УДК 004.432.42,004.77

Савенков О. А.

РОЗРОБКА HTTP-СЕРВІСУ НА ФУНКЦІЙНІЙ ПРЕДМЕТНО-ОРІЄНТОВАНІЙ МОВІ ПРОГРАМУВАННЯ НА БАЗІ ВІЛЬНИХ МОНАД

Розробка веб-сервісів чистим функційним способом є нетривіальною задачею через те, що там зазвичай наявні виклики бази даних на всіх етапах обробки запиту сервера, що ускладнює виокремлення функційного ядра коду, а це є основним принципом розробки функційної архітектури програми. Цю публікацію присвячено вирішенню цієї проблеми вільними монадами, за допомогою яких код, що працює зі змінними даними, можна розділити на функційне представлення імперативних операцій, що визначаються створеними на базі вільних монад предметно-орієнтованими мовами програмування (англ. Domain-Specific Language, DSL), та інтерпретатор, що власне виконує ці операції. У статті наведено основні принципи використання вільних монад та їхні переваги в контексті задачі розробки HTTP-сервісу.

Ключові слова: функційне програмування, вільні монади, предметно-орієнтована мова програмування, чисті функції, функційний веб-сервіс.

Вступ

Функційне програмування має багато переваг порівняно з імперативним [1, с. 41]. Використання переважно чистих функцій значно зменшує кількість помилок, що виникають унаслідок побічних ефектів (англ. side effects) та неконтрольованих залежностей даних від глобального стану програми або зовнішнього світу [3, p. 169]. Окрім цього, чистий функційний код набагато легше тестувати [3, p. xxv].

У типовій архітектурі програми, написаній чистим функційним методом, код ділиться на дві частини: незмінне функційне ядро, побудоване виключно на чистих функціях, та імперативна оболонка, де відбувається вся взаємодія із зовнішнім світом та зберігається весь змінний глобальний стан програми (якщо він є необхідним).

Під час розробки веб-серверів важко виокремити функційне ядро, тому що на всіх етапах обробки запиту може відбуватися безпосередня робота з базою даних або іншим змінним сховищем, що відповідає за глобальний стан сервера.

Без використання допоміжних засобів результатом розбиття веб-сервера на функційне ядро та імперативну оболонку може бути дуже велика оболонка та маленьке ядро, хоча за суттю розбиття потрібно, щоб було навпаки.

Базові поняття

Функційне програмування ґрунтується на теорії категорій, трьома основними елементами якої є функтор, аплікатив та монада. Усі три поняття об'єднує те, що вони «загортають» довільний тип у певний контекст. Причому аплікатив наслідується від функтора, а монада – від аплікатива. Найчастіше вони використовуються як абстракції над колекціями [4, p. 43] та обчисленнями [4, p. 42].

Функтор, аплікатив та монада мають визначати функції fmap, <*> та >>= відповідно з такими сигнатурами:

```
fmap :: (a -> b)  -> F a -> F b
<*> :: A (a -> b)  -> A a -> A b
>>= :: M a -> (a -> M b) -> M b
```

Причому імплементації функцій fmap, <*>, >>= мають задовольняти не лише сигнатуру, а й функторні, аплікативні та монадні закони відповідно, що детально розглянуто у [5].

Як приклад розглянемо реалізації функторів, аплікативів та монад для таких типів:

- списку;
- Maybe (у деяких мовах програмування – Option, Optional, Nullable). Maybe t може містити 0 або 1 елемент вказаного типу;
- типу Either. Either a b може мати значення одного з двох типів: Right a або Left b;
- функції з одним аргументом.

Спочатку розглянемо поведінку функторів.

- Для списку функція fmap застосовує трансформацію, визначену функцією-параметром на кожному елементі списку.
- Для типу Maybe функтор трансформує значення функцією-параметром, якщо це значення існує.

- Функтор Either трансформує, тільки якщо воно має підтип Right a. Left b залишається без змін. Ця поведінка дуже схожа на поведінку функтора Maybe.
- У випадку з функцією з одним аргументом функція fmap має сигнатуру (b -> c) -> (a -> b) -> (a -> c) та просто послідовно застосовує дві функції до аргумента.

```
fmap (+1) [1,2,3]
-- Результат: [2,3,4]
fmap (*2) (Just 7)
fmap (*2) (Right 7)
-- Результат: Just 14; Right 14
fmap (*2) Nothing
fmap (*2) (Left 1)
-- Результат: Nothing; Left 1
(fmap (*3) (+1)) 1
-- Результат: 6
-- детально: (1 + 1) * 3
```

Далі розглянемо поведінку функції <*> в аплікативах.

- Для списку: кожна функція зі списку функцій попарно застосовується до кожного елемента зі списку елементів. Усі результати попарних застосувань заносяться до нового списку.
- Для типу Maybe: результат повертається тільки тоді, якщо і функція, і параметр присутні. В іншому випадку повертається Nothing.
- Для Either: результат повертається, тільки якщо і функція, і параметр є підтипами Right.
- Для типу функції: функція <*> має сигнатуру (a->b->c) -> (a->b) -> (a->c). Параметр результуючої функції спочатку застосовується до другої функції-аргумента типу (a->b), потім результат типу *b*, а також той самий параметр типу *a* застосовуються до першої функції. Результат першої