Functional Programming 101

Introduction to Functional Thinking & Haskell

Vijay Anant & Raghu Ugare

2026 Edition

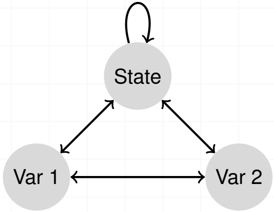
Software is Hard

- We keep inventing new tools, but the bugs stay the same.
- Programs are growing faster than our ability to reason about them.
- **The Core Problem:** Shared mutable state.
- When "Value A" can change anywhere in the jungle, you can’t trust "Value A" anywhere else.

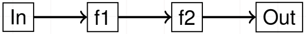
The State Trap

- Imperative programming is about "Boxes."
- We name a box, and then we spend the rest of our time changing what is inside it.
- This hidden state makes systems non-deterministic.

Imperative Spaghetti



Functional Flow



The Antidote: Haskell

- We use Haskell not just as a language, but as a **laboratory** for functional concepts.
- **Why?** Because Haskell enforces these rules strictly.
- In other languages, you *can* do FP. In Haskell, you *must*.

Pillar 1: Immutability

- In a functional world, data doesn’t change.
- $x = 5$ is not an assignment; it is a declaration of truth.
- **Values vs. Boxes:** If data never changes, two parts of the system can never disagree about what it was.

Haskell: Definitions, Not Assignments

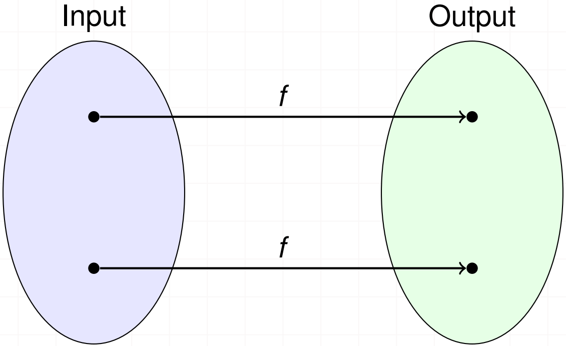
```haskell
-- Immutability is the default
val = 10

-- This would be a COMPILE ERROR:
-- val = 11
```

Once a value is created, it is carved in stone. There is no "set" keyword.

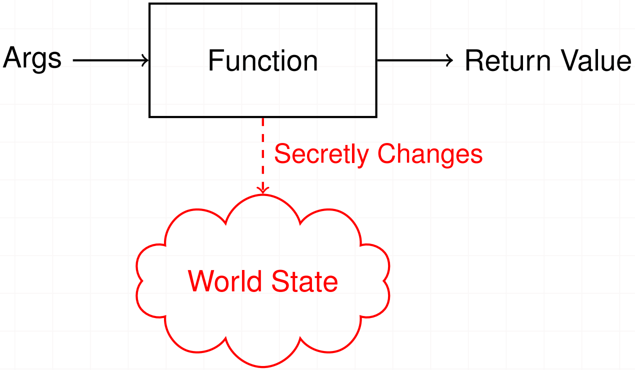
Pillar 2: Purity (Mapping Sets)

A Pure Function is a relationship between two sets.



Crucial: For a given input, it points to **exactly one** output. Always.

The Enemy: Side Effects



- Side effects are hidden dependencies (printing, saving, launching missiles).
- They break the mapping contract.

Haskell: Types as Contracts

In Haskell, the type signature is a legally binding contract.

```haskell
-- The signature guarantees:
-- 1. I need two Ints.
-- 2. I will give you an Int.
-- 3. I will DO NOTHING ELSE (no network, no disk, no mutation).
add :: Int -> Int -> Int
add x y = x + y
```

Pillar 3: Higher-Order Functions

- Functions are values, just like integers.
- We pass "What to do" into standard containers of "How to do it."

The Pipeline (Composed)

The Loop (Manual)

The Ripple Effect of Change

Imperative

```
total = 0
for num in numbers:
  if num % 2 == 0:
    total += num

total_squares = 0
for num in numbers:
  if num % 2 == 0:
    total_squares += num * num
```

Functional

```
total = sum (filter even numbers)

total_of_squares = sum (map (^2)
↪   (filter even numbers))
```

In the functional style, the original functions (`sum`, `filter`) are untouched. We compose, we don’t edit.

Haskell: Functions as Data

We use standard functions to replace imperative loops.

```haskell
-- map: transform every item
map (*2) [1, 2, 3]
-- Result: [2, 4, 6]

-- filter: keep what matters
filter (>5) [1..10]
-- Result: [6, 7, 8, 9, 10]
```

Pillar 4: Modeling with Types (ADTs)

We use types to represent the reality of our business logic.

- **The Core Idea:** We are making illegal states impossible instead of handling them at runtime.
- You don’t write checks for things that cannot exist.

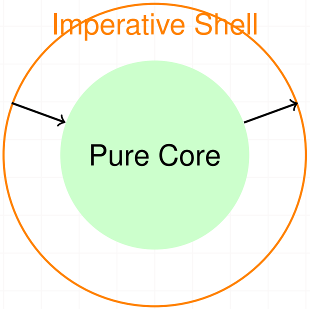
```
-- A value is either a Success OR a Failure
data Result = Success String | Failure Error
```

Haskell: Pattern Matching

How do we work with these types? We don’t use if-else; we match the shape of the data.

```haskell
-- Handling the choice with Pattern Matching
render :: Result -> String
render result = case result of
    Success msg -> "Done: " ++ msg
    Failure err -> "Error: " ++ show err
```

Architecture: Functional Core, Imperative Shell



- **Functional Core:** The pure logic. Deterministic, easy to test.
- **Imperative Shell:** The messy outside world.
- **Strategy:** Push side effects to the edge.

What is FP actually? (Summary)

- It is the shift from **Commands** (telling the computer *how* to change memory) to **Expressions** (describing *what* a value is).
- It is the strict avoidance of **Mutation**. We don't change state; we calculate new states from old ones.
- It is the practice of building systems by **Composing** simple, predictable functions into complex ones.

Resolving the Original Pains

- **“Bugs that appear randomly”** → Determinism from purity.
- **“Works locally, fails in prod”** → Explicit inputs and effects.
- **“Can’t trust value A”** → Immutability.
- **“Concurrency is scary”** → No shared mutable state.

Questions?

?

Thank You

Thank You!