Functional thinking: Transformations and optimizations

More functional comparisons across languages

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

16 October 2012

Various functional languages and frameworks feature many of the same abstractions and behaviors but name them differently. In this *Functional thinking* article, series author Neal Ford optimizes the solution from the preceding installment by improving the algorithm and adding caching, illustrating how each language or framework accommodates the required changes.

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.

Functional programming has its roots in both mathematics and computer science, both of which have strong opinions about terminology. Language and frameworks designers develop their favorite nomenclature, only to discover that the underlying paradigms already have names. Learning functional programming paradigms is difficult because of the inconsistency of terminology.

In "Tons of transformations," I took the problem of classifying prime numbers and implemented a solution across several functional languages on the JVM and two functional Java frameworks. Continuing that theme, this installment optimizes the previous algorithm in a couple of ways, showing the subsequent changes across languages. This installment, like the last, illustrates the differences in terminology and availability of features across tools and languages. In particular, I exercise `map`, `filter`, and `memoize` for these examples.

Prime-number classification optimized, in pure Java

The stated problem is determining whether a number is a *prime number*, one whose only factors are 1 and itself. Among the several algorithms that solve this problem, I've chosen to illustrate *filtering* and *mapping* in the functional programming world.

In the last installment, I took a naive approach to the algorithm to determine a number's factors, opting for simple code rather than optimally performing code. In this installment, I optimize that algorithm using several different functional concepts. In addition, I optimize each version for the use case in which the class is invoked multiple times to classify the same number.

My original Java code for determining prime numbers appears in Listing 1:

Listing 1. Original Java version of the prime-number classifier

```
public class PrimeNumberClassifier {
    private Integer number;

    public PrimeNumberClassifier(int number) {
        this.number = number;
    }

    public boolean isFactor(int potential) {
        return number % potential == 0;
    }

    public Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<Integer>();
        factors.add(1);
        factors.add(number);
        for (Integer i = 2; i < number; i++)
            if (isFactor(i))
                factors.add(i);
        return factors;
    }

    public int sumFactors() {
        int sum = 0;
        for (int i : getFactors())
            sum += i;
        return sum;
    }

    public boolean isPrime() {
        return sumFactors() == number + 1;
    }
}
```

In Listing 1, the `getFactors()` method iterates the potential factors from 2 to the number being classified, which is quite inefficient. Consider the fact that factors always occur in pairs; that suggests that when I find one factor, I can ascertain its mate by simple division. Thus, I don't need to iterate all the way to the number; I can instead iterate up to the number's square root, harvesting the factors in pairs. The improved `getFactors()` method appears within the overall-improved version in Listing 2:

Listing 2. Optimized pure Java version

```
public class PrimeNumber {
    private Integer number;
```

```
    private Map<Integer, Integer> cache;

    public PrimeNumber() {
        cache = new HashMap<Integer, Integer>();
    }

    public PrimeNumber setCandidate(Integer number) {
        this.number = number;
        return this;
    }

    public static PrimeNumber getPrime(int number) {
        return new PrimeNumber().setCandidate(number);
    }

    public boolean isFactor(int potential) {
        return number % potential == 0;
    }

    public Set<Integer> getFactors() {
        Set<Integer> factors = new HashSet<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < sqrt(number) + 1; i++)
            if (isFactor(i)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    public int sumFactors() {
        int sum = 0;
        if (cache.containsValue(number))
            sum = cache.get(number);
        else
            for (int i : getFactors())
                sum += i;
        return sum;
    }

    public boolean isPrime() {
        return number == 2 || sumFactors() == number + 1;
    }

}
```

In the `getFactors()` method in Listing 2, I iterate from 2 to the square root of the number (plus 1, to handle rounding errors) and harvest the factors in pairs. Returning a `Set` is important in this code because of an edge case concerning perfect-square numbers. Consider the number 16, whose square root is 4. In the `getFactors()` method, using a `List` instead of a `Set` will generate duplicate 4s in the list. Unit tests exist to find edge cases like this!

The other optimization in Listing 2 concerns multiple invocations. If the typical usage of this code is to evaluate the same number many times for primeness, the calculation that the `sumFactors()` method in Listing 1 performs is inefficient. Instead, in the `sumFactors()` method in Listing 2, I create a class-wide cache to hold previously calculated values.

Achieving the second optimization requires a little dubious class design, forcing it to be stateful so that the instance can act as the owner of the cache. This could be improved, but the improvement is trivial in subsequent examples, so I won't bother here.

