Classical Themes of Computer Science
Functional Programming (Part 1/2)

Bernhard K. Aichernig

Institute for Software Technology
Graz University of Technology
Austria

Winter Term 2017/18

Version: November 22, 2017
Agenda

- Functional Programming?
- Elements of Functional Programming
- Recursive Functions
- Higher-Order Functions
- Lists
Programming Paradigms

Paradigm: In science, a paradigm describes distinct concepts or thought patterns in some scientific discipline.

Main programming paradigms:
- imperative programming
- functional programming
- logic programming

Orthogonal to it:
- object-oriented programming
History of Functional Programming Languages

1959     Lisp
1975-77  ML, FP, Scheme
1978     Smalltalk
1986     Standard ML
1988     Erlang
1990     Haskell
1999     XSLT
2000     OCaml
2003     Scala, XQuery
2005     F#
2007     Clojure
What is a Functional Program?

- A Functional Program (FP) is an expression (syntax).
- This expression denotes a Value (semantics).
- To run a program has the same meaning as evaluating the expression.

Functions are values, too!
Expressions

An expression is

- a constant, or
- a variable, or
- a term of the form $e_1(e_2)$, with $e_1$ and $e_2$ being expressions. The evaluation of $e_1$ must return a function. (prefix form)

Consequence: A function may be applied to a function or returning a function (= higher-order functions).

Note: All n-ary functions can be transformed into unary functions (= currying).

Scala: All binary functions can be written in infix form.
Variables

- are like in mathematics
- refer to values, not memory locations
- are immutable
- allow only single assignment

Easy to parallelize: no mutable state, no shared memory, and no locks.
Pure FP Languages

Pure functional programming languages accept functional programs only.
They do not have mutable variables, assignments, or imperative control structures.

- Pure Lisp, XSLT, XPath, XQuery, FP
- Haskell (without I/O Monad or UnsafePerformIO)
FP Languages in a Wider Sense

In a wider sense a functional programming language enables the construction of elegant programs that focus on functions.

- Lisp, Scheme, Racket, Clojure
- SML, Ocaml, F#
- Haskell (full language)
- Scala
- Smalltalk, Ruby (!)
Scala

- Combines functional programming and OO
- By Martin Odersky, EPFL
- Compiles to JVM
- Strongly typed (generic types)
- Type inferencing (minimal set of type declarations, rest inferred)
- Pattern matching, case classes
- Functions as parameters
- Language is extensible (DSL), e.g., Erlang’s actor model
- LinkedIn, Twitter, Novell, Sony, Siemens, etc.
Evaluation of Expressions

Expressions evaluate like in a calculator and left to right:

\[(4 - 1) \times (4 + 5)\]
\[\rightarrow 3 \times (4 + 5)\]
\[\rightarrow 3 \times 9\]
\[\rightarrow 27\]

In FP we can give expressions a name:

```latex
val \ x = 4 + 5
3 \times x
\rightarrow
val \ x = 9
3 \times x
\rightarrow 3 \times 9
\rightarrow 27
```

**Substitution principle:** we replace variable names with their definitions.
Lazy Evaluation of Expressions

Alternatively, using `def`, we may delay the evaluation of the definition until the variable is needed:

```python
def x = 4 + 5
3 * x
→
3 * (4 + 5)
→
3 * 9
→
27
```

This is called **lazy evaluation**.
Functions

We may parameterize our definitions:

```scala
def square(x: Int) = x * x
def sumOfSquares(x: Int, y: Int) = square(x) + square(y)
```

By default, function parameters are evaluated first, from left to right:

```
sumOfSquares(1 + 2, 1 + 3)
→ sumOfSquares(3, 1 + 3)
→ sumOfSquares(3, 4)
→ square(3) + square(4)
→ 3 * 3 + square(4)
→ 9 + square(4)
→ 9 + 4 * 4
→ 9 + 16
→ 25
```

