CSci 555, Functional Programming, Spring 2016
Functional Programming in Scala
Functional Data Structures

H. Conrad Cunningham

19 April 2016 (minor edit 4 February 2018

Contents

Functional Data Structures **2**
- Introduction . . . 2
- A `List` algebraic data type . . . 2
  - Algebraic data types . . . 2
  - Using a Scala trait . . . 3
  - Polymorphism . . . 4
  - Variance . . . 5
  - Defining functions in the companion object . . . 6
  - Function to sum a list . . . 7
  - Function to multiply a list . . . 8
  - Function to remove adjacent duplicates . . . 9
  - Variadic function apply . . . 10
- Data sharing . . . 10
  - Function to take tail of list . . . 11
  - Function to drop from beginning of list . . . 12
  - Function to append lists . . . 13
- Other list functions . . . 14
  - Tail recursive function reverse . . . 14
  - Higher-order function dropWhile . . . 16
  - Curried function dropWhile . . . 16
- Generalizing to Higher Order Functions . . . 17
  - Fold Right . . . 17
  - Fold Left . . . 20
  - Map . . . 21
  - Filter . . . 23
  - Flat Map . . . 24
- Classic algorithms on lists . . . 25
  - Insertion sort and bounded generics . . . 25

Merge sort 27
Lists in the Scala standard library 29
Source Code for Chapter 31

Acknowledgements: This is a set of notes written to accompany my lectures on Chapter 3 of the book *Functional Programming in Scala* by Paul Chiusano and Runar Bjarnason (Manning, 2015). I constructed the notes around the ideas, general structure, and Scala examples from that chapter and its associated materials. I also adapted some text and examples from my *Notes on Functional Programming with Haskell*.

I expanded the discussion of algebraic data types, polymorphism, and variance. For this expansion, I examined other materials including the Wikipedia articles on Algebraic Data Type, Abstract Data Type, Polymorphism, Ad Hoc Polymorphism, Parametric Polymorphism, Subtyping, Function Overloading, and Covariance and Contravariance. I also examined the discussion of variance in the textbook *Programming Scala*, Second Edition, by Dean Wampler and Alex Payne (O'Reilly, 2014). I adapted the sorting algorithms from Martin Odersky's *Scala by Example*.

Prerequisite: This discussion assumes the reader is familiar with the programming concepts and Scala features covered in my *Notes on Scala for Java Programmers* (adapted from a tutorial on the Scala website) and *Recursion Concepts and Terminology*.

Advisory: The HTML version of this document uses MathML in a few locations. For best results, use a browser that supports the display of MathML. A good choice as of April 2016 seems to be a recent version of Firefox from Mozilla.

Functional Data Structures

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.

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?

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.

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.

Perhaps to add to the confusion, in functional programming we sometimes use an algebraic data type to help define an abstract data type. (See the "functional module style" implementation of the Natural number example, for instance.)

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?

Polymorphism

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

In general, two primary kinds of polymorphism exist in programming languages:

1. *Ad hoc polymorphism*, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.

   There are two subkinds of ad hoc polymorphism.

   a. *Overloading* refers to ad hoc polymorphism in which the language's compiler or interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at compile time based on the declared types of the entities. They exhibit *early binding*.

      Java overloads a few operator symbols, such as using the `+` symbol for both addition of numbers and concatenation of strings. Java also overloads calls of functions defined with the same name but different signatures (patterns of parameter types and return value). Java does not support user-defined operator overloading; C++ does.

   b. *Subtyping* (also known as *subtype polymorphism* or *inclusion polymorphism*) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate implementation to invoke. This usually occurs at runtime, so this exhibits *late binding*.

      The object-oriented programming community often refers to inheritance-based subtype polymorphism as simply *polymorphism*.

2. *Parametric polymorphism*, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e., exhibiting early binding).

   The object oriented programming community often calls this type of polymorphism *generics* or *generic programming*. The functional programming community often calls this simply *polymorphism*.

Scala is a hybrid, object-functional language. Its type system supports all three types of polymorphism: subtyping by extending classes and traits, parametric polymorphism by using generic type parameters, and overloading through both the Java-like mechanisms described above 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]`?

Variance

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

Suppose we have a supertype `Fish` with a subtype `Bass`. 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]`, the 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 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 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.

Defining functions in the 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*.

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.

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.

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.

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.

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.

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 the figure below.

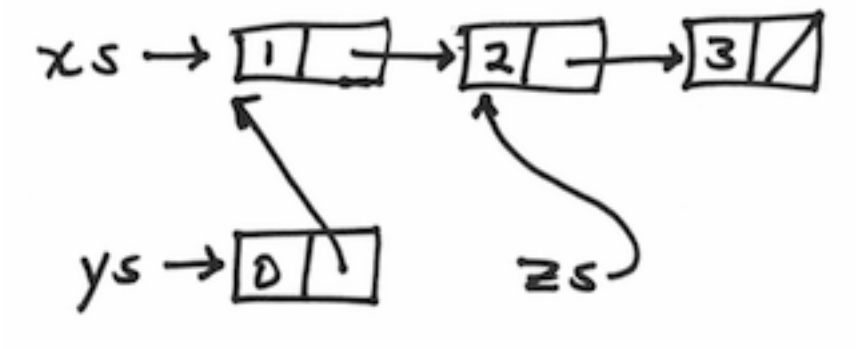

Figure: Data sharing in lists

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.

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 `tail` so that it explicitly throws an exception. It can be defined in the companion object for `List` as follows:

```
def tail[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 `tail` 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.

Function to drop from beginning of list

We can generalize `tail` 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 `tail` does. Although it seems illogical to take the `tail` 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.

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.

Other list functions

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 the following:

> 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.

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.

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.

Generalizing to Higher Order Functions

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?

Both functions exhibit the same *pattern of computation*. They both take a list of elements and insert a binary operator between all the consecutive elements of the list in order to reduce the list to a single value. The operations are grouped from the right to the left. Function `sum` takes a list of integers and applies addition; `product` takes a list of double-precision floating point numbers and applies multiplication.

In addition, `sum` returns integer 0 when its argument is nil; if this is a recursive call, the return value is added to the right of the previous results. Similarly, `product` returns 1.0 when its argument is nil. The values 0 and 1.0 are the identity elements for addition and multiplication, respectively. Function `sum` processes a list of integers and returns an integer; `product` processes a list of double-precision floating point numbers and returns a double-precision floating point number.

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

- Pull the parts that differ into the generalized function's parameter list.
- Leave the parts that are the same in the generalized function's body.
- If a part moved to the generalized function's parameter list accesses local variables, then make that part a function with a parameter for each local variable accessed.

- If data types differ at some points, then add type parameters to the generalized function.
- If the same data type appears in multiple roles, then consider adding a distinct type parameter for each.

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 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.

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(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.

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.

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.

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 a previous section.

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.

Classic algorithms on lists

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
  define 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.

Merge sort

The insertion sort given in the previous section 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 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 $O(n\ \log(n))$, where n is the length of the input list.

- 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 $O(\log(n))$ recursive calls.

So the total cost is $O(n\ \log(n))$. 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.

Lists in the Scala standard library

In this discussion (and in Chapter 3 of *Functional Programming in Scala*), 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 2.11.7 `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 java.io.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`, `tail`, `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.

Source Code for Chapter

- List2.scala