Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

October 24, 2019

Graph Algorithms

- Many problems can be expressed as traversals or computations over graphs
  - Travel planning
  - Circuit design
  - Social networks
  - etc.

Graph Algorithms

- We consider the problem of finding a path from one vertex to another in a graph

Data Analysis and Design

- We model graphs as Maps of Strings to Lists of Strings

```
case class Graph(elements: (String, List[String])*)
extends Function1[String, List[String]] {
  val _elements = Map(elements:_*)
  def apply(s: String) = _elements(s)
}
```

Data Analysis and Design

- We model graphs as Maps of Strings to Lists of Strings

```
val sampleGraph =
  new Graph ("A" -> List("E", "B"),
             "B" -> List("A"),
             "C" -> List("D"),
             "D" -> List(),
             "E" -> List("C", "F"),
             "F" -> List("A", "G"),
             "G" -> List())
```

What is a Trivially Solvable Problem?

- If the start and end vertices are identical

How Do We Generate Sub-Problems?

- Find nodes connected to start and recur

How Do We Relate the Solutions?

- We need only find one solution; no need to combine multiple solutions

Contract Attempt 1

```
/**
 * Create a path from start to finish in G
 */
def findRoute(start: String, end: String,
              graph: Graph): List[String]
```

But what if there is no path?

Options

- Often the result of a computation is that no satisfactory value could be found
  - Lookup in a table with a key that does not exist
  - Attempting to find a path that does not exist

Scala Options

```
abstract class Option[+A] {…}

object None extends Option[Nothing] {…}

class Some[+A](val contained: A) extends Option[A] {
  …
}
```

Options Are Monads!

```scala
abstract class Option[+A] {
  def flatMap[B](f: (A) ⇒ Option[B]): Option[B]
  def map[B](f: (A) ⇒ B): Option[B]
  def withFilter(p: (A) ⇒ Boolean):
    FilterMonadic[A, collection.Iterable[A]]
}
```

Contract Attempt 2

```
/**
 * Create a path from start to finish in G, if
 * it exists.
 */
def findRoute(start: String, end: String,
              graph: Graph):
             Option[List[String]]
```

Reduce to Backtracking Cases

```scala
def findRoute(start: String, end: String,
              graph: Graph): Option[List[String]] = {
  if (start == end) Some(List(end))
  else for (route <- routeFromOrigins(graph(start), end, graph))
         yield start :: route
}
```

Recursive Sub-Problems

```scala
def routeFromOrigins(origins: List[String], destination: String,
                     graph: Graph): Option[List[String]] = {
  origins match {
    case Nil => None
    case origin :: origins => {
      findRoute(origin, destination, graph) match {
        case None => routeFromOrigins(origins, destination,graph)
        case Some(route) => Some(route)
      }
    }
  }
}
```

Termination

- `routeFromOrigins` is structurally recursive:
  - It terminates provided that findRoute terminates
- But `findRoute` terminates only if there are no cycles in the graph it traverses

Contract for findRoute Attempt #2

```
/**
 * Create a path from start to finish in G, if
 * it exists.
 */
def findRoute(start: String, end: String,
              graph: Graph):
             Option[List[String]]
```

Reduce to Backtracking Cases

```
def findRoute(start: String, end: String,
              graph: Graph): Option[List[String]] = {
  if (start == end) Some(List(end))
  else for (route <- routeFromOrigins(graph(start), end, graph))
         yield start :: route
}
```

How does Scala's for-expression work with an `Option`*?*

Recursive Sub-Problems

```scala
def routeFromOrigins(origins: List[String], destination: String,
                     graph: Graph): Option[List[String]] = {
  origins match {
    case Nil => None
    case origin :: origins => {
      findRoute(origin, destination, graph) match {
        case None => routeFromOrigins(origins, destination,graph)
        case Some(route) => Some(route)
      }
    }
  }
}
```

Termination

- `routeFromOrigins` is structurally recursive:
    - terminates provided that findRoute terminates
- `findRoute` terminates only if graph is acyclic

Contract for findRoute Attempt #3

```
/**
 * Create a path from start to finish in G, if
 * it exists. May diverge if graph has a cycle.
 */
def findRoute(start: String, end: String,
             graph: Graph):
            Option[List[String]]
```

Accumulating Knowledge

Accumulating Knowledge

- Remember visited nodes to prevent infinite regress
- Pass this to recursive calls via an *accumulator*

