Chapter 8: Applicative functors and profunctors

Part 1: Practical examples

Sergei Winitzki

Academy by the Bay

2018-06-25

Motivation for applicative functors

- Monads are inconvenient for expressing *independent* effects

Monads perform effects *sequentially* even if effects are independent:

```
x ← Future { c1 }
y ← Future { c2 }
z ← Future { c3 }
```

```
Future { c1 }.flatMap { x ⇒
  Future { c2 }.flatMap { y ⇒
    Future { c3 }.map { z ⇒ ...}
} }
```

- We would like to parallelize independent computations
- We would like to accumulate *all* errors, rather than stop at the first one

Changing the order of monad's effects will (generally) change the result:

```
for {
  x ← List(1, 2)
  y ← List(10, 20)
} yield f(x, y)
// f(1, 10), f(1, 20), f(2, 10), f(2, 20)
```

```
for {
  y ← List(10, 20)
  x ← List(1, 2)
} yield f(x, y)
// f(1, 10), f(2, 10), f(1, 20), f(2, 20)
```

- We would like to express a computation where effects are unordered
  - This can be done using a method `map2`, *not* defined via `flatMap`: the desired type signature is $\text{map2} : F^A \times F^B \Rightarrow (A \times B \Rightarrow C) \Rightarrow F^C$
  - **Applicative functor** has `map2` and `pure` but is not necessarily a monad

Defining `map2`, `map3`, etc.

Consider 1, 2, 3, ... commutative and independent "effects"

`for { x1 ← c1` `} yield f(x1)`	`c1.map(f)`

`for { x1 ← c1`
`  x2 ← c2`
`} yield f(x1, x2)`
	`(c1, c2).map2(f)`

`for { x ← c1`
`  x2 ← c2`
`  x3 ← c3`
`} yield f(x1, x2, x3)`
	`(c1, c2, c3).map3(f)`

- Generalize from `map`, `map2`, `map3` to `mapN`:

$$\text{map}_1 : F^A \Rightarrow (A \Rightarrow Z) \Rightarrow F^Z$$
$$\text{map}_2 : F^A \times F^B \Rightarrow (A \times B \Rightarrow Z) \Rightarrow F^Z$$
$$\text{map}_3 : F^A \times F^B \times F^C \Rightarrow (A \times B \times C \Rightarrow Z) \Rightarrow F^Z$$

Practical examples of using `mapN`

- $F^A \equiv Z + A$ where Z is a monoid: collect all errors
- $F^A = Z + A$: Create a validated case class out of validated parts
- $F^A \equiv$ `Future[A]`: perform several computations concurrently
- $F^A \equiv E \Rightarrow A$: pass standard arguments to functions more easily
- $F^A \equiv \text{List}^A$: transposing a matrix by using `map2`
- Applicative contrafunctors and applicative profunctors
  - defining an instance of `Semigroup` type class from `Semigroup` parts
  - implement `imap2` for non-disjunctive profunctors, e.g. $Z \times A \Rightarrow A \times A$
- "Fused `fold`": automatically merge several `fold`s into one
  - compute several running averages in one traversal (scala-folds)
- The difference between applicative and monadic functors
  - define monadic folds using the "free functor" construction
  - compute running averages that depend on previous running averages
    - do not confuse this with *monad-valued* folds (origami)!
  - applicative parsers vs. monadic parsers
    - applicative: parse independent data, collecting all errors
    - monadic: parse depends on previous results, stops on errors

Exercises I

Implement `map2`, or `imap2` if appropriate, for these type constructors F^A:

1. $F^A \equiv 1 + A + A \times A$
2. $F^A \equiv E \Rightarrow A \times A$
3. $F^A \equiv Z \times A \Rightarrow A$
4. $F^A \equiv A \Rightarrow A \times Z$ where Z is a `Monoid`
5. Write a function that defines an instance of the `Monoid` type class for a tuple (A, B) whose parts already have a `Monoid` instance.
6. Define a `Monoid` instance for the type F^S where F is an applicative functor that has `map2` and `pure`, while S is itself a monoid type.
7. Define a "regexp extractor" as a type constructor R^A describing extraction of various data from strings; the extracted data is converted to a value of type `Option[A]`. Implement `zip` and `map2` for R^A.
8. Use parser combinators to implement an evaluator for arithmetic language containing integers and $+$ symbols, for example $1 + 321 + 20$.
9. Use folding combinators to implement a `Fold` that computes the standard deviation of a sequence in one traversal.