Subtyping and Generics

Principles of Functional Programming

Polymorphism

Two principal forms of polymorphism:

- subtyping
- generics

In this session we will look at their interactions.

Two main areas:

- bounds
- variance

Type Bounds

Consider the method `assertAllPos` which

- takes an `IntSet`
- returns the `IntSet` itself if all its elements are positive
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`? Maybe:

Type Bounds

Consider the method `assertAllPos` which

- takes an `IntSet`
- returns the `IntSet` itself if all its elements are positive
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`? Maybe:

```
def assertAllPos(s: IntSet): IntSet
```

In most situations this is fine, but can one be more precise?

Type Bounds

One might want to express that `assertAllPos` takes `Empty` sets to `Empty` sets and `NonEmpty` sets to `NonEmpty` sets.

A way to express this is:

```
def assertAllPos[S <: IntSet](r: S): S = ...
```

Here, "`<: IntSet`" is an *upper bound* of the type parameter `S`:

It means that `S` can be instantiated only to types that conform to `IntSet`.

Generally, the notation

- `S <: T` means: `S` *is a subtype of* `T`, and
- `S >: T` means: `S` *is a supertype of* `T`, or `T` *is a subtype of* `S`.

Lower Bounds

You can also use a lower bound for a type variable.

Example

```
[S >: NonEmpty]
```

introduces a type parameter `S` that can range only over *supertypes* of `NonEmpty`.

So `S` could be one of `NonEmpty`, `IntSet`, `AnyRef`, or `Any`.

We will see later on in this session where lower bounds are useful.

Mixed Bounds

Finally, it is also possible to mix a lower bound with an upper bound.

For instance,

```
[S >: NonEmpty <: IntSet]
```

would restrict `S` any type on the interval between `NonEmpty` and `IntSet`.

Covariance

There's another interaction between subtyping and type parameters we need to consider. Given:

```
NonEmpty <: IntSet
```

is

```
List[NonEmpty] <: List[IntSet]    ?
```

Covariance

There's another interaction between subtyping and type parameters we need to consider. Given:

```
NonEmpty <: IntSet
```

is

```
List[NonEmpty] <: List[IntSet]    ?
```

Intuitively, this makes sense: A list of non-empty sets is a special case of a list of arbitrary sets.

We call types for which this relationship holds *covariant* because their subtyping relationship varies with the type parameter.

Does covariance make sense for all types, not just for `List`?

Arrays

For perspective, let's look at arrays in Java (and C#).

Reminder:

- An array of `T` elements is written `T[]` in Java.
- In Scala we use parameterized type syntax `Array[T]` to refer to the same type.

Arrays in Java are covariant, so one would have:

```
NonEmpty[] <: IntSet[]
```

Array Typing Problem

But covariant array typing causes problems.

To see why, consider the Java code below.

```
NonEmpty[] a = new NonEmpty[]{
  new NonEmpty(1, new Empty(), new Empty())};
IntSet[] b = a;
b[0] = new Empty();
NonEmpty s = a[0];
```

It looks like we assigned in the last line an `Empty` set to a variable of type `NonEmpty`!

What went wrong?

The Liskov Substitution Principle

The following principle, stated by Barbara Liskov, tells us when a type can be a subtype of another.

> *If* `A <: B`, *then everything one can to do with a value of type* `B` *one should also be able to do with a value of type* `A`.

[The actual definition Liskov used is a bit more formal. It says:

> *Let* `q(x)` *be a property provable about objects* `x` *of type* `B`. *Then* `q(y)` *should be provable for objects* `y` *of type* `A` *where* `A <: B`.

]

Exercise

The problematic array example would be written as follows in Scala:

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a
b(0) = Empty()
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

- 0 A type error in line 1
- 0 A type error in line 2
- 0 A type error in line 3
- 0 A type error in line 4
- 0 A program that compiles and throws an exception at run-time
- 0 A program that compiles and runs without exception

Exercise

The problematic array example would be written as follows in Scala:

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a
b(0) = Empty()
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

- 0 A type error in line 1
- 0 A type error in line 2
- 0 A type error in line 3
- 0 A type error in line 4
- 0 A program that compiles and throws an exception at run-time
- 0 A program that compiles and runs without exception

Tuples and Generic Methods

Principles of Functional Programming

Sorting Lists Faster

As a non-trivial example, let's design a function to sort lists that is more efficient than insertion sort.