Optimized Functional Java

Functional Java (see Resources) is a framework to add functional features to Java. The two methods that optimization affects are the `getFactors()` and `sumFactors()` methods, whose original (unoptimized) versions appear in Listing 3:

Listing 3. Original Functional Java `getFactors()` and `sumFactors()` methods

```
public List<Integer> getFactors() {
    return range(1, number + 1)
            .filter(new F<Integer, Boolean>() {
                public Boolean f(final Integer i) {
                    return isFactor(i);
                }
            });
}

public int sumFactors() {
    return getFactors().foldLeft(fj.function.Integers.add, 0);
}
```

The `getFactors()` method in Listing 3 filters the range of numbers from 1 to the target number plus 1 (because `range`s in Functional Java are noninclusive) using the `isFilter()` method to determine inclusion. The optimized version of the Functional Java prime-number classifier appears in Listing 4:

Listing 4. Optimized Functional Java version

```
import fj.F;
import fj.data.List;
import java.util.HashMap;
import java.util.Map;
import static fj.data.List.range;
import static fj.function.Integers.add;
import static java.lang.Math.round;

import static java.lang.Math.sqrt;

public class FjPrimeNumber {
    private int candidate;
    private Map<Integer, Integer> cache;

    public FjPrimeNumber setCandidate(int value) {
        this.candidate = value;
        return this;
    }

    public FjPrimeNumber(int candidate) {
        this.candidate = candidate;
        cache = new HashMap<Integer, Integer>();
    }

    public boolean isFactor(int potential) {
```

```
        return candidate % potential == 0;
    }

    public List<Integer> getFactors() {
        final List<Integer> lowerFactors = range(1, (int) round(sqrt(candidate) + 1))
                .filter(new F<Integer, Boolean>() {
                    public Boolean f(final Integer i) {
                        return isFactor(i);
                    }
                });
        return lowerFactors.append(lowerFactors.map(new F<Integer, Integer>() {
            public Integer f(final Integer i) {
                return candidate / i;
            }
        }))
        .nub();
    }

    public int sumFactors() {

        if (cache.containsKey(candidate))
            return cache.get(candidate);
        else {
            int sum = getFactors().foldLeft(add, 0);
            cache.put(candidate, sum);
            return sum;
        }
    }

    public boolean isPrime() {
        return candidate == 2 || sumFactors() == candidate + 1;
    }
}
```

In the `getFactors()` method in Listing 4, I use the same `range()` and `filter()` methods more selectively. The first range harvests the factors up to the square root, using the `filter()` method as in Listing 3. The second line of the method uses the `map()` method from Functional Java to generate the factors above the square root. The `map()` method applies the function to each element in the collection, returning a transformed collection. This list of factors above the square root is appended to the factors below the square root (`lowerFactors`). The last method call to Functional Java's `nub()` method converts the list to a set, alleviating the perfect-square duplication problem.

The `sumFactors()` optimization in Listing 4 uses a cache identically to the pure Java version in Listing 2, implying the same statefulness requirement on the class as in that version.

Optimized Groovy

The original Groovy versions of the `getFactors()` and `sumFactors()` methods appear in Listing 5:

Listing 5. Original Groovy `getFactors()` and `sumFactors()` methods

```
public def getFactors() {
  (1..number).findAll { isFactor(it) }.toSet()
}

public def sumFactors() {
  getFactors().inject(0, {i, j -> i + j})
}
```

In Groovy, the `findAll()` method filters the range of numbers, and the `sumFactors()` method uses Groovy's `inject()` method, applying the code block to each element to reduce the list to one element (which will be the sum, because the code block sums each pair as the reduction operation). The optimized Groovy version of the prime number classifier appears in Listing 6:

Listing 6. Optimized Groovy version

```
import static java.lang.Math.sqrt

class PrimeNumber {
  static def isFactor(potential, number) {
    number % potential == 0;
  }

  static def factors = { number ->
    def factors = (1..sqrt(number)).findAll { isFactor(it, number) }
    factors.addAll factors.collect { (int) number / it}
    factors.toSet()
  }

  static def getFactors = factors.memoize();

  static def sumFactors(number) {
    getFactors(number).inject(0, {i, j -> i + j})
  }

  static def isPrime(number) {
    number == 2 || sumFactors(number) == number + 1
  }
}
```

Just as in the Functional Java version, the `factors()` method in Listing 6 partitions the factors using the square root and converts the resulting list into a set via the `toSet()` method. The primary difference is the terminology difference between Functional Java and Groovy; in Functional Java, the `filter()` and `foldLeft()` methods are synonyms for Groovy's `findAll()` and `inject()`, respectively.

