Matt Malone’s Old-Fashioned Software Development Blog

July 10, 2009

Scala Code Review: foldLeft and foldRight

Posted by Matt under Scala | Tags: folding, foldLeft, foldRight, list, recursion, Scala |
[25] Comments

One of my favorite functional programming tricks is folding. The fold left and fold right functions can do a lot of complicated things with a small amount of code. Today, I’d like to (1) introduce folding, (2) make note of some surprising, nay, ***shocking*** fold behavior, (3) review the folding code used in Scala’s List class, and (4) make some presumptuous suggestions on how to improve List.

Update: *I’ve created a new post in which I list lots and lots of foldLeft examples in case you’d like to learn more about what folding can accomplish.*

Know When to Hold ‘Em, Know When to Fold ‘Em

In case you’re not familiar with folding, I’ll describe it as briefly as I can.

Here’s the signature of the foldLeft function from List[A], a list of items of type A:

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

Firstly, foldLeft is a curried function (So is foldRight). If you don’t know about currying, that’s ok; this function just takes its two parameters (z and f) in two sets of parentheses instead of one. Currying isn’t the important part anyway.

The first parameter, z, is of type B, which is to say it can be different from the list contents type. The second parameter, f, is a function that takes a B and an A (a list item) as parameters, and it returns a value of type B. So the purpose of function f is to take a value of type B, use a list item to modify that value and return it.

The foldLeft function goes through the whole List, from head to tail, and passes each value to f. For the first list item, that first parameter, z, is used as the first parameter to f. For the second list item, the result of the first call to f is used as the B type parameter.

For example, say we had a list of Ints 1, 2, and 3. We could call foldLeft(“X”)((b,a) => b + a). For the first item, 1, the function we define would add string “X” to Int 1, returning string “X1”. For the second list item, 2, the function would add string “X1” to Int 2, returning “X12”. And for the final list item, 3, the function would add “X12” to 3 and return “X123”.

Here are a few more examples.

```
1 list.foldLeft(0)((b,a) => b+a)
2 list.foldLeft(1)((b,a) => b*a)
3 list.foldLeft(List[Int]())((b,a) => a :: b)
```

The first line is super simple. It’s almost like the example I described above, but the z value is the number 0 instead of string “X”. This fold combines the elements of the list by addition instead of concatenation. So the fold returns the sum of all Ints in the list. Line 2 combines them through multiplication. Do you see why the z value is 1 in this case?

Line 3 is a little more complex. Can you guess what it does? It starts out with an empty list of Ints and adds each item to the accumulator (We call the b parameter of our function the accumulator because it accumulates data from each of our list items). Because it starts with the head and adds to the beginning of the accumulator list until it gets to the last item of the original list, it returns the original list in reverse order.

The foldRight function works in much the same way as foldLeft. Can you guess the difference? You got it. It starts at the end of the list and works its way up to the head.

Folds can be used for MUCH more than I’ve shown here. With folds, you can solve lots of different problems with a standard construct. You should read up on them if you’re just starting out in functional programming.

All That Glitters Is Not Fold

Now for the moment you’ve been waiting for. Fold’s dirty little secret! The below is taken from a scala interpreter session.

```
scala> var shortList = 1 to 10 toList
shortList: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> var longList = 1 to 325000 toList
longList: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, ...

scala> shortList.foldLeft("")((x,y) => "X")
res1: java.lang.String = X

scala> shortList.foldRight("")((x,y) => "X")
res2: java.lang.String = X

scala> longList.foldLeft("")((x,y) => "X")
res3: java.lang.String = X

scala> longList.foldRight("")((x,y) => "X")
java.lang.StackOverflowError
        at scala.List.foldRight(List.scala:1079)
        at scala.List.foldRight(List.scala:1081)
        at scala.List.foldRight(List.scala:1081)
        at scala.List.foldRight(List.scala:1081)
        at scala.List.foldRight(List.scala:1081)
        at scala.List.foldRight(List.scala:1081)
        at scala.List.foldRight(List.scala:1081)
        at scala.List.foldRight(List.scala:1081)
        at scala.List.foldRig...
```

We created two lists: shortList with 10 items, and longList with 325,000 items. Then we perform a trivial foldLeft and foldRight on shortList. It's trivial because the passed-in function always returns the string "X"; it doesn't even use the list data.

