Przyczajona funkcja, ukryty parametr

czyli kung-fu w Scali

https://github.com/AVSystem/Wyklady_2016

Rzecz o implicitach w Scali

`implicits`, ponieważ:

- unikalny, charakterystyczny element języka
- fundamentalny dla programowania funkcyjnego (typelevel)
- podstawa kilku bardzo ważnych wzorców programowania w Scali
- względnie zaawansowany i skomplikowany mechanizm

Rzecz o implicitach w Scali

borkdude @borkdude · 15.03

@travisbrown @adelbertchang It's still magic to me

Travis Brown @travisbrown · 15.03

@borkdude @adelbertchang Fair enough! But if you restrict your use of Scala magic to a few principled applications, it's less overwhelming.

Słowo kluczowe `implicit`

- `val`, `lazy val`, `var`, `def`, `object`, lambda param
- *term members*
- lokalne lub jako składowe klas, obiektów, traitów
- również `class` - nieco inne znaczenie

Modyfikator `implicit` zezwala kompilatorowi użyć symbolu automatycznie w sytuacjach gdy jest on “potrzebny”.

Trzy oblicza `implicitów`

a właściwie dwa

- implicit parameters
- implicit conversions
- implicit views

implicit views ~ implicit conversions

Implicit parameters

- metody i konstruktory w Scali mogą przyjmować wiele *list argumentów*
- ostatnia *lista argumentów* może zostać oznaczona jako `implicit`
- programista może nie podawać niejawnych argumentów
- wówczas kompilator sam znajdzie odpowiednie wartości na podstawie typów parametrów

Implicit parameters

```
def weźDużoListParametrów(str: String, i: Int)
  (implicit ec: ExecutionContext, ss: SomeService): Unit = ???

implicit val ec: ExecutionContext = ???
implicit val ss: SomeService = ???

weźDużoListParametrów("lol", 42)
```

Implicit scope

- kompilator szuka implicitów wśród lokalnych i zaimportowanych symboli (local scope)
- jeśli implicit odpowiedniego typu nie zostanie znaleziony w lokalnym zakresie, kompilator przeszukuje tzw. *implicit scope* związany z typem wymaganego implicita

Implicit scope, c.d.

- *implicit scope* to zawartość *companion object'ów* wszystkich klas i traitów które "mają coś wspólnego" z typem poszukiwanego implicita
- type parameters, superclasses, enclosing classes
- *implicit scope* pozwala definiować implicity widoczne "globalnie" - nie wymagające importowania

Przekazywanie “kontekstów”

- ten sam obiekt, wielokrotnie przekazywany w obrębie tego samego fragmentu kodu
- `Future` i `ExecutionContext`
- istotne szczególnie w przypadku składni for-comprehensions

Przekazywanie “kontekstów”

```scala
def getUserIdByName(name: String): Future[Long] = ???
def getUserAge(id: Long): Future[Int] = ???

import scala.concurrent.ExecutionContext.Implicits.global

def getMessage(username: String): Future[String] = {
  getUserIdByName(username)
    .flatMap(id => getUserAge(id)
      .map(age => s"User $username (ID $id) is $age years old")
    )
}

def getMessage2(username: String): Future[String] =
  for {
    id <- getUserIdByName(username)
    age <- getUserAge(id)
  } yield s"User $username (ID $id) is $age years old"
```

Wstrzykiwanie zależności

- wymaga jedynie deklaracji zależności na każdym poziomie zagnieżdżenia kodu
- zależności są automatycznie przekazywane przez kompilator “w dół”
- łatwe dodawanie nowych zależności
- w całości realizowane na etapie kompilacji
- zależności weryfikowane są przez kompilator
- typ powinien jednoznacznie identyfikować zależność

Wstrzykiwanie zależności

```
trait StatisticsService
trait UserService

trait SubPanel
class StatisticsPanel(implicit statService: StatisticsService)
  extends SubPanel
class UserInfoPanel(implicit userService: UserService)
  extends SubPanel

class UserDashboardPanel(implicit
  statService: StatisticsService,
  userService: UserService) {

  def addSubpanel(subPanel: SubPanel): Unit = ???

  addSubpanel(new StatisticsPanel)
  addSubpanel(new UserInfoPanel)
}
```

Implicit parameters - podsumowanie

- nie nadużywać dla wygody!
- typ niejawnego parametru identyfikuje go
- inne (bardzo ważne) zastosowania - później

Implicit conversions

- jednoargumentowe funkcje oznaczone jako `implicit`
- metody są widziane przez kompilator jako funkcje dzięki tzw. mechanizmowi eta-ekspansji
- używane automatycznie przez kompilator w sytuacji niezgodności typów
- niezgodność typów obejmuje również odwołania do nieistniejących składowych (np. metod)
  - implicit views, extension methods, implicit classes

