Functional Programming

CIS-3030, Vermont Technical College
Peter C. Chapin

Characteristics

- Typical features of functional languages…
  - Immutable data, side-effect free operations
  - “First class” functions
    - Anonymous function literals
    - Functions can be stored in data structures (Lists, Arrays)
    - Functions can be passed to functions
    - Functions can be returned from functions
  - Programs seen as “data transformers”
  - Pattern matching

Side Effects

- Pure function only returns a result.
  - No other “side effects” such as…
    - Input/Output
    - State change of hardware
    - Modification of (global) memory state
    - Modification of (object) memory state
    - Modification of operating system state
      - Network connections
      - Open files

No I/O?

- Paradox…
  - Programs must obtain input from and write output to the external universe.
  - “Impossible” to do in a purely functional setting!
- Solution…
  - Allow mixed paradigm programming (Scala)
  - Abstract external universe into a value and thread it through the functions (Haskell, Mercury).
    - Discuss more later.

Which?

- “The universe is imperative!”
  - When we act we change the state of the universe.
    - Our actions have side effects.
    - The universe is modified by our actions (mutable)
- “The universe is functional!”
  - Our actions are a function taking the past universe into the future universe.
    - Past is not changed by our actions (immutable)
    - The future depends only on the past.

Example

- "The universe is imperative"
  - ```
    def blowUpBuilding() = {
      placeBomb()
      lightFuse()
      runAway()
    }
    ```
- "The universe is functional"
  - ```
    def blowUpBuilding(old: Universe): Universe = {
      val withBombPlaced = placeBomb(old)
      val withFuselite = lightFuse(withBombPlaced)
      runAway(withFuseLite)
    }
    ```

Global Data

- You've heard "global data is bad." Why?
  - Modifications of global data hard to track.
    - Confusing. Does `doStuff(1, "Hello")` change the state of global variable x? Who knows?
- Problem is really side effects.
  - Other side effects are just as hard to track.
    - Confusing. Does `doStuff(1, "Hello")` change the state of the network? Who knows?
    - ```
      val newNetwork =
        doStuff(1, "Hello", oldNetwork)
      ```
      ```

Transformative

- Emphasis on transformations of data
  - ```
    val page = getWebPage("http://www.xyz.com")
    val errors = validate(page)
    errors.length  // Number of errors.
    ```
  - The vals are just names for intermediate values.
    - ```
      validate(
        getWebPage("http://www.xyz.com")).length
      ```
  - Output of one function feeds the next.
    - Everything is just one expression.
    - Program is a big function: takes input and produces output.

Referential Transparency

- ***Defn**: An expression can be replaced by its result everywhere the expression occurs.*
- Example:
  - `(a + b) / 2 – (a + b) / 3`
  - ```
    val sum = a + b
    sum / 2 – sum / 3
    ```
  - The expression a + b is referentially transparent.
  - In a pure functional language, *all* expressions have this property.
    ```

Expressions and Side Effects

- Side effects wreck referential transparency
  - `readLine() + readLine()`
  - ```
    val aLine = readLine()
    aLine + aLine
    ```
  - The behavior is very different!
- Benefits of referential transparency
  - Easier to reason about problem
  - Easier to optimize program
  - Easier to restructure (refactor) program
    ```

Type Unit

- Functions using Unit
  - Functions taking no parameters either
    - … always return the same thing (constants)
    - … have side effects
  - Functions returning Unit either
    - … do nothing
    - … have side effects

Side Effects in Scala

- Scala is a mixed paradigm language
  - Allows pure functional programming
  - Allows imperative programming with side effects
- Good?
  - Use imperative style when appropriate
    - I/O
    - Interacting with external hardware
    - Interfacing with imperative libraries

Imperative Programming

- You are doing imperative programming if…
  1. You are using functions with no parameters (that do something other than return a constant)
  2. You are using the type Unit
  3. You are using vars (assignment is a side effect)
  4. You are using while loops
- *These things arise naturally in Scala when doing I/O, interfacing with Java, and similar things.*

First Class Functions

- Functions can be treated as data
  - *“Functions are values”*
  - … can be written literally
  - … can be stored in data structures
  - … can be passed around the program

Function Types

- Syntax of function types
  - *(parameter_type_list) => result_type*
  - If only one parameter the parenthesis are optional
- Examples
  - `Int => Int` (pronounced "Int to Int")
  - `(Int, String) => Unit`
  - `(Int, List[Cat]) => (Int, String)`
  - `(Int, (Int, String)) => Cat`
  - ```
    (Int, String => Double) =>
        List[Int => Int]
    ```
    ```