Then we do a foldLeft on longList. This goes off without a hitch. Finally we try to do a foldRight, the same foldRight that succeeded on the shorter list, and it *fails*! The foldLeft worked. Why didn't the foldRight work? It's a perfectly reasonable call against a perfectly reasonable List. Something funny is going on here.

The error message says there was a stack overflow, and the stack trace shows a long list of calls at List.scala:1081. If you've read my post about tail-recursion, then you probably suspect that some recursive code is to blame.

Let's look into List.scala, maybe the single most important Scala source file.

Fool's Fold

Without further ado, here's the code for foldLeft and foldRight from List.scala:

```
override def foldLeft[B](z: B)(f: (B, A) => B): B = {
  var acc = z
  var these = this
  while (!these.isEmpty) {
    acc = f(acc, these.head)
    these = these.tail
  }
  acc
}

override def foldRight[B](z: B)(f: (A, B) => B): B = this match {
  case Nil => z
  case x :: xs => f(x, xs.foldRight(z)(f))
}
```

Wow! Those two definitions are very different!

The foldLeft function is the one that worked for short and long lists. You can see why? It isn't head-recursive. In fact, it isn't recursive at all. It is implemented as a while loop. On each iteration, the next list item is passed to the function f and the accumulator (called acc) is updated. When there are no more list items, the accumulator is returned. No recursion means no stack overflows.

The foldRight function is implemented in a totally different way. If the list is empty, the z parameter is returned. Otherwise, a recursive call is made on the tail (the whole list minus the first item) of this list, and that result is passed to the function f. Study the foldRight definition. Do you understand how it works? It's an elegant recursive solution, and the code really is quite pretty, but it's not tail recursive so it fails for large lists.

Why didn't Mr Odersky just write foldRight using a while loop, too? Then this problem wouldn't exist, right? The reason is that Scala's List is a implemented as a singly-linked list. Each list element has access to the next item in the list, but not to the previous item. You can only traverse a list in one direction! This works fine for foldLeft, which goes from head to tail, but foldRight has to start at the end of the list and work its way forward to the head. If foldRight uses recursion, it must recurse all the way to the end and then use the results of those recursive calls as the accumulator passed into function f.

See? The results of the recursive call must be used for further calculation, so the recursive call can't be the last thing that happens, so it can't be written as a tail-recursive function. If you don't know what I'm talking about, read my introduction to tail-recursion.

Out With The Fold, In With The New

So is that it for foldRight? Is it hopeless? I say no!

There is a way to get the same result as foldRight, but using foldLeft. Can you guess what it is? Here's how:

```
list.foldRight("X")((a,b) => a + b)
list.reverse.foldLeft("X")((b,a) => a + b)
```

These two lines are equivalent! They give the same result no matter what's in list. Since foldRight processes list elements from last to first, that's the same as processing the reversed list from first to last.

Here are three possible implementations of foldRight that could replace the current one.

```
def foldRight[B](z: B)(f: (A, B) => B): B =
  reverse.foldLeft(z)((b,a) => f(a,b))

def foldRight[B](z: B)(f: (A, B) => B): B =
  if (length > 50) reverse.foldLeft(z)((b,a) => f(a,b))
  else             originalFoldRight(z)(f)

def foldRight[B](z: B)(f: (A, B) => B): B =
  try {
    originalFoldRight(z)(f)
  } catch {
    case e1: StackOverflowError => reverse.foldLeft(z)((b,a) => f(a,b))
  }
```

The first one simply replaces the original recursive logic with the equivalent call to reverse and foldLeft. Why wasn't foldRight implemented this way to begin with? It may be, in part, that the authors thought the extra overhead of reversing the list was unwarranted. To me, it doesn't seem that bad. The original foldRight and foldLeft functions are O(n), meaning they run in an amount of time roughly proportional to the number of items in the list. If you look at the source for the reverse function, you'll see it's also O(n). So running reverse followed by foldLeft is O(n).

