

Chapter 7: Computations lifted to a functor context II. Monads and semimonads

Part 1: Practical work with monads and semimonads

Sergei Winitzki

Academy by the Bay

2018-03-11

Computations within a functor context: Semimonads

Intuitions behind adding more “generator arrows”

Example:

$$\sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n f(i, j, k)$$

Using Scala's `for`/`yield` syntax (“functor block”)

```
(for { i ← 1 to n          (1 to m).flatMap { i ⇒
    j ← 1 to n             (1 to n).flatMap { j ⇒
    k ← 1 to n             (1 to n).map { k ⇒
    } yield f(i, j, k)      f(i, j, k)
  }.sum                    } } }.sum
```

- `map` replaces the last left arrow, `flatMap` replaces other left arrows
 - ▶ When the functor is *also* filterable, we can use “`if`” as well
- Standard library defines `flatMap()` as replacement of `map() ∘ flatten`
 - ▶ `(1 to n).map(j ⇒ ...).flatten` is `(1 to n).flatMap(j ⇒ ...)`
- Functors having `flatMap`/`flatten` are “flattenable” or **semimonads**
 - ▶ Most of them also have method `pure: A ⇒ F[A]` and so are **monads**
 - ★ The method `pure` is not relevant in the functor block
 - ★ We will not need `pure` in this part of the tutorial; focus on semimonads

What is `flatMap` doing with the data in a collection?

Consider this schematic code using `Seq` as the container type:

```
val result = for {
  i ← 1 to m
  j ← 1 to n
  x = f(i,j)
  k ← 1 to p
  y = g(i,j,k)
} yield h(x,y)

val result = {
  (1 to m).flatMap { i ⇒
    (1 to n).flatMap { j ⇒
      val x = f(i,j)
      (1 to p).map { k ⇒
        val y = g(i,j,k)
        h(x,y) } } } }
```

Computations are repeated for all i , for all j , etc., from each collection

- All “generator lines” must use the same container type
 - ▶ Each generator line finally computes a container of *the same* type
 - ▶ The total number of resulting data items is $\leq m * n * p$
 - ▶ All the resulting data items must fit within *the same* container type!
 - ▶ The set of *container capacity counts* must be closed under multiplication
- What container types have this property?
 - ▶ `Seq`, `NonEmptyList` – can hold *any* number of elements \geq min. count
 - ▶ `Option`, `Either`, `Try`, `Future` – can hold 0 or 1 elements (“pass/fail”)
 - ▶ “Tree-like” containers, e.g. can hold only 3, 6, 9, 12, ... elements
 - ▶ “Non-standard” containers: $F^A \equiv \text{String} \Rightarrow A$; $F^A \equiv (A \Rightarrow \text{Int}) \Rightarrow \text{Int}$

Worked examples: List-like monads

`Seq`, `NonEmptyList`, `Iterator`, `Stream`

Typical tasks for “list-like” monads:

- Create a list of all combinations or all permutations of a sequence
- Traverse a “solution tree” with DFS and filter out incorrect solutions
 - ▶ Can use eager (`Seq`) or lazy (`Iterator`, `Stream`) evaluation strategies
 - ▶ Usually, list-like containers have many additional methods
 - ★ `append`, `prepend`, `concat`, `fill`, `fold`, `scan`, etc.

Worked examples: see code

- 1 All permutations of `Seq("a", "b", "c")`
- 2 All subsets of `Set("a", "b", "c")`
- 3 All subsequences of length 3 out of a given sequence
- 4 Generalize examples 1-3 to support arbitrary length n instead of 3
- 5 All solutions of the “8 queens” problem
- 6 Generalize example 5 to solve n -queens problem
- 7 Transform Boolean formulas between CNF and DNF

Intuitions for pass/fail monads

Option, Either, Try, Future

- Container F^A can hold $n = 1$ or $n = 0$ values of type A
- Such containers will have methods to create “pass” and “fail” values

Schematic example of a functor block program using the `Try` functor:

```
val result: Try[A] = for { // computations in the Try functor
  x ← Try(...) // first computation; may fail
  y = f(x) // no possibility of failure in this line
  if p(y) // the entire expression will fail if this is false
  z ← Try(g(x, y)) // may fail here
  r ← Try(...) // may fail here as well
} yield r // r is of type A, so result is of type Try[A]
```

- Computations may yield a result ($n = 1$), or may fail ($n = 0$)
- The functor block chains several such computations *sequentially*
 - ▶ Computations are sequential even if using the `Future` functor!
 - ▶ Once any computation fails, the entire functor block fails ($0 * n = 0$)
 - ▶ Only if *all* computations succeed, the functor block returns *one* value
 - ▶ Filtering can also make the entire expression fail
- “Flat” functor block replaces a chain of nested `if/else` or `match/case`

Worked examples: Pass/fail monads

Type constructors:

- `Option[A]` $\equiv 1 + A$
- `Either[Z, A]` $\equiv Z + A$
- `Try[A]` $\equiv \text{Either}[\text{Throwable}, A]$

Typical tasks for pass/fail monads:

- Perform a linear sequence of computations that may fail
- Avoid crashing on failure, instead return an *error value*

Worked examples: see code

- 1 Read values of Java properties, checking that they all exist
- 2 Obtain values from `Future` computations in sequence
- 3 Make arithmetic safe by returning error messages in `Either`
- 4 Fail less: chain computations that may throw an exception

Worked examples: Tree-like monads

Examples of tree-like type constructors:

- $F^A \equiv A + F^A \times F^A$ (binary tree, recursive)
- $F^A \equiv A + S^{F^A}$ (S -shaped tree, where S is a functor)
- $F^A \equiv A \times A + F^A \times F^A$ (binary tree with binary leaves)
- $F^A \equiv S^A + S^{F^A}$ (S -shaped tree with S -shaped leaves)

Typical tasks for tree-like monads:

- Traverse a tree, graft subtrees at leaves
- Substitute subexpressions in a syntax tree

Worked examples: see code

- 1 Implement a tree of `String` properties with arbitrary branching
- 2 Implement variable substitution for a simple arithmetic language

Example of a *non-tree-like* type constructor:

- $F^A \equiv A + A \times A + A \times A \times A \times A + \dots$ (powers of 2, *non-recursive*)

Worked examples: Single-value monads

- Container holds exactly 1 value, together with a “context”
- Usually, methods exist to insert a value and to work with the “context”

Typical tasks for single-value monads:

- Collecting extra information about computations along the way
- Chaining computations with a nonstandard evaluation strategy

Examples: see code

- 1 **Writer**: Perform computations and log information about each step
 - ▶ $\text{Writer}^A \equiv A \times W$ where W is a monoid or a semigroup
- 2 **Reader**: Read-only context, or dependency injection
 - ▶ $\text{Reader}^A \equiv E \Rightarrow A$ where E represents the “environment”
- 3 **Eval**: Perform a sequence of lazy or memoized computations
 - ▶ $\text{Eval}^A \equiv A + (1 \Rightarrow A)$
- 4 **Cont**: A chain of asynchronous operations
 - ▶ $\text{Cont}^A \equiv (A \Rightarrow R) \Rightarrow R$ where R is the fixed “result” type
- 5 **State**: A sequence of steps that update state while returning results
 - ▶ $\text{State}^A \equiv S \Rightarrow A \times S$ where S is the fixed “state” value type

Exercises I

- 1 Compute all subsequences of length 3 out of the sequence (1 to m)
- 2 Implement a semimonad instance for $F^A \equiv E \Rightarrow A \times W$ where W is a semigroup