Open recursion and fixed points

Bernie Pope
Melbourne Scala Users Group
2011

Outline

- Computing the Fibonacci sequence as a motivating example.
- Implicit open recursion in object oriented style.
- Explicit open recursion using higher order functions and fixed points.

The Fibonacci sequence

- 1, 1, 2, 3, 5, 8, 13, 21 ...
- $X_0 = 1$
- $X_1 = 1$
- $X_n = X_{n-1} + X_{n-2}$

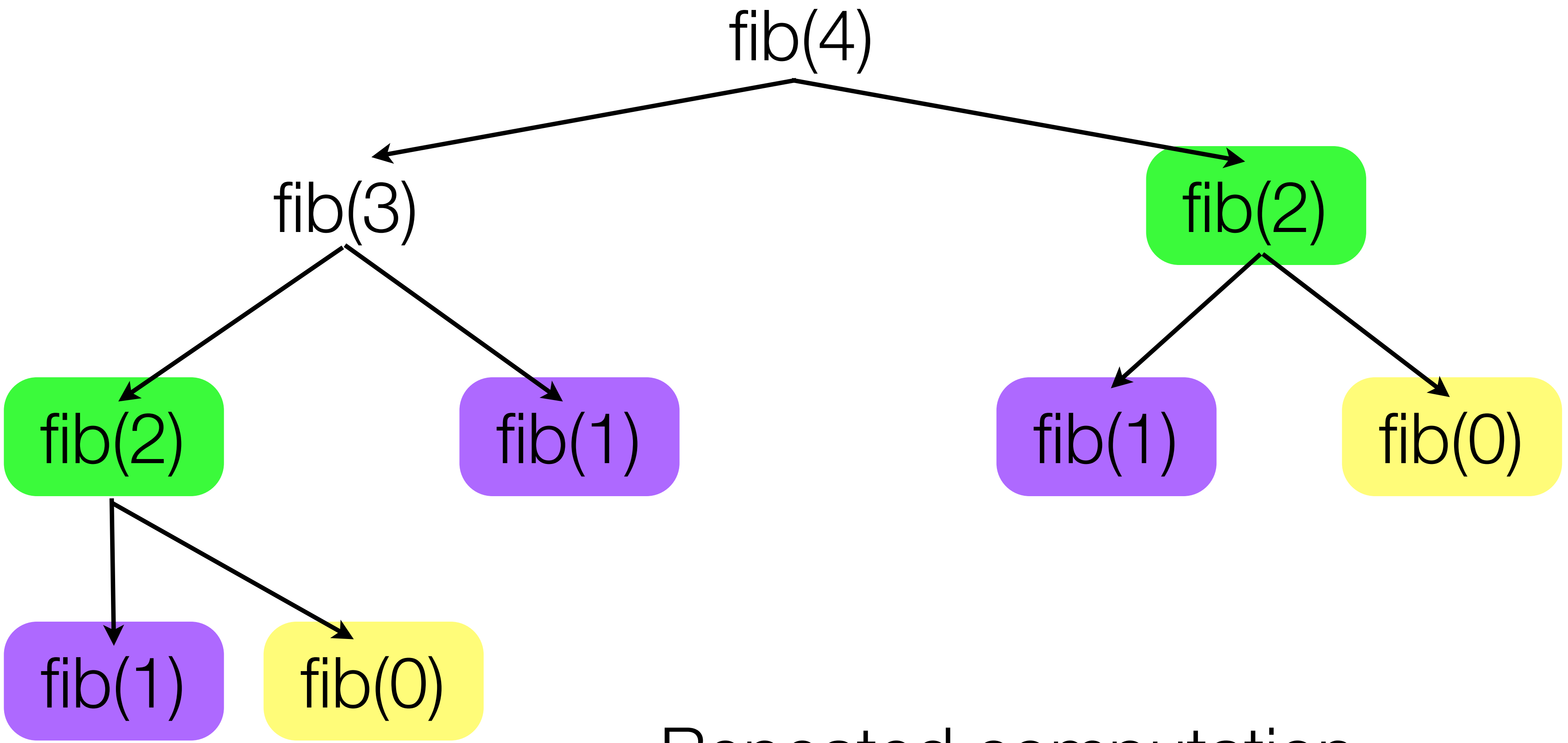
Computing the nth Fibonacci number in Scala

```
object Main {

   def fib(n:BigInt):BigInt =
      if (n <= 1) 1 else fib(n-1) + fib(n-2)

   def main(args: Array[String]) =
      (0 to 100) map (x => println(fib(x)))

}
```

Our previous solution is correct but slow

- Curiously, the complexity of the previous fib function is O(Fib(n)).
- That is to say, as the input n grows larger, the run time grows proportionally to the magnitude of the output.
- The growth of the Fibonacci sequence is exponential, so the run time of the previous fib function grows exponentially.

Why is it slow?



Repeated computation.

How to make it fast?

- This is an obvious candidate for *Dynamic Programming*.
- Tabulate the results of recursive calls: at each new call check if it was already computed, and if so, retrieve result from the table.
- Assuming that arithmetic is O(1), then we can improve the complexity of fib to O(n).
- We could modify the definition of fib directly to add the tabulation, but instead we are going to use it as a model for writing *extensible* programs.

