Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

Thoughts on Accumulators

- Accumulator-based functions are not always faster
  - Accumulator-based factorial tends to be slower
- Accumulator-based functions do not always take less space

Thoughts on Accumulators

- Accumulator-based functions are usually harder to understand

- Programmers new to functional programming are seduced by them because sometimes they can be similar to loops

Thoughts on Accumulators

- Use accumulators judiciously and understand the benefits you are trying to achieve

Accumulators and Trees

```
abstract class Tree[+T]

case object Empty extends Tree[Nothing]

case class Branch[+T](data: T, left: Tree[T], right: Tree[T])
extends Tree[T]
```

Accumulators and Trees

```
def height[T](tree: Tree[T]): Int = {
  tree match {
    case Empty => 0
    case Branch(d,l,r) => max(height(l), height(r)) + 1
  }
}
```

Accumulators and Trees

- One natural thing to try is to include an accumulator of type `Int`
- This accumulator can maintain the distance we have descended from the root of the tree

Accumulators and Trees

```
def height2[T](tree: Tree[T]): Int = {
  def inner(tree: Tree[T], accumulator: Int): Int = {
    tree match {
      case Empty => accumulator
      case Branch(d,l,r) => max(inner(l, accumulator + 1),
                                inner(r, accumulator + 1))
    }
  }
  inner(tree, 0)
}
```

Family Trees Revisited

```
abstract class FamilyTree

case object Empty extends FamilyTree

case class Cons(father: FamilyTree, mother: FamilyTree,
                name: String, birthYear: Int, eyes: String)
extends FamilyTree
```

Family Trees Revisited

- Let’s develop a method blueEyedAncestors that finds *all* blue-eyed ancestors in a tree

Family Trees Revisited

```
def blueEyedAncestors(tree: FamilyTree): List[String] = {
  tree match {
    case Empty => Nil
    case Cons(father,mother,name,_,eyes) => {
      val inParents = blueEyedAncestors(father) ++
                      blueEyedAncestors(mother)

      eyes match {
        case "blue" => name :: inParents
        case _ => inParents
      }
    }
  }
}
```

Family Trees Revisited

- We have defined a structurally recursive function that relies on an auxiliary recursive function: ++
- As discussed, functions of this form often benefit from the use of an accumulator
- We sketch a template for our accumulator-based function in the usual way

Family Trees Revisited

```
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: ...) = {
    tree match {
      case Empty => {...}
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(...father...accumulator...) ...
                        inner(...mother...accumulator...)
        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree...)
}
```

Formulating an Accumulator Invariant

- Our accumulator should remember knowledge about the family tree lost as we descend the tree
- There are two recursive applications: To the father tree and the mother tree
- Options:
  - Denote all blue-eyed ancestors encountered so far
  - Denote all the trees we still need to look at

Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]):
  List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, inner(mother, accumulator))

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]):
  List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, inner(mother, accumulator))

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Return type is determined by our choice of accumulator invariant

Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]):
  List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, inner(mother, accumulator))

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

We must pass in the result of one descent to the other to maintain the invariant.

Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]):
  List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, inner(mother, accumulator))

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Thus, our combining operator is function composition.

Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]):
  List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, inner(mother, accumulator))

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Our choice of invariant determines what to return in the Empty case.

Option 1: Denote All Blue-Eyed Ancestors Encountered So Far

```
def blueEyedAncestors2(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[String]):
  List[String] = {
    tree match {
      case Empty => accumulator
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, inner(mother, accumulator))

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Our choice also determines the initial value of the accumulator.

Option 2: Denote All Family Trees Not Yet Processed

```
def blueEyedAncestors3(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[FamilyTree]):
  List[String] = {
    tree match {
      case Empty => {...}
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, mother :: accumulator)

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

We must cons the mother tree on our accumulator for the recursive call to father, to maintain our invariant.

