F [SCALA]

АЛЕКСЕЙ
ТРОИЦКИЙ

ВОРКШОП

TUPLE, ЕЩЕ TUPLEE И МЕТАПРОГРАММИРОВАНИЕ

Про формат

Про формат

- Я буду что-то рассказывать

Про формат

- Я буду что-то рассказывать
- Если вам что-то непонятно, то задавайте вопросы **сразу**

Про формат

- Я буду что-то рассказывать
- Если вам что-то непонятно, то задавайте вопросы **сразу** кроме вопроса “а зачем?”

Про формат

- Я буду что-то рассказывать
- Если вам что-то непонятно, то задавайте вопросы **сразу**
  кроме вопроса “а зачем?” (до определенного момента времени)

Про формат

- Я буду что-то рассказывать
- Если вам что-то непонятно, то задавайте вопросы **сразу**
  кроме вопроса “а зачем?” (до определенного момента времени)
- Я тоже буду задавать вам вопросы

Уличная магия

Уличная магия

```
val langs = magic("scala,clojure,rust")
println(s"Hello from F[${langs.scala}]!")

langs.rust // "rust"
// langs.python // not compiling
```

`magic - ???`

new class / object / trait

```
val langs = { // magic("scala,rust,clojure")
  class Impl:
    val scala: String = "scala"
    val rust: String = "rust"
    val clojure: String = "clojure"
  new Impl
}

langs.scala
langs.clojure
```

new class / object / trait

```
val langs = { // magic("scala,rust,clojure")
  class Impl:
    val scala: String = "scala"
    val rust: String = "rust"
    val clojure: String = "clojure"
  new Impl
}

langs.scala
langs.clojure
```

- Внутри макроса можно генерировать новые классы / объекты
- Но снаружи их новые поля не будут видны (`langs: Object`)

Selectable

```
val langs: scala.reflect.Selectable {
  def scala: String
  def rust: String
  def clojure: String
} = { // magic("scala,rust,clojure")
  new scala.reflect.Selectable:
    def scala: String = "scala"
    def rust: String = "rust"
    def clojure: String = "clojure"
}
langs.rust
// langs.python // not compiling
```

Selectable

```
val langs: scala.reflect.Selectable {
  def scala: String
  def rust: String
  def clojure: String
} = { // magic("scala,rust,clojure")
  new scala.reflect.Selectable:
    def scala: String = "scala"
    def rust: String = "rust"
    def clojure: String = "clojure"
}
langs.rust
// langs.python // not compiling
```

- Можно генерировать в компайл-тайме (создавать на лету)

Selectable

```
val langs: scala.reflect.Selectable {
  def scala: String
  def rust: String
  def clojure: String
} = { // magic("scala,rust,clojure")
  new scala.reflect.Selectable:
    def scala: String = "scala"
    def rust: String = "rust"
    def clojure: String = "clojure"
}
langs.rust
// langs.python // not compiling
```

- Можно генерировать в компайл-тайме (создавать на лету)
- Рефлексия / своя реализация `Selectable` через мапку

Selectable

```
val langs: scala.reflect.Selectable {
  def scala: String
  def rust: String
  def clojure: String
} = { // magic("scala,rust,clojure")
  new scala.reflect.Selectable:
    def scala: String = "scala"
    def rust: String = "rust"
    def clojure: String = "clojure"
}
langs.rust
// langs.python // not compiling
```

- Можно генерировать в компайл-тайме (создавать на лету)
- Рефлексия / своя реализация `Selectable` через мапку
- В макросе придется работать с AST и понадобится `@experimental` (как минимум в `3.7.4`)

NamedTuple

```
val langs = { // magic("scala,rust,clojure")
  (scala = "scala", rust = "rust", clojure = "clojure")
}

langs.scala
langs.rust
// langs.python // not compiling
```

NamedTuple

```
val langs = { // magic("scala,rust,clojure")
  (scala = "scala", rust = "rust", clojure = "clojure")
}

langs.scala
langs.rust
// langs.python // not compiling
```

- Легко генерировать в компайл-тайме (на лету)

NamedTuple

```
val langs = { // magic("scala,rust,clojure")
  (scala = "scala", rust = "rust", clojure = "clojure")
}

langs.scala
langs.rust
// langs.python // not compiling
```

- Легко генерировать в компайл-тайме (на лету)
- Не будет оверхеда в рантайме

NamedTuple

```
val langs = { // magic("scala,rust,clojure")
  (scala = "scala", rust = "rust", clojure = "clojure")
}

langs.scala
langs.rust
// langs.python // not compiling
```

- Легко генерировать в компайл-тайме (на лету)
- Не будет оверхеда в рантайме
- Начиная со `Scala 3.7` (с `3.5` в `experimental`)

План

- `Tuple`

План

- `Tuple`
- ~~еще tuplee~~ `NamedTuple`

План

- `Tuple`
- ~~еще tuplee~~ `NamedTuple`
- ~~метапрограммирование~~ Примеры

Tuple

```
val user: (Int, String) = (42, "Bob")
```

Tuple

```scala
val user: (Int, String) = (42, "Bob")


def greet(user: (Int, String)): String =
  val (_, name) = user
  s"Hello, $name!" // или user._2

greet(user) // Hello, Bob!
```

Tuple

```scala
val user: (Int, String) = (42, "Bob")


def greet(user: (Int, String)): String =
  val (_, name) = user
  s"Hello, $name!" // или user._2

greet(user) // Hello, Bob!
```

```scala
val ids: List[Int] = List(42, 43, 44)
val names: List[String] = List("Bob", "John", "Alice")

ids.zip(names): List[(Int, String)]
```

Tuple

```scala
val user: (Int, String) = (42, "Bob")


def greet(user: (Int, String)): String =
  val (_, name) = user
  s"Hello, $name!" // или user._2

greet(user) // Hello, Bob!
```

```scala
val ids: List[Int] = List(42, 43, 44)
val names: List[String] = List("Bob", "John", "Alice")

ids.zip(names): List[(Int, String)]
val (even, odd) = ids.partition(x ⇒ x % 2 == 0) // (List(42, 44), List(43))
```

Scala3 Tuple

Кортежи стали действительно гетерогенными списками!

```
val user: Int *: String *: EmptyTuple = 42 *: "Bob" *: EmptyTuple
// val user: (Int, String) = (42, "Bob")
```

Scala3 Tuple

Кортежи стали действительно гетерогенными списками!

```
val user: Int *: String *: EmptyTuple = 42 *: "Bob" *: EmptyTuple
// val user: (Int, String) = (42, "Bob")


val userInfo: Boolean *: String *: Option[Int] *: EmptyTuple = (true, "Russia", None)

val userFull: Int *: String *: Boolean *: String *: Option[Int] *: EmptyTuple =
  user ++ userInfo
```

Scala3 Tuple

Кортежи стали действительно гетерогенными списками!

```
val user: Int *: String *: EmptyTuple = 42 *: "Bob" *: EmptyTuple
// val user: (Int, String) = (42, "Bob")


val userInfo: Boolean *: String *: Option[Int] *: EmptyTuple = (true, "Russia", None)

val userFull: Int *: String *: Boolean *: String *: Option[Int] *: EmptyTuple =
  user ++ userInfo


userFull.size // 5
user.zip(userInfo) // ((42, true), ("Bob", "Russia"))
```

