2 — Functional Data Structures

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2023

Lehrstuhl für Informatik VI, Uni Würzburg

Hands-on Lecture

Starting with this lecture, we will have small programming exercises in between explanations. We think, programming things yourself helps understanding. Also, talking about problems

Slides with exercises will have the marker in the lower left corner.

Overview

Functional data structures are operated on using only pure functions. We've already had a short look at functional lists with `head` and `tail` functions last week.

Recap: pure functions are functions, that do not change data in place or perform other side effects.

This means functional data structures must be immutable, e.g. appending to a functional list returns a new list.

Digression: Variance

Digression: Variance

You may have noticed in statically typed languages, that some subtype relations you would expect are not given by the type system, when parameterized types are involved.

For example, if you have a class `Animal` with subclasses `Cat` and `Dog`, one could expect, that a `List<Cat>` can be used when a `List<Animal>` is expected.

We'll now see, why this is generally not the case (and why functional data structures allow it).

Digression: Variance

Variance describes the subtyping relationship for parameterized types.

There are four types of variance. In Scala, variance is declared in the type definition.

Variance	Syntax	`Foo[X] <: Foo[Y]` when
invariant	`class Foo[A]`	never
covariant	`class Foo[+A]`	`X <: Y`
contravariant	`class Foo[-A]`	`Y <: X`
bivariant	not supported	always

Digression: Variance — implications

Variance places restriction on where the type may be used:

- A covariant type parameter may not be used in a method argument
- A contravariant type parameter may not be used as a return type

We'll see some examples that explain these restrictions, see also the variance section of the online Scala Book.

Digression: Variance — Covariance Counterexample

```
trait MutableList[+A]:
  def add(a: A): Boolean

val anyList: MutableList[Any] = MutListImpl[Int](1,2,3)
anyList.add("not an int") // A = Any, String <: Any, so add should take strings
```

A covariant type in method parameter would cause the implementation to expect only a subtype of what we are allowed to pass in. This trait definition will not compile.

Digression: Variance — Contravariance Counterexample

```
trait ContraList[-A]:
  def get(index: Int): A

val stringList: ContraList[String] =
  ContraListImpl[Object](new Object, new Object)

stringList.get(0) // not a string
```

A contravariant type as return type would cause supertypes of our expected type to be returned. This trait definition will not compile.

In Java, variance is declared at use site instead of declaration site: `List<? extends Foo> fooList` declares a covariant list that may be assigned a `List<T>` with any `T` that is a subtype of `Foo` (contravariance is declared with `super`).

The same rules apply regarding usage of the type. Java will for example prevent calling the `add` method on `fooList`.

Immutable Lists

Defining singly linked lists

A basic data structure for sequences of elements is the singly linked list:

```
// `List` data type, parameterized on a type `A`
enum List[+A]:
  // Nil represents the empty list
  case Nil
  // Cons represents nonempty lists
  case Cons(_head: A, _tail: List[A])
```

Remember: fields of case classes are immutable by default. This way, our list definition can be covariant.

`List` is an algebraic data type (ADT), a type that is a combination of other types. In this case, it is a combination of `Cons` and `Nil`, both are considered subtypes of `List[A]`.

Defining singly linked lists

```
enum List[+A]:
  case Nil
  case Cons(_head: A, _tail: List[A])
```

With this, we can create lists of elements by nesting `Cons` objects:

empty list of doubles:	`val ex1: List[Double] = Nil`
list with an int	`val ex2: List[Int] = Cons(1, Nil)`
list with two strings	`val ex3 = Cons("a", Cons("b", Nil))`

Note: the slides assume an ***`import List.*`*** *for all code not inside the enum or its companion object, otherwise we'd need to write* ***`List.Cons`*** *and* ***`List.Nil`***

Linked List — Companion Object

```
object List:
  def sum(ints: List[Int]): Int = ints match
    case Nil => 0
    case Cons(hd, tl) => hd + sum(tl)

  def product(ds: List[Double]): Double = ds match
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(hd, tl) => hd * product(tl)

  def apply[A](as: A*): List[A] = // Variadic function syntax
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail*))
```