Option 2: Denote All Family Trees Not Yet Processed

```
def blueEyedAncestors3(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[FamilyTree]):
  List[String] = {
    tree match {
      case Empty => {...}
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, mother :: accumulator)

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Naturally, the only tree to process initially is tree, so our accumulator is Nil.

Option 2: Denote All Family Trees Not Yet Processed

```
def blueEyedAncestors3(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[FamilyTree]):
  List[String] = {
    tree match {
      case Empty => {...}
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, mother :: accumulator)

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

The Empty case is more difficult for this accumulator invariant.

Option 2: Denote All Family Trees Not Yet Processed

- When the tree is empty, we choose the next element in our accumulator to recur on

Option 2: Denote All Family Trees Not Yet Processed

```
def blueEyedAncestors3(tree: FamilyTree): List[String] = {
  def inner(tree: FamilyTree, accumulator: List[FamilyTree]): List[String] = {
    tree match {
      case Empty => accumulator match {
        case Nil => Nil
        case tree :: trees => inner(tree, trees)
      }
      case Cons(father,mother,name,_,eyes) => {
        val inParents = inner(father, mother :: accumulator)

        eyes match {
          case "blue" => name :: inParents
          case _ => inParents
        }
      }
    }
  }
  inner(tree, Nil)
}
```

Tail Recursion

Tail Recursion

- Some functions defined using accumulators have a special property:
  - The recursive call occurs as the last step in the computation

Nats

```
abstract class Nat {
  def !(): Nat
  def *(m: Nat): Nat
  def +(m: Nat): Nat
}
```

Note that this is a postfix operator.
(This follows from the rules for
method application syntax.)

Nats

```
case object Zero extends Nat {
  def !() = Next(Zero)
  def *(m: Nat) = Zero
  def +(m: Nat) = m
}
```

Nats

```
case class Next(n: Nat) extends Nat {
  def !() = this * (n!)
  def *(m: Nat) = m + (n * m)
  def +(m: Nat) = Next(n + m)
}
```

Nats

```
Next(Next(Next(Zero)))! ↦
Next(Next(Next(Zero))) * Next(Next(Zero))! ↦
Next(Next(Next(Zero))) * Next(Next(Zero)) * Next(Zero)! ↦
Next(Next(Next(Zero))) * Next(Next(Zero)) * Next(Zero) * Zero! ↦
Next(Next(Next(Zero))) * Next(Next(Zero)) * Next(Zero) * Next(Zero) ↦
…
Next(Next(Next(Next(Next(Next(Zero))))))
```

Tail Recursion

```
def !() = this * (n!)
```

Tail Recursion

```
 def !() = {
   def inner(n: Nat, acc: Nat): Nat = {
     n match {
       case Zero => acc
       case Next(m) => inner(m, n * acc)
     }
   }
   inner(this, Next(Zero))
 }
}
```

Nats

```
Next(Next(Next(Zero)))! ↦
inner(Next(Next(Next(Zero))), Next(Zero)) ↦
inner(Next(Next(Zero)), Next(Next(Next(Zero)))) ↦
inner(Next(Zero), Next(Next(Next(Next(Next(Next(Zero)))))) ↦
inner(Zero, Next(Next(Next(Next(Next(Next(Zero))))))) ↦
Next(Next(Next(Next(Next(Next(Zero))))))
```

Translating for Ints

```
def factorial(n: Int): Int = {
  if (n == 0) 1
  else n * factorial(n - 1)
}

def factorial2(n: Int) = {
  def inner(n: Int, acc: Int): Int = {
    if (n == 0) acc
    else inner(n - 1, n * acc)
  }
  inner(n, 1)
}
```

Pure Recursion with Ints

```
3! ↦
3 * 2! ↦
3 * 2 * 1! ↦
3 * 2 * 1 * 0! ↦
3 * 2 * 1 * 1 ↦
…
6
```

Tail Recursion with Ints

```
3! ↦
inner(3, 1) ↦
inner(2, 3) ↦
inner(1, 6) ↦
inner(0, 6) ↦
6
```