The second implementation is a compromise. It uses the original recursive version of foldRight (referred to as originalFoldRight in the above code) only when the list is shorter than 50 elements. The reverse.foldLeft is used for lists of 50 elements or longer. 50 is just an arbitrary number, just a guess at a sensible limit on the number of recursive calls to allow.

The third implementation tries the original foldRight logic first and if the call stack overflows then it uses reverse.foldLeft. This solution is, of course, completely ridiculous, but even this would be better than a foldRight which sometimes crashes your program.

That's All, Folds!

As I pointed out before, the reverse.foldLeft implementation of foldRight is O(n), same as the original recursive version. The original foldRight may work just fine when your Scala application is young and working with small data sets. Over time more customers are added, more products are created, more orders are placed, and then one day, *POOF*, a runtime error! It's a ticking time-bomb.

As you may well guess, I would like to see the reverse.foldLeft logic used instead of the recursive version. That would prevent the stack overflow errors. But I would settle for just deprecating foldRight. It would be better to eliminate foldRight and force the coder to work around it than to leave it in its current state. In fact, I don't think any head-recursive functions belong in the List class.

Do any readers have any insight into why foldRight is coded the way it is?

Don't forget to subscribe to my RSS feed, or follow this blog on Twitter.

25 Responses to “Scala Code Review: foldLeft and foldRight”

1. Little tricks on working with LARGE list in Scala « Ekkmanz in geeky life! Says:

   November 18, 2009 at 6:37 am
   […] is dropRight since I use this method in order to reduce list size. Others also had shown that foldRight could cause this error as well. If you want to know which method may harm you then you may tries […]

   Reply
2. Zoheb Vacheri Says:

   August 7, 2010 at 8:15 pm
   reverse.foldLeft has at least O(n) complexity whereas foldLeft can execute in O(1) time.

   List(true,false,false).foldRight(false)(_||_) = = true

   In Haskell, foldr can operate on infinite lists, while foldl cannot

   Reply
3. Zoheb Vacheri Says:

   August 10, 2010 at 3:21 am
   I meant to say that *foldRight* can execute in O(1) time

   Reply
4. Matt Says:

   August 10, 2010 at 10:46 pm
   Zoheb, your comment is interesting, but I’m having trouble understanding how it can be so.

   Firstly, you say that reverse.foldLeft is at least O(n). I don’t see how it can be anything other than O(n). Reverse is clear enough. It builds a second singly linked list out of the first, one item at a time. That’s O(n). And then foldLeft runs, which is implemented as a while loop. That’s O(n). The combination is O(n+n) which is the same as O(n).

   Secondly (and this is the one that I’ve been puzzling over) I don’t think foldRight runs in O(1). Let’s take the foldRight call in your example.

```
scala> var x: Boolean = true
x: Boolean = true

scala> def time = (new java.util.Date).getTime
time: Long

scala> var list = (1 to 10).toList.map(x => x==1)
list: List[Boolean] = List(true, false, false, false, false, false, false, false, false, false)

scala> val t1 = time; for (_  val t1 = time; for (_ <- 1 to 10000) { x=list.foldRight(false)(_||_) }; println (time - t1)
16
t1: Long = 1281477156551
```

   This foldRight applied to a list of 10 booleans 10,000 times takes 16 milliseconds.

```
scala> list = (1 to 100).toList.map(x => x==1)
list: List[Boolean] = List(true, false, false, false, ...

scala> val t1 = time; for (_  val t1 = time; for (_ <- 1 to 10000) { x=list.foldRight(false)(_||_) }; println (time - t1)
94
t1: Long = 1281477234035
```

This foldRight applied to a list of 100 booleans 10,000 times takes 94 milliseconds. That's 10 times as many items and it took about 6 times as long.

```
scala> list = (1 to 1000).toList.map(x => x==1)
list: List[Boolean] = List(true, false, false, false, ...

scala> val t1 = time; for (_  val t1 = time; for (_ <- 1 to 10000) { x=list.foldRight(false)(_||_) }; println (time - t1)
984
t1: Long = 1281477246817
```