Scala3 Tuple

Кортежи стали действительно гетерогенными списками!

```
val user: Int *: String *: EmptyTuple = 42 *: "Bob" *: EmptyTuple
// val user: (Int, String) = (42, "Bob")


val userInfo: Boolean *: String *: Option[Int] *: EmptyTuple = (true, "Russia", None)

val userFull: Int *: String *: Boolean *: String *: Option[Int] *: EmptyTuple =
  user ++ userInfo


userFull.size // 5
user.zip(userInfo) // ((42, true), ("Bob", "Russia"))
userFull.map([A] => (x: A) ⇒ Some(x))
// (Some(42),Some(Bob),Some(true),Some(Russia),Some(None))
```

Пример: деривации

```
trait Eq[A]:
  def eqv(l: A, r: A): Boolean
```

Хотим генерировать инстансы `Eq` для `case class`:

- в компайл-тайме
- на основе структуры `case class`
- в соответствии с инстансами `Eq` для типов полей

Пример: деривации

```
trait Eq[A]:
  def eqv(l: A, r: A): Boolean
```

Хотим генерировать инстансы `Eq` для `case class`:

- в компайл-тайме
- на основе структуры `case class`
- в соответствии с инстансами `Eq` для типов полей

```
case class Employee(name: String, age: Int, manager: Boolean)

object Employee:
  given Eq[Employee] = new Eq[Employee]: // Eq.derived[Employee]
    def eqv(l: Employee, r: Employee): Boolean =
      summon[Eq[String]].eqv(l.name, r.name) && summon[Eq[Int]].eqv(l.age, r.age) &&
      summon[Eq[Boolean]].eqv(l.manager, r.manager)
```

Пример: деривации

```scala
trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  //  инстансы для базовых типов
  //   ...
  // хотим:
  def derived[A]/*( ... )*/: Eq[A] = ???
```

Пример: деривации

```
trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  //  инстансы для базовых типов
  //   ...
  // хотим:
  def derived[A]/*( ... )*/: Eq[A] = ???
```

```
case class Employee(name: String, age: Int, manager: Boolean) derives Eq
```

Пример: деривации

```
trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  //  инстансы для базовых типов
  //   ...
  // хотим:
  def derived[A]/*( ... )*/: Eq[A] = ???
```

```
case class Employee(name: String, age: Int, manager: Boolean) derives Eq
case class IceCream(name: String, numCherries: Int, inCone: Boolean) derives Eq
```

Пример: деривации

```
trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  //  инстансы для базовых типов
  //   ...
  // хотим:
  def derived[A]/*( ... )*/: Eq[A] = ???
```

```
case class Employee(name: String, age: Int, manager: Boolean) derives Eq
case class IceCream(name: String, numCherries: Int, inCone: Boolean) derives Eq

// summon[Eq[(String, Int, Boolean)]]
```

Не интересуют имена полей класса, имя самого класса, нужны только **типы** полей

Пример: деривации

```
trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  // ...
  given [T <: Tuple]: Eq[T] = ???
```

Пример: деривации

```
object Eq:
  // ...
  given Eq[EmptyTuple]:
    def eqv(l: EmptyTuple, r: EmptyTuple): Boolean = true

  given [A1: Eq as e1] => Eq[A1 *: EmptyTuple]:
    def eqv(l: A1 *: EmptyTuple, r: A1 *: EmptyTuple): Boolean =
      (l, r) match
        case (l1 *: EmptyTuple, r1 *: EmptyTuple) ⇒
          e1.eqv(l1, r1)

  given [A1: Eq as e1, A2: Eq as e2] => Eq[(A1, A2)]:
    def eqv(l: (A1, A2), r: (A1, A2)): Boolean =
      (l, r) match
        case (l1 *: l2 *: EmptyTuple, r1 *: r2 *: EmptyTuple) ⇒
          e1.eqv(l1, r1) && e2.eqv(l2, r2)
  // up to TupleN
```

Пример: деривации

```
object Eq:
  // ...
  given Eq[EmptyTuple]:
    def eqv(l: EmptyTuple, r: EmptyTuple): Boolean = true

  given [A1: Eq as e1] => Eq[A1 *: EmptyTuple]:
    def eqv(l: A1 *: EmptyTuple, r: A1 *: EmptyTuple): Boolean =
      (l, r) match
        case (l1 *: EmptyTuple, r1 *: EmptyTuple) ⇒
          e1.eqv(l1, r1)

  given [A1: Eq as e1, A2: Eq as e2] => Eq[(A1, A2)]:
    def eqv(l: (A1, A2), r: (A1, A2)): Boolean =
      (l, r) match
        case (l1 *: l2 *: EmptyTuple, r1 *: r2 *: EmptyTuple) ⇒
          e1.eqv(l1, r1) && e2.eqv(l2, r2)
  // up to TupleN
```

Можно ли обобщить?

Пример: деривации

```
trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  // ...
  given Eq[EmptyTuple]: // base
    def eqv(l: EmptyTuple, r: EmptyTuple): Boolean = true

  given [H: Eq as heq, T <: Tuple: Eq as teq] => Eq[H *: T]: // step
    def eqv(l: H *: T, r: H *: T): Boolean =
      heq.eqv(l.head, r.head) &&
      teq.eqv(l.tail, r.tail)
```

Пример: деривации

```
trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  // ...
  def derived[A, T <: Tuple](using /* T is tuple of fileds of T,*/ Eq[T]): Eq[A] = ???
```

Пример: деривации

```
import scala.deriving.Mirror

trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  // ...
  def derived[A <: Product](using ev: Mirror.ProductOf[A], inst: Eq[ev.MirroredElemTypes]):
Eq[A] = new Eq[A]:
    def eqv(l: A, r: A): Boolean =
      inst.eqv(Tuple.fromProductTyped(l), Tuple.fromProductTyped(r))
```

Пример: деривации

```
import scala.deriving.Mirror

trait Eq[A]:
  def eqv(l: A, r: A): Boolean

object Eq:
  // ...
  def derived[A <: Product](using ev: Mirror.ProductOf[A], inst: Eq[ev.MirroredElemTypes]):
Eq[A] = new Eq[A]:
    def eqv(l: A, r: A): Boolean =
      inst.eqv(Tuple.fromProductTyped(l), Tuple.fromProductTyped(r))
```

```
case class Employee(name: String, age: Int, manager: Boolean) derives Eq
```

Mirror

Представление информации о типе данных:

- на уровне типов
- генерируется компилятором
- для типов сумм (enum / sealed trait) и произведений (case class / object)

Mirror

Представление информации о типе данных:

- на уровне типов
- генерируется компилятором
- для типов сумм (enum / sealed trait) и произведений (case class / object)

```
case class Employee(name: String, age: Int, manager: Boolean)

type EmployeeMirror = Mirror.Product {
  type MirroredType = Employee
  type MirroredMonoType = Employee
  type MirroredLabel = "Employee"
  type MirroredElemTypes = (String, Int, Boolean)
  type MirroredElemLabels = ("name", "age", "manager")
}

val employeeMirror: EmployeeMirror = summon[Mirror.Of[Employee]]
```

Операции над Tuple

```
val user: (Int, String) = (42, "Bob")
val userInfo: (Boolean, String, Option[Int]) = (true, "Russia", None)

user ++ userInfo: (Int, String, Boolean, String, Option[Int])
```

