FP in Scala

Chapter 5:
Strictness & Laziness
(recap)

Topics

- Non-strictness
- Strictness
- Type of functions
  - Strict functions
  - Non-strict functions
- Thunk
- Unevaluated thunk
- The `lazy` keyword
- (Forcing) evaluation of a thunk
- Streams
- Infinite streams
- Recursive
- Corecursive
  - Unfold

Non-strictness

- *aka* laziness
- short-circuiting Boolean functions `&&` and `||`
  - `&&` does not evaluate the arguments after the first one, if the first one is `false`
  - `||` does not evaluate the arguments after the first one, if the first one is `true`
- the `if ()` control construct in Scala is
  - non-strict with its *branch* evaluation

Strictness

- evaluates an expression immediately
- the `if()` control construct in Scala is
  - strict with its *conditional* evaluation

Type of functions

- Strict functions
- Non-strict functions

Strict functions

- evaluates expressions immediately
- in Scala, a strict function takes parameters
  - *by value*
  - and not *by name*
- function `f()` is strict if
  - `f(x)` evaluates to bottom of all `x` that evaluates to bottom
    - evaluates to bottom means
      - throws an exception instead of a definite value
      - or `f(x)` does not terminate

Non-strict functions

- do not always evaluate its expressions
  - accepts arguments but does not always evaluate them
  - unevaluated arguments have a `()=>` before them in Scala
  - the conditional part of the `if()` function is an example
    - not all the conditions of an `and` or an `or` operator are evaluated
- In Scala, a non-strict function takes parameters
  - *by name*
  - and not *by value*

Thunk

- an *unevaluated* form of an expression is called ***thunk***
  - we can force the thunk to evaluate the expression and get a result
- an unevaluated expression is wrapped in a ***thunk*** in Scala

Unevaluated thunk

- In Scala
  - unevaluated ***thunk(s)*** can be passed in as parameters to a function
  - by passing the arguments with a => notation immediately before their type

```
case class Cons[+A](head: () => A,
                    tail: () => Stream[A]) extends Stream[A]
```

The `lazy` keyword

- Scala by default does not cache the value of an evaluating expression
  - either as a parameter to a function
  - or in the body of the function

The `lazy` keyword

- helps memoizing arguments or other expressions
- helps separate an expression's description from its evaluation
- adding **`lazy`** to the **`val`** declaration in Scala
  - does not execute until it is first referenced
  - caches the result
  - prevents re-evaluation of the above operation
  - re-uses result from the cache

(Forcing) evaluation of a thunk

- In Scala
  - by passing an empty parameter list to the ***thunk***
  - that is, by adding the `()` to the name of the ***thunk***

(Forcing) evaluation of a thunk

for the below declaration of `Cons()`:

```
case class Cons[+A](h: () => A,
                    t: () => Stream[A]) extends Stream[A]
```

parameters of `Cons()` can be executed by doing the below:

```
Case Cons(h, f) => Some(h())
```

Recursive

- consumes data
- terminates by recursing smaller inputs
- tail recursion
  - consumes constant memory even if we keep a reference to it
- other recursions
  - consumes incremental memory

Streams

- is a “first-class loop”
- is a lazy list
- its logic can be combined with higher-order functions like `map` and `filter`
- initial and intermediate stream operations are not evaluated
- terminal operation results in immediate terminate
- makes operation resource efficient
- unused memory can be claimed quickly

Infinite streams

- An example of infinite stream is

```
val ones: Stream[Int] = Stream.cons(1, ones)
```

- The above stream is lazily evaluated and only returns elements that requested by the function(s) working on it

```
scala> ones.take(5).toList

res0: List[Int] = List(1, 1, 1, 1, 1)
```

Corecursive

- produces data
- does not terminate (guarded recursion)
  - so long as they remain productive
- productivity is also called cotermination

Corecursive: unfold

- **`unfold()`** is a corecursive function in Scala

```
def unfold[A, S](z: S)(f: S => Option[(A, S)]): Stream[A]{

    …

}
```

- **`unfold()`** function is productive as long as **`f`** terminates
- does not consume constant memory

Summary

- non-strictness: fundamental way to write efficient and modular code
- non-strict code allows separation of concerns
  - separating the *description* from the *how-what-when* of its evaluation
  - allow re-use of the *description* in multiple contexts
  - evaluating different portions of the expression to obtain different results
- ***thunks*** help in improving *execution* efficiency
- memoizing: the `lazy` keyword helps cache results of an executed expression
  - prevents re-evaluation of expression if executed more than once