Decomposition

Principles of Functional Programming

Decomposition

Suppose you want to write a small interpreter for arithmetic expressions.

To keep it simple, let's restrict ourselves to numbers and additions.

Expressions can be represented as a class hierarchy, with a base trait `Expr` and two subclasses, `Number` and `Sum`.

To treat an expression, it's necessary to know the expression's shape and its components.

This brings us to the following implementation.

Expressions

```
trait Expr:
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr

class Number(n: Int) extends Expr:
  def isNumber = true
  def isSum = false
  def numValue = n
  def leftOp = throw Error("Number.leftOp")
  def rightOp = throw Error("Number.rightOp")
```

Expressions (2)

```
class Sum(e1: Expr, e2: Expr) extends Expr:
  def isNumber = false
  def isSum = true
  def numValue = throw Error("Sum.numValue")
  def leftOp = e1
  def rightOp = e2
```

Evaluation of Expressions

You can now write an evaluation function as follows.

```
def eval(e: Expr): Int =
  if e.isNumber then e.numValue
  else if e.isSum then eval(e.leftOp) + eval(e.rightOp)
  else throw Error("Unknown expression " + e)
```

Problem: Writing all these classification and accessor functions quickly becomes tedious!

Problem: There's no static guarantee you use the right accessor functions. You might hit an `Error` case if you are not careful.

Adding New Forms of Expressions

So, what happens if you want to add new expression forms, say

```
class Prod(e1: Expr, e2: Expr) extends Expr   // e1 * e2
class Var(x: String) extends Expr             // Variable ‘x’
```

You need to add methods for classification and access to all classes defined above.

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

0	9
0	10
0	19
0	25
0	35
0	40

Question

To integrate `Prod` and `Var` into the hierarchy, how many new method definitions do you need?

(including method definitions in `Prod` and `Var` themselves, but not counting methods that were already given on the slides)

Possible Answers

0	9
0	10
0	19
0	25
0	35
0	40

Non-Solution: Type Tests and Type Casts

A "hacky" solution could use type tests and type casts.

Scala let's you do these using methods defined in class `Any`:

```
def isInstanceOf[T]: Boolean  // checks whether this object's type conforms to
def asInstanceOf[T]: T        // treats this object as an instance of type 'T'
                              // throws 'ClassCastException' if it isn't.
```

These correspond to Java's type tests and casts

Scala	Java
`x.isInstanceOf[T]`	`x instanceof T`
`x.asInstanceOf[T]`	`(T) x`

But their use in Scala is discouraged, because there are better alternatives.

Eval with Type Tests and Type Casts

Here's a formulation of the `eval` method using type tests and casts:

```
def eval(e: Expr): Int =
  if e.isInstanceOf[Number] then
    e.asInstanceOf[Number].numValue
  else if e.isInstanceOf[Sum] then
    eval(e.asInstanceOf[Sum].leftOp)
    + eval(e.asInstanceOf[Sum].rightOp)
  else throw Error("Unknown expression " + e)
```

This is ugly and potentially unsafe.

Solution 1: Object-Oriented Decomposition

For example, suppose that all you want to do is *evaluate* expressions.

You could then define:

```
trait Expr:
  def eval: Int

class Number(n: Int) extends Expr:
  def eval: Int = n

class Sum(e1: Expr, e2: Expr) extends Expr:
  def eval: Int = e1.eval + e2.eval
```

But what happens if you'd like to display expressions now?

You have to define new methods in all the subclasses.

Assessment of OO Decomposition

- OO decomposition mixes *data* with *operations* on the data.
- This can be the right thing if there's a need for encapsulation and data abstraction.
- On the other hand, it increases complexity(*) and adds new dependencies to classes.
- It makes it easy to add new kinds of data but hard to add new kinds of operations.

(*) In the literal sense of the word:
complex = plaited, woven together

Thus, complexity arises from mixing several things together.

Limitations of OO Decomposition

OO decomposition only works well if operations are on a *single* object.

What if you want to simplify expressions, say using the rule:

```
a * b + a * c   ->   a * (b + c)
```

Problem: This is a non-local simplification. It cannot be encapsulated in the method of a single object.

You are back to square one; you need test and access methods for all the different subclasses.

Pattern Matching

Principles of Functional Programming

Reminder: Decomposition

The task we are trying to solve is find a general and convenient way to access heterogeneous data in a class hierarchy.

Attempts seen previously:

- *Classification and access methods*: quadratic explosion
- *Type tests and casts*: unsafe, low-level
- *Object-oriented decomposition*: causes coupling between data and operations, need to touch all classes to add a new method.