Lambda Terms

- Function expressions go by many names
  - "Lambda term" or just "lambdas"
    - Comes from the Lambda Calculus
  - "Function literal"
  - "Anonymous function"
  - "Closure"
    - A closure is actually something more. See later slides.

Scala Syntax

- Examples
  - `(x: Int, y: Int) => x + y`
    - Parameters declared as usual
    - Result given by single expression after =>
    - Type interference computes result type
    - Function has no name (anonymous)
    - Body can have any complexity (enclose in braces)
  - `((x: Int, y: Int) => x + y)(1, 2)`
    - Applies anonymous function to argument list

Functions as Values

- Examples
  - `val f: Int => Int = (x: Int) => x + 1`
    - Type annotation not needed
    - `val f = (x: Int) => x + 1`
  - ```
    val myList = List(
      (x: Int) => x + 1,
      (x: Int) => x – 1,
      (x: Int) => 2 * x )
    ```
    - `myList` has type `List[Int => Int]`.
    ```

Function Expressions

- Expressions can evaluate to functions
  - ```
    val operator = if (x < y)
        (x: Int, y: Int) => x + y
      else
        (x: Int, y: Int) => x – y

    val result = operator(1, 2)
    ```
  - Notice: *x and y used in the functions are different than the x and y used in the condition*
    - The function parameters hide x and y from the outer scope.
    ```

Function Expressions

- Or even just…
  - ```
    val result =
      (if (x < y)
         (x: Int, y: Int) => x + y
       else
         (x: Int, y: Int) => x - y)(1, 2)
    ```
    ```

Type Aliases

- You can define short names for long types
  - ```
    type CatProcessor = (Int, Cat) => Cat
    val f: CatProcessor = …
    def workWith(p: CatProcessor) = …
    ```
  - Good for documentation
  - Improves readability
  - Does *not* introduce a new type
    - Replacing the alias with the original type does not change the meaning of the program.
    ```

Functions? Methods?

- Scala distinguishes between them
  - … but converts methods to functions by creating a closure (see future slides)
  - **`def`** `inc(x: Int) = x + 1`
    - Has type `(Int)Int`
  - **`val`** `inc = (x: Int) => x + 1`
    - Has type `Int => Int`
  - *Methods are always applied to an object and have access to the fields of that object*.

Filter

- Selects elements that satisfy a predicate
  - ```
    def filter[A](myList: List[A],
                  pred  : A => Boolean): List[A] =
      myList match {
        case Nil => List()
        case head :: tail =>
          if (pred(head))
            head :: filter(tail, pred)
          else
            filter(tail, pred)
      }
    ```
    ```

Use of `filter`

- Example
  - ```
    def isEven(x: Int) =
      if (x % 2 == 0) true else false

    val myList = List(1, 2, 3, 4)
    val filteredList = filter(myList, isEven)
    ```
  - The result: `List(2, 4)`
  - ```
    val filteredList = filter(myList,
      (x: Int) => if (x % 2 == 0) true else false)
    ```
  - ```
    val filteredList = filter(myList,
      (x: Int) => x % 2 == 0)
    ```
    ```

Foreach

- Applies a function to each element of a list
  - **def** foreach[A](myList: List[A], f: A => Unit): Unit
- Example
  - **val** myList = List(“Hello”, “World”)
    foreach(myList, (s: String) => println(s))

