

# The Constrained-Monad Problem

Neil Sculthorpe

(joint work with Jan Bracker, George Giorgidze and Andy Gill)

Functional Programming Group  
Information and Telecommunication Technology Center  
University of Kansas  
neil@ittc.ku.edu

Nottingham, England  
16th July 2013

# Monads in Haskell

```
{-# LANGUAGE KindSignatures #-}
```

## The Monad Type Class

```
class Monad (m :: * → *) where
  return :: a → m a
  ( >>= ) :: m a → (a → m b) → m b
```

## The Monad Laws

- $\text{return } a \gg= g \equiv g a$  (left-identity law)
- $m \gg= \text{return} \equiv m$  (right-identity law)
- $(m \gg= g) \gg= h \equiv m \gg= (\lambda x \rightarrow g x \gg= h)$  (associativity law)

# Sets in Haskell

```
import Data.Set
```

## Selected functions from the Data.Set library

```
singleton :: a → Set a  
toList    :: Set a → [a]  
fromList  :: Ord a ⇒ [a] → Set a  
unions    :: Ord a ⇒ [Set a] → Set a
```

# Sets in Haskell

```
import Data.Set
```

## Selected functions from the Data.Set library

```
singleton :: a → Set a  
toList    :: Set a → [a]  
fromList  :: Ord a ⇒ [a] → Set a  
unions    :: Ord a ⇒ [Set a] → Set a
```

## Monadic Set Operations

```
returnSet :: a → Set a  
returnSet = singleton  
bindSet   :: Ord b ⇒ Set a → (a → Set b) → Set b  
bindSet s k = unions (map k (toList s))
```

# Sets in Haskell

```
import Data.Set
```

## Selected functions from the Data.Set library

```
singleton :: a → Set a  
toList    :: Set a → [a]  
fromList  :: Ord a ⇒ [a] → Set a  
unions    :: Ord a ⇒ [Set a] → Set a
```

## Monadic Set Operations

```
returnSet :: a → Set a  
returnSet = singleton  
bindSet   :: Ord b ⇒ Set a → (a → Set b) → Set b  
bindSet s k = unions (map k (toList s))
```

```
instance Monad Set where
```

```
return = returnSet  
( >>= ) = bindSet    -- does not type check
```

# Vectors

## A Vector Representation

```
type Vec (a :: *) = (a → ℂ)
```

```
class Finite (a :: *) where  
  enumerate :: [a]
```

# Vectors

## A Vector Representation

```
type Vec (a :: *) = (a → ℂ)
```

```
class Finite (a :: *) where  
  enumerate :: [a]
```

## Monadic Vector Operations

```
returnVec :: Eq a ⇒ a → Vec a
```

```
returnVec a = λ b → if a == b then 1 else 0
```

```
bindVec :: Finite a ⇒ Vec a → (a → Vec b) → Vec b
```

```
bindVec v k = λ b → sum [v a × (k a) b | a ← enumerate]
```

# Vectors

## A Vector Representation

```

type Vec (a :: *) = (a → ℂ)
class Finite (a :: *) where
  enumerate :: [a]
  
```

## Monadic Vector Operations

```

returnVec :: Eq a ⇒ a → Vec a
returnVec a = λ b → if a == b then 1 else 0

bindVec :: Finite a ⇒ Vec a → (a → Vec b) → Vec b
bindVec v k = λ b → sum [v a × (k a) b | a ← enumerate]

instance Monad Vector where
  return = returnVec      -- does not type check
  ( >>= ) = bindVec       -- does not type check
  
```



# Embedded Domain Specific Languages

```
{-# LANGUAGE GADTs #-}
```

## Embedding Monadic Operations

```
data EDSL :: * → * where
```

```
...
```

```
IfThenElse :: EDSL Bool → EDSL a → EDSL a → EDSL a
```

# Embedded Domain Specific Languages

{-# LANGUAGE GADTs #-}

## Embedding Monadic Operations

**data** EDSL :: \* → \* **where**

...