This is called call by value.
Call by Name

We may also delay the evaluation parameters using a special type declaration: 
\[ \text{def sumOfSquares(x: \Rightarrow \text{Int}, y: \text{Int}) = square(x) + square(y)} \]

\[ \text{sumOfSquares(1 + 2, 1 + 3)} \]

\[ \rightarrow \text{sumOfSquares(1 + 2, 4)} \]
\[ \rightarrow \text{square(1 + 2) + square(4)} \]
\[ \rightarrow \text{square(3) + square(4)} \]
\[ \rightarrow 3 \times 3 + \text{square(4)} \]
\[ \rightarrow 9 + \text{square(4)} \]
\[ \rightarrow 9 + 4 \times 4 \]
\[ \rightarrow 9 + 16 \]
\[ \rightarrow 25 \]

This lazy evaluation of parameters is known as call by name.
Call by Value vs. Call by Name

- Both evaluation strategies reduce an expression to the same value, provided both evaluations terminate.
- If Call by Value evaluation of an expression terminates, then its Call by Name evaluation terminates, too.
- The reverse is not true!
Non-termination Example

Consider a non-terminating (recursive) expression:

```scala
1 def loop : Int = loop
2 def firstByValue(x: Int, y: Int) = x
3 def firstByName(x: Int, y: => Int) = x
```

Evaluating an expression under:

**call by value:**

```
firstByValue(1 + 2, loop)
→
firstByValue(3, loop)
→
firstByValue(3, loop)
→
...
```

**call by name:**

```
firstByName(1 + 2, loop)
→
firstByName(3, loop)
→
3
```

Here, Call by Name is also more efficient!
Boolean Expressions

Reduction rules for Boolean expressions:

\[
\begin{align*}
!true & \rightarrow false \\
!false & \rightarrow true \\
true \&\& b & \rightarrow b \\
false \&\& b & \rightarrow false \\
true \mid\mid b & \rightarrow true \\
false \mid\mid b & \rightarrow b \\
\end{align*}
\]

b is an arbitrary Boolean expression

Note that \&\& and \mid\mid use short-circuit evaluation

Hence, in Boolean expressions lazy evaluation, e.g. Call by Name, may be beneficial.
In FP we use conditional expressions, not conditional statements, to express alternatives.

```scala
def abs(x: Int) = if (x >= 0) x else -x
```

The Scala conditional `if-else` looks like Java’s, but has expression semantics.

Reduction rules for conditional expressions:

- `if (true) e else f` $\rightarrow$ `e`
- `if (false) e else f` $\rightarrow$ `f`

`e`, `f` are arbitrary expressions.
Recursive Functions

In FP we use recursion in order to express iteration:

```scala
def fact(n: Int): Int = 
  if (n == 0) 1 
  else n * fact(n - 1)
```

We can easily explain recursion using the substitution model:

```
fact(3)
→ if (3 == 0) 1 else 3 * fact(3 - 1)
→ if (false) 1 else 3 * fact(3 - 1)
→ 3 * fact(3 - 1)
→ 3 * fact(2)
→ 3 * if (2 == 0) 1 else 2 * fact(2 - 1)
→ 3 * if (false) 1 else 2 * fact(2 - 1)
→ 3 * 2 * fact(2 - 1)
→ 3 * 2 * fact(1)
```
Recursive Functions (cont.)

\[ 3 \times 2 \times \text{fact}(1) \]
\[ 3 \times 2 \times \text{if (} 1 \text{ == } 0 \text{) } 1 \text{ else } 1 \times \text{fact}(1 - 1) \]
\[ 3 \times 2 \times \text{if (false) } 1 \text{ else } 1 \times \text{fact}(1 - 1) \]
\[ 3 \times 2 \times 1 \times \text{fact}(1 - 1) \]
\[ 3 \times 2 \times 1 \times \text{fact}(0) \]
\[ 3 \times 2 \times 1 \times \text{if (} 0 \text{ == } 0 \text{) } 1 \text{ else } 0 \times \text{fact}(0 - 1) \]
\[ 3 \times 2 \times 1 \times \text{if (true) } 1 \text{ else } 0 \times \text{fact}(0 - 1) \]
\[ 3 \times 2 \times 1 \times 1 \]
\[ 6 \]

Note the redundant last call!

