Lecture 9 – Advanced Types

SWS121: Secure Programming

Jihyeok Park

PLRG

2024 Spring

Recall

- Generic Classes
- Generic Methods/Functions
- Type Bounds
- Variances
- Abstract Type Members
- Inner Classes

Contents

Contents

Intersection Types

The & operator is used to create an **intersection type**.

Intersection Types

The `&` operator is used to create an **intersection type**.

The type `A & B` represents values of **both** type `A` and `B` at the same time.

Intersection Types

The `&` operator is used to create an **intersection type**.

The type `A & B` represents values of **both** type `A` and `B` at the same time.

For example, consider the following code:

```
trait A { def foo(x: Int): Int }
trait B { def bar(x: Int): Int }

def f(x: A & B): Int = x.foo(10) + x.bar(20)
```

Intersection Types

The & operator is used to create an **intersection type**.

The type A & B represents values of **both** type A and B at the same time.

For example, consider the following code:

```
trait A { def foo(x: Int): Int }
trait B { def bar(x: Int): Int }

def f(x: A & B): Int = x.foo(10) + x.bar(20)
```

Since x is of type A & B, it can access both the foo in A and the bar in B.

Intersection Types

PLRG

The & operator is used to create an **intersection type**.

The type `A & B` represents values of **both** type `A` and `B` at the same time.

For example, consider the following code:

```
trait A { def foo(x: Int): Int }
trait B { def bar(x: Int): Int }

def f(x: A & B): Int = x.foo(10) + x.bar(20)
```

Since `x` is of type `A & B`, it can access both the `foo` in `A` and the `bar` in `B`.

We can call `f` with an object that implements both `A` and `B`.

```
class C extends A with B:
  def foo(x: Int): Int = x + 1
  def bar(x: Int): Int = x + 2

f(new C) // (10 + 1) + (20 + 2) = 33
```

Union Types

On the other hand, the | operator is used to create a **union type**.

Union Types

On the other hand, the `|` operator is used to create a **union type**.

The type `A | B` represents values of **either** type `A` or `B`.

Union Types

On the other hand, the | operator is used to create a **union type**.

The type `A | B` represents values of **either** type `A` or `B`.

For example, consider the following code:

```
case class Username(name: String)
case class Password(hash: String)

def getData(x: A | B): String = ???
```

Union Types

On the other hand, the | operator is used to create a **union type**.

The type `A | B` represents values of **either** type `A` or `B`.

For example, consider the following code:

```
case class Username(name: String)
case class Password(hash: String)

def getData(x: A | B): String = x match
  case Username(name) => name
  case Password(hash) => hash
```

You can use **pattern matching** to extract the value from the union type.

Union Types

On the other hand, the | operator is used to create a **union type**.

The type `A | B` represents values of **either** type `A` or `B`.

For example, consider the following code:

```
case class Username(name: String)
case class Password(hash: String)

def getData(x: A | B): String = x match
  case Username(name) => name
  case Password(hash) => hash
```

You can use **pattern matching** to extract the value from the union type.

We can call `getData` with either a `Username` or a `Password`.

```
getData(Username("alice")) // "alice"
getData(Password("x2ef3")) // "x2ef3"
```

Union Types – Inferece

The compiler assigns a **union type** to an expression only if such a type is explicitly declared.

Union Types – Inferece

The compiler assigns a **union type** to an expression only if such a type is explicitly declared.

For example, the following code does not infer a union type:

```
case class Username(name: String)
case class Password(hash: String)
val name = Username("alice")  // name: Username = Username("alice")
val pass = Password("x2ef3")  // pass: Password = Password("x2ef3")
```

Union Types – Inferece

The compiler assigns a **union type** to an expression only if such a type is explicitly declared.

For example, the following code does not infer a union type:

```
case class Username(name: String)
case class Password(hash: String)
val name = Username("alice")  // name: Username = Username("alice")
val pass = Password("x2ef3")  // pass: Password = Password("x2ef3")
```

The following code infers the `Object` type rather than a union type:

```
val x = if (true) name else pass
// x: Object = Username("alice")
```

Union Types – Inferece

The compiler assigns a **union type** to an expression only if such a type is explicitly declared.

