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

Bernhard K. Aichernig
Institute of Software Technology
Graz University of Technology
Austria

Winter Term 2017/18
Version: November 30, 2017
Agenda

Part 1:
- Functional Programming?
- Elements of Functional Programming
- Recursive Functions
- Higher-Order Functions
- Lists

Part 2:
- Inductive Proofs
- Curried Functions
- Higher-Order List Functions
- Concrete Data-types: Immutable Classes
- Actors
Q: How can we prove properties of list programs?
A: Structural induction!

Proof rule for proving a list property \( P(xs) \) via structural induction:

\[
\begin{align*}
P(\text{Nil}) & \quad \text{(base case)} \\
\forall x, xs : P(xs) \Rightarrow P(x :: xs) & \quad \text{(induction step)} \\
\forall xs : P(xs) & \quad \text{(consequence)}
\end{align*}
\]

\( P(xs) \) in induction step is called **induction hypothesis**
Theorem 1

Let's prove that

\[ P : \text{length}(\text{append}(as, bs)) = \text{length}(as) + \text{length}(bs) \]

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

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

1. **Base case:** we substitute Nil for \( as \) in \( P \):

\[
\text{length}(\text{append}(\text{Nil}, bs)) = \text{length}(bs) = 0 + \text{length}(bs) = \text{length}(\text{Nil}) + \text{length}(bs)
\]

[def. append l.r.] [arithmetic] [def. length r.l.]
Theorem 1 (cont.)

2. Induction step: we extend as with $a :: as$ in $P$:

\[
\begin{align*}
\text{length}(\text{append}(a :: as, bs)) & = \text{length}(a :: \text{append}(as, bs)) & \text{[def. append l.r.]} \\
& = 1 + \text{length}(\text{append}(as, bs)) & \text{[def. length l.r.]} \\
& = 1 + \text{length}(as) + \text{length}(bs) & \text{[induction hypothesis $P$ l.r.]} \\
& = \text{length}(a :: as) + \text{length}(bs) & \text{[def. length r.l.]} \\
\end{align*}
\]

q.e.d.
Theorem 2

Given a tail-recursive length function:

```scala
def len[T](as: List[T], n: Int): Int = as match {
  case Nil => n
  case _::xs => len(xs, n+1)
}
```

Let's prove that both length functions are equivalent:

\[ P : \text{len}(xs, 0) = \text{length}(xs) \]

1. **Base case:** we substitute \( \text{Nil} \) for \( xs \) in \( P \):

\[
\text{len}(\text{Nil}, 0) \\
= 0 \\
= \text{length}(\text{Nil})
\]

[def. len l.r.]  
[def. length r.l.]
2. Induction step: we extend \( xs \) with \( x :: xs \) in \( P \):

\[
\begin{align*}
\text{len}(x :: xs, 0) &= \text{len}(xs, 0 + 1) \\
&= \text{len}(xs, 1) \\
&= 1 + \text{length}(xs) \\
&= \text{length}(x :: xs)
\end{align*}
\]

We need to prove the more general

Lemma 1: \( \text{len}(xs, n) = n + \text{length}(xs) \)
Theorem 2, Lemma 1

\[ P : \text{len}(xs, n) = n + \text{length}(xs) \]

1. Base case: we substitute \textit{Nil} for \( xs \) in \( P \):

\[
\text{len}(\text{Nil}, n) \\
= n \\
= n + 0 \\
= n + \text{length}(\text{Nil})
\]

[def. \text{len} l.r.]  
[arithmetic]  
[def. \text{length} r.l.]

2. Induction step: we extend \( xs \) with \( x :: xs \) in \( P \):

\[
\text{len}(x :: xs, n) \\
= \text{len}(xs, n + 1) \\
= n + 1 + \text{length}(xs) \\
= n + \text{length}(x :: xs)
\]

[def. \text{len} l.r.]  
[induction hypothesis l.r.]  
[def. \text{length} r.l.]

q.e.d.
Curried Functions

Currying
- is a method to transform n-ary functions into unary functions.
- enables the partial binding of function parameters.