The substitution principle is possible, because we do not have side-effects (mutable data).
Pattern Expressions

Most FP languages support pattern matching:

```scala
1  def fact1(n: Int): Int = n match {
2     case 0 => 1              /* constant pattern */
3     case _ => n * fact1(n - 1)  /* don't care pattern */
4  }
```

Avoiding the last call:

```scala
1  def fact2(n: Int): Int = n match {
2     case 0 => 1
3     case 1 => 1
4     case _ => n * fact2(n - 1)
5  }
```
A defensive function using guards:

```scala
def fact3(n: Int): Int = n match {
  case _ if n < 0 => throw new Error("wrong argument!")
  case 0 => 1
  case 1 => 1
  case _ => n * fact3(n - 1)
}
```

In Scala, we may throw any Java exception.
A MatchError exception is thrown if no pattern matches the value of the selector.

Hence, we could also guard against non-termination, like this:

```scala
def fact4(n: Int): Int = n match {
  case 0 => 1
  case 1 => 1
  case _ if n > 1 => n * fact4(n - 1)
}
```

We will see more forms of patterns later!
Evaluating Match Expressions

An expression of the form

\[ e \text{ match}\{\text{case } p_1 \rightarrow e_1 \ldots \text{ case } p_n \rightarrow e_n}\]  

matches the value of the selector \( e \) with the patterns \( p_1, \ldots, p_n \) in the order in which they are written.

The whole match expression is rewritten to the right-hand side of the first case where the pattern matches the selector \( e \).
Evaluating Match Expressions: Example

```scala
def fact2(n: Int): Int = n match {
    case 0 => 1
    case _ => n * fact2(n - 1)
}

fact2(1)

→ 1 match {case 0 => 1 case _ => 1 * fact2(1 - 1)}
→ 1 * fact2(1 - 1)
→ 1 * fact2(0)
→ 1 * 0 match {case 0 => 1 case _ => 0 * fact2(0 - 1)}
→ 1 * 1
→ 1
```
Forms of Recursion

- **Linear recursion**: one recursive call per branch
- **Tail recursion**: one recursive call per branch being the outermost operation
- **Nested recursion**: recursive calls include recursive calls as arguments
- **Tree-like recursion**: more than one recursive call per branch (also known as cascade recursion)
- **Mutual recursion**: two or more functions call each other
Linear Recursion Example

Functions $\text{fact} - \text{fact4}$ are linear.

Calculating $n^k$:

```scala
def power(n: Int, k: Int): Int = k match {
  case 0 => 1
  case _ if k > 0 => n * power(n, k-1)
}
```

Note: in branch $k > 0$ the operator $*$ is outer-most operation, not $\text{power}$!
Greatest common divisor:

```scala
def gcd(n: Int, m: Int): Int = n match {
  case 0 => m
  case _ if m >= n => gcd(m - n, n)
  case _ => gcd(n - m, m)
}
```
Nested Recursion Example

McCarthy 91 function:

```java
1  def fun91(n: Int): Int =
2      if (n > 100) n - 10
3      else fun91(fun91(n + 11))
```

Quiz: Why is it called 91 function?
Tree-like Recursion Example

Binomial coefficients defined recursively:

\[
\binom{n}{0} = \binom{n}{n} = 1 \quad \text{if } n \geq 0
\]

\[
\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{if } 1 \leq k \leq n - 1
\]

In Scala:

```scala
1 def binom(n: Int, k: Int): Int =
2     if (k==0 || n==k) 1
3     else binom(n-1,k-1) + binom(n-1,k)
```
Mutual Recursion Example

```scala
def isEven(n: Int): Boolean = n match {
  case 0 => true
  case _ if n > 0 => isOdd(n-1)
}

def isOdd(n: Int): Boolean = n match {
  case 0 => false
  case _ if n > 0 => isEven(n-1)
}
```
Higher-Order Functions

... are functions that receive functions as input or return a function as output.

