Monads II: Theory & Applications

Hype for Types

March 23, 2026

Review: The Pattern of "And Then"

Recap: The Core Abstraction

The Payoff

Monads separate the **Logic** (your code) from the **Plumbing** (hidden effects like failure, state, or latency).

Recap: The Core Abstraction

The Payoff

Monads separate the **Logic** (your code) from the **Plumbing** (hidden effects like failure, state, or latency).

- `'a t`: A value in a context (failure, state, or a value that is late).

Recap: The Core Abstraction

The Payoff

Monads separate the **Logic** (your code) from the **Plumbing** (hidden effects like failure, state, or latency).

- `'a t`: A value in a context (failure, state, or a value that is late).
- `return x`: Lifts a value into the context.

Recap: The Core Abstraction

The Payoff

Monads separate the **Logic** (your code) from the **Plumbing** (hidden effects like failure, state, or latency).

- `'a t`: A value in a context (failure, state, or a value that is late).
- `return x`: Lifts a value into the context.
- `bind` **(>>=)**: Chaining. It says: "Unwrap the result of the last step, and pass it to the next function."

Recap: The Core Abstraction

The Payoff

Monads separate the **Logic** (your code) from the **Plumbing** (hidden effects like failure, state, or latency).

- `'a t`: A value in a context (failure, state, or a value that is late).
- `return x`: Lifts a value into the context.
- `bind` **(>>=)**: Chaining. It says: "Unwrap the result of the last step, and pass it to the next function."

Recap: The Core Abstraction

The Payoff

Monads separate the **Logic** (your code) from the **Plumbing** (hidden effects like failure, state, or latency).

- `'a t`: A value in a context (failure, state, or a value that is late).
- `return x`: Lifts a value into the context.
- `bind` **(>>=)**: Chaining. It says: "Unwrap the result of the last step, and pass it to the next function."

The SML Signature

```
signature MONAD = sig
  type 'a t
  val return : 'a -> 'a t
  val bind   : 'a t * ('a -> 'b t) -> 'b t
end
```

The Laws: Why "Trust" Matters

Why do we need laws?

An interface tells the compiler how parts fit together. **Laws** tell humans how the program behaves when refactored.

Why do we need laws?

An interface tells the compiler how parts fit together. **Laws** tell humans how the program behaves when refactored.

If a monad follows the laws, it is **referentially transparent**. You can swap code around without changing the meaning of the program.

Why do we need laws?

An interface tells the compiler how parts fit together. **Laws** tell humans how the program behaves when refactored.

If a monad follows the laws, it is **referentially transparent**. You can swap code around without changing the meaning of the program.

The three laws are:

1. **Left Identity**
2. **Right Identity**
3. **Associativity**

Law 1: Left Identity

The Law

`return x >>= f` $\equiv$ `f x`

Law 1: Left Identity

The Law

`return x >>= f` $\equiv$ `f x`

Intuition: `return` should be a "no-op" wrapper. If you wrap a value and immediately feed it into a function, it should be the same as just calling the function.

Law 1: Left Identity

The Law

`return x >>= f` $\equiv$ `f x`

Intuition: `return` should be a "no-op" wrapper. If you wrap a value and immediately feed it into a function, it should be the same as just calling the function.

Breaking the Law: The Loggy Return

```
fun return x = (print "Creating Value!"; x)
```

Now, `return 5 >>= f` prints a message, but `f 5` does not.

Law 1: Left Identity

The Law

`return x >>= f ≡ f x`

Intuition: `return` should be a "no-op" wrapper. If you wrap a value and immediately feed it into a function, it should be the same as just calling the function.

Breaking the Law: The Loggy Return

```
fun return x = (print "Creating Value!"; x)
```

Now, `return 5 >>= f` prints a message, but `f 5` does not.

The Problem: You can't refactor! You can no longer replace a redundant `return` with a direct function call without changing the program's output (the logs).

Law 2: Right Identity

The Law

`m >>= return` $\equiv$ `m`

Law 2: Right Identity

The Law

`m >>= return` $\equiv$ `m`

Intuition: If you have a monadic value, unwrap it, and immediately re-wrap it with `return`, you should end up exactly where you started.

Law 2: Right Identity

The Law

`m >>= return` $\equiv$ `m`

Intuition: If you have a monadic value, unwrap it, and immediately re-wrap it with `return`, you should end up exactly where you started.

Breaking the Law: The Sticky State

Suppose our Monad tracks a counter. Every time `return` is called, it increments a global `count`. Now `m >>= return` leaves the counter higher than `m` did alone.

Law 2: Right Identity

The Law

`m >>= return` $\equiv$ `m`

Intuition: If you have a monadic value, unwrap it, and immediately re-wrap it with `return`, you should end up exactly where you started.