For example, the following code does not infer a union type:

```
case class Username(name: String)
case class Password(hash: String)
val name = Username("alice")  // name: Username = Username("alice")
val pass = Password("x2ef3")  // pass: Password = Password("x2ef3")
```

The following code infers the `Object` type rather than a union type:

```
val x = if (true) name else pass
// x: Object = Username("alice")
```

To assign a union type to `x`, you need to explicitly declare it:

```
val x: Username | Password = if (true) name else pass
// x: Username | Password = Username("alice")
```

Union Types – Alternatives

Without union types, we can represent the same concept in two ways.

Union Types – Alternatives

Without union types, we can represent the same concept in two ways.

First, we can define a common supertype using a trait:

```
trait UsernameOrPassword
case class Username(name: String) extends UsernameOrPassword
case class Password(hash: String) extends UsernameOrPassword
```

Union Types – Alternatives

Without union types, we can represent the same concept in two ways.

First, we can define a common supertype using a trait:

```
trait UsernameOrPassword
case class Username(name: String) extends UsernameOrPassword
case class Password(hash: String) extends UsernameOrPassword
```

Second, we can use `enum` (algebraic data types, ADTs) to represent the union type because ADTs are tagged union (sum) types of product types:

```
enum UsernameOrPassword:
  case Username(name: String)
  case Password(hash: String)
```

To directly access the constructors for ADTs, you need to import them:

```
import UsernameOrPassword.*
```

Properties of Intersection and Union Types

Intersection and union types have the following properties:

Properties of Intersection and Union Types

Intersection and union types have the following properties:

- **Commutativity** for intersection and union types:

$$\texttt{A \& B} \equiv \texttt{B \& A} \quad \text{and} \quad \texttt{A | B} \equiv \texttt{B | A}$$

Properties of Intersection and Union Types

Intersection and union types have the following properties:

- **Commutativity** for intersection and union types:

$$\texttt{A \& B} \equiv \texttt{B \& A} \quad \text{and} \quad \texttt{A | B} \equiv \texttt{B | A}$$

- **Associativity** for intersection and union types:

$$\texttt{A \& (B \& C)} \equiv \texttt{(A \& B) \& C} \quad \text{and} \quad \texttt{A | (B | C)} \equiv \texttt{(A | B) | C}$$

Properties of Intersection and Union Types

Intersection and union types have the following properties:

- **Commutativity** for intersection and union types:

$$\texttt{A \& B} \equiv \texttt{B \& A} \quad \text{and} \quad \texttt{A | B} \equiv \texttt{B | A}$$

- **Associativity** for intersection and union types:

$$\texttt{A \& (B \& C)} \equiv \texttt{(A \& B) \& C} \quad \text{and} \quad \texttt{A | (B | C)} \equiv \texttt{(A | B) | C}$$

- **Distributivity** for intersection over union:

$$\texttt{A | (B \& C)} \equiv \texttt{(A | B) \& (A | C)} \quad \text{and}$$
$$\texttt{A \& (B | C)} \equiv \texttt{(A \& B) | (A \& C)}$$

Properties of Intersection and Union Types

Intersection and union types have the following properties:

- **Commutativity** for intersection and union types:

$$\texttt{A \& B} \equiv \texttt{B \& A} \quad \text{and} \quad \texttt{A | B} \equiv \texttt{B | A}$$

- **Associativity** for intersection and union types:

$$\texttt{A \& (B \& C)} \equiv \texttt{(A \& B) \& C} \quad \text{and} \quad \texttt{A | (B | C)} \equiv \texttt{(A | B) | C}$$

- **Distributivity** for intersection over union:

$$\texttt{A | (B \& C)} \equiv \texttt{(A | B) \& (A | C)} \quad \text{and}$$
$$\texttt{A \& (B | C)} \equiv \texttt{(A \& B) | (A \& C)}$$

- **Idempotence** for intersection and union types:

$$\texttt{A \& A} \equiv \texttt{A} \quad \text{and} \quad \texttt{A | A} \equiv \texttt{A}$$

Properties of Intersection and Union Types

Intersection and union types have the following properties:

- **Commutativity** for intersection and union types:

$$\texttt{A \& B} \equiv \texttt{B \& A} \quad \text{and} \quad \texttt{A | B} \equiv \texttt{B | A}$$

- **Associativity** for intersection and union types:

$$\texttt{A \& (B \& C)} \equiv \texttt{(A \& B) \& C} \quad \text{and} \quad \texttt{A | (B | C)} \equiv \texttt{(A | B) | C}$$