The concept goes back to the logician Haskell B. Curry (1900–1982).
Addition as a Unary Function

Consider the addition of two integers:

1
```scala
def add(x: Int, y: Int) = x + y
```

We can rewrite it into a unary higher-order function returning a function that takes the second argument and adds it to the first argument:

1
```scala
def cadd(x: Int): Int => Int = (y: Int) => x+y
```

The function `cadd` returns an anonymous function.

We just curried the addition function.

Now, we can partially apply the parameters, e.g. defining a successor function:

1
```scala
val succ = cadd(1)
```
What is Going On?

1  ```scala
def cadd(x: Int): Int => Int = (y: Int) => x+y
```

Well, let’s rewrite the expression

cadd(1)(2)

→ ((y: Int) => 1+y)(2)

→ 1 + 2

→ 3

Note, function application associates to the left!
This is so useful that FP languages provide special syntax to define curried functions.

For example, in Scala

```scala
1 def caddScala(x: Int)(y: Int) = x+y
```

is semantically equivalent to

```scala
1 def cadd(x: Int): Int => Int = (y: Int) => x+y
```

However, the syntax for step-wise parameter instantiation is different:

```scala
1 val succ = caddScala(1) _
```
def isort[T](leq: (T, T) => Boolean)(as: 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)
}
Curried Insertion Sort (cont.)

Now, we may define specialised sorting functions:

```
1 val sortIncr = isort((x: Int, y: Int) => x < y) _
2 val sortDecr = isort((x: Int, y: Int) => x > y) _
```

We may even pass trivial relations:

```
1 val identity = isort((x: Any, y: Any) => true) _
2 val reverse = isort((x: Any, y: Any) => false) _
```

Or discriminate against certain professors :)

```
1 val sortAichernigLast =
2   isort((x: String, y: String) => ((x < y && x != "Aichernig")
3          || y == "Aichernig") ) _
```
Let’s define a useful auxiliary function that generates a list of natural numbers up to a constant $n$:

```scala
def intsto(n: Int): List[Int] = n match {
  case 0 => Nil
  case n => intsto(n-1) ++ List(n)
}
```

The operator `++` is Scala’s built in list concatenation (append).
Furthermore, we define two mutual recursive functions taking every second element:

```scala
1 def take[T](as: List[T]) : List[T] = as match {
2   case Nil => Nil
3   case x::xs => x::skip(xs)
4 }
5 def skip[T](as: List[T]) : List[T] = as match {
6   case Nil => Nil
7   case x::xs => take(xs)
8 }
```
Composition of List Functions (cont.)

Now, we may generate odd (even) numbers by functional composition:

```
1 val odds = take[Int] _ compose intsto _
2 val evens = skip[Int] _ compose intsto _
```

E.g. `odds(10) ↳ List(1, 3, 5, 7, 9)`

- We can interpret functional composition as dataflow:

  \[
  \text{output} \leftarrow \text{take} \leftarrow \text{intsto} \leftarrow \text{input}
  \]

- This architectural style is known as pipe and filter (see Unix pipes).
- In FP it is also known as variable-free or point-free programming.
We often apply a function $f$ to all elements of list.

The higher-order function `map` does this and is predefined in Scala:

```scala
List(1,2,3,4).map(x => x * -1) \rightarrow List(-1,-2,-3,-4)
```

We can easily define `map` as follows:

```scala
1 def map[A,B](f: A => B)(as: List[A]): List[B] = as match {
2   case Nil    => Nil
3   case x::xs => f(x)::map(f)(xs)
4 }
```

E.g. `map((x: Int) => x * -1)(intsto(5))`

or `map[Int,Int](x => x * -1)(intsto(5))`

or `map[Int,Int](_ * -1)(intsto(5)) \rightarrow List(-1,-2,-3,-4,-5)`
map in Scala

In Scala, the map function is actually a method of the (immutable) class List. Therefore, the dot notation.

In Scala, we can also write methods of arity 2 in infix notation:

```scala
def scale(factor: Double)(as: List[Int]) = 
  as map (x => x * factor)

val discount = scale(0.9) _
```

E.g. 