Breaking the Law: The Sticky State

Suppose our Monad tracks a counter. Every time `return` is called, it increments a global `count`. Now `m >>= return` leaves the counter higher than `m` did alone.

The Problem: This ruins **Dead Code Elimination**. A compiler should be able to remove `x >>= return` if the result isn't used. If laws are broken, the compiler accidentally changes your program logic.

Law 3: Associativity

The Law

```
(m >>= f) >>= g ≡ m >>= (fn x => f x >>= g)
```

Law 3: Associativity

The Law

```
(m >>= f) >>= g ≡ m >>= (fn x => f x >>= g)
```

Intuition: The grouping of operations shouldn't matter. "Do A, then B, then C" should be the same regardless of parentheses.

Law 3: Associativity

The Law

```
(m >>= f) >>= g ≡ m >>= (fn x => f x >>= g)
```

Intuition: The grouping of operations shouldn't matter. "Do A, then B, then C" should be the same regardless of parentheses.

Breaking the Law

If grouping changes the result, you cannot take two steps and pull them into a helper function safely.

Law 3: Associativity

The Law

```
(m >>= f) >>= g ≡ m >>= (fn x => f x >>= g)
```

Intuition: The grouping of operations shouldn't matter. "Do A, then B, then C" should be the same regardless of parentheses.

Breaking the Law

If grouping changes the result, you cannot take two steps and pull them into a helper function safely.

The Problem: This ruins **Composition**. Associativity is what allows us to write flat `let%bind` or `do` blocks without worrying about nesting.

The Theory: Kleisli Arrows

The Composition Problem

Standard function composition: `f : A -> B` and `g : B -> C` yields `g o f : A -> C`.

The Composition Problem

Standard function composition: `f : A -> B` and `g : B -> C` yields `g o f : A -> C`.

But effectful functions return `M B`. You can't plug `A -> M B` into `B -> M C`. **The types don't match**.

The Composition Problem

Standard function composition: `f : A -> B` and `g : B -> C` yields `g o f : A -> C`.

But effectful functions return `M B`. You can't plug `A -> M B` into `B -> M C`. **The types don't match**.

The Solution

A **Kleisli arrow** is a function of type `'a -> 'b t`.
Monadic Bind is the tool that allows these "crooked" arrows to compose into a straight line.

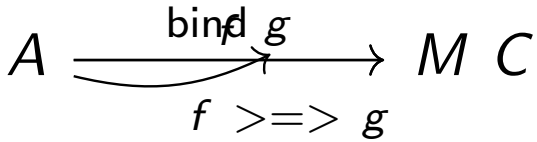
The Composition Problem

Standard function composition: `f : A -> B` and `g : B -> C` yields `g o f : A -> C`.

But effectful functions return `M B`. You can't plug `A -> M B` into `B -> M C`. **The types don't match**.

The Solution

A **Kleisli arrow** is a function of type `'a -> 'b t`.
Monadic Bind is the tool that allows these "crooked" arrows to compose into a straight line.



The State Monad

The Problem: The "Counter" Noise

Suppose we are writing a compiler. We need to generate unique names: `v1`, `v2`, `v3`...

The Problem: The "Counter" Noise

Suppose we are writing a compiler. We need to generate unique names: v1, v2, v3...

```
fun compile (Exp1, count) =
  let
    val (res1, count1) = compileStep1 (Exp1, count)
    val (res2, count2) = compileStep2 (res1, count1)
  in (res2, count2) end
```

The Problem: The "Counter" Noise

Suppose we are writing a compiler. We need to generate unique names: v1, v2, v3...

```
fun compile (Exp1, count) =
  let
    val (res1, count1) = compileStep1 (Exp1, count)
    val (res2, count2) = compileStep2 (res1, count1)
  in (res2, count2) end
```

The Pain

You're manually threading count1, count2 through every line. If you accidentally pass the old count instead of count1 to Step 2, you have a bug.

The Problem: The "Counter" Noise

Suppose we are writing a compiler. We need to generate unique names: v1, v2, v3...

```
fun compile (Exp1, count) =
  let
    val (res1, count1) = compileStep1 (Exp1, count)
    val (res2, count2) = compileStep2 (res1, count1)
  in (res2, count2) end
```

The Pain

You're manually threading `count1`, `count2` through every line. If you accidentally pass the old `count` instead of `count1` to Step 2, you have a bug.

Monadic Insight: A stateful computation is just a function: `'s -> ('a * 's)`.

Concurrency vs. Parallelism

Structure vs. Execution

People often use these interchangeably, but they are different tools.

Structure vs. Execution

People often use these interchangeably, but they are different tools.

- **Parallelism**: Multiple people working on multiple tasks at once. (Multiple CPU cores doing independent math).

Structure vs. Execution

People often use these interchangeably, but they are different tools.

- **Parallelism**: Multiple people working on multiple tasks at once. (Multiple CPU cores doing independent math).
- **Concurrency**: One person multitasking between multiple tasks. Progress is made on both, but only one is active at a time.