A good algorithm for this is *merge sort*. The idea is as follows:

If the list consists of zero or one elements, it is already sorted.

Otherwise,

- Separate the list into two sub-lists, each containing around half of the elements of the original list.
- Sort the two sub-lists.
- Merge the two sorted sub-lists into a single sorted list.

First MergeSort Implementation

Here is the implementation of that algorithm in Scala:

```
def msort(xs: List[Int]): List[Int] =
  val n = xs.length / 2
  if n == 0 then xs
  else
    def merge(xs: List[Int], ys: List[Int]) = ???
    val (fst, snd) = xs.splitAt(n)
    merge(msort(fst), msort(snd))
```

The SplitAt Function

The `splitAt` function on lists returns two sublists

- the elements up the the given index
- the elements from that index

The lists are returned in a *pair*.

Detour: Pair and Tuples

The pair consisting of `x` and `y` is written `(x, y)` in Scala.

Example

```
val pair = ("answer", 42)   > pair : (String, Int) = (answer,42)
```

The type of `pair` above is `(String, Int)`.

Pairs can also be used as patterns:

```
val (label, value) = pair   > label : String = answer
                            | value : Int = 42
```

This works analogously for tuples with more than two elements.

Translation of Tuples

For small (*) n, the tuple type $(T_1, ..., T_n)$ is an abbreviation of the parameterized type

$$\texttt{scala.Tuple}n[T_1, ..., T_n]$$

A tuple expression $(e_1, ..., e_n)$ is equivalent to the function application

$$\texttt{scala.Tuple}n(e_1, ..., e_n)$$

A tuple pattern $(p_1, ..., p_n)$ is equivalent to the constructor pattern

$$\texttt{scala.Tuple}n(p_1, ..., p_n)$$

(*) Currently, "small" = up to 22. There's also a TupleXXL class that handles Tuples larger than that limit.

The Tuple class

Here, all `Tuple`*n* classes are modeled after the following pattern:

```
case class Tuple2[T1, T2](_1: +T1, _2: +T2) {
  override def toString = "(" + _1 + "," + _2 +")"
}
```

The fields of a tuple can be accessed with names `_1`, `_2`, ...

So instead of the pattern binding

```
val (label, value) = pair
```

one could also have written:

```
val label = pair._1
val value = pair._2
```

But the pattern matching form is generally preferred.

Definition of Merge

Here is a definition of the `merge` function:

```
def merge(xs: List[Int], ys: List[Int]) = (xs, ys) match
  case (Nil, ys) => ys
  case (xs, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
```

Making Sort more General

Problem: How to parameterize `msort` so that it can also be used for lists with elements other than `Int`?

```
def msort[T](xs: List[T]): List[T] = ???
```

does not work, because the comparison < in `merge` is not defined for arbitrary types `T`.

Idea: Parameterize `merge` with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) =
  ...
    merge(msort(fst)(lt), msort(snd)(lt))
```

Merge then needs to be adapted as follows:

```
def merge(xs: List[T], ys: List[T]) = (xs, ys) match
  ...
  case (x :: xs1, y :: ys1) =>
    if lt(x, y) then ...
    else ...
```

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val fruits = List("apple", "pear", "orange", "pineapple")

msort(xs)((x: Int, y: Int) => x < y)
msort(fruits)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `msort(xs)`:

```
msort(xs)((x, y) => x < y)
```

Variance

Principles of Functional Programming

Variance

You have seen the the previous session that some types should be covariant whereas others should not.

Roughly speaking, a type that accepts mutations of its elements should not be covariant.

But immutable types can be covariant, if some conditions on methods are met.

Definition of Variance

Say `C[T]` is a parameterized type and `A`, `B` are types such that `A <: B`.

In general, there are *three* possible relationships between `C[A]` and `C[B]`:

`C[A] <: C[B]`	`C` is *covariant*
`C[A] >: C[B]`	`C` is *contravariant*
neither `C[A]` nor `C[B]` is a subtype of the other	`C` is *nonvariant*

Definition of Variance

Say `C[T]` is a parameterized type and `A`, `B` are types such that `A <: B`.

In general, there are *three* possible relationships between `C[A]` and `C[B]`:

`C[A] <: C[B]`	`C` is *covariant*
`C[A] >: C[B]`	`C` is *contravariant*
neither `C[A]` nor `C[B]` is a subtype of the other	`C` is *nonvariant*

Scala lets you declare the variance of a type by annotating the type parameter:

`class C[+A] { ... }`	`C` is *covariant*
`class C[-A] { ... }`	`C` is *contravariant*
`class C[A] { ... }`	`C` is *nonvariant*

Exercise

Assume the following type hierarchy and two function types:

