Notes on Functional Programming in Scala Chapters 3-5

H. Conrad Cunningham

16 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Note: I wrote this set of notes to accompany my lectures in the CSci 555 (Functional Programming) course based partly on the first edition of the book *Functional Programming in Scala* [4] (i.e., the Red Book). In particular, I used chapters 3, 4, and 5.

Prerequisites: In this set of notes, I assume the reader is familiar with the programming concepts and Scala features covered in *Notes on Scala for Java Programmers* [10], *Recursion Styles, Correctness, and Efficiency (Scala Version)* [9], and *Type System Concepts* [11,16:5.2].

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good of April 2022 is a recent version of Firefox from Mozilla.

3 Functional Data Structures

3.1 Chapter Introduction

To do functional programming, we construct programs from collections of pure functions. Given the same arguments, a *pure function* always returns the same result. The function application is thus referentially transparent. By *referentially transparent* we mean that a name or symbol always denotes the same value in some well-defined context in the program.

Such a pure function does not have *side effects*. It does not modify a variable or a data structure in place. It does not set throw an exception or perform input/output. It does nothing that can be seen from outside the function except return its value Thus the data structures in pure functional programs must be *immutable*, not subject to change as the program executes. (If *mutable* data structures are used, no changes to the structures must be detectable outside the function.)

For example, the Scala empty list—written as `Nil` or `List()`—represents a value as immutable as the numbers `2` and `7`.

Just as evaluating the expression `2 + 7` yields a new number `9`, the concatenation of list `c` and list `d` yields a new list (written `c ++ d`) with the elements of `c` followed by the elements of `d`. It does not change the values of the original input lists `c` and `d`.

Perhaps surprisingly, list concatenation does not require both lists to be copied, as we see below.

3.2 A `List` Algebraic Data Type

To explore how to build immutable data structures in Scala, we examine a simplified, singly linked list structure implemented as an algebraic data type. This *list data type* is similar to the builtin Scala `List` data type.

What do we mean by algebraic data type?

3.2.1 Algebraic data types

An *algebraic data type* is a type formed by combining other types, that is, it is a *composite* data type. The data type is created by an algebra of operations of two primary kinds:

- a *sum* operation that constructs values to have one variant among several possible variants. These sum types are also called *tagged*, *disjoint union*, or *variant* types.

  The combining operation is the alternation operator, which denotes the choice of one but not both between two alternatives.

- a *product* operation that combines several values (i.e., *fields*) together to construct a single value. These are *tuple* and *record* types.

  The combining operation is the Cartesian product from set theory.

We can combine sums and products recursively into arbitrarily large structures.

An *enumerated type* is a sum type in which the constructors take no arguments. Each constructor corresponds to a single value.

3.2.2 ADT confusion

Although sometimes the acronym ADT is used for both, an *algebraic data type* is a different concept from an *abstract data type*.

- We specify an algebraic data type with its *syntax* (i.e., structure)—with rules on how to compose and decompose them.
- We specify an abstract data type with its *semantics* (i.e., meaning)—with rules about how the operations behave in relation to one another.

  The modules we build with abstract interfaces, contracts, and data abstraction, such as the Rational Arithmetic modules from the book *Exploring Languages with Interpreters and Functional Programming* [16, Ch. 7], are abstract data types.

Perhaps to add to the confusion, in functional programming we sometimes use an algebraic data type to help define an abstract data type.

3.2.3 Using a Scala trait

A *list* consists of a sequence of values, all of which have the same type. It is a hierarchical data structure. It is either *empty* or it is a pair consisting of a *head* element and a *tail* that is itself a list of elements.

We define `List` as an abstract type using a Scala `trait`. (We could also use an `abstract class` instead of a `trait`.) We define the *constructors* for the algebraic data type using the Scala `case class` and `case object` features.

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Thus `List` is a *sum type* with two alternatives:

- `Nil` constructs the singleton case object that represents the empty list.
- `Cons(h,t)` constructs a new list from an element `h`, called the *head*, and a list `t`, called the *tail*.

`Cons` itself is a *product (tuple) type* with two fields, one of which is itself a `List`.

The `sealed` keyword tells the Scala compiler that all alternative cases (i.e., subtypes) are declared in the current source file. No new cases can be added elsewhere. This enables the compiler to generate safe and efficient code for pattern matching.

As we have seen previously, for each `case class` and `case object`, the Scala compiler generates:

- a constructor function (e.g., `Cons`)
- accessor functions (methods) for each field (e.g., `head` and `tail` on `Cons`)
- new definitions for `equals`, `hashcode`, and `toString`

In addition, the `case object` construct generates a *singleton object*—a new type with exactly one instance.

Programs can use the constructors to build instances and use the pattern matching to recognize the structure of instances and decompose them for processing.

`List` is a polymorphic type. What does polymorphic mean? We examine that in Section 3.2.4.

3.2.3.1 Aside on Haskell In Haskell, an algebraic data type similar to the Scala `List[A]` is the following:

```
data List a = Nil | Cons a (List a)
              deriving (Show, Eq)
```

The Haskell `List a` is a type similar to the Scala `List[A]`. However, `Nil` and `Cons[A]` are subtypes of `List` in Scala, but they are not types in Haskell. In Haskell, they are constructor functions that return values of type `List a`.

3.2.4 Polymorphism

Polymorphism refers to the property of having "many shapes". In programming languages, we are primarily interested in how *polymorphic* function names (or operator symbols) are associated with implementations of the functions (or operations).

Scala is a hybrid, object-functional language. Its type system supports all three types of polymorphism discussed in the notes on type systems concepts [11,16:5.2].

- *subtyping* by extending classes and traits
- *parametric polymorphism* by using generic type parameters,
- *overloading* through both the Java-like mechanisms and Haskell-like "type classes"

Scala's *type class pattern* builds on the languages's `implicit` classes and conversions. A type class enables a programmer to enrich an existing class with an

extended interface and new methods without redefining the class or subclassing it.

For example, Scala extends the Java `String` class (which is `final` and thus cannot be subclassed) with new features from the `RichString` wrapper class. The Scala `implicit` mechanisms associate the two classes "behind the scene". We defer further discussion of implicits until later in the semester.

Note: The type class feature arose from the language Haskell. Similar capabilities are called extension methods in C# and protocols in Clojure and Elixir.

The `List` data type defined above is polymorphic; it exhibits both subtyping and parametric polymorphism. `Nil` and `Cons` are subtypes of `List`. The generic type parameter `A` denotes the type of the elements that occur in the list. For example, `List[Double]` denotes a list of double-precision floating point numbers.

What does the `+` annotation mean in the definition `List[+A]`?

3.2.5 Variance

The presence of both subtyping and parametric polymorphism leads to the question of how these features interact—that is, the concept of *variance*.

3.2.5.1 Covariance and contravariance Suppose we have a supertype `Fish` with a subtype `Bass`, which in turn has a subtype `BlackBass`.

For generic data type `List[A]` as defined above, consider `List[Fish]` and `List[Bass]`.

- If `List[Bass]` is a subtype of `List[Fish]`, preserving the subtyping order, then the relationship is *covariant*.
- If `List[Fish]` is a subtype of `List[Bass]`, reversing the subtyping order, then the relationship is *contravariant*.
- If there is no subtype relationship between `List[Fish]` and `List[Bass]`, then the relationship is *invariant* (sometimes called *nonvariant*).

In the Scala definition `List[+A]` above, the `+` annotation in front of the `A` is a *variance annotation*. The `+` means that parameter `A` is a *covariant* parameter of `List`. That is, for all types `X` and `Y` such that `X` is a subtype of `Y`, then `List[X]` is a is subtype of `List[Y]`.

If we leave off the variance annotation, then `List` would be *invariant* in the type parameter. Regardless of how types `X` and `Y` may be related, `List[X]` and `List[Y]` are unrelated.

If we were put a `-` annotation in front of `A`, then we declare parameter `A` to be *contravariant*. That is, for all types `X` and `Y` such that `X` is a subtype of `Y`, then `List[Y]` is a is subtype of `List[X]`.

In the definition of the `List` algebraic data type, `Nil` extends `List[Nothing]`. `Nothing` is a subtype of all other types. In conjunction with covariance, the `Nil` list can be considered a list of any type.

Aside: Note the position of `Nothing` at the bottom of Scala's unified type hierarchy diagram [33].

3.2.5.2 Function subtypes Now consider first-class functions. When is one function a subtype of another?

From the notes on type systems concepts [11,16:5.2], we recall that we should be able to safely *substitute* elements of a subtype `S` for elements of type `T` because `S`'s operations behave the "same" as `T`'s operations. That is, the relationship satisfies the *Liskov Substitution Principle* [23,42].

Using the `Fish` type hierarchy above, consider a function of type `Bass => X` (for some type `X`). It would be unsafe to use a function of type `BlackBass => X` in its place. The function would be undefined for any values that are of type `Bass` but are not of type `BlackBass`. So a function with a input type `BlackBass` is not a subtype of a function with input `Bass`.

However, a function of type `Fish => X` would be defined for any value that is of type `Bass`. So we need to examine the relationships between the output types to determine what the subtyping relationship is between the functions.

Consider a function of type `X => Bass`. A function of type `X => BlackBass` can be safely used in its place because a `BlackBass` is a `Bass`, so the function yields a value of the expected type.

However, a function of type `X => Fish` cannot be safely used in place of the `X => Bass` function. It may yield some value that is a `Fish` but not a `Bass`.

Thus we could safely use a `Bass => Bass` function in place of a `Bass => Fish`, `BlackBass => Bass`, or `BlackBass => Fish` function. Thus `Bass => Bass` is a subtype of the others.

However, we could not safely use a `Bass => Bass` function in place of a `Bass => BlackBass`, `BlackBass => BlackBass`, `Fish => Fish`, `Fish => Bass`, or `Fish => BlackBass` function. Thus `Bass => Bass` is a not a subtype of the others.

Bringing these observations together, a function type `S1 => S2` is a subtype of a function type `T1 => T2` if and only if:

- `T1` is a subtype of `S1` (i.e., contravariant on the input type)
- `S2` is a subtype of `T2` (i.e., covariant on the output type)

This general observation is consistent with the applicable theory.

For a Scala function of type `S => T`, we thus say the `S` is in a *contravariant position* and `T` is in a *covariant position*.

TODO: May want to discuss multiargument functions.

3.2.6 Defining functions in companion object

The *companion object* for a trait or class is a singleton object with the same name as the trait or class. The companion object for the `List` trait is a convenient place to define functions for manipulating the lists.

Because `List` is a Scala algebraic data type (implemented with case classes), we can use pattern matching in our function definitions. Pattern matching helps enable the *form of the algorithm* to match the *form of the data structure*. Or, in terms that Chiusano and Bjarnason use, it helps in *following types to implementations* [4].

Note: Other writers call this design approach *type-driven development* [1] or *type-first development* [27].

This is considered elegant. It is also pragmatic. The structure of the data often suggests the algorithm needed for a task.

In general, lists have two cases that must be handled: the empty list (represented by `Nil`) and the nonempty list (represented by `Cons`). The first yields a *base leg* of a recursive algorithm; the second yields a *recursive leg*.

Breaking a definition for a list-processing function into these two cases is usually a good place to begin. We must ensure the recursion *terminates*—that each successive recursive call gets closer to the base case.

3.2.7 Function to sum a list

Consider a function `sum` to add together all the elements in a list of integers. That is, if the list is $v_1, v_2, v_3, \cdots, v_n$, then the sum of the list is the value resulting from inserting the addition operator between consecutive elements of the list:

$$v_1 + v_2 + v_3 + \cdots + v_n$$

Because addition is an *associative* operation, the additions can be computed in any order. That is, for any integers x, y, and z:

$$(x + y) + z = x + (y + z)$$

We can use the form of the data to guide the form of the algorithm—or *follow the type to the implementation* of the function.

What is the sum of an empty list?

Because there are no numbers to add, then, intuitively, zero seems to be the proper value for the sum.

In general, if some binary operation is inserted between the elements of a list, then the result for an empty list is the *identity element* for the operation. Zero is the identity element for addition because, for all integers x:

$$0 + x = x = x + 0$$

Now, how can we compute the sum of a nonempty list?

Because a nonempty list has at least one element, we can remove one element and add it to the sum of the rest of the list. Note that the "rest of the list" is a simpler (i.e., shorter) list than the original list. This suggests a recursive definition.

The fact that we define lists recursively as a `Cons` of a head element with a tail list suggests that we structure the algorithm around the structure of the *beginning* of the list.

Bringing together the two cases above, we can define the function `sum` in Scala using pattern matching as follows:

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

The length of a non-nil argument decreases by one for each successive recursive application. Thus `sum` will eventually be applied to a `Nil` argument and terminate.

For a list consisting of elements 2, 4, 6, and 8, that is,

```
Cons(2,Cons(4,Cons(6,Cons(8,Nil))))
```

function `sum` computes:

```
2 + (4 + (6 + (8 + 0)))
```

Function `sum` is backward linear recursive; its time and space complexity are both O(`n`), where `n` is the length of the input list.

We could, of course, redefine this to use a tail-recursive auxiliary function. With *tail call optimization*, the recursion could be converted into a loop. It would still be order O(`n`) in time complexity (but with a smaller constant factor) and O(1) space.

3.2.8 Function to multiply a list

Now consider a function `product` to multiply together a list of floating point numbers. The product of an empty list is 1 (which is the identity element for multiplication). The product of a nonempty list is the head of the list multiplied by the product of the tail of the list, except that, if a 0 occurs anywhere in the

list, the product of the list is 0. We can thus define `product` with two bases cases and one recursive case, as follows:

```
def product(ds: List[Double]): Double = ds match {
    case Nil          => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(x,xs)   => x * product(xs)
}
```

Note: 0 is the *zero element* for the multiplication operation on real numbers. That is, for all real numbers x:

$$0 * x = x * 0 = 0$$

For a list consisting of elements 2.0, 4.0, 6.0, and 8.0, that is,

```
Cons(2.0,Cons(4.0,Cons(6.0,Cons(8.0,Nil))))
```

function `product` computes:

```
2.0 * (4.0 * (6.0 * (8.0 * 1.0)))
```

For a list consisting of elements 2.0, 0.0, 6.0, and 8.0, function `product` "short circuits" the computation as:

```
2.0 * 0.0
```

Like `sum`, function `product` is backward linear recursive; it has a worst-case time complexity of O(`n`), where `n` is the length of the input list. It terminates because the argument of each successive recursive call is one element shorter than the previous call, approaching one of the base cases.

3.2.9 Function to remove adjacent duplicates

Consider the problem of removing adjacent duplicate elements from a list. That is, we want to replace a group of adjacent elements having the same value by a single occurrence of that value.

As with the above functions, we let the form of the data guide the form of the algorithm, following the type to the implementation.

The notion of adjacency is only meaningful when there are two or more of something. Thus, in approaching this problem, there seem to be three cases to consider:

- The argument is a list whose first two elements are duplicates; in which case one of them should be removed from the result.
- The argument is a list whose first two elements are not duplicates; in which case both elements are needed in the result.
- The argument is a list with fewer than two elements; in which case the remaining element, if any, is needed in the result.

Of course, we must be careful that sequences of more than two duplicates are handled properly.

