Обобщённые типы

В предыдущей статье раскрывались некоторые основные понятия теории типов. В этот раз мы рассмотрим необходимость появления такой абстракции, как обобщённые типы (*generics*), рассмотрим их ключевые особенности и различные сценарии использования в программировании.

Оглавление

В статье основная тема раскрывается с различных точек зрения:

Предисловие

Однажды я делал доклад для наших сотрудников о технических решениях на одном из наших Scala-проектов. Там рассказывалось про применение монадических трансформеров, Tagless Final и прочих ФП-шных вкусностей. После доклада по обратной связи я получил разные замечания, в том числе и вопрос, который звучал примерно так: "а зачем вообще нужны обобщённые типы?". В тот момент вопрос показался неожиданным, я пообещал, что в следующий раз обязательно раскрою эту тему.

Тем не менее, быстро не получалось подготовить ответ на поставленный вопрос. Собственных знаний не хватало, чтобы объяснить эту тему самому себе. То, что подсказывал интернет, даже если бы и удовлетворило любопытство спросившего, но не убеждало меня. Обычно там говорятся общее слова, мол, обобщённые типы дают "типобезопасность", "оптимизацию" (не требуется явное приведение к типам-значений и т.п.), "переиспользумеость" и прочая бессодержательная "вода". Зачастую, далее приводится пример, в котором раскрываются преимущества типизированных (обобщённых) *коллекций* над нетипизированными...

Однако, во многих языках программирования со строгой типизацией уже есть встроенные типизированные коллекции со своими методами, например, массивы.

Выглядит так, что будто бы и нет необходимости давать программистам возможность создавать свои обобщённые типы?

На самом деле, полезными бывают, конечно же, не только обобщённые списки и методы для работы с ними. Здесь мы рассмотрим, **какими задачами обусловлена необходимость появления концепции** обобщённых типов, рассмотрим ключевые особенности этих типов и различные сценарии их использования в программировании.

Так же как и предыдущая статья, эта не является подробным учебником с систематическим изложением материала. Основная задача - поверхностный обзор темы с самых разных точек зрения, с тем, чтобы заинтересовавшийся каким-либо аспектом читатель смог самостоятельно найти подробности, например, в интернете.

Далее предполагается, что читатель уже знаком с базовыми понятиями теории типов, освещёнными, например, в этой статье.

Обобщённые типы

Мотивация

Представим, что у нас есть функция, которая может завершится ошибкой, которую в дальнейшем можно будет обработать. Это может быть поиск элемента в коллекции (искомого значения может не найтись), или запрос на web-сервер (тут может быть много разных ошибок). В привычном императивном программировании это обычно решается с применением механизмов выброса исключений и их отловом. Такой подход, как правило, недекларативен - по сигнатуре функции нет возможности понять, какие "несчастливые" исходы возможны. В любом случае, такие механизмы усложняют язык, вводя новые синтаксические конструкции, вроде `try/catch`. В то же время, задача решается благодаря естественному понятию теории типов - сумме типов:

$$findItem : RegEx => ItemType + NotFound$$

$$getSmthFromWeb : Request => Response + Unaccessable + BadRequest + InternalError$$

Благодаря очевидному алгебраическому изоморфизму, описанного в предыдущей статье, для каждой такой суммы типов однозначно определяется функция, которая из значений этого типа вычисляет значения произвольного типа X - механизм сопоставления с шаблонами проверяет, чтобы программист предоставил функции в тип X из всех типов-слагаемых:

```
def foldToString(itemOrNot: ItemType | NotFound)( // ItemType + NotFound
  itemToString: ItemType => String,  // на вход приходят две функции
  notFoundResult:   Unit => String   // по количеству слагаемых типа
```

```
): String =
  itemOrNot match     // тело функции не зависит от конкретного ItemType!
    case it: ItemType => itemToString(it)
    case _: NotFound  => notFoundResult()
```

Здесь пересечение типов `ItemType | NotFound` можно считать суммой типа `ItemType` и единичного типа `NotFound`.

И поиск элемента в коллекции, и загрузка контента из веб-сервиса являются весьма распространёнными сценариями, которые неоднократно встречаются во многих приложениях. При этом, конкретные типы `ItemType`, или `Response` будут различаться в разных местах, но *способы использования* результирующих значений типов суммы - реализации методов `fold` - не будут зависеть ни от `ItemType`, или `Response`! Более того, различные реализации метода `fold` будут отличаться только заменой типа `ItemType` (или, аналогично, `Response`) на какой-нибудь другой. Можно сказать, что этот тип является *параметром* метода! Многократное дублирование этого кода для каждого такого конкретного "типа-параметра" выглядело бы весьма неудачным решением...

Помимо сценариев с суммами типов, рассмотрим и другие. Например, если какой-либо метод ищет некий контент на разных ресурсах, было бы полезно вместе с результатом вернуть и сведения об источнике, откуда этот контент был получен. Тут уже поможет произведение типов вида `Content × Source`, где тип `Content` может отличаться в разных конкретных случаях, а `Source` - это фиксированный тип, например, `String`. В таких случаях способы использования такого кортежа на основе проекторов в типы `Content` и `Source` не будут зависеть от того, какой именно тип-параметр `Content` будет там использоваться.

```
def foldToInt(contentWithSource: Conent & Source)( // Conent × Source
  handler: Conent => Source => Int  // на вход приходит каррированная функция
): Int =
  // обраотчик handler вполне может игнориовать свой второй параметр source
  handler(contentWithSource: Conent)(contentWithSource: Source)
```

Ещё один популярный сценарий - "избавление от зависимостей". Если в неком методе требуется использование какого-то сервиса типа `Service`, то обычно его называют зависимостью и передают в самом начале, например, вместе "бизнесовыми" параметрами метода (или же по-ООПшному, чуть раньше - в конструктор класса, где расположен метод). Получается, так называемый, "жадный" захват зависимости ("утром деньги, вечером стулья"). Сигнатура получается примерно такая: `(Service, BusinessParam) => Result`. "Жадному" механизму противопоставляется "ленивый", когда метод не требует предоставление

зависимости в самом начале, но возвращает функцию, которая на вход требует только значение `Service`: `BusinessParam => (Service => Result)`. Получается, что метод "избавился от зависимости"! Таким образом всю бизнес-логику можно описать чисто, **вынеся всю возню с зависимостями в самый конец приложения** (для обработки библиотечными методами или средой исполнения). В такого рода сценариях наблюдается знакомая закономерность - способ использования значений экспоненциального типа `Service => Result` не зависит от типа `Result` - он является параметром метода.

```
def foldToResult(resultByService: Service => Result)( // Result^service
  service: Service          // на вход приходит сервис
): Result =
  resultByService(service) // что такое Result - не важно!
```

Можно привести и другие аналогичные сценарии, в которых конкретизация некоторых типов (вроде `Result`, `Content`, `Response`, `ItemType`) не нужна - в разных вариантах это могут быть самые разные типы, но сам алгоритм сценария остаётся тем же самым! Было бы весьма расточительно многократно копипастить такие функции, отличающиеся только своей сигнатурой, но имеющие одинаковую реализацию.

Для решения такой проблемы необходима новая абстракция - *обобщённые методы*.

Обобщённые методы

Идея простая - достаточно описать единственную обобщённую функцию, которая будет описывать все варианты сценария с различными конкретными типами, передаваемыми ей как параметры. В Scala параметры-типы передаются не в круглых скобках, как обычные аргументы, а в квадратных. Перепишем методы `fold`, объявленные в предыдущем примере, в обобщённой форме:

```
def foldToString[ItemType](itemOrNot: ItemType | NotFound)( // ItemType +
NotFound
  itemToString: ItemType => String,  // на вход приходят две функции
  notFoundResult:   Unit => String   // по количеству слагаемых типа
): String = ???                      // реализация такая же, как и раньше

def foldToInt[Conent](contentWithSource: Conent & Source)( // Conent × Source
  handler: Conent => Source => Int  // на вход приходит каррированная функция
): Int = ???                        // реализация такая же, как и раньше

def foldToResult[Result](resultByService: Service => Result)( // Result^service
```

```
  service: Service                    // на вход приходит сервис
): Result = ???                       // реализация такая же, как и раньше
```

Использовать их можно так:

```
val doubleOrNot: Double | NotFound = ??? // какое-то значение
val str = foldToString[Double](doubleOrNot)(d => s"double: $d", _ => "no double
found")
//                     ^^^^^^ - тип-параметр указан явно (но это не обязательно)

val intOrNot: Int | NotFound = ??? // какое-то значение
val str = foldToString(intOrNot)(i => s"int: $i", _ => "no int found")
//                   ^^ - зачастую, тип-параметр компилятор и сам может вывести
```

Тут обобщённый метод `foldToString`, определённый единообразно **для любого** конкретного типа `ItemType`, используется со значениями разных типов: `Double` и `Int`. Причём во втором случае тип не указан явно - компилятор сам определяет его по типу первого параметра метода.

Телом обобщённых методов являются выражения, в которых некоторые используемые там термы могут быть любого типа. Такие типы участвуют в выражении *свободно* - они могут меняться в зависимости от контекста использования этого выражения. Метод же образуется путём *связывания* всех свободных параметров выражения с аргументами, которые нужно передать в метод (замыкания сейчас не рассматриваем). В случае обобщённых методов некоторыми из таких аргументов будут типы.

Приведённые выше обобщённые методы являются примерами гомоморфизма типов ("одной половинкой [изоморфизма](ссылка на раздел из предыдущей статьи)"). В таком случае их реализация определяется типами аргументов и результата метода. Но, конечно же, можно написать и свои обобщённые методы, которые будут связывать бизнес-логикой типы модели конкретной предметной области.

Внимательный читатель обратит внимание на типы аргументов представленных выше обобщённых методов - в типах `ItemType | NotFound`, `Conent & Source`, `Service => Result` упоминаются неопределённые типы-параметры `ItemType`, `Conent` и `Result`. То есть, параметризация методов типами отразилась прежде всего на типах их аргументов и результата. С другой стороны, тип функции, соответствующий сигнатуре этих методов, так же оказывается параметризирован типом!