Операции над Tuple

```scala
val user: (Int, String) = (42, "Bob")
val userInfo: (Boolean, String, Option[Int]) = (true, "Russia", None)

user ++ userInfo: (Int, String, Boolean, String, Option[Int])
```

```scala
sealed trait Tuple extends Product {
  // ...
  inline def ++ [This >: this.type <: Tuple](that: Tuple) // : ???
}
```

Операции над Tuple

```scala
val user: (Int, String) = (42, "Bob")
val userInfo: (Boolean, String, Option[Int]) = (true, "Russia", None)

user ++ userInfo: (Int, String, Boolean, String, Option[Int])
```

```scala
sealed trait Tuple extends Product {
  // ...
  inline def ++ [This >: this.type <: Tuple](that: Tuple): Tuple.Concat[This, that.type]
}

object Tuple:
  type Concat[X <: Tuple, +Y <: Tuple] <: Tuple = X match {
    case EmptyTuple ⇒ Y
    case x1 *: xs1 ⇒ x1 *: Concat[xs1, Y]
  }
```

Операции над Tuple

```scala
val userInfo: (Boolean, String, Option[Int]) = (true, "Russia", None)

userInfo.map([A] => (x: A) ⇒ Some(x)) // (Some(true), Some("Russia"), Some(None))
```

```scala
sealed trait Tuple extends Product {
  // ...
  inline def ++ [This >: this.type <: Tuple](that: Tuple): Map[this.type, F]
}

object Tuple:
  type Map[Tup <: Tuple, F[_ <: Union[Tup]]] <: Tuple = Tup match {
    case EmptyTuple ⇒ EmptyTuple
    case h *: t ⇒ F[h] *: Map[t, F]
  }
```

Подтипирование

Подтипирование

`Tuple` ковариантны:

```
@showAsInfix
sealed abstract class *:[+H, +T <: Tuple] extends NonEmptyTuple
```

```
summon[
  (1, String, Boolean) <:< (Int, String, Any)
]
```

Итоги: `Tuple`

Итоги: `Tuple`

- Много где используются в стандартной библиотеке и других для обобщений (работа с коллекциями, эндпоинты тапира)

Итоги: `Tuple`

- Много где используются в стандартной библиотеке и других для обобщений (работа с коллекциями, эндпоинты тапира)
- С помощью `Tuple`, `given`-ов, `Mirror` можно писать деривации для типов-произведений даже без `inline`

Итоги: `Tuple`

- Много где используются в стандартной библиотеке и других для обобщений (работа с коллекциями, эндпоинты тапира)
- С помощью `Tuple`, `given`-ов, `Mirror` можно писать деривации для типов-произведений даже без `inline`
- Типы операций над `Tuple` хорошо типизированы с помощью `match` типов

Итоги: `Tuple`

- Много где используются в стандартной библиотеке и других для обобщений (работа с коллекциями, эндпоинты тапира)
- С помощью `Tuple`, `given`-ов, `Mirror` можно писать деривации для типов-произведений даже без `inline`
- Типы операций над `Tuple` хорошо типизированы с помощью `match` типов
- Благодаря тому, что создаются на лету, их удобно использовать для метапрограммирования

NamedTuple

NamedTuple: мотивация

```
def partition[T](seq: Seq[T])(f: T ⇒ Boolean): (Seq[T], Seq[T]) = ...
```

NamedTuple: мотивация

```
def partition[T](seq: Seq[T])(f: T ⇒ Boolean): (Seq[T], Seq[T]) = ...
```

- Сигнатура метода не говорит о том, что где лежит (успехи, ошибки)

`NamedTuple`: мотивация

```
def partition[T](seq: Seq[T])(f: T ⇒ Boolean): (Seq[T], Seq[T]) = ...
```

- Сигнатура метода не говорит о том, что где лежит (успехи, ошибки)
- Доступ по `_1`, `_2` плохо читается

NamedTuple: мотивация

```
case class Result[T](matches: Seq[T], mismatches: Seq[T])

def partition[T](seq: Seq[T])(f: T ⇒ Boolean): Result[T] = ???
```

`NamedTuple`: мотивация

```
case class Result[T](matches: Seq[T], mismatches: Seq[T])

def partition[T](seq: Seq[T])(f: T ⇒ Boolean): Result[T] = ???
```

- Тяжелее в рантайме
- Больше кода

NamedTuple: мотивация

```
def partition[T](seq: Seq[T])(f: T ⇒ Boolean): (matches: Seq[T], mismatches: Seq[T]) = ...
```

`NamedTuple`: мотивация

```
def partition[T](seq: Seq[T])(f: T ⇒ Boolean): (matches: Seq[T], mismatches: Seq[T]) = ...
```

- Нет оверхеда в рантайме

NamedTuple: мотивация

```
case class Flags(
  flag1: Boolean,
  flag2: Boolean,
  flag3: Boolean,
  flag4: Boolean,
  /*  ... 100500 других флагов */
)
```

NamedTuple: мотивация

```
case class Flags(
  flag1: Boolean,
  flag2: Boolean,
  flag3: Boolean,
  flag4: Boolean,
  /*  ... 100500 других флагов */
)

flags match
  case Flags(_, false, _, _, _, true,  ... ) ⇒ ...
  case  ...                                  ⇒ ...
```

NamedTuple: мотивация

```
case class Flags(
  flag1: Boolean,
  flag2: Boolean,
  flag3: Boolean,
  flag4: Boolean,
  /*  ... 100500 других флагов */
)

flags match
  case Flags(_, false, _, _, _, true,  ... ) ⇒ ...
  case  ...                                  ⇒ ...
```

- Плохо читается, непонятно к каким флагам относятся значения
- Каждый раз при добавлении нового поля придется менять шаблон

NamedTuple: мотивация

```
case class Flags(
  flag1: Boolean,
  flag2: Boolean,
  flag3: Boolean,
  flag4: Boolean,
  /*  ... 100500 других флагов */
)

flags match
  case Flags(flag2 = false, flag4 = true)  ⇒ ...
  case  ...                                ⇒ ...
```

NamedTuple: мотивация

```
case class Flags(
  flag1: Boolean,
  flag2: Boolean,
  flag3: Boolean,
  flag4: Boolean,
  /*  ... 100500 других флагов */
)

flags match
  case Flags(flag2 = false, flag4 = true)  ⇒ ...
  case  ...                                ⇒ ...
```

Unlocked: именованные аргументы в паттерн матчинге (ишью 2012-го года)

NamedTuple

```
type Person = (name: String, age: Int)
val p = (name = "Bob", age = 42)
p.name
```

NamedTuple

```
type Person = (name: String, age: Int)
val p = (name = "Bob", age = 42)
p.name
```

```
type Person = NamedTuple.NamedTuple[("name", "age"), (String, Int)]
val p = NamedTuple.build[("name", "age")]()(("Bob", 42))
NamedTuple.apply[("name", "age"), (String, Int)](p)(0)
```

NamedTuple: реализация

```
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

NamedTuple: реализация

```
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

- Имена полей есть только в компайл-тайме, в рантайме нет оверхеда

NamedTuple: реализация

NamedTuple: реализация

```
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

- Имена полей есть только в компайл-тайме, в рантайме нет оверхеда
- `unnamed` → `named` через подтипирование
  `name` → `unnamed` через специальную конверсию

NamedTuple: реализация