Map

- Transforms elements
  - `def map[A, B](myList: List[A], trans: A => B): List[B]`
- Example
  - ```
    val myList = List(1, 2, 3)
    map(myList, (x: Int) => x + 1)
    ```
  - Evaluates to `List(2, 3, 4)`
    ```

FlatMap

- Transform elements to lists and flattens result
  - ```
    def flatMap[A, B](myList: List[A],
                      trans : A => List[B]):
                                         List[B]
    ```
- Example
  - ```
    def getWords(lines: List[String]) =
      flatMap(
        lines,
        (line: String) => line.split(“\\W+”)

    val words =
      getWords(List(“Line One”, “Line Two”))
    ```
  - Evaluates to `List(“Line”, “One”, “Line”, “Two”)`

Workhorses

- These higher order methods are essential
  - `filter`
  - `foreach`
  - `map`
  - `flatMap`
- *Learn them well!*
  - Scala provides these methods with all the collections in the Scala library!

Example Transformations

- Let `args` be Array[String] command line.
  - Each string starting with “-” is an option…
  - ```
    val options = args.filter( (arg: String) =>
      if (arg.charAt(0) == ‘-’) true else false )
    val rawOptions = options.map( (opt: String) =>
      opt.substring(1) )
    rawOptions.foreach( (opt: String) =>
      process(option) )
    ```
  - No loops. No mutable data.
    - … but `process(option)` must have side effects. (Why?)
    ```

Syntactic Abbreviations

- Together very powerful…
  - Methods taking one parameter…
    - The dot and parenthesis around argument optional
    - The argument can be enclosed in braces
  - Anonymous functions…
    - Can use _ for the parameter name provided
      - The parameter is used only once
      - The parameters are used in the same order as declared
      - The parameter declarations can be omitted in this case
      - Type inference at the use site will infer parameter types

Example Abbreviations

- Consider
  - ```
    val f = (x: Int) => x + 1
    val f: Int => Int = (x: Int) => x + 1
    val f: Int => Int = _ + 1
    ```
  - The expression `_ + 1` represents a function
    - … taking one parameter we don't name
    - … and returning the result of adding one to that param
    - The type of `_ + 1` is inferred to be `Int => Int` because of the context of where it is used.
    ```

More Typical Usage

- Abbreviations with higher order methods
  - ```
    val myList = List(1, 2, 3, 4)
    myList.filter( (x: Int) => x % 2 == 0 )
    myList.filter( _ % 2 == 0 )
    myList filter { _ % 2 == 0 }
    ```
  - The last form is typical Scala.
    - It is a *syntactic sugar* for the earlier forms.
    ```

Example Transformations

- Let `args` be Array[String] command line.
  - Each string starting with "-" is an option…
  - ```
    args filter  { _.charAt(0) == '-' }
         map     { _.substring(1)     }
         foreach { process(_)         }
    ```
  - Remember your workhorses
    - *Now we're talking Scala!*
    ```

Option Revisited

- Option represents optional data
  - … but it can also be a collection of 0 or 1 items.
  - Fully supports filter, map, foreach.
  -
```
getUser(ID) filter  { _.startsWith(“J”) }
            map     { lookupAccount(_)  }
            foreach { account =>
  if (account.balance < 0) sendMail(account) }
```
    - If getUser returns None, there is no effect.
    - If the name doesn’t start with “J” filter returns None
    - *This is idiomatic Scala!*
    - Notice wildcard can’t be used in last function. (Why?)

FoldLeft

- Collapses a sequence into a single value
  - ```
    def foldLeft[A, B](myList  : List[A],
                       accum   : B,
                       combiner: (B, A) => B) : B =
      myList match {
        case Nil => accum
        case head :: tail =>
          foldLeft(tail,
                   combiner(accum, head),
                   combiner)
      }
    ```
  - The code is elegant: simple, yet powerful
    ```

Maximum Length?

- Find the length of the longest string…
  - ```
    def findLongest(lines: List[String]) =
      foldLeft(lines, 0, (curMax, curString) =>
        if (curString.length > curMax)
          curString.length
        else
          curMax)
    ```
  - Example
    - `findLongest(List("Hi", "There"))` evaluates to 5.
    ```

Wait, What?

- The previous example doesn't work!
  - Scala can't infer the parameter types of the function from it's context:
    - foldLeft(List[String=A], Int=B, (?, ?) => ?)
    - Compiler learns the type B too late to use it later in the same argument list
    - Quirk/weakness of Scala type inference

Multiple Parameter Lists

- Methods can have multiple parameter lists
  –
    ```
    def m(x: Int, y: Int)(s: String) = {
      // Use x, y, and s
    }
    ```
  – `val result = m(1, 2)("Hello")`
  – This feature has several uses.
    - Right now: *types inferred in one parameter list are known when analyzing the next parameter list*.

FoldLeft Revisted

- Collapses a sequence into a single value
  - ```
    def foldLeft[A, B](myList  : List[A],
                       accum   : B)
                      (combiner: (B, A) => B) : B =
      myList match {
        case Nil => accum
        case head :: tail =>
          foldLeft(tail,
                   combiner(accum, head))
                  (combiner)
      }
    ```
  - Note small change to two parameter lists.
    ```

