Monads

Kovács György

2026

Table of Contents

What is Computation?

In 1928 David Hilbert and Wilhelm Ackermann posed a challenge called the "Entscheidungsproblem": It asked whether or not it is possible to create an algorith that determines if a statement is universally valid.

In 1936 Alan Turing and Alonzo Church published papers proving that it is not possible. They used totally different methods, and layed the foundations of computation:

1. Turing Machine (Imperative Paradigm)
2. Lambda Calculus (Functional Paradigm)

The Turing Machine Model

- The model is made up of variables (memory locations) and a list of instructions.
- Computation happens by executing instructions one after the other in order to change the variables.
- The computation is done when there are no more instructions.

The Lambda Calculus Model

- The model is made up of a lambda expression, and some rules of evaluation.
- Computation happens by evaluating lambda expressions.
- The computation is done when there are no more expressions to evaluate.

What is Computation?

I recommend reading up on these articles if you are interested:

1. "On Computable Numbers, with an Aplication to the Entscheidungsproblem" by Alan Turing, 1936
2. "An Unsolvable Problem of Elementary Number Theory" by Alonzo Church, 1936

What is the problem we are trying to solve?

What is the problem we are trying to solve?

Important Fact #1!

The Computer works like a Turing Machine!

What is the problem we are trying to solve?

Important Fact #1!

The Computer works like a Turing Machine!

Important Fact #2!

The Functional Programming languages run on the Computer!

What is the problem we are trying to solve?

Important Fact #1!

The Computer works like a Turing Machine!

Important Fact #2!

The Functional Programming languages run on the Computer!

Conclusion

We need to be able to model variables and mutation!

What is the problem we are trying to solve?

Compare how sorting a list is done in Rust vs Haskell:

```rust
// rust mutates in place
let mut list = vec![2,1,3,5,4];
list.sort();
```

```haskell
-- haskell produces a new list
let list = [2,1,3,5,4]
let sorted = sort list
```

Effects in functional programming are explicit. They are returned by functions.

What is the problem we are trying to solve?

In Functional Programming effects are explicit and values are immutable.

```
// mutate in place
let mut a = ...;
mutate(a);
```

```
-- produces new value
let a = ...
let newA = update a
```

How can we model the processor?

What is the problem we are trying to solve?

In imperative languages you can work with values you are interested in.
State is handled implicitly.

```
// definition of execute command for processor
// mutation is implicit
fn execute(mut self, command: Command) -> Result

let processor = Processor::new();

let contents = processor.execute(ReadFromFile("input.txt"));
processor.execute(Print("Done!"));
let result = processor.execute(ProcessText);
processor.execute(WriteFile("output.txt"));
```

What is the problem we are trying to solve?

In functional languages you can work with values you are interested in, but you also need to worry about state!

```
-- definition of execute command for processor
-- mutation is explicitly part of the result
execute : Processor → Command → (Result, Processor)

let p0 = Processor.new

let (contents, p1) = execute p0 (ReadFromFile "input.txt")
let (_, p2) = execute p1 (Print "Done!")
let (result, p3) = execute p2 ProcessText
let (_, p4) = execute p3 (WriteFile "output.txt")
```

What is the problem we are trying to solve?

In functional languages you can work with values you are interested in, but you also need to worry about state!

```
-- definition of execute command for processor
-- mutation is explicitly part of the result
execute : Processor → Command → (Result, Processor)

let p0 = Processor.new

let (contents, p1) = execute p0 (ReadFromFile "input.txt")
let (_, p2) = execute p1 (Print "Done!")
let (result, p3) = execute p2 ProcessText
let (_, p4) = execute p3 (WriteFile "output.txt")
```

Problem 1: Time Travel

Nothing is stopping you from using an old value of p that is no longer correct.

What is the problem we are trying to solve?

In functional languages you can work with values you are interested in, but you also need to worry about state!

```
-- definition of execute command for processor
-- mutation is explicitly part of the result
execute : Processor → Command → (Result, Processor)

let p0 = Processor.new

let (contents, p1) = execute p0 (ReadFromFile "input.txt")
let (_, p2) = execute p1 (Print "Done!")
let (result, p3) = execute p2 ProcessText
let (_, p4) = execute p3 (WriteFile "output.txt")
```