```scala
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

- Имена полей есть только в компайл-тайме, в рантайме нет оверхеда
- `unnamed` → `named` через подтипирование
  `name` → `unnamed` через специальную конверсию

```scala
val t1: (age: Int, status: Boolean) = (42, true) // subtyping
val t2: (Int, Boolean) = (age = 42, status = true) // conversion
```

NamedTuple: реализация

```scala
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

- Имена полей есть только в компайл-тайме, в рантайме нет оверхеда
- `unnamed` → `named` через подтипирование
  `name` → `unnamed` через специальную конверсию

```scala
val t1: (age: Int, status: Boolean) = (42, true) // subtyping
val t2: (Int, Boolean) = (age = 42, status = true) // conversion
```

- Благодаря ковариантности:
  `(age: Int, name: String) <:< (age: Int | Null, name: Any)`

NamedTuple: Операции

```
type Names[X <: AnyNamedTuple] <: Tuple = X match
  case NamedTuple[n, _] ⇒ n
```

NamedTuple: Операции

```
type Names[X <: AnyNamedTuple] <: Tuple = X match
  case NamedTuple[n, _] ⇒ n

type DropNames[NT <: AnyNamedTuple] <: Tuple = NT match
  case NamedTuple[_, x] ⇒ x
```

NamedTuple: Операции

```
type Names[X <: AnyNamedTuple] <: Tuple = X match
  case NamedTuple[n, _] ⇒ n

type DropNames[NT <: AnyNamedTuple] <: Tuple = NT match
  case NamedTuple[_, x] ⇒ x

type Map[X <: AnyNamedTuple, F[_ <: Tuple.Union[DropNames[X]]]] =
    NamedTuple[Names[X], Tuple.Map[DropNames[X], F]]
```

NamedTuple: Операции

```
type Names[X <: AnyNamedTuple] <: Tuple = X match
  case NamedTuple[n, _] ⇒ n

type DropNames[NT <: AnyNamedTuple] <: Tuple = NT match
  case NamedTuple[_, x] ⇒ x

type Map[X <: AnyNamedTuple, F[_ <: Tuple.Union[DropNames[X]]]] =
    NamedTuple[Names[X], Tuple.Map[DropNames[X], F]]

type Concat[X <: AnyNamedTuple, Y <: AnyNamedTuple] =
    NamedTuple[Tuple.Concat[Names[X], Names[Y]], Tuple.Concat[DropNames[X], DropNames[Y]]]
```

NamedTuple: Операции

```
type Names[X <: AnyNamedTuple] <: Tuple = X match
  case NamedTuple[n, _] ⇒ n

type DropNames[NT <: AnyNamedTuple] <: Tuple = NT match
  case NamedTuple[_, x] ⇒ x

type Map[X <: AnyNamedTuple, F[_ <: Tuple.Union[DropNames[X]]]] =
    NamedTuple[Names[X], Tuple.Map[DropNames[X], F]]

type Concat[X <: AnyNamedTuple, Y <: AnyNamedTuple] =
    NamedTuple[Tuple.Concat[Names[X], Names[Y]], Tuple.Concat[DropNames[X], DropNames[Y]]]


extension [N <: Tuple, V <: Tuple](x: NamedTuple[N, V]):
  inline def map[F[_]](f: [t] ⇒ t ⇒ F[t]): Map[NamedTuple[N, V], F] =
    x.toTuple.map[F](f)

  inline def ++ [N2 <: Tuple, V2 <: Tuple](
    that: NamedTuple[N2, V2]
  )(using Tuple.Disjoint[N, N2] =:= true): Concat[NamedTuple[N, V], NamedTuple[N2, V2]]
    = x.toTuple ++ that.toTuple
```

NamedTuple: реализация

```scala
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

Какие минусы в такой реализации вы видите?

NamedTuple: реализация

```scala
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

Какие минусы в такой реализации вы видите?

- Нет никакой гарантии консистентности N и V

NamedTuple: реализация

```scala
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

Какие минусы в такой реализации вы видите?

- Нет никакой гарантии консистентности `N` и `V`
- `opaque type` и все что с ним связано (паттерн матчинг, матчинг по типу)

NamedTuple: реализация

```scala
package scala

object NamedTuple:
  opaque type NamedTuple[N <: Tuple, +V <: Tuple] >: V <: AnyNamedTuple = V

  opaque type AnyNamedTuple = Any
```

Какие минусы в такой реализации вы видите?

- Нет никакой гарантии консистентности `N` и `V`
- `opaque type` и все что с ним связано (паттерн матчинг, матчинг по типу)
- Мало информации в `AnyNamedTuple`
  (`AnyNamedTuple <: Tuple` или `AnyNamedTuple <: Product`)

Минусы реализации

Можно собирать сломанные кортежи:

```
import NamedTuple.NamedTuple

val broken: NamedTuple["x" *: EmptyTuple, EmptyTuple] = EmptyTuple
```

Минусы реализации

Можно собирать сломанные кортежи:

```
import NamedTuple.NamedTuple

val broken: NamedTuple["x" *: EmptyTuple, EmptyTuple] = EmptyTuple
val person = (name = "Bob", age = 42)

(broken ++ person).name: Int // :(
```

Минусы реализации

Для `Tuple` компилятор проверяет exhaustiveness:

```
val f: (Int, Boolean) = (42, true)
f match  // match may not be exhaustive warn
  case (1, y) ⇒ ()
```

Минусы реализации

Для `Tuple` компилятор проверяет exhaustiveness:

```
val f: (Int, Boolean) = (42, true)
f match  // match may not be exhaustive warn
  case (1, y) ⇒ ()
```

А для `NamedTuple` не проверяет :(

```
val f: (Int, Boolean) = (x = 42, y = true)
f match  // no warn
  case (1, y) ⇒ ()
```

Минусы реализации

Для `Tuple` компилятор проверяет exhaustiveness:

```
val f: (Int, Boolean) = (42, true)
f match  // match may not be exhaustive warn
  case (1, y) ⇒ ()
```

А для `NamedTuple` не проверяет :(

```
val f: (Int, Boolean) = (x = 42, y = true)
f match  // no warn
  case (1, y) ⇒ ()
```

https://github.com/scala/scala3/issues/23621

`NamedTuple` и патерн матчинг

Как `NamedTuple` помогают с именованным паттерн матчингом?

```
case class Flags(
  flag1: Boolean,
  flag2: Boolean,
  flag3: Boolean,
  flag4: Boolean,
  /*  ... 100500 других флагов */
)

flags match
  case Flags(flag2 = false, flag4 = true)  ⇒ ...
  case  ...                                ⇒ ...
```

unapply

- `unapply` может возвращать `NamedTuple`
- для `case class` компилятор генерирует `unapply` с именованными полями

unapply

- `unapply` может возвращать `NamedTuple`
- для `case class` компилятор генерирует `unapply` с именованными полями

```
object Flags:
  def unapply(f: Flags): (
    flag1: Boolean,
    flag2: Boolean,
    flag3: Boolean,
    flag4: Boolean
    /*, ... */
  ) =
    (f.flag1, f.flag2, f.flag3, f.flag4/*, ... */)
```

NamedTuple.From

```
case class Flags(
  flag1: Boolean,
  flag2: Boolean,
  flag3: Boolean,
  flag4: Boolean,
  /*  ... 100500 других флагов */
)

summon[
  NamedTuple.From[Flags] =:=
    (flag1: Boolean, flag2: Boolean, flag3: Boolean, flag4: Boolean)
]
```