- companion object to our `List` algebraic data type

Linked List — Companion Object

```
object List:
  def sum(ints: List[Int]): Int = ints match
    case Nil => 0
    case Cons(hd, tl) => hd + sum(tl)

  def product(ds: List[Double]): Double = ds match
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(hd, tl) => hd * product(tl)

  def apply[A](as: A*): List[A] = // Variadic function syntax
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail*))
```

- calculate sum of a list of ints

Linked List — Companion Object

```
object List:
  def sum(ints: List[Int]): Int = ints match
    case Nil => 0
    case Cons(hd, tl) => hd + sum(tl)

  def product(ds: List[Double]): Double = ds match
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(hd, tl) => hd * product(tl)

  def apply[A](as: A*): List[A] = // Variadic function syntax
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail*))
```

- calculate product of a list of doubles

Linked List — Companion Object

```scala
object List:
  def sum(ints: List[Int]): Int = ints match
    case Nil => 0
    case Cons(hd, tl) => hd + sum(tl)

  def product(ds: List[Double]): Double = ds match
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(hd, tl) => hd * product(tl)

  def apply[A](as: A*): List[A] = // Variadic function syntax
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail*))
```

- allows us to initialize a list like this: `List(1,2,3)`. Variadic functions can take any number of parameters and return them as a `scala.collection.Seq`. Details about this syntax is out of scope for the lecture.

Pattern Matching — Exercise

What is the result of following match-expression?

```
import List.*
val x = List(1,2,3,4,5) match
  case Cons(x, Cons(2, Cons(4, _))) => x
  case Nil => 42
  case Cons(x, Cons(y, Cons(3, Cons(4, _)))) => x + y
  case Cons(hd, tl) => hd + sum(tl)
  case _ => 101
```

Pattern Matching — Exercise

What is the result of following match-expression?

```
import List.*
val x = List(1,2,3,4,5) match
  case Cons(x, Cons(2, Cons(4, _))) => x
  case Nil => 42
  case Cons(x, Cons(y, Cons(3, Cons(4, _)))) => x + y
  case Cons(hd, tl) => hd + sum(tl)
  case _ => 101
```

- `x == 3`, the third case matches and captures first two elements as x and y.
- First case would match list with 2 at second and 4 at third position
- Fourth and fifth case would also match, but come later

Data sharing in functional data structures

Modifying operations on immutable data structures

- Functional lists are immutable. How to add or remove elements?
  $\Rightarrow$ By returning new Lists
- Example: given a list `xs`, prepend `1` to it: `Cons(1, xs)`
  - Existing list is not changed
  - `xs` is immutable $\Rightarrow$ can be safely reused
  - No „pessimistic copying“ required, like seen in typical non-fp programs
- If operations on a data structure keep and reuse the existing structure, we call it persistent

Example: head

Using pattern matching, we can add methods to the `List` enum, so they can be called on any list. For example, let's define a method to get the first element in the list:

```
def head: A = this match
  case Nil => sys.error("head of empty list")
  case Cons(hd, _) => hd
```

- Not ideal, throws an error when the list is empty, which is not referentially transparent.
- But also can't return an A, when we have none
- We'll see functional ways to handle errors next lecture

Exercise: Implement tail

Implement `tail` in the `List` enum, which removes the first element of a list and returns the rest.

```
def tail: List[A] = ???
```

Your template already contains the implementation for `head`. Your solution should be similar.

What possible actions exist when called on an empty list, that does not exist for head?

Exercise: Implement tail — Solution

```
def tail: List[A] = this match
  case Nil => sys.error("tail of empty list")
  case Cons(_, tl) => tl
```

- For tail on `Nil`, we throw an error here
- We could also return `Nil`, but usually tail on `Nil` is a bug. Not throwing an error makes it harder to find.
- A similar solution wouldn't be possible for **head**, because there is no "empty" **A** (except if you allow *`null`*, which we don't).
- Aside from the error handling options in the next lecture, you should be using pattern matching instead of head and tail, if the list could be empty.