Our algorithm thus can examine the first two elements of the list. If they are equal, then the first is discarded and the process is repeated recursively on the list remaining. If they are not equal, then the first element is retained in the result and the process is repeated on the list remaining. In either case the remaining list is one element shorter than the original list. When the list has fewer than two elements, it is simply returned as the result.

In Scala, we can define function `remdups` as follows:

```
def remdups[A](ls: List[A]): List[A] = ls match {
    case Cons(x, Cons(y,ys)) =>
        if (x == y)
            remdups(Cons(y,ys))          // duplicate
        else
            Cons(x,remdups(Cons(y,ys))) // non-duplicate
    case _                    => ls
}
```

Function `remdups` puts the base case last in the pattern match to take advantage of the wildcard match using `_`. This needs to match either `Nil` and `Cons(_,Nil)`.

The function also depends upon the ability to compare any two elements of the list for equality. Because `equals` is builtin operation on all types in Scala, we can define this function polymorphically without constraints on the type variable `A`.

Like the previous functions, `remdups` is backward linear recursive; it takes a number of steps that is proportional to the length of the list. This function has a recursive call on both the duplicate and non-duplicate legs. Each of these recursive calls uses a list that is shorter than the previous call, thus moving closer to the base case.

3.2.10 Variadic function `apply`

We can also add a function `apply` to the companion object `List`.

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

Scala treats an `apply` method in an `object` specially. We can invoke the `apply` method using a postfix `()` operator. Given a singleton object `X` with an `apply` method, the Scala complier translates the notation `X(p)` into the method call `X.apply(p)`.

In the `List` data type, function `apply` is a *variadic function*. It accepts zero or more arguments of type `A` as denoted by the type annotation `A*` in the parameter list. Scala collects these arguments into a `Seq` (sequence) data type for processing within the function. The special syntax `_*` reverses this and passes a sequence to another function as variadic parameters. Builtin Scala data structures such as lists, queues, and vectors implement `Seq`. It provides methods such as the `isEmpty`, `head`, and `tail` methods used in `apply`.

It is common to define a variadic `apply` methods for algebraic data types. This method enables us to create instances of the data type conveniently. For example, `List(1,2,3)` creates a three-element list of integers with `1` at the head.

3.3 Data Sharing

Suppose we have the declaration

```
val xs = Cons(1,Cons(2,Cons(3,Nil)))
```

or the more concise equivalent using the `apply` method:

```
val xs = List(1,2,3)
```

As we learned in the data structures course, we can implement this list as a linked list `xs` with three cells with the values `1`, `2`, and `3`, as shown in Figure 3.1.

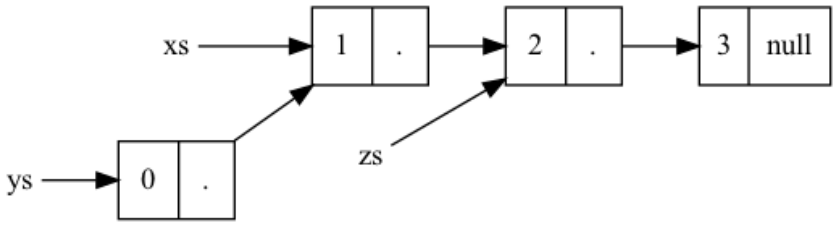

Figure 3.1: Data sharing in lists.

Consider the following declarations

```
val ys = Cons(0,xs)
val zs = xs.tail
```

where

- `Cons(0,xs)` returns a list that has a new cell containing `0` in front of the previous list
- `xs.tail` returns the list consisting of the last two elements of `xs`

If the linked list `xs` is immutable (i.e., the values and pointers in the three cells cannot be changed), then neither of these operations requires any copying.

- The first just constructs a new cell containing `0`, links it to the first cell in list `xs`, and initializes `ys` with a reference to the new cell.
- The second just returns a reference to the second cell in list `xs` and initializes `zs` with this reference.
- The original list `xs` is still available, unaltered.

This is called *data sharing*. It enables the programming language to implement immutable data structures efficiently, without copying in many key cases.

Also, such functional data structures are *persistent* because existing references are never changed by operations on the data structure.

3.3.1 Function to take tail of list

Consider a function that takes a `List` and returns its tail `List`. (This is different from the `tail` accessor method on `Cons`.)

If the `List` is a `Cons`, then the function can return the `tail` element of the cell. What should it do if the list is a `Nil`?

There are several possibilities:

- return `Nil`
- throw an exception (with perhaps a custom error string)
- leave the function undefined in this case (which would result with a standard pattern match exception)

Generally speaking, the first choice seems misleading. It seems illogical for an empty list to have a tail. And consider a typical usage of the function. It is normally an error for a program to attempt to get the tail of an empty list. A program can efficiently check whether a list is empty or not. So, in this case, it is probably better to take the second or third approach.

We choose to implement `tailList` so that it explicitly throws an exception. It can be defined in the companion object for `List` as follows:

```
def tailList[A](ls: List[A]): List[A] = ls match {
    case Nil        => sys.error("tail of empty list")
    case Cons(_,xs) => xs
}
```

Above, the value of the `head` field of the `Cons` pattern is irrelevant in the computation on the right-hand side. There is no need to introduce a new variable for that value, so we use the wildcard variable `_` to indicate that the value is not needed.

Function `tailList` is O(1) in time complexity. It does not need to copy the list. It is sufficient for it to just return a reference to the tail of the original immutable list. This return value shares the data with its input argument.

3.3.2 Function to drop from beginning of list

We can generalize `tailList` to a function `drop` that removes the first `n` elements of a list, as follows:

```
def drop[A](ls: List[A], n: Int): List[A] =
    if (n <= 0) ls
    else ls match {
        case Nil        => Nil
        case Cons(_,xs) => drop(xs, n-1)
}
```

The `drop` function terminates when either the list argument is `Nil` or the integer argument 0 or negative. The function eventually terminates because each recursive call both shortens the list and decrements the integer.

This function takes a different approach to the empty list issue than `tailList` does. Although it seems illogical to take the `tailList` of an empty list, dropping the first element from an empty list seems subtly different. Given that we often use `drop` in cases where the length of the input list is unknown, dropping the first element of an empty list does not necessarily indicate a program error.

Suppose `drop` throws an exception when called with an empty list. To avoid this situation, the program might need to determine the length of the list argument. This is inefficient, usually requiring a traversal of the entire list to count the elements.

3.3.3 Function to append lists

Consider the definition of an *append* (list concatenation) function. We must define the `append` function in terms of the constructors `Nil` and `Cons`, already defined list functions, and recursive applications of itself.

As with previous functions, we follow the type to the implementation—let the form of the data guide the form of the algorithm.

The `Cons` constructor takes an element as its left operand and a list as its right operand and returns a new list with the left operand as the head and the right operand as the tail.

Similarly, append must take a list as its left operand and a list as its right operand and return a new list with the left operand as the initial segment and the right operand as the final segment.

Given the definition of `Cons`, it seems reasonable that an algorithm for `append` must consider the structure of its left operand. Thus we consider the cases for nil and non-nil left operands.

- If the left operand is `Nil`, then the function can just return the right operand.

- If the left operand is a `Cons` (that is, non-nil), then the result consists of the left operand's head followed by the append of the left operand's tail to the right operand.

In following the type to the implementation, we use the form of the left operand in a pattern match. We define `append` as follows:

```
def append[A](ls: List[A], rs: List[A]): List[A] = ls match {
    case Nil        => rs
    case Cons(x,xs) => Cons(x, append(xs, rs))
}
```

For the recursive application of `append`, the length of the left operand decreases by one. Hence the left operand of an `append` application eventually becomes `Nil`, allowing the evaluation to terminate.

The number of steps needed to evaluate `append(as,bs)` is proportional to the length of `as`, the left operand. That is, it is O(`n`), where `n` is the length of list `as`.

Moreover, `append(as,bs)` only needs to copy the list `as`. The list `bs` is shared between the second operand and the result. If we did a similar function to append two (mutable) arrays, we would need to copy both input arrays to create the output array. Thus, in this case, a linked list is more efficient than arrays!

The append operation has a number of useful mathematical (algebraic) properties, for example, associativity and an identity element.

Associativity—For any finite lists `xs`, `ys`, and `zs`:

```
append(xs,append(ys,zs)) = append(append(xs,ys),zs)
```

Identity—For any finite list `xs`:

```
append(Nil,xs) = append(xs,Nil) = xs
```

Scala's builtin `List` type uses the infix operator `++` for the "append" operation. For this operator, associativity can be stated conveniently with the equation:

```
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

Mathematically, the `List` data type and the binary operation `append` form a kind of abstract algebra called a *monoid*. Function `append` is closed (i.e., it takes two lists and gives a list back), is associative, and has an identity element.

Aside: For more information on operations and algebraic structures, see the Review of Relevant Mathematics chapter [16, Ch. 80] in my book *Exploring Languages with Interpreters and Functional Programming*. For discussion of how to prove properties like those above, see the Proving Haskell Laws chapter [16, Ch. 25] in the same book.

3.4 Other List Functions

3.4.1 Tail recursive function reverse

Consider the problem of reversing the order of the elements in a list.

Again we can use the structure of the data to guide the algorithm development. If the argument is a nil list, then the function returns a nil list. If the argument is a non-nil list, then the function can append the head element at the back of the reversed tail.

```
def rev[A](ls: List[A]): List[A] = ls match {
    case Nil        => Nil
    case Cons(x,xs) => append(rev(xs),List(x))
}
```

Given that evaluation of `append` terminates, the evaluation of `rev` also terminates because all recursive applications decrease the length of the argument by one.

How efficient is this function?

The evaluation of `rev` takes O(`n^2`) steps, where `n` is the length of the argument. There are O(`n`) applications of `rev` . For each application of `rev` there are O(`n`) applications of `append` .

The initial list and its reverse do not share data.

Function `rev` has a number of useful properties, for example a distribution and an inverse properties.

Distribution—For any finite lists `xs` and `ys`:

```
rev(append(xs,ys)) = append(rev(ys), rev(xs))
```

Inverse—For any finite list `xs`:

```
rev(rev(xs)) = xs
```

Can we define a function to reverse a list using a "more efficient" tail recursive solution?

As we have seen, a common technique for converting a backward linear recursive definition like `rev` into a *tail recursive* definition is to use an *accumulating parameter* to build up the desired result incrementally. A possible definition for a tail recursive auxiliary function is:

```
def revAux[A](ls: List[A], as: List[A]): List[A] = ls match {
    case Nil        => as
    case Cons(x,xs) => revAux(xs,Cons(x,as))
}
```

In this definition parameter `as` is the accumulating parameter. The head of the first argument becomes the new head of the accumulating parameter for the tail

recursive call. The tail of the first argument becomes the new first argument for the tail recursive call.

We know that `revAux` terminates because, for each recursive application, the length of the first argument decreases toward the base case of `Nil`.

We note that `rev(xs)` is equivalent to `revAux(xs,Nil)` .

To define a single-argument replacement for `rev` , we can embed the definition of `revAux'` as an *auxiliary* function within the definition of a new function `reverse` .

```
def reverse[A](ls: List[A]): List[A] = {
    def revAux[A](rs: List[A], as: List[A]): List[A] =
        rs match {
            case Nil        => as
            case Cons(x,xs) => revAux(xs,Cons(x,as))
        }
    revAux(ls,Nil)
}
```

Function `reverse(xs)` returns the value from `revAux(xs,Nil)`.

How efficient is this function?

The evaluation of `reverse` takes O(`n`) steps, where `n` is the length of the argument. There is one application of `revAux` for each element; `revAux` requires a single O(1) `Cons` operation in the accumulating parameter.

Where did the increase in efficiency come from?

Each application of `rev` applies `append`, a linear time (i.e., O(`n`)) function. In `revAux`, we replaced the applications of `append` by applications of `Cons`, a constant time (i.e., O(1)) function.

In addition, a compiler or interpreter that does tail call optimization can translate this tail recursive call into a loop on the host machine.

3.4.2 Higher-order function dropWhile

Consider a function `dropWhile` that removes elements from the front of a `List` while its predicate argument (a Boolean function) holds.

```
def dropWhile [A](ls: List[A], f: A => Boolean): List[A] =
    ls match {
        case Cons(x,xs) if f(x) => dropWhile(xs, f)
        case _                  => ls
    }
```

This higher-order function terminates when either the list is empty or the head of the list makes the predicate false. For each successive recursive call, the list

argument is one element shorter than the previous call, so the function eventually terminates.

If evaluation of function argument `p` is O(1), then function `dropWhile` has worst-case time complexity O(`n`), where `n` is the length of its first operand. The result list shares data with the input list.

3.4.3 Curried function dropWhile

We often pass *anonymous functions* to higher-order utility functions like `dropwhile`, which has the signature:

```
def dropWhile[A](ls: List[A], f: A => Boolean): List[A]
```

When we call `dropWhile` with an anonymous function for `f`, we must specify the type of its argument, as follows:

```
val xs: List[Int] = List(1,2,3,4,5)
val ex1 = dropWhile(xs, (x: Int) => x < 4)
```

Even though it is clear from the first argument that higher order argument `f` must take an integer as its argument, the Scala *type inference* mechanism cannot detect this.

However, if we rewrite `dropWhile` in the following form, type inference can work as we want:

```
def dropWhile2[A](ls: List[A])(f: A => Boolean): List[A] =
    ls match {
        case Cons(x,xs) if f(x) => dropWhile2(xs)(f)
        case _                  => ls
    }
```

Function `dropWhile2` is written in *curried* form above. In this form, a function that takes two arguments can be represented as a function that takes the first argument and returns a function, which itself takes the second argument.

If we apply `dropWhile2` to just the first argument, we get a function. We call this a *partial application* of `dropWhile2`.

More generally, a function that takes multiple arguments can be represented by a function that takes its arguments in groups of one or more from left to right. If the function is partially applied to the first group, it returns a function that takes the remaining groups, and so forth.

Currying and partial application are directly useful in a number of ways in our programs. Here currying is indirectly useful by assisting type inference. If a function is defined with multiple groups of arguments, the type information flows from one group to another, left to right. In `dropWhile2`, the first argument group binds type variable `A` to `Int`. Then this binding can be used in the second argument group.

3.5 Generalizing to Higher Order Functions

3.5.1 Fold right

Consider the `sum` and `product` functions we defined above, ignoring the short-cut handling of the zero element in `product`.