Implicit views

- sposób na rozszerzanie API istniejących typów
- rozszerzone API to tzw. extension methods
- dedykowana składnia: implicit classes
- extension methods - de facto bardziej ogólne i dokładniejsze niż “prawdziwe” metody
- narzędzie pomocnicze - value classes - zachowują wydajność “prawdziwych” metod

Implicit views

```
implicit class StringOptionOps(private val opt: Option[String])
  extends AnyVal {

  def orEmpty = opt.getOrElse("")
}

Some("lol").orEmpty
```

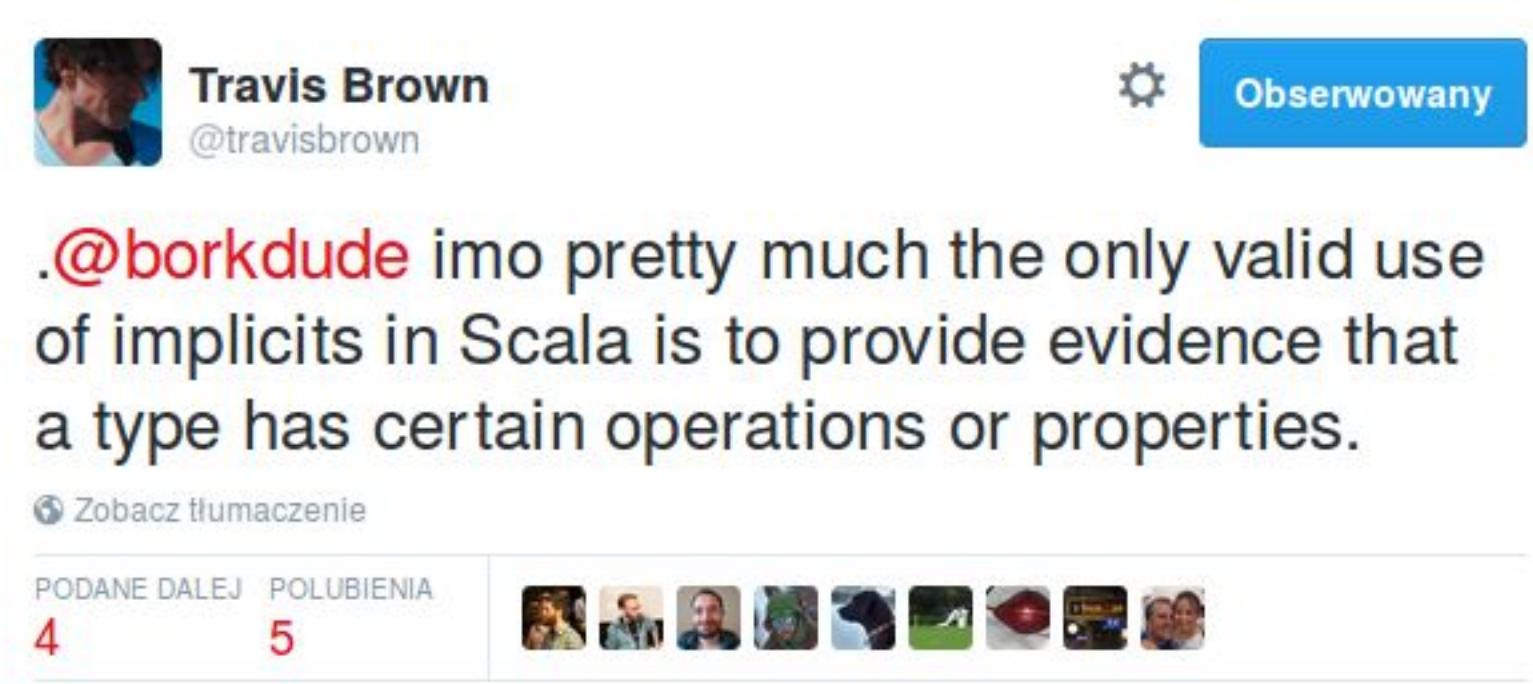
“Wygodne” niejawne konwersje

- stosowane w bibliotece standardowej
- subiektywnie niezalecane
- alternatywa: extension methods do jawnej konwersji

Priorytety implicitów

- kompilator poszukując niejawnych parametrów i konwersji preferuje pewne w stosunku do innych
- bardziej “specyficzne” implicity mają preferencję
- możliwe ręczne wpływanie na priorytety przez przenoszenie do klas i traitów bazowych
- dobra praktyka: zawsze jawnie specyfikować typy implicitów w celu uniknięcia niespodzianek

Niejawne parametry a polimorfizm



Klasa typów (*typeclass*)

