(Perhaps Less Simple) Monadic Equational Reasoning

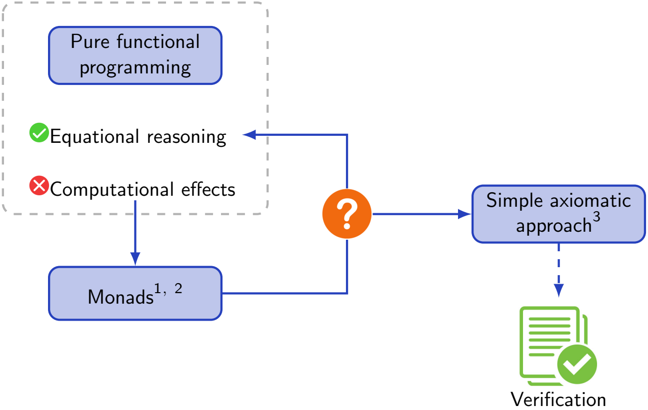
Elisabet Lobo-Vesga

Joint work with:
Andrés Sicard-Ramírez

Universidad EAFIT
Medellín, Colombia

January 2017

Introduction

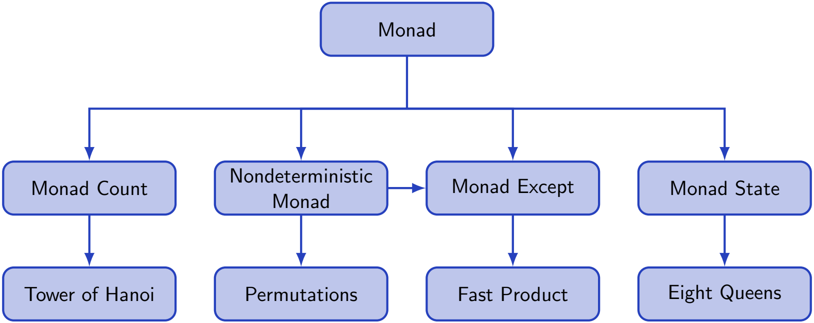


[1] Moggi, E. (1991) Notions of computation and monads.

[2] Wadler, P. (1995) Monads for functional programming.

[3] Gibbons, J., & Hinze, R. (2011) Just do it: simple monadic equational reasoning.

Just do It: Simple Monadic Equational Reasoning[3]



[3] Gibbons, J., & Hinze, R. (2011) Just do it: simple monadic equational reasoning.

Haskell

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

Properties

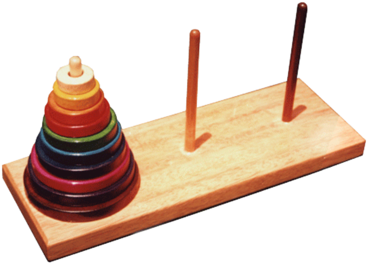
```
return x ≫= f    = f x
mx ≫= return     = mx
(mx ≫= f) ≫= g =
    mx ≫= (λ x → f x ≫= g)
```

Agda[4]

```
record Monad (M : Set → Set) : Set₁ where
  constructor mkMonad

  field
    return : {A : Set} → A → M A
    _≫=_   : {A B : Set} → M A → (A → M B) → M B

    unity-left    : {A B : Set} {f : A → M B} (x : A) →
                    (return x) ≫= f ≡ f x

    unity-right   : {A : Set} (mx : M A) → mx ≫= return ≡ mx

    associativity : {A B C : Set} {f : A → M B} {g : B → M C} (mx : M A) →
                    (mx ≫= f) ≫= g ≡ mx ≫= (λ x → f x ≫= g)
```

[4] Villa Izasa, J. P. (2014) Category Theory Applied to Functional Programming.

Tower of Hanoi



(Source: Blogspot. Image by Unknown)

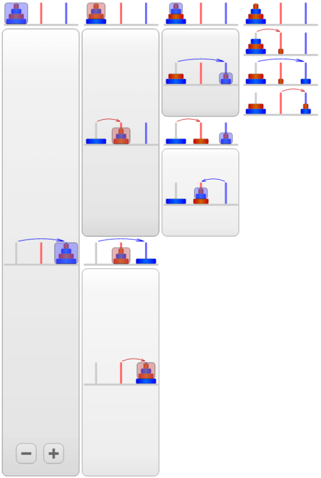
Rules

1. Only one disk can be move at a time
2. A disk can only be moved if it's the uppermost disk on a stack
3. No disk may be placed on top of a smaller disk

Recursive solution

- Let n be the total number of discs
- Number the discs from 1 (topmost) to n (bottom-most)

1. Move $n-1$ discs from the source to the spare peg
2. Move disk n from the source to the target peg
3. Move $n-1$ discs from the spare to the target peg



(Source: Wikipedia. Image by Cmglee)

MonadCount

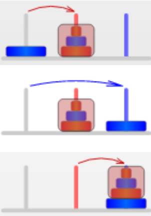
```
-- Supports effect of counting
record MonadCount {M : Set → Set} (monad : Monad M) : Set₁ where
  constructor mkMonadCount

    field
      tick : M ⊤
```

Extra functions

```
-- Sequential composition
_≫_ : {A B : Set} → M A → M B → M B
mx ≫ my = mx ≫= λ _ → my

-- Identity computation
skip : M ⊤
skip = return tt
```