```
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)
}
```

What do `sum` and `product` have in common? What differs?

Both exhibit the same *pattern of computation*.

- Both take a list as input.

  But the element type differs. Function `sum` takes a list of `Int` values and `product` takes a list of `Double` values.

- Both insert a binary operator between all the consecutive elements of the list in order to reduce the list to a single value.

  But the binary operation differs. Function `sum` applies integer addition and `product` applies double-precision floating-point multiplication.

- Both group the operations from the right to the left.

- Both functions return some value for an empty list. The function extends nonempty input lists to implicitly include this value as the "rightmost" value of the input list.

  But the actual value differs.

  Function `sum` returns integer 0, the (right) identity element for addition.

  Function `product` returns 1.0, the (right) identity element for multiplication.

  In general, this value could be something other than the (right) identity element.

- Both return a value of the same element type as the input list.

  But the input type differs, as we noted above.

Both functions insert operations of type `(A,A) => A` between elements a list of type `List[A]`, for some generic type `A`.

But these are special cases of more general operations of type `(A,B) => B`. In this case, the value returned must be of type `B` in the case of both empty and nonempty lists.

Whenever we recognize a pattern like this, we can systematically *generalize the function* definition as follows:

1. Do a *scope-commonality-variability* (SCV) analysis [7] on the set of related functions.

   That is, identify what is to be included and what not (i.e., the *scope*), the parts of functions that are the same (the *commonalities* or *frozen spots*), and the parts that differ (the *variabilities* or *hot spots*).

2. Leave the commonalities in the generalized function's body.

3. Move the variabilities into the generalized function's header—its type signature and parameter list.

   - If the part moved to the generalized function's parameter list is an expression, then make that part a function with a parameter for each local variable accessed.
   - If a data type potentially differs from a specific type used in the set of related functions, then add a type parameter to the generalized function.
   - If the same data value or type appears in multiple roles, then consider adding distinct type or value parameters for each role.

4. Consider other approaches if the generalized function's type signature and parameter list become too complex.

   For example, we can introduce new data or procedural abstractions for parts of the generalized function. These may be in the same "module" (i.e., object, class, etc.) as the generalized function, in an appropriately defined separate "module" that is imported, etc. A separate module may better accomplish the desired parameterization of the function.

A similar approach can be used to generalize a whole class.

Following the above guidelines, we can express the common pattern from `sum` and `product` as a new (broadly useful) polymorphic, higher-order function `foldRight`, which we define as follows:

```
def foldRight[A,B](ls: List[A], z: B)(f: (A, B) => B): B =
    ls match {
        case Nil        => z
        case Cons(x,xs) => f(x, foldRight(xs, z)(f))
    }
```

This function:

- passes in the binary operation `f` that combines the list elements
- passes in the element `z` to be returned for empty lists (often the right identity element for the operation, but this is not required)
- uses two type parameters `A` and `B`—one for the type of elements in the list and one for the type of the result

The `foldRight` function "folds" the list elements (of type `A`) into a value (of type `B`) by "inserting" operation `f` between the elements, with value `z` "appended" as the rightmost element. For example, `foldRight(List(1,2,3),z)(f)` expands to `f(1,f(2,f(3,z)))`.

Function `foldRight` is not tail recursive, so it needs a new stack frame for each element of the input list. If its list argument is long or the folding function itself is expensive, then the function can terminate with a *stack overflow* error.

We can specialize `foldRight` to have the same functionality as `sum` and `product`.

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

def product2(ns: List[Double]) =
    foldRight(ns, 1.0)(_ * _)
```

The expression `(_ * _)` in `product2` is a concise notation for the anonymous function `(x,y) => x * y`. The two underscores denote two distinct anonymous variables. This concise notation can be used in a context where Scala's type inference mechanism can determine the types of the anonymous variables.

We can construct a recursive function to compute the length of a polymorphic list. However, we can also express this computation using `foldRight`, as follows:

```
def length[A](ls: List[A]): Int =
    foldRight(ls, 0)((_,acc) => acc + 1)
```

We use the `z` parameter to accumulate the count, starting it at 0. Higher order argument `f` is a function that takes an element of the list as its left argument and the previous accumulator as its right argument and returns it incremented by 1. In this application, `z` is not the identity element for `f` by a convenient beginning value for the counter.

We can construct an "append" function that uses `foldRight` as follows:

```
def append2[A](ls: List[A], rs: List[A]): List[A] =
    foldRight(ls, rs)(Cons(_,_))
```

Here the list that `foldRight` operates on the first argument of the append. The `z` parameter is the entire second argument and the combining function is just `Cons`. So the effect is to replace the `Nil` at the end of the first list by the entire second list.

We can construct a recursive function that takes a list of lists and returns a "flat" list that has the same elements in the same order. We can also express this `concat` function in terms of `foldRight`, as follows:

```
def concat[A](ls: List[List[A]]): List[A] =
    foldRight(ls, Nil: List[A])(append)
```

Function `append` takes time proportional to the length of its first list argument. This argument does not grow larger because of right associativity of `foldRight`. Thus `concat` takes time proportional to the total length of all the lists.

Above, we "pass" the `append` function without writing an explicit anonymous function definition (i.e., *function literal*) such as `(xs,ys) => append(xs,ys)` or `append(_,_)`.

In `concat`, for which Scala can infer the types of `append`'s arguments, the compiler can generate the needed function literal. In other cases, we would need to use *partial application* notation such as

```
append _
```

or an explicit function literal such as

```
(xs: List[A], ys: List[A]) => append(xs,ys)
```

to enable the compiler to infer the types.

Above we defined function `foldRight` as a backward recursive function that processes the elements of a list one by one. However, as we have seen, it is often more useful to think of `foldRight` as a powerful list operator that reduces the element of the list into a single value. We can combine `foldRight` with other operators to conveniently construct list processing programs.

3.5.2 Fold left

We designed function `foldRight` above as a backward linear recursive function with the signature:

```
foldRight[A,B](as: List[A], z: B)(f: (A, B) => B): B
```

As noted:

```
foldRight(List(1,2,3),z)(f) == f(1,f(2,f(3,z)))
```

Consider a function `foldLeft` such that:

```
foldLeft(List(1,2,3),z)(f) == f(f(f(z,1),2),3)
```

This function folds from the left. It offers us the opportunity to use parameter `z` as an accumulating parameter in a tail recursive implementation, as follows:

```
@annotation.tailrec
def foldLeft[A,B](ls: List[A], z: B)(f: (B, A) => B): B =
    ls match {
```

```
        case Nil        => z
        case Cons(x,xs) => foldLeft(xs, f(z,x))(f)
    }
```

In the first line above, we *annotate* function `foldLeft` as tail recursive using `@annotation.tailrec`. If the function is not tail recursive, the compiler gives an error, rather than silently generating code that does not use tail call optimization (i.e., does not convert the recursion to a loop).

We can implement list sum, product, and length functions with `foldLeft`, similar to what we did with `foldRight`.

```
def sum3(ns: List[Int]) =
    foldLeft(ns, 0)(_ + _)

def product3(ns: List[Double]) =
    foldLeft(ns, 1.0)(_ * _)
```

Given that addition and multiplication of numbers are associative and have identity elements, `sum3` and `product3` use the same values for parameters `z` and `f` as `foldRight`.

Function `length2` that uses `foldLeft` is like `length` except that the arguments of function `f` are reversed.

```
def length2[A](ls: List[A]): Int =
    foldLeft(ls, 0)((acc,_) => acc + 1)
```

We can also implement list reversal using `foldLeft` as follows:

```
def reverse2[A](ls: List[A]): List[A] =
    foldLeft(ls, List[A]())((acc,x) => Cons(x,acc))
```

This gives a solution similar to the tail recursive `reverse` function above. The `z` value is initially an empty list; the folding function `f` uses `Cons` to "attach" each element of the list to front of the accumulator, incrementally building the list in reverse order.

Because `foldLeft` is tail recursive and `foldRight` is not, `foldLeft` is usually safer and more efficient to use in than `foldRight`. (If the list argument is lazily evaluated or the function argument `f` is nonstrict in at least one of its arguments, then there are other factors to consider. We will discuss what we mean by "lazily evaluated" and "nonstrict" later in the course.)

To avoid the stack overflow situation with `foldRight`, we can first apply `reverse` to the list argument and then apply `foldLeft` as follows:

```
def foldRight2[A,B](ls: List[A], z: B)(f: (A,B) => B): B =
    foldLeft(reverse(ls), z)((b,a) => f(a,b))
```

The combining function in the call to `foldLeft` is the same as the one passed to `foldRight2` except that its arguments are reversed.

3.5.3 Map

Consider the following two functions, noting their type signatures and patterns of recursion.

The first, `squareAll`, takes a list of integers and returns the corresponding list of squares of the integers.

```
def squareAll(ns: List[Int]): List[Int] = ns match {
    case Nil         => Nil
    case Cons(x, xs) => Cons(x*x, squareAll(xs))
}
```

The second, `lengthAll`, takes a list of lists and returns the corresponding list of the lengths of the element lists

```
def lengthAll[A](lss: List[List[A]]): List[Int] =
    lss match {
        case Nil           => Nil
        case Cons(xs, xss) => Cons(length(xs),lengthAll(xss))
    }
```

Although these functions take different kinds of data (a list of integers versus a list of polymorphically typed lists) and apply different operations (squaring versus list length), they exhibit the same pattern of computation. That is, both take a list and apply some function to each element to generate a resulting list of the same size as the original.

As with the fold functions, the combination of polymorphic typing and higher-order functions allows us to abstract this pattern of computation into a higher-order function.

We can abstract the pattern of computation common to `squareAll` and `lengthAll` as the (broadly useful) function `map`, defined as follows:

```
def map[A,B](ls: List[A])(f: A => B): List[B] = ls match {
    case Nil        => Nil
    case Cons(x,xs) => Cons(f(x),map(xs)(f))
}
```

Function `map` takes a list of type `A` elements, applies function `f` of type `A => B` to each element, and returns a list of the resulting type `B` elements.

Thus we can redefine `squareAll` and `lengthAll` using `map` as follows:

```
def squareAll2(ns: List[Int]): List[Int] =
    map(ns)(x => x*x)

def lengthAll2[A](lss: List[List[A]]): List[Int] =
    map(lss)(length)
```

We can implement `map` itself using `foldRight` as follows:

```
def map1[A,B](ls: List[A])(f: A => B): List[B] =
    foldRight(ls, Nil: List[B])((x,xs) => Cons(f(x),xs))
```

The folding function `(x,xs) => Cons(f(x),xs)` applies the mapping function `f` to the next element of the list (moving right to left) and attaches the result on the front of the processed tail.

As implemented above, function `map` is backward recursive; it thus requires a stack frame for each element of its list argument. For long lists, the recursion can cause a stack overflow error. Function `map1` uses `foldRight`, which has similar characteristics. So we need to use these functions with care. However, we can use the reversal technique illustrated in `foldRight2` if necessary.

We could also optimize function `map` using *local mutation*. That is, we can use a mutable data structure within the `map` function but not allow this structure to be accessed outside of `map`. The following function takes that approach, using a `ListBuffer`:

```
def map2[A,B](ls: List[A])(f: A => B): List[B] = {
    val buf = new collection.mutable.ListBuffer[B]

    @annotation.tailrec
    def go(ls: List[A]): Unit = ls match {
        case Nil        => ()
        case Cons(x,xs) => buf += f(x); go(xs)
    }

    go(ls)
    List(buf.toList: _*)
}
```

A `ListBuffer` is a mutable list data structure from the Scala library. The operation += appends a single element to the end of the buffer in constant time. The method `toList` converts the `ListBuffer` to a Scala immutable list, which is similar to the data structure we are developing in this module.

3.5.4 Filter

Consider the following two functions.

The first, `getEven`, takes a list of integers and returns the list of those integers that are even (i.e., are multiples of 2). The function preserves the relative order of the elements in the list.

```
def getEven(ns: List[Int]): List[Int] = ns match {
    case Nil        => Nil
    case Cons(x,xs) =>
        if (x % 2 == 0)  // divisible evenly by 2
            Cons(x,getEven(xs))
```

```
        else
            getEven(xs)
}
```

The second, `doublePos`, takes a list of integers and returns the list of doubles of the positive integers from the input list; it preserves the order of the elements.

```
def doublePos(ns: List[Int]): List[Int]  = ns match {
    case Nil        => Nil
    case Cons(x,xs) =>
        if (0 < x)
            Cons(2*x, doublePos(xs))
        else
            doublePos(xs)
}
```

We can abstract the pattern of computation common to `getEven` and `doublePos` as the (broadly useful) function `filter`, defined as follows:

```
def filter[A](ls: List[A])(p: A => Boolean): List[A] =
    ls match {
        case Nil        => Nil
        case Cons(x,xs) =>
            val fs = filter(xs)(p)
            if (p(x)) Cons(x,fs) else fs
    }
```

Function `filter` takes a predicate `p` of type `A => Boolean` a list of type `List[A]` and returns a list containing those elements that satisfy `p`, in the same order as the input list.

Therefore, we can redefine `getEven` and `doublePos` as follows:

```
def getEven2(ns: List[Int]): List[Int] =
    filter(ns)(x => x % 2 == 0)

def doublePos2(ns: List[Int]): List[Int] =
    map(filter(ns)(x => 0 < x))(y => 2 * y)
```

Function `doublePos2` exhibits both the `filter` and the `map` patterns of computation.

The higher-order functions `map` and `filter` allowed us to restate the definitions of `getEven` and `doublePos` in a succinct form.

We can implement `filter` in terms of `foldRight` as follows:

```
def filter1[A](ls: List[A])(p: A => Boolean): List[A] =
    foldRight(ls, Nil:List[A])((x,xs) => if (p(x)) Cons(x,xs) else xs)
```

Above, the folding function

```
(x,xs) => if (p(x)) Cons(x,xs) else xs
```

applies the filter predicate `p` to the next element of the list (moving right to left). If the predicate evaluates to true, the folding function attaches that element on the front of the processed tail; otherwise, it omits the element from the result.

3.5.5 Flat map

The higher-order function `map` applies its function argument `f` to every element of a list and returns the list of results. If the function argument `f` returns a list, then the result is a list of lists. Often we wish to flatten this into a single list, that is, apply a function like `concat` defined in Section 3.5.1.

This computation is sufficiently common that we give it the name `flatMap`. We can define it in terms of `map` and `concat` as

```
def flatMap[A,B](ls: List[A])(f: A => List[B]): List[B] =
    concat(map(ls)(f))
```

or by combining `map` and `concat` into one `foldRight` as:

```
def flatMap1[A,B](ls: List[A])(f: A => List[B]): List[B] =
    foldRight(ls, Nil: List[B])(
        (x: A, ys: List[B]) => append(f(x),ys))
```

Above, the function argument to `foldRight` applies the `flatMap` function argument `f` to each element of the list argument and then appends the resulting list in front of the result from processing the elements to the right.

We can also define `filter` in terms of `flatMap` as follows:

```
def filter2[A](ls: List[A])(p: A => Boolean): List[A] =
    flatMap(ls)(x => if (p(x)) List(x) else Nil)
```

The function argument to `flatMap` generates a one-element list if the filter predicate `p` is true and an empty list if it is false.

3.6 Classic Algorithms on Lists

3.6.1 Insertion sort and bounded generics

Consider a function to sort the elements of a list into ascending order. A simple algorithm to do this is *insertion sort*. To sort a non-empty list with head `x` and tail `xs`, sort the tail `xs` and insert the element `x` at the right position in the result. To sort an empty list, just return it.

If we restrict the function to integer lists, we get the following Scala functions:

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

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
    case Nil        => List(x)
    case Cons(y,ys) =>
        if (x <= y)
            Cons(x,xs)
        else
            Cons(y,insert(x,ys))
}
```

Insertion sort has a (worst and average case) time complexity of O(`n^2`) where `n` is the length of the input list. (Function `isort` requires n consecutive recursive calls; each call uses function `insert` which itself requires on the order of n recursive calls.)

Now suppose we want to generalize the sorting function and make it polymorphic. We cannot just add a type parameter `A` and substitute it for `Int` everywhere. Although all Scala data types support equality and inequality comparison, not all types can be compared on a *total ordering* (<, <=, >, and >= as well).

Fortunately, the Scala library provides a trait `Ordered`. Any class that provides the other comparisons can extend this trait; the standard types in the library do so. This trait adds the comparison operators as methods so that they can be called in infix form.

