Functional thinking: Laziness, Part 2

Delving deeper into lazy evaluation

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

19 December 2012
(First published 18 December 2012)

Implementing lazy evaluation is easy in a language that supports closures. This *Functional thinking* installment shows how to derive a lazy list using Groovy's closures as building blocks. Then it explores some of the performance and conceptual benefits of lazy evaluation, including the ability to initialize fields lazily in some languages.
19 Dec 2012 - *Author clarified information in first two paragraphs following Listing 4.*

View more content in this series

About this series

This series aims to reorient your perspective toward a functional mindset, helping you look at common problems in new ways and find ways to improve your day-to-day coding. It explores functional programming concepts, frameworks that allow functional programming within the Java language, functional programming languages that run on the JVM, and some future-leaning directions of language design. The series is geared toward developers who know Java and how its abstractions work but have little or no experience using a functional language.

In "Laziness, Part 1," I explored lazy libraries in Java™. In this installment, I demonstrate how to build a simple lazy list using closures as building blocks, then explore some of the performance benefits of lazy evaluation along with some lazy aspects of Groovy, Scala, and Clojure.

Building a lazy list

In an early installment of this series, I showed a simple implementation of a lazy list in Groovy. However, I didn't show the derivation of how it works, which is the subject here.

As you know from the last installment, languages can be categorized as *strict* (eagerly evaluating all expressions) or *lazy* (deferring evaluation) until absolutely needed. Groovy is a strict language by nature, but I can transform a nonlazy list into a lazy one by recursively wrapping a strict list within a closure. This lets me defer evaluation of subsequent values by delaying execution of the closure block.

A strict empty list in Groovy is represented by an array, using empty square braces: `[]`. If I wrap it in a closure, it becomes a lazy empty list:

```
{-> [] }
```

If I need to add an `a` element to the list, I can add it to the front, then make the entire new list lazy again:

```
{-> [ a, {-> [] } ] }
```

The method for adding to the front of the list is traditionally called either *prepend* or *cons*. To add more elements, I repeat this operation for each new item; adding three elements (`a`, `b`, and `c`) to the list yields:

```
{-> [a, {-> [b, {-> [ c, {-> [] } ] } ] } ] }
```

This syntax is clumsy, but once I understand the principle, I can create a class in Groovy that implements a traditional set of methods for a lazy collection, as shown in Listing 1:

Listing 1. Building a lazy list in Groovy using closures

```
class PLazyList {
  private Closure list

  private PLazyList(list) {
    this.list = list
  }

  static PLazyList nil() {
    new PLazyList({-> []})
  }

  PLazyList cons(head) {
    new PLazyList({-> [head, list]})
  }

  def head() {
    def lst = list.call()
    lst ? lst[0] : null
  }

  def tail() {
    def lst = list.call()
    lst ? new PLazyList(lst.tail()[0]) : nil()
  }

  boolean isEmpty() {
    list.call() == []
  }

  def fold(n, acc, f) {
    n == 0 || isEmpty() ? acc : tail().fold(n - 1, f.call(acc, head()), f)
  }

  def foldAll(acc, f) {
    isEmpty() ? acc : tail().foldAll(f.call(acc, head()), f)
  }

  def take(n) {
    fold(n, []) {acc, item -> acc << item}
```

```
  }

  def takeAll() {
    foldAll([]) {acc, item -> acc << item}
  }

  def toList() {
    takeAll()
  }
}
```

In Listing 1, the constructor is private; it's called by starting with an empty list using `nil()`, which constructs an empty list. The `cons()` method enables me to add new elements by prepending the passed parameter, then wrapping the result in a closure block.

The next three methods enable list traversal. The `head()` method returns the first element of the list, and `tail()` returns the sublist of all elements except the first. In both cases, I `call()` the closure block — known as *forcing* the evaluation in lazy terms. Because I'm retrieving values, it ceases to be lazy as I harvest the values. Not surprisingly, the `isEmpty()` method checks to see if any terms are left to resolve.