This foldRight applied to a list of 1000 booleans 10,000 times takes about 984 milliseconds. That's 10 times as many items and it took about 10 times as long.

```
scala> list = (1 to 2000).toList.map(x => x==1)
list: List[Boolean] = List(true, false, false, false, ...

scala> val t1 = time; for (_  val t1 = time; for (_ <- 1 to 10000) { x=list.foldRight(false)(_||_) }; println (time - t1)
1797
t1: Long = 1281477309082
```

This foldRight applied to a list of 2000 booleans 10,000 times takes about 1797 milliseconds. That's 2 times as many items and it took about 1.8 times as long.

This is all consistent with an O(n) foldRight, and it's what I would expect from the implementation. When you say that foldRight "can execute in O(1) time", do you mean that it should theoretically be possible for the compiler to short circuit the logic in certain applications of foldRight? Because I'm not seeing any such optimization in these numbers.

Reply

5. eddy Says:

   November 29, 2010 at 7:20 pm
   I think it's more a case of providing room for optimisation within the compiler, in the future. The compiler *could* short-circuit certain situations, meaning that deprecating foldRight just now would be unwise until such optimisation has been attempted.

   As Knuth said: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil"

   Thanks for your post; it was a good read.

   Reply
6. Josh Says:

   February 20, 2011 at 3:43 pm
   Please, stop punitively punishing us with your puns, pal.

   Reply
   1. Matt Says:

      February 21, 2011 at 1:28 am
      Never!

      Reply
7. Apache Camel med Scala « Stacktrace.se Says:

   March 22, 2011 at 2:57 pm
   […] sista vi gör är att använda den högre ordningens funktion foldLeft på översättningslistan. Här ser vi hur funktionell programmering kan bli otroligt kompakt och […]

   Reply
8. Christopher Currie Says:

   August 31, 2011 at 7:22 pm
   For future readers, this post was written just before Scala 2.8 was released. in 2.8+, non-strict collections are your friends:

   scala> 1 to 325000
   res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3…
   scala> res0.foldRight("")((x,y)=>"X")
   res1: java.lang.String = X

   scala> res0.toList.view
   res2: java.lang.Object with scala.collection.SeqView[Int,List[Int]] = SeqView(…)

   scala> res2.foldRight("")((x,y)=>"X")
   res3: java.lang.String = X

   Reply
   1. Matt Says:

September 1, 2011 at 10:34 pm
Thanks Christopher. Those look like good alternatives to a plain List.

Reply

9. gokcehan Says:

May 6, 2013 at 11:28 am
I'm checking the source of 2.10 and it looks like they have recently (3 months ago) done what you suggested in this commit:

https://github.com/scala/scala/commit/6db4db93a7d976f1a3b99f8f1bffff23a1ae3924

(don't know if you are responsible for this commit)

so the implementation in List.scala is now:

override def foldRight[B](z: B)(op: (A, B) => B): B =
reverse.foldLeft(z)((right, left) => op(left, right))

Reply

1. Matt Says:

May 31, 2013 at 1:56 pm
Thanks for pointing that out! I had nothing to do with the commit, but I'm glad to see it.

Reply

10. katagorikal Says:

September 17, 2013 at 8:30 am
The reason for making foldr 'slow', is so that people learn that foldr is slow and then adjust their algorithm to take that into account. The aim is to make something that shouldn't be done, painful enough that it is not done without thinking.

For example, including the reverse in the foldr is not an efficient solution if the algorithm contains many foldr operations, because you should reverse once, then use many foldl's. Only the algorithm writer can make these optimizations (unless you have a very good algebraic solver embedded in the compiler). Using your naive optimization would perform many needless reversals and end up being more inefficient than a well-informed programmer.