Exercise: Implement init

Implement `init`, which returns all but the last element of a list („reverse tail“).

```
def init: List[A] = ???
```

Hint: you may want to use more than two cases.

Why can’t this be implemented in constant time?

Exercise: Implement init — Solution

```
def init: List[A] = this match
  case Nil => sys.error("init of empty list")
  case Cons(_,Nil) => Nil
  case Cons(hd, tl) => Cons(hd, tl.init)
```

- Because of cons structure, copying of all earlier elements is needed when removing an entry
- $\Rightarrow$ removing the last element: $\mathcal{O}(n-1)$
- Also not tail-recursive!

Exercise: Implement init — Tailrecursive Solution

- Tail recursive by accumulating backwards, then reversing
- Implementation of **`reverse`** will be an exercise

```
def initTailrec: List[A] =
  @annotation.tailrec
  def go(cur: List[A], init: List[A]): List[A] = cur match
    case Nil => sys.error("init of empty list")
    case Cons(_,Nil) => init
    case Cons(hd, tl) => go(tl, Cons(hd, init))

  go(this, Nil).reverse
```

Widening the type

Let's look at another method, `setHead`, which replaces the first element of a list:

```
def setHead[AA >: A](head: AA): List[AA] = ???
```

Don't be confused by the types. The signature says "AA is a supertype of A". As our List is covariant in A, we would not be able to have a parameter of type A.

We solve the problem by returning another type of list. The compiler will automatically infer AA, so that our current elements and the new head element are fitting. E.g. setting the head of a `List[Cat]` to a `Dog` will return a `List[Animal]`.

Exercise: Implement setHead

Now implement it, using the same idea as with tail (empty list should throw an error).

```
def setHead[AA >: A](head: AA): List[AA] = ???
```

(the implementation will not have to care about the more complex types in the signature either, the compiler will take care of that)

Exercise: Implement setHead — Solution

```
def setHead[AA >: A](head: AA): List[AA] = this match
  case Nil => sys.error("setHead on empty list")
  case Cons(_, tl) => Cons(head, tl)
```

or, using tail:

```
def setHead[AA >: A](head: AA): List[AA] = Cons(head, this.tail)
```

Dropping elements

We can generalize `tail` to `drop`, which removes the first `n` elements in $\mathcal{O}(n)$.

```
def drop(n: Int): List[A] =
  if n <= 0 then this
  else this match
    case Nil => Nil
    case Cons(_, tl) => tl.drop(n-1)
```

- Usually implemented to *not* throw on `n > l.length`, as number of elements to drop often calculated from some other source (e.g. a step size).

More Data Sharing

Concatenating two lists with length l_1 and l_2 only takes $\mathcal{O}(l_1)$:

```
def append[AA >: A](other: List[AA]): List[AA] =
  this match
    case Nil => other
    case Cons(hd, tl) => Cons(hd, tl.append(other))
```

Doing the same using a non-persistent structure like Arrays takes $\mathcal{O}(l_1 + l_2)$, as both have to be copied. Here we can keep the second list unchanged.

Parametricity

Parametricity

Remember our `stringExists` function from last lecture? We can write a (more sane) version for our list implementation:

```
@annotation.tailrec
def stringExists(strings: List[String], s: String): Boolean =
  strings match
    case List.Nil => false
    case List.Cons(hd, tl) => if(hd == s) true else stringExists(tl, s)
```

But the signature still has the same problem. We can not tell from it, if the function just looks at equality or still converts to lower case.

Parametricity

We can fix this by generalizing our function:

```
@annotation.tailrec
final def exists[AA >: A](a: AA): Boolean = this match
  case Nil => false
  case Cons(hd, tl) => hd == a || tl.exists(a)
```

Our function does not know the type of list elements, so it cannot use specific operations on it.

We saw the same effect earlier, where we couldn't create a functional solution in `head` for empty lists, because `head` can't know how to create a value of the type parameter `A`.