Так как Scala позиционируется как мульти-парадигменный язык, то в нём есть традиционные ОПП-шные классы. И также, как и во многих популярных языках, параметризированные типы можно получить, определяя *обобщённые классы*.

Обобщённые классы

Типы-параметры обобщённых классов в Scala указываются в квадратных скобках сразу после имени класса.:

```
case class ContentAndSource[Content](content: Content, source: String)

trait ContentHandler[Content, Result] {
  def handleContent(contentAndSource: ContentAndSource[Content]): Result
}
```

Чтобы создать экземпляр обобщённого класса, помимо значений-параметров конструктора, нужно конкретизировать и типы-параметры:

```
case class MyContent(test: String) // будем подставлять его как тип-параметр

val contentHandler = new ContentHandler[MyContent, String] {
  def handleContent(contentAndSource: ContentAndSource[Content]) =
          s"Из источника '${contentAndSource.source}' получено содержимое:
${contentAndSource.content.text}"
          //
^^^^^
}     //                                          используем проектор конкретного
класса MyContent.text

val myContentAndSource = ContentAndSource(MyContent("text"), "hardcoded")
//                                        ^^
//        тут необязательно явно указывать тип - он будет конкретизирован на
основе типа аргумента

val str = contentHnadler.handleContent(myContentAndSource) // String
// "Из источника 'hardcoded' получено содержимое: text"
```

Практически о всех ФП-шных Scala-библиотеках обобщённые типы представлены именно как обобщённые классы (трейты). Это обусловлено, в частности, полезностью разграничения контекстов - определённые через классы, *типы считаются разными*, даже если реализованы они одинаково. Это значит, что применимые к ним наборы функций не будут пересекаться. В то же время, с рассматриваемыми в следующей главе псевдонимами λ-выражений типов ситуация обратная - типы сравниваются по реализации, а не по названию, значит, могут быть (потенциально нежелательные) коллизии между наборами функций, предоставляемыми для этих типов разными библиотеками. Избежать таких

нежелательных коллизий возможно и ФП-шными средствами Scala, но, зачастую, такие решения считаются нецелесообразными.

Далее же мы рассмотрим более ФП-ориентированные способы определения обобщённых типов.

Конструкторы типов

Термин "конструкторы типов" буквально отражает идею возможности построения новых типов из существующих. Единожды описав такой конструктор (или используя уже встроенные в язык) можно создавать новые типы, подставляя разные известные типы в этот конструктор. Термин отражает функциональную природу обобщённых типов - это, по сути, отображения из типов в тип (а не из значений в значения, как обычные функции).

Любая функция строится вокруг некоторого выражения, которое становится её телом. Поэтому в первую очередь рассмотрим, какие выражения над типами можно использовать в языке Scala.

Встроенные операции над типами в Scala

В Scala 3 есть встроенные конструкции для построения новых типов из существующих:

- просто тип `A` - случай настолько тривиальный, что легко упустить его полезность,
- кортеж `(A, B, C)` - суть тип-произведение,
- объединение типов `A | B` - если типы разные, то это аналогично их сумме,
- пересечение `A & B` - если типы разные, то это аналогично их произведению,
- "вызов" другого конструктора типов, например,
  - `A => B` - тут используется псевдоним для `Function[A, B]`,
  - `A Either B` - синоним для `Either[A, B]`,
  - `Option[A]`,
  - `Map[K, V]`,
  - `MySupperClass[A, B, C]` - конечно, можно вызывать и конструкторы типов, связанных с классами (или трейтами), не определёнными в стандартной библиотеке Scala.
- сопоставление с шаблонами для типов.

> Механизм сопоставления с образцом для типов сочетает в себе несколько разных аспектов: специальный полиморфизм, деконструкция выражений, зависимые типы. Только первый пункт имеет отношение к обобщённым типам – он будет рассмотрен далее. Два оставшихся аспекта выходят далеко за рамки темы этой

> статьи, поэтому и механизм сопоставления с образцом здесь не будет рассматриваться.

Комбинируя эти конструкции можно определять типы произвольной сложности:

```
type NoAction = Unit   // Unit - встроенный единичный тип
type Log      = String // также вполне корректное выражение
type NewType  = NoAction | ((Service1 & Service2) => MyIO[(SingleResult Either
List[AltResult], Log)])
//
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^
//                       Вот такое кучерявое выражение получилось. Конечно,
можно и длиннее))
```

Функции над типами

Идентификаторы, которые используются в выражении, могут относится как к известным в текущем контексте конструкциями, так быть неопределёнными, *свободными* переменными. Чтобы иметь возможность вычислить выражение, можно построить функцию, *связать все свободные переменные* выражения с параметрами, которые требуется передать при вызове этой функции.

С этой точки зрения удобно рассматривать и **обобщённые типы - это функции, отображающие из одного или нескольких типов в тип**. Аналогия обобщённых типов Scala с обычными функциями нагляднее прослеживается в объявлениях *псевдонимов типов* с помощью ключевого слова `type`:

```
type Union[A, B]           = A | B                    // объединение типов
type OrErrorAndContext[x] = Union[(X, String), Error] // "вызывается" функция
(конструктор типов) Union
```

Действительно, похоже на определения обычных функций, только "тип" аргументов и результата нигде не указывается. У них у всех одинаковый "тип" - это тип!)))

В Scala 3 также есть возможность использовать литералы обобщённых типов – λ-выражения типов:

```
type WithContext = [A] =>> (A, String)                // псевдоним типа,
опеределённый через λ-выражение
```

Опять же, синтаксис похож на λ-выражения обычных функций.

Для обобщённых типов, принимающий ровно два аргумента возможна инфиксная запись, которая иногда бывает полезна, а иногда – не очень:

```
// если такой код компилируется, значит использованы эквивалентные типы по обе стороны =:=
summon[(Either[String, Int]) =:= (String Either Int)]     // близко к естественному языку "строка или целое число"
//      ^^^^^^                              ^^^^^^
//   обычная запись                    инфиксная запись
summon[(Function[String, Int]) =:= (String Function Int)] // а с таким типом инфиксная запись читается хуже
```

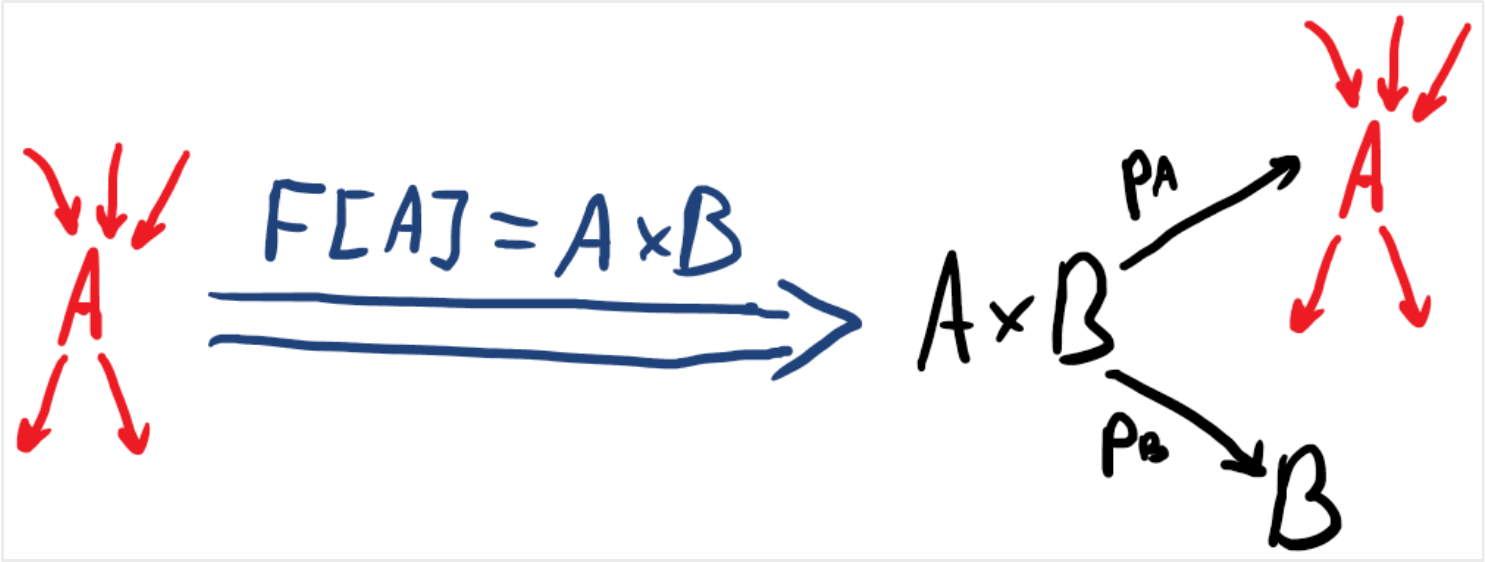
С другой стороны, Scala позволяет объявлять идентификаторы, состоящие из специальных символов, что может повысить читаемость:

```
type =>[A, B] = Function[A, B] // такой псевдоним уже есть в стандартной библиотеке
type  ×[A, B] = (A, B)         // новый псевдоним для типа произведения

// используем псевдонимы из спец-сиволов в инфиксной форме:
val func: Int => String = _.toString
val pair: Int × String  = (5, "пять")
```

Важно отметить, что с помощью ключевого слова `type` объявляются не сами типы, а *псевдонимы выражений* типов. Т.е. это некие синтаксические деревья, которые можно нарастить, например, в другом определении псевдонима типа, или же преобразовать с помощью упомянутого ранее механизма сопоставления с шаблонами типов. Эти выражения вычисляются в процессе типизации конкретных термов (например, переменных). С помощью различных построений псевдонимов типов можно получить схожие деревья выражения, дающие в итоге одинаковые типы, что не всегда может быть желанным результатом. Семантически различимые новые типы в Scala можно ввести только описав новые классы (трейты, объекты), либо используя литеральные типы. Работа с синтаксическими деревьями выходит далеко за рамками данной статьи.