**IfThenElse** :: EDSL Bool → EDSL a → EDSL a → EDSL a

**Return** :: a → EDSL a

**Bind** :: EDSL x → (x → EDSL a) → EDSL a

# Embedded Domain Specific Languages

{-# LANGUAGE GADTs #-}

## Embedding Monadic Operations

```
data EDSL :: * → * where
```

...

```
IfThenElse :: EDSL Bool → EDSL a → EDSL a → EDSL a
```

```
Return      :: a → EDSL a
```

```
Bind        :: EDSL x → (x → EDSL a) → EDSL a
```

```
instance Monad EDSL where
```

```
return = Return
```

```
( >>= ) = Bind
```

# Embedded Domain Specific Languages

{-# LANGUAGE GADTs #-}

## Embedding Monadic Operations

**data** EDSL :: \* → \* **where**

...

**IfThenElse** :: EDSL Bool → EDSL a → EDSL a → EDSL a

**Return** :: a → EDSL a

**Bind** :: EDSL x → (x → EDSL a) → EDSL a

**instance** Monad EDSL **where**

return = **Return**

(**>>=**) = **Bind**

compile :: Reifiable a ⇒ EDSL a → Code

# Embedded Domain Specific Languages

{-# LANGUAGE GADTs #-}

## Embedding Monadic Operations

**data** EDSL :: \* → \* **where**

...

**IfThenElse** :: EDSL Bool → EDSL a → EDSL a → EDSL a

**Return** :: a → EDSL a

**Bind** :: Reifiable x ⇒ EDSL x → (x → EDSL a) → EDSL a

**instance** Monad EDSL **where**

return = **Return**

(**>>=**) = **Bind** -- does not type check

compile :: Reifiable a ⇒ EDSL a → Code

# Why is this a Problem?

- A Monad instance is useful because the Haskell language and libraries provide a significant amount of infrastructure to support arbitrary monads.
- The problem generalises from monads to any type class with polymorphic class methods.
- This talk will mostly be about monads, but will conclude with some other examples.
- Our solution generalises to some, but not all, type classes.
- Future work:
  - characterising the type classes for which it works;
  - extending/adapting the solution to other type classes.

# Constraint Kinds

```
{-# LANGUAGE ConstraintKinds #-}
```

```
import GHC.Exts (Constraint)
```

## Constraint Kinds in GHC

The kind of a fully applied type class is the literal kind `Constraint`.

For example:

```
Ord    :: * → Constraint
```

```
Monad :: (* → *) → Constraint
```

# A Partial Solution: Restricted Type Classes

```
{-# LANGUAGE MultiParamTypeClasses, InstanceSigs #-}
```

## Restricted Monad Class

```
class RMonad (c :: * → Constraint) (m :: * → *) where  
  return :: c a      ⇒ a → m a  
  (≫=) :: (c a, c b) ⇒ m a → (a → m b) → m b
```



# A Partial Solution: Restricted Type Classes

```
{-# LANGUAGE MultiParamTypeClasses, InstanceSigs #-}
```

## Restricted Monad Class

```
class RMonad (c :: * → Constraint) (m :: * → *) where
  return :: c a      ⇒ a → m a
  ( >>= ) :: (c a, c b) ⇒ m a → (a → m b) → m b
```

## Example: Set and Ord

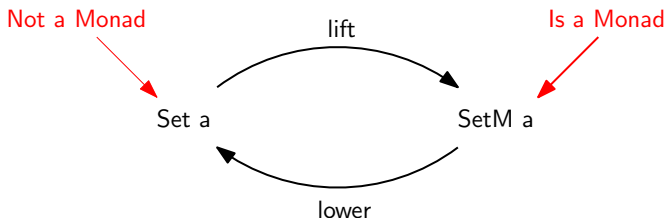
```
instance RMonad Ord Set where
  return :: Ord a ⇒ a → Set a
  return = returnSet
  ( >>= ) :: (Ord a, Ord b) ⇒ Set a → (a → Set b) → Set b
  ( >>= ) = bindSet
```

# An alternative: Embedding and Normalisation

- An alternative is to embed the type in another data type that does form a monad.