Solution 2: Functional Decomposition with Pattern Matching

Observation: the sole purpose of test and accessor functions is to *reverse* the construction process:

- Which subclass was used?
- What were the arguments of the constructor?

This situation is so common that many functional languages, Scala included, automate it.

Case Classes

A *case class* definition is similar to a normal class definition, except that it is preceded by the modifier `case`. For example:

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Like before, this defines a trait `Expr`, and two concrete subclasses `Number` and `Sum`.

However, these classes are now empty. So how can we access the members?

Pattern Matching

Pattern matching is a generalization of `switch` from C/Java to class hierarchies.

It's expressed in Scala using the keyword `match`.

Example

```
def eval(e: Expr): Int = e match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

Match Syntax

Rules:

- `match` is preceded by a selector expression and is followed by a sequence of *cases*, `pat => expr`.
- Each case associates an *expression* `expr` with a *pattern* `pat`.
- A `MatchError` exception is thrown if no pattern matches the value of the selector.

Forms of Patterns

Patterns are constructed from:

- *constructors*, e.g. `Number`, `Sum`,
- *variables*, e.g. `n`, `e1`, `e2`,
- *wildcard patterns* `_`,
- *constants*, e.g. `1`, `true`.
- *type tests*, e.g. `n: Number`

Variables always begin with a lowercase letter.

The same variable name can only appear once in a pattern. So, `Sum(x, x)` is not a legal pattern.

Names of constants begin with a capital letter, with the exception of the reserved words `null`, `true`, `false`.

Evaluating Match Expressions

An expression of the form

$$\texttt{e match } \{ \texttt{ case } p_1 \texttt{ => } e_1 \texttt{ ... case } p_n \texttt{ => } e_n \texttt{ } \}$$

matches the value of the selector e with the patterns $p_1, ..., p_n$ in the order in which they are written.

The whole match expression is rewritten to the right-hand side of the first case where the pattern matches the selector *e*.

References to pattern variables are replaced by the corresponding parts in the selector.

What Do Patterns Match?

- A constructor pattern $C(p_1, ..., p_n)$ matches all the values of type C (or a subtype) that have been constructed with arguments matching the patterns $p_1, ..., p_n$.
- A variable pattern x matches any value, and *binds* the name of the variable to this value.
- A constant pattern c matches values that are equal to c (in the sense of ==)

Example

Example

```
eval(Sum(Number(1), Number(2)))
```

→

```
Sum(Number(1), Number(2)) match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

→

```
eval(Number(1)) + eval(Number(2))
```

Example (2)

$\rightarrow$

```
Number(1) match
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
+ eval(Number(2))
```

$\rightarrow$

```
1 + eval(Number(2))
```

$\twoheadrightarrow$

```
3
```

Pattern Matching and Methods

Of course, it's also possible to define the evaluation function as a method of the base trait.

Example

```
trait Expr:
  def eval: Int = this match
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
```

Exercise

Write a function `show` that uses pattern matching to return the representation of a given expressions as a string.

```
def show(e: Expr): String = ???
```

Exercise (Optional, Harder)

Add case classes `Var` for variables `x` and `Prod` for products `x * y` as discussed previously.

Change your `show` function so that it also deals with products.

Pay attention you get operator precedence right but to use as few parentheses as possible.

Example

```
Sum(Prod(2, Var("x")), Var("y"))
```

should print as "`2 * x + y`". But

```
Prod(Sum(2, Var("x")), Var("y"))
```

should print as "`(2 + x) * y`".

Lists

Principles of Functional Programming

Lists

The list is a fundamental data structure in functional programming.

A list having $x_1, \ldots, x_n$ as elements is written `List(`$x_1, \ldots, x_n$`)`

Example

```
val fruit  = List("apples", "oranges", "pears")
val nums   = List(1, 2, 3, 4)
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty  = List()
```

There are two important differences between lists and arrays.

- Lists are immutable — the elements of a list cannot be changed.
- Lists are recursive, while arrays are flat.

Lists

```
val fruit  = List("apples", "oranges", "pears")
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
```

The List Type

Like arrays, lists are homogeneous: the elements of a list must all have the same type.

The type of a list with elements of type `T` is written `scala.List[T]` or shorter just `List[T]`

Example

```
val fruit: List[String]    = List("apples", "oranges", "pears")
val nums : List[Int]       = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Nothing]   = List()
```

Constructors of Lists

All lists are constructed from:

- the empty list `Nil`, and
- the construction operation `::` (pronounced *cons*):
  `x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`.

For example:

```
fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
empty = Nil
```

Right Associativity

Convention: Operators ending in ":" associate to the right.