Каждый тип характеризуется набором функций-стрелок, связывающих одни типы с другими. Обобщённые типы позволяют связывать между собой любые такие наборы стрелок с помощью функций, определяемых телом обобщённого типа (комбинации сумм, произведений, экспоненциалов и т.п.):



Обобщённый тип F[_] связывает с помощью проекторов произведения типов фиксированный тип B с типом, который может быть подставлен в тело F вместо A.

Обобщённые рекурсивные типы

Особого внимания заслуживает возможность объявлять типы рекурсивно, когда в теле выражения используется идентификатор объявляемого типа.

В Scala рекурсивные типы зачастую описываются в ООП-стиле, например,

```
sealed trait List[A]
case object Nil                                    extends List[Nothing] // почему
Nothing - будет рассказано далее
final case class Cons[A](head: A, tail: List[A]) extends List[A]

val myList: List[Int] = Cons(1, Cons(2, Cons(3, Cons(2, Cons(1, Nil))))) // 1,
2, 3, 2, 1
```

Рекурсивность `List` с первого взгляда, возможно, незаметна. Она спрятана за наследованием классов (`extends`). Соответствующие объявленным классам типы можно записать так:

$$\begin{aligned} List[A] &= Nil + Cons[A], \\ Cons[A] &= A \times List[A], \\ Nil &\cong 1. \end{aligned}$$

Если подставить два последних выражения в первое, то получается равенство, в котором рекурсия видна явно:

$$List[A] \cong 1 + A \times List[A].$$

(Если вместо A подставить тип $Nothing \cong 0$, то можно убедиться, что $List[Nothing] \cong 1 \cong Nil$.)

Таким образом, список - это либо пустой список, либо элемент (голова) и другой список (хвост).

Просуммируем такой список:

```
def sum(list: List[Int]): Int = list match
  case Nil              => 0
  case Cons(head, tail) => head + sum(tail) // тут рекурсия!
```

Прослеживается очевидная закономерность - для рекурсивных типов нужны рекурсивные методы!

Пользуясь только суммой и произведением типов с помощью рекурсии можно построить любые древовидные структуры, например,

```
sealed trait NonEmptyTree[A]
final case class Leaf[A](value: A) extends NonEmptyTree[A]
final case class Node[A](left: NonEmptyTree[A], right: NonEmptyTree[A]) extends
NonEmptyTree[A]
```

Трейту `NonEmptyTree` соответствует такой тип:

$$NonEmptyTree[A] \cong A + NonEmptyTree[A] \times NonEmptyTree[A].$$

Объявить рекурсивный тип в актуальной версии Scala без привлечения ООП несколько затруднительно. Например, такая конструкция не скомпилируется:

```
type List[A] = Unit + (A, List[A]) // illegal cyclic type reference...
```

Компилятор пытается "жадно" вычислить выражение тип, спрятанный за псевдонимом, но, определив бесконечную рекурсию, сообщает об ошибке. Выглядит как недоработка компилятора, так как есть обходной путь с использованием механизма сопоставления с шаблонами на уровне типов - там типы вычисляются "лениво" и рекурсия считается вполне легальной.

Такие обобщённые древовидные типы иногда называются *абстрактными типами данных* (неизменяемыми). Они описывают коллекции определённой структуры, которые могут "хранить" любое количество элементов любого типа. В частности, обычный список является вырожденной разновидностью дерева.

Нетривиальность концепции рекурсивных типов можно ощутить, если для построения задействовать экспоненциал типов. Например, давайте посмотрим на два симметричных друг другу типа:

```
case class SelfModifyingHandler[A](f: A      => SelfModifyingHandler[A])
case class Wtf[A]                 (f: Wtf[A] => A)
```

Попробуйте самостоятельно создать значения этих типов и в комментариях описать сценарии их использования (названия толсто намекают...).

Тема рекурсивных типов и рекурсии в целом настолько богата, что статья о ней может оказаться даже больше чем эта. В ней хотелось бы рассказать о таких вкусностях, как

- ограничения на рекурсию (или "что не так с типом `Wtf`");
- хвостовая рекурсия, трамплины;
- ряд Тейлора и корни многочленов для обобщённых рекурсивных типов ;
- производные от обобщённых рекурсивных типов;
- комбинаторы неподвижной точки;
- схемы рекурсии (ката-, ана- и прочие морфизмы).
  Очень надеюсь, что найдётся возможность подготовить и такой обзор.

Полиморфные типы

Разновидности полиморфизма

В интернете можно встретить различные определения полиморфизма. Вряд ли какое-либо из них можно считать “общепринятым”. Например, определение полиморфизма в Википедии отличается от аналогичного в Wikipedia. Там указаны немного разные источники, а в качестве первоисточника приводится статья 1967 года “*Fundamental Concepts in Programming Languages*” Кристофера Стрейчи, переизданная в 2000м. Но и там нет именно *определения* полиморфизма. Пожалуй, в наиболее общем виде это понятие можно сформулировать так:

> *Полиморфизм в программировании* – это свойство фрагмента кода определять разное поведение в зависимости от типов термов, использованных в нём.
> Такой код описывает целое "множество смысловых форм", то есть, является *полиморфным по типу*. В той статье Стрейчи говорится о двух основных разновидностях полиморфизма - универсальном (параметрическом) и специальном (ad-hoc).

Разные исследовали приводят множество разновидностей полиморфизма. Если собрать их все вместе, то наберётся целая дюжина. Но, так или иначе, любой такой подвид можно притянуть за уши либо к универсальному, либо к специальному (или же к их сочетаниям).

Отдельно стоит сказать про полиморфизм в языках с динамической типизацией. Как правило, в таких языках явно типизированными остаются лишь некоторые функции, например, встроенные арифметические операции. В остальных местах типы появляются у значений только в момент попытки их использования в типизированных функциях. Вне контекста выполнения типы термов просто не определены, что, в частности, не позволяет проводить статическую проверку корректности программы до её выполнения. Но теория типов создавалась больше для обеспечения корректности алгоритма в целом, *для любых начальных условий*, а не для проверки в некоторых частных случаев. Так что, не смотря на то, что при динамической типизации код является фактически полиморфным (может сработать со значениями разных типов), далее динамический полиморфизм не будет рассматриваться.

При нынешнем засилье объектно-ориентированной парадигмы у многих программистов могло сложиться представление о полиморфизме, как лишь об одном из "столпов ООП". Например, перейдя по одной из первых ссылок в результатах поиска по слову "полиморфизм", можно прочесть:

> Полиморфизм (polymorphism) — это понятие из объектно-ориентированного программирования, которое позволяет разным сущностям выполнять одни и те же действия. При этом неважно, как эти сущности устроены внутри и чем они различаются.

Не уверен, что имеет смысл упрекать авторов подобных цитат в том, что они вводят в заблуждение читателя. Популярность какого-либо мнения превалирует в массовом сознании над его качеством, корректностью. Давайте лучше рассмотрим ООП-шный полиморфизм чуть подробнее.

Полиморфизм подтипов

Рассмотрим конкретный пример:

```
trait Person { val name: String def sayHello(): String }
case class Français(name: String) extends Person { def sayHello() = "Bon après-
midi!" }
case class Русский (name: String) extends Person { def sayHello() = "Здаров!" }

def dialogWith(person: Person) = {
  println("Привет, " + person.name)
  println(person.sayHello())              // 1
}

val Француа = Français("François")
```

```
val Василий = Русский("Вася")
dialogWith(Француа); dialogWith(Василий)  // 2
```

Возможно, у кого-то возникнет впечатление, что полиморфизм тут проявляется в строке с пометкой `1`, ведь именно там конкретное поведение метода `sayHello` определяется "типом значения", спрятанного за переменной `person`. Но проблема в том, что, по определению, типы привязываются к термам языка (литералам, переменным, выражениям), а не к значениям, размещённым в памяти компьютера.

> Очередной раз замечу, что, да, у значения, зачастую, есть ссылка на описание класса, конструктором которого это значение создано, и класс связан с конкретным типом, который используется *во время выполнения* при проверке возможности использования значения в тех или иных операциях, но, как уже было сказано, основанный на этом динамический полиморфизм не помогает в проверке корректности алгоритма, и мы не будем его рассматривать.

В данном же примере у переменной `person` тип `Person` фиксирован в сигнатуре метода, следовательно, по (более корректному) определению из Википедии, тут полиморфизма нет. Но, согласно тому же определению, он есть в строке с пометкой `2` - при вызовах метода, принимающих параметр типа `Person`, используются переменные других типов, `Français` и `Русский`. В этой строке срабатывает неявное приведение от подтипов к супертипу. Именно неявное приведение от подтипа к супертипу определяет ООП-шный *полиморфизм подтипов*! Например, если бы с помощью `dialogWith` обрабатывалась коллекция значений типа `Person`, то полиморфным было бы добавление значений классов `Français` и `Русский` в эту коллекцию.

Вызов по имени метода, перегруженного в неизвестном подклассе, который определяется по значению `person` - становится менее "чудесным", если вспомнить, что методы суть значения функционального типа. По ссылке `person` можно добраться до размещённой в памяти таблицы виртуальных методов, а уже в этой таблице лежат *ссылки на методы* именно того подкласса, конструктором которого это значение было создано. Т.е. в предыдущем примере достаточно просто переписать эти три строчки (и выкинуть классы-наследники)

```
case class Person(val name: String, val sayHello: Unit => String)
val Француа = Person(       "François",          ()   => "Bon après-midi!")
val Василий = Person(       "Вася",              ()   => "Здаров!")
```

и получить тот же самый результат, но уже без привлечения полиморфизма.