Any FP language treats functions as first-class values. Therefore, functions can be passed as parameters or being returned.

We know them from mathematics!

For example, differentiation takes a function \( f(x) = x^2 \) and returns another function \( f'(x) = x \).

In FP they provide a flexible way to

- compose programs (combine several functions into a new one)
- to modify programs (take a function and produce another one)

In ordinary programming languages these combinators are syntactically fixed.
Example

Take the sum of integers between a and b:

```scala
1  def sumInts(a: Int, b: Int): Int =
2     if (a > b) 0
3     else a + sumInts(a + 1, b)
```

Take the sum of the cubes of all the integers between a and b:

```scala
1  def cube(x: Int): Int = x * x * x
2
3  def sumCubes(a: Int, b: Int): Int =
4     if (a > b) 0
5     else cube(a) + sumCubes(a + 1, b)
```
Example (cont.)

Take the sum of the factorials of all the integers between a and b:

```scala

def sumFactorials(a: Int, b: Int): Int =
  if (a > b) 0
  else fact(a) + sumFactorials(a + 1, b)
```

These are special cases of

$$\sum_{i=a}^{b} f(i)$$

for different values of $f$.

Can we factor out the common pattern?
A Higher-Order Sum Function

```scala
1 def sum(f: Int => Int, a: Int, b: Int): Int =
2   if (a > b) 0
3   else f(a) + sum(f, a + 1, b)
```

Now, we can specialize this function into

```scala
1 def sumInts(a: Int, b: Int) = sum(id, a, b)
2 def sumCubes(a: Int, b: Int) = sum(cube, a, b)
3 def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

where

```scala
1 def id(x: Int): Int = x
```

In Scala, we write function types `TypeA => TypeB`. 
Anonymous Functions

If functions are only used as parameters, we often do not want to define them, like e.g., `id` and `cube`.

An anonymous function definition returns a function without the need to define it with a name.

Example: an anonymous cube function:

```plaintext
(x: Int) => x * x * x
```

Now, we can specialize the two sum functions

```plaintext
1 def sumInts(a: Int, b: Int) = sum((x: Int) => x, a, b)
2 def sumCubes(a: Int, b: Int) = sum((x: Int) => x * x * x, a, b)
```

without auxiliary function definitions.
Function Composition

An important higher-order function is function composition \( f \circ g \).

\[
(f \circ g)(x) = f(g(x))
\]

In Scala, this operator is predefined and called `compose`.

Example: combining two anonymous squaring functions

```scala
1  def sumPowerOfFour(a: Int, b: Int) =
2     sum(((x: Int) => x * x) compose ((x: Int) => x * x), a, b)
```
Lists

Lists are an important inductive data structure in functional programming.

In contrast to arrays, lists are immutable (i.e., elements of a list cannot be changed).

Like arrays, lists are homogenous: the elements of a list must all have the same type.

Examples:

```scala
val fruits = List("apples", "oranges", "pears")
val nums = List(1, 2, 3)
val diagM = List(List(1,0,0), List(0,1,0), List(0,0,1))
val empty = List()
```
Lists are inductively defined. All lists are constructed from

- the empty list \texttt{Nil}, and

- the construction operation :: (named cons):
  \hspace{1cm} \texttt{x::xs} defines a new list with the first element (called the head) \texttt{x}, followed by the elements of \texttt{xs} (called the tail).

**Examples:**

1. \hspace{1cm} \texttt{val fruitsCons = "apples"::("oranges"::("pears"::Nil))}
2. \hspace{1cm} \texttt{val numsCons = 1::(2::(3::Nil))}
3. \hspace{1cm} \texttt{val emptyCons = Nil}

Scala convention: operators ending in : associate to the right. Hence, we can omit the parantheses:

1. \hspace{1cm} \texttt{val fruitsCons = "apples":"oranges":"pears":Nil}
2. \hspace{1cm} \texttt{val numsCons = 1::2::3::Nil}
Essential List Operations

All functions on lists can be expressed in terms of three operations:

- **head**: first element of list (exception if list is empty)
- **tail**: a list without its head
- **isEmpty**

In Scala they are defined as methods of objects of type list:

