The Constrained-Monad Problem

Neil Sculthorpe, Jan Bracker, George Giorgidze and Andy Gill

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

International Conference on Functional Programming
Boston, Massachusetts
27th September 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

- return a >>= k ≡ k a (left-identity law)
- ma >>= return ≡ ma (right-identity law)
- (ma >>= h) >>= k ≡ ma >>= (λ a → h a >>= k) (associativity law)

Sets in Haskell

```
import Data.Set
```

Selected functions from the Data.Set library

```
singleton :: a → Set a
toList    :: Set a → [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]
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]
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
```

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       :: EDSL x → (x → EDSL a)       → EDSL a

instance Monad EDSL where
   return  = Return
   ( >>= ) = Bind

compile :: Reifiable a ⇒ EDSL a → Code
compile (IfThenElse b t e) = ... compile b ... compile t ... compile e ...
```

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
compile (IfThenElse b t e) = ... compile b ... compile t ... compile e ...
compile (Bind mx k)        = ... compile mx ... compile ∘ k ......
```

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 typecheck

compile :: Reifiable a ⇒ EDSL a → Code
compile (IfThenElse b t e) = ... compile b ... compile t ... compile e ...
compile (Bind mx k)        = ... compile mx ... compile ∘ k ......
```

The Problem

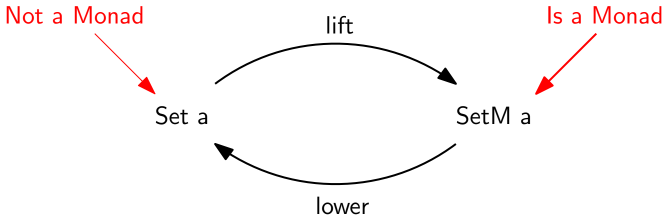
- The shallow constrained-monad problem: Monad instances cannot be defined using ad-hoc polymorphic functions.
- The deep constrained-monad problem: Monadic computations cannot be reified.
- The problem generalises to any type class with parametrically polymorphic methods.

Embedding and Normalisation

- Solution: embed the type into a data type that does form a monad.

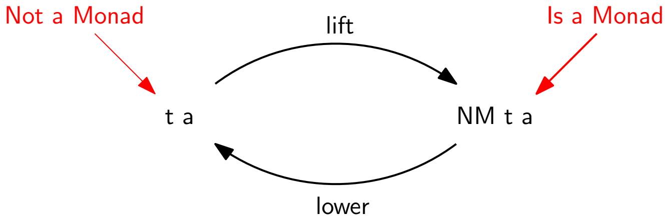
Embedding and Normalisation

- Solution: embed the type into a data type that does form a monad.



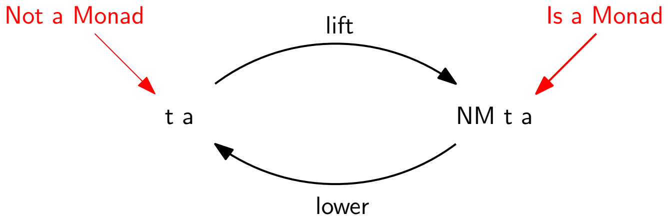
Embedding and Normalisation

- Solution: embed the type into a data type that does form a monad.



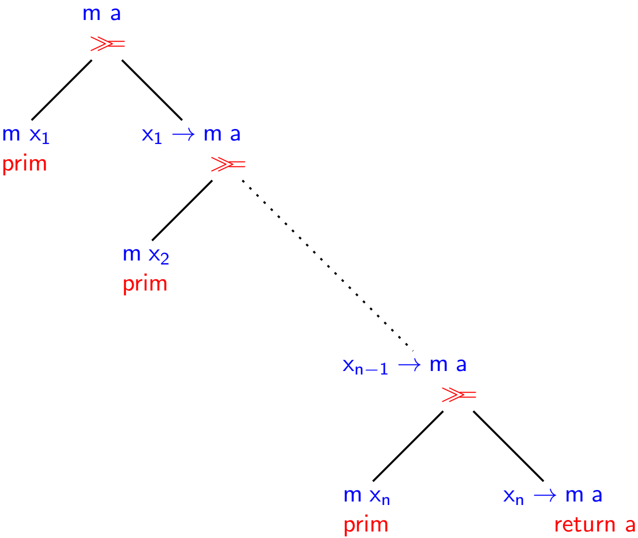
Embedding and Normalisation

- Solution: embed the type into a 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, ConstraintKinds #-}
import GHC.Exts (Constraint)
```

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, ConstraintKinds #-}
import GHC.Exts (Constraint)
```

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 (λ x → h x >>= k)      -- associativity law
```

Embedding Constrained Monadic Computations

```
{-# LANGUAGE GADTs, ConstraintKinds #-}
import GHC.Exts (Constraint)
```

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, ConstraintKinds, RankNTypes, ScopedTypeVariables #-}
import GHC.Exts (Constraint)
```

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, ConstraintKinds, RankNTypes, ScopedTypeVariables #-}
import GHC.Exts (Constraint)
```

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 Monadic Computations

```
{-# LANGUAGE GADTs, ConstraintKinds, RankNTypes, ScopedTypeVariables #-}
import GHC.Exts (Constraint)
```

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
```

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.

References

Heinrich Apfelmus.
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`.