Reply

1. Matt Says:

September 17, 2013 at 4:05 pm
Why the quotes around 'slow'? That word doesn't appear in the article. I was never concerned that foldright might be too slow, but that it is liable to stack overflows. My naive optimization (which was committed to List.scala earlier this year ;)) is only intended to prevent overflows. But you are correct that, in general, foldleft is preferable to foldright. In the article I even held out the possibility that it be deprecated.

Reply

11. Michael Shepanski Says:

November 24, 2013 at 3:33 am
Your solution uses an additional O(N) temporary storage for the reversed list, in exchange for saving O(N) temporary storage on the call stack.

This is a practical improvement, yes, but only because the JRE has more stringent checks for the runtime stack than other dynamically allocated memory. That may explain why the original authors of foldRight() didn't take this approach.

Reply

12. Apache Camel med Scala | Cygni Says:

March 28, 2014 at 2:55 pm
[…] sista vi gör är att använda den högre ordningens funktion foldLeft på översättningslistan. Här ser vi hur funktionell programmering kan bli otroligt kompakt och […]

Reply

13. Ethereal Says:

June 8, 2014 at 12:10 pm
Outstanding blog. Very informative and deep sighted.

Reply

14. Andrei Suiu Says:

February 2, 2015 at 4:37 pm
I think that a better name for these fold functions would be "reduce", as it's named in Python, as well as there's no need of foldRight, as an reverse iterator can be easily applied over the list.

Reply

1. chesmartin Says:

   April 22, 2015 at 11:28 am
   Fold is a generalization of reduce, reduce is often implemented in terms of fold. Scala has a *reduce* method on collections in fact, with a simpler interface eschewing curried parameters:

   ```
   scala> List(1, 2, 3).reduce(_ * _)
   res14: Int = 6

   scala> List(2, 3).foldLeft(1)(_ * _)
   res15: Int = 6
   ```

   Scala's *reduce* implementation uses an imperative approach rather than *foldLeft* for efficiency, but a commenter did actually share an example implementation using fold in Matt's other article of fold examples.

   Reply

15. ravi Says:

    June 5, 2015 at 9:15 am
    nice blog… thank you

    Reply

16. [FYI] fold in Functional Programming | Tianhan's Blog Says:

    October 31, 2015 at 9:25 pm
    […] f [a1; …; an] b is f a1 (f a2 (… (f an b) …)). Not tail-recursive. References: 1. http://oldfashionedsoftware.com/2009/07/10/scala-code-review-foldleft-and-foldright/ 2. http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html Posted in […]

    Reply

17. Pratik Khadloya Says:

    November 16, 2015 at 8:13 pm
    Thanks for this helpful post!

    Reply

18. Evan Says:

    December 11, 2015 at 12:01 am
    I think there's little to no value in your performance comparisons. For one, `reverse.foldLeft` is certainly O(2n) not O(n). The reverse is O(n) and after it's completed you do another O(n) operation on n items so O(2n). Secondly, the cost of O is what really matters here. As you stated the recursive foldRight is also O(n) however, O is very expensive in this case so the performance difference isn't explained at all by that measure. Just my 2 cents, enjoyed the article overall.

    Another 2 cents about Scala; providing a better implementation of foldRight is a waste of effort. A better idea would be to provide a better implementation of List. I'm not sure what their arguments are for that choice but I've spent plenty of time comparing the two in other languages such as C# where List is backed by a dynamically resized array while LinkedList is implement as an actual linked list like Scala's List. There is no doubt about the performance differences. In most comparisons I've seen, even the operations you're taught linked lists are optimal for are substantially slower and for the things where we know arrays are faster, the performance isn't even comparable, it differs by something like a factor of 10.

    Reply

19. Evan Says:

    December 11, 2015 at 12:04 am
    fyi page only displays first 17 comments and has no way to get to the remaining five (six now).

    Reply

20. Apache Camel med Scala - Cygni | Cygni Says:

    January 21, 2016 at 9:16 pm
    […] sista vi gör är att använda den högre ordningens funktion foldLeft på översättningslistan. Här ser vi hur funktionell programmering kan bli otroligt kompakt och […]

    Reply