```
1 empty.isEmpty == true
2 fruits.head == "apples"
3 fruits.tail == List("oranges", "pears")
```

Alternatively, list patterns can be used to decompose a list!
List Patterns

- **Nil**: matches empty list
- **x::xs**: matches non-empty list and binds identifiers \( x \) and \( xs \) to head and tail.
- **List(e_1, \ldots, e_n)**: same as \( e_1::\ldots::e_n::\text{Nil} \)

Also more complex patterns can be constructed:

- **1::2::xs**: a list starting with 1 and 2
- **x::Nil**: a list of length one
- **List(x)::xs**: a list of lists starting with a list of length one
def member(e: Int, l: List[Int]) : Boolean = l match {
  case Nil => false
  case `e`::xs => true
  case _::xs => member(e,xs)
}

Interesting patterns:

- Stable identifier pattern `e` ensures that first element matches parameter `e`.
- Note the use of the don’t care pattern, since no variable binding to first element is needed!
Appending Two Lists

```scala
def append(as: List[Int], bs: List[Int]): List[Int] = as match {
  case Nil => bs
  case x::xs => x::append(xs, bs)
}
```

It is not very useful to restrict this function to a certain type!
Generic Append Function

We may use a type variable T to generalize the function:

```scala
def appendG[T](as: List[T], bs: List[T]): List[T] = as match {
  case Nil => bs
  case x::xs => x::appendG(xs, bs)
}
```
Length of a List

def length[T](as: List[T]): Int = as match {
  case Nil         => 0
  case _::xs       => 1 + length(xs)
}

B.K. Aichernig

Classical Themes of Computer Science

45 / 49
Insertion Sort

```scala
def isort(as: List[Int]): List[Int] = as match {
  case Nil    => Nil
  case x::xs  => insert(x, isort(xs))
}

def insert(e: Int, as: List[Int]): List[Int] = as match {
  case Nil    => List(e)
  case x::_  if (e <= x) => e::as
  case x::xs  => x::insert(e,xs)
}
```

Can we write a generic insertion sort?
Generic Insertion Sort

We pass the order relation as an additional parameter:

```python
def isortG[T](leq: (T,T) => Boolean, as: List[T]): List[T] = 
as match {
  case Nil => Nil
  case x::xs => insertG(leq ,x, isortG(leq,xs))
}

def insertG[T](leq:(T,T) => Boolean, e: T, as: List[T]): List[T] = 
as match {
  case Nil => List(e)
  case x::_ if leq(e,x) => e::as
  case x::xs => x::insertG(leq,e,xs)
}
```

```text
isortG((x: Int, y: Int) => x <= y, List(3,1,4,2)) returns List(1, 2, 3, 4)
isortG((x: Int, y: Int) => x > y, List(3,1,4,2)) returns List(4, 3, 2, 1)
```
Generic Insertion Sort with Local Function

We may hide the insert function in a block expression:

```python
def isortB[T](leq: (T,T) => Boolean, as: List[T]): List[T] = {
    def insert(e: T, as: List[T]): List[T] = as match {
        case Nil => List(e)
        case x:: _ if leq(e, x) => e::as
        case x::xs => x::insert(e, xs)
    }
    as match {
        case Nil => Nil
        case x::xs => insert(x, isortB(leq, xs))
    }
}
```

Here, the order relation `leq` is known inside the block.
Simplified Generic Insertion Sort

We can even hide all recursive definitions inside the block:

```python
def isortC[T](leq: (T,T) => Boolean, as: List[T]): List[T] = {
  def insert(e: T, as: List[T]): List[T] = as match {
    case Nil => List(e)
    case x::_ if leq(e,x) => e::as
    case x::xs => x::insert(e,xs)
  }
  def isort(as: List[T]): List[T] = as match {
    case Nil => Nil
    case x::xs => insert(x, isort(xs))
  }
  isort(as)
}
```

Now, both inner recursive functions do not need \( leq \) as a parameter.