```
trait Ordered[A] {
    def compare(that: A): Int
    def < (that: A): Boolean = (this compare that) <  0
    def > (that: A): Boolean = (this compare that) >  0
    def <=(that: A): Boolean = (this compare that) <= 0
    def >=(that: A): Boolean = (this compare that) >= 0
    def compareTo(that: a) = compare(that)
}
```

We thus need to restrict the polymorphism on `A` to be a subtype of `Ordered[A]` by putting an *upper bound* on the type as follows:

```
def isort[A <: Ordered[A]](ls: List[A]): List[A]
```

Note: In addition to upper bounds, we can use a *lower bound*. A constraint `A :> T` requires type `A` to be a supertype of type `T`. We can also specify both an upper and a lower bound on a type such as `T1 <: A <: T2`,

By using the upper bound constraint, we can sort data from any type that extends `Ordered`. However, the primitive types inherited from Java do not extend `Ordered`.

Fortunately, the Scala library defines implicit conversions between the Java primitive types and Scala's enriched wrapper types. (This is the "type class" mechanism we discussed earlier.) We can use a weaker *view bound* constraint,

denoted by <% instead of <:. This A to be any type that is a subtype of or convertible to Ordered[A].

```
def isort1[A <% Ordered[A]](ls: List[A]): List[A] =
    ls match {
        case Nil        => Nil
        case Cons(x,xs) => insert1(x,isort1(xs))
    }

def insert1[A <% Ordered[A]](x: A, xs: List[A]): List[A] =
    xs match {
        case Nil        => List(x)
        case Cons(y,ys) =>
            if (x <= y)
                Cons(x,xs)
            else
                Cons(y,insert1(x,ys))
    }
```

We could define insert inside isort and avoid the separate type parameterization. But insert is separately useful, so it is reasonable to leave it external.

An alternative to use of the bound would be to pass in the needed comparison predicate, as follows:

```
def isort2[A](ls: List[A])(leq: (A,A) => Boolean): List[A] =
    ls match {
        case Nil        => Nil
        case Cons(x,xs) => insert2(x,isort2(xs)(leq))(leq)
    }

def insert2[A](x:A, xs:List[A])(leq:(A,A)=>Boolean):List[A] =
    xs match {
        case Nil        => List(x)
        case Cons(y,ys) =>
            if (leq(x,y))
                Cons(x,xs)
            else
                Cons(y,insert2(x,ys)(leq))
    }
```

Above we expressed both functions in curried form. By putting the comparison function last, we enabled the compiler to infer the argument types for the function.

If we placed the function in the first argument group, the user of the function would have to supply the types. However, putting the comparison function first might allow a more useful partial application of the isort to a comparison function.

3.6.2 Merge sort

The insertion sort given in Section 3.6.2 has an average case time complexity of O(`n^2`) where `n` is the length of the input list.

We now consider a more efficient function to sort the elements of a list: *merge sort*. Merge sort works as follows:

- If the list has fewer than two elements, then it is already sorted.
- If the list has two or more elements, then we split it into two sublists, each with about half the elements, and sort each recursively.
- We merge the two ascending sublists into an ascending list.

For a general implementation, we specify the type of list elements and the function to be used for the comparison of elements, giving the following implementation:

```
def msort[A](less: (A, A) => Boolean)(ls: List[A]): List [A] =
{
    def merge(as: List[A], bs: List[A]): List[A] =
        (as,bs) match {
            case (Nil,_)                => bs
            case (_,Nil)                => as
            case (Cons(x,xs),Cons(y,ys)) =>
                if (less(x,y))
                    Cons(x,merge(xs,bs))
                else
                    Cons(y,merge(as,ys))
    }

    val n = length(ls)/2
    if (n == 0)
        ls
    else
        merge(msort(less)(take(ls,n)),
              msort(less)(drop(ls,n)))
}
```

The `merge` forms a tuple of the two lists and does pattern matching against that tuple. This allowed the pattern match to be expressed more symmetrically.

The above function uses a function we have not yet defined.

```
def take[A](ls: List[A], n: Int): List[A]
```

returns the first `n` elements of the list; it is the dual of `drop`.

By nesting the definition of `merge`, we enabled it to directly access the parameters of `msort`. In particular, we did not need to pass the comparison function to `merge`.

The average case time complexity of `msort` is O(`n * (log2 n)`), where `n` is the length of the input list. (Here `log2` denotes a function that computes logarithms base 2.)

- Each call level requires splitting of the list in half and merging of the two sorted lists. This takes time proportional to the length of the list argument.
- Each call of `msort` for lists longer than one results in two recursive calls of `msort`.
- But each successive call of `msort` halves the number of elements in its input, so there are O(`log2 n`) recursive calls.

So the total cost is O(`n * (log2 n)`). The cost is independent of distribution of elements in the original list.

We can apply `msort` as follows:

```
msort((x: Int, y: Int) => x < y)(List(5, 7, 1, 3))
```

We defined `msort` in curried form with the comparison function first (unlike what we did with `isort1`). This enables us to conveniently specialize `msort` with a specific comparison function. For example,

```
val intSort     = msort((x: Int, y: Int) => x < y) _
val descendSort = msort((x: Int, y: Int) => x > y) _
```

However, we do have to give explicit type annotations for the parameters of the comparison function.

3.7 Lists in the Scala Standard Library

In this discussion (and in Chapter 3 of *Functional Programming in Scala* [4]), we developed several functions for a simple `List` module. Our module is related to the builtin Scala `List` module (from `scala.collection.immutable`), but it differs in several ways.

Our `List` module is standalone module; the Scala `List` inherits from an abstract class with several traits mixed in. These classes and traits structure the interfaces shared among several data structures in the Scala library. Many of the functions work for different data structures. For example, in Scala release 2.12.8 `List` is defined as follows:

```
sealed abstract class List[+A] extends AbstractSeq[A]
    with LinearSeq[A]
    with Product
    with GenericTraversableTemplate[A, List]
    with LinearSeqOptimized[A, List[A]]
    with scala.Serializable
```

Our `List` module consists of functions in which all arguments must be given explicitly; the Scala `List` consists of methods on the `List` class. Scala enables

methods with one implicit argument (i.e., `this`) and one explicit argument to be called as infix operators with different associativities. It allows symbols such as < to be used for method names.

Scala's approach to functional programming uses *method chaining* in its object system to support composition of pure functions. Each method returns an immutable object that becomes the receiver of the subsequent method call in the same statement.

Extensive use of method chaining in an object-oriented program with mutable objects—sometimes called a *train wreck*—can make programs difficult to understand. However, disciplined use of method chaining helps make the functional and object-oriented aspects of Scala work together. (In different ways, method chaining is also useful in development of fluent library interfaces for domain-specific languages.)

Our `Cons(x,xs)` is written as `x :: xs` using the standard Scala library. The `::` is a method that has one implicit argument (the tail list) and one explicit argument (the head element).

Any Scala method name that ends with a `:` is right associative. Thus method `x :: xs` represents the method call `xs.::(x)`, which in turn calls the data constructor. We can write `x :: y :: z :: zs` without parentheses to mean `x :: (y :: (z :: zs))`.

We can also use multiple `::` constructors in cases for pattern matching. For example, where we wrote the pattern

```
case Cons(x, Cons(y,ys))
```

in the `remdups` function, we can write the pattern:

```
case x :: y :: ys
```

Our `append` function is normally written with the infix operator ++ in the Scala library. (But there are several variations for special circumstances.)

Several of our functions with a single list parameter may appear as parameterless methods with the same name in the Scala library. These include `sum`, `product`, `reverse`, and `length`. There is also a `head` function to retrieve the head element of a nonempty list.

Our `concat` function is parameterless method `flatten` in the Scala library.

Our functions with two parameters, a list and a modifier, are one-parameter methods with the same name in the Scala library, and, hence, usable as infix operators. These include `drop`, `dropWhile`, `map`, `filter`, and `flatMap`. There are also analogous functions `take` and `takeWhile`.

Our functions `foldRight` and `foldLeft`, which have three parameters, are methods in the Scala library with two curried parameters. The list argument

becomes implicit; the other arguments are in the same order. The Scala library contains several folding and reducing functions with related functionality.

Other than `head`, `take`, `takeWhile`, and the appending and folding methods mentioned above, the Scala List library has other useful methods such as `forall`, `exists`, `scanLeft`, `scanRight`, `zip`, and `zipWith`.

Check out the Scala API documentation on the Scala website [32].

3.8 Source Code for Chapter

The Scala source code files for the functions in this chapter (3) are as follows:

- `List2.scala` for the
- `TestList2.scala` for testing code

3.9 Exercise Set A

In the following exercises, extend the `List2.scala` algebraic data type implementation developed in these notes to add the following functions. In the descriptions below, type `List` refers to the trait defined in that package, not the standard Scala list.

1. Write a Scala function `orList` that takes a `List` of `Boolean` values and returns the logical `or` of the values (i.e., true if any are true, otherwise false).
2. Write a Scala function `andList` that takes a `List` of `Boolean` values and returns the logical `and` of the values (i.e., true if all are true, otherwise false).
3. Write a Scala function `maxList` that takes a nonempty `List` of values and returns its maximum value.

   Hint: First solve this with `Int`, then generalize to a generic type. Study insertion sort in Section {#sec:Scala-insertion-sort}.
4. Write a Scala `remdups1` that is like `remdups` except that it is implemented using either `foldRight` or `foldLeft`.
5. Write a Scala function `total` that takes a nonnegative integer `n` and a function `f` of an appropriate type and returns `f(0) + f(1) + ... f(n)`.
6. Write a Scala function `flip` that takes a function of polymorphic type `(A,B) => C` and returns a function of type `(B,A) => C` such that, for all `x` and `y`:

   ```
   f(x,y) == flip(f)(y,x)
   ```
7. Write the following Scala functions using tail recursive definitions:

   a. `sumT` with same functionality as `sum`

b. `productT` with the same functionality as `product`

8. Write a Scala function `mean` that takes a nonempty `List` of `Double` values and returns its mean (i.e., average) value.

9. Write a Scala function `adjPairs` that takes a `List` of pairs (i.e., two-tuples) and returns the list of all pairs of adjacent elements. For example, `adjPairs(List(2,1,11,4))` returns `List((2,1),(1,11),(11,4))`.

10. Write a Scala function `splitAt` that takes a `List` of values and an integer `n` and returns a pair (i.e., two tuple) of `List`s, where the first component consists of the first `n` elements of the input list (in order) and the second component consists of the remaining elements (in order).

11. Number base conversion.

    a. Write a Scala function `natToBin` that takes a natural number and returns its binary representation as a `List` of `0`'s and `1`'s with the most significant digit at the head. For example, `natToBin(23)` returns `List(1,0,1,1,1)`.

    In computer science, we usually consider 0 as natural number along with the positive integers.

    b. Generalize `natToBin` to Scala function `natToBase` that takes base `b` (`b >= 2`) as its first paramenter and the natural number as its second. The function should return the base `b` representation of the natural number as a list of integer digits with the most significant digit at the head. For example, `natToBase(5,42)` returns `List(1,3,2)`.

    c. Write Scala function `baseToNat` that is the inverse of the `natToBase` function. For any base `b` (`b >= 2`) and natural number `n`:

    ```
    baseToNat(b,natToBase(b,n)) == n
    ```

12. For each of the following specifications, write a Scala function that has the given arguments and result. Use the higher functions from the `List` algebraic data type from these notes, such as `map`, `filter`, `foldRight`, and `foldLeft`, as appropriate.

    a. Function `numof` takes a value and a list and returns the number of occurrences of the value in the list.

    b. Function `ellen` takes a list of lists and returns a list of the lengths of the corresponding lists.

    c. Function `ssp` takes a list of integers and returns the sum of the squares of the positive elements of the list.

13. Write a Scala function `scalarProd` with type

    ```
    (List[Int],List[Int]):: Int
    ```

to compute the scalar product of two lists of integers (e.g., representing vectors).

The *scalar product* is the sum of the products of the elements in corresponding positions in the lists. That is, the scalar product of two lists `xs` and `ys`, of length `n`, is:

$$\sum_{i=0}^{i=n} xs_i * ys_i$$

For example, `scalarprod(List(1,2,3),List(3,3,3))` yields `18`.

14. Write a Scala function `mapper` that takes a list of functions and a list of values and returns the list of results of applying each function in the first list to the corresponding value in the second list.

15. Write a Scala function `removeFirst` that takes a predicate (i.e., Boolean function) and a list of values and returns the list with the first element that satisfies the predicate removed.

16. Define a Scala function `removeLast` that takes a predicate (i.e., Boolean function) and a list of values and returns the list with the last element that satisfies the predicate removed.

    How could you define it using `removeFirst`?

3.10 General Tree Algebraic Data Type

3.10.1 Description

A *general tree* is a hierarchical data structure in which each node of the tree has zero or more subtrees. We can define this as a Scala algebraic data type as follows:

```
sealed trait GTree[+A]
case class Leaf[+A](value: A) extends GTree[A]
case class Gnode[+A](gnode: List[GTree[A]]) extends GTree[A]
```

An object of class `Leaf(x)` represents a *leaf* of the tree holding some value `x` of generic type `A`. A leaf does not have any subtrees. It has height 1.

An object of type `Gnode` represents an *internal* (i.e., non-leaf) node of the tree. It consists of a nonempty list of subtrees, ordered left-to-right. A `Gnode` has a height that is one more than the maximum height of its subtrees.

3.10.2 Exercise Set B

In the following exercises, write the Scala functions to operate on the `GTree`s. You may use functions from the extended `List` module as needed.

1. Write Scala function `numLeavesGT` that takes a `GTree` and returns the count of its leaves.

2. Write Scala function `heightGT` that takes a `GTree` and returns its height (i.e., the number of levels).
3. Write Scala function `sumGT` that takes a `GTree` of integers and returns the sum of the values.
4. Write Scala function `findGT` that takes a `GTree` and a value and returns `true` if and only if the element appears in some leaf in the tree.
5. Write Scala function `mapGT` that takes a `GTree` and a function and returns a `GTree` with the structure but with the function applied to all the values in the tree.
6. Write Scala function `flattenGT` that takes a `GTree` and returns a `List` with the values from the tree leaves ordered according to a left-to-right traversal of the tree.

3.11 Acknowledgements

In Spring 2016, I wrote this set of notes to accompany my lectures on Chapter 3 of the book *Functional Programming in Scala* [4] (i.e., the Red Book) for my Scala-based, Spring 2016 version of CSci 555 (Functional Programming) at the University of Mississippi. I constructed the notes around the ideas, general structure, and Scala examples from the Red Book and its associated materials [5,6]. I also adapted some discussion and examples from my Haskell-based *Notes on Functional Programming with Haskell* [8], Odersky's *Scala by Example* [24], and my previous notes on teaching mulitparadigm programming using Scala.

The Red Book [4] is an excellent book for self-study, but it needed expanded explanations to be useful to me as a textbook for students who were novices in both Scala and functional programming.

I expanded the Red Book's discussion of algebraic data types, polymorphism, and variance. For this expansion, I examined other materials including the Wikipedia articles on Abstract Data Type [37], Algebraic Data Type [38], Polymorphism [43], Ad Hoc Polymorphism [45], Parametric Polymorphism [46], Subtyping [47], Liskov Substitution Principle [42], Function Overloading [44], and Covariance and Contravariance (computer science) [39]. I also examined the discussion of variance in the Wampler textbook [36]. I adapted the sorting algorithms from *Scala by Example* [24].

In 2018 and 2019, I updated the format to be more compatible with my evolving document structures for instructional materials (e.g., such as the textbool *Exploring Languages with Interpreters and Functional Programming* (ELIFP) [16]).

In Spring 2019, I also moved the discussion of the kinds of polymorphism to the new notes on Type System Concepts [11], expanded the discussion of Variance, and added two exercise sets. Several items from Exercise Set A were adapted from the list processing chapters of ELIFP [16].

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on the ELIFP textbook and other instructional materials. In January 2022, I began refining the ELIFP content and related documents such as this one. I am integrating separately developed materials better, reformatting the documents (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

3.12 Terms and Concepts

TODO: Update

Function, pure function, referential transparency, side effects, mutable, immutable, list data type (head, tail, empty), algebraic data type (composite, sum, product, enumerated), abstract data type, ADT, syntax, semantics, `trait`, `sealed trait`, `case class`, `case object`, singleton object, polymorphism, subtyping, parametric polymorphism, generics, overloading, type classes, variance (covariant, contravariant, invariant/nonvariant), following types to implementations (type-driven or type-first development).

4 Handling Errors without Exceptions

4.1 Introduction

A benefit of Java or Scala exception handling (i.e., using `try-catch` blocks) is that it consolidates error handling to well-defined places in the code.

However, code that throws exceptions typically exhibits two problems for functional programming.

1. It is not *referentially transparent*. It has side effects. Its meaning is thus dependent upon the context in which it is executed.
2. It is not *type safe*. Exceptions may cause effects that are of a different type than the return value of the function.

The key idea from Chapter 4 *is to use ordinary data values to represent failures and exceptions* in programs. This preserves referential transparency and type safety while also preserving the benefit of exception-handling mechanisms, that is, the consolidation of error-handling logic.

To do this, we introduce the `Option` and `Either` algebraic data types. These are standard types in the Scala library, but for pedagogical purposes Chapter 4 introduces its own definition that is similar to the standard one.

This set of notes also introduces Scala features that we have not previously discussed extensively: type variance, call-by-name parameter passing, and for-comprehensions.

4.2 Aside: On Null References

What should a function do when it is required to return an object but no suitable object exists?

One common approach is to return a *null reference*. British computing scientist Tony Hoare, who introduced the null reference into the Algol type system in the mid-1960s, calls that his "billion dollar mistake" because it "has led to innumerable errors, vulnerabilities, and system crashes" [20].

The software patterns community documented a safer approach to this situation by identifying the *Null Object* design pattern [51,54]. This well-known object-oriented pattern seeks to "encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior" [34]. That is, the object must be of the correct type. It must be possible to apply all operations on that type to the object, but the operations should have neutral behaviors, with no side effects. The null object should actively do nothing!

The functional programming community introduced *option types* [52] as an approach to the same general problem. For example, Haskell defines the `Maybe` and `Either` types [16,35]. More recently designed functional languages—such as Idris [1,2], Elm [17,18], and PureScript [19,28]—include similar types.

Regardless of language paradigm, most other recently designed languages have option types or similar features. As we discuss in this chapter, the hybrid object-oriented/functional language Scala supports the `Option[T]` and `Either` case classes that correspond to Haskell's `Maybe` and `Either` types. Similarly, Rust [22,31] has an `Option` type and Swift has an `Optional` type that are similar to Haskell's `Maybe`.

The concept of *nullable type* [50] is closely related to the option type. Several older languages support this concept (e.g.,`Optional` in Java 8, `None` in Python [29,30], and `?` type annotations in C#).

4.3 An `Option` Algebraic Data Type

We define a Scala algebraic data type `Option` using sealed trait `Option` with case class `Some` to wrap a value and case object `None` to denote an empty value. We specify the operations on the data type using the *method chaining* style.

```
import scala.{Option => _, Either => _, _} // hide standard
sealed trait Option[+A] {
    def map[B](f: A => B): Option[B]
    def getOrElse[B >: A](default: => B): B
    def flatMap[B](f: A => Option[B]): Option[B]
    def orElse[B >: A](ob: => Option[B]): Option[B]
    def filter(f: A => Boolean): Option[A]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

The `import` statement above hides the standard definitions of `Option` and `Either` from the Scala standard library. The versions developed in this chapter are similar, but add useful features above what is available in the standard library.

Before we move on to the definition of these methods, let's review the concept of method chaining and then consider issues raised in the signatures of the `getOrElse` and `orElse` methods: one related to the generic parameters and the other to the `default` parameters.

4.3.1 Method chaining in Scala

Consider function `map` in the `Option` trait. It is implemented in this chapter as a method on the classes that extend the `Option` trait.

Method `map` takes two arguments and returns a result object. Its implicit "left" argument is the receiver object for the method call, an `Option` object represented internally by the variable `this`. Its explicit "right" argument is a function. The result returned is itself an `Option` object.

Suppose `obj` is an `Option[A]` object and `f` is an `A => B` function. In standard object-oriented style, we can issue the call `obj.map(f)` to operate on object `obj`

with method `map` and argument `f`. Scala allows us to apply such a method in an *infix operator* style as follows:

```
obj map f
```

Note that `map` takes an `Option[A]` as its left operand (i.e., argument) and returns an `Option[B]` object as its result. Thus we can *chain* such method calls as follows (where functions `p` and `g` have the appropriate types):

```
obj.map(f).filter(p).map(g)
```

In Scala's infix operator style, this would be:

```
(obj map f) filter p)) map g
```

But these operators associate to the left, so can leave off the parentheses and write the above as follows

```
obj map f filter p map g
```

or perhaps more readably as:

```
obj map(f) filter(p) map(g)
```

Note that this last way of writing the chain suggests the *data flow* nature of this computation. The data originates with the source `obj`, which is then transformed by a `map(f)` operator, then by a `filter(p)` operator, and then by a `map(g)` operator to give the final result.

Now let's consider an issue raised in the signatures of the `getOrElse` and `orElse` methods related to the generic parameter.

4.3.2 Type variance issues

As we have discussed previously, the `+A` annotation in the `Option[+A]` definition declares that parameter `A` is *covariant*. That is, if `S` is a subtype of `T`, then `Option[S]` is a subtype of `Option[T]`. Also remember that `Nothing` is a subtype of all other types.

For example, suppose type `Beagle` is a subtype of type `Dog`, which, in turn, is a subtype of type `Mammal`. Then, because of the covariant definition, `Option[Beagle]` is a subtype of `Option[Dog]`, which is a subtype of `Option[Mammal]`. This is intuitively what we expect.

However, in `getOrElse` and `orElse`, we use type constraint `B >: A`. This means that `B` must be equal to `A` or a supertype of `A`. We also define value parameter of these functions to have type `Option[B]` instead of `Option[A]`.

Why must we have this constraint?

See the chapter notes [6] for Chapter 4 of the *Functional Programming in Scala* book [4] for more detail on this complicated issue. We sketch the argument below.

In some sense, the `+A` annotation declares that, in all contexts, it is safe to cast this type `A` to some supertype of `A`. The Scala compiler does not allow us to use this annotation unless we can cast all members of a type safely.

Suppose we declare `orElse` (incorrectly!) as follows:

```
trait Option[+A] {
    def orElse(o: Option[A]): Option[A]
    ...
}
```

We have a problem because this declaration of `orElse` only allows us to cast `A` to a subtype of `A`.

Why?

As with any function, `orElse` can only be safely passed a subtype of its declared input type. That is, a function of type `Dog => R` can be passed an object of subtype like `Beagle` or of type `Dog` itself, but it cannot be passed a supertype object of `Dog` such as `Mammal`.

As we saw in the notes on Chapter 3 [12], Scala functions are contravariant in their input types.

But `orElse` has a parameter type of `Option[A]`. Because of the covariance of `A` (declared in the trait), this parameter only allows subtypes of `A`—not supertypes as required by the covariance.

So, we have a contradiction.

For the incorrect signature of `orElse`, the Scala compiler generates an error message such as "Covariant type `A` occurs in contravariant position."

We can get around this error by using a more complicated signature that does not mention `A` (declared in the trait) in any of the function arguments, such as:

```
trait Option[+A] {
    def orElse[B >: A](o: Option[B]): Option[B]
    ...
}
```

Now consider the second new feature appearing in the signatures of `getOrElse` and `orElse`—the `=> B` and `=> Option[B]` types for the parameters.

4.3.3 Parameter-passing modes

Scala's primary parameter-passing mode is *call by value* as in Java and C. That is, the program evaluates the caller's argument and the resulting value is bound to the corresponding formal parameter within the called function.

If the argument's value is of a primitive type, then the value itself is passed. If the value is an object, then the address of (i.e., reference to) the object is passed.

A call-by-value parameter is called *strict* because the called function always requires that parameter's value before it can execute. The corresponding argument must be evaluated *eagerly* before transferring control to the called function.

Scala also has *call-by-name* parameter passing. Consider the `default: => B` feature in the declaration

```
def getOrElse[B >: A](default: => B): B
```

The type notation `=> B` means the calling program leaves the argument of type `B` unevaluated. That is, the calling program wraps the argument expression in an parameterless function and passes the function to the called method. This automatically generated parameterless function is sometimes called a *thunk*.

Every reference to the corresponding parameter causes the thunk to be evaluated. If the method does not access the corresponding parameter during some execution, then the parameter is never evaluated.

As with all higher-order arguments in Scala, a thunk is passed as a *closure*. In addition to the function, the closure captures any *free variables* occurring in the expression—that is, the variables defined in the caller's environment but not within the expression itself.

Note: The closure actually captures the variable itself, not its value. So if the free variable is either a reference to a `var` or to a mutable data structure, then changes in the value are seen inside the called function. But in this course, we normally use immutable data structures and `val` declarations.

A call-by-name parameter is called *nonstrict* because the called function does not always require that parameter's value for its execution. The corresponding argument can thus be evaluated *lazily*, that is, evaluated only if and when its value is needed.

We look at more implications of strict and nonstrict functions in Chapter 5 of the Red Book [4].

Note: See `ClosureExample.scala` and `ThunkExample.scala` for examples of using closures and thunks, respectively.

4.3.4 Implementing the `Option` methods

Now, let's define the methods in the `Option` data type.

Above we defined the trait `Option` as follows. It is parameterized by covariant type `A`.

```
import scala.{Option => _, Either => _, _} // hide standard
sealed trait Option[+A] {
    def map[B](f: A => B): Option[B]
    def getOrElse[B >: A](default: => B): B
    def flatMap[B](f: A => Option[B]): Option[B]
```

```
    def orElse[B >: A](ob: => Option[B]): Option[B]
    def filter(f: A => Boolean): Option[A]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

The `Option` data type is similar to a list that is either empty or has one element. As with the `List` algebraic data type from Chapter 3, we *follow the types to implementations*. (This is sometimes called *type-driven development*.) That is, we use the form of the data to guide us to design the form of the function.

The `map` method applies a function to its implicit `Option[A]` argument. If the implicit argument is a `Some`, the method applies the function to the wrapped value and returns the resulting `Some`. If it is a `None`, the method just returns `None`. We can implement `map` using pattern matching directly as follows:

```
def map[B](f: A => B): Option[B] = this match {
    case None    => None
    case Some(a) => Some(f(a))
}
```

Similarly, we can use pattern matching directly to implement the `getOrElse` function. If the implicit argument is of type `Some`, this function returns the value it wraps. If the implicit argument is of type `None`, this function returns the value denoted by the `default` argument. By passing `default` by name, the argument is only evaluated when its value is needed.

```
def getOrElse[B >: A](default: => B): B = this match {
    case None    => default // evaluate the thunk
    case Some(a) => a
}
```

Function `flatMap` applies its argument function `f`, which might fail, to its implicit `Option` argument when this value is not `None`. We can define `flatMap` in terms of `map` and `getOrElse` as shown below. (Reminder: If we apply method names as operators in an infix manner, they associate to the left. The leftmost operator implicitly operates on `this` object.)

```
def flatMap[B](f: A => Option[B]): Option[B] =
    map(f) getOrElse None
```

We can also define `flatMap` using pattern matching directly.

```
def flatMap_1[B](f: A => Option[B]): Option[B] = this match {
   case None    => None
   case Some(a) => f(a)
}
```

Function `orElse` returns the implicit `Option` argument if is not `None`; otherwise, it returns the explicit `Option` argument. We can define `orElse` in terms of `map` and `getOrElse` or by directly using pattern matching.

```
def orElse[B >: A](ob: => Option[B]): Option[B] =
    this map (Some(_)) getOrElse ob

def orElse_1[B>:A](ob: => Option[B]): Option[B] =
    this match {
        case None => ob
        case _    => this
    }
```

The `filter` function converts its implicit argument from `Some` to `None` if it does not satisfy the boolean function `p`. We can define `filter` by using pattern matching directly or by using `flatMap`.

```
def filter(p: A => Boolean): Option[A] =
    this match {
        case Some(a) if p(a) => this
        case _               => None
    }

def filter_1(p: A => Boolean): Option[A] =
    flatMap(a => if (p(a)) Some(a) else None)
```

4.3.5 Using `Option` for statistical `mean` and `variance`

Consider a function to calculate and return the *mean* (i.e., average value) of a list of numbers. This function must sum the list of numbers and divide by the number of elements. It might have a signature such as:

```
def mean(xs: List[Double]): Double
```

But what should be returned for empty lists?

We can modify the signature to use `Option` and define the function as follows:

```
def mean(xs: Seq[Double]): Option[Double] =
    if (xs.isEmpty)
        None
    else
        Some(xs.sum / xs.length)
```

The return type now allows the possibility that the mean may be undefined. We thus extend a partial function to a total function in a meaningful way.

Above we also generalize the `List` type to its supertype `Seq` from the Scala library. Type `Seq` denotes a family of sequential data types that includes the `List` type, array-like collections, etc. This type defines the methods `isEmpty`, `sum`, and `length`.

If the mean of a sequence s is m, then the (statistical) *variance* of the sequence s is the mean of the sequence formed by the terms $(x - m)^2$ for each $x \in s$

(perserving the order).

Using the `mean` function defined above, we can compute the variance of a sequence of numbers as follows:

```
def variance(xs: Seq[Double]): Option[Double] =
    mean(xs) flatMap
        (m => mean(xs.map(x => math.pow(x - m, 2))))
```

4.3.6 Using `Option` in the labelled digraph

In the doubly labelled `Digraph` case study, we define the following method to return the label on a vertex:

```
def getVertexLabel(ov: A): B
```

In the case study's `DigraphList` implementation, we define this function as follows. The function terminates with an error when the the argument vertex is not present in the digraph.

```
def getVertexLabel(ov: A): B =
    (vs dropWhile (w => w._1 != ov)) match {
        case Nil =>
            sys.error("Vertex " + ov + " not in digraph")
        case ((_,l)::_) => l
    }
```

We can avoid the error termination in this function by changing the method signature to return an `Option[B]` instead of a `B`.