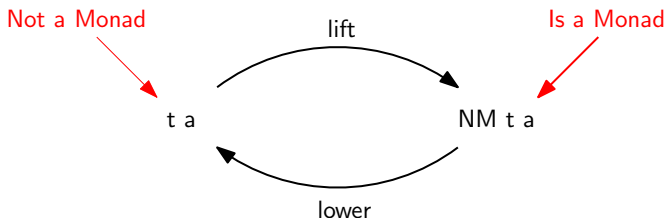
# An alternative: Embedding and Normalisation

- An alternative is to embed the type in another data type that does form a monad.



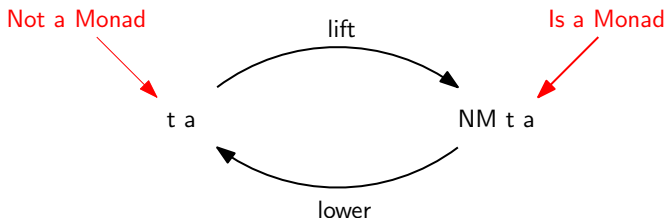
# An alternative: Embedding and Normalisation

- An alternative is to embed the type in another data type that does form a monad.



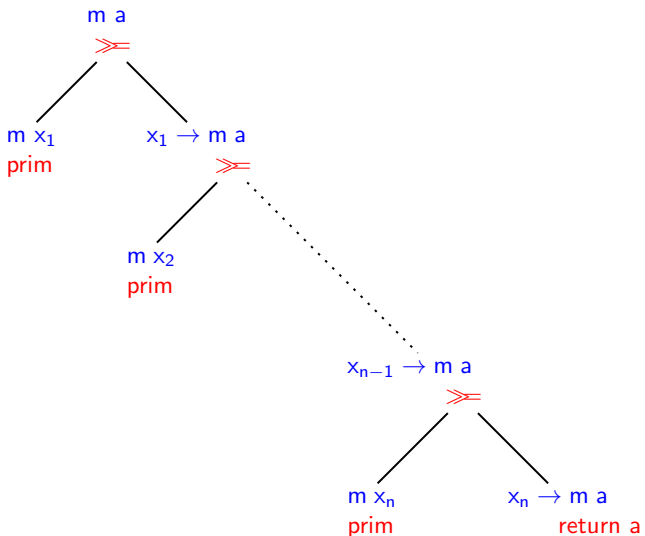
# An alternative: Embedding and Normalisation

- An alternative is to embed the type in another data type that does form a monad.



- The key ideas are:
  - `NM` represents a monadic computation in a **normal form**;
  - the `lift` and `lower` functions enforce the constraint.

# A Normal Form for Monadic Computations



# Embedding Constrained Monadic Computations

{-# LANGUAGE GADTs #-}

Normalised Monads as a GADT

```
data NM :: (* -> *) -> * -> * where
  Return :: a -> NM t a
  Bind   :: t x -> (x -> NM t a) -> NM t a
```

# Embedding Constrained Monadic Computations

{-# LANGUAGE GADTs #-}

## Constrained Normalised Monads as a GADT

```
data NM :: (* → Constraint) → (* → *) → * → * where
  Return :: a → NM c t a
  Bind    :: c x ⇒ t x → (x → NM c t a) → NM c t a
```



# Embedding Constrained Monadic Computations

{-# LANGUAGE GADTs #-}

## Constrained Normalised Monads as a GADT

```
data NM :: (* -> Constraint) -> (* -> *) -> * -> * where
  Return :: a -> NM c t a
  Bind   :: c x => t x -> (x -> NM c t a) -> NM c t a
```

## Constrained Normalised Monads are (standard) Monads!

```
instance Monad (NM c t) where
  return :: a -> NM c t a
  return = Return

  ( >>= ) :: NM c t a -> (a -> NM c t b) -> NM c t b
  (Return a) >>= k = k a -- left-identity law
  (Bind tx h) >>= k = Bind tx (\x -> h x >>= k) -- associativity law
```

# Embedding Constrained Monadic Computations

{-# LANGUAGE GADTs #-}

## Constrained Normalised Monads as a GADT

```
data NM :: (* -> Constraint) -> (* -> *) -> * -> * where
  Return :: a -> NM c t a
  Bind    :: c x => t x -> (x -> NM c t a) -> NM c t a
```