Structure vs. Execution

People often use these interchangeably, but they are different tools.

- **Parallelism**: Multiple people working on multiple tasks at once. (Multiple CPU cores doing independent math).
- **Concurrency**: One person multitasking between multiple tasks. Progress is made on both, but only one is active at a time.

Structure vs. Execution

People often use these interchangeably, but they are different tools.

- **Parallelism**: Multiple people working on multiple tasks at once. (Multiple CPU cores doing independent math).
- **Concurrency**: One person multitasking between multiple tasks. Progress is made on both, but only one is active at a time.

Analogy: The Video Project

- **Exporting Video (CPU-bound)**: Hard math. Your CPU is at 100%. You need **Parallelism** to speed this up.
- **Downloading Video (I/O-bound)**: Your CPU is at 0%. It's sitting idle waiting for the network. You need **Concurrency** to do other work while you wait.

The Human Scale of Waiting

Why do we need Monadic Concurrency? Because modern CPUs are **too fast**.

The Human Scale of Waiting

Why do we need Monadic Concurrency? Because modern CPUs are **too fast**.

The Comparison

If 1 CPU instruction (reading a register) took **1 second**:

- Reading from RAM would take **4 minutes**.
- A Network request to a nearby server would take **4 months**.

The Human Scale of Waiting

Why do we need Monadic Concurrency? Because modern CPUs are **too fast**.

The Comparison

If 1 CPU instruction (reading a register) took **1 second**:

- Reading from RAM would take **4 minutes**.
- A Network request to a nearby server would take **4 months**.

The Goal: Instead of the CPU sitting idle for "4 months," we want a way to **pause** a function, save its state, and **resume** it when the mail arrives.

Async/Await — Concurrency as a Monad

Future¡T¿ is a Monad

In Rust, a `Future<T>` is a value that will eventually become a `T`.

- `Option<T>`: Value might be **missing**.
- `Future<T>`: Value might be **late**.

Future¡T¿ is a Monad

In Rust, a `Future<T>` is a value that will eventually become a `T`.

- `Option<T>`: Value might be **missing**.
- `Future<T>`: Value might be **late**.

The Monadic Mapping

- `return`: The `async { }` block. It wraps a value in a Future that is "Ready."
- `bind`: The `.await` keyword. It says: "Pause this function. When the Future is ready, take the inner value and continue."

The Type System as a State Machine

How does Rust "remember" where it was when it pauses? The compiler transforms your function into an **enum-based state machine**.

```
// Your Code:
let user = fetch_user(id).await;
let post = fetch_post(user.id).await;

// The Compiler's Transformation (Simplified):
enum MyFunctionState {
    Start { id: u32 },
    WaitingForUser { fut: FetchUserFuture },
    WaitingForPost { user: User, fut: FetchPostFuture },
    Done
}
```

The Type System as a State Machine

How does Rust "remember" where it was when it pauses? The compiler transforms your function into an **enum-based state machine**.

```
// Your Code:
let user = fetch_user(id).await;
let post = fetch_post(user.id).await;

// The Compiler's Transformation (Simplified):
enum MyFunctionState {
    Start { id: u32 },
    WaitingForUser { fut: FetchUserFuture },
    WaitingForPost { user: User, fut: FetchPostFuture },
    Done
}
```

Payoff

The "paused" state is a **Type**. The compiler calculates exactly how much memory it needs. Because it's a Monad, the type system can statically track exactly which functions can "pause" and which cannot.

Why Rust's Type System is Robust

Rust uses the Monad pattern to make concurrency safe:

Why Rust's Type System is Robust

Rust uses the Monad pattern to make concurrency safe:

- **Type Mismatch**: You cannot use a `Future<T>` as a `T`. You **must** await (bind) it.

Why Rust's Type System is Robust

Rust uses the Monad pattern to make concurrency safe:

- **Type Mismatch**: You cannot use a `Future<T>` as a `T`. You **must** await (bind) it.
- **Effect Visibility**: You cannot accidentally call a months-long network request inside a high-speed synchronous loop. The types won't match!

Why Rust's Type System is Robust

Rust uses the Monad pattern to make concurrency safe:

- **Type Mismatch**: You cannot use a `Future<T>` as a `T`. You **must** await (bind) it.
- **Effect Visibility**: You cannot accidentally call a months-long network request inside a high-speed synchronous loop. The types won't match!
- **No Runtime Overhead**: Because the State Machine is just an enum, there is no need for heavy "Green Threads" or hidden stack-copying.

What's Next

Next Week: The SML Bee!

High-speed, typed competition. Review your Bind, your Return, and your pattern matching.

Next Week: The SML Bee!

High-speed, typed competition. Review your Bind, your Return, and your pattern matching.

No lecture. Just code.