```
def getVertexLabelOption(ov: A): Option[B] =
    (vs dropWhile (w => w._1 != ov)) match {
        case Nil        => None
        case ((_,l)::_) => Some(l)
}
```

Code that uses this new `Digraph` method can call the various `Option` methods (or directly use pattern matching) to process the result appropriately.

For example, if the vertex label is a string, it may be appropriate in some scenarios to just use a null string for the label of a nonexistent vertex. Let `g` be a `Digraph` and be `v` be a possible vertex, then

```
(g getVertexLabelOption v) getOrElse ""
```

would either return the label string for `v` if it exists or the null string if `v` does not exist.

Similarly, the code

```
(g getVertexLabelOption v) getOrElse
    (sys.error("undefined vertex " + v))
```

would still terminate with an error. However, this design enables the user of the `Digraph` library to decide under what circumstances and at what point in the code to terminate.

Idiom: A common pattern for computing with the `Option` type is to use `map`, `flatMap`, and `filter` to transform values generated by a function like `getVertexLabelOption` and then to use `getOrElse` for error handling at the end.

4.3.7 Lifting

It seems that that deciding to use of `Option` could cause lots of changes to ripple through our code, much like the introduction of extensive exception-handling would. However, we can avoid that somewhat by using a technique called *lifting*.

For example, the `Option` type's `map` function enables us to transform values of type `Option[A]` into values of type `Option[B]` using a function of type `A => B`.

Alternatively, we could consider `map` as transforming a function of type `A => B` into a function of type `Option[A] => Option[B]`. That is, we *lift* an ordinary function into a function on `Option`.

We can formalize this alternative view with the following function:

```
def lift[A,B](f: A => B): Option[A] => Option[B] = _.map(f)
```

Thus any existing function can be transformed to work within the context of a single `Option` value. For example, we can lift the square root function from type `Double` to work with `Option[Double]` as follows:

```
def sqrtO: Option[Double] => Option[Double] =
    lift(math.sqrt _)<
```

We can now use `sqrtO` such as `sqrtO(Some(4))`. This evaluates to `Some(2)`.

Chapter 4 of *Functional Programming in Scala* [4] gives several examples where `Option` types can be used effectively in realistic scenarios. One of these examples illustrates how to wrap an exception-oriented API to provide users with `Option` results in error situations.

TODO: Add example of wrapping an exception-oriented API.

Note: See `WrapException.scala` for examples of how to wrap exception-throwing functions to return `Option` and `Either` objects, respectively.

4.3.8 For comprehensions

Another useful function on `Option` data types is the function `map2` that combines two `Option` values by lifting a binary function. If either of the arguments are `None`, then the result should also be `None`. We can define `map2` as follows:

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C):
    Option[C] =
        a flatMap (aa =>
            b map (bb =>
                f(aa, bb)))
```

This function applies a series of `map` and `flatMap` calls.

Because lifting is so common in functional programming, Scala provides a syntactic construct called a *for-comprehension* to facilitate its use. This construct is really *syntactic sugar* for a series of applications of `map`, `flatMap`, and `withFilter`. (Method `withFilter` works like `filter` except it filters on demand, without creating a new collection as a result.)

Here is the same code expressed as a for-comprehension:

```
def map2fc[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C):
    Option[C] =
        for {
            aa <- a
            bb <- b
        } yield f(aa, bb)
```

Of course, for-comprehensions are more general than just their use with `Option`. They can be used for the lists, arrays, iterators, ranges, streams, and other types in the Scala standard library that support the `map`, `flatMap`, and `withFilter` (or `filter`) operations.

Consider a list `persons` of person objects with `name` and `age` fields. We can collect the names of all persons who are age 21 and above as follows:

```
for (p <- persons if p.age >= 21) yield p.name
```

This is equivalent to the following `List` expression

```
filter (p => p.age >= 21) map (p => p.name)
```

4.3.9 Translating (desugaring) for-comprehensions

In general, a for-comprehension

```
for (enums) yield e
```

evaluates expression `e` for each binding generated by the enumerator sequence `enums` and collects the results. An enumerator sequence begins with a generator and may be followed by additional generators, value definitions, and guards.

- A *generator* `p <- e` produces a sequence of zero or more bindings from expression `e` by matching each value against pattern `p`.
- A *value definition* `p = e` binds the names in pattern `p` to the result of evaluating the expression `e`.

- A *guard* `if e` restricts the bindings to those that satisfy the boolean expression `e`.

We can translate (or *desugar*) for-comprehension (more or less) as follows:

1. We replace every generator `p <- e`, where `p` is a pattern and `e` is an expression, by:

   ```
   p <- e.withFilter { case p => true ; case _ => false }
   ```

   Here we use `withFilter` to filter out those items that do not match the pattern `p`.

2. While all comprehension have not been eliminated, repeat the following:

   a. Translate for-comprehension

      ```
      for (p <- e) yield e1
      ```

      to the expression:

      ```
      e.map { case p => e1 }
      ```

   b. Translate for-comprehension

      ```
      for (p <- e; p1 <- e1; ... ) yield e2
      ```

      where `...` is a (possibly empty) sequence of generators, definitions, or guards, to the expression:

      ```
      e.flatMap { case p => for (p1 <- e1; ... ) yield e2 }
      ```

   c. Translate a generator `p <- e` followed by a guard `if g` to a single generator

      ```
      p <- e.withFilter((x1,...,xn) => g)
      ```

      where `x,...,xn` are the free variables of `p`.

   d. Translate a generator `p <- e` followed by a value definition `p1 = e1` to the following generator of pairs of values, where `x` and `x1` are fresh names:

      ```
      (p, p1) <- for (x@p <- e) yield
          { val x1@p1 = e1; (x, x1) }
      ```

The Scala notation `x@p` means that name `x` is bound to the value of the expression `p`.

Note: Above we do not consider the imperative for-loops. These can also be translated as above except that the imperative method `forEach` is also needed.

As an example, we can translate (desugar) the for-comprehension

```
for(x <- e1; y <- e2; z <- e3) yield {...}
```

into the expression:

```
e1.flatMap(x => e2.flatMap(y => e3.map(z => {...})))
```

As a second example, we can also translate the for-comprehension

```
for(x <- e; if p) yield {...}
```

into the expression:

```
e.withFilter(x => p).map(x => {...})
```

If no `withFilter` method is available, we can instead use:

```
e.filter(x => p).map(x => {...})
```

As a third example, we can translate for-comprehension

```
for(x <- e1; y = e2) yield {...}
```

into the expression:

```
e1.map(x => (x, e2)).map((x,y) => {...})
```

4.3.10 Adding for-comprehensions to data types

The Scala language has no typing rules for the for-comprehensions themselves. The Scala compiler first translates for-comprehensions into calls on the various method and then checks the types. It does not require that methods `map`, `flatMap`, and `withFilter` have particular type signatures. However, a particular setup for some collection type `C` with elements of type `A` is the following:

```
trait C[A] {
    def map[B](f: A => B): C[B]
    def flatMap[B](f: A => C[B]): C[B]
    def withFilter(p: A => Boolean): C[A]
}
```

We can define our own data types to support for-comprehension by providing one or more of the required operations above.

- If the data type defines just `map`, Scala allows for-comprehensions consisting of a single generator.
- If the data type defines both `flatMap` and `map`, Scala allows for-comprehensions consisting of several generators.
- If the data type defines `withFilter`, Scala allows for-comprehensions with guards. (If `withFilter` is not defined but `filter` is, Scala will currently use `filter` instead. However, this gives a deprecation warning, so this fallback feature may be eliminated in a future release of Scala.)

We added for-comprehensions to our own `Option` type earlier. We do the same for the `Either` type in the next section.

Note: A for-comprehension is, in general, convenient syntactic sugar for expressing compositions of monadic operators. If time allows, we will discuss monads later in the semester.

4.4 An `Either` Algebraic Data Type

We can use data type `Option` to encode that a failure or exception has occurred. However, it does not give any information about what went wrong.

We can encode this additional information using the algebraic data type `Either`.

```
import scala.{Option => _, Either => _, _} // hide builtin
sealed trait Either[+E,+A] {
    def map[B](f: A => B): Either[E,B]
    def flatMap[EE >: E, B](f: A => Either[EE,B]):
        Either[EE,B]
    def orElse[EE >: E, AA >: A](b: => Either[EE,AA]):
        Either[EE,AA]
    def map2[EE >: E, B, C](b: Either[EE, B])(f: (A,B) => C):
        Either[EE, C]
}
case class Left[+E](get: E) extends Either[E,Nothing]
case class Right[+A](get: A) extends Either[Nothing,A]
```

By convention, we use the constructor `Right` to denote success and constructor `Left` to denote failure.

We can implement `map`, `flatMap`, and `orElse` directly using pattern matching on the `Either` type.

```
def map[B](f: A => B): Either[E, B] =
    this match {
        case Left(e)  => Left(e)
        case Right(a) => Right(f(a))
    }

def flatMap[EE >: E, B](f: A => Either[EE, B]):
    Either[EE, B] = this match {
        case Left(e)  => Left(e)
        case Right(a) => f(a)
    }

def orElse[EE >: E, AA >: A](b: => Either[EE, AA]):
    Either[EE, AA] = this match {
        case Left(_)  => b
        case Right(a) => Right(a)
    }
```

The availability of `flatMap` and `map` enable us to use for-comprehension generators with `Either`. We can thus implement `map2` using a for-comprehension, as follows:

```
def map2[EE >: E, B, C](b: Either[EE, B])(f: (A, B) => C):
    Either[EE, C] =
        for { a <- this; b1 <- b } yield f(a,b1)
```

Let's again use `mean` as an example and use a `String` to describe the failure in `Left`.

```
def mean(xs: IndexedSeq[Double]): Either[String, Double] =
    if (xs.isEmpty)
        Left("mean of empty list!")
    else
        Right(xs.sum / xs.length)
```

We can use the `Left` value to encode more information, such as the location of the error in the program. For example, we might catch and return the value of an `Exception` generated as we do in the `safeDiv` function below.

```
def safeDiv(x: Int, y: Int): Either[Exception, Int] =
    try Right(x / y)
        catch { case e: Exception => Left(e) }
```

We can abstract the computational pattern in the `safeDiv` function as function `Try` defined below:

```
def Try[A](a: => A): Either[Exception, A] =
    try Right(a)  // evaluate thunk
    catch { case e: Exception => Left(e) }
```

Chapter 4 of *Functional Programming in Scala* [4] describes other functions for type `Either`.

4.5 Standard Library

Both `Option` and `Either` appear in the standard library.

The standard library `Option` type is similar to the one developed here, but the library version is missing some of the extended functions described in Chapter 4 [4].

The standard library `Either` type is similar but more complicated, using projections on the left and right. It is also missing some of the extended functions from Chapter 4 [4].

Study the Scala API documentation [32] for more information on these data types.

4.6 Summary

The big idea in this chapter is to use ordinary values to represent exceptions and use higher-order functions for handling and propagating errors. As examples, we considered the algebraic data types `Option` and `Either` and functions such as `map`, `flatMap`, `filter`, and `orElse` to process their values.

We use this general technique of using values to represent effects in the subsequent studies in this course.

We introduced the idea of nonstrict functions. We examine the implications and use of these more in Chapter 5 [4,14].

4.7 Source Code for Chapter

The Scala source code files for the functions in this chapter (4) are as follows:

- `Option2.scala` for the Option algebraic data type from this chapter
- `Either2.scala` for the Either algebraic data type from this chapter
- `WrapException.scala` for the exc
- `List2.scala` for the List algebraic data type from Chapter 3
- `ClosureExample.scala`
- `ThunkExample.scala`

4.8 Exercises

TODO: Add

4.9 Acknowledgements

In Spring 2016, I wrote this set of notes to accompany my lectures on Chapter 4 of the book *Functional Programming in Scala* [4] (i.e., the Red Book). I constructed the notes around the ideas, general structure, and Scala examples from Chapter 4 and its chapter notes [6], and exercises [5].

I also patterned some of the discussion of for-comprehensions on Chapter 10 of the document *Scala by Example* by Martin Odersky [24], on Chapter 23 of 2nd Edition of the book *Programming in Scala* [25], on the relevant parts of the Scala language specification [32], and on the Scala FAQ [32].

In 2018 and 2019, I updated the format of the domdmdcument to be more compatible with my evolving document structures and corrected a few errors. In 2019, I also added the sections on null references and method chaining.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on the ELIFP textbook and other instructional materials. In January 2022, I began refining the ELIFP content and related

documents such as this one. I am integrating separately developed materials better, reformatting the documents (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

4.10 Terms and Concepts

Big idea: Using ordinary data values to represent failures and exceptions in programs. This preserves referential transparency and type safety while also preserving the benefit of exception-handling mechanisms, that is, the consolidation of error-handling logic.

Concepts: Error handling, exceptions referential transparency, type safety, null reference, Null Object design pattern, option types, nullabile types, `Option` and `Either` algebraic data types in Scala, method chaining, type variance, covariant and contravariant, parameter passing (by-value, by-name), thunk, free variables, closure, strict and nonstrict parameters/functions, eager and lazy evaluation, follow the types to implementation (type-driven development), lifting, for-comprehensions, syntactic sugar, (generators, definitions, guards), desugaring.

NOT FINISHED e.g., URL for citation [15] on abstract data types

5 Strictness and Laziness

5.1 Introduction

The big idea we discuss in this chapter is how we can exploit nonstrict functions to increase efficiency, increase code reuse, and improve modularity in functional programs.

5.1.1 Motivation

In our discussion [12] of Chapter 3 of *Functional Programming in Scala* [4], we examined purely functional data structures—in particular, the immutable, singly linked list.

We also examined the design and use of several bulk operations—such as `map`, `filter`, `foldLeft`, and `foldRight`. Each of these operations makes a pass over the input list and often constructs a new list for its output.

Consider the Scala expression

```
List(10,20,30,40,50).map(_/10).filter(_%2 == 1).map(_*100)
```

that generates the result:

```
List(100, 300, 500)
```

The evaluation of the expression requires three passes through the list. However, we could code a specialized function that does the same work in one pass.

```
def mfm(xs: List[Int]): List[Int] = xs match {
  case Nil     => Nil
  case (y::ys) =>
    val z = y / 10
    if (z % 2 == 1) (z*100) :: mfm(ys) else mfm(ys)
}
```

Note: In this chapter, we use the method-chaining formulation of `List` from the standard library, not the one we developed in the "Functional Data Structures" chapter. `::` constructs a list with its left operand as the head value and its right operand as the tail list.

It would be convenient if we could instead get a result similar to `mfm` by composing simpler functions like `map` and `filter`.

Can we do this?

We can by taking advantage of *nonstrict* functions to build a *lazy* list structure. We introduced the concepts of strict and nonstrict functions in Chapter 4 [13]; we elaborate on them in this chapter.

5.1.2 What are strictness and nonstrictness?

If the evaluation of an expression runs forever or throws an exception instead of returning an explicit value, we say the expression does not *terminate*—or that it evaluates to *bottom* (written symbolically as ⊥).

A function `f` is *strict* if `f(x)` evaluates to bottom for all `x` that themselves evaluate to bottom. That is, `f(⊥) == ⊥`. A strict function's argument must always have a value for the function to have a value.

A function is *nonstrict* (sometimes called *lenient*) if it is not strict. That is, `f(⊥) != ⊥`. The function can sometimes have value even if its argument does not have a value.