Следуя Википедии, полиморфизм подтипов относится к разновидности *универсального полиморфизма с ограничениями подтипизации* - об этом будет далее. Но с другой стороны, сами эти ограничения можно реализовать с помощью *специального полиморфизма*, о котором будет рассказано в разделе **Классы типов**.

Универсальный полиморфизм и λ-исчисление

Само понятие универсального (параметрического) полиморфизма появилось в процессе развития λ**-исчисления**. Эта формальная система создавалось как фундаментальный язык, алгоритмы на котором можно было проверить на логическую корректность.

Изначально λ-исчисление было безтипововым. Там было введено понятие λ -абстракции с правилом аппликации (применения):

$$f := \lambda x.\ 2 \cdot x + 1;$$
$$f\ 5 \equiv f(5) = 2 \cdot 5 + 1 = 11.$$

Затем появилось просто-типизированная система $\lambda^{\rightarrow}$. Здесь у каждой переменной, которая связывается квантором λ фиксируется метка-тип (явно - в стиле Чёрча):

$$\lambda x : A.\, M;$$

Тип самого λ-выражения однозначно определяется по типу аргумента x и телу выражения M.

Обозначение $\lambda^{\rightarrow}$ (стрелочка) связано с тем, что простые типы можно выразить в кодировке Чёрча, т.е. как функции высшего порядка:

```
type Zero     = Any         // 𝟘 ≅ Any
type One      = Any => Any  // 𝟙 ≅ Any => Any
type Bool     = Any => One  // 𝟚 ≅ Any => Any => Any
type Natural  = One => One  // ℕ ≅ (Any => Any) => Any => Any - натуральные числа по Пеано
```

Например:

```
val ifTrue: Bool = happyWay => unhappyWay => happyWay

val one  : Natural = suc => z =>         suc(z)
val three: Natural = suc => z => suc(suc(suc(z)))

val choose: Bool = ifTrue
val result = choose(one)(three) // one
```

Дальнейшее развитие - полиморфная система $\lambda 2$ (система F). Теперь в λ-выражении стало возможно связывать не только обычные переменные, но и типовые:

$$\lambda A : \star.\ \lambda x : A.\ M;$$

Здесь $\star$ - это "тип типа". Система $\lambda 2$ позволяет определять *обобщённые функции*. В изоморфизме Карри-Ховарда типы соответствуют утверждениям, свойствам объектов, следовательно, система $\lambda 2$ ассоциируется с логикой второго порядка (отсюда и двойка в названии).

В Scala 3 добавлен специальный синтаксис для типов обобщённых функций:

```
def foldWithContext[A, B](aWithContext: WithContext[A])(folder: (A, String) =>
B)
  = folder(aWithContext._1, aWithContext._2)
val foldWithContextVal                                  // переменная
обобщённого функционального типа
  : [A, B] => WithContext[A] => ((A, String) => B) => B // тип      ∀A,B.
WithContext[A] → ((A, String) → B) → B
  = foldWithContext                                     // значение λA,B.
λa:WithContext[A]. λf:(A,String) → B. f(a._1, a._2)
```

Значения такого типа можно передавать в методы, в которых будет решаться, какие типы подставить для этих обобщённых функций.

Наконец ещё более продвинутая система $\lambda\omega$ (или $F\omega$) позволяет строить типы, используя λ-выражения

$$List = \lambda A.\ \forall B.\ (A \to B \to B) \to B \to B;$$

Именно тут появляются полиморфные (обобщённые) типы:

```
type List = [A] =>> [B] => (A => B => B) => B => B

def emptyList[A]: List[A] =
  [B] => (aggr: A => B => B) => (start: B) => start

def append[A](lst: List[A], a: A): List[A] =
  [B] => (aggr: A => B => B) => (start: B) => aggr(a)(lst(aggr)(start))

def fold[A, B](lst: List[A], start: B, aggr: A => B => B): B = lst(aggr)(start)

val myList = append(append(append(emptyList[Int], 1), 2), 3) // [1, 2, 3]
fold(myList, "список:", i => _ + " " + i)                    // "список: 1 2 3"
```

Сами выражения, на которых строятся полиморфные типы, определяют способы их использования - связанные с ними полиморфные функции. Компилятор сам может (его можно научить) строить такие функции для полиморфных типов, что может значительно сократить количество кода, которого приходится писать программистам вручную.

Ввиду того, что в таких полиморфных типах параметр связывается через квантор "*для любого*", параметрический полиморфизм называется *универсальным*. Полиморфизм, где используется дуальный квантор "*существует*", называется *специальным* - он будет рассмотрен позже.

Вариантность

Является ли список строк списком произвольных объектов? Конкретнее, можно ли передать терм типа `List[String]` в функцию, которая принимает аргумент типа `List[Any]`? Опытным путём можно убедиться, что в Scala (да и во многих других языках) это можно сделать. Но вот, наоборот, передать терм типа `List[Any]` в функцию, принимающую `List[String]` уже не получится. Получается, что `List[String]` является подтипом `List[Any]` и, очевидно, безо всякого наследования! Попробовав с другими простыми типами, можно заметить, что `List` *транслирует* отношения подтипизации для своих аргументов.

Но если мы попробуем провести аналогичные эксперименты с типами, скажем, `String => Int` и `Any => Int` то увидим прямо противоположную картину:

```
type Func[X] = X => Int // псевдоним для конструтора типов Function1

// компилируется, значит такая подтипизация корректна
summon[     String  <:<      Any]
summon[List[String] <:< List[Any]]
summon[Func[Any]    <:< Func[String]]
```

Получается, что конструктор типов `[X] =>> X => Int` обращает подтипизацию в обратную сторону! (Для дуального типа `[X] =>> Int => X` подтипизация снова окажется прямой)))

А вот если мы наивно попробуем то же самое со своим классом, у нас ничего не получится:

```
class Test[A](val a: A)
summon[Test[String] <:< Test[Any]]    // ошибка!
summon[Test[Any]    <:< Test[String]] // ошибка!
```

Чтобы разобраться в чём причина, достаточно ~~просто прочесть документацию~~ посмотреть на определение типов `List` и `Function` (`=>`) в стандартной библиотеке:

```
type List    [+A]     = scala.collection.immutable.List[A]
type Function[-A, +B] = Function1[A, B]
```

Именно значки перед аргументами конструкторов типов определяют вид *вариантности*:

- `+` означает прямую *ковариантную* трансляцию подтипизации,
- `-` – инвертированную, *контравариантную*,
- нет значка – подтипизация блокируется – *инвариантность*.
  (Псевдонимы типов по-умолчанию пробрасывают вариантность своих выражений без изменений.)

!!!!! ПЕРЕПРОВЕРИТЬ !!!!!
В представленном выше классе `Test` параметр `A` используется двояко - и как тип аргумента конструктора (контравариантно), и как тип проектора поля `a` (ковариантно). Следовательно, полиморфный тип, связанный с этим классом, является инвариантным. Попытки поставить в квадратных скобках перед `A` плюс или минус приведут к ошибке компиляции вида `contravariant type A occurs in covariant position in type A of value a` ("контравариантный тип A появляется в ковариантной *позиции* типа переменной a"). О какой же позиции тут идёт речь?

На первый взгляд может показаться, что имеется ввиду позиция "относительно стрелки" `=>`:

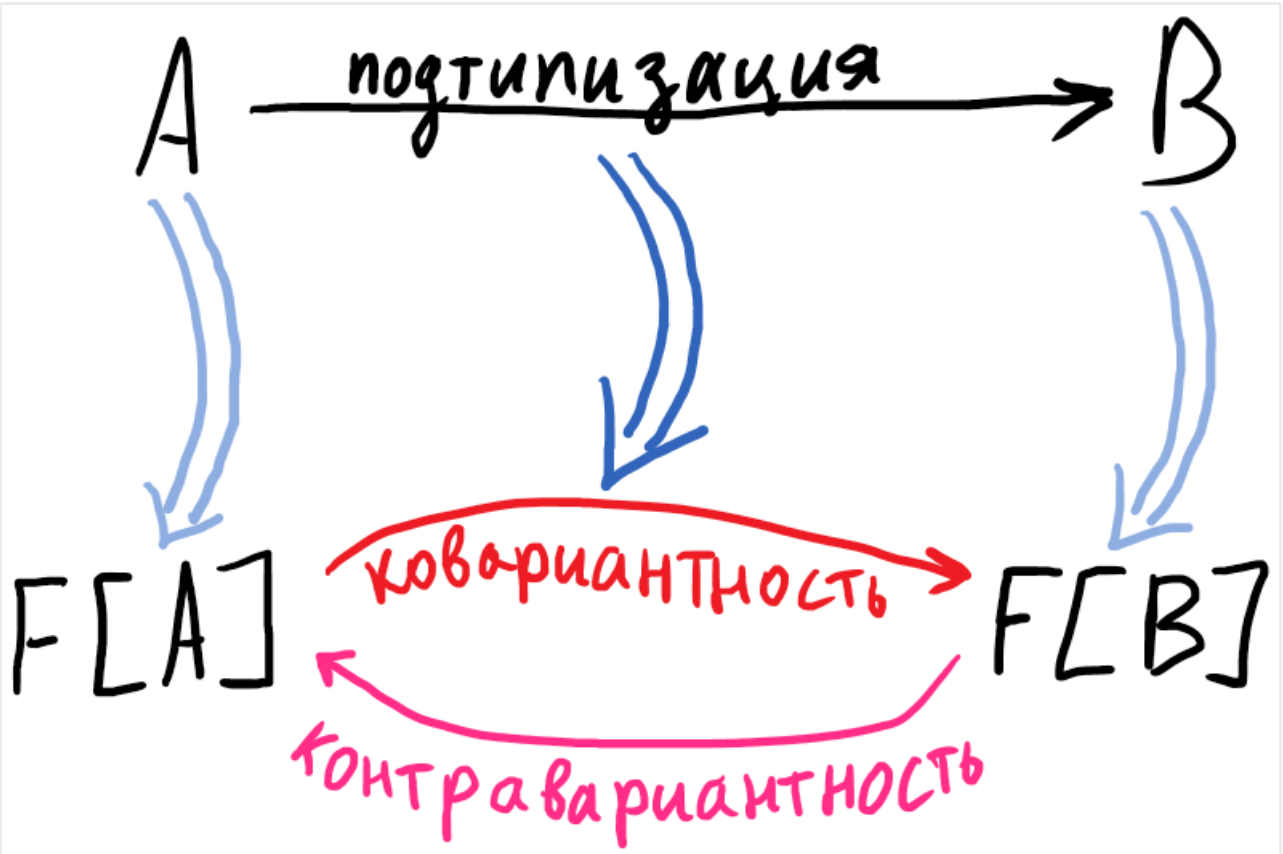
- отрицательная позиция (знак `-`), если слева от стрелки и
- положительная позиция (знак `+`), если справа.
  Но не всё так просто))). Например, вот этот тип внезапно оказывается ковариантным:

```
type Func[+X] = (X => Int) => Int
summon[Func[String] <:< Func[Any]]
```

Аргумент `X` стоит левее сразу двух стрелок - дважды отрицательная позиция превращается в положительную! О причинах такого поведения знаков будет сказано далее в разделе **Типы-контейнеры**.