NamedTuple и Selectable

```
trait Selectable:
  type Fields <: NamedTuple.AnyNamedTuple
```

NamedTuple и Selectable

```
trait Selectable:
  type Fields <: NamedTuple.AnyNamedTuple
```

```
class Langs[T <: Tuple](langs: List[String]) extends Selectable:
  type Fields = NamedTuple[T, Tuple.Map[T, [x] =>> String]]
  def selectDynamic(name: String): String = name

val langs = { // magic("scala,rust,clojure")
  new Langs[("scala","rust","clojure")](List("scala","rust","clojure"))
}

langs.rust
langs.clojure
// langs.python // not compiling
```

NamedTuple и Selectable

При этом используя Fields в Selectable теряется подтипирование:

```
Selectable { def name: String; def age: Int }  <: Selectable { def name: String }
```

NamedTuple и Selectable

При этом используя `Fields` в `Selectable` теряется подтипирование:

```
Selectable { def name: String; def age: Int }  <: Selectable { def name: String }


Selectable { type Fields = (name: String, age: Int) }
     не подтип
Selectable { type Fields = (name: String) }
```

Итоги

- `NamedTuple` можно создавать на лету как `Tuple`, но с именами полей
- По сравнению с `Selectable`:
  - в случае `scala.reflect.Selectable` не будет оверхеда в рантайме
  - `NamedTuple` имеет строгий порядок полей
  - нету структурного подтипирования

Итоги

- `NamedTuple` можно создавать на лету как `Tuple`, но с именами полей
- По сравнению с `Selectable`:
  - в случае `scala.reflect.Selectable` не будет оверхеда в рантайме
  - `NamedTuple` имеет строгий порядок полей
  - нету структурного подтипирования
- Дружит с другими конструкциями языка (`unapply`, `Selectable`)

Примеры

NamedTuple: деривации

NamedTuple: деривации

```
trait Log[A]:
  def log(a: A): String
```

Хотим генерировать инстансы `Log` для `case class` (в компайл-тайме, на основе структуры, в соответствии с инстансами `Log` для полей):

NamedTuple: деривации

```
trait Log[A]:
  def log(a: A): String
```

Хотим генерировать инстансы Log для `case class` (в компайл-тайме, на основе структуры, в соответствии с инстансами Log для полей):

```
case class Employee(name: String, age: Int, manager: Boolean)

object Employee:
  given Log[Employee] = new Log[Employee]: // Log.derived[Employee]
    def log(e: Employee): String =
      s"{name=${summon[Log[String]].log(e.name)}, age=${summon[Log[Int]].log(e.age)},
manager=${summon[Log[Boolean]].log(e.manager)}}"

summon[Log[Employee]].log(Employee("Bob", 42, false)) // {name=Bob, age=42, manager=false}
```

NamedTuple: деривации

```
// object Log

given Log[NamedTuple[EmptyTuple, EmptyTuple]]:
  def log(a: NamedTuple[EmptyTuple, EmptyTuple]): String = ""

given [
  N <: Singleton & String, V: Log,
  Ns <: Tuple, Vs <: Tuple
](using l: Log[NamedTuple[Ns, Vs]], v: ValueOf[N]):
  Log[NamedTuple[N *: Ns, V *: Vs]] = new:
  def log(a: NamedTuple[N *: Ns, V *: Vs]): String =
    s"${v.value}=${summon[Log[V]].log(a.head)}" +
      (if (a.tail.size == 0) "" else s", ${l.log(a.tail)}")
```

NamedTuple: деривации

```
// object Log

given Log[NamedTuple[EmptyTuple, EmptyTuple]]:
  def log(a: NamedTuple[EmptyTuple, EmptyTuple]): String = ""

given [
  N <: Singleton & String, V: Log,
  Ns <: Tuple, Vs <: Tuple
](using l: Log[NamedTuple[Ns, Vs]], v: ValueOf[N]):
  Log[NamedTuple[N *: Ns, V *: Vs]] = new:
  def log(a: NamedTuple[N *: Ns, V *: Vs]): String =
    s"${v.value}=${summon[Log[V]].log(a.head)}" +
      (if (a.tail.size == 0) "" else s", ${l.log(a.tail)}")


def derived[A <: Product](using inst: Log[NamedTuple.From[A]]): Log[A] = new Log[A]:
  def log(a: A): String =
    "{" + inst.log(Tuple.fromProduct(a).asInstanceOf[NamedTuple.From[A]]) + "}"
```

Пример: частичный доступ

Когда нужна только часть данных (например, подмножество фича флагов):

```
case class Flags(
  flag1: Flag1,
  flag2: Flag2,
  flag3: Flag3,
  flag4: Flag4,
  /*  ...  */
)
```

Пример: частичный доступ

Когда нужна только часть данных (например, подмножество фича флагов):

```
case class Flags(
  flag1: Flag1,
  flag2: Flag2,
  flag3: Flag3,
  flag4: Flag4,
  /*  ...  */
)


Api.partially[("flag2", "flag4")]: (flag2: Flag2, flag4: Flag4)
// Api.partially[("flag2", "notExists")] // not compiling
```

Пример: частичный доступ

Когда нужна только часть данных (например, подмножество фича флагов):

```
case class Flags(
  flag1: Flag1,
  flag2: Flag2,
  flag3: Flag3,
  flag4: Flag4,
  /*  ...  */
)


Api.partially[("flag2", "flag4")]: (flag2: Flag2, flag4: Flag4)
// Api.partially[("flag2", "notExists")] // not compiling
```

```
def partially[T <: Tuple](
  using ContainsAll[flags.MirroredElemLabels, T] =:= true
): FieldsOf[flags.MirroredElemLabels, flags.MirroredElemTypes, T] = ???
```

Higher Kinded Data

```
case class Employee(
  name: String,
  age: Int,
  country: Option[String]
)
```

Higher Kinded Data

```
case class Employee(
  name: String,
  age: Int,
  country: Option[String]
)
```

Заполнение формочек:

```
case class EmployeeApplication(
  name: Option[String],
  age: Option[Int],
  country: Option[Option[String]]
)
```

Higher Kinded Data

```
case class Employee(
  name: String,
  age: Int,
  country: Option[String]
)
```

Заполнение формочек:

```
case class EmployeeApplication(
  name: Option[String],
  age: Option[Int],
  country: Option[Option[String]]
)
```

Варианты для выбора:

```
case class EmployeeSuggestions(
    name: List[String],
    age: List[Int],
    country: List[Option[String]]
)
```

Higher Kinded Data

```
case class Employee(
  name: String,
  age: Int,
  country: Option[String]
)
```

Заполнение формочек:

```
case class EmployeeApplication(
  name: Option[String],
  age: Option[Int],
  country: Option[Option[String]]
)
```

Варианты для выбора:

```
case class EmployeeSuggestions(
    name: List[String],
    age: List[Int],
    country: List[Option[String]]
)
```

Валидация:

```
case class EmployeeValidation(
  name: String ⇒ Either[String, String],
  age: String ⇒ Either[String, Int],
  country: String ⇒ Either[String, Option[String]]
)
```

Higher Kinded Data

```scala
case class EmployeeH[F[_]](
  name: F[String],
  age: F[Int],
  country: F[Option[String]]
)

type Employee            = EmployeeH[[x] ⟹ x]
type EmployeeApplication = EmployeeH[Option]
type EmployeeSuggestions = EmployeeH[List]
type EmployeeValidation  = EmployeeH[[x] ⟹ String ⇒ Either[String, x]]
```

Higher Kinded Data

```scala
case class EmployeeH[F[_]](
  name: F[String],
  age: F[Int],
  country: F[Option[String]]
)

type Employee            = EmployeeH[[x] =>> x]
type EmployeeApplication = EmployeeH[Option]
type EmployeeSuggestions = EmployeeH[List]
type EmployeeValidation  = EmployeeH[[x] =>> String => Either[String, x]]
```

```scala
case class Employee(name: String, age: Int, country: Option[String])

type HKD[A, F[_]] = NamedTuple.Map[NamedTuple.From[Employee], F]
type EmployeeApplication = HKD[Employee, Option]
type EmployeeSuggestions = HKD[Employee, List]
type EmployeeValidation  = HKD[Employee, [x] =>> String => Either[String, x]]
```

Пример: scalasql

ScalaSql - Scala ORM библиотека для типо-безопасного написания SQL запросов.