Maximum Length Revisited

- Find the length of the longest string…
  - ```
    def findLongest(lines: List[String]) =
      foldLeft(lines, 0)( (curMax, curString) =>
        if (curString.length > curMax)
          curString.length
        else
          curMax)
    ```
  - Example
    - `findLongest(List("Hi", "There"))` evaluates to 5.
    - … and it works this time!
    ```

Adding a List of Integers

- Very simple application of foldLeft
  - Fully desugared
    - `myList.foldLeft(0)((x: Int, y: Int) => x + y)`
  - Types can be inferred
    - `myList.foldLeft(0)( (x, y) => x + y )`
  - Wildcards can be used (Why?)
    - `myList.foldLeft(0)( _ + _ )`
  - Multiplying a list of integers
    - `myList.foldLeft(1)( _ * _ )`

Exercise

1. Revise `findLongest` so that it returns a pair `(Int, String)` consisting of the length of the longest string and the text of the longest string.

Consider…

- Write a function `associate` that takes a `String` and a `List[String])` and returns a `List[(String, String)]` where the first component of each pair in the result list is the first parameter.
  - `associate( “afile.txt”, List(“Error: line 1”, “Error: line 2”) )`
  - Evaluates to `List((“afile.txt”, “Error: line 1”), (“afile.txt”, “Error: line 2”))`

Implementation

- Associate uses a closure
  - `def associate[A, B](common: A, notes:List[B]) = notes map { (common, _) }`
  - Inside function `(x: B) => (common, x)` where does `common` come from?

Free vs Bound

- ***Defn**: Bound Variable is a name bound to a declaration*
  - `(x: Int) => x + 1`
    - `x` is "bound" to the parameter declaration
- ***Defn**: Free Variable is a name that is not bound*
  - `(x: Int) => x + y`
    - y is "free" because there is declaration of y here

Closed Expressions

- ***Defn**: A closed expression is one with no free variables*
  - `(y: Int) => ((x: Int) => x + y)`
    - This expression is closed
  - `(y: Int) => common + y`
    - This expression is not closed because `common` is free
  - All meaningful programs are closed expressions
    - Free variables are “unresolved references” or “undefined identifiers.”

Closures

- Consider this example
  - `def makeAdder(x : Int) = (v: Int) => x + v`

    ```
    def makeAdder(x : Int) =
      (v: Int) => x + v
    ```
    - Returns a function that depends on parameter x
    - The function returned is not, by itself closed
    - Yet this compiles
  - Compiler returns a *closure*: a function together with references to all the free variables required
    - Can be used like any function

Examples

- How `makeAdder` might be used
  - ```
    val f = makeAdder(5)
    println( f(3) )  // prints 8

    val g = makeAdder(10)
    println( g(3) ) // prints 13
    ```
  - Inside `f` and `g`, usages of “`x`” reference the object named `x` when the closure was created.
    ```

Closures and Mutability

- Consider…
  - ```
    def makeArrayAccessor(a: Array[Int]) =
      (index: Int) => a(index)

    myArray = Array(1, 2, 3)
    val accessor = makeArrayAccessor(myArray)
    println( accessor(0) )  // prints 1
    myArray(0) = 2
    println( accessor(0) )  // prints 2!
    ```
  - Lesson: *Avoid mutable data*.
    ```

Closures are Natural

- You don't have to think about them
  - `def scaleList(myList: List[Int], factor: Int) = myList map { factor * _ }`
    - Returns a new list where each element is scaled by factor
    - The function `factor * _` (which is syntactic sugar for `(x: Int) => factor * x`) is a closure. (Why?)
  - `scaleList( List(1, 2), 2 ) == List(2, 4)`