```
(discount compose intsto)(4) ⇝ List(0.9, 1.8, 2.7, 3.6)
```

Note that map fixes a recurring recursion pattern. This pattern can then be applied without the use of explicit recursion.
Filtering elements of a list according to some condition is a further recurring pattern.

For example, the evens function can be easily defined with filter:

```scala
1 def evensF(n: Int) = intsto(n) filter (x => (x & 1) == 0)
```

E.g. `evensF(8) ⇝ List(2, 4, 6, 8)

Filter can be defined as follows:

```scala
1 def filter[T](p: T => Boolean)(as: List[T]): List[T] = as match {
2   case Nil => Nil
3   case x::xs if p(x) => x::filter(p)(xs)
4   case x::xs => filter(p)(xs)
5 }
```
## Different Filters in Scala

Scala knows different filter methods extracting sublists:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs filterNot p</code></td>
<td>Filters elements in <code>xs</code> not satisfying <code>p</code>.</td>
</tr>
<tr>
<td><code>xs partition p</code></td>
<td>Same as <code>(xs filter p, xs filterNot p)</code>, but computed in a single traversal.</td>
</tr>
<tr>
<td><code>xs takeWhile p</code></td>
<td>The longest prefix of <code>xs</code> with elements satisfying <code>p</code>.</td>
</tr>
<tr>
<td><code>xs dropWhile p</code></td>
<td>The postfix of <code>xs</code> after removing all leading elements satisfying <code>p</code>.</td>
</tr>
<tr>
<td><code>xs span p</code></td>
<td>Same as <code>(xs takeWhile p, xs dropWhile p)</code>, but computed in a single traversal.</td>
</tr>
</tbody>
</table>
Quicksort

```scala
def qsort(as: List[Int]): List[Int] = as match {
  case Nil => Nil
  case x::xs =>
    val (smaller, greater) = xs partition (e => e < x)
    qsort(smaller) ++ List(x) ++ qsort(greater)
}
```

Note the elegant use of a pair-pattern! (all Scala patterns can be used in value definitions)

Note the tree-like recursion of quicksort.
Generic Quicksort

```
1 def gqsort[T](less: (T,T) => Boolean)(as: List[T]) = {
2     def qsort(as: List[T]): List[T] = as match {
3         case Nil => Nil
4         case x::xs =>
5             val (smaller,greater) = xs partition (e => less(e, x))
6             qsort(smaller) ++ List(x) ++ qsort(greater)
7         }
8     qsort(as)
9 }
10
11 val qsortInt = gqsort[Int]((a, b) => a < b) _
12
E.g. qsortInt(List(3,2,1,3,4)) ~> List(1, 2, 3, 3, 4)
```
Another common operation on lists is to combine the elements of a list using a given operator. The function `foldRight` takes the following arguments as input:

1. a list
2. an initial value
3. a 2-ary function combining the accumulated value with the next list element.

```
foldRight(List(e1,e2,...,en))(0)(_ + _) ⇝
```

```
1 + (e2 + ... (en + 0)...))
```

```scala
def foldRight[A,B](as: List[A])(e: B)(f: (A,B) => B) : B =
  as match {
    case Nil  => e
    case x::xs => f(x, foldRight(xs)(e)(f))
  }
```
One-Liners with foldRight

The sum of the first n integers:

```scala
def sum(n: Int) = foldRight(intsto(n))(0)(_ + _)
```

The factorial without explicit recursion:

```scala
def factorial(n: Int) = foldRight(intsto(n))(1)(_ * _)
```

Appending two lists (concatination):

```scala
def append[T](as: List[T], bs: List[T]) = foldRight(as)(bs)(_ :: _)
```

Note, the built-in Scala methods are called infix, e.g.