Пример: scalasql

ScalaSql - Scala ORM библиотека для типо-безопасного написания SQL запросов.

```
import scalasql.*, H2Dialect.*

case class Employee[F[_]](
  name: F[String],
  age: F[Int],
  country: F[Option[String]]
)

object Employee extends Table[Employee]
```

Пример: scalasql

ScalaSql - Scala ORM библиотека для типо-безопасного написания SQL запросов.

```
import scalasql.*, H2Dialect.*

case class Employee[F[_]](
  name: F[String],
  age: F[Int],
  country: F[Option[String]]
)

object Employee extends Table[Employee]
```

```
client.transaction: db ⇒
  val query = Employee.select.filter(_.name === "Bob")
```

Пример: scalasql

ScalaSql - Scala ORM библиотека для типо-безопасного написания SQL запросов.

```
import scalasql.*, H2Dialect.*

case class Employee[F[_]](
  name: F[String],
  age: F[Int],
  country: F[Option[String]]
)

object Employee extends Table[Employee]
```

```
client.transaction: db ⇒
  val query = Employee.select.filter(_.name === "Bob")
  db.renderSql(query) // SELECT employee0.name AS name, employee0.age AS age,
                      //        employee0.country AS country
                      // FROM employee employee0 WHERE (employee0.name = ?)
```

Пример: scalasql

ScalaSql - Scala ORM библиотека для типо-безопасного написания SQL запросов.

```
import scalasql.*, H2Dialect.*

case class Employee[F[_]](
  name: F[String],
  age: F[Int],
  country: F[Option[String]]
)

object Employee extends Table[Employee]
```

```
client.transaction: db ⇒
  val query = Employee.select.filter((x: Employee[Expr]) ⇒
    (x.name: Expr[String]) === "Bob"
  )
  db.run(query): Seq[Employee[[x] ⟹ x]]
```

Пример: scalasql

Теперь с синтаксисом без HKD!

Пример: scalasql

Теперь с синтаксисом без HKD!

```
import scalasql.simple.*, H2Dialect.*

case class Employee(name: String, age: Int, country: Option[String])

object Employee extends SimpleTable[Employee]
```

Пример: scalasql

Теперь с синтаксисом без HKD!

```
import scalasql.simple.*, H2Dialect.*

case class Employee(name: String, age: Int, country: Option[String])

object Employee extends SimpleTable[Employee]
```

```
client.transaction: db ⇒
  val query = Employee.select.filter(_.name === "Bob")
  db.run(query)
```

Пример: scalasql

Теперь с синтаксисом без HKD!

```
import scalasql.simple.*, H2Dialect.*

case class Employee(name: String, age: Int, country: Option[String])

object Employee extends SimpleTable[Employee]
```

```
import scalasql.namedtuples.SimpleTable.MapOver

client.transaction: db ⇒
  val query = Employee.select.filter(
    (x: MapOver[Employee, Expr]) ⇒ // ~ NamedTuple.Map[NamedTuple.From[Employee], Expr]
      (x.name: Expr[String]) === "Bob"
  )
  db.run(query): Seq[Employee]
```

Пример: scalasql

Теперь с синтаксисом без HKD!

```
import scalasql.simple.*, H2Dialect.*

case class Employee(name: String, age: Int, country: Option[String])

object Employee extends SimpleTable[Employee]
```

```
import scalasql.namedtuples.NamedTupleQueryable.given

client.transaction: db ⇒
  val query = Employee.select.filter(_.name === "Bob").map(c ⇒ (a = c.age, c = c.country))
  db.run(query): Seq[(a: Int, c: Option[String])]
```

Идея: regex

Найдите ошибку:

```
import scala.util.matching.Regex

val date: Regex = """(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})""".r

"2019-11-26" match
  case date(y, _, _, _) ⇒
    s"$y was a good year for Scala conferences in Russia"
```

Идея: regex

Найдите ошибку:

```
import scala.util.matching.Regex

val date: Regex = """(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})""".r

"2019-11-26" match
  case date(y, _, _, _) ⇒
    s"$y was a good year for Scala conferences in Russia"


// runtime error:
// Caused by: scala.MatchError: 2019-11-26 (of class java.lang.String)
```

Идея: regex

Найдите ошибку:

```
import scala.util.matching.Regex

val date: Regex = """(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})""".r

"2019-11-26" match
  case date(y, _, _, _) ⇒
    s"$y was a good year for Scala conferences in Russia"


// runtime error:
// Caused by: scala.MatchError: 2019-11-26 (of class java.lang.String)
```

Идея: если регулярное выражение известно на этапе компиляции, то известно число групп и ошибку можно отловить на этапе компиляции

Идея: regex

```
val date: Regex = """(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})""".r

"2019-11-26" match
  case date(year = y) ⇒
    s"$y was a good year for Scala conferences in Russia"
```

Идея: схемы

```
val langs = magic("scala,clojure,rust")
```

Идея: схемы

```
val schema = magic(
  """|{
     |  "type": "record",
     |  "name": "UserInfo",
     |  "fields": [
     |    {
     |      "name": "name",
     |      "type": "string"
     |    },
     |    {
     |      "name": "age",
     |      "type": "int"
     |    }
     |  ]
     |}""".stripMarginCompileTime
  )
```

Идея: схемы

```
val schema: Schema[(name: String, age: Int)] = magic(
  """|{
     |  "type": "record",
     |  "name": "UserInfo",
     |  "fields": [
     |    {
     |      "name": "name",
     |      "type": "string"
     |    },
     |    {
     |      "name": "age",
     |      "type": "int"
     |    }
     |  ]
     |}""".stripMarginInCompileTime
  )
```

Идея: схемы

```
val schema: Schema[(name: String, age: Int)] = magicFromFile("path/to/file")
```