The object oriented approach, first the slow way

```
class Fib() {
   def fib(n:BigInt):BigInt =
      if (n <= 1) 1 else this.fib(n-1) + this.fib(n-2)
}

object Main {
   def main(args: Array[String]) {
      val fibber = new Fib()
      (1 to 100) map (x => println(fibber.fib(x)))
   }
}
```

Now extend it to use a table - make it fast

```
import scala.collection.mutable.Map

class FibMemo () extends Fib() {
   val memo:Map[BigInt,BigInt] = Map()
   override def fib(n:BigInt):BigInt = {
      if (memo.contains(n))
          memo(n)
      else {
         val result = super.fib(n)
         memo(n) = result
         result
      }
   }
}
```

Implicit open recursion

From Fib.fib:

```
if (n <= 1) 1 else this.fib(n-1) + this.fib(n-2)
```

From FibMemo.fib:

```
val result = super.fib(n)
```

Closed recursion

Back to the original version:

```
def fib(n:BigInt):BigInt =
    if (n <= 1) 1 else fib(n-1) + fib(n-2)
```

The instance of fib in the body
is fixed at compile time.

Can we make it open?

Explicit open recursion

```
def fibOpen(r:BigInt=>BigInt)(n:BigInt):BigInt =
    if (n <= 1) 1 else r(n-1) + r(n-2)
```

Now the function called in the body is a parameter.

Explicit open recursion

Notice the change in type:

```
fib: (BigInt=>BigInt)

fibOpen: (BigInt=>BigInt)=>(BigInt=>BigInt)
```

How to close the recursion?

To get back the original fib,
we want something like:

```
fibOpen (fibOpen (fibOpen (fibOpen ...)))
```

Fixed points (in Mathematics)

- Given some function f, x is a fixed point of f if:

  $x = f(x)$

- Some functions have no fixed points:

  $f(x) = x+1$

- Some functions have exactly one fixed point:

  $f(x) = 3$

- Some functions have infinitely many fixed points:

  $f(x) = x$

Finding fixed points

- Given some function f, x is a fixed point of f if:

  x = f(x)

- We can say:

  x = fix(f)

  assuming some function fix, which can compute fixed points.

- So, substituting x = fix(f) into x = f(x):

  fix(f) = f(fix(f))

- Do some expanding:

  fix(f) = f(f(f(f ...)))

Writing fix in Scala

```
def fix[T](f:(T=>T)=>(T=>T)):T=>T =
   f((x:T) => fix(f)(x))
```

Remove some junk:

```
 fix   (f                ) =
   f(        fix(f)   )
```

Writing fix in Haskell

```
fix f = f (fix f)
```

Coping with eager evaluation

We have this:

```
def fix[T](f:(T=>T)=>(T=>T)):T=>T =
   f((x:T) => fix(f)(x))
```

But we really wanted this:

```
def fix[T](f:T=>T):T =
   f(fix(f))
```

Why the compromise?

Closing fibOpen

Note the types:

```
fix:       ((T=>T)=>(T=>T))=>T=>T
fibOpen:   (BigInt=>BigInt)=>(BigInt=>BigInt)
```

Take the fixed point of fibOpen

```
val fibSlow:BigInt=>BigInt = fix(fibOpen)
```

Do some expanding:

```
fibSlow = fibOpen(fibOpen(fibOpen ...))
```

How to make it go fast?

- So far we have:

  fibSlow = fix(fibOpen)

- We want to make a fast version by using the same tabling trick as before.

- Basic idea is to write an open recursive version of fibMemo, and then combine with fibOpen.

Open recursive version of tabled fib

```
val memo:Map[BigInt,BigInt] = Map()

def fibMemo(r:BigInt=>BigInt)(n:BigInt):BigInt = {
   if (memo.contains(n))
       memo(n)
   else {
      val result = r(n)
      memo(n) = result
      result
   }
}
```

Open recursive version of tabled fib

Notice the types:

```
fibOpen: (BigInt=>BigInt)=>(BigInt=>BigInt)

fibMemo: (BigInt=>BigInt)=>(BigInt=>BigInt)
```

Function composition (in Mathematics)

- Given some functions f and g, we define a composition operator:

  $(f \circ g)\ x = f\ (g\ (x))$

- Recall the fix function

  $fix(f) = f(fix(f))$

- We can take the fixed point of a function composition:

  $fix(f \circ g)$

  $= (f \circ g)(fix(f \circ g))$

  $= f(g(fix(f \circ g))$

  $= f(g(f(g(f(g(f(g\ \ldots)))))))$

Closing the fast version:

```
val fibFast:BigInt=>BigInt = fix(fibMemo _ compose fibOpen)
```

We can call fibFast like usual:

```
def main(args: Array[String]) =
    (1 to 100) map (x => println(fibFast(x)))
```

Closing the fast version:

```
val fibFast:BigInt=>BigInt = fix(fibMemo _ compose fibOpen)
```

Expanding a bit:

```
fibFast = fibMemo(fibOpen(fibMemo(fibOpen ...)))
```

Extending further

- There's nothing stopping us from composing fibOpen with other functions to extend it in other ways.

- Homework: write a version which prints the value of is argument at each recursive call.

Conclusion

- Open recursion is built into object oriented classes.
- Higher order functions provide all the tools we need to achieve the same affect.
- However, you generally don’t see this kind of extensibility in functional programming libraries.
    - Maybe not needed that often.
    - Quite tedious.