The optimization solution in Listing 6 is radically different from the previous Java versions. Rather than add statefulness to the class, I use Groovy's `memoize()` method. The `factors` method in Listing 6 is a *pure function*, meaning that it relies on no state except its parameter. Once that requirement is met, the Groovy runtime can cache the values automatically via the `memoize()` method, which returns a cached version of the `factors()` method named `getFactors()`. This is a great example of the ability of functional programming to reduce the number of mechanisms the developer must maintain, such as caching. I cover memoization more fully in the "Functional design patterns, Part 1" installment of this series.

Optimized Scala

The original Scala versions of the `getFactors()` and `sumFactors()` methods appear in Listing 7:

Listing 7. Original Scala `factors()` and `sum()` methods

```
def factors(number: Int) =
  (1 to number) filter (isFactor(number, _))

def sum(factors: Seq[Int]) =
 factors.foldLeft(0)(_ + _)
```

The code in Listing 7 makes use of the convenient `_` placeholder for parameters whose names aren't important. The optimized version of the prime-number classifier appears in Listing 8:

Listing 8. Optimized Scala version

```
import scala.math.sqrt;

object PrimeNumber {
  def isFactor(number: Int, potentialFactor: Int) =
    number % potentialFactor == 0

  def factors(number: Int) = {
    val lowerFactors = (1 to sqrt(number).toInt) filter (isFactor(number, _))
    val upperFactors = lowerFactors.map(number / _)
    lowerFactors.union(upperFactors)
  }

  def memoize[A, B](f: A => B) = new (A => B) {
    val cache = scala.collection.mutable.Map[A, B]()
    def apply(x: A): B = cache.getOrElseUpdate(x, f(x))
  }

  def getFactors = memoize(factors)

  def sum(factors: Seq[Int]) =
    factors.foldLeft(0)(_ + _)

  def isPrime(number: Int) =
    number == 2 || sum(getFactors(number)) == number + 1
}
```

The optimized `factors()` method uses the same technique as previous examples (such as in Listing 3), adapted to Scala's syntax, yielding a straightforward implementation.

Scala does not have a standard memoization facility, although it is suggested as a future addition. It can be implemented numerous ways; an easy implementation relies on Scala's built-in mutable map and its handy `getOrElseUpdate()` method.

Optimized Clojure

The Clojure versions of the `factors` and `sum-factors` methods appear in Listing 9:

Listing 9. Original `factors` and `sum-factors` methods

```
(defn factors [n]
  (filter #(factor? n %) (range 1 (+ n 1))))

(defn sum-factors [n]
  (reduce + (factors n)))
```

As in the other preoptimized versions, the original Clojure code filters the range of numbers from 1 to the number plus 1, and it uses Clojure's `reduce` function to apply the `+` function to each element, yielding the sum. The optimized Clojure prime-number classifier appears in Listing 10:

Listing 10. Optimized Clojure version

```
(ns primes)

(defn factor? [n, potential]
  (zero? (rem n potential)))

(defn factors [n]
  (let [factors-below-sqrt (filter #(factor? n %) (range 1 (inc (Math/sqrt n))))
        factors-above-sqrt (map #(/ n %) factors-below-sqrt)]
    (concat factors-below-sqrt factors-above-sqrt)))

(def get-factors (memoize factors))

(defn sum-factors [n]
  (reduce + (get-factors n)))

(defn prime? [n]
  (or (= n 2) (= (inc n) (sum-factors n))))
```

The `factors` method uses the same optimized algorithm as used in previous examples (such as Listing 3), harvesting the factors below the square root by filtering the range from 1 to the square root plus 1: `(filter #(factor? n %) (range 1 (inc (Math/sqrt n))))`. The Clojure version uses its own symbol (`%`) for unnamed parameters, like the Scala version in Listing 8. The `#(/ n %)` syntax creates an anonymous function, as syntactic-sugar shorthand for `(fn [x] (/ n x))`.

Clojure includes the ability to memoize pure functions via the `memoize` function, just as in the Groovy version, making the second optimization trivial to implement.

Conclusion

In this installment as in the last, I illustrated how similar concepts have grown different names across languages and frameworks, as well as built-in abilities. Groovy is the odd language out when it comes to function naming (`findAll()` rather than the more common `filter()`, and `collect()` rather than `map()`, for example). The presence of memoization makes a huge difference for the ease and safety of implementing caching.

In the next installment, I'll explore laziness more fully across languages and functional frameworks.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's book discusses tools and practices that help you improve your coding efficiency.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- *Practically Groovy*: Explore Groovy's functional (and other) features in this developerWorks article series.
- "Execution in the Kingdom of Nouns" (Steve Yegge, March 2006): An entertaining rant about some aspects of Java language design.
- 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

- Check out developerWorks blogs and get involved in the developerWorks community.

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.