Идея: схемы

```
val schema: Schema[(name: String, age: Int)] = magicFromFile("path/to/file")

val bytes = schema.serialize(name = "Bob", age = 42)
schema.deserialize(bytes) // Right((name = "Bob", age = "42"))
```

kentavro

- Spec first

kentavro

- Spec first
- Кодогенерация происходит на этапе компиляции (а не перед ней сторонними тулзами)

kentavro

- Spec first
- Кодогенерация происходит на этапе компиляции (а не перед ней сторонними тулзами)
- IDE-friendly

kentavro

- Spec first
- Кодогенерация происходит на этапе компиляции (а не перед ней сторонними тулзами)
- IDE-friendly
- Типобезопасное API ~~exhaustive паттерн матчинг~~ (wip)

kentavro

```
import kentavro.Avdl

val schema =
  Avdl.fromFile(rootDir + "/models/user.avdl")
```

```
schema User;

record User {
    string name;
    int age;
}
```

kentavro

```scala
import kentavro.Avdl

val schema: KSchema["User" ~ (name : String, age : Int)] =
  Avdl.fromFile(rootDir + "/models/user.avdl")
```

```
schema User;

record User {
    string name;
    int age;
}
```

kentavro

```scala
import kentavro.{Avdl, Named}

val schema: KSchema["User" ~ (name : String, age : Int)] =
  Avdl.fromFile(rootDir + "/models/user.avdl")

schema.serialize(Named("User", (name = "Bob", age = 42)))
```

```
schema User;

record User {
    string name;
    int age;
}
```

kentavro

```scala
import kentavro.Avdl

val schema: KSchema["User" ~ (name : String, age : Int)] {
  val User: NamedCompanion.Record["User", ...]
} =
  Avdl.fromFile(rootDir + "/models/user.avdl")

schema.serialize(schema.User("Bob", 42))
```

```
schema User;

record User {
    string name;
    int age;
}
```

kentavro

```scala
import kentavro.Avdl

val schema = Avdl.fromFile(rootDir + "/models/user.avdl")
```

```
namespace example.fscala;

schema User;

import idl "status.avdl";

record User {
  string name;
  int age;
  Status status;
}
```

```
namespace example.fscala;

enum Status {
    ACTIVE,
    INACTIVE,
    PENDING,
    SUSPENDED
}
```

kentavro

```scala
import kentavro.Avdl

val s: KSchema[
  "example.fscala.User" ~ (
    name : String,
    age : Int,
    status : "example.fscala.Status" ~ (
      "ACTIVE" | "INACTIVE" | "PENDING" | "SUSPENDED"
    )
  )]{
    val `example.fscala`: Namespace {
      val User: NamedCompanion.Record[...]
      val Status: NamedCompanion.Enum[...]
    }
  } = Avdl.fromFile(rootDir + "/models/user.avdl")
```

```
namespace example.fscala;

schema User;

import idl "status.avdl";

record User {
  string name;
  int age;
  Status status;
}
```

```
namespace example.fscala;

enum Status {
    ACTIVE,
    INACTIVE,
    PENDING,
    SUSPENDED
}
```

kentavro

```scala
import kentavro.Avdl

val schema = Avdl.fromFile(rootDir + "/models/user.avdl")
val ns = schema.`example.fscala`
val bytes = schema.serialize(
  ns.User("Bob", 42, ns.Status.ACTIVE)
)
```

```
namespace example.fscala;

schema User;

import idl "status.avdl";

record User {
  string name;
  int age;
  Status status;
}
```

```
namespace example.fscala;

enum Status {
    ACTIVE,
    INACTIVE,
    PENDING,
    SUSPENDED
}
```

kentavro

```scala
import kentavro.Avdl

val schema = Avdl.fromFile(rootDir + "/models/user.avdl")
val ns = schema.`example.fscala`
val bytes = schema.serialize(
  ns.User("Bob", 42, ns.Status.ACTIVE)
)

schema.deserialize(bytes).map { v =>
  v.value.status.value match // compiler checks exhaustiveness
    case ns.Status.ACTIVE.value | ns.Status.PENDING.value ⇒
      println(s"hi, ${v.value.name}!")
    case ns.Status.INACTIVE.value | ns.Status.SUSPENDED.value ⇒
      println(s"bye, ${v.value.name}!")
}
```

```
namespace example.fscala;

schema User;

import idl "status.avdl";

record User {
  string name;
  int age;
  Status status;
}
```

```
namespace example.fscala;

enum Status {
    ACTIVE,
    INACTIVE,
    PENDING,
    SUSPENDED
}
```

В полный рост

Что если проектировать домен целиком на `NamedTuple`?

В полный рост

Что если проектировать домен целиком на `NamedTuple`?

Как прикапывать инстансы?

В полный рост

Что если проектировать домен целиком на `NamedTuple`?

Как прикапывать инстансы?

```
type User = User.Type // User.Type <: (id: Long, name: Name, description: String)
object User extends Record[(
  id: Long,
  name: Name,
  description: String
)]:
  given Log[User] = Log.derivedNT
```

В полный рост

```
type User = User.Type
object User extends Record[(id: Long, name: Name, description: String)]:
  given Log[User] = Log.derivedNT

type Meta = Meta.Type
object Meta extends Record[(created_at: Instant, updated_at: Instant)]:
  given Log[Meta] = Log.derivedNT


type UserWithMeta = UserWithMeta.Type
object UserWithMeta extends Record.Of(
  (User.schema ++ Meta.schema).rename("name" → "lastName")
):
  given Log[UserWithMeta] = Log.derivedNT
  // не будет учитывать инстансы Log[User] и Log[Meta]
```

Итоги

Итоги

- `NamedTuple` может классно помочь в задачах, где надо описывать DSL-и и в других задачах около метапрограммирования

Итоги

- `NamedTuple` может классно помочь в задачах, где надо описывать DSL-и и в других задачах около метапрограммирования
- Если где-то очень хочется использовать HKD, посмотрите на `NamedTuple`

Итоги

- `NamedTuple` может классно помочь в задачах, где надо описывать DSL-и и в других задачах около метапрограммирования
- Если где-то очень хочется использовать HKD, посмотрите на `NamedTuple`
- В макросах можно запускать **любой код** на этапе компиляции

Итоги

- `NamedTuple` может классно помочь в задачах, где надо описывать DSL-и и в других задачах около метапрограммирования
- Если где-то очень хочется использовать HKD, посмотрите на `NamedTuple`
- В макросах можно запускать **любой код** на этапе компиляции (даже джавовый, даже читать из внешних источников).

Итоги

- `NamedTuple` может классно помочь в задачах, где надо описывать DSL-и и в других задачах около метапрограммирования
- Если где-то очень хочется использовать HKD, посмотрите на `NamedTuple`
- В макросах можно запускать **любой код** на этапе компиляции (даже джавовый, даже читать из внешних источников). Например, можно парсить спецификации

Bonus

Альтернативная вселенная: kyo-data

```
//> using dep io.getkyo::kyo-data:0.19.0

import kyo.*

val person =
  ("name" ~ "Bob") & ("age" ~ 42)

person.name
person.age
```

Bonus

Альтернативная вселенная: kyo-data

```
//> using dep io.getkyo::kyo-data:0.19.0

import kyo.*

val person: Record["name" ~ String & "age" ~ Int] =
  ("name" ~ "Bob") & ("age" ~ 42)

person.name
person.age
```

Bonus

Альтернативная вселенная: kyo-data

```
//> using dep io.getkyo::kyo-data:0.19.0

import kyo.*
val person: Record["name" ~ String & "age" ~ Int] =
  ("name" ~ "Bob") & ("age" ~ 42)

val info: Record["status" ~ Boolean] =
  ("status" ~ true)

person & info: Record["name" ~ String & "age" ~ Int & "status" ~ Boolean]
```