- **Distributivity** for intersection over union:

$$\texttt{A | (B \& C)} \equiv \texttt{(A | B) \& (A | C)} \quad \text{and}$$
$$\texttt{A \& (B | C)} \equiv \texttt{(A \& B) | (A \& C)}$$

- **Idempotence** for intersection and union types:

$$\texttt{A \& A} \equiv \texttt{A} \quad \text{and} \quad \texttt{A | A} \equiv \texttt{A}$$

- Intersection types have a **higher precedence** than union types:

$$\texttt{A \& B | C} \equiv \texttt{(A \& B) | C}$$

Contents

Self Types

Self-types are a way to declare that a trait must be mixed into another trait, and they **narrow** the type of `this` to be the type of mixed-in trait.

Self Types

Self-types are a way to declare that a trait must be mixed into another trait, and they **narrow** the type of `this` to be the type of mixed-in trait.

For example, `Tweeter` trait has `User` as a self-type:

```
trait User { def username: String }
trait Tweeter { this: User =>
  def tweet(msg: String): Unit = println(s"$msg by $username")
}
class VerifiedTweeter(val username: String) extends User with Tweeter
```

Self Types

Self-types are a way to declare that a trait must be mixed into another trait, and they **narrow** the type of `this` to be the type of mixed-in trait.

For example, `Tweeter` trait has `User` as a self-type:

```
trait User { def username: String }
trait Tweeter { this: User =>
  def tweet(msg: String): Unit = println(s"$msg by $username")
}
class VerifiedTweeter(val username: String) extends User with Tweeter
```

It means that we need to mix in `User` when we mix in `Tweeter`.

```
// error: illegal inheritance
case class InvalidTweeter(val username: String) extends Tweeter

// OK
case class ValidTweeter(val username: String) extends Tweeter with User

ValidTweeter("alice").tweet("Hello")    // "Hello by alice"
```

Self Types

We can use any other name instead of `this` for the **self-type**:

```
trait Tweeter { self: User => ... }
```

Self Types

We can use any other name instead of `this` for the **self-type**:

```
trait Tweeter { self: User => ... }
```

If we omit its type annotation, it does not restrict the type of `this`:

```
trait Tweeter { self => ... }     // self: Tweeter
```

Self Types

We can use any other name instead of `this` for the **self-type**:

```
trait Tweeter { self: User => ... }
```

If we omit its type annotation, it does not restrict the type of `this`:

```
trait Tweeter { self => ... }     // self: Tweeter
```

We can refer to `this` of the outer class in the inner class using **self-types**.

```
case class A { self =>
  val name = "Alice"
  case class B {
    val name = self.name
    def printName: Unit =
      println(s"Inner name: ${name}")
      println(s"Inner name: ${this.name}")
      println(s"Outer name: ${self.name}")
  }
}
```

Self Types – Dependency Injection

Self-types are useful for **dependency injection** and it is often called the **cake pattern** in Scala.

```scala
trait UserTrait { def name: String }
// TweeterTrait depends on UserTrait without real implementation
trait TweeterTrait { this: UserTrait =>
  def tweet(msg: String): Unit = println(s"$msg by $name")
}
```

Self Types – Dependency Injection

Self-types are useful for **dependency injection** and it is often called the **cake pattern** in Scala.

```
trait UserTrait { def name: String }
// TweeterTrait depends on UserTrait without real implementation
trait TweeterTrait { this: UserTrait =>
  def tweet(msg: String): Unit = println(s"$msg by $name")
}
```

We can mix in different implementations of `UserTrait` to `TweeterTrait`.

```
trait UserImpl1 extends UserTrait { val name = "Alice" }
object TweeterImpl1 extends TweeterTrait with UserImpl1
TweeterImpl1.tweet("Hello")    // "Hello by Alice"

trait UserImpl2 extends UserTrait { val name = "Bob" }
object TweeterImpl2 extends TweeterTrait with UserImpl2
TweeterImpl2.tweet("Hi")       // "Hi by Bob"
```

Self Types – Dependency Injection

Why we need **self-types** rather than **inheritance**?

Self Types – Dependency Injection

Why we need **self-types** rather than **inheritance**?

More fine-grained control over the encapsulation of the implementation.