```
trait Fruit
class Apple extends Fruit
class Orange extends Fruit

type A = Fruit => Orange
type B = Apple => Fruit
```

According to the Liskov Substitution Principle, which of the following should be true?

- ○ `A <: B`
- ○ `B <: A`
- ○ `A and B are unrelated.`

Exercise

Assume the following type hierarchy and two function types:

```
trait Fruit
class Apple extends Fruit
class Orange extends Fruit

type A = Fruit => Orange
type B = Apple => Fruit
```

According to the Liskov Substitution Principle, which of the following should be true?

- ○ `A <: B`
- ○ `B <: A`
- ○ `A and B are unrelated.`

Exercise

Assume the following type hierarchy and two function types:

```
trait Fruit
class Apple extends Fruit
class Orange extends Fruit

type A = Fruit => Orange
type B = Apple => Fruit
```

According to the Liskov Substitution Principle, which of the following should be true?

X	`A <: B`
O	`B <: A`
O	`A and B are unrelated.`

Typing Rules for Functions

Generally, we have the following rule for subtyping between function types:

If `A2 <: A1` and `B1 <: B2`, then

```
A1 => B1  <:  A2 => B2
```

Function Trait Declaration

So functions are *contravariant* in their argument type(s) and *covariant* in their result type.

This leads to the following revised definition of the `Function1` trait:

```
package scala
trait Function1[-T, +U] {
  def apply(x: T): U
}
```

Variance Checks

We have seen in the array example that the combination of covariance with certain operations is unsound.

In this case the problematic operation was the update operation on an array.

If we turn `Array` into a class, and `update` into a method, it would look like this:

```
class Array[+T] {
  def update(x: T) = ...
}
```

The problematic combination is

- the covariant type parameter `T`
- which appears in parameter position of the method `update`.

Variance Checks (2)

The Scala compiler will check that there are no problematic combinations when compiling a class with variance annotations.

Roughly,

- *covariant* type parameters can only appear in method results.
- *contravariant* type parameters can only appear in method parameters.
- *invariant* type parameters can appear anywhere.

The precise rules are a bit more involved, fortunately the Scala compiler performs them for us.

Variance-Checking the Function Trait

Let's have a look again at Function1:

```
trait Function1[-T, +U]
  def apply(x: T): U
```

Here,

- `T` is contravariant and appears only as a method parameter type
- `U` is covariant and appears only as a method result type

So the method is checks out OK.

Variance and Lists

Let's get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.

Variance and Lists

Let's get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.

Here are the essential modifications:

```
trait List[+T]
  ...
object Empty extends List[Nothing]
  ...
```

Making Classes Covariant

Sometimes, we have to put in a bit of work to make a class covariant.

Consider adding a `prepend` method to `List` which prepends a given element, yielding a new list.

A first implementation of `prepend` could look like this:

```
trait List[+T]
  def prepend(elem: T): List[T] = Node(elem, this)
```

But that does not work!

Exercise

Why does the following code not type-check?

```
trait List[+T]
  def prepend(elem: T): List[T] = Node(elem, this)
```

Possible answers:

0 `prepend` turns `List` into a mutable class.

0 `prepend` fails variance checking.

0 `prepend`'s right-hand side contains a type error.

Exercise

Why does the following code not type-check?

```
trait List[+T]
  def prepend(elem: T): List[T] = Node(elem, this)
```

Possible answers:

0 `prepend` turns `List` into a mutable class.

0 `prepend` fails variance checking.

0 `prepend`'s right-hand side contains a type error.

Prepend Violates LSP

Indeed, the compiler is right to throw out `List` with `prepend`, because it violates the Liskov Substitution Principle:

Here's something one can do with a list `xs` of type `List[Fruit]`:

```
xs.prepend(Orange)
```

But the same operation on a list `ys` of type `List[Apple]` would lead to a type error:

```
ys.prepend(Apple)
           ^ type mismatch
           required: Apple
           found   : Orange
```

So, `List[Apple]` cannot be a subtype of `List[Fruit]`.

Lower Bounds

But `prepend` is a natural method to have on immutable lists!

Q: How can we make it variance-correct?

Lower Bounds

But `prepend` is a natural method to have on immutable lists!

Q: How can we make it variance-correct?

We can use a *lower bound*:

```
def prepend [U >: T] (elem: U): List[U] = Node(elem, this)
```

This passes variance checks, because:

- *covariant* type parameters may appear in *lower bounds* of method type parameters
- *contravariant* type parameters may appear in *upper bounds*.

Exercise

Assume `prepend` in trait `List` is implemented like this:

```
def prepend [U >: T] (elem: U): List[U] = Node(elem, this)
```

What is the result type of this function:

```
def f(xs: List[Apple], x: Orange) = xs.prepend(x)   ?
```

Possible answers:

0	does not type check
0	`List[Apple]`
0	`List[Orange]`
0	`List[Fruit]`
0	`List[Any]`

Exercise

Assume `prepend` in trait `List` is implemented like this:

```
def prepend [U >: T] (elem: U): List[U] = Node(elem, this)
```

What is the result type of this function:

```
def f(xs: List[Apple], x: Orange) = xs.prepend(x)   ?
```

Possible answers:

0	does not type check
0	`List[Apple]`
0	`List[Orange]`
0	`List[Fruit]`
0	`List[Any]`

Exercise

Assume `prepend` in trait `List` is implemented like this:

```
def prepend [U >: T] (elem: U): List[U] = Node(elem, this)
```

What is the result type of this function:

```
def f(xs: List[Apple], x: Orange) = xs.prepend(x)   ?
```

Possible answers:

O	does not type check
O	`List[Apple]`
O	`List[Orange]`
X	`List[Fruit]`
O	`List[Any]`

EPFL

Other Collections

Principles of Functional Programming

Other Sequences

We have seen that lists are *linear*: Access to the first element is much faster than access to the middle or end of a list.

The Scala library also defines an alternative sequence implementation, `Vector`.

This one has more evenly balanced access patterns than `List`.

Operations on Vectors

Vectors are created analogously to lists:

```
val nums = Vector(1, 2, 3, -88)
val people = Vector("Bob", "James", "Peter")
```

They support the same operations as lists, with the exception of `::`

Instead of `x :: xs`, there is

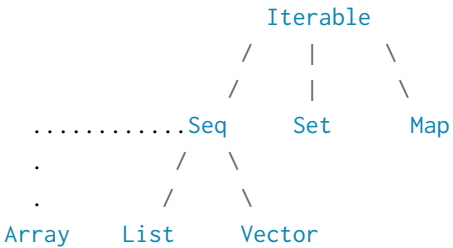
`x +: xs`	Create a new vector with leading element `x`, followed by all elements of `xs`.
`xs :+ x`	Create a new vector with trailing element `x`, preceded by all elements of `xs`.

(Note that the `:` always points to the sequence.)

Collection Hierarchy

A common base class of `List` and `Vector` is `Seq`, the class of all *sequences*.

`Seq` itself is a subclass of `Iterable`.



Arrays and Strings

Arrays and Strings support the same operations as `Seq` and can implicitly be converted to sequences where needed.

(They cannot be subclasses of `Seq` because they come from Java)

```
val xs: Array[Int] = Array(1, 2, 3)
xs.map(x => 2 * x)

val ys: String = "Hello world!"
ys.filter(_.isUpper)
```

Ranges

Another simple kind of sequence is the *range*.

It represents a sequence of evenly spaced integers.

Three operators:

`to` (inclusive), `until` (exclusive), `by` (to determine step value):

```
val r: Range = 1 until 5
val s: Range = 1 to 5
1 to 10 by 3
6 to 1 by -2
```

A `Range` is represented as a single object with three fields: lower bound, upper bound, step value.

Some more Sequence Operations:

`xs.exists(p)`	`true` if there is an element `x` of `xs` such that `p(x)` holds, `false` otherwise.
`xs.forall(p)`	`true` if `p(x)` holds for all elements `x` of `xs`, `false` otherwise.
`xs.zip(ys)`	A sequence of pairs drawn from corresponding elements of sequences `xs` and `ys`.
`xs.unzip`	Splits a sequence of pairs `xs` into two sequences consisting of the first, respectively second halves of all pairs.
`xs.flatMap(f)`	Applies collection-valued function `f` to all elements of `xs` and concatenates the results
`xs.sum`	The sum of all elements of this numeric collection.
`xs.product`	The product of all elements of this numeric collection
`xs.max`	The maximum of all elements of this collection (an `Ordering` must exist)
`xs.min`	The minimum of all elements of this collection

Example: Combinations

To list all combinations of numbers `x` and `y` where `x` is drawn from `1..M` and `y` is drawn from `1..N`:

```
(1 to M).flatMap(x =>
```

Example: Combinations

To list all combinations of numbers `x` and `y` where `x` is drawn from `1..M` and `y` is drawn from `1..N`:

```
(1 to M).flatMap(x => (1 to N).map(y => (x, y)))
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map(xy => xy._1 * xy._2).sum
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map(xy => xy._1 * xy._2).sum
```

An alternative way to write this is with a *pattern matching function value*.

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map { case (x, y) => x * y }.sum
```

Generally, the function value

```
{ case p1 => e1 ... case pn => en }
```

is equivalent to

```
x => x match { case p1 => e1 ... case pn => en }
```

Example: Scalar Product

For simple tuple decomposition, the `case` prefix in the pattern can be omitted.

So, the previous code can be simplified to:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map((x, y) => x * y).sum
```

Or, even simpler

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map(_ * _).sum
```

Exercise:

A number `n` is *prime* if the only divisors of `n` are `1` and `n` itself.

What is a high-level way to write a test for primality of numbers? For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean = ???
```

Exercise:

A number `n` is *prime* if the only divisors of `n` are `1` and `n` itself.

What is a high-level way to write a test for primality of numbers? For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean =
  (2 to n - 1).forall(d => n % d == 0)
```

EPFL

Combinatorial Search and For-Expressions

Principles of Functional Programming

Handling Nested Sequences

We can extend the usage of higher order functions on sequences to many calculations which are usually expressed using nested loops.

Example: Given a positive integer `n`, find all pairs of positive integers `i` and `j`, with `1 <= j < i < n` such that `i + j` is prime.

For example, if `n = 7`, the sought pairs are

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

Algorithm

A natural way to do this is to:

- Generate the sequence of all pairs of integers `(i, j)` such that `1 <= j < i < n`.
- Filter the pairs for which `i + j` is prime.

One natural way to generate the sequence of pairs is to:

- Generate all the integers `i` between `1` and `n` (excluded).
- For each integer `i`, generate the list of pairs `(i, 1), ..., (i, i-1)`.

This can be achieved by combining `until` and `map`:

```
(1 until n).map(i =>
  (1 until i).map(j => (i, j)))
```

Generate Pairs

The previous step gave a sequence of sequences, let's call it `xss`.

We can combine all the sub-sequences using `foldRight` with `++`:

```
xss.foldRight(Seq[Int]())(_ ++ _)
```

Or, equivalently, we use the built-in method `flatten`

```
xss.flatten
```

This gives:

```
((1 until n).map(i =>
  (1 until i).map(j => (i, j)))).flatten
```

Generate Pairs (2)

Here's a useful law:

```
xs.flatMap(f)  =  xs.map(f).flatten
```

Hence, the above expression can be simplified to

```
(1 until n).flatMap(i =>
    (1 until i).map(j => (i, j)))
```

Assembling the pieces

By reassembling the pieces, we obtain the following expression:

```
(1 until n)
  .flatMap(i => (1 until i).map(j => (i, j)))
  .filter((x, y) => isPrime(x + y))
```

This works, but makes most people's head hurt.

Is there a simpler way?

For-Expressions

Higher-order functions such as `map`, `flatMap` or `filter` provide powerful constructs for manipulating lists.

But sometimes the level of abstraction required by these function make the program difficult to understand.

In this case, Scala's `for` expression notation can help.

For-Expression Example

Let `persons` be a list of elements of class `Person`, with fields `name` and `age`.

```
case class Person(name: String, age: Int)
```

To obtain the names of persons over 20 years old, you can write:

```
for p <- persons if p.age > 20
yield p.name
```

which is equivalent to:

```
persons
  .filter(p => p.age > 20)
  .map(p => p.name)
```

The for-expression is similar to loops in imperative languages, except that it builds a list of the results of all iterations.

Syntax of For

A for-expression is of the form

```
for s yield e
```

where `s` is a sequence of *generators* and *filters*, and `e` is an expression whose value is returned by an iteration.