The remaining methods are higher-order functions for performing operations on the list. The `fold()` and `foldAll()` methods perform the *fold* abstraction, also known as *reduce* — or, in Groovy only, `injectAll()`. I've shown the use of this family of methods in many previous installments (such as "Thinking Functionally, Part 3"), but this is the first time I've shown a recursive definition written purely in terms of closure blocks. The `foldAll()` method checks to see if the list is empty and, if it is, returns `acc` (the accumulator, the seed value for the fold operation). Otherwise, it recursively calls `foldAll()` on the `tail()` of the list, passing the accumulator and the head of the list. The function (the `f` parameter) should accept two parameters and yield a single result; this is the "fold" operation as you fold one element atop the adjacent one.

Building a list and manipulating it appears in Listing 2:

Listing 2. Exercising lazy lists

```
def lazylist = PLazyList.nil().cons(4).cons(3).cons(2).cons(1)
println(lazylist.takeAll()) //[1, 2, 3, 4]
println(lazylist.foldAll(0, {i, j -> i + j})) // 10
lazylist = PLazyList.nil().cons(1).cons(2).cons(4).cons(8)
println(lazylist.take(2))  //[8, 4]
```

In Listing 2, I create a list by `cons()`ing values onto an empty list. Notice that when I `takeAll()` of the elements, they come back in the reverse order of their addition to the list. Remember, `cons()` is really shorthand for *prepend*; it adds elements to the front of the list. The `foldAll()` method enables me to sum the list by providing a transformation code block, `{i, j -> i + j}`, which uses addition as the fold operation. Last, I use the `take()` method to force evaluation of only the first two elements.

Real-world lazy-list implementations differ from this one, avoiding recursion and adding more-flexible manipulation methods. However, knowing conceptually what's happening inside the implementation aids understanding and use.

Benefits of laziness

Lazy lists have several benefits. First, you can use them to create infinite sequences. Because the values aren't evaluated until needed, you can model infinite lists using lazy collections. I show an example of this implemented in Groovy in the "Functional Features in Groovy, Part 1" installment. A second benefit is reduced storage size. If — rather than hold an entire collection — I can derive subsequent values, then I can trade storage for execution speed. Choosing to use a lazy collection becomes a trade-off between the expense of storing the values versus calculating new ones.

Third, one of the key benefits of lazy collections is that the runtime can generate more-efficient code. Consider the code in Listing 3:

Listing 3. Finding palindromes in Groovy

```
def isPalindrome(s) {
  def sl = s.toLowerCase()
  sl == sl.reverse()
}

def findFirstPalindrome(s) {
  s.tokenize(' ').find {isPalindrome(it)}
}

s1 = "The quick brown fox jumped over anna the dog";
println(findFirstPalindrome(s1)) // anna

s2 = "Bob went to Harrah and gambled with Otto and Steve"
println(findFirstPalindrome(s2)) // Bob
```

The `isPalindrome()` method in Listing 3 normalizes the case of the subject word, then determines if the word has the same characters in reverse. The `findFirstPalindrome()` method tries to find the first palindrome in the passed string by using Groovy's `find()` method, which accepts a code block as the filtering mechanism.

Suppose I have a huge sequence of characters within which I need to find the first palindrome. During the execution of the `findFirstPalindrome()` method, the code in Listing 3 first eagerly tokenizes the entire sequence, creating an intermediate data structure, then issues the `find()` command. Groovy's `tokenize()` method isn't lazy, so in this case it might build a huge temporary data structure, only to discard most of it.

Consider the same code written in Clojure, appearing in Listing 4:

Listing 4. Clojure's palindromes

```
(defn palindrome? [s]
  (let [sl (.toLowerCase s)]
  (= sl (apply str (reverse sl)))))

(defn find-palindromes [s]
    (filter palindrome? (clojure.string/split s #" ")))

(println (find-palindromes "The brown fox jumped over anna."))
; (anna)
(println (find-palindromes "Bob went to Harrah and gambled with Otto"))
; (Bob Harrah Otto)
(println (take 1 (find-palindromes "Bob went to Harrah and gambled with Otto")))
; (Bob)
```

The implementation details in Listing 3 and Listing 4 are the same but use different language constructs. In the Clojure `palindrome?` function, I make the parameter string lowercase, then check equality with the reversed string. The extra call to `apply` converts the character sequence returned by `reverse` back to a string for comparison. The `find-palindromes` function uses Clojure's `filter` function, which accepts a function to act as the filter and the collection to be filtered. For the call to the `palindrome?` function, Clojure provides several alternatives. I can create an anonymous function to call it such as `#(palindrome? %)`, which is syntactic sugar for an anonymous function that accepts a single parameter; the long-hand version would look like:

```
    (fn [x]
      (palindrome? x))
```