Как уже не раз упоминалось, отношение подтипизации - это просто функция от одного типа к другому (неявное преобразование). Тогда, вариантность - это то, как подобные функции преобразуются "под действием" конструкторов типов. Такие преобразования описываются специальным классом типов - функтором

(ковариантным или контравариантным). Про классы типов будет рассказано в **соответствующем разделе**.



Тип вариантности полиморфного типа F[_] проявляется в том, как отношения подтипизации (функции неявных преобразований) между A и B переносятся на типы F[A] и F[B]. Синими стрелками обозначены ключевые черты функтора: ковариантного или контравариантного, в зависимости от направления красной стрелки.

В дополнение можно рекомендовать эту статью: Demystifying Variance in Scala.

Ограничения подтипизации

Привычный ООП-шный полиморфизм подтипов подразумевает, что в любую функцию мы можем передавать значения *любых типов*, являющихся подтипами к типу соответствующего параметра функции.

```
trait SupperClass { val i: Int }
case class SubClass(i: Int) extends SupperClass

def someFunc(x: SupperClass) =
  x.i + 3

val sub = SubClass(5)
someFunc(sub) // полиморфизм тут
```

Это эквивалентно ситуации, как если бы функция явно декларировала, что может принимать значения *разных типов* (т.е. имеет тип-параметр, превращающий её в обобщённую), но для которых задано определённое ограничение подтипизации.

Такая система типов (система $F_{<:}$), в которой универсальный полиморфизм сочетается с ограничениями подтипизации, реализуется во многих популярных языках, в том числе и в Scala. Например, во фрагменте кода выше достаточно изменить сигнатуру метода `someFunc`, чтобы продемонстрировать полиморфизм явно:

```
def someFunc[A <: SupperClass](x: A) =
  x.i + 3 // тело функции не меняется
```

Таким образом, система $F_{<:}$ позволяет относить полиморфизм подтипов к разновидности универсального полиморфизма.

Ограничения подтипизации для параметров обобщённых методов мощнее, чем полиморфизм подтипов. Рассмотрим пару примеров.

Следующий фрагмент кода показывает, что полиморфизм с ограничением подтипизации позволяет без потерь передать тип аргумента на выход функции:

```
trait SupperClass { val i: Int }
case class SubClass(i: Int, s: String) extends SupperClass // в подклассе новое
поле s
val sub: SubClass = SubClass(5, "string")

def extractInt1[A <: SupperClass](x: A): (A,           Int) = (x, x.i)
def extractInt2(x: SupperClass)        : (SupperClass, Int) = (x, x.i)

extractInt1(sub)._1.s // "string"
extractInt2(sub)._1.s // !!! ОШИБКА !!! Компилятор не видит тут никакого s!
```

В последней строке будет ошибка компиляции, так как функция `extractInt2` ничего не знает об исходном типе переданного ей аргумента - неявное преобразование подтипизации сработало *до того*, как значение было передано в функцию.

Выше были представлены такие ограничения, когда тип-параметр должен быть чьим-то подтипом - так называемые, *верхние границы типа*. Но существует также понятие *нижней границы типа*: запись `X >: SubType` означает, что тип-параметр `X` должен быть супертипом для некого фиксированного типа `SubType`. Наглядный пример использования такого ограничения можно подсмотреть в официальной документации Scala.

В ранних версиях Scala была возможность использовать, так называемые, *ограничения представления*. Запись `A <% SomeType` означала, что известно некое неявное преобразование от типа `A` к типу `SomeType`, то есть, возможности типа `SomeType` применимы к значениям типа `A`. Такое преобразование вовсе не обязательно завязывалось на наследовании классов, вариантности обобщённых типов (`List[String] <: List[Any]`) или встроенной подтипизации (`Short <: Int`) - могут быть использованы в том числе и пользовательские неявные преобразования:

```
given def intToSubClass(i: Int) // объявляем неявное преобразование Int =>
SubClass
  = SubClass(i, "нечто")
def getString[A <% SubClass](a: A) = a.s

getString(5) // "нечто"
```

В Scala 3 такой синтаксис убрали, но эффектов, аналогичных использованию ограничений `<%` можно достичь с помощью классов типов - об этом будет далее.

!!! Проверить ещё и подтипизацию `F[_] <: List[_]` !!!
Отношения подтипизации можно ввести не только для простых типов, но и для полиморфных. Сделать это можно разными способами, но, пожалуй, самый очевидный из них - это *поточечная подтипизация*: считаем `F` подтипом `G`, если `F[X]` является подтипом `G[X]` для любого типа `X`. При подготовке статьи очень удивился, обнаружив, что такая подтипизация уже присутствует в Scala:

```
def headOption[F <: [X] =>> Iterable[X], A](fa: F[A]): Option[A] = fa.headOption

headOption(List(5))           // Some(5)
headOption(Option("строка")) // Some("строка")
headOption(Map(1.2 -> true)) // Some((1.2, true))
```

К сожалению, в актуальной на данный момент версии Scala в ограничении приходится явно указывать λ-выражение для типов, что выглядит коряво. Тем не менее, язык живёт, меняется, так что посмотрим, что будет дальше.

Очевидно, поточечную подтипизацию по индукции можно распространить и на полиморфные типы более высокого рода. Так что давайте посмотрим на обобщённые типы с точки зрения их "высокородности".

Типы высокого рода

Higher Kinded Types

В математике существуют такие понятия, как

- функционал, превращающий функций в "простые" нефункциональные значения (например, интегральная свёртка),
- оператор, превращающий одни функции в другие (например, дифференцирование).
  В отличие от обычных функций, такие отображения связывают не только простые множества, вроде действительных чисел, но и множества функций!

Отображения, из функций и/или в функции в информатике называются *функциями высших порядков* (higher-order functions). Такие функции могут оперировать не только "простыми" функциями (первого порядка), но и такими же функциями высших порядков. При этом, "порядок" новой функции будет выше чем, у тех, которые она принимает/возвращает.

> Функция, как набор инструкций, размещённых в памяти, представляет собой обычное значение, и даже в Ассемблере можно было передавать указатель на функцию в другую функцию, вызывать её по указателю, или возвращать его. Но сам термин "функции высших порядков" вошёл в оборот, только кода в популярных высокоуровневых языках появилась *возможность использовать функции как значения* - "функции первого класса".

В свою очередь, конструкторы типов, как уже было замечено, представляют собой функции на уровне типов - отображают одни типы в другие. Имеющиеся в системе $F2$ "простые" конструкторы типов (переводящие один простой тип в другой) в системе $F\omega$ расширяются *функциями высших порядков на типах* - **higher kinded types** (HKT).

Слово "kind" вызывает известные трудности перевода. Встречаются такие варианты:

- **кайнд** - незамысловатая транслитерация, заимствование из английского, которое не все считают оправданным;
- **сорт** в русском языке привычно индексируется натуральными числами, что порождает эклектику вроде "типов третьего сорта";
- **вид** типа кажется хорошим вариантом перевода, если избегать словосочетания "типы высокого вида";
- **род** привносит в предметную область не очень уместную, но ставшую уже привычной, "высокородность".
  Далее будут рассмотрены *виды* типов высокого *рода*)), так что будут использоваться два варианта перевода слова "kind".

Рассмотрим примеры:

```
type Option = [A] =>> Unit | A
type Reader = [Dep] =>> [A] =>> Dep => A
```

```
type Functor = [F[_]] =>> [A, B] => (A => B) => (F[A] => F[B])
type TaglessFinalProgram = [Algebra[_[_]]] =>> [A] =>> [F[_]] => Algebra[F] =>
F[A]
```

Конструктор `Reader` принимает на вход тип зависимости и возвращает другой конструктор, принимающей тип результата и возвращающий тип функции. По сути, `Reader` является каррированной версией конструктора типов, принимающего два параметра. В свою очередь `Functor` принимает на вход уже не простой тип, а другой конструктор, и возвращает простой тип полиморфной функции.

Не каждый тип может быть подставлен в эти конструкторы типов:

```
type FOpt = Functor[Option]
//type FReader1 = Functor[Reader]       // нельзя!
type FReader2 = Functor[Reader[Int]]
//type FInt = Functor[Int]              // нельзя!
type OptInt = Option[Int]
//type OptReader1 = Option[Reader]      // нельзя!
//type OptReader2 = Option[Reader[Int]] // нельзя!
type OptReader3 = Option[Reader[Int][String]]
type OptF = Option[Functor[Option]]
//type Reader1 = Reader[Int]            // нельзя!
type Reader2 = Reader[Int][String]
```

Полиморфные типы добавляют красок в однообразие простых типов - теперь типы не все одинаковые)). Но это приносит не только радость. Ведь даже для простых конструкторов типов вроде `Option` возникает вопрос: можно ли считать типы высокого рода, собственно, типами, если их нельзя приписать к значениям, как метки? Какой смысл может иметь запись `val opt: Option`?