- A *generator* is of the form `p <- e`, where `p` is a pattern and `e` an expression whose value is a collection.
- A *filter* is of the form `if f` where `f` is a boolean expression.
- The sequence must start with a generator.
- If there are several generators in the sequence, the last generators vary faster than the first.

Use of For

Here are two examples which were previously solved with higher-order functions:

Given a positive integer `n`, find all the pairs of positive integers `(i, j)` such that `1 <= j < i < n`, and `i + j` is prime.

```
for
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =
```

Exercise

Write a version of `scalarProduct` (see last session) that makes use of a `for`:

```
def scalarProduct(xs: List[Double], ys: List[Double]) : Double =

  (for ((x, y) <- xs.zip(ys)) yield x * y).sum
```

EPFL

Combinatorial Search Example

Principles of Functional Programming

Sets

Sets are another basic abstraction in the Scala collections.

A set is written analogously to a sequence:

```
val fruit = Set("apple", "banana", "pear")
val s = (1 to 6).toSet
```

Most operations on sequences are also available on sets:

```
s.map(_ + 2)
fruit.filter(_.startsWith("app"))
s.nonEmpty
```

(see `Iterables` Scaladoc for a list of all supported operations)

Sets vs Sequences

The principal differences between sets and sequences are:

1. Sets are unordered; the elements of a set do not have a predefined order in which they appear in the set

2. sets do not have duplicate elements:

```
s.map(_ / 2  )      // Set(2, 0, 3, 1)
```

3. The fundamental operation on sets is `contains`:

```
s.contains(5)       // true
```

Example: N-Queens

The eight queens problem is to place eight queens on a chessboard so that no queen is threatened by another.

- In other words, there can't be two queens in the same row, column, or diagonal.

We now develop a solution for a chessboard of any size, not just 8.

One way to solve the problem is to place a queen on each row.

Once we have placed `k - 1` queens, one must place the `k`th queen in a column where it's not "in check" with any other queen on the board.

Algorithm

We can solve this problem with a recursive algorithm:

- Suppose that we have already generated all the solutions consisting of placing `k-1` queens on a board of size `n`.
- Each solution is represented by a list (of length `k-1`) containing the numbers of columns (between `0` and `n-1`).
- The column number of the queen in the `k-1`th row comes first in the list, followed by the column number of the queen in row `k-2`, etc.
- The solution set is thus represented as a set of lists, with one element for each solution.
- Now, to place the `k`th queen, we generate all possible extensions of each solution preceded by a new queen:

Implementation

```
def queens(n: Int) =
  def placeQueens(k: Int): Set[List[Int]] =
    if k == 0 then Set(List())
    else
      for
        queens <- placeQueens(k - 1)
        col <- 0 until n
        if isSafe(col, queens)
      yield col :: queens
  placeQueens(n)
```

Exercise

Write a function

```
def isSafe(col: Int, queens: List[Int]): Boolean
```

which tests if a queen placed in an indicated column `col` is secure amongst the other placed queens.

It is assumed that the new queen is placed in the next available row after the other placed queens (in other words: in row `queens.length`).

Maps

Principles of Functional Programming

Map

Another fundamental collection type is the *map*.

A map of type `Map[Key, Value]` is a data structure that associates keys of type `Key` with values of type `Value`.

Examples:

```
val romanNumerals = Map("I" -> 1, "V" -> 5, "X" -> 10)
val capitalOfCountry = Map("US" -> "Washington", "Switzerland" -> "Bern")
```

Maps are Iterables

Class `Map[Key, Value]` extends the collection type `Iterable[(Key, Value)]`.

Therefore, maps support the same collection operations as other iterables do. Example:

```
val countryOfCapital = capitalOfCountry.map((x, y) => (y, x))
                     // Map("Washington" -> "US", "Bern" -> "Switzerland")
```

Note that maps extend iterables of key/value *pairs*.

In fact, the syntax `key -> value` is just an alternative way to write the pair `(key, value)`.

Maps are Functions

Class `Map[Key, Value]` also extends the function type `Key => Value`, so maps can be used everywhere functions can.

In particular, maps can be applied to key arguments:

```
capitalOfCountry("US")          // "Washington"
```

Querying Map

Applying a map to a non-existing key gives an error:

```
capitalOfCountry("Andorra")
  // java.util.NoSuchElementException: key not found: Andorra
```

To query a map without knowing beforehand whether it contains a given key, you can use the `get` operation:

```
capitalOfCountry.get("US")      // Some("Washington")
capitalOfCountry.get("Andorra") // None
```

The result of a `get` operation is an `Option` value.

