INFO0054 Programmation Fonctionnelle – Exercises

Christophe Debruyne

Exercises 7: Non-Strict Evaluation II

```
import Flux.*

enum Flux[+A]:
    case Empty
    case Cons(h: () => A, t: () => Flux[A])

    def headOption: Option[A] = this match
        case Empty => None
        case Cons(h, t) => Some(h())

    def toList: List[A] = this match
        case Cons(h,t) => h() :: t().toList
        case Empty => Nil

    def take(n: Int): Flux[A] = this match
        case Cons(h, t) if n > 1 => cons(h(), t().take(n - 1))
        case Cons(h, _) if n == 1 => cons(h(), empty)
        case _ => empty

    def filter(f: A => Boolean): Flux[A] = this match
        case Cons(h, t) if f(h()) => cons(h(), t().filter(f))
        case Cons(_, t) => t().filter(f)
        case _ => empty

    def map[B](f: A => B): Flux[B] = this match
        case Cons(h, t) => cons(f(h()), t().map(f))
        case _ => empty

    def takeWhile(p: A => Boolean): Flux[A] = ???
    def exists(p: A => Boolean): Boolean = ???
    def foldRight[B](acc: => B)(f: (A, => B) => B): B = ???

object Flux:
    def cons[A](hd: => A, tl: => Flux[A]): Flux[A] =
        lazy val head = hd
        lazy val tail = tl
        Cons(() => head, () => tail)

    def empty[A]: Flux[A] = Empty

    def apply[A](as: A*): Flux[A] =
        if (as.isEmpty) empty
        else cons(as.head, apply(as.tail*))
```

Exercise 1:

Exercise 5.4 in the book: Implement `forAll`, which checks that all elements in a `Flux` match a predicate. Your implementation should terminate the traversal as soon as it encounters a non matching value.

Solution 1:

```
    def forAll(p: A => Boolean): Boolean =
        foldRight(true)((a, b) => p(a) && b)
```

Exercise 2:

Exercise 5.5 in the book: Use `foldRight` to implement `takeWhile`.

Solution 2:

```
    def takeWhileBis(p: A => Boolean): Flux[A] =
        foldRight(empty)((a, b) => if p(a) then cons(a, b) else empty)
```

Exercise 3:

Exercise 5.6 in the book: Implement `headOption` using `foldRight`.

Solution 3:

```
    def headOptionBis: Option[A] =
        foldRight(None: Option[A])((h, _) => Some(h))
```

Exercise 4:

Exercise 5.7 in the book: Implement `map`, `filter`, `append`, and `flatMap` using `foldRight`. The `append` should be non-strict in its argument.

Solution 4:

```
    def mapBis[B](f: A => B): Flux[B] =
        foldRight(empty[B])((a, acc) => cons(f(a), acc))

    def filterBis(f: A => Boolean): Flux[A] =
        foldRight(empty[A])((a, acc) => if f(a) then cons(a, acc) else acc)

    def flatMapBis[B](f: A => Flux[B]): Flux[B] =
        foldRight(empty[B])((a, acc) => f(a).append(acc))

    def append[A2>:A](that: => Flux[A2]): Flux[A2] =
        foldRight(that)((a, acc) => cons(a, acc))
```

References

[1] Paul Chiusano and Rnar Bjarnason. 2015. Functional Programming in Scala (2nd. ed.). Manning Publications Co., USA.