Problem 1: Time Travel

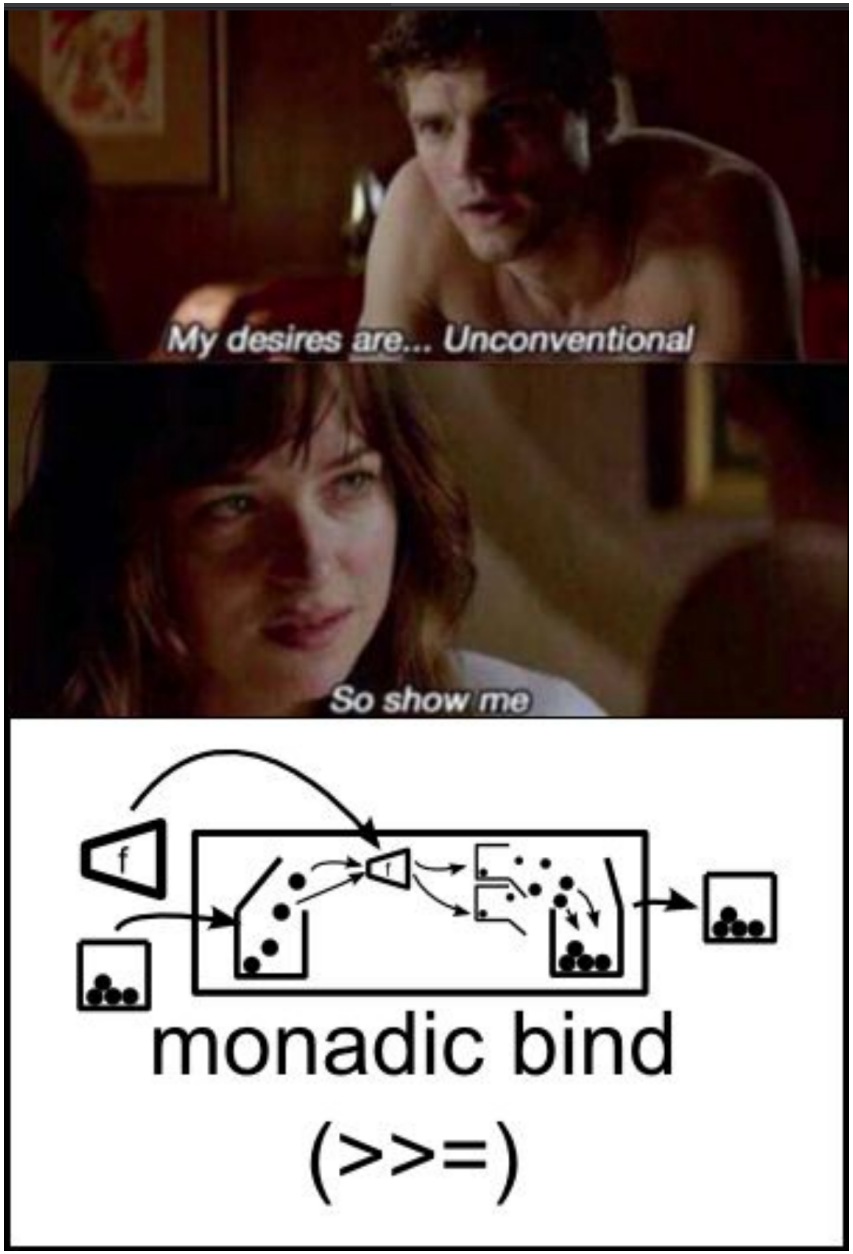
Nothing is stopping you from using an old value of p that is no longer correct.

Problem 2: Multiverse

You can pass the same p to multiple functions.

join, bind, composition

join, bind, composition



join, bind, composition

When discussing Monads, we always deal with join and bind.

```
class Monad m where
  pure : a -> m a
  bind : m a -> (a -> m b) -> mb
```

However, the phrase everyone is repeating is:
“a monad is a monoid in the category of endofunctors” - Wikipedia

join, bind, composition

When discussing Monads, we always deal with join and bind.

```
class Monad m where
  pure : a -> m a
  bind : m a -> (a -> m b) -> mb
```

However, the phrase everyone is repeating is:
“a monad is a monoid in the category of endofunctors” - Wikipedia
Where is the monoid?

join, bind, composition

```
class Monoid m where
  combine : μ → μ → μ
  empty : μ

-- combine empty x = x
-- combine x empty = x
```

join, bind, composition

```
class Monoid m where
  combine : μ → μ → μ
  empty : μ

-- combine empty x = x
-- combine x empty = x
```