## Lifting Primitive Operations

```
lift :: c a => t a -> NM c t a
lift ta = Bind ta Return    -- right-identity law
```

# Embedding Constrained Monadic Computations

{-# LANGUAGE GADTs, RankNTypes, ScopedTypeVariables #-}

## Constrained Normalised Monads as a GADT

```
data NM :: (* → Constraint) → (* → *) → * → * where
  Return :: a → NM c t a
  Bind   :: c x ⇒ t x → (x → NM c t a) → NM c t a
```

## Lowering Monadic Computations

```
lower :: ∀ a c t. (a → t a) → (∀ x. c x ⇒ t x → (x → t a) → t a) → NM c t a → t a
lower ret bind = lower'
```

where

```
lower' :: NM c t a → t a
lower' (Return a) = ret a
lower' (Bind tx k) = bind tx (lower' ∘ k)
```

# Embedding Constrained Monadic Computations

{-# LANGUAGE GADTs, RankNTypes, ScopedTypeVariables #-}

## Constrained Normalised Monads as a GADT

```

data NM :: (* → Constraint) → (* → *) → * → * where
  Return :: a → NM c t a
  Bind    :: c x ⇒ t x → (x → NM c t a) → NM c t a
  
```

## Example: Set and Ord

```

type SetM a = NM Ord Set a
liftSet :: Ord a ⇒ Set a → SetM a
liftSet = lift
lowerSet :: Ord a ⇒ SetM a → Set a
lowerSet = lower returnSet bindSet
  
```

# Embedding Constrained Monadic Computations

{-# LANGUAGE GADTs, RankNTypes, ScopedTypeVariables #-}

## Constrained Normalised Monads as a GADT

```
data NM :: (* -> Constraint) -> (* -> *) -> * -> * where
  Return :: a -> NM c t a
  Bind    :: c x => t x -> (x -> NM c t a) -> NM c t a
```

## Folding Monadic Computations

```
fold :: ∀ a c r t. (a -> r) -> (∀ x. c x => t x -> (x -> r) -> r) -> NM c t a -> r
fold ret bind = fold'
```

where

```
fold' :: NM c t a -> r
fold' (Return a) = ret a
fold' (Bind tx k) = bind tx (fold' ∘ k)
```

# Embedding Constrained Functor Computations

## Constrained Normalised Functors as a GADT

```
data NF :: (* → Constraint) → (* → *) → * → * where  
  FMap :: c x ⇒ (x → a) → t x → NF c t a
```

# Embedding Constrained Functor Computations

## Constrained Normalised Functors as a GADT

```
data NF :: (* -> Constraint) -> (* -> *) -> * -> * where
  FMap :: c x => (x -> a) -> t x -> NF c t a
```

## Constrained Normalised Functors are (standard) Functors

```
instance Functor (NF c t) where
  fmap :: (a -> b) -> NF c t a -> NF c t b
  fmap g (FMap h tx) = FMap (g ∘ h) tx    -- composition law
```

# Embedding Constrained Functor Computations

## Constrained Normalised Functors as a GADT

```
data NF :: (* -> Constraint) -> (* -> *) -> * -> * where
  FMap :: c x => (x -> a) -> t x -> NF c t a
```

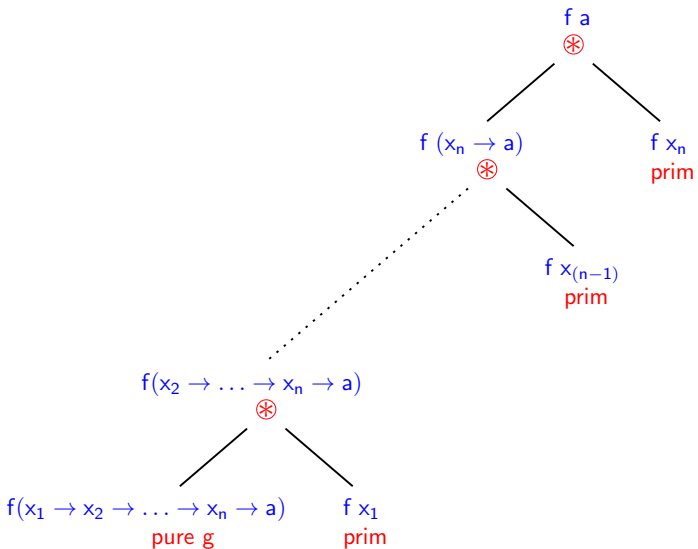
## Lifting and Lowering

```
liftNF :: c a => t a -> NF c t a
liftNF ta = FMap id ta      -- identity law

lowerNF :: (forall x. c x => (x -> a) -> t x -> t a) -> NF c t a -> t a
lowerNF fmp (FMap g tx) = fmp g tx
```



