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

Part 1: Intuitions, examples, use cases

Sergei Winitzki

Academy by the Bay

2018-03-25

Computations within a functor context: Semimonads

Intuitions behind adding more "generator arrows"

Example of nested iterations:

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

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

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

```
(1 to n).flatMap { i ⇒
  (1 to n).flatMap { j ⇒
    (1 to n).map { k ⇒
      f(i, j, k)
}}}.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

How `flatMap` works with lists

- consider `List(x1, x2, x3).flatMap(x => f(x))`
- assume that

```
f: X => List[Y]
f(x1) = List(y0, y1)
f(x2) = List(y2)
f(x3) = List(y3, y4, y5, y6)
```

- then the result is `List(y0, y1, y2, y3, y4, y5, y6)`
- if we first do `.map(f)` then `flatten`:

```
List(x1, x2, x3).map(f).flatten =
  List(List(y0, y1), List(y2), List(y3, y4, y5, y6)).flatten =
  List(y0, y1, y2, y3, y4, y5, y6)
```

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 *that* 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 I: 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 functor block fail
- "Flat" functor block replaces a chain of nested `if/else` or `match/case`

Worked examples II: Pass/fail monads

Type constructors:

- `Option[A]` $\equiv 1 + A$
- `Either[Z, A]` $\equiv Z + A$
- `Try[A]` $\equiv$ `Either[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. Pass/fail chain: sequencing computations that may throw an exception

Intuitions for tree-like monads

Examples of tree-like recursive type constructors:

- $F^A \equiv A + F^A \times F^A$ (binary tree)
- $F^A \equiv A + S^{F^A}$ (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)

Implementing `flatMap` for these type constructors is recursive

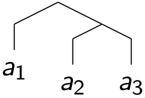
- See example code

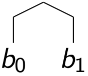
Example of a *non-tree-like* type constructor:

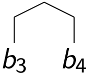
- $F^A \equiv A + A \times A + A \times A \times A \times A + ...$ (powers of 2, *non-recursive*)

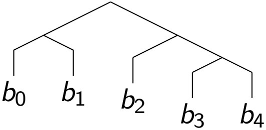
Worked examples III: Tree-like monads

How `flatMap` works for a binary tree: assume `f: A ⇒ Tree[B]` and

`tree1` $=$ 

; $f(a_1) =$ 

; $f(a_2) = b_2$; $f(a_3) =$ 

- then `tree1.flatMap(f)` $=$ 

- grafting subtrees plays the role of "flattening"

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

Worked examples IV: Single-value monads

- Pretend that 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:

- Managing 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

Deriving the types of single-value monads

Motivation for the choice of the type constructors Writer^A, Reader^A, State^A, Cont^A

We want previous values to be transformed via `flatMap` to next values

- `Writer`: a computation $(A \Rightarrow B)$ and some info (W) about it
  - $x^A \Rightarrow f(x) : B$ and $x^A \Rightarrow g(x) : W$; the type is $(A \Rightarrow B) \times (A \Rightarrow W)$
  - this function should have type $A \Rightarrow \text{Writer}^B$, hence $\text{Writer}^B \equiv B \times W$
    - ★ use the "arithmetic" Curry-Howard to transform types: $b^a w^a = (bw)^a$
- `Reader`: Read-only context, or "environment" of type E
  - $x^A \Rightarrow f(r, x) : B$ where r^E is fixed; the type is $A \times E \Rightarrow B$
  - this function should have type $A \Rightarrow \text{Reader}^B$, hence $\text{Reader}^B \equiv E \Rightarrow B$
    - ★ we used the "arithmetic" Curry-Howard to transform $b^{ae} = (b^e)^a$
- `Cont`: A computation that registers an asynchronous callback
  - $x^A \Rightarrow f(cb) : 1$ where $cb : B \Rightarrow 1$ (usually, callbacks return `Unit`)
  - the type is $A \Rightarrow (B \Rightarrow 1) \Rightarrow 1$; this function should have type $A \Rightarrow \text{Cont}^B$, hence $\text{Cont}^B \equiv (B \Rightarrow 1) \Rightarrow 1$
  - generalize to $\text{Cont}^A \equiv (A \Rightarrow R) \Rightarrow R$ where R is a fixed "result" type
- `State`: A computation can update state (S) while producing a result
  - $x^A \Rightarrow f(x, s)$ and $s^S := g(x, s)$; the type is $(A \times S \Rightarrow B) \times (A \times S \Rightarrow S)$
  - this will be $A \Rightarrow \text{State}^B$ if $\text{State}^B \equiv (S \Rightarrow B) \times (S \Rightarrow S) \equiv S \Rightarrow B \times S$
    - ★ we used the "arithmetic" Curry-Howard: $b^{as} s^{as} = (b^s s^s)^a = ((bs)^s)^a$

Exercises I

1. For a given `Set[Int]`, compute all subsets (w, x, y, z) of size 4 such that $w < x < y < z$ and $w + x = y + z$
2. Given 3 sequences xs, ys, zs of type `Seq[Int]`, compute all (x, y, z) such that $x \in xs$, $y \in ys$, $z \in zs$ and $x < y < z$ and $x + y + z < 10$
3. How many chess queens can avoid each other on an $3 \times 3 \times 3$ cube?
4. Write a tiny library for arithmetic using `Future`'s; use it to compute $1 + 2 + ... + 100$ via `for`/`yield` and verify the result. E.g. implement:

   ```
   const: Int ⇒ Future[Int]
   add(x: Int): Int ⇒ Future[Int]
   isEqual(x: Int): Int ⇒ Future[Boolean]
   ```

5. Read a file into a string and write it to another file using Java `Files` and `Paths` API. Use `Try` and `for`/`yield` to make this safe.
6. Given a semigroup W, make a semimonad out of $F^A \equiv E \Rightarrow A \times W$
7. Implement a semimonad instance for the (recursive) type constructor $F^A = A + A \times A + F^A + F^A \times F^A$
8. Find the largest prime number below 1000 via a simple Sieve of Eratosthenes; use the `State[S, Int]` monad with `S = Array[Boolean]`