Self Types – Dependency Injection

Why we need **self-types** rather than **inheritance**?

More fine-grained control over the encapsulation of the implementation.

For example, assume that we want to share the `f` method in `A` with `B` but not with `C`.

Self Types – Dependency Injection

Why we need **self-types** rather than **inheritance**?

More fine-grained control over the encapsulation of the implementation.

For example, assume that we want to share the `f` method in `A` with `B` but not with `C`.

We can use **self-types** to achieve this:

```
trait A { def f: Int }
trait B { self: A => def g: Int = f }
trait C extends B { def h: Int = f } // error: f is not accessible
```

Self Types – Dependency Injection

Why we need **self-types** rather than **inheritance**?

More fine-grained control over the encapsulation of the implementation.

For example, assume that we want to share the `f` method in `A` with `B` but not with `C`.

We can use **self-types** to achieve this:

```
trait A { def f: Int }
trait B { self: A => def g: Int = f }
trait C extends B { def h: Int = f } // error: f is not accessible
```

However, we cannot achieve the same with **inheritance**:

```
trait A { def f: Int }
trait B extends A { def g: Int = f }
trait C extends B { def h: Int = f } // No error -- breaks encapsulation
```

Contents

Opaque Types

Let us assume we want to define a module that offers arithmetic on numbers, which are represented by their logarithm.

Opaque Types

PLRG

Let us assume we want to define a module that offers arithmetic on numbers, which are represented by their logarithm.

We can define a class Logarithm with Double:

```
class Logarithm(val underlying: Double):
  def toDouble: Double = math.exp(underlying)
  def + (that: Logarithm): Logarithm =
    Logarithm(this.toDouble + that.toDouble)
  def * (that: Logarithm): Logarithm =
    new Logarithm(this.underlying + that.underlying)
object Logarithm:
  def apply(d: Double): Logarithm = new Logarithm(math.log(d))
```

```
val x = Logarithm(2.0)       // x.underlying = log(2.0) = 0.693147
val y = Logarithm(3.0)       // y.underlying = log(3.0) = 1.098612
println((x + y).toDouble)    // 5.0
println((x * y).toDouble)    // 6.0
```

Opaque Types

Let us assume we want to define a module that offers arithmetic on numbers, which are represented by their logarithm.

We can define a class `Logarithm` with `Double`:

```
class Logarithm(val underlying: Double):
  def toDouble: Double = math.exp(underlying)
  def + (that: Logarithm): Logarithm =
    Logarithm(this.toDouble + that.toDouble)
  def * (that: Logarithm): Logarithm =
    new Logarithm(this.underlying + that.underlying)
object Logarithm:
  def apply(d: Double): Logarithm = new Logarithm(math.log(d))
```

```
val x = Logarithm(2.0)      // x.underlying = log(2.0) = 0.693147
val y = Logarithm(3.0)      // y.underlying = log(3.0) = 1.098612
println((x + y).toDouble)   // 5.0
println((x * y).toDouble)   // 6.0
```

However, it has unnecessary **performance overhead** because of the boxing and unboxing of `Double` values.

Opaque Types

PLRG

We can use a **type alias** to remove the performance overhead:

```
object Logarithms:
  type Logarithm = Double
  def add(x: Logarithm, y: Logarithm): Logarithm =
    make(extract(x) + extract(y))
  def mul(x: Logarithm, y: Logarithm): Logarithm = x + y
  def make(d: Double): Logarithm = math.log(d)
  def extract(x: Logarithm): Double = math.exp(x)
```

```
import Logarithms.*
val x: Logarithm = make(2.0)  // x = log(2.0) = 0.693147
val y: Logarithm = make(3.0)  // y = log(3.0) = 1.098612
println(extract(add(x, y)))   // 5.0
println(extract(mul(x, y)))   // 6.0
```

Opaque Types

We can use a **type alias** to remove the performance overhead:

```
object Logarithms:
  type Logarithm = Double
  def add(x: Logarithm, y: Logarithm): Logarithm =
    make(extract(x) + extract(y))
  def mul(x: Logarithm, y: Logarithm): Logarithm = x + y
  def make(d: Double): Logarithm = math.log(d)
  def extract(x: Logarithm): Double = math.exp(x)
```

```
import Logarithms.*
val x: Logarithm = make(2.0)  // x = log(2.0) = 0.693147
val y: Logarithm = make(3.0)  // y = log(3.0) = 1.098612
println(extract(add(x, y)))   // 5.0
println(extract(mul(x, y)))   // 6.0
```