Implementation

```
-- Ticks the counter once for each move of a disk
hanoi : ℕ → M ⊤
hanoi zero    = skip
hanoi (suc n) = hanoi n ≫ tick ≫ hanoi n

-- Repeats a unit computation a fixed number of times
rep : ℕ → M ⊤ → M ⊤
rep zero    mx = skip
rep (suc n) mx = mx ≫ rep n mx
```



hanoi n

tick

hanoi n

Properties of rep

```
rep-1  : (mx : M ⊤) → rep 1 mx ≡ mx

rep-mn : ∀ m n → (mx : M ⊤) → rep (m + n) mx ≡ (rep m mx ≫ rep n mx)
```

Proof

```
-- Solving a Tower of Hanoi of n discs requires 2^n-1 moves (by induction)
moves : ∀ n → hanoi n ≡ rep (2^n ∸ 1) tick
moves zero    = refl -- Base case
moves (suc n) =      -- Inductive step
  begin
    (hanoi n ≫ tick ≫ hanoi n)
      ≡⟨ cong f (moves n) ⟩ -- Inductive Hypothesis
    (rep (2^n ∸ 1) tick ≫ tick ≫ rep (2^n ∸ 1) tick)
      ≡⟨ cong g (sym (rep-1 tick)) ⟩
    (rep (2^n ∸ 1) tick ≫ rep 1 tick ≫ rep (2^n ∸ 1) tick)
      ≡⟨ cong (λ x → x ≫ r) (sym (rep-mn (2^n ∸ 1) 1 tick)) ⟩
    (rep (2^n ∸ 1 + 1) tick ≫ rep (2^n ∸ 1) tick)
      ≡⟨ sym (rep-mn (2^n ∸ 1 + 1) (2^n ∸ 1) tick) ⟩
    rep ((2^n ∸ 1) + 1 + (2^n ∸ 1)) tick
      ≡⟨ cong (λ x → rep x tick) (sym (thm n)) ⟩
    rep (2^(n + 1) ∸ 1) tick
      ≡⟨ cong (λ x → rep (2^x ∸ 1) tick) (sym (succ n)) ⟩
    rep (2^(suc n) ∸ 1) tick
  ■
    where f = λ x → x ≫ tick ≫ x
          r = rep (2^n ∸ 1) tick
          g = λ x → r ≫ x ≫ r
```

What did just happened?

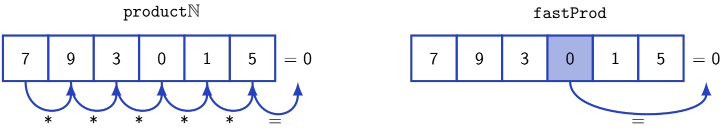
- We modeled a problem using monads in Agda
- We proved that our solution behaves as expected only using the properties of monads (not their instances)
- We were able to use ("simple") equational reasoning in our proofs

Monad Except
Another subclass of Monad

```
-- Exceptional computations
record MonadExcept {M : Set → Set} {Mnd : Monad M}
  (monad : MonadNonDet Mnd) : Set₁ where

  constructor mkMonadExcept

  field
    catch        : {A : Set} → M A → M A → M A

    catch-fail₁  : {A : Set} (h : M A) → catch fail h ≡ h

    catch-fail₂  : {A : Set} (m : M A) → catch m fail ≡ m

    catch-catch  : {A : Set} (m h h' : M A) →
      catch m (catch h h') ≡ catch (catch m h) h'

    catch-return : {A : Set} (x : A) (h : M A) → catch (return x) h ≡ return x
```

Fast Product

Reasoning with exceptions




```
-- Computes the product of a list of Natural numbers
productℕ : List ℕ → ℕ
productℕ []        = 1
productℕ (x :: xs) = x * productℕ xs

work : List ℕ → M ℕ
work xs = if (elem 0 xs) then fail else (return (productℕ xs))

fastProd : List ℕ → M ℕ
fastProd xs = catch (work xs) (return 0)
```

Fast Product
Reasoning with exceptions

```
-- Fast product is equivalent to product
pureFastProd : (xs : List ℕ) → fastProd xs ≡ return (productℕ xs)
pureFastProd xs =
  begin
    catch (if (elem 0 xs) then fail else (return (productℕ xs))) (return 0)
      ≡⟨ pop-if catch (elem 0 xs) ⟩
    (if (elem 0 xs) then mx else my)
      ≡⟨ cong (λ x → (if (elem 0 xs) then x else my))
              (catch−fail₁ (return 0)) ⟩
    (if (elem 0 xs) then (return 0) else my)
      ≡⟨ cong (λ x → (if (elem 0 xs) then (return 0) else x))
              (catch−return (productℕ xs) (return 0)) ⟩
    (if (elem 0 xs) then (return 0) else (return (productℕ xs)))
      ≡⟨ sym (push-function-into-if return (elem 0 xs)) ⟩
    return (if (elem 0 xs) then 0 else (productℕ xs))
      ≡⟨ cong return extra-if ⟩
    return (productℕ xs)
  ∎
    where mx       = catch fail (return 0)
          my       = catch (return (productℕ xs)) (return 0)
          extra-if = if-cong (λ p → sym (product0₂ xs p)) (λ _ → refl)
```

Questions?