For multiparameter functions, we sometimes apply these terms to individual parameters. A *strict* parameter of a function must always be evaluated by the function. A *nonstrict* parameter of a function may sometimes be evaluated by the function and sometimes not.

5.1.3 Exploring nonstrictness

By default, Scala functions are strict.

However, some operations are nonstrict. For example, the "short-circuited" `&&` operation is nonstrict; it does not evaluate its second operand when the first operation is `false`. Similarly, `||` does not evaluate its second operand when its first operand is `true`.

Consider the `if` expression as a ternary operator. When the condition operand evaluates to `true`, the operator evaluates the second (i.e., then) operand but not the third (i.e., else) operand. Similarly, when the condition is `false`, the operator evaluates the third operand but not the second.

We could implement `if` as a function as follows:

```
def if2[A](cond: Boolean, onTrue: () => A,
           onFalse: () => A): A =
    if (cond) onTrue() else onFalse()
```

Then we can call `if2` as in the code fragment

```
val age = 21
if2(age >= 18, () => println("Can vote"),
               () => println("Cannot vote"))
```

and get the output:

```
Can vote
```

The parameter type `() => A` means that the corresponding argument is passed as a parameterless function that returns a value of type `A`. This function wraps the

expression, which is not evaluated before the call. This function is an explicitly specified *thunk*.

When the value is needed, then the called function must *force* the evaluation of the thunk by calling it explicitly, for example by using `onTrue()`.

To use the approach above, the caller must explicitly create the thunk. However, as we saw in the previous chapter, Scala provides *call-by-name* parameter passing that relieves the caller of this requirement in most circumstances. We can thus rewrite `if2` as follows:

```
def if2[A](cond: Boolean, onTrue: => A, onFalse: => A): A =
    if (cond) onTrue else onFalse
```

The `onTrue: => A` notation makes the argument expression a by-name parameter. Scala automatically creates the thunk for parameter `onTrue` and enables it to be referenced within the function without explicitly forcing its evaluation, for example by using `onTrue`.

An advantage of call-by-name parameter passing is that the evaluation of an expression can be delayed until its value is referenced, which may be never. A disadvantage is that the expression will be evaluated every time it is referenced.

To determine how to address this disadvantage, consider function

```
def maybeTwice(b: Boolean, i: => Int) = if (b) i + i else 0
```

which can be called as

```
println(maybeTwice(true, {println("hi"); 1 + 41}))
```

to generate output:

```
hi
hi
84
```

Note that the argument expression `i` is evaluated twice.

We can address this issue by defining a new variable and initializing it *lazily* to have the same value as the by-name parameter. We do this by declaring the temporary variable as a `lazy val`. The temporary variable will not be initialized until it is referenced, but it *caches* the calculated value so that it can be used without reevaluation on subsequent references.

We can rewrite `maybeTwice` as follows:

```
def maybeTwice2(b: Boolean, i: => Int) = {
    lazy val j = i
    if (b) j+j else 0
}
```

Now calling it as

```
println(maybeTwice2(true, {println("hi"); 1 + 41}))
```

generates output:

```
hi
84
```

This technique of caching the result of the evaluation gives us *call-by-need* parameter passing as it is called in Haskell and other lazily evaluated languages.

5.2 Lazy Lists

Now let's return to the problem discussed in the Motivation subsection. How can we use laziness to improve efficiency and modularity of our programs?

In this section, we answer this question by developing *lazy lists* or *streams*. These allow us to carry out multiple operations on a list without always making multiple passes over the elements.

Consider a simple "stream" algebraic data type `StreamC`. A nonempty stream consists of a head and a tail, both of which must be nonstrict.

Note: The *Functional Programming in Scala* book uses algebraic data type `Stream`, which differs from the implementation of the similar `Stream` type in Scala's standard library. To avoid conflicts with the standard library type, these notes use `StreamC`.

For technical reasons, Scala does not allow by-name parameters in the constructors for case classes. Thus these components must be explicitly defined thunks whose evaluations are explicitly forced when their values are required.

```
import StreamC._

sealed trait StreamC[+A]
case object Empty extends StreamC[Nothing]
case class Cons[+A](h: () => A, t: () => StreamC[A])
    extends StreamC[A]

object StreamC {
    def cons[A](hd: => A, tl: => StreamC[A]): StreamC[A] = {
        lazy val head = hd  // cache values once computed
        lazy val tail = tl
        Cons(() => head, () => tail)  // create thunks for Cons
    }
    def empty[A]: StreamC[A] = Empty
    def apply[A](as: A*): StreamC[A] =
        if (as.isEmpty)
            empty
        else
```

```
            cons(as.head, apply(as.tail: _*))
    }
```

5.2.1 Smart constructors and memoized streams

In the `StreamC` data type, we define two *smart constructors* to create new streams. By convention, these are functions defined in the companion object with the same names as the corresponding type constructors except they begin with a lowercase letter. They construct a data type object, ensuring that the needed integrity invariants are established. In the `StreamC` type, these take care of the routine work of creating the thunks, caching the values, and enabling transparent use of the parameters.

Smart constructor function `cons` takes the head and tail of the new `StreamC` as by-name parameters, equates these to lazy variables to cache their values, and then creates a `Cons` cell. The `h` and `t` fields of the `Cons` are explicitly defined thunks wrapping the head and the tail of the stream, respectively.

The evaluation of the thunk `h` of a `Cons` cell returns the value of the lazy variable `head` in the cell's closure. If this is the first access to `head`, then the access causes the corresponding by-name argument `hd` to be evaluated and cached in `head`. Subsequent evaluations of `h` get the cached value.

The evaluation of the thunk `t` of a `Cons` cell causes similar effects on the lazy variable `tail` and the corresponding by-name argument `tl`. However, the value of this argument is itself a `StreamC`, which may include lazily evaluated fields.

Smart constructor function `empty` just creates an `Empty StreamC`.

We define both smart constructors to have return type `StreamC[A]`. In addition to establishing the needed invariants, the use of the smart constructors helps Scala's type inference mechanism infer the `StreamC` type (which is what we usually want) instead of the subtypes associated with the case class/object constructors (which is what often will be inferred in Scala's object-oriented type system).

Convenience function `apply` takes a sequence of zero or more arguments and creates the corresponding `StreamC`.

If a function examines or traverses a `StreamC`, it must explicitly force evaluation of the thunks. In general, we should encapsulate such accesses within functions defined as a part of the `StreamC` implementation. (That is, we should practice *information hiding*, hide this design detail as a *secret* of the `StreamC` implementation as we discuss in the notes on abstract data types [15].)

An example of this is function `headOption` that optionally extracts the head of the stream.

```
    def headOption: Option[A] = this match {
        case Empty      => None
```

```
        case Cons(h,t) => Some(h()) // force thunk
    }
```

It explicitly forces evaluation of the thunk and thus enables code that called it to work with the values.

This technique for caching the value of the by-name argument is an example of memoizing the function. In general, *memoization* is an implementation technique in which a function stores the return value computed for certain arguments. Instead of recomputing the value on a subsequent call, the function just returns the cached value. This technique uses memory space to (potentially) save computation time later.

5.2.2 Helper functions

Now let's define a few functions that help us manipulate streams. We implement these as methods on the `StreamC` trait.

First, let's define a function `toList` that takes a `StreamC` (as its implicit argument) and constructs the corresponding Scala `List`. A standard backward recursive method can be defined as follows:

```
    def toListRecursive: List[A] = this match {
        case Cons(h,t) => h() :: t().toListRecursive // force thunks
        case _         => List()
    }
```

Of course, this method may suffer from stack overflow for long streams. We can remedy this by using a tail recursive auxiliary function that uses an accumulator to build up the list in reverse order and then reverses the constructed list.

```
    def toList: List[A] = {
        @annotation.tailrec
        def go(s: StreamC[A], acc: List[A]): List[A] = s match {
            case Cons(h,t) => go(t(), h() :: acc) // force thunks
            case _         => acc
        }
        go(this, List()).reverse
    }
```

To avoid the `reverse`, we could instead build up the list in a mutable `ListBuffer` using a loop and then, when finished, convert the buffer to an immutable `List`. We preserve the *purity* of the `toList` function by encapsulating use of the mutable buffer inside the function.

```
    def toListFast: List[A] = {
        val buf = new collection.mutable.ListBuffer[A]
        @annotation.tailrec
        def go(s: StreamC[A]): List[A] = s match {
            case Cons(h,t) =>
```

```
            buf += h() // force head thunk, add to end of buffer
            go(t())    // force tail thunk, process recursively
        case _ => buf.toList  // convert buffer to immutable list
    }
    go(this)
}
```

Next, let's define function `take` to return the first `n` elements from a `StreamC` and function `drop` to skip the first `n` elements.

We can define method `take` using a standard backward recursive form that matches on the structure of the implicit argument. However, we must be careful not to evaluate either the head or the tail thunks unnecessarily (e.g., by treating the `n == 1` and `n == 0` cases specially).

```
def take(n: Int): StreamC[A] = this match {
    case Cons(h, t) if n > 1  => cons(h(), t().take(n - 1))
    case Cons(h, _) if n == 1 => cons(h(), empty)
    case _ => empty  // stream empty or n < 1
}
```

Function `take` does its work *incrementally*. The recursive leg of the definition (i.e., the first leg) returns a `Cons` cell with the recursive call to `take` embedded in the lazily evaluated tail field. It will only be evaluated if its value is required.

We can define method `drop` to recursively calling `drop` on the forced tail. This yields the following tail recursive function.

```
@annotation.tailrec
final def drop(n: Int): StreamC[A] = this match {
    case Cons(_, t) if n > 0 => t().drop(n - 1)
    case _ => this
}
```

Unlike `take`, `drop` is not incremental. The recursive call is not lazily evaluated.

Finally, let's also define method `takeWhile` to return all starting elements of the `StreamC` that satisfy the given predicate.

```
def takeWhile(p: A => Boolean): StreamC[A] = this match {
    case Cons(h,t) if p(h()) => cons(h(), t() takeWhile p)
    case _ => empty
}
```

In the first case, we apply method `takeWhile` as an infix operator.

5.3 Separating Program Description from Evaluation

One of the fundamental design concepts in software engineering and programming is *separation of concerns*. A concern is some set of information that affects the design and implementation of a software system [53]. We identify the key

concerns in a software design and try to keep them separate and independent from each other. The goal is to implement the parts independently and then combine the parts to form a complete solution.

We apply separation of concerns in modular programming and abstract data types as *information hiding* [15,26,48]. We hide the *secrets* of how a module is implemented (e.g., what algorithms and data structures are used, what specific operating system or hardware devices are used, etc.) from the external users of the module or data type. We encapsulate the secrets behind an *abstract interface* [3,15].

We also apply separation of concerns in software architecture for computing applications. For example, we try to keep an application's *business logic* (i.e., specific knowledge about the application area) separate from its user interface such as described by the *Model-View-Controller* (MVC) architectural design pattern [49] commonly used in Web applications.

In functional programming, we also apply separation of concerns by seeking to *keep the description of computations separate from their evaluation* (execution). Examples include:

- first-class functions that express computations in their bodies but which must be supplied arguments before they execute
- use of `Option` or `Either` to express that an error has occurred but deferring the handling of the error to other parts of the program
- use of `StreamC` operators to assemble a computation that generates a sequence without actually running the computation until later when its result in needed

5.3.1 Laziness promotes reuse

In general, lazy evaluation enables us to separate the description of an expression from the evaluation of the expression. It enables us to to describe a "larger" expression than we need and then to only evaluate the portion that we actually need. This offers us the potential for greater *code reuse*.

Note: For a classic discussion of how higher-order and first-class functions and lazy evaluation promote software modularity and reuse, see the John Hughes paper "Why Functional Programming Matters" [21].

Consider a method `exists` on `StreamC` that checks whether an element matching a Boolean function `p` occurs in the stream. We can define this using an explicit tail recursion as follows:

```
def exists(p: A => Boolean): Boolean = this match {
    case Cons(h,t) => p(h()) || t().exists(p)
    case _         => false
}
```

Given that `||` is nonstrict in its second argument, this function terminates and returns `true` as soon as it finds the first element that makes `p` true. Because the stream holds the tail in a `lazy val`, it is only evaluated when needed. So `exists` does not evaluate the stream past the first occurrence.

As with the `List` data type in Chapter 3, we can define a more general method `foldRight` on `StreamC` to represent the pattern of computation exhibited by `exists`.

```
def foldRight[B](z: => B)(f: (A, => B) => B): B = this match {
    case Cons(h,t) => f(h(), t().foldRight(z)(f))
    case _         => z
}
```

The notation `=> B` in the second parameter of combining function `f` takes its second argument by-name and, hence, may not evaluate it in all circumstances. If `f` does not evaluate its second argument, then the recursion terminates. Thus the overall `foldRight` computation can terminate before it completes the complete traversal through the stream.

We can now redefine `exists` to use the more general function as follows:

```
def exists2(p: A => Boolean): Boolean =
    foldRight(false)((a, b) => p(a) || b)
```

Here parameter `b` represents the unevaluated recursive step that folds the tail of the stream. If `p(a)` returns `true`, then `b` is not evaluated and the computation terminates early.

Caveat: The second version of `exists` illustrates how we can use a general function to represent a variety of more specific computations. But, for a large stream in which all elements evaluate to `false`, this version is not stack safe.

Because the `foldRight` method on `StreamC` can terminate its traversal early, we can use it to implement `exists` efficiently. Unfortunately, we cannot implement the `List` version of `exists` efficiently in terms of the `List` version of `foldRight`. We must implement a specialized recursive version of `exists` to get early termination.

Laziness thus enhances our ability to reuse code.

5.3.2 Incremental computations

Now, let's flesh out the `StreamC` trait and implement the basic `map`, `filter`, `append`, and `flatMap` methods using the general function `foldRight`, as follows:

```
def map[B](f: A => B): StreamC[B] =
    foldRight(empty[B])((h,t) => cons(f(h), t))

def filter(p: A => Boolean): StreamC[A] =
    foldRight(empty[A])((h,t) => if (p(h)) cons(h, t) else t)
```

```
def append[B >: A](s: => StreamC[B]): StreamC[B] =
    foldRight(s)((h,t) => cons(h,t))

def flatMap[B](f: A => StreamC[B]): StreamC[B] =
    foldRight(empty[B])((h,t) => f(h) append t)
```

These implementations are *incremental*. They do not fully generate all their answers. No computation takes place until some other computation examines the elements of the output `StreamC` and then only enough elements are generated to give the requested result.

Because of their incremental nature, we can call these functions one after another without fully generating the intermediate results.

We can now address the problem raised in the Introduction section of these notes. There we asked the question of how can we compute the result of the expression

```
List(10,20,30,40,50).map(_/10).filter(_%2 == 1).map(_*100)
```

without producing two unneeded intermediate lists.

The `StreamC` expression

```
StreamC(10,20,30,40,50).map(_/10).filter(_%2 == 1).
    map(_*100).toList
```

generates the result:

```
List(100, 300, 500)
```

which is the same as the `List` expression. The expression looks the same except that we create a `StreamC` initially instead of a `List` and we call `toList` to force evaluation of stream at the end.

When executed, the lazy evaluation interleaves two `map`, the `filter`, and the `toList` transformations. The computation does not fully instantiate any intermediate streams. It is a similar interleaving to what we did in the special purpose function `mfm` in the introduction.