Parametricity

If we go further, we can even have function signatures, for which only one possible pure implementation remains. We call this concept parametricity, because it works by replacing types with type parameters.

When you use type parameters, you don't have any operations on the types available except those, that you pass in as functions.

Caveat: In many languages, there still exist ways to create wrong implementations, for example a function could still return null, which tricks the type checker. But these are much simpler to detect and will usually fail in a way that is always detected.

Parametricity — Exercise

For example, take the following signature:

```
def para[A,B,C](a: A, b: B)(f: (A,B) => C): C
```

Try to implement it. There is only one way, that satisfies the type checker.

Parametricity — Solution

```
def para[A,B,C](a: A, b: B)(f: (A,B) => C): C = f(a, b)
```

All we have is values of type A and B, and a function that takes an A and B to produce a C. The signature requires C as result.

As we do not know which type C is, we cannot create one, except with the given function.

Higher order functions

```
def para[A,B,C](a: A, b: B)(f: (A,B) => C): C = f(a, b)
```

Our function is also a higher order function, i.e. a function that takes other functions as parameters.

Using functions this way, passing them around and generalizing functions by moving part of their behaviour into a parameter is a core concept in *functional* programming.

Recursion over lists and generalizing to Higher Order Functions

Back to our first list operations...

Looking back, `sum` and `product` are very similar:

```
def sum(ints: List[Int]): Int = ints match
  case Nil => 0
  case Cons(x,xs) => x + sum(xs)

def product(ds: List[Double]): Double = ds match
  case Nil => 1.0
  case Cons(x,xs) => x * product(xs)
```

- We can generalize by pulling subexpressions to arguments
- Additionally, we can turn subexpressions referring to local vars into functions

we're ignoring optimization of zeroes in multiplication here, as it's only an optimization

Back to our first list operations...

Looking back, `sum` and `product` are very similar:

```
def sum(ints: List[Int]): Int = ints match
  case Nil => 0
  case Cons(x,xs) => x + sum(xs)

def product(ds: List[Double]): Double = ds match
  case Nil => 1.0
  case Cons(x,xs) => x * product(xs)
```

- We can generalize by pulling subexpressions to arguments
  empty list result
- Additionally, we can turn subexpressions referring to local vars into functions
  function to add element to result

we're ignoring optimization of zeroes in multiplication here, as it's only an optimization

Right folds

Let's move those parts out:

```
def foldRight(z: A)(f: (A, A) => A): A =
  this match
    case Nil => z
    case Cons(hd, tl) => f(hd, tl.foldRight(z)(f))
```

`z` is our "zero" element, when the list is empty
`f` is the operation we use to combine two elements
But we can make this function even more generic

Right folds

Let's move those parts out:

```
def foldRight[B](z: B)(f: (A, B) => B): B =
  this match
    case Nil => z
    case Cons(hd, tl) => f(hd, tl.foldRight(z)(f))
```

Just by changing the signature, we now can also return something with a type different from the the list element type. This makes folding a list very powerful, as you can express most recursions over the whole list with it.

Right folds

Let’s see, how our summing and multiplying looks with fold:

```
def sum2(ints: List[Int]) =
  ints.foldRight(0)((x,y) => x + y)
```

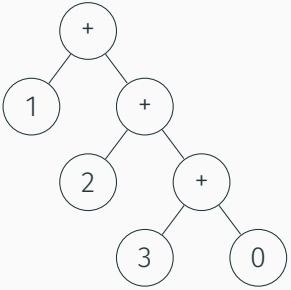
```
def product2(doubles: List[Double]) =
  doubles.foldRight(1.0)(_ * _) // `_ * _` is shorthand for `(x,y) => x * y`
```

Right folds — step by step example

```
def foldRight[B](z: B)(f: (A, B) => B): B =
  this match
    case Nil => z
    case Cons(hd, tl) => f(hd, tl.foldRight(z)(f))
```