Kleisli Composition

```
(>=>) : (α → m β) → (β → m γ) → (α → m γ)
pure : α → m α

-- pure >=> f = f
-- f >=> pure = f
```

join, bind, composition

```
class Monoid m where
  combine : μ → μ → μ
  empty : μ

-- combine empty x = x
-- combine x empty = x
```

Kleisli Composition

```
(>=>) : (α → m β) → (β → m γ) → (α → m γ)
pure : α → m α

-- pure >=> f = f
-- f >=> pure = f
```

If you have an implementation of join, bind or compose, you can derive all three!

join, bind, composition

```
-- helper functions
id : α → α
map : (α → β) → m α → m β
```

join, bind, composition

```
-- helper functions
id : α → α
map : (α → β) → m α → m β

-- (1) define bind with join
-- bind : m α → (α → m β) → m β
bind a f :=
  -- a : m α
  -- map f a : m (m β)
  join (map f a) -- : m β
```

join, bind, composition

```
-- helper functions
id : α → α
map : (α → β) → m α → m β

-- (1) define bind with join
-- bind : m α → (α → m β) → m β
bind a f :=
  -- a : m α
  -- map f a : m (m β)
  join (map f a) -- : m β

-- (2) define join with bind
-- join : m (m α) → m α
join a :=
  -- bind a : (m α → m β) → m β
  bind a id -- : m α
```

join, bind, composition

```
-- helper functions
id : α → α
map : (α → β) → m α → m β
```

join, bind, composition

```
-- helper functions
id : α → α
map : (α → β) → m α → m β

-- (3) define composition with bind
-- (>=>) : (α → m β) → (β → m γ) → (α → m γ)
f >=> g :=
  -- λ a => a : (α → α)
  -- λ a => f a : (α → m β)
  λ a => bind (f a) g -- : (α → m γ)

-- (4) define join with composition
join := id >=> id

-- proof of this and the missing definitions are left
-- as an exercise to the reader
```

Simplified definition of a Monad

```
class Monad (m : Type → Type) where
  pure : α → m α
  bind : m α → (α → m β) → m β


-- Note: (bind a f) is equivalent to (a >>= f)
```

Definition of a Monad from Prelude.lean

```
class Monad (m : Type u → Type v) : Type (max (u+1) v)
  extends Applicative m, Bind m
  where

  map f x := bind x (pure ∘ f)
  -- also 3 other definitions for seq, seqLeft and seqRight
```

Definition of a Monad from Prelude.lean

```
class Monad (m : Type u → Type v) : Type (max (u+1) v)
  extends Applicative m, Bind m
  where

  map f x := bind x (pure ∘ f)
  -- also 3 other definitions for seq, seqLeft and seqRight

class Bind (m : Type u → Type v) where
  bind : {α β : Type u} → m α → (α → m β) → m β
```

Definition of a Monad from Prelude.lean

```
class Monad (m : Type u → Type v) : Type (max (u+1) v)
  extends Applicative m, Bind m
  where

  map f x := bind x (pure ∘ f)
  -- also 3 other definitions for seq, seqLeft and seqRight

class Bind (m : Type u → Type v) where
  bind : {α β : Type u} → m α → (α → m β) → m β

class Applicative (f : Type u → Type v)
  extends Functor f, Pure f, Seq f, SeqLeft f, SeqRight f
  where
  -- definitions for map, seqLeft and seqRight

class Functor (f : Type u → Type v) : Type (max (u+1) v)
  where
  map : {α β : Type u} → (α → β) → f α → f β
```

Definition of a Monad from Prelude.lean

```
class Monad (m : Type u → Type v) : Type (max (u+1) v)
  extends Applicative m, Bind m
  where

  map f x := bind x (pure ∘ f)
  -- also 3 other definitions for seq, seqLeft and seqRight

class Bind (m : Type u → Type v) where
  bind : {α β : Type u} → m α → (α → m β) → m β

class Applicative (f : Type u → Type v)
  extends Functor f, Pure f, Seq f, SeqLeft f, SeqRight f
  where
  -- definitions for map, seqLeft and seqRight

class Functor (f : Type u → Type v) : Type (max (u+1) v)
  where
  map : {α β : Type u} → (α → β) → f α → f β

class Pure (f : Type u → Type v) where
  pure {α : Type u} : α → f α
```

The Option Monad

```
-- definition of option
inductive Optiune (α : Type) : Type where
  | nimic
  | ceva (val : α)

-- monad instance of option
instance : Monad Optiune where
  pure x := Optiune.ceva x

  bind opt f := match opt with
    | Optiune.nimic => Optiune.nimic
    | Optiune.ceva val => f val
```