Можно, конечно, пофантазировать, сказав, что подставив в значение `opt` конкретный тип, мы сможем получить новый тип, например `val intOpt: getType[Int](opt) = Some(42)`. Конечно, всё зависит от языка, и, хотя конкретно так Scala не умеет, но с помощью определённых костылей (основанных на полиморфных функциях и зависимых типах) можно добиться схожего результата, но…

Действительно, для типов высокого рода не предусмотрено использование в качестве меток термов, так что на самом деле **полиморфные типы не являются типами**! (См. также тут.) Конструкторы типов называют типами скорее по аналогии с функциями высших порядков, которые все являются значениями (функции первого класса). В Scala есть синтаксис, использование которого позволяет трактовать конструкторы типов как метки, но уже не для значений, а для других типов - это будет рассмотрено при обсуждении классов типов.

Другие примеры использования типов высокого рода можно подсмотреть, например, тут:

- Higher-kinded data in Scala,
- GitHub-проект DataPrizm (и статья об использовании типов высокого рода, в этом проекте).

Логика высших порядков

С точки зрения изоморфизма Карри-Ховарда полиморфные типы высокого рода связаны с логикой высших порядков. Например, тип

```
type Product = [A, B] =>> (A, B)
```

можно прочесть как
"*Для любых типов* `A` *и* `B` *из их обитаемости следует обитаемость типа-произведения* `A × B`."
Изоморфное этому типу утверждение в логике второго порядка будет выглядеть так:
"*Для любых высказываний* `A` *и* `B` *из их истинности следует истинность высказывания* `A ∧ B`."

Именно наличие квантора "для любого" (или "существует") над типами/высказываниями определяет с одной стороны изоморфизма "высокородность" типа, а с другой - высший порядок логики. Связь более сложных полиморфных типов с логиками высших порядков выводится по индукции.

Предоставление реализации некоторого метода является доказательством обитаемости соответствующего функционального типа. Если компилятор не ругается на такую реализацию, следовательно, тип результата метода удалось *логически корректно* связать с типом его аргумента. С полиморфными типами (функциональными отображениями из типов в тип) ситуация совершенно аналогичная. Полиморфные типы представляют собой инструмент мощной выразительной силы, но при этом сохраняют ключевую особенность простых типов - благодаря изоморфизму Карри-Ховарда, компилятор по-прежнему может проверить корректность алгоритма до его исполнения. (Да, рекурсивные типы, упомянутые ранее, могут усложнить проверку корректности, а то и вовсе сделать её невозможной, но это отдельная история.)

Вид типа

Приведённые выше примеры демонстрируют, что типы высокого рода могут относится к разным **видам типа** (type kinds). В определениях этих типов однозначно указывается, параметры какого вида типов они принимают: `A`, `F[_]`, `Algebra[_[_]]`.

Такие параметры должны быть либо простыми типами, либо другими конструкторами типов - отображениями из типов в простые типы.

Для видов типов разработана своя система обозначений. Вид простых типов обозначается звёздочкой $\star$. Вид конструкторов типов, принимающих на вход простые типы, записывается как $\star \to \star$. Все остальные виды типов строятся по индукции. В языках семейства ML (том же Haskell) функции сразу записываются в каррированном виде, это же правило относится и к конструкторам типов. В Scala использован более традиционный синтаксис, когда функция может принимать несколько параметров, разделённых запятыми - также реализуются и конструкторы типов. В итоге получаем следующую формализацию:

> *Вид типа - это либо вид простых типов ($\star$), либо отображение ($\to$) из одного вида типа (или нескольких через запятую) в другой вид типа.*

Стрелка, как оператор, обычно подразумевает правую ассоциативность, поэтому вид $\star \to (\star \to \star)$ эквивалентен виду $\star \to \star \to \star$. Таким образом, в конце каждого вида типа после стрелочки будет вид простого типа $\star$, следовательно, любой вид типа описывает конструкторы простых, истинных типов!

В этой таблице демонстрируется соответствие видов типа и параметров конструкторов типа:

Вид типа	**Параметр конструкторов типа этого вида**	**Пример типа этого вида**
$\star$	`A`	`Int`
$\star \to \star$	`F[_]`	`Option`
$\star \to \star \to \star$	`F[_][_]`, но Scala не умеет работать с параметрами каррированного вида((	`Reader` из примера выше
$(\star \to \star) \to \star$	`Alg[_[_]]`	`Functor`
$((\star \to \star) \to \star) \to \star$	`Prog[_[_[_]]]`	`TaglessFinalProgram` из примера выше
$(\star, \star) \to \star$	`M[_, _]`	`Map`

Ещё раз стоит обратить внимание на вид типа `Functor` из примера выше - в Scala конструкции вроде `[A, B] => ...` обозначают *простой тип* полиморфной функции, поэтому в итоге для `Functor` и получается вид $(\star \to \star) \to \star$.

В Scala REPL (интерпретатор командной строки) есть возможность подсмотреть вид типа, воспользовавшись командой `:kind` (с опциональным параметром `-v`). Например, для типа `Functor` из библиотеки Cats вывод команды будет такой:

```
scala> :kind -v cats.Functor
cats.Functor's kind is X[F[A]]
(* -> *) -> *
This is a type constructor that takes type constructor(s): a higher-kinded type.
```

К сожалению, команда `:kind` работает только с определёнными ранее обобщёнными классами (трейтами). Псевдонимы типов или λ-выражения эта команда не воспринимает.

При подготовки статьи была идея реализовать что-то вроде *типа вида типа*))):

```
type Kind
type GetKind[A <: AnyKind] // <: Kind   // (про AnyKind будет дальше)
def instantinate[K <: Kind]: Kind = ??? // конструтор значения типа вида типа
def toString(kind: Kind): String = ???  // строка вида "(⋆ → ⋆) → ⋆"
```

Сам тип `Kind` можно было бы определить как-то так:

```
opaque type ⋆ = Unit                    // вид простых типов
type -->[K1, K2] = (K1, K2)             // просто некая стрелка
type Kind = ⋆ | (Kind --> Kind)         // тип вида типа (без множественных
параметров)
```

Но, такой код, к сожалению, не компилируется - рекурсивные определения псевдонимов типов не поддерживается в Scala. Есть различные обходные манёвры для этого препятствия, но, так или иначе, все они завели в тупик. Чего-то похожего можно добиться, предоставив неявные экземпляры некоторого класса типов, но такой метод работает *только для конечного числа* видов типа. Если у кого-то из читателей получилось победить компилятор Scala и реализовать эту задумку для всех видов типов, пожалуйста, поделитесь рецептом в комментариях.

Вычисления на уровне типов

Итак, у нас есть возможность строить алгебраические выражения из типов, а также строить функции из типов в тип. И этих возможностей достаточно, чтобы заняться программированием на уровне типов!

Рассмотрим такие определения типов:

```
type ⊥        = Unit
type ⊤        = [A] =>> A

type IfFlase  = [False] =>> [True] =>> False
```

```
type IfTrue   = [False] =>> [True] =>> True

type Zero     = [Z] =>> [Succ[_]] =>> Z
type One      = [Z] =>> [Succ[_]] =>> Succ[Z]
type Two      = [Z] =>> [Succ[_]] =>> Succ[Succ[Z]]
type Three    = [Z] =>> [Succ[_]] =>> Succ[Succ[Succ[Z]]]
```

Использование представленных типов сопряжено с трудностями, связанными с отсутствием поддержки в Scala каррированных типов-параметров, но всё же возможно. С аналогичным решением можно ознакомится в статье Typelevel Church Numerals in Scala 3 - там также определены типы сложения и умножения для таких "натуральных чисел".

Принадлежность представленных выше типов одному "типу" определяется *видом эти типов*:

```
type Zero                 // *
type One[A]               // * → *
type Bool[False, True]    // * → * → *       тут подразумевается каррирование
type Natural[Succ[_], Z]  // (* → *) → * → *
type Pair[Bool[_, _]]     // (* → * → *) → * тут тоже
```

И это логично, так как вид типа является, по сути, типом типа! Не сложно заметить, что представленные тут виды типов коррелируют с типами в кодировке Чёрча, представленными **ранее**.

Трактовка таких конструкторов типов достаточно очевидна. Например, типы натуральных чисел соответствуют итерациям применения некого конструктора типов `Succ` к типу `Z`. Но также эти типы можно интерпретировать, как *фантомные*, не предусматривающие использование в качестве меток термов, но которые можно передавать в другие конструкторы типов. С помощью таких фантомных типов можно, например, построить тип списка фиксированной на этапе компиляции длины - при конкатенации длинна нового списка будет также рассчитываться на этапе компиляции. И, в принципе, алгоритмы на уровне типов используются именно для каких-либо вычислений на этапе компиляции. Эта идея хорошо раскрывается в habr-статье Решение задачи о 8 ферзях на трёх уровнях Scala — программа, типы, метапрограмма.

Плюсом программирования на типах является его *чистота* с точки зрения функционального программирования. Нет никаких эффектов – вообще никакого “нечистого” взаимодействия с окружением. Но это же является и минусом, ведь каждый такой алгоритм может решить лишь единственную задачу, условия которой известны до компиляции. Во многих программах есть как минимум одна такая

сверхзадача - построение корректной композиции функций, благодаря которой в процессе работы приложения обычные значения будут передаваться между объектами окружения (клавиатура, монитор, память, БД, web-службы и т.п.) согласно техническому заданию. Так вот с такой сверхзадачей программирование на типах справляется вполне успешно.