Reduce to Backtracking

```
def findRoute(start: String, end: String, graph: Graph,
              visited: List[String] = Nil):
Option[List[String]] = {
  if (start == end) Some(List(end))
  else if (visited contains start) None
  else for (route <- routeFromOrigins(graph(start), end, graph,
                                       start :: visited))
         yield start :: route
}
```

Reduce to Backtracking

```
def routeFromOrigins(origins: List[String], destination: String,
                     graph: Graph, visited: List[String]):
Option[List[String]] = {
  origins match {
    case Nil => None
    case origin :: origins => {
      findRoute(origin, destination, graph, visited) match {
        case None => routeFromOrigins(origins, destination,
                                      graph, origin :: visited)
        case Some(route) => Some(route)
      }
    }
  }
}
```

Can we still guarantee termination without this `cons` *operation?*

Accumulators

- An *accumulator* parameter allows us to “remember” knowledge from one recursive call to another
  - Often essential for correctness in generative recursion
  - Useful for saving space in structural recursion
  - Also critical for supporting tail-calls in many cases

Using Accumulators for Structural Recursion

- Let us define a function `fromOrigin`, which:
  - Takes a list of `Int` values, with each value denoting a relative distance to the point to its left
  - Returns a list of `Int` values denoting the absolute distances to the origin

fromOrigin Example

Applying `fromOrigin` to the following input list

2	3	5	2	8	...

results in the following output list

2	5	10	12	20	...

Defining fromOrigin

```
def fromOrigin(xs: List[Int]): List[Int] = {
  xs match {
    case Nil => Nil
    case x :: xs => x :: fromOrigin(xs).map(_+x)
  }
}
```

Defining fromOrigin

```
def fromOrigin(xs: List[Int]): List[Int] = {
  xs match {
    case Nil => Nil
    case x :: xs =>
      x :: { for (y <- fromOrigin(xs)) yield y+x }
  }
}
```

How many steps does it take to compute an application of fromOrigin, in comparison to the length of the list?

Cost of fromOrigin

```
fromOrigin(List(2,3,5,2,8)) ↦
    List(2,3,5,2,8) match {
      case Nil => Nil
      case x :: xs => x :: fromOrigin(xs).map(_+x)
    } ↦

  2 :: (fromOrigin(List(3,5,2,8)) map (_+2)) ↦*
  2 :: (3 :: (fromOrigin(List(5,2,8) map (_+3))) map(_+2)) ↦*
  2 :: (3 :: (List(5, 7, 15) map (_+3))) map(_+2)) ↦*
  2 :: (3 :: (List(8, 10, 18)) map(_+2)) ↦*
  2 :: (List(5, 10, 12, 20)) ↦*
  List(2, 5, 10, 12, 20)
```

Cost of fromOrigin

- Each recursive call map over the argument list
  - which takes *n* steps for a list of length *n*

$$\sum_{i=1}^{n} i = \frac{(n)(1+n)}{2} = O(n^2)$$

Big O Notation

We say:

$$f(x) = O(g(x)) \;\; \texttt{as} \;\; x \to \infty$$

meaning there is a constant *k* and some value x_0 such that

$$|f(x)| \leq k|g(x)| \;\; \texttt{for all} \;\; x \geq x_0$$

Big O Notation

Typically the part:

$$\texttt{as}\ \ x \to \infty$$

is implicit

Effectively, we are defining equivalence classes of functions

Accumulating Distance to the Origin

We could reduce the time taken by instead accumulating the distance to the origin in a parameter

Accumulating Distance to the Origin

```scala
def fromOriginAcc(xs: List[Int]) = {
  def inner(xs: List[Int], fromOrigin: Int): List[Int] = {
    xs match {
      case Nil => Nil
      case x :: xs => {
        val xToOrigin = x + fromOrigin
        xToOrigin :: inner(xs, xToOrigin)
      }
    }
  }
  inner(xs, 0)
}
```

Guidelines for Using Accumulators in Functions

- Start with the standard design recipes!
- Add an accumulator *only after* the initial design attempt

Guidelines for Using Accumulators in Functions

- Recognize the benefit of having an accumulator
- Understand what the accumulator denotes

Recognizing the Benefit of an Accumulator

- If the function is structurally recursive and uses an auxiliary function, consider an accumulator
    - Study hand evaluations to see if an accumulator helps in reducing time or space costs