However, it make the equality `Logarithm = Double` **visible** to the users, who might misuse it by **accidentally mixing** `Logarithm` and `Double`.

```
val d: Double = x     // type checks AND leaks the equality!
```

Opaque Types

We can use **opaque types** to hide the equality and still remove the performance overhead of boxing and unboxing `Double` values:

```
object Logarithms:
  opaque type Logarithm = Double
  ...
```

```
import Logarithms.*
val x: Logarithm = make(2.0)  // x = log(2.0) = 0.693147
val y: Logarithm = make(3.0)  // y = log(3.0) = 1.098612
println(extract(add(x, y)))   // 5.0
println(extract(mul(x, y)))   // 6.0
```

Opaque Types

We can use **opaque types** to hide the equality and still remove the performance overhead of boxing and unboxing Double values:

```
object Logarithms:
  opaque type Logarithm = Double
  ...
```

```
import Logarithms.*
val x: Logarithm = make(2.0)  // x = log(2.0) = 0.693147
val y: Logarithm = make(3.0)  // y = log(3.0) = 1.098612
println(extract(add(x, y)))   // 5.0
println(extract(mul(x, y)))   // 6.0
```

Now, the equality Logarithm = Double is **hidden** from the users and the type system prevents the misuse of Logarithm and Double.

```
val d: Double = x     // error: found: Logarithm, required: Double
```

Contents

Structural Types

Structural types specify that types must have a certain structure.

Structural Types

Structural types specify that types must have a certain structure.

It is often called **duck typing** because it is based on the principle that if it looks like a duck and **quacks** like a duck, it must be a **duck**.

Structural Types

Structural types specify that types must have a certain structure.

It is often called **duck typing** because it is based on the principle that if it looks like a duck and **quacks** like a duck, it must be a **duck**.

For example, consider the following code:

```scala
class Duck { def fly = println("Duck flies") }
class Eagle { def fly = println("Eagle flies") }
class Dog { def walk = println("Dog walks") }
```

Structural Types

Structural types specify that types must have a certain structure.

It is often called **duck typing** because it is based on the principle that if it looks like a duck and **quacks** like a duck, it must be a **duck**.

For example, consider the following code:

```
class Duck { def fly = println("Duck flies") }
class Eagle { def fly = println("Eagle flies") }
class Dog { def walk = println("Dog walks") }
```

How can we define a method that takes any object that has a `fly` method without changing the definition of the classes?

Structural Types

Structural types specify that types must have a certain structure.

It is often called **duck typing** because it is based on the principle that if it looks like a duck and **quacks** like a duck, it must be a **duck**.

For example, consider the following code:

```
class Duck { def fly = println("Duck flies") }
class Eagle { def fly = println("Eagle flies") }
class Dog { def walk = println("Dog walks") }
```

How can we define a method that takes any object that has a `fly` method without changing the definition of the classes?

We can use a **structural type** to do so:

```
import scala.reflect.Selectable.reflectiveSelectable
def makeItFly(x: { def fly: Unit }): Unit = x.fly
makeItFly(new Duck)    // "Duck flies"
makeItFly(new Eagle)   // "Eagle flies"
makeItFly(new Dog)     // error: Dog does not have a fly method
```

Structural Types – Example

Let's use a **structural type** to define a `autoClose` method to automatically close a resource after using it.

Structural Types – Example

Let's use a **structural type** to define a autoClose method to automatically close a resource after using it.

```
import scala.reflect.Selectable.reflectiveSelectable

class File { def close = println("File closed") }
class InputStarem { def close = println("InputStream closed") }

type Closable = { def close: Unit }
def autoClose(resource: Closable)(op: Closable => Unit): Unit =
  try op(resource) finally resource.close
```

Structural Types – Example

Let's use a **structural type** to define a autoClose method to automatically close a resource after using it.

```
import scala.reflect.Selectable.reflectiveSelectable

class File { def close = println("File closed") }
class InputStarem { def close = println("InputStream closed") }

type Closable = { def close: Unit }
def autoClose(resource: Closable)(op: Closable => Unit): Unit =
  try op(resource) finally resource.close
```

```
autoClose(new File)(f => println("Reading from file"))
// "Reading from file"
// "File closed"
```