(For a more detailed discussion of this interleaving, see Listing 5.3 in the first edition of thr *Functional Programming in Scala* book [4].)

Because stream computations do not generate intermediate streams in full, we are free to use stream operations in ways that might seem counterintuitive at first. For example, we can use `filter` (which seems to process the entire stream) to implement `find`, a function to return the first occurrence of an element in a stream that satisfies a given predicate, as follows:

```
def find(p: A => Boolean): Option[A] = filter(p).headOption
```

The incremental nature of these computations can sometimes save memory. The computation may only need a small amount of working memory; the garbage

collector can quickly recover working memory that the current step does not need.

Of course, some computations may require more intermediate elements and each element may itself require a large amount of memory, so not all computations are as well-behaved as the examples in this section.

5.3.3 For comprehensions on streams

Given that we have defined `map`, `filter`, and `flatMap`, we can now use sequence comprehensions on our `StreamC` data. For example, the code fragment

```
val seq = for (x <- StreamC(1,2,3,4) if x > 2;
               y <- StreamC(1,2)) yield x
println(seq.toList)
```

causes the following to print on the console:

```
List(3, 3, 4, 4)
```

Note: During compilation, some versions of the Scala compiler may issue a deprecation warning that `filter` is used instead of `withFilter`. In a future release of Scala, this substitution may no longer work. Because `filter` is lazy for streams, we could define `f<ilter` as an alias for `withFilter` with the following:

```
def withFilter = filter _
```

However, `filter` does generate a new `StreamC` where `withFilter` normally does not generate a new collection. Although this gets rid of the warning, it would be better to implement a proper `withFilter` function.

5.4 Infinite Streams snd Corecursion

Because the streams are incremental, the functions we have defined also work for *infinite streams*.

Consider the following definition for an infinite sequence of ones:

```
lazy val ones: StreamC[Int] = cons(1, ones)
```

Note: The book *Functional Programming in Scala* [4] does not add the `lazy` annotation, but that version gives a compilation error in some versions of Scala. Adding `lazy` seems to fix the problem, but this issue should be investigated further.

Although `ones` is infinite, the `StreamC` functions only reference the finite prefix of the stream needed to compute the needed result.

For example:

- `ones.take(5).toList` yields `List(1,1,1,1,1)`
- `ones.map(_+2).take(5).toList` yields `List(3,3,3,3,3)`

- What about `ones.map(_+2).toList`?

We can generalize `ones` to a `constant` function as follows:

```
def constant[A](a: A): StreamC[A] = {
    lazy val tail: StreamC[A] = Cons(() => a, () => tail)
    tail
}
```

An alternative would be just to make the body `cons(a, constant(a))`. But the above is more efficient because it is just one object referencing itself.

We can also define an increasing `StreamC` of all integers beginning with `n` as follows:

```
def from(n: Int): StreamC[Int] =
    cons(n, from(n+1))
```

The (second-order) Fibonacci sequence begins with the elements 0 and 1; each subsequent element is the sum of the two previous elements. We can define the Fibonacci sequence as a stream `fibs` with the following definition:

```
val fibs = {
    def go(f0: Int, f1: Int): StreamC[Int] =
        cons(f0, go(f1, f0+f1))
    go(0, 1)
}
```

5.4.1 Prime numbers: Sieve of Erastosthenes

A positive integer greater than 1 is *prime* if it is divisible only by itself and 1. The *Sieve of Eratosthenes* algorithm works by removing multiples of numbers once they are identified as prime.

- We begin the increasing stream of integers starting with 2, a prime number.
- The head is 2, so we remove all the multiples of 2 from the stream.
- The head of the tail is 3, so it is prime because it was not removed as a multiple of 2 and it is the smallest integer remaining.
- Continue the process recursively on the tail.

We can define this calculation with the following `StreamC` functions.

```
def sieve(ints: StreamC[Int]): StreamC[Int] =
    ints.headOption match {
        case None =>
            sys.error(
                "Should not occur: No head on infinite stream.")
        case Some(x) =>
            cons(x,sieve(ints drop 1 filter (_ % x > 0)))
```

```
    }
val primes: StreamC[Int] = sieve(from(2))
```

We can then use `primes` to define a function `isPrime` to test whether an integer is prime.

```
def isPrime(c: Int): Boolean =
    (primes filter (_ >= c) map (_ == c)).headOption getOrElse
        sys.error(
            "Should not occur: No head on infinite list.")
```

5.4.2 Function `unfold`

Now let's consider `unfold`, a more general stream-building function. Function `unfold` takes an initial state and a function that produces both the next state and the next value in the stream and builds the resulting stream. We can define it as follows:

```
def unfold[A, S](z: S)(f: S => Option[(A, S)]): StreamC[A] =
    f(z) match {
        case Some((h,s)) => cons(h, unfold(s)(f))
        case None        => empty
    }
```

This function applies `f` to the current state `z` to generate the next state `s` and the next element `h` of the stream. We use `Option` so `f` can signal when to terminate the `StreamC`.

Function `unfold` is an example of a corecursive function.

A *recursive* function consumes data. The input of each successive call is "smaller" than the previous one. Eventually the recursion terminates when input size reaches the minimum.

A *corecursive* function produces data. Corecursive functions need not terminate as long as they remain *productive*. By productive, we mean that the function can continue to evaluate more of the result in a finite amount of time.

Where we seek to argue that recursive functions terminate, we seek to argue that corecursive functions are productive.

The `unfold` function remains productive as long as its argument function `f` terminates. Function `f` must terminate for the `unfold` computation to reach its next state.

Some writers in the functional programming community use the term *guarded recursion* instead of corecursion and the term *cotermination* instead of productivity. See the Wikipedia articles on corecursion [40] and coinduction [41] for more information and links.

The function `unfold` is very general. For example, we can now define `ones`, `constant`, `from`, and `fibs` with `unfold`.

```
val onesViaUnfold = unfold(1)(_ => Some((1,1)))

def constantViaUnfold[A](a: A) =
    unfold(a)(_ => Some((a,a)))

def fromViaUnfold(n: Int) =
    unfold(n)(n => Some((n,n+1)))

val fibsViaUnfold =
    unfold((0,1)) { case (f0,f1) => Some((f0,(f1,f0+f1))) }
```

5.5 Summary

The big idea in this chapter is that we can exploit nonstrict functions to increase efficiency, increase code reuse, and improve the modularity in functional programs.

5.6 Source Code for Chapter

The Scala source code files for the functions in this chapter (5) are as follows:

- `StreamC.scala`

5.7 Exercises

TODO: Add

5.8 Acknowledgements

In Spring 2016, I wrote this set of notes to accompany my lectures on Chapter 5 of the first edition of the book *Functional Programming in Scala* [4] (i.e., the Red Book). I constructed the notes around the ideas, general structure, and Scala examples from that chapter and its associated materials [5,6].

In 2018 and 2019, I updated the format of the document to be more compatible with my evolving document structures. In 2019, I also renamed the `Stream` (used in the Red Book) to `StreamC` to better avoid conflicts with the standard library type `Stream`.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on the ELIFP textbook and other instructional materials. In January 2022, I began refining the ELIFP content and related documents such as this one. I am integrating separately developed materials better, reformatting the documents (e.g., using CSS), constructing a unified

bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

5.9 Terms and Concepts

Big idea: Exploiting nonstrict function to increase efficiency, increase code reuse, and improve modularity

Concepts: Strict and nonstrict (lenient) functions/parameters, termination, bottom, call-by-name, thunk, forcing, call-by-need, lazy evaluation, lazy lists or streams, `Stream` data type, smart constructors, memoization, `lazy` variables, purity of functions, separation of concerns, information hiding, design secret, abstract interface, business logic, Model-View-Controller (MVC) design pattern, keeping program description separate from evaluation, incremental computation, prime number, Sieve of Eratosthenes, recursive, corecursive (guarded recursion), productivity (cotermination).

References

[1] Edwin Brady. 2017. *Type-driven development with Idris*. Manning, Shelter Island, New York, USA.

[2] Edwin Brady. 2022. Idris: A language for type-driven development. Retrieved from https://www.idris-lang.org

[3] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. 1981. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the 5th international conference on software engineering*, IEEE, San Diego, California, USA, 195–204.

[4] Paul Chiusano and Runar Bjarnason. 2015. *Functional programming in Scala* (First ed.). Manning, Shelter Island, New York, USA.

[5] Paul Chiusano and Runar Bjarnason. 2022. FP in Scala exercises, hints, and answers. Retrieved from https://github.com/fpinscala/fpinscala

[6] Paul Chiusano and Runar Bjarnason. 2022. FP in Scala community guide and chapter notes. Retrieved from https://github.com/fpinscala/fpinscala/wiki

[7] James Coplien, Daniel Hoffman, and David Weiss. 1998. Commonality and variability in software engineering. *IEEE Software* 15, 6 (1998), 37–45.

[8] H. Conrad Cunningham. 2014. *Notes on functional programming with Haskell*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/Notes_FP_Haskell/Notes_on_Functional_Programming_with_Haskell.pdf

[9] H. Conrad Cunningham. 2019. *Recursion concepts and terminology: Scala version*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/RecursionStyles/Scala/RecursionStylesScala.html

[10] H. Conrad Cunningham. 2019. *Notes on Scala for Java programmers*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScaldFP/ScalaForJava/ScalaForJava.html

[11] H. Conrad Cunningham. 2019. *Type system concepts*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/TypeConcepts/TypeSystemConcepts.html

[12] H. Conrad Cunningham. 2019. *Functional data structures (Scala)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScalaFP/FPS03/FunctionalDS.html

[13] H. Conrad Cunningham. 2019. *Handling errors without exceptions (Scala)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScalaFP/FPS04/ErrorHandling.html

[14] H. Conrad Cunningham. 2019. *Strictness and laziness (Scala)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScalaFP/FPS05/Laziness.html

[15] H. Conrad Cunningham. 2019. *Abstract data types in Scala*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/Digraph/Scala/AbstractDataTypes.html

[16] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[17] Evan Czaplicki. 2022. Elm: A delightful language for reliable web applications. Retrieved from https://elm-lang.org

[18] Richard Feldman. 2020. *Elm in action*. Manning, Shelter Island, New York, USA.

[19] Phil Freeman. 2017. *Purescript by example: Functional programming for the web*. Leanpub, Victoria, British Columbia, Canada. Retrieved from https://book.purescript.org/

[20] Tony Hoare. 2009. Null references: The billion dollar mistake (presentation). Retrieved from https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[21] John Hughes. 1989. Why functional programming matters. *Computer Journal* 32, 2 (1989), 98–107.

[22] Steve Klabnik, Carol Nichols, and Conributers. 2019. *The Rust programming language* (Rust 2018th ed.). No Starch Press, San Francisco, California, USA. Retrieved from https://doc.rust-lang.org/book/

[23] Barbara Liskov. 1987. Keynote address—Data abstraction and hierarchy. In *Proceedings on object-oriented programming systems, languages, and applications (OOPSLA '87): addendum*, ACM, Orlando, Florida, USA, 17–34.

[24] Martin Odersky. 2014. *Scala by example*. École Polytechnique Fédérale Lausanne (EPFL), Programming Methods Laboratory, Lausanne, Switzerland.

[25] Martin Odersky, Lex Spoon, and Bill Venners. 2011. *Programming in Scala* (Second ed.). Artima, Inc., Walnut Creek, California, USA.

[26] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[27] Tomas Petricek. 2012. Why type-first development matters (blog post. Retrieved from http://tomasp.net/blog/type-first-development.aspx/

[28] purescript.org. 2022. PureScript: A strongly-typed functional programming language that compiles to javascript. Retrieved from https://www.purescript.org/

[29] Python Software Foundation. 2022. Python. Retrieved from https://www.python.org/

[30] Luciano Ramalho. 2013. *Fluent Python: Clear, concise, and effective programming.* O'Reilly Media, Sebastopol, California, USA.

[31] Rust Team. 2022. Rust: A language empowering everyone to build reliable and efficient software. Retrieved from https://www.rust-lang.org/

[32] Scala Language Organization. 2022. The Scala programming language. Retrieved from https://www.scala-lang.org/

[33] Scala Language Organization. 2022. Tour of Scala: Unified types. Retrieved from https://docs.scala-lang.org/tour/unified-types.html

[34] Source Making. 2022. Null object design pattern. Retrieved from https://sourcemaking.com/design_patterns/null_object

[35] Simon Thompson. 2011. *Haskell: The craft of programming* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA.

[36] Dean Wampler and Alex Payne. 2014. *Programming Scala: Scalability = functional programming + objects* (Second ed.). O'Reilly Media, Sebastopol, California, USA.

[37] Wikpedia: The Free Encyclopedia. 2022. Abstract data type. Retrieved from https://en.wikipedia.org/wiki/Abstract_data_type

[38] Wikpedia: The Free Encyclopedia. 2022. Algebraic data type. Retrieved from https://en.wikipedia.org/wiki/Algebraic_data_type

[39] Wikpedia: The Free Encyclopedia. 2022. Covariance and contravariance (computer science). Retrieved from https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

[40] Wikpedia: The Free Encyclopedia. 2022. Corecursion. Retrieved from https://en.wikipedia.org/wiki/Corecursion

[41] Wikpedia: The Free Encyclopedia. 2022. Coinduction. Retrieved from https://en.wikipedia.org/wiki/Coinduction

[42] Wikpedia: The Free Encyclopedia. 2022. Liskov substitution principle. Retrieved from https://en.wikipedia.org/wiki/Liskov_substitution_principle

[43] Wikpedia: The Free Encyclopedia. 2022. Polymorphism (computer science). Retrieved from https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

[44] Wikpedia: The Free Encyclopedia. 2022. Function overloading. Retrieved from https://en.wikipedia.org/wiki/Function_overloading

[45] Wikpedia: The Free Encyclopedia. 2022. Ad hoc polymorphism. Retrieved from https://en.wikipedia.org/wiki/Ad_hoc_polymorphism

[46] Wikpedia: The Free Encyclopedia. 2022. Parametric polymophism. Retrieved from https://en.wikipedia.org/wiki/Parametric_polymorphism

[47] Wikpedia: The Free Encyclopedia. 2022. Subtyping. Retrieved from https://en.wikipedia.org/wiki/Subtyping

[48] Wikpedia: The Free Encyclopedia. 2022. Information hiding. Retrieved from https://en.wikipedia.org/wiki/Information_hiding

[49] Wikpedia: The Free Encyclopedia. 2022. Model-view-controller. Retrieved from https://en.wikipedia.org/wiki/Model-view-controller

[50] Wikpedia: The Free Encyclopedia. 2022. Nullable type. Retrieved from https://en.wikipedia.org/wiki/Nullable_type

[51] Wikpedia: The Free Encyclopedia. 2022. Null object pattern. Retrieved from https://en.wikipedia.org/wiki/Null_object_pattern

[52] Wikpedia: The Free Encyclopedia. 2022. Option type. Retrieved from https://en.wikipedia.org/wiki/Option_type

[53] Wikpedia: The Free Encyclopedia. 2022. Separation of concerns. Retrieved from https://en.wikipedia.org/wiki/Separation_of_concerns

[54] Bobby Woolf. 1997. Null object. In *Pattern languages of program design 3*, Robert Martin, Dirk Riehle and Frank Buschmann (eds.). Addison-Wesley, Boston, Massachusetts, USA, 5–18.