RT lets us replace foldRight by its definition step by step:

```
Cons(1, Cons(2, Cons(3, Nil))).foldRight(0)((x,y) => x + y)
1 + Cons(2, Cons(3, Nil)).foldRight(0)((x,y) => x + y)
1 + (2 + Cons(3, Nil).foldRight(0)((x,y) => x + y))
1 + (2 + (3 + Nil.foldRight(0)((x,y) => x + y)))
1 + (2 + (3 + (0)))
6
```



Right folds

```
def foldRight[B](z: B)(f: (A, B) => B): B =
  this match
    case Nil => z
    case Cons(hd, tl) => f(hd, tl.foldRight(z)(f))
```

- What happens, if we pass `Nil` and `Cons` to `foldRight`?

```
List(1,2,3).foldRight(Nil: List[Int])(Cons(_,_))
```

Right folds

```
def foldRight[B](z: B)(f: (A, B) => B): B =
  this match
    case Nil => z
    case Cons(hd, tl) => f(hd, tl.foldRight(z)(f))
```

- What happens, if we pass `Nil` and `Cons` to `foldRight`?

```
List(1,2,3).foldRight(Nil: List[Int])(Cons(_,_))
```

- We get the original list back
- You can think of foldRight as replacing the list constructors `Nil` and `Cons` with `z` and `f`:

```
Cons(1, Cons(2, Nil))
f   (1, f   (2, z  ))
```

Short-circuiting

- Can `product` be implemented using `foldRight` and still halt recursion immediately when encountering a 0.0? Why or why not?

Short-circuiting

- Can `product` be implemented using `foldRight` and still halt recursion immediately when encountering a 0.0? Why or why not?
- No! The argument to `f` is evaluated before calling it, which in this case means traversing rest of the list. To support early termination, we need to defer that until we need it. We'll learn how in a later lecture.

Left folds

```
def foldRight[B](z: B)(f: (A, B) => B): B =
  this match
    case Nil => z
    case Cons(hd, tl) => f(hd, tl.foldRight(z)(f))
```

- `foldRight` is not tail-recursive. Implement a list recursion function `foldLeft`, which is tail-recursive:

  ```
  def foldLeft[B](z: B)(f: (B, A) => B): B = ???
  ```

  Hints: One case stays the same. To be tail-recursive, one case must make the recursive call as last action. Try to follow the types.

Left folds

```
def foldRight[B](z: B)(f: (A, B) => B): B =
  this match
    case Nil => z
    case Cons(hd, tl) => f(hd, tl.foldRight(z)(f))
```

- `foldRight` is not tail-recursive. Implement a list recursion function `foldLeft`, which is tail-recursive:

```
final def foldLeft[B](z: B)(f: (B, A) => B): B = this match
  case Nil => z
  case Cons(hd, tl) => tl.foldLeft(f(z, hd))(f)
```

- Note that the recursive call replaces `z` with our current intermediate result

map, filter and flatMap

- Let's look at general `List` methods, that we'll see on several other structures later: `map`, `filter` and `flatMap`
- Their implementation will be covered on the exercise sheet. We will concentrate on how to use them.

map

- `map` applies a function to each element of a list, producing a new list of same length with the results

```
def map[B](f: A => B): List[B]
```

- Examples:

```
// double all values
List(1,2,3).map(_ * 2) == List(2,4,6)
```

```
// result type can be different
List(1,2,3).map(_.toString) == List("1", "2", "3")
```

filter

- `filter` applies a predicate to each element of a list, producing a new list containing all elements, for which the predicate returned true

```
def filter(p: A => Boolean): List[A]
```

- Example:

```
// only keep odd numbers
List(1,2,3,4,5,6).filter(_ % 2 == 1) == List(1,3,5)
```

flatMap

- `flatMap` is similar to `map`, but the given function returns a list of elements. The resulting lists are concatenated.

```
def flatMap[B](f: A => List[B]): List[B]
```

- Example:

```
// for each number, add it multiplied by 10 to the list
List(1,2,3).flatMap(i => List(i, i * 10)) == List(1, 10, 2, 20, 3, 30)
```

Which is the most powerful?