`A :: B :: C` is interpreted as `A :: (B :: C)`.

We can thus omit the parentheses in the definition above.

Example

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

Operations on Lists

All operations on lists can be expressed in terms of the following three:

`head`	the first element of the list
`tail`	the list composed of all the elements except the first.
`isEmpty`	'true' if the list is empty, 'false' otherwise.

These operations are defined as methods of objects of type `List`. For example:

```
fruit.head      == "apples"
fruit.tail.head == "oranges"
diag3.head      == List(1, 0, 0)
empty.head      == throw NoSuchElementException("head of empty list")
```

List Patterns

It is also possible to decompose lists with pattern matching.

`Nil`	The `Nil` constant
`p :: ps`	A pattern that matches a list with a `head` matching `p` and a `tail` matching `ps`.
`List(p1, ..., pn)`	same as `p1 :: ... :: pn :: Nil`

Example

`1 :: 2 :: xs`	Lists of that start with `1` and then `2`
`x :: Nil`	Lists of length 1
`List(x)`	Same as `x :: Nil`
`List()`	The empty list, same as `Nil`
`List(2 :: xs)`	A list that contains as only element another list that starts with `2`.

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length `L` of the lists it matches?

- 0 `L == 3`
- 0 `L == 4`
- 0 `L == 5`
- 0 `L >= 3`
- 0 `L >= 4`
- 0 `L >= 5`

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length `L` of the lists it matches?

O	`L == 3`
O	`L == 4`
O	`L == 5`
X	`L >= 3`
O	`L >= 4`
O	`L >= 5`

Sorting Lists

Suppose we want to sort a list of numbers in ascending order:

- One way to sort the list `List(7, 3, 9, 2)` is to sort the tail `List(3, 9, 2)` to obtain `List(2, 3, 9)`.
- The next step is to insert the head `7` in the right place to obtain the result `List(2, 3, 7, 9)`.

This idea describes *Insertion Sort* :

```
def isort(xs: List[Int]): List[Int] = xs match
  case List()  => List()
  case y :: ys => insert(y, isort(ys))
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case List() => ???
  case y :: ys => ???
```

What is the worst-case complexity of insertion sort relative to the length of the input list N?

- O(1) the sort takes constant time
- O(N) proportional to N
- O(N log(N)) proportional to N log(N)
- O(N * N) proportional to N * N

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

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

What is the worst-case complexity of insertion sort relative to the length of the input list `N`?

```
O      the sort takes constant time
X      proportional to N
O      proportional to N * log(N)
O      proportional to N * N
```

Enums

Principles of Functional Programming

Pure Data

In the previous sessions, you have learned how to model data with class hierarchies.

Classes are essentially bundles of functions operating on some common values represented as fields.

They are a very useful abstraction, since they allow encapsulation of data.

But sometimes we just need to compose and decompose *pure data* without any associated functions.

Case classes and pattern matching work well for this task.

A Case Class Hierarchy

Here's our case class hierarchy for expressions again:

```
trait Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
```

This time we have put all case classes in the `Expr` companion object, in order not to pollute the global namespace.

So it's `Expr.Number(1)` instead of `Number(1)`, for example.

One can still "pull out" all the cases using an import.

```
import Expr.*
```

A Case Class Hierarchy

Here's our case class hierarchy for expressions again:

```
trait Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr
```

Pure data definitions like these are called *algebraic data types*, or ADTs for short.

They are very common in functional programming.

To make them even more convenient, Scala offers some special syntax.

Enums for ADTs

An *enum* enumerates all the cases of an ADT *and nothing else*.

Example

```
enum Expr:
  case Var(s: String)
  case Number(n: Int)
  case Sum(e1: Expr, e2: Expr)
  case Prod(e1: Expr, e2: Expr)
```

This enum is equivalent to the case class hierarchy on the previous slide, but is shorter, since it avoids the repetitive `class ... extends Expr` notation.

Pattern Matching on ADTs

Match expressions can be used on enums as usual.

For instance, to print expressions with proper parameterization:

```
def show(e: Expr): String = e match
  case Expr.Var(x) => x
  case Expr.Number(n) => n.toString
  case Expr.Sum(a, b) => s"${show(a)} + ${show(a)}}"
  case Expr.Prod(a, b) => s"${showP(a)} * ${showP(a)}"

def showP(e: Expr): String = e match
  case e: Sum => s"(${show(expr)})"
  case _ => show(expr)
```

Simple Enums

Cases of an `enum` can also be simple values, without any parameters.