```
(List(1,2,3) foldRight 0)(_ + _)
```

or like a method:

```
List(1,2,3).foldRight(0)(_ + _)
```
foldLeft

There is also a fold that starts with the left-most element in the list:

\[
\text{foldLeft}((e_1, e_2, \ldots, e_n))(0)(\_ + \_ ) \mapsto \\
(\text{xn} + (\ldots + (x_2 + (x_1 + 0))\ldots) + \text{xn})
\]

```scala
1 def foldLeft[A,B](as: List[A])(e: B)(f: (A,B) => B) : B =
2   as match {
3     case Nil => e
4     case x::xs => foldLeft(xs)(f(x,e))(f)
5   }
```
foldLeft vs. foldRight

For operators that are associative and commutative both functions are equivalent.

However, sometimes only one version is appropriate, e.g.: with the constructor operator we may reverse a list elegantly:

```scala
1 def rev[T](as: List[T]) = foldLeft[T,List[T]](as)(Nil)(_::_)
```

Here, the initial value is the empty list.
Sum Function

```
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)
```

In the pipe and filter style with built-in functions:

```
1  def sum2(f: Int => Int, a: Int, b: Int): Int =
2      intsto(b).filter((x : Int) => x >= a).map(f).foldRight(0)(+_+)
```

The above is not very efficient! Better:

```
1  def sum3(f: Int => Int, a: Int, b: Int): Int =
2      intsto(b).foldRight(0)((x: Int,y: Int) => if (x < a) y
3          else f(x) + y)
```
We organise our functions in classes:

```scala
class Rational(x: Int, y: Int) {
  def numer = x
  def denom = y
}
```

Objects are created like in Java, e.g.,

```scala
val a = new Rational(1,2)
```

We may access the members with `a.numer` and `a.denom`
We may implement rational number arithmetic via functions:

```scala
def addRational(a: Rational, b: Rational) : Rational =
  new Rational(a.numer * b.denom + b.numer * a.denom,
               a.denom * b.denom)

def toString(r: Rational) : String =
  r.numer + "/" + r.denom
```
Methods

We may go further and package the functions operating on a data type in the data type.

Such functions are called **methods**.

For example, we may package all functions on rationals as methods in class Rationals:

```scala
class Rational(x: Int, y: Int) {
  def numer = x
  def denom = y

  def add(b: Rational): Rational =
    new Rational(numer * b.denom + b.numer * denom,
                 denom * b.denom)

  override def toString: String =
    numer + "/" + denom
}
```
Operator Overloading

... is possible in Scala

```scala
class Rational(x: Int, y: Int) {
  def numer = x
  def denom = y

  def +(b: Rational): Rational =
    new Rational(numer * b.denom + b.numer * denom,
                 denom * b.denom)

  override def toString: String =
    numer + "/" + denom
}
```

Any method and operator with one parameter can be used infix.
Actually ...

Everything, in Scala is an object.

Even functions are objects!

Functions are objects with an apply method, e.g.:

```scala
object square{
  def apply(x: Int): Int = x * x
}
```

E.g., both `square.apply(2)` and `square(2) ⇝ 4`

Hence, \( f(x) \) is just syntactic sugar for \( f.apply(x) \).
Case Classes

Case classes represent terms.

Case objects can be constructed without the keyword new.

Case objects are accessed via pattern matching.

Defining an abstract syntax tree for Boolean expressions:

```scala
abstract class BExp

case object True extends BExp
case object False extends BExp

case class Variable(id: String) extends BExp
case class Not(b: BExp) extends BExp
case class Or(left: BExp, right: BExp) extends BExp
```
Now we can write an elegant simplification function for Boolean expressions:

```scala
def simplify(b: BExp): BExp = b match {
  case Or(True,r) => True
  case Or(l,True) => True
  case Or(False,r) => simplify(r)
  case Or(l,False) => simplify(l)
  case Not(Not(e)) => simplify(e)
  case _ => b
}
```

E.g., `simplify(Not(Not(Or(Variable("a"),False))))` ⇝ `Variable("a")`
The actor model of concurrency [Hewitt, Bishop, Steiger 1973] got fame in the FP language Erlang (e.g. WhatsApp).

Actors are concurrent processes that share no state.

Actors have a unique identity.

Actors communicate via asynchronous messages via identity.

Each actor has a mailbox serving as a message buffer.

Each actor runs a pattern-matching over the messages in the mailbox.

If a message matches a certain pattern, then the message is removed from the mailbox and the actor reacts accordingly.

Scala provides Actors in the Akka library (included since version 2.11).
Why Actors?

- Simple and high-level abstractions for concurrency and parallelism.
- Asynchronous, non-blocking and highly performant event-driven programming model.
- Very lightweight event-driven processes (approximately 2.7 million actors per GB RAM).
- Actors also work in a distributed environment.
- Actor systems are fault tolerant through supervisor hierarchies with “let-it-crash” semantics.

Forget threads! (like you forgot assembler)
import akka.actor._

class Greeter extends Actor {

  def receive = {
    case "hello" => println("hello\back\at\you")
    case "bye"   => println("bye,\bye,\see\you!")
       context.stop(self)
    case _       => println("huh?")
  }
}

The method receive implements a server loop, listening and reacting to messages.
object Main extends App {
  
  val system = ActorSystem("Hello") /* create actor system */
  val greeter = system.actorOf(Props[Greeter]) /* add actor */
  greeter ! "hello"
  greeter ! "the meaning of life is 42"
  greeter ! "bye"
}

a ! m sends a message m to an Actor a, asynchronously (no waiting).
Defining a Protocol

With case classes/objects, we can define the types of messages:

```scala
abstract class Message

case object Hello extends Message

case object Bye extends Message

case class Say(words: String) extends Message
```
Reacting to Typed Messages

class GreeterP extends Actor {

def receive = {
  case Hello => println("hello\back\at\you")
  case Say(w) => println(w)
  case Bye => println("bye,\bye,\see\you!")
    context.stop(self)
  case _ => println("huh?")
}
}

Note the pattern matching of Say(w) binding w to the sent string.
Playing Ping Pong

Two actors talking to each other

```scala
class Ping extends Actor {
  def receive = {
    case Play(pong) => println("Ping\ started")
    pong ! PingM
    case PongM => println("Ping")
    sender ! PingM
    case Stop => println("Ping\ stops!")
    context.stop(self)
  }
}
class Pong extends Actor {
  def receive = {
    case PingM => println("Pong")
    sender ! PongM
    case Stop => println("Pong\ stops!")
    context.stop(self)
  }
}
```
**Starting Ping Pong**

Actor ping starts playing the game knowing pong:

```scala
object Main extends App {
  val ping = system.actorOf(Props[Ping])
  val pong = system.actorOf(Props[Pong])
  ping ! Play(pong)
  Thread.sleep(10) // ok, Main is still a thread :)
  ping ! Stop
  pong ! Stop
}
```

The message definitions:

```scala
case class Play(to: ActorRef) extends Message
case object PingM extends Message
case object PongM extends Message
case object Stop extends Message
```
Beyond Ping Pong

There is much more:

- Full Akka framework (also for Java): http://akka.io
- Scala’s parallel collections
Summary

▶ Basics:
  ▶ Functional Programming?
  ▶ Elements of Functional Programming
  ▶ Recursive Functions
  ▶ Higher-Order Functions
  ▶ Lists

▶ Advanced:
  ▶ Inductive Proofs
  ▶ Curried Functions
  ▶ Higher-Order List Functions
  ▶ Concrete Data-types: Immutable Classes
  ▶ Actors