Which of the three functions would you think to be most powerful, i.e. which can be used to implement the others? Why is the other way round not possible?

```
def map[B](f: A => B): List[B]
```

```
def filter(p: A => Boolean): List[A]
```

```
def flatMap[B](f: A => List[B]): List[B]
```

Which is the most powerful?

Which of the three functions would you think to be most powerful, i.e. which can be used to implement the others? Why is the other way round not possible?

```
def map[B](f: A => B): List[B]
```

```
def filter(p: A => Boolean): List[A]
```

```
def flatMap[B](f: A => List[B]): List[B]
```

`flatMap` is more powerful than map and filter, because you can implement both of them via `flatMap` (at least in combination with List constructors).

With `map`, you can only change each *element*, but not the *structure* of the list (i.e. the length). With `filter`, you can only shorten, but not increase the length.

We'll discuss this in detail later, when we have seen several structures with `flatMap`.

Combining higher order functions

The higher order functions on `List` that we learned are general enough, that we won't need manual recursion in most cases. We can combine them to create more complex operations:

```
case class Order(order: String, date: LocalDate, price: Int)
//Find out how much we paid for pizzas this year
val totalPizzaCosts =
  orders.filter(_.order.contains("Pizza"))
        .filter(_.date.getYear == 2022)
        .map(_.price)
        .foldLeft(0)(_ + _)
```

`for` comprehensions

If we nest several layers of `map` and `flatMap`, this can become confusing:

```
List("a", "b", "c").flatMap(letter =>
    List("1", "2", "3").map(number =>
        letter + number
        )
    )
// returns: List("a1", "a2", "a3", "b1", "b2", "b3", "c1", "c2", "c3")
```

Scala provides syntactic sugar for this, which reads more like a `for` loop in other languages.

`for` comprehensions

```
for
  letter <- List("a", "b", "c")
  number <- List("1", "2", "3")
yield letter + number
// returns: List("a1", "a2", "a3", "b1", "b2", "b3", "c1", "c2", "c3")
```

Each line of a for comprehension is translated to a call to `flatMap`, and the last line to a call to `map`.

The keyword `yield` after the braces marks the return value of the innermost `map` call. So the result of our for comprehension is a `List` with the element type of the yielded expression (here: `String`).

`for` comprehensions

It is possible to use filtering in for comprehensions, but this is more subtle (it uses a different method, which we did not implement, for performance reasons). Using the standard library `List`, you can add `if` lines to a comprehension:

```
for
  a <- List(1,2,3,4)
  if a % 2 == 0 // remove odd numbers
  b <- List(2,4,6,8)
yield a + b
```

which translates to

```
List(1,2,3,4)
  .filter(a => a % 2 == 0)
  .flatMap(a =>
    List(2,4,6,8).map(b => a + b)
    )
```

Lists in the standard library

We have implemented our own list to understand it better. For future exercises you may use the standard library.

Most functions are the same or very similar. One notable difference is `exists`, which takes a function to check for any condition instead of being hardcoded to equality. Examples:

```
val ints = List(1,2,3)  // construction
ints.head == 1
ints.tail == List(2,3)
ints.foldLeft(0)(_ + _) == 6
ints.drop(2) == List(3)
ints.exists(_ > 2) == true   // exists is higher order
ints.exists(_ < 0) == false
```

Lists in the standard library

A more prominent difference is seen during appending and pattern matching: `Cons` is called `::` and usually used in infix notation. Nil is the same.

```
val strings = List("x","y","z")

val moreStrings = "w" :: strings   // prepend to list
moreStrings == List("w", "x","y","z")

moreStrings == "w" :: "x" :: "y" :: "z" :: Nil  // construct manually

val check = moreStrings match
  case Nil => "empty"
  case x :: xs => s"head is $x and tail is $xs" // matching with ::

check == "head is w and tail is List(x, y, z)"
```

Conclusion

We learned today...

- what variance means for our types
- how to define an algebraic data type using an enum
- how to work with immutable and persistent lists
- using pattern matching for destructuring nested datastructures
- what higher order functions are and how we can generalize methods by making them HOFs
- how to use HOFs on list instead of manual recursion.