- zbiór typów posiadających pewne wspólne cechy
- fundamentalny element systemu typów w Haskellu
- wspólna cecha typów to np.
  - możliwość wykonania określonych operacji związanych z danym typem
  - dostępność metadanych związanych z typem
  - powiązanie danego typu z jakimś innym
- polimorfizm ad-hoc - alternatywa dla dziedziczenia

Klasy typów w Scali

AVSYSTEM

Type Classes as Objects and Implicits

Bruno C. d. S. Oliveira
ROSAEC Center, Seoul National University
bruno@ropas.snu.ac.kr

Adriaan Moors Martin Odersky
EPFL
{adriaan.moors, martin.odersky}@epfl.ch

Type Classes

- Przykłady
  - `ClassTag`
  - `Ordering`
  - `Monoid`
  - `GenCodec`
  - `Functor, Applicative, Monad`

Dostępność operacji

```scala
trait HasDefaultValue[T] {
  def defaultValue: T
}

object HasDefaultValue {
  implicit val DefaultInt: HasDefaultValue[Int] =
    new HasDefaultValue[Int] {
      def defaultValue = 42
    }
}

def getOrDefault[T](option: Option[T])(implicit hdv: HasDefaultValue[T]): T =
  option.getOrElse(hdv.defaultValue)

def getOrDefault2[T: HasDefaultValue](option: Option[T]): T =
  option.getOrElse(implicitly[HasDefaultValue[T]].defaultValue)

getOrDefault(Option.empty[Int])
```

Zależności między implicitami

```
implicit def DefaultSet[T](implicit dt: HasDefaultValue[T]): HasDefaultValue[Set[T]] =
  new HasDefaultValue[Set[T]] {
    def defaultValue = Set(dt.defaultValue)
  }

getOrDefault(Option.empty[Set[Int]])
```

Dostępność operacji

```scala
trait Ordering[T] {
  def compare(x: T, y: T): Int
}

trait List[+A] {
  def sortBy[B](f: A => B)(implicit bo: Ordering[B]): List[A]
}
```

`Ordering` vs `Comparable`

- `Ordering` jest lepszy, bo:
  - jest związany z **typem**, a nie **klasą** i jej instancjami
    - typy, np. `Int`, `List[String]`, `Map[String, Set[Int]]`
    - klasy, np. `Int`, `List`, `String`, `Map`, `Set`
  - może być zależny od innych instancji `Ordering` (np. dla typu elementów listy)
  - istnieje dla typów, które nie mogą dziedziczyć `Comparable`
  - nie jest częścią definicji klasy ==> można go zdefiniować dla typów, nad którymi nie mamy kontroli

Ordering vs Comparable

```scala
implicit def listOrdering[T: Ordering]: Ordering[List[T]] =
  new Ordering[List[T]] {
    def compare(x: List[T], y: List[T]): Int = (x, y) match {
      case (Nil, Nil) => 0
      case (Nil, _) => -1
      case (_, Nil) => 1
      case (xhead :: xtail, yhead :: ytail) =>
        implicitly[Ordering[T]].compare(xhead, yhead) match {
          case 0 => compare(xtail, ytail)
          case res => res
        }
    }
  }
```

Dostępność metadanych

```scala
def extractInts(list: List[Any]): List[Int] =
  list.collect {
    case i: Int => i
  }

def extract[T](list: List[Any]): List[T] =
  list.collect {
    case t: T => t // warning + runtime failure!
  }
```

Dostępność metadanych

```
def extract[T](list: List[Any])(implicit ct: ClassTag[T]): List[T] =
  list.collect {
    case t: T => t
  }

def extractCb[T: ClassTag](list: List[Any]): List[T] =
  list.collect {
    case t: T => t
  }
```

Klasy typów i algebra

```scala
trait Monoid[T] {
  def zero: T
  def plus(t1: T, t2: T): T
}

object Monoid {
  val IntMonoid: Monoid[Int] = new Monoid[Int] {
    def zero: Int = 0
    def plus(t1: Int, t2: Int): Int = t1 + t2
  }
  val StringMonoid: Monoid[String] = new Monoid[String] {
    def zero: String = ""
    def plus(t1: String, t2: String): String = t1 + t2
  }
}
```

Wieloparametrowe klasy typów

- wyrażają *relację* pomiędzy dwoma lub więcej typami
- w szczególności mogą wyrażać *funkcje typów*, tj. “funkcje” na poziomie systemu typów Scali, które na podstawie typów-argumentów “zwracają” typ-rezultat
- powiązania mogą być arbitralne
- pozwalają programiście sterować inferencją typów, a nawet zmusić kompilator do wykonania skomplikowanych obliczeń na typach