Recognizing the Benefit of an Accumulator

```scala
def invert[T](xs: List[T]): List[T] = {
  xs match {
    case Nil => Nil
    case x :: xs => makeLastItem(x, invert(xs))
  }
}

def makeLastItem[T](x: T, xs: List[T]): List[T] = {
  xs match {
    case Nil => List(x)
    case y :: ys => y :: makeLastItem(x, ys)
  }
}
```

Recognizing the Benefit of an Accumulator

- there is nothing for invert to forget
- consider accumulating the items walked over

Recognizing the Benefit of an Accumulator

```
def invert[T](xs: List[T]): List[T] = {
  def inner(xs: List[T], accumulator: List[T]): List[T] = {
    xs match {
      case Nil => …
      case y :: ys => … inner(… ys … y … accumulator …)
    }
  }
  inner(xs, Nil)
}
```

Recognizing the Benefit of an Accumulator

- accumulator must stand for a list
- it could stand for all elements that precede `xs`

Recognizing the Benefit of an Accumulator

```
def invert[T](xs: List[T]): List[T] = {
  def inner(xs: List[T], accumulator: List[T]): List[T] = {
    xs match {
      case Nil => …
      case y :: ys => … inner(… ys … y :: accumulator)
    }
  }
  inner(xs, Nil)
}
```

Recognizing the Benefit of an Accumulator

- Now it is clear that the accumulator contains all the elements that precede xs *in reverse order*

Recognizing the Benefit of an Accumulator

```
def invert[T](xs: List[T]): List[T] = {
  def inner(xs: List[T], accumulator: List[T]): List[T] = {
    xs match {
      case Nil => accumulator
      case y :: ys => inner(ys, y :: accumulator)
    }
  }
  inner(xs, Nil)
}
```

Recognizing the Benefit of an Accumulator

- The key step in the design process is to establish the invariant that describes the relationship between the accumulator and the parameters of a function
- Establish appropriate accumulator invariant is an art that takes practice

Recognizing the Benefit of an Accumulator

```scala
def sum1(xs: List[Int]): Int = {
  xs match {
    case Nil => 0
    case y :: ys => y + sum1(ys)
  }
}
```

An Accumulator for Sum

- walking over elements of a list to return their sum
- obvious thing to accumulate is the the sum so far

An Accumulator for Sum

```
def sum2(xs: List[Int]): Int = {
  def inner(xs: List[Int], accumulator: Int): Int = {
    xs match {
      case Nil => accumulator
      case y :: ys => inner(ys, y + accumulator)
    }
  }
  inner(xs, 0)
}
```

Reducing Naïve Sum

```
      sum1(List(5, 3, 7, 9)) ↦*
     5 + sum1(List(3, 7, 9)) ↦*
    5 + 3 + sum1(List(7, 9)) ↦*
   5 + 3 + 7 + sum1(List(9)) ↦*
5 + 3 + 7 + 9 + sum1(List()) ↦*
        5 + 3 + 7 + 9 + 0 ↦
          8 + 7 + 9 + 0 ↦
            15 + 9 + 0 ↦
              24 + 0 ↦
                 24
```

Reducing Accumulated Sum

```
sum2(List(5, 3, 7, 9)) ↦*
inner(List(5, 3, 7, 9), 0) ↦*
inner(List(3, 7, 9), 5 + 0) ↦*
inner(List(3, 7, 9), 5) ↦*
inner(List(7, 9), 5 + 3) ↦*
inner(List(7, 9), 8) ↦*
inner(List(9), 7 + 8) ↦*
inner(List(9), 15) ↦*
inner(List(), 9 + 15) ↦*
inner(List(), 24) ↦*
24
```

An Accumulator for Sum

- The key advantage of our accumulator version of sum is space
- The advantage is not a matter as to whether the space is used on the stack or in the heap as an argument!
- The ability to reduce the sum as we recur is the primary cause of space savings

This Would Not Save Space

```scala
def sum3(xs: List[Int]): Int = {
  def inner(xs: List[Int], accumulator: () => Int): Int = {
    xs match {
      case Nil => accumulator()
      case y :: ys => inner(ys, () => (y + accumulator()))
    }
  }
  inner(xs, () => 0)
}
```

Thoughts on Accumulators

- Accumulator-based functions are not always faster
- 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