Bonus

Альтернативная вселенная: kyo-data

```
//> using dep io.getkyo::kyo-data:0.19.0

import kyo.*
val person: Record["name" ~ String & "age" ~ Int] =
  ("name" ~ "Bob") & ("age" ~ 42)

val info: Record["status" ~ Boolean] =
  ("status" ~ true)

person & info: Record["name" ~ String & "age" ~ Int & "status" ~ Boolean]
```

```
summon[
  Record["name" ~ String & "age" ~ Int] <:< Record["age" ~ Int]
]
```

Bonus

```
final class Record[+Fields] private (
  val toMap: Map[Field[?, ?], Any]
) extends AnyVal with Dynamic:
    def selectDynamic[Name <: String & Singleton, Value](name: Name)(using
        ev: Fields <:< Name ~ Value,
        tag: Tag[Value]
    ): Value =
        toMap(Field(name, tag)).asInstanceOf[Value]

    def &[A](other: Record[A]): Record[Fields & A] =
        Record(toMap ++ other.toMap)

case class Field[Name <: String, Value](name: Name, tag: Tag[Value])
```

Bonus

```
final class Record[+Fields] private (
  val toMap: Map[Field[?, ?], Any]
) extends AnyVal with Dynamic:
    def selectDynamic[Name <: String & Singleton, Value](name: Name)(using
        ev: Fields <:< Name ~ Value,
        tag: Tag[Value]
    ): Value =
        toMap(Field(name, tag)).asInstanceOf[Value]

    def &[A](other: Record[A]): Record[Fields & A] =
        Record(toMap ++ other.toMap)

case class Field[Name <: String, Value](name: Name, tag: Tag[Value])
```

- в рантайме будет хуже чем `NamedTuple`

Bonus

Bonus

```
final class Record[+Fields] private (
  val toMap: Map[Field[?, ?], Any]
) extends AnyVal with Dynamic:
    def selectDynamic[Name <: String & Singleton, Value](name: Name)(using
        ev: Fields <:< Name ~ Value,
        tag: Tag[Value]
    ): Value =
        toMap(Field(name, tag)).asInstanceOf[Value]

    def &[A](other: Record[A]): Record[Fields & A] =
        Record(toMap ++ other.toMap)

case class Field[Name <: String, Value](name: Name, tag: Tag[Value])
```

- в рантайме будет хуже чем `NamedTuple`
- IDE не подсказывает

Bonus

Bonus

```
final class Record[+Fields] private (
  val toMap: Map[Field[?, ?], Any]
) extends AnyVal with Dynamic:
    def selectDynamic[Name <: String & Singleton, Value](name: Name)(using
        ev: Fields <:< Name ~ Value,
        tag: Tag[Value]
    ): Value =
        toMap(Field(name, tag)).asInstanceOf[Value]

    def &[A](other: Record[A]): Record[Fields & A] =
        Record(toMap ++ other.toMap)

case class Field[Name <: String, Value](name: Name, tag: Tag[Value])
```

- в рантайме будет хуже чем `NamedTuple`
- IDE не подсказывает
- но есть подтипирование!

Bonus

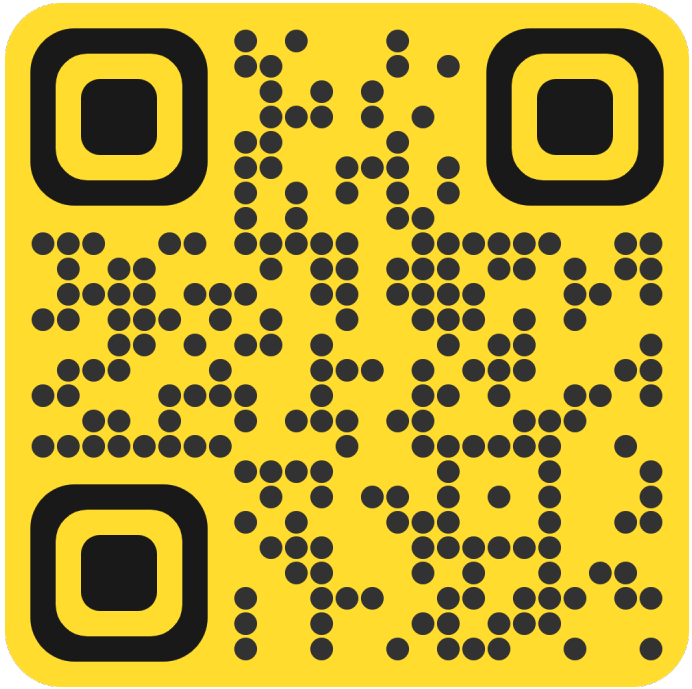
Bonus

```scala
final class Record[+Fields] private (
  val toMap: Map[Field[?, ?], Any]
) extends AnyVal with Dynamic:
    def selectDynamic[Name <: String & Singleton, Value](name: Name)(using
        ev: Fields <:< Name ~ Value,
        tag: Tag[Value]
    ): Value =
        toMap(Field(name, tag)).asInstanceOf[Value]

    def &[A](other: Record[A]): Record[Fields & A] =
        Record(toMap ++ other.toMap)

case class Field[Name <: String, Value](name: Name, tag: Tag[Value])
```

- в рантайме будет хуже чем `NamedTuple`
- IDE не подсказывает
- но есть подтипирование!
- их тоже можно `map`-ать!

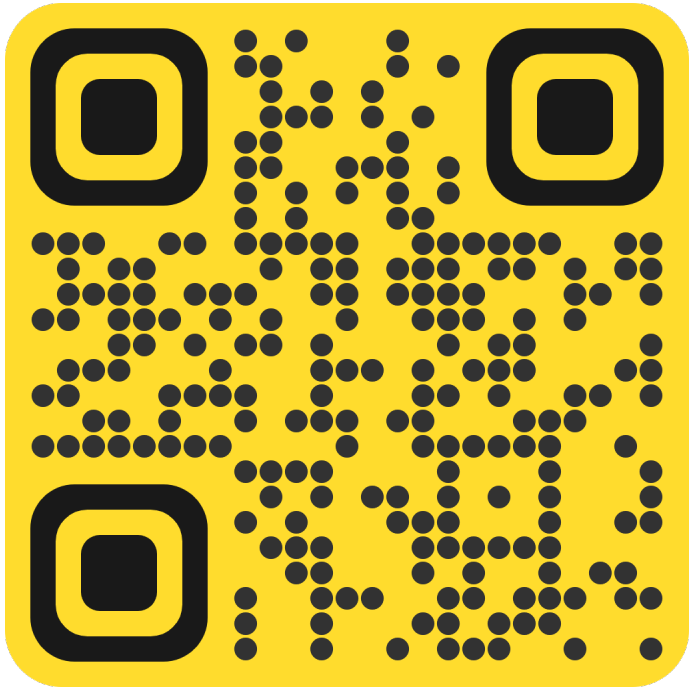
Ссылки

- статья про деривации на Tuple
- документация NamedTuple
- доклад про NamedTuple от Jamie Thompson
- доклад Олега Нижникова про HKD
- доклад Дениса Буздалова про Type Providers в Idris
- библиотеки ScalaSql, jing, kyo, kentavro



Ссылки

- статья про деривации на Tuple
- документация NamedTuple
- доклад про NamedTuple от Jamie Thompson
- доклад Олега Нижникова про HKD
- доклад Дениса Буздалова про Type Providers в Idris
- библиотеки ScalaSql, jing, kyo, kentavro



Спасибо!

made with love, `typst` and `scala-cli`

F [SCALA]

* ОЦЕНИТЕ ДОКЛАД