalization.

```
join :: Functor f => Free f (Free f a) -> Free f a
join (Pure a) = a
join (Free as) = Free (fmap join as)
```

Huh, what did that mean

- Define a monad in terms of `return`, `fmap` and `join`, rather than `return` and `(>>=)`.

```
m >>= f = join (fmap f m)
```

- fmap is performing substitution and join is dealing with any re-normalization.
- Done this way, `(m >>= f)` on the `Maybe` monad would first `fmap` to obtain `Just (Just a)`, `Just Nothing` or `Nothing` before flattening.

```
join :: Functor f => Free f (Free f a) -> Free f a
join (Pure a) = a
join (Free as) = Free (fmap join as)
```

Huh, what did that mean

- Define a monad in terms of `return`, `fmap` and `join`, rather than `return` and `(>>=)`.

```
m >>= f = join (fmap f m)
```

- fmap is performing substitution and join is dealing with any re-normalization.
- Done this way, `(m >>= f)` on the `Maybe` monad would first `fmap` to obtain `Just (Just a)`, `Just Nothing` or `Nothing` before flattening.
- In the `Maybe a` case, the association of binds is largely immaterial, the normalization pass fixes things up to basically the same size.

```
join :: Functor f => Free f (Free f a) -> Free f a
join (Pure a) = a
join (Free as) = Free (fmap join as)
```

Huh, what did that mean

- Define a monad in terms of `return`, `fmap` and `join`, rather than `return` and `(>>=)`.

```
m >>= f = join (fmap f m)
```

- fmap is performing substitution and join is dealing with any re-normalization.
- Done this way, `(m >>= f)` on the `Maybe` monad would first `fmap` to obtain `Just (Just a)`, `Just Nothing` or `Nothing` before flattening.
- In the `Maybe a` case, the association of binds is largely immaterial, the normalization pass fixes things up to basically the same size.
- In Free monad, the monad is purely defined in terms of substitution.

```
join :: Functor f => Free f (Free f a) -> Free f a
join (Pure a) = a
join (Free as) = Free (fmap join as)
```

Free monads performance

- Vanilla free monads don't have great performance.

```
newtype FT f m a =
FT { runFT :: forall r. (a -> m r) -> (forall x. (x -> m r) -> f x -> m r)
```

Free monads performance

- Vanilla free monads don't have great performance.
- Solutions like `Codensity` monad transformer and Church encoded free monad exist.[3][4]

```
newtype FT f m a =
FT { runFT :: forall r. (a -> m r) -> (forall x. (x -> m r) -> f x -> m r)
```

[3] Asymptotic Improvement of Computations over Free Monads - Janis Voigtländer
[4] The Free and The Furious: And by 'Furious' I mean Codensity. - raichoo

Free monads performance

- Vanilla free monads don't have great performance.
- Solutions like `Codensity` monad transformer and Church encoded free monad exist.[3][4]

```
newtype FT f m a =
FT { runFT :: forall r. (a -> m r) -> (forall x. (x -> m r) -> f x -> m r) -> m r) }
```

- Think of `Codensity` as a type level construction which ensures that you end up with a right associated bind.[5]

[3] Asymptotic Improvement of Computations over Free Monads - Janis Voigtländer
[4] The Free and The Furious: And by 'Furious' I mean Codensity. - raichoo
[5] Free Monads for less - Edward Kmett

Reflection without remorse

- A left associated expression is asymptotically slower than the equivalent right associated expression. $O(n^2)$ vs $O(n)$ respectively.

Reflection without remorse

- A left associated expression is asymptotically slower than the equivalent right associated expression. $O(n^2)$ vs $O(n)$ respectively.
- What's meant by reflection? Build and observe.

Reflection without remorse

- A left associated expression is asymptotically slower than the equivalent right associated expression. $O(n^2)$ vs $O(n)$ respectively.
- What's meant by reflection? Build and observe.
- Efficient data structures give asymptotic improvement for problematic occurrences of build and observe pattern like monads and monadic reflection.

Extensible effects

- Defines only one effect `Eff`

Extensible effects

- Defines only one effect `Eff`
- Type level list of effects

Extensible effects

- Defines only one effect `Eff`
- Type level list of effects
- What does it mean to be extensible?

Freer monads

- Improves on extensible effects

```
data FFree f a where
    Pure :: a → FFree f a
    Impure :: f x → (x → FFree f a) → FFree f a

instance Monad (FFree f) where
    Impure fx k' >>= k = Impure fx (k' >>> k)
```

The construction lets this implementation choose how to perform the `fmap` operation fixed to the appropriate `output type`.

Freer monads

- Improves on extensible effects
- How?

```haskell
data FFree f a where
    Pure :: a → FFree f a
    Impure :: f x → (x → FFree f a) → FFree f a

instance Monad (FFree f) where
    Impure fx k’ >>= k = Impure fx (k’ >>> k)
```

The construction lets this implementation choose how to perform the `fmap` operation fixed to the appropriate `output type`.

Freer monads

- Improves on extensible effects
- How?
  - Relaxes the `Functor` constraint, becoming `Freer`!

```
data FFree f a where
    Pure :: a → FFree f a
    Impure :: f x → (x → FFree f a) → FFree f a

instance Monad (FFree f) where
    Impure fx k’ >>= k = Impure fx (k’ >>> k)
```

The construction lets this implementation choose how to perform the `fmap` operation fixed to the appropriate `output type`.