The `Option` Type

The `Option` type is defined as:

```
trait Option[+A]
case class Some[+A](value: A) extends Option[A]
object None extends Option[Nothing]
```

The expression `map.get(key)` returns

- `None` if `map` does not contain the given `key`,
- `Some(x)` if `map` associates the given `key` with the value `x`.

Decomposing Option

Since options are defined as case classes, they can be decomposed using pattern matching:

```
def showCapital(country: String) = capitalOfCountry.get(country) match
  case Some(capital) => capital
  case None => "missing data"

showCapital("US")      // "Washington"
showCapital("Andorra") // "missing data"
```

Options also support quite a few operations of the other collections.

I invite you to try them out!

Updating Maps

Functional updates of a map are done with the + and ++ operations:

`m + (k -> v)`	The map that takes key ‘k‘ to value ‘v‘ and is otherwise equal to ‘m‘
`m ++ kvs`	The map ‘m‘ updated via ‘+‘ with all key/value pairs in ‘kvs‘

These operations are purely functional. For instance,

```
val m1 = Map("red" -> 1, "blue" -> 2)   > m1 = Map(red -> 1, blue -> 2)
val m2 = m1 + ("blue" -> 3)             > m2 = Map(red -> 1, blue -> 3)
m1                                      > Map(red -> 1, blue -> 2)
```

Sorted and GroupBy

Two useful operation of SQL queries in addition to for-expressions are `groupBy` and `orderBy`.

`orderBy` on a collection can be expressed by `sortWith` and `sorted`.

```
val fruit = List("apple", "pear", "orange", "pineapple")
fruit.sortWith(_.length < _.length)  // List("pear", "apple", "orange", "pineap
fruit.sorted                         // List("apple", "orange", "pear", "pineap
```

`groupBy` is available on Scala collections. It partitions a collection into a map of collections according to a *discriminator function* `f`.

Example:

```
fruit.groupBy(_.head)   //> Map(p -> List(pear, pineapple),
                        //|     a -> List(apple),
                        //|     o -> List(orange))
```

Putting the Pieces Together

Principles of Functional Programming

Task

Phone keys have mnemonics assigned to them.

```
val mnemonics = Map(
      '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
      '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")
```

Assume you are given a dictionary `words` as a list of words.

Design a method `encode` such that

```
encode(phoneNumber)
```

produces all phrases of words that can serve as mnemonics for the phone number.

Example: The phone number "7225247386" should have the mnemonic `Scala is fun` as one element of the set of solution phrases.

Outline

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] = ???

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = ???

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] = ???

  /** All ways to encode a number as a list of words */
  def encode(number: String): Set[List[String]] = ???
}
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String =
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] =
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] =
    words.groupBy(wordCode).withDefaultValue(Nil)
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
```

Idea: use divide and conquer

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then ???
  else ???
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else ???
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- ???
      rest <- ???
    yield word :: rest
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- wordsForNum(number.take(splitPoint))
      rest <- ???
    yield word :: rest
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- wordsForNum(number.take(splitPoint))
      rest <- encode(number.drop(splitPoint))
    yield word :: rest
```

Testing It

A test program:

```
@main def code(number: String) =
  val coder = Coder(List(
    "Scala", "Python", "Ruby", "C",
    "rocks", "socks", "sucks", "works", "pack"))
  coder.encode(number).map(_.mkString(" "))
```

A sample run:

```
> scala code "7225276257"
HashSet(Scala rocks, pack C rocks, pack C socks, Scala socks)
```

Background

This example was taken from:

> *Lutz Prechelt: An Empirical Comparison of Seven Programming Languages. IEEE Computer 33(10): 23-29 (2000)*

Tested with Tcl, Python, Perl, Rexx, Java, C++, C.

Code size medians:

- 100 loc for scripting languages
- 200-300 loc for the others

Benefits

Scala's immutable collections are:

- *easy to use*: few steps to do the job.
- *concise*: one word replaces a whole loop.
- *safe*: type checker is really good at catching errors.
- *fast*: collection ops are tuned, can be parallelized.
- *universal*: one vocabulary to work on all kinds of collections.

This makes them an attractive tool for software development