# A Normal Form for Applicative Computations



# Remarks

- The normalisation solution requires a normal form where all existential types are parameters on primitive operations. E.g.
  - this is true of Category
  - but not Arrow

# Remarks

- The normalisation solution requires a normal form where all existential types are parameters on primitive operations. E.g.
  - this is true of `Category`
  - but not `Arrow`
- The monadic normalisation is the same as used by `Unimo` [Lin06], `MonadPrompt` [IF08], and `Operational` [Apf10], and brings the same benefits:
  - enforces the monad laws
  - separates structure from interpretation
  - allows multiple interpretations

# Remarks

- The normalisation solution requires a normal form where all existential types are parameters on primitive operations. E.g.
  - this is true of `Category`
  - but not `Arrow`
- The monadic normalisation is the same as used by `Unimo` [Lin06], `MonadPrompt` [IF08], and `Operational` [Apf10], and brings the same benefits:
  - enforces the monad laws
  - separates structure from interpretation
  - allows multiple interpretations
- The first use of normalisation to overcome the constrained-monad problem was by the `RMonad` library [SG08].

# Remarks

- The normalisation solution requires a normal form where all existential types are parameters on primitive operations. E.g.
  - this is true of `Category`
  - but not `Arrow`
- The monadic normalisation is the same as used by `Unimo` [Lin06], `MonadPrompt` [IF08], and `Operational` [Apf10], and brings the same benefits:
  - enforces the monad laws
  - separates structure from interpretation
  - allows multiple interpretations
- The first use of normalisation to overcome the constrained-monad problem was by the `RMonad` library [SG08].
- An alternative means of normalising is to use a continuation transformer [PAS12].

# Remarks

- The normalisation solution requires a normal form where all existential types are parameters on primitive operations. E.g.
  - this is true of `Category`
  - but not `Arrow`
- The monadic normalisation is the same as used by `Unimo` [Lin06], `MonadPrompt` [IF08], and `Operational` [Apf10], and brings the same benefits:
  - enforces the monad laws
  - separates structure from interpretation
  - allows multiple interpretations
- The first use of normalisation to overcome the constrained-monad problem was by the `RMonad` library [SG08].
- An alternative means of normalising is to use a continuation transformer [PAS12].
- Normalisation preserves semantics, but can change the operational behaviour of the monad.

## Further Reading

See our paper for more details:



Neil Sculthorpe, Jan Bracker, George Giorgidze and Andy Gill.  
The Constrained-Monad Problem.

*In International Conference on Functional Programming*. ACM, 2013.

<http://www.ittc.ku.edu/~neil/publications.html>.

# References



Heinrich Apfeldmus.

The Operational monad tutorial.

*The Monad.Reader*, 15:37–55, 2010.



Ryan Ingram and Bertram Felgenhauer, 2008.

<http://hackage.haskell.org/package/MonadPrompt>.



Chuan-kai Lin.

Programming monads operationally with Unimo.

In *International Conference on Functional Programming*, pages 274–285. ACM, 2006.



Anders Persson, Emil Axelsson, and Josef Svenningsson.

Generic monadic constructs for embedded languages.

In *Implementation and Application of Functional Languages 2011*, volume 7257 of *LNCS*, pages 85–99. Springer, 2012.



Ganesh Sittampalam and Peter Gavin, 2008.

<http://hackage.haskell.org/package/rmonad>.