Freer monads

- Improves on extensible effects
- How?
  - Relaxes the `Functor` constraint, becoming `Freer`!
  - No need for `Functor` and `Typeable` on `Union`

```
data FFree f a where
    Pure :: a → FFree f a
    Impure :: f x → (x → FFree f a) → FFree f a

instance Monad (FFree f) where
    Impure fx k’ >>= k = Impure fx (k’ >>> k)
```

The construction lets this implementation choose how to perform the `fmap` operation fixed to the appropriate `output type`.

Freer monads

- Improves on extensible effects
- How?
  - Relaxes the `Functor` constraint, becoming `Freer`!
  - No need for `Functor` and `Typeable` on `Union`
- `freer` and `freer-simple` are based on `Freer` monads

```
data FFree f a where
    Pure :: a → FFree f a
    Impure :: f x → (x → FFree f a) → FFree f a

instance Monad (FFree f) where
    Impure fx k' >>= k = Impure fx (k' >>> k)
```

The construction lets this implementation choose how to perform the `fmap` operation fixed to the appropriate `output type`.

Freer monads

- The continuation can now be accessed directly rather than via `fmap`, which has to rebuild the mapped data structure.

```
class Member t r where
    inj :: t v -> Union r v
    prj :: Union r v -> Maybe (t v)
```

and

```
data FEFree r a where
    Pure :: a → FEFree r a
    Impure :: Union r x → (x → FEFree r a) → FEFree r a
```

Freer monads

- The continuation can now be accessed directly rather than via `fmap`, which has to rebuild the mapped data structure.
- The explicit continuation of `FFree` also makes it easier to change its representation.

```
class Member t r where
    inj :: t v -> Union r v
    prj :: Union r v -> Maybe (t v)
```

and

```
data FEFree r a where
    Pure :: a → FEFree r a
    Impure :: Union r x → (x → FEFree r a) → FEFree r a
```

Freer monads

- `FEFree r` becomes `Eff r`, where `r` is the list of effect labels.

```
type Arr r a b = a → Eff r b

data FTCQueue m a b where
  Leaf :: (a -> m b) -> FTCQueue m a b
  Node :: FTCQueue m a x -> FTCQueue m x b -> FTCQueue m a b

type Arrs r a b = FTCQueue (Eff r) a b

data Eff r a where
    Pure :: a → Eff r a
    Impure :: Union r x → Arrs r x a → Eff r a
```

Freer monads

- `FEFree r` becomes `Eff r`, where `r` is the list of effect labels.
- The request continuation which receives the reply `x` and works towards the final answer `a`, then has the type `x → Eff r a`.

```
type Arr r a b = a → Eff r b

data FTCQueue m a b where
  Leaf :: (a -> m b) -> FTCQueue m a b
  Node :: FTCQueue m a x -> FTCQueue m x b -> FTCQueue m a b

type Arrs r a b = FTCQueue (Eff r) a b

data Eff r a where
    Pure :: a → Eff r a
    Impure :: Union r x → Arrs r x a → Eff r a
```

Resources

- Why Free monads matter

Resources

- Why Free monads matter
- Free monad considered harmful

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads
- Free Monads for less

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads
- Free Monads for less
- When to use CPS vs codensity vs reflection without remorse

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads
- Free Monads for less
- When to use CPS vs codensity vs reflection without remorse
- ReaderT pattern is just extensible effects

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads
- Free Monads for less
- When to use CPS vs codensity vs reflection without remorse
- ReaderT pattern is just extensible effects
- My Effects Bibliography

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads
- Free Monads for less
- When to use CPS vs codensity vs reflection without remorse
- ReaderT pattern is just extensible effects
- My Effects Bibliography
- Effects Bibliography

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads
- Free Monads for less
- When to use CPS vs codensity vs reflection without remorse
- ReaderT pattern is just extensible effects
- My Effects Bibliography
- Effects Bibliography
- Freer simple effects examples

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads
- Free Monads for less
- When to use CPS vs codensity vs reflection without remorse
- ReaderT pattern is just extensible effects
- My Effects Bibliography
- Effects Bibliography
- Freer simple effects examples
- Continuation Passing Style

Resources

- Why Free monads matter
- Free monad considered harmful
- Building real-world Haskell applications using Tagless-Final and ReaderT
- Free monads from scratch
- An earlier talk of my own on Free Monads
- Free Monads for less
- When to use CPS vs codensity vs reflection without remorse
- ReaderT pattern is just extensible effects
- My Effects Bibliography
- Effects Bibliography
- Freer simple effects examples
- Continuation Passing Style
- Existential Quantification

Questions?

- Reach out on

Questions?

- Reach out on
  - Email: sanchayan@sanchayanmaity.net

Questions?

- Reach out on
  - Email: sanchayan@sanchayanmaity.net
  - Mastodon: https://sanchayanmaity.com/@sanchayan

Questions?

- Reach out on
  - Email: sanchayan@sanchayanmaity.net
  - Mastodon: https://sanchayanmaity.com/@sanchayan
  - Blog: https://sanchayanmaity.net

Questions?

- Reach out on
  - Email: sanchayan@sanchayanmaity.net
  - Mastodon: https://sanchayanmaity.com/@sanchayan
  - Blog: https://sanchayanmaity.net
  - Telegram:

Questions?

- Reach out on
  - Email: sanchayan@sanchayanmaity.net
  - Mastodon: https://sanchayanmaity.com/@sanchayan
  - Blog: https://sanchayanmaity.net
  - Telegram:
    - t.me/fpncr

Questions?

- Reach out on
  - Email: sanchayan@sanchayanmaity.net
  - Mastodon: https://sanchayanmaity.com/@sanchayan
  - Blog: https://sanchayanmaity.net
  - Telegram:
    - t.me/fpncr
    - t.me/SanchayanMaity