When I have a single parameter, Clojure allows me to avoid declaring the anonymous function and naming the parameter, which I substitute with `%` in the `#(palindrome? %)` function call. In Listing 4, I can use the even shorter form of the function name directly; `filter` is expecting a method that accepts a single parameter and returns a boolean, which matches `palindrome?`.

The translation from Groovy to Clojure entailed more than just syntax. All of Clojure's data structures that *can* be lazy *are* lazy, including operations on collections like `filter` and `split`. Thus, in the Clojure version, everything is automatically lazy, which manifests in the second example in Listing 4, when I call `find-palindromes` on the collection with multiples. The return from `filter` is a lazy collection that is forced as I print it. If I want only the first entry, I must `take` the number of lazy entrants I need from the list.

Scala approaches laziness in a slightly different way. Rather than make everything lazy by default, it offers lazy *views* on collections. Consider the Scala implementation of the palindrome problem in Listing 5:

Listing 5. Scala palindromes

```
def isPalindrome(x: String) = x == x.reverse
def findPalidrome(s: Seq[String]) = s find isPalindrome

findPalindrome(words take 1000000)
```

In Listing 5, pulling 1 million words from the collection via the `take` method will be quite inefficient, especially if the goal is to find the first palindrome. To convert the `words` collection to a lazy one, use the `view` method:

```
findPalindrome(words.view take 1000000)
```

The `view` method allows lazy traversal of the collection, making for more-efficient code.

Lazy field initialization

Before leaving the subject of laziness, I'll mention that two languages have a nice facility to make expensive initializations lazy. By prepending `lazy` onto the `val` declaration, you can convert fields in Scala from eager to as-needed evaluation:

```
lazy val x = timeConsumingAndOrSizableComputation()
```

This is basically syntactic sugar for the code in Listing 6:

Listing 6. Scala's generated syntactic sugar for lazy fields

```
var _x = None
def x = if (_x.isDefined) _x.get else {
  _x = Some(timeConsumingAndOrSizableComputation())
  _x.get
}
```

Groovy has a similar facility using an advanced language feature known as *Abstract Syntax Tree (AST) Transformations*. They enable you to interact with the compiler's generation of the underlying abstract syntax tree, allowing user transformations at a low level. One of the predefined transformations is the `@Lazy` attribute, shown in action in Listing 7:

Listing 7. Lazy fields in Groovy

```
class Person {
    @Lazy pets = ['Cat', 'Dog', 'Bird']
}

def p = new Person()
assert !(p.dump().contains('Cat'))

assert p.pets.size() == 3
assert p.dump().contains('Cat')
```

In Listing 7, the `Person` instance `p` doesn't appear to have a `Cat` value until the data structure is accessed the first time. Groovy also allows you to use a closure block to initialize the data structure:

```
class Person {
    @Lazy List pets = { /* complex computation here */ }()
}
```

Finally, you can also tell Groovy to use *soft references* — Java's version of a pointer reference that can be reclaimed if needed — to hold your lazily initialized field:

```
class Person {
    @Lazy(soft = true) List pets = ['Cat', 'Dog', 'Bird']
}
```

Conclusion

In this installment, I delved even deeper into laziness, building a lazy collection from scratch using closures in Groovy. I also discussed why you might want to consider a lazy structure, listing some of the benefits. In particular, the ability for your runtime to optimize resources is a huge win. Finally, I showed some esoteric but useful manifestations of laziness in Scala and Groovy relating to lazily initialized fields.

Resources

Learn

- Lazy lists in Groovy: Thanks to Andrey Paramonov's blog for his perspective on constructing lazy lists using closures.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- Totally Lazy: The Totally Lazy framework adds tons of functional extensions to Java, using an intuitive DSL-like interface.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- *The busy developer's guide to Scala*: Learn more about Scala in this developerWorks series.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Neal Ford

Neal Ford is a software architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He also designs and develops applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.