Structural Types – Example

Let's use a **structural type** to define a autoClose method to automatically close a resource after using it.

```
import scala.reflect.Selectable.reflectiveSelectable

class File { def close = println("File closed") }
class InputStarem { def close = println("InputStream closed") }

type Closable = { def close: Unit }
def autoClose(resource: Closable)(op: Closable => Unit): Unit =
  try op(resource) finally resource.close
```

```
autoClose(new File)(f => println("Reading from file"))
// "Reading from file"
// "File closed"
```

```
autoClose(new InputStarem)(in => println("Reading from input stream"))
// "Reading from input stream"
// "InputStream closed"
```

Contents

Type Lambdas

Type lambdas are a way to define anonymous type constructors.

Type Lambdas

Type lambdas are a way to define anonymous type constructors.

For example, consider the following code:

```
type MapInt = [X] =>> Map[Int, X]
val m1: MapInt[String] = Map(1 -> "one", 2 -> "two")
val m2: MapInt[Double] = Map(1 -> 1.0, 2 -> 2.0)
```

Type Lambdas

Type lambdas are a way to define anonymous type constructors.

For example, consider the following code:

```
type MapInt = [X] =>> Map[Int, X]
val m1: MapInt[String] = Map(1 -> "one", 2 -> "two")
val m2: MapInt[Double] = Map(1 -> 1.0, 2 -> 2.0)
```

A **parameterized type** is regarded as a shorthand for a type lambda:

```
type T[X] = R
// is equivalent to
type T = [X] =>> R
```

Type Lambdas

Type lambdas are a way to define anonymous type constructors.

For example, consider the following code:

```
type MapInt = [X] =>> Map[Int, X]
val m1: MapInt[String] = Map(1 -> "one", 2 -> "two")
val m2: MapInt[Double] = Map(1 -> 1.0, 2 -> 2.0)
```

A **parameterized type** is regarded as a shorthand for a type lambda:

```
type T[X] = R
// is equivalent to
type T = [X] =>> R
```

The body of a type lambda can again be a type lambda:

```
type Pair = [X] =>> [Y] =>> (X, Y)
val p: Pair[Int][String] = (1, "one")
```

Type Lambdas – Example

For example, let's define our own `Try` type using a type lambda.

Type Lambdas – Example

For example, let's define our own Try type using a type lambda.

```scala
type MyTry = [X] =>> Either[Throwable, X]

val myTryInt: MyTry[Int] = Right(10)
val myTryStr: MyTry[String] = Right("Hello")
val myTryLeft: MyTry[Int] = Left(new Exception("Error"))

println(myTryInt)    // Right(10)
println(myTryStr)    // Right("Hello")
println(myTryLeft)   // Left(java.lang.Exception: Error)
```

The left side of Either is always a Throwable and the right side is remained as a **blank** to be filled with any type.

Contents

Polymorphic Function Types

Scala supports **polymorphic methods** as follows:

```scala
def reverse[A](xs: List[A]): List[A] = xs.reverse
```

Polymorphic Function Types

Scala supports **polymorphic methods** as follows:

```
def reverse[A](xs: List[A]): List[A] = xs.reverse
```

Similarly, a **polymorphic function type** is a function type which accepts type parameters.

```
val reverse: [A] => List[A] => List[A] =
  [A] => (xs: List[A]) => xs.reverse
```

Polymorphic Function Types

Scala supports **polymorphic methods** as follows:

```
def reverse[A](xs: List[A]): List[A] = xs.reverse
```

Similarly, a **polymorphic function type** is a function type which accepts type parameters.

```
val reverse: [A] => List[A] => List[A] =
  [A] => (xs: List[A]) => xs.reverse
```

This type describes function values which takes a type `A` as a parameter and a list of type `List[A]` and returns the same type `List[A]`.

Polymorphic Function Types

Scala supports **polymorphic methods** as follows:

```
def reverse[A](xs: List[A]): List[A] = xs.reverse
```

Similarly, a **polymorphic function type** is a function type which accepts type parameters.

```
val reverse: [A] => List[A] => List[A] =
  [A] => (xs: List[A]) => xs.reverse
```

This type describes function values which takes a type `A` as a parameter and a list of type `List[A]` and returns the same type `List[A]`.