Другие примеры программирования на типах можно найти тут:

- Type-Level Programming in Scala из журнала Apocalisp.
- Type-Level Programming in Scala с ресурса Rock the JVM Blog (Part I, Part II, Part III)
- Meta-Programming with Scala: (Part I, Part II, Part III)
- Презентация Type-Level Computations in Scala

Также посмотреть такие реализации SKI на уровне типов:

- Scala type level encoding of the SKI calculus
- GitHub-проект Type level lambda calculus in Scala

Полиморфизм видов

Есть ли какие-нибудь операции, которые можно было бы сформулировать единообразно для полиморфных типов разного вида? Конечно, есть - это и отношения конструкторов типов (в частности, эквивалентность), и банальный проброс типа произвольного вида (аналог функции `identity` на значениях), какие-либо модификации вида типа (например, усложнение). Вариантов множество.

Такого рода операции над конструкторами типов должны быть полиморфными по виду Полиморфизм видов (kind polimorphism) также называют *полиморфизм высокого рода* (higher kinded polymorpism), подразумевая, что в этом случае квантификация идёт по *типам типов*. Язык Haskell позволяет работать с видами типов также как и с обычными типами, соответственно, для них также реализован и полиморфизм видов. В Scala 3 же для этих целей предусмотрен специальный синтаксис, похожий на ограничение подтипизации для параметров обобщённых типов: `<: AnyKind`:

```
type SomeTypeConstructor[A <: AnyKind]
```

Из за такой записи может показаться, что `AnyKind` является ещё одним типом, но это не так. На самом деле это просто метка для компилятора, допускающая вольную трактовку вида типа-параметра.

Практическое применение полиморфизма видов в Scala ограничено реализацией поддержки `AnyKind`. На упомянутой ранее странице официальной документации

Scala до сих пор есть такая строка:

> (todo: insert good concise example)

В интернете же можно найти такие примеры использования `AnyKind`:

- внутри Scala, например, при работе с макросами;
- проверка концепции полиморфизма видов, по уже упоминавшейся ранее ссылке;
- две любопытных реализации моноида, полиморфного по видам типа: gist s5bug, gist mandubian;
- Implement SKI combinator calculus с использованием `AnyKind`.
  Если у читателей есть другие примеры, пожалуйста, приведите их в комментариях.

Вселенные типов

Итак, виды типов - это типы типов высокого рода. Но “типы типов” не могут сами быть типами – это привело бы к так называемому *парадоксу Жирара*.

> Парадокс Жирара в теории типов – это аналог парадокса Рассела, демонстрирующего, что совокупность всех множеств сама не может быть множеством. В интернете сложно найти более-менее популярную литературу по парадоксу Жирара, разве что pdf-статья Хёркенса A Simplification of Girard’s Paradox.

Наличие логических парадоксов может скомпрометировать всю теорию типов, делая её ненадёжной. Во избежание этого вводится концепция *вселенных типов*: типы высокого рода относятся к “обычной” вселенной типов U_0, в то время как все виды типов принадлежат уже к другой вселенной U_1. В Haskell реализована работа с типами из обеих вселенных и считается, что этого достаточно для задач языка.

Тем не менее, аналогию между видами типов и самими типами можно продолжить, введя конструкторы видов типа – виды высокого рода! Соответственно, для классификации таких полиморфных видов типов потребуется ввести некие *виды видов типов*, которые, во избежание всё того же парадокса Жирара, будут относится уже к следующей вселенной типов U_2. Таким образом по индукции можем получить бесконечную последовательность вселенных типов: $U_0,\ U_1,\ U_2,\ \ldots$

Интересной является возможность написания кода, который бы одинаково работал для типов из любой вселенной. Например, ранее был представлен вид типов, аналогичный типу натуральных чисел, и таким же способом можно описать обитателей и более далёких вселенных. Такая возможность уже реализована в некоторых языках (Agda, Coq) и называется она “**полиморфизм вселенных**“.

Можно также заметить, что вселенная видов типов похожа на вселенную типов примерно также, как типы похожи на значения – выше было продемонстрированно, как вычисления можно производить на уровне типов почти также, как это можно делать и со значениями. Это даёт основания для включения в последовательность вселенных типов ещё одну – *вселенную значений* с не очень удачным индексом U_{-1}. Она является выделенной, начальной, в том смысле, что “типы” этой вселенной уже больше ничего не индексируют, вселенных с меньшими индексами просто не существует (но так ли это?))).

Пожалуй, самое интересное в концепции вселенных типов то, что между ними можно путешествовать! Для этого достаточно построить функцию от типа из одной вселенной к типу другой. И хотя в Sacla для взаимодействия с далёкими вселенными есть только упомянутый ранее `Anykind`, но всё же доступны червоточины между U_{-1} и U_0, то бишь между значениями и типами.

Отображение из U_0 в U_{-1} уже встречались в этой статье – это обобщённые функции. Действительно, достаточно передать туда типы-параметры, и обобщённая функция становится обычным значением функционального типа:

```
val doubleA                   // червоточина
  : [A] => A      => (A, A)   // из вселенной типов в значение-функцию
  = [A] => (a: A) => (a, a)   // техника сейчас не важна

val doubleInt                 // сюда положим значение
  : Int => (Int, Int)         // простого типа
  = doubleA[Int]              // вжух!
```

В Scala существуют также возможности и обратного путешествия – от значений в типы! О, это очень сильное колдунство под названием **зависимые типы**. Посмотрим на такой код:

```
trait IHaveAType { type Type; val v: Type }     // трейт содержит членом
абстрактный тип!
val valWithInt:    IHaveAType =                 // червоточина 1
  new IHaveAType { type Type = Int;    val v = 42 }
val valWithString: IHaveAType =                 // червоточина 2
  new IHaveAType { type Type = String; val v = "вселенная" }

val doubleDep // функция, ТИП результата которой зависит от ЗНАЧЕНИЯ первого
аргумента
  : (carier: IHaveAType) => (carier.Type, carier.Type)       // вжух туда!
  = (carier: IHaveAType) => doubleA(carier.v)                // вжух обратно!

val intsPair:    (Int, Int)       = doubleDep(valWithInt)    // (42, 42)
```

```
val stringsPair: (String, String) = doubleDep(valWithString) // ("вселенная",
"вселенная")
//                ^^^^^^^^^^^^^^^^
// указание иных типов приведёт к ошибке компиляции!
// а всё потому что valWithString: IHaveAType { type Type = String }
```

Тут в строке `вжух туда!` выражение `carier.Type` телепортирует из U_{-1} в U_0, а в следующей строке возвращаемся обратно уже знакомым способом (тип-параметр `doubleA` определяется компилятором автоматически по аргументу). В последних строчках при вызовах `doubleDep` передаются значения одного типа `IHaveAType` но получаются значения разных типов, определяемых исходными значениями. Зависимые типы не редко используется в Scala – по большей части в популярных библиотеках, но можно встретить и в продуктовом коде. Это очень интересная, но очень объёмная тема, рассмотрение которой стоит отложить на другой раз.

Сама функция, переводящая из одной вселенной в другую, относится ко вселенной с максимальным индексом, и в случае описанных выше путешествий, это была вселенная типов U_0. Но чаще считают, что такие функции работают с типами одной и той же вселенной, однако действует простое, но важное правило: вселенные со старшими индексами *включают все вселенные* с меньшими: $U_{-1} : U_0 : U_1 : U_2 : \dots$ Таким образуется *кумулятивная иерархия вселенных*. А вот мультивселенной U_∞, включающей в себя все остальные вселенные попросту не существует (это всё выдумки киношников!) из-за всё того же парадокса Жирара.

Ещё почитать про вселенные типов можно тут:

- пара ответов на вопросы о кумулятивных вселенных;
- статья на nLab;
- pdf-статьи:
  - On Universes in Type Theory,
  - Type Theory with Explicit Universe Polymorphism, -
  - Notes on Universes in Type Theory.

Путешествие к далёким вселенным таит множество опасностей для неподготовленного приключенца. Давайте пока вернёмся в нашу вселенную, к привычным классам и типам.

Классы типов

Это не типы классов!

Если “типы классов” – это, буквально, типы, ассоциированные с классами, то перестановка слов обладает совершенно иной семантикой. Давайте разберёмся,

какой смысл имеет словосочетание “классы типов”.

Рассмотрим такой код:

```
case class Context[A](meta: String)
def enrichWithMeta[A](a: A, context: Context[A]) = (a, context.meta)
//                 1     4?             2    3

val contextForInt = Context[Int]("целые числа")
enrichWithMeta[Int](42, contextForInt) // (42, "целые числа")
```

Нет, никакой особой магии тут искать не нужно, сейчас важна лишь определённая трактовка элементов этого кода. Обобщённый метод `enrichWithMeta` параметризирован неким типом `A`, и в качестве одного из аргументов принимает значение `Context[A]`, то есть, обобщённого типа, параметризированного тем же типом `A`. Ключевой вопрос тут – какие типы можно использовать при вызове метода `enrichWithMeta`? С первого взгляда кажется очевидным, что универсальный полиморфизм позволяет использовать *любой тип* в качестве параметра.

Однако, если рассматривать аргумент `context: Context[A]` как *вспомогательный* (в любом смысле!), то правильным может стать и такой ответ: *при вызове метода `enrichWithMeta` можно использовать не любые типы, а только те `SomeType`, для которых существует значение `Context[SomeType]`* (с тем, чтобы его передать в метод, как аргумент). Если же значение типа `Context[SomeType]` не доступно, то вызвать метод `enrichWithMeta` не получится.

Таким образом, само требование предоставления значения обобщённого типа, конкретизированного типом-параметром метода, накладывает ограничения на этот тип-параметр. Причём, важен не сам аргумент метода, а только его тип, связанные с ним возможности-функции. Более того, так как тип-параметр может быть любой, то за ограничение отвечает не простой тип `Context[A]`, а конструктор типов `Context`. Во фрагменте кода выше ключевые моменты отмечены цифрами: тип-параметр метода (1), тип аргумента метода, полученный применением конструктора типа (2) к типу-параметру метода (3). Про это будет сказано далее, но пока предлагаю задуматься, можно ли считать схожим ограничением тип первого аргумента (4)?