Example

Define a `Color` type with values `Red`, `Green`, and `Blue`:

```
enum Color:
  case Red
  case Green
  case Blue
```

We can also combine several simple cases in one list:

```
enum Color:
  case Red, Green, Blue
```

Pattern Matching on Simple Enums

For pattern matching, simple cases count as constants:

```
enum DayOfWeek:
  case Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

import DayOfWeek.*

def isWeekend(day: DayOfWeek) = day match
  case Saturday | Sunday => true
  case _ => false
```

More Fun With Enums

Enumerations can take parameters and can define methods.

Example:

```
enum Direction(val dx: Int, val dy: Int):
  case Right extends Direction( 1,  0)
  case Up    extends Direction( 0,  1)
  case Left  extends Direction(-1,  0)
  case Down  extends Direction( 0, -1)

  def leftTurn = Direction.values((ordinal + 1) % 4)
end Direction

val r = Direction.Right
val u = x.leftTurn       // u = Up
val v = (u.dx, u.dy)     // v = (1, 0)
```

More Fun With Enums

Notes:

- Enumeration cases that pass parameters have to use an explicit `extends` clause
- The expression `e.ordinal` gives the ordinal value of the enum case `e`. Cases start with zero and are numbered consecutively.
- `values` is an immutable array in the companion object of an enum that contains all enum values.
- Only simple cases have `ordinal` numbers and show up in `values`, parameterized cases do not.

Enumerations Are Shorthands for Classes and Objects

The `Direction` enum is expanded by the Scala compiler to roughly the following structure:

```
abstract class Direction(val dx: Int, val dy: Int):
  def rightTurn = Direction.values((ordinal - 1) % 4)
object Direction:
  val Right = new Direction( 1,  0) {}
  val Up    = new Direction( 0,  1) {}
  val Left  = new Direction(-1,  0) {}
  val Down  = new Direction( 0, -1) {}
end Direction
```

There are also compiler-defined helper methods `ordinal` in the class and `values` and `valueOf` in the companion object.

Domain Modeling

ADTs and enums are particularly useful for domain modelling tasks where one needs to define a large number of data types without attaching operations.

Example: Modelling payment methods.

```
enum PaymentMethod:
  case CreditCard(kind: Card, holder: String, number: Long, expires: Date)
  case PayPal(email: String)
  case Cash

enum Card:
  case Visa, Mastercard, Amex
```

Summary

In this unit, we covered two uses of `enum` definitions:

- as a shorthand for hierarchies of case classes,
- as a way to define data types accepting alternative values,

The two cases can be combined: an enum can comprise parameterized and simple cases at the same time.

Enums are typically used for pure data, where all operations on such data are defined elsewhere.

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 in the next session examples 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

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 FtoO = Fruit => Orange
type AtoF = Apple => Fruit
```

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

- ○ `FtoO <: AtoF`
- ○ `AtoF <: FtoO`
- ○ `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 FtoO = Fruit => Orange
type AtoF = Apple => Fruit
```

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

- ○ `FtoO <: AtoF`
- ○ `AtoF <: FtoO`
- ○ `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
```

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]
  ...
```

Idealized Lists

Here a definition of lists that implements all the cases we have seen so far:

```
trait List[+T]:

  def isEmpty = this match
    case Nil => true
    case _ => false

  override def toString =
    def recur(prefix: String, xs: List[T]): String = xs match
      case x :: xs1 => s"$prefix$x${recur(", ", xs1)}"
      case Nil => ")"
    recur("List(", this)
```

Idealized Lists(2)

```
case class ::[+T](head: T, tail: List[T]) extends List[T]
case object Nil extends List[Nothing]

extension [T](x: T) def :: (xs: List[T]): List[T] = ::(x, xs)

object List:
  def apply() = Nil
  def apply[T](x: T) = x :: Nil
  def apply[T](x1: T, x2: T) = x1 :: x2 :: Nil
  ...
```

(We'll see later how to do with just a single `apply` method using a *vararg* parameter.)

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] = ::(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] = ::(elem, this)
```

Possible answers:

- `prepend` turns `List` into a mutable class.
- `prepend` fails variance checking.
- `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] = ::(elem, this)
```

Possible answers:

- `prepend` turns `List` into a mutable class.
- `prepend` fails variance checking.
- `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] = ::(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] = ::(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] = ::(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]`

Extension Methods

The need for a lower bound was essentially to decouple the new parameter of the class and the parameter of the newly created object. Using an extension method such as in `::` above, sidesteps the problem and is often simpler:

```
extension [T](x: T):
  def :: (xs: List[T]): List[T] = ::(x, xs)
```