Another example is a `map` method for tuples:

```
(1, "one", true).map((x: Any) => Option(x))        // type mismatch
(1, "one", true).map([T] => (x: T) => Option(x))
// (Some(1), Some("one"), Some(true))
// : (Option[Int], Option[String], Option[Boolean])
```

Polymorphic Function Types vs. Type Lambdas

Polymorphic function types should not be confused with **type lambdas**.

Polymorphic Function Types vs. Type Lambdas

Polymorphic function types should not be confused with **type lambdas**.

A good way of understanding the difference is to notice that

- **polymorphic functions** are applied in **terms**

```
// Polymorphic function type
val id: [A] => A => A = [A] => (x: A) => x
val idInt: Int => Int = id[Int]
```

Polymorphic Function Types vs. Type Lambdas

Polymorphic function types should not be confused with **type lambdas**.

A good way of understanding the difference is to notice that

- **polymorphic functions** are applied in **terms**

```
// Polymorphic function type
val id: [A] => A => A = [A] => (x: A) => x
val idInt: Int => Int = id[Int]
```

- **type lambdas** are applied in **types**, whereas

```
// Type lambda
type Id = [A] =>> A => A
type IdInt = Id[Int]

// Mixing type lambda and polymorphic function
val idInt2: IdInt = id[Int]
```

Contents

Match Types

A **match type** reduces to one of its right-hand sides, depending on the match of the type argument.

```
type Elem[X] = X match
  case String    => Char
  case List[t]   => t
  case Vector[t] => t
```

Match Types

A **match type** reduces to one of its right-hand sides, depending on the match of the type argument.

```
type Elem[X] = X match
  case String    => Char
  case List[t]   => t
  case Vector[t] => t
```

For example, the following code defines a method that returns the first element of data structures:

```
def firstElem[X](xs: X): Elem[X] = xs match
  case x: String    => x.charAt(0)
  case x: List[t]   => x.head
  case x: Vector[t] => x.head
```

Match Types

A **match type** reduces to one of its right-hand sides, depending on the match of the type argument.

```
type Elem[X] = X match
  case String    => Char
  case List[t]   => t
  case Vector[t] => t
```

For example, the following code defines a method that returns the first element of data structures:

```
def firstElem[X](xs: X): Elem[X] = xs match
  case x: String    => x.charAt(0)
  case x: List[t]   => x.head
  case x: Vector[t] => x.head
```

```
val x: Char   = firstElem("Hello")    // 'H'
val y: Int    = firstElem(List(1, 2))  // 1
val z: Double = firstElem(Vector(1.0, 2.0))  // 1.0
```

Match Types

We can define recursive match types as follows:

```
type LeafElem[X] = X match
  case Int => Int
  case String => Char
  case List[t] => LeafElem[t]
  case Vector[t] => LeafElem[t]
```

Match Types

We can define recursive match types as follows:

```
type LeafElem[X] = X match
  case Int => Int
  case String => Char
  case List[t] => LeafElem[t]
  case Vector[t] => LeafElem[t]
```

For example, it returns the first leaf element of data structures:

```
def leafElem[X](xs: X): LeafElem[X] = xs match
  case x: Int => x
  case x: String => x.charAt(0)
  case x: List[t] => leafElem(x.head)
  case x: Vector[t] => leafElem(x.head)
```

Match Types

We can define recursive match types as follows:

```
type LeafElem[X] = X match
  case Int => Int
  case String => Char
  case List[t] => LeafElem[t]
  case Vector[t] => LeafElem[t]
```

For example, it returns the first leaf element of data structures:

```
def leafElem[X](xs: X): LeafElem[X] = xs match
  case x: Int => x
  case x: String => x.charAt(0)
  case x: List[t] => leafElem(x.head)
  case x: Vector[t] => leafElem(x.head)
```

```
val x: Char = leafElem("Hello")                          // 'H'
val y: Int  = leafElem(List(List(1, 2), List(3, 4)))     // 1
val z: Char = leafElem(Vector(List(Vector("Hi", "Bye")))) // 'H'
```

Summary

1. Intersection and Union Types
2. Self Types
3. Opaque Types
4. Structural Types
5. Type Lambdas
6. Polymorphic Function Types
7. Match Types

Next Lecture

PLRG

- Contextual Abstractions

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr