Программирование на уровне типов

Программирование на уровне типов

Этот раздел посвящен продвинутыми техникам работы с типами в scala. Он включает в себя разнообразные приемы от просты type lambdas до работы с библиотекой shapeless. Некоторые приемы мы так или иначе встречали в процессе обучения, но не заостряли на них внимание. Настало время это сделать.
Краткое содержание раздела

- Type lambda
- Типы, зависимые по параметру
- Фантомные типы
- Aux паттерн
- Тэгированные типы
- F-bound полиморфизм
- Partial unification
- Плагин kind projector

Программирование на уровне типов

Type lambda (TL)
Это прием, чем-то схожий с каррированием функций, только для типов принимающих параметры-типы. Т.е. мы можем создать **TL**, например, для типа **Either[A, B]**. Применяют его по тем же причинам, что и каррирование. Например, если мы уже связали часть типов и хотим оперировать новым типом с меньшим количеством параметров. Для того, чтобы сократить число принимаемых параметров-типов, нам нужно, в общем случае, создать новый тип, у которого часть типов-параметров заполнена. Для **Either** это может выглядеть так:

```
object TypeDefinitionExample {
 type λ[α] = Either[α, Any]
  ???
}
```

В примере, мы определили новый тип **λ[α],** который, принимает только один тип-параметр. Тип можно определять внутри объектов, классов, структурных классов, трейтов и т.д. В зависимости от того, где определен тип, доступ к нему будет осуществляться немного по-разному.

Программирование на уровне типов

```
// в случае если тип определен в объекте
val e: TypeDefinitionExample.λ[Int]
// или если внутри класса или трейта
trait TypeDefinitionExample {
 type λ[α] = Either[α, Any]
}
val e: TypeDefinitionExample#λ[Int]
```

Часто **TL** определяют прямо по месту подстановки, для этого используют прием с анонимным структурным типом. Запись в этом случае выглядит немного запутанно. В таких случаях полезным оказывается плагин к компилятору по названием kind-projector, о котором мы поговорим чуть позже.

Пример: **lectures.types.lambda.ToOptionProjector.scala**
В примере выражение **({type λ[α] = Either[α, _]})#λ** определяет анонимный структурный тип (все, что находится в фигурных скобках). В структурном классе есть единственный член, - это тип **λ[A] == Either[A, Any].** Обратите внимание, что пришлось ввести дополнительный тип **type l[_] = F[_],** т. к. если бы мы в методе написали вот так: **projectEither[A](va: ({type λ[α] = Either[α, _]})#λ), λ** из метода это был бы совершенно другой тип, неравный **λ** из сигнатуры класса.

Программирование на уровне типов

Задание: lectures.types.lambda.SeqToStringK.scala

Программирование на уровне типов

Зависимые типы. Типы зависимые по параметру.
Зависимые типы, в общем случае - это типы которые зависят каким-то образом от контекста, в котором они определены. В скале существует две разновидности зависимых типов. Типы, зависимые от пути и типы, зависимые от параметра.

Типы, зависимые от пути
С этой разновидностью зависимых типов мы уже встречались. **Path dependency** - это свойство типов вложенных в классы или трейты. Свойство это заключается в том, что инстансы вложенных типов равны по типу только тогда, когда созданы внутри одного и того же инстанса внешнего типа. Рассмотрим пример:

Объявим класс **Cnt**. Объявим, внутри класса **Cnt** внутренний тип **Inner**. Тогда все использования **Inner**, без спецификации полного “пути” типа, будут **path dependent**. Это значит, что для каждого объекта внешнего класса будет определен свой внутренний тип, неравный типу в других объектах. Полный путь типа можно описать с помощью символа **#.** Например полный путь типа Inner будет Cnt#Inner. Путь может быть описан для любой глубины вложенности типа.

Программирование на уровне типов

Path dependent types

```
import scala.reflect.runtime.universe._

object Cnt {
  def tagged[T: TypeTag](t: T) = typeTag[T]
}

class Cnt {
  class Inner
  def getInner = new Inner
  def doPathIndependent(t: Cnt#Inner) = println("Path dependent")
  def doPathDependent(t: Inner) = println("PathDependent")
}
val cnt = new Cnt
val cnt2 = new Cnt
val in = cnt.getInner
val in2 = cnt2.getInner
val inT = Cnt.tagged(in)
val in2T = Cnt.tagged(in2)
cnt.doPathIndependent(in)
cnt.doPathIndependent(in2)
cnt.doPathDependent(in)
//cnt.doPathDependent(in2) won't compile due to  path dependency
assert(!(inT.tpe =:= in2T.tpe))
```

Программирование на уровне типов

Типы методов зависимые по параметру.
В скале есть возможность определить метод, чей возвращаемый тип, будет зависеть от типа входного параметра. Выглядит это обычно примерно так:

```
// Тип с подтипом V
trait DepValue{
 type V
 val value: V
}
//Метод, возвращающий зависимый тип
def magic(that: DepValue): that.V = that.value

// Вспомогательный метод, создающий инстансы разных типов
def mk[T](x: T) = new DepValue{
 type V = T
 val value = x
}
// Пример вызова
val depInt = mk(1)
val depString = mk("two")
val itWorks: Int = magic(depInt)
val again: String = magic(depString)
```

Программирование на уровне типов

Сейчас в скале невозможно описать функции с зависимыми типами, только методы. Однако в Dotty ожидается такая возможность.
Типы, зависимые по параметру похожи на тайп параметры методов, однако они оказываются лучшим выбором, когда

- методу, по какой-то причине нельзя иметь тайп параметр
- зависимый тип является результатом сложного вычисления, которое для каждого конкретного типа входного параметра может быть своё.

Давайте рассмотрим еще один пример, в котором зависимый тип становиться немного сложнее
lectures.types.SeqTransformer.scala

задание: lectures.types.lambda.SeqTpStringK

Программирование на уровне типов

AUX pattern

Теперь, когда мы знакомы с типами, зависимыми по параметру, у нас может возникнуть вопрос, может ли что-нибудь еще, кроме возвращаемого типа зависеть от типа входного параметра. Например, другой параметр. Вспомним наш пример с **SeqTransformer.** В нем мы превращали Seq в произвольный тип **G[_]** с помощью трансформеров. Тип **G,** в этом преобразовании, полностью определялся доступным в данном контексте трансформером. Представим теперь, что после нам понадобилось применить некую функцию трансформации к содержимому **G.** Появится соблазн добавить еще один параметр в функцию transform, чей тип, тоже будет зависеть от переданного трансформера, вот так:

```
trait ValueTransformer[T, G[_]] {
 def transformValue(v: T): T
}

def transform[T](vals: T*)(implicit tr: SeqTransformer, valueTransformer: ValueTransformer[T, tr.G]): tr.G[T]
```

Если мы так сделаем то получим ошибку компиляции: **illegal dependent method type: parameter may only be referenced in a subsequent parameter section**

Программирование на уровне типов

Дело в том, что мы скала компилятор не позволяет описывать зависимость на тип, описанный в том же выражении. Одним из решений этой проблемы является передача типа G, снаружи и в **SeqTransformer** и в **ValueTransformer**. Таким образом типы параметров не зависят друг от друга, при этом тип выходного параметра может остаться зависимым от типа параметра **SeqTransformer** Для этого определим тип **SeqTransformerWithAux.Aux** следующим образом:

```
object SeqTransformerWithAux {

 type Aux[G1[_]] = SeqTransformerWithAux{ type G[_] = G1[_]}

 def transform[T, G1[_]](vals: T*)(implicit tr: SeqTransformerWithAux.Aux[G1], valueTransformer:
Option[ValueTransformer[T, G1]] = None): tr.G1[T] =
   tr.transform[T](vals: _*)
}
```

Все что мы сделали в этом примере, это превратили внутренний тип **G** в тайп параметр **G1.** Теперь его можно передавать, как параметр.

полный код примера и задание в lectures.types.AuxPattern.scala

Программирование на уровне типов

Фантомные типы
Это типы, используемые только на этапе компиляции. Обычно они нужны, чтобы наделить какой-либо класс дополнительными свойствами или признаками, которые не нужны в рантайме. Обычно фантомными типами наделяют инстансы классов доменной модели.

Есть 2 основные причины применять фантомные типы
- необходимость в строгой типизации в какой-либо части логики
- и стремление избежать накладных расходов на дополнительные инстансы

Пример: **lectures.types.PhantomTypes.scala**

Несмотря на то, что фантомные типы часто приводят как пример мощи системы типов в скале, они же показывают ее ограниченность. Например мы легко можем проверить что кейс класс обладает определенным тайп параметром, но очень трудно без применения рефлексии проверить что этот же кейс класс НЕ обладает определенным тайп параметром. Возьмем функцию из примера **def overdueCheck(check: PayCheck[Sent]).** Она доступна для всех оплаченных чеков. Возникает вопрос: если мы хотим сделать такую проверку для всех НЕ просроченных чеков, как нам описать такой метод?

Программирование на уровне типов

Тоже самое касается дизъюнкции или конъюнкции типов. Сложно описать функцию принимающую тип T или G. Отчасти эта проблема может быть решена применением имплиситов.

Еще одна проблемы с фантомными типами в том, что их применение может потребовать модификации доменной модели. Кроме того, описав с помощью фантомных типов одну категорию свойств, может быть невозможно применить к тем же доменным объектам другие фантомные типы. Последние 2 задачи можно решить, применив так называемые тайп теги.

Type tags

То, что мы сделали с помощью фантомных типов, не всегда возможно. В предыдущем примере мы ввели класс **PayCheck[T]** и методы работающие с этим классом. Но что, если мы не хотим создавать новые инстансы или вообще вводить новые типы, но тем не менее, хотим более строгой типизации? Возможно ли это?

В предыдущем примере мы, как бы, пометили(тегировали) один тип, другим типом. Оказывается такой трюк можно проделать с любым типом, даже не принимающим тайп параметры. Ниже пример механизма тегирования из библиотеки **shapless**

Программирование на уровне типов

```
object tag {
 trait Tagged[U]

 def apply[U] = new Tagger[U]

 type @@[+T, U] = T with Tagged[U]

 class Tagger[U] {
   def apply[T](t : T) : T @@ U = t.asInstanceOf[T @@ U]
 }
}
```

Все гениальное просто. Как в фантомных типах, вводится тип **Tagged[U]**, который принимает тайп параметр. Чтобы в рантайме мы никогда не увидели ошибок, этот тип не имеет ни полей ни методов. Теперь осталось сделать метод, который метит любой тип **T** тэгом **Tagged[U]**, где **U**, любой подходящий тип. В нашем примере с фантомными типами, это был бы один из наследников **PayCheckStatus**

Программирование на уровне типов

Очень часто важная информация представляет собой просто строку. Когда видим тип String, мы не всегда можем предположить, какие же данные содержит объект этого типа. С помощью тэгов можно реализовать полезный паттерн **Id[_]**, позволяющий накладывать дополнительные ограничения на уровне типов, на строки. Этими ограничениями мы даем понять компилятору и разработчикам, какие именно данные должна содержать строка. Реализация **Id** с помощью **@@**

```
object Id {
 def apply[U](str: String): Id[U] = str.asInstanceOf[Id[U]]

 type Id[U] = String @@ U
}
```

Пример применения Id: lectures.types.IdExample.scala

Задание: lectures.types.LoggedTypeCheck.scala

Программирование на уровне типов

F-bound polymorphism

Представим, что мы создали трейт с методом, принимающим на вход параметр того же типа. Примером может служить трейт, описывающий метод-компаратор:

```
trait Container[T] {
 val value: T

 def compare(that: Container[T]): Int
}
```

Определим 2 наследника, этого трейта контейнер для целых и контейнер для целых по модулю

```
class AbsIntContainer(override val value: Int) extends Container[Int]{
val absValue = Math.abs(value).toInt
override def compare(that: Container[Int]): Int = ???
}
class IntContainer(override val value: Int) extends Container[Int]{
 override def compare(that: Container[Int]): Int = ???
}
```

Программирование на уровне типов

Понятно, что метод сравнения, определенный так, как это сделано в наследниках трейта, может привести к ошибкам. Что бы нам хотелось - это разрешить сравнение только инстансов одинаковых типов, наследников container. В этом нам может помочь прием, называемый **F-bound polymorphism** или рекурсивный полиморфизм. Суть приема заключается в том, что класс параметризуется типом особого вида.

- тип-параметр(обычно) должен быть наследником, типа, параметром которого является
- должен быть параметризован самим собой

Модифицируем наш пример с контейнерами, и посмотрим как это работает.

Полный код примера можно найтив **lectures.types.FBoundContainer.scala**

Программирование на уровне типов

```
class AbsIntContainer(override val value: Float) extends Container[Float, AbsIntContainer]{
 val absValue = Math.abs(value).toInt
 override def compare(that: AbsIntContainer): Int = absValue - that.absValue
}

class IntContainer(override val value: Int) extends Container[Int, IntContainer]{
 override def compare(that: IntContainer): Int = ???
}
val ai = new AbsIntContainer(1000.01f)
val ai2 = new AbsIntContainer(1000.02f)
val ii = new IntContainer(1000)

//ai.compare(ii)
// ii.compare(ai2)
ai.compare(ai2) == 0

class Int2Container(override val value: Int) extends Container[Int, IntContainer]{
 override def compare(that: IntContainer): Int = ???
 // Won't compile
 //override def compare(that: Int2Container): Int = ???
}
```

Программирование на уровне типов

Задание: lectures.types.FBoundSemigroup.scala

Программирование на уровне типов

Частичное обобщение (Partial Unification)

Часто мы сталкиваемся с проблемой несовпадения арности (размерности) тайпконструкторов. Допустим, что какой -то метод или конструктор типизирован, тайп параметром с одной дыркой **F[_]**. Возьмем для примера следующий метод:

```
def identityParameter[F[_]](x: F[_]): F[_] = x
```

Этот метод примет любой тип, принимающий один тайп параметр. List, Option, Functor и т.д. Но что будет, если мы передадим в этот метод **Either[_, _]** ? Произойдет следующее:

```
identityParameter(Left(10))

No type parameters for method identityParameter: (x: F[_])F[_] exist so that it can be applied to arguments
(scala.util.Left[Int,Nothing])
--- because ---
argument expression's type is not compatible with formal parameter type;
found   : scala.util.Left[Int,Nothing]
required: ?F[_$1] forSome { type _$1 }
```

Программирование на уровне типов

Частичное обобщение (Partial Unification)

Ошибка с предыдущего слайда, кажется логичной, ведь формы типов действительно не совпадают. Но, можно взглянуть на тайп конструктор, как на обычную функцию, т. е., например: тайпконструктор формы **F[_],** был бы функцией следующего вида: **F [A] == (A) => F[A], а F[A, B] == (A, B) => F[A, B]**. Т.е. **F[_] -** это функция над типами, принимающая на вход тип **A** и возвращающая тип **F[A], F[A, B]** - это функция от 2-х входных параметров - типа **A** и типа **B** и возвращающая тип **F[A,B].** Про обычные функции мы знаем, что их можно каррировать, т.е. превращать в функции от одного переменного. **Partial Unification** - это такое же каррирование, но над тиами. Инстанс типа **F[A,B]** в случае, когда нужен будет тип **G[_]** будет представлен как **G[B => F[A, B]].** Где **B => F[A, B] == F[_, B],** тип ожидающий в точности один тайп параметр, что и требовалось.
Примеры из статьи Даниела Спивака:

```
// требуется
F[_]
// передан инстанс
Foo[Int, String, Boolean]
// выведен тип
[A]Foo[Int, String, A]
```

```
// требуется
F[_, _, _]
// передан инстанс
Foo[Monad, Int, String, Boolean, Unit]
// выведен тип
[A, B, C]Foo[Monad, Int, A, B, C]
```

Программирование на уровне типов

Частичное обобщение (Partial Unification)

Еще пара примеров в **lectures.types.UnificationExample.scala**

Программирование на уровне типов

Подчеркивание в описании типов.

Подчеркивание активно применяется в описании и определении типов. Смысл этого символа сильно зависит от контекста, в котором он применяется. Это приводит к тому что, зачастую, не очевидно, что он значит и к каким результатам приводит его применение. В этом разделе мы рассмотрим применение подчеркивания в разрезе работы с типами.
Подчеркивание в описании классов(трейтов, объектов и т.д.). Тип **G[_]** - говорит, что класс **UnderscoreTypeBehaviour** принимает в качестве параметра тайпконструктор. Обратите внимание ниже параметр **List** - это не тип, это тайпконструктор.

```
//подчеркивание в наследовании
class UnderscoreTypeBehaviour[G[_]] {
 //так нельзя, хотя type параметры передаются именно так
 //так как справа может быть только тип, но не род(kind)
 //type T[_] = List
}
class UnderscoreTypeBehaviourSubClass extends UnderscoreTypeBehaviour[List] {}
//val ub2: UnderscoreTypeBehaviour[List[_]] = new UnderscoreTypeBehaviour[List[Int]]
val ub1: UnderscoreTypeBehaviour[List] = new UnderscoreTypeBehaviour[List]
```

Программирование на уровне типов

Подчеркивание в описании типов.

Подчеркивание, как экзистенциальный тип. В остальных случаях, когда мы встречаем символ _ он является сокращенной записью экзистенциального типа. Экзистенциальный тип - это тайп параметр без имени. Он обычно применяется, что бы показать, что в данной конкретной части кода нам подходит любое значение тайп параметра и кроме того, это значение нам безразлично.

```
// Подчеркивание как экзистенциальный тип
//a будет иметь тип Any
val a = 1: T forSome {type T}
// Такая запись невозможна т.к. _ - это тайп параметр, а не тип
//val a2 :_ = 1
//  val a2  = 1:_
 val s2 = Set[_](1,2,3,4,5)
// а так можно
val s1 = Set[T forSome {type T}](1,2,3,4,5)

// Подчеркивание связывается с самым внутренним типом. Типы s2 и s3 идентичны
val ss2 = Set[Set[_]]()
val ss1 = Set[Set[T] forSome {type T}](Set(1,2,3,4,5))
```

Программирование на уровне типов

Подчеркивание в описании типов.

```
// подчеркивание в определении типов с помощью ключевого слова type
 type T2[_] = List[_]
 type T3[Pr] = List[Pr]
 def transformExistential[P](vals: P*): T2[P] = vals.toList
 def transformUniversal[P](vals: P*): T3[P] = vals.toList//
//  val resExist: List[Int] = transformExistential(1,2,3)// WON'T COMPILE т.к. List[_] != List[Int]
 val resExist2 = transformExistential(1,2,3)
 val resUniversal: List[Int] = transformUniversal(1,2,3)
```

```
// подчеркивание в определении методов
def m[F[_]](t: F[_]): F[_] = t
def m2[F <: List[_]](f: F): F = ??? // F - это тип, не принимающий параметров
def m3(t: List[_]): List[_] = List(1,2,3)
// def m4[F[_]: List[_]](t: F[_]) = ???
def m5[F[_] <: List[_]](): F[_] = List(1,2,3)
def m52[F[_] <: List[_]](t: F[_]): F[_] = t///List(1,2,3)
val m52r = m52(List(1,2,3))
```

Программирование на уровне типов

Подчеркивание в описании типов.

Больше примеров в **lectures.types.UnderscoreTypeBehaviour.scala**

Ограничения системы типов scala
Partial unification и **_,** являясь мощными средствами, все же не позволяют преодолеть некоторые ограничения, в работе с типами. Некоторые из этих ограничений приведены в **lectures.types.ScalaTypeConstraints.scala**

Приведенные выше проблемы могли бы быть решены, если бы кроме самих тайп параметров мы могли передавать выражения на ними. Именно это позволяет сделать плагин к компилятору **(Non)KindProjector**

Программирование на уровне типов

Справочник по плагину KindProjector

KindProjector - это плагин к компилятору, который позволяет короче и понятнее описывать выражения над типами.

К sbt проекту его можно подключить, написав в вашем **build.sbt** следующее: **addCompilerPlugin("org.spire-math" %% "kind-projector" % "0.9.4").** Версия, скорее всего, будет отличаться от указанной в примере. Текущую версию и документацию по kindProjector можно найти в репозитории на github **https://github.com/non/kind-projector**

По сути плагин позволяет определять лямбды(неименованные выражения) над тайп параметрами. Выражения, написанные с применением плагина имеют 2 основные формы: сокращенную и полную.

Программирование на уровне типов

KindProjector, сокращенная запись
Сокращенная запись, позволяет описывать лямбды над тайп параметрами. Лямбды могут принимать несколько не именованных параметров.

```
Tuple2[?, Double]       // equivalent to: type R[A] = Tuple2[A, Double]
Either[Int, +?]         // equivalent to: type R[+A] = Either[Int, A]
Function2[-?, Long, +?] // equivalent to: type R[-A, +B] = Function2[A, Long, B]
EitherT[?[_], Int, ?]   // equivalent to: type R[F[_], B] = EitherT[F, Int, B]
```

Чтобы лучше понять, что происходит, можно провести аналогию с обычными функциями. В описании типов выше знак **?** выполняет ту же функцию,что и символ **_** в функции ниже.

```
val f: Int => Int = _ + 1
```

Программирование на уровне типов

KindProjector, сокращенная запись
Вспомним ограничение, с которыми мы столкнулись ранее(**ScalaTypeConstraints.scala**), когда определяли наследников и инстансы класса **UnderscoreTypeBehaviour[G[_]]**. С помощью KindProjector мы можем привести типы большей арности к нужной размерности.

```
// теперь мы можем привести типы большей арности к нужной размерности
abstract class UnderscoreTypeBehaviourSubClass2[A] extends UnderscoreTypeBehaviour[Either[?, A]]

new UnderscoreTypeBehaviour[Either[?, Int]] {
 override val value: Either[Int, Int] = Left[Int, Int](10)
}

new UnderscoreTypeBehaviour[Either[Int, ?]] {
 override val value: Either[Int, Long] = Left[Int, Long](10)
}
```

Программирование на уровне типов

KindProjector, сокращенная запись. Ограничения
Может показаться, что сокращенная запись является более мощным вариантом экзистенциального типа, но это не так. Есть несколько случаев, когда _ работает, а ? неприменим

```
// Either[Int, ?] - это не тип, это Kind
// Either[T, ?] == [A]Either[String, A], т.е один парамер не связанн
val l: Either[Int, ?] = Left(1)
val r: Either[?, Int] = Right(1)
type E2[T] = Either[T, _]
type ENop[T] = Either[T, ?]
```

? всегда связывает самый внутренний тип. Это значит, что мы не сможем заменить **type R[T] = List[List[T]]** на **List[List[?]].** Последний тип можно представить с помощью псевдокода следующим образом **List[A => List[A]].** Чтобы описать тип **R[T]** с помощью **KindProjector** используют полную запись

Программирование на уровне типов

KindProjector, полная запись

KindProjector позволяет описать лямбда выражения над типами с помощью ключевого слова **lambda** или символа **λ.** Оба варианта записи эквивалентны.

```
//несколько простых примеров
Lambda[A => (A, A)]            // equivalent to: type R[A] = (A, A)
Lambda[(A, B) => Either[B, A]]  // equivalent to: type R[A, B] = Either[B, A]
Lambda[A => Either[A, List[A]]] // equivalent to: type R[A] = Either[A, List[A]]

//примеры с вариативностью
λ[`-A` => Function1[A, Double]]         // equivalent to: type R[-A] = Function1[A, Double]
λ[(-[A], +[B]) => Function2[A, Int, B]] // equivalent to: type R[-A, +B] = Function2[A, Int, B]
λ[`+A` => Either[List[A], List[A]]]     // equivalent to: type R[+A] = Either[List[A], List[A]]

//примеры с родАми
Lambda[A[_] => List[A[Int]]] // equivalent to: type R[A[_]] = List[A[Int]]
Lambda[(A, B[_]) => B[A]]    // equivalent to: type R[A, B[_]] = B[A]
```

Программирование на уровне типов

KindProjector, полная запись

Имея в своем распоряжении лямбды над типами, мы можем решить проблему метода **compose** из **ScalaTypeConstraints.scala**

```
def compose[G[_],F[_]](): UnderscoreTypeBehaviour[λ[A => G[F[A]]]] = ???
```

Примеры использования KindProjector: KindProjectorExample.scala