The Result Monad

```
-- definition of result
inductive Rezultat (ϵ : Type) (α : Type) : Type where
  | eroare (e : ϵ)
  | succes (a : α)

-- monad instance of result
instance : Monad (Rezultat ϵ) where
  pure x := Rezultat.succes x

  bind rez f := match rez with
    | Rezultat.eroare e => Rezultat.eroare e
    | Rezultat.succes r => f r
```

The List Monad

```
-- definition of list
inductive Lista (α : Type) : Type where
  | nimic
  | elem (val : α) (rest : Lista α)

-- example of function on list
def sum (nrs : Lista Nat) : Nat :=
  match nrs with
    | Lista.nimic => 0
    | Lista.elem val rest => val + sum rest
```

The List Monad

```
-- definition of list
inductive Lista (α : Type) : Type where
  | nimic
  | elem (val : α) (rest : Lista α)

-- monad instance implementation
instance : Monad Lista where
  pure x := Lista.elem x Lista.nimic
  bind xs f := list_bind xs f

def list_bind (as : Lista α) (f : α -> Lista β) : Lista β :=
  match as with
    | Lista.nimic => Lista.nimic
    | Lista.elem v r =>  list_bind r f ++ f v
```

The Writer Monad Definition

```
structure Writer (β : Type) (α : Type) where
  run : α × β

def mkWriter (b : β) (a : α) : Writer β α := {
  run := (a, b)
}

instance [Append β] [EmptyCollection β] : Monad (Writer β)
  where
  pure x := mkWriter ∅ x

  bind w f :=
    let (a, s1) := w.run
    let (b, s2) := (f a).run
    mkWriter (s1 ++ s2) b
```

The Writer Monad Usage

```
def write (s : String) : Writer (List String) Unit :=
  mkWriter [s] ()

def logging: Writer (List String) Nat := do
  write "program starting:"
  let first <- pure 10
  write "created first..."
  let second <- pure 20
  write "created second..."
  let sum := first + second
  write "done."
  pure sum

#eval logging.run
-- ( 30
-- , [ "program starting:"
-- , "created first..."
-- , "created second..."
-- , "done."]
-- )
```

The Reader Monad Definition

```
structure Reader (σ : Type) (α : Type) where
  run : σ → α

def mkReader (f : σ → α) : Reader σ α := {
  run := f
}

instance : Monad (Reader σ) where
  pure x := mkReader (λ _ => x)

  bind r f := mkReader (λ s => (f (r.run s)).run s)
```

The Reader Monad Usage

```
def read : Reader α α :=
  mkReader id

def calc_eur : Reader Float Float := do
  let initial_ron <- pure 1000
  let rate <- read
  let initial_eur := initial_ron * rate
  pure (initial_eur * 1.25)

#eval calc_eur.run 5.0934
-- 6366.750000
```

The State Monad Definition

```
structure State (σ : Type) (α : Type) where
  run : σ → σ × α

def mkState (f : σ → σ × α) : State σ α := {
  run := f
}

instance : Monad (State σ) where
  pure x := mkState (λ s => (s, x))

  bind r f :=
    mkState ( λ s =>
      let (s', a) := r.run s
      let (s'', b) := (f a).run s'
      (s'', b)
    )
```

The State Monad Usage

```
def get : State σ σ := mkState (λ s => (s, s))

def set (s : σ) : State σ Unit := mkState (λ _ => (s, ()))

def get_random : State Nat Nat := do
  let seed <- get
  let new_seed := (1103515245 * seed + 12345) % (2^31)
  set new_seed
  pure new_seed

def random_numbers : State Nat Nat := do
  let first <- get_random
  let second <- get_random
  let third <- get_random
  let extra := first - second
  pure (first + second + third + extra)

#eval (random_numbers.run 42).snd
-- 3501668807
```

The End

Thank you for your attention, here are some nice sources:

1. (Article) *"On Computable Numbers, with an Aplication to the Entscheidungsproblem"* by Alan Turing, 1936
2. (Article) *"An Unsolvable Problem of Elementary Number Theory"* by Alonzo Church, 1936
3. (Article) *"Comprehending Monads"* by Philip Wadler, 1990
4. (Youtube) *"Category Theory Course"* by Bartosz Milewski, 2016
5. (Youtube) *"Functional programming design patterns by Scott Wlaschin"* uploaded by Ivan Plyusnin, 2015
6. (Youtube) *"Simon Peyton Jones - Haskell is useless"* uploaded by bunidanoable, 2011