В математике ограничение, позволяющее классифицировать некую совокупность на “подходящее” сущности или “неподходящее”, называется *классом*. Привычные для программистов классы фильтруют значения – они являются *классами значений*. В то же время, использованные представленным выше способом обобщённые типы классифицируют другие типы, следовательно, их можно трактовать как **классы типов**.

В хабр-статье Что такое класс типов? утверждается, что для точного определения класса типов могут потребоваться некие *законы* – дополнительные ограничения на те возможности, что предоставляет класс типов. Законы (как правило, некие отношения, в частности, равенства) нужны для математических абстракций, которые формулируются в виде классов типов, но для того, чтобы просто обобщённый тип считать классом типов никакие законы требуются. В той статье как раз ставится открытый вопрос, можно ли некий обобщённый класс называть классом типов.

Тут важно заметить, что классы типов и обобщённые типы – это *разные понятия*. Существуют так называемые *фантомные типы*, для которых вообще нельзя сконструировать значение, следовательно, ни при каком раскладе классов типов из них не выйдет. Прочие *обобщённые типы можно трактовать как описания классов типов*, но всё же это подразумевает *определённый способ использования* таких типов.

Здесь приведён пример очень простого обобщённого класса `Context[A]`, предоставляющего единственный проектор типа `String`, и совсем не использующий параметр `A`. Ближе к концу раздела будут рассмотренные более полезные примеры классов типов, предоставляющие различные функции, задействующие тип-параметр. Пример `Context[A]` призван подчеркнуть, что классы типов могут быть описаны любыми (обитаемыми!) обобщёнными типами.

Также далее будет рассказано, что именно предлагает Scala для работы с классами типов, но сперва стоит обратить внимание на использованную формулировку ограничения на тип-параметр `enrichWithMeta[A]`: *существуют такие типы* `A`*, для которых доступно значение* `Context[A]`. Такого рода логические утверждения при переносе в теорию типов соответствуют понятию “*экзистенциальные типы*”.

Экзистенциальные типы

Рассмотрим выражение `A Either String`, определяющее сумму типов `A` и `String`. Здесь `A` - свободная переменная, и для использования выражения её нужно связать каким-либо квантором. Ранее для мы пользовались только квантором λ (или аналогичным $\forall$), который можно читать как "для любого":

```
//   OrStringA = λA.     A   +    String
type OrStringA = [A] =>> A Either String
```

"Для любого типа `A` есть тип-сумма `A` и `String`". Стрелка в определении типа демонстрирует, что речь идёт именно о *функциональном отображении* из типов в типы.

Дуальным к утверждению "для любого `A`" является утверждение "существует такой `A`, что". В математике, в том числе и в теории типов, для таких случаев используется квантор ∃ (exists). Если этот квантор использовать с выражением из примера выше, то получится такой тип: $OrStringE = \exists A.\ A + String$.

> Дуальность кванторов ∀ и ∃ в логике проявляется по отношению к отрицанию:
>
> - утверждение *“**не** для любого `A` верно `B`”* равносильно *“существует такое `A`, что **не** верно `B`”*;
> - утверждение *“**не** существует такого `A`, что верно `B`”* равносильно *“для любого `A` **не** верно `B`”*.

Что может быть значением такого типа? При трактовке типов как утверждений (изоморфизм Карри-Ховарда) доказательством корректности определения типа является его обитаемость, *свидетельством* которой является предоставление любого "значения" этого типа. Для полиморфного типа $OrStringA = \lambda A.\ A + String$ его "значением" является определение `[A] =>> A Either String`, отражающее "функциональный" характер квантора λ ("для любого"). В случае типа $OrStringE$ в качестве свидетельства мы должны прежде всего предъявить "тот самый" тип (например, `Int`), существование которого утверждается, и вместе с ним значение выражения типа, в котором свободный параметр заменён на "тот самый" тип (в нашем случае, `Int Either String`). То есть, в итоге получаем такой экзистенциальный тип:

```
trait OrStringE:
  type A
  val value: A Either String
```

Если универсальный тип `OrStringA` представлял собой **функцию** от *любого* типа `A` в тип значения, представленный выражением, то экзистенциальный тип `OrStringE` описывает **пару** из *некого* типа `A` и значения типа, представленного выражением. Ввиду этого, иногда используют альтернативное обозначение для экзистенциальных в виде пары $\{\exists A,\ Expr\}$, где `Expr` – некое выражение, например, `A Either Strig`. Первый элемент такой пары называют *скрытым типом-свидетелем*, в том смысле, что его никто и не собирается искать, но само его наличие “свидетельствует” о существовании подходящего типа.

Трейт `OrStringE` демонстрирует механизм *зависимых (от пути) типов* в Scala. Такие типы реализуют механизм позднего связывания, когда тип значения проверяется *динамически*, во время выполнения с использованием мета-информации, на которую ссылается это значение. На этапе компиляции не получится выяснить, какого типа будет выражение `(a: OrStringE).value`.

Но, учитывая, что элемент экзистенциального типа представляет собой *пару* из типа и значения, то можно применить кодирование по Чёрчу, допускающее статическую типизацию (раннее связывание). В случае обычного произведения типов кодирование Чёрча даёт $PairAB = \forall X.\ (A \to B \to X) \to X$. Соответственно, для экзистенциального типа, построенного на выражении `Expr`, получается такое альтернативное представление:

$$\{\exists X,\ Expr\} = \forall Y.\ (\forall X.\ Expr \to Y) \to Y$$

На Scala можно таким образом записать универсальный и экзистенциальный типы, построенные на одном и том же выражении `Expr`:

```
type Universal   = [Expr[_]] =>> [X] =>> Expr[X] // Universal[F] ≡ F
type Existential = [Expr[_]] =>> [Y] => ([X] => Expr[X] => Y) => Y

type ListU = [A] =>> Universal[List][A] // List[A]
type ListE = Existential[List] // [Y] => ([X] => List[X] => Y) => Y

val listUInt: ListU[Int] = List(4, 2)
val listEInt: ListE = // {*Int, List[Int]}: {∃X, List[X]}
        [Y] => (continuation: [X] => List[X] => Y) => continuation(listUInt)

listEInt[String]([X] => (_: List[X]).mkString("; ")) // "4; 2"
```

В коде Scala лучше видны все “стрелочки” кодировки Чёрча. В таком представлении экзистенциальный тип представляет собой полиморфную функцию, принимающую в качестве аргумента другую полиморфную функцию-продолжение `continuation`. В определении значения `listEInt` этого типа в функцию-продолжение передаётся значение `listUInt` простого типа, полученного подстановкой в тип-выражение `List[_]` конкретного типа `Int`.

> В Scala 2 существовал специальный синтаксис для экзистенциальных типов:
> `type F = SomeClass[A] forSome { type A }`
> Ещё почитать про это можно тут: Existential Types in Scala
> В Scala 3 такой синтаксис решили упразднить, ссылаясь на сложности поддержки компилятором и то, что схожих результатов не сложно добиться с помощью других синтаксических конструкций.
> Ещё один способ записи экзистенциального типа: `a: List[_ <: SomeClass]` эквивалентен (ввиду ковариантности `List`) более простому варианту: `a: List[SomeClass]`.

В последней строчке представленного выше фрагмента кода в качестве продолжения подставлена функция,

Stackoverflow. What is an existential type?

Специальный полиморфизм

> В специальном полиморфизме не существует единого систематического способа определения типа результата по типу аргументов. Может существовать несколько правил ограниченного действия, которые сокращают количество случаев, но они сами по себе являются специальными как по объему, так и по содержанию. В эту категорию попадают все обычные арифметические операторы и функции. Более того, кажется, что автоматическая вставка передаточных функций компилирующей системой ограничивается этим классом.

Примеры полиморфизма: перегрузка функций

В любом случае, концепция классов типов заслуживает рассмотрения в данной статье хотя бы по той причине, что она представляет “обратную сторону” рассмотренного ранее полиморфизма универсального – *специальный полиморфизм*.

В сочетании с универсальным - классы типов

Классы типов в Scala

Неявности в Scala

Отличие от Haskell

Type classes in Scala

> In comparison, a type class in Haskell can **only have one instance**. In Scala, we can define an instance and pass it as a parameter explicitly (not relying on implicit resolution), which makes the usage less convenient, but may be useful.

Ключевое слово `derived`

Примеры классов типов

Реализуем ООП

Класс типа Set

Любой ли обобщённый тип можно интерпретировать, как описание класса типов? Рассмотрим такой фрагмент кода:

```
def isValid[A: Set] // Тип A должен принадлежать некому классу Set
  : A => Boolean    // Аннотация типа не обязательна
  = summon[Set[A]].contains
```

```
given validInts // Помещаем значение класса Set типа Int в контекст
  : Set[Int]
  = Set(2, 3, 5, 7, 11)
isValid(42)     // Класс Set для типа Int неявно найден в контексте
//isValid("42") // Ошибка! В контексте не найден класс Set для типа String!
```

Безотносительно того, что размещать в контексте неявное значение "слаботипизированного" типа `Set[Int]` достаточно безответственно, но, в целом, очевидно, что ответ на поставленный выше вопрос утвердительный.

Класс типа Monad

Класс типа Functor

Литература по классам типов:

- Хабр-статьи
  - Type classes в Scala
  - Классы типов в Scala (с небольшим обзором библиотеки cats)
- Статья от авторов Scala Type Classes as Objects and Implicits
- Typeclassery — A sure way of making generic programs context aware

Типы-контейнеры

Идея контейнера

Для обобщённых типов есть ещё одна любопытная трактовка

Эффекты

Ленивые контейнеры и Future/Try
Линейные типы в Scala как альтернатива конкурентности.
Список как эффект - недетерминированность

Tagless Final

Optimizing Tagless Final – saying farewell to Free

Ко-контейнеры и другие

Подкатегории типов (анонс)

Заключение

`trait TraverseK[Alg[_[_]]]`

https://stackoverflow.com/questions/72407103/problem-with-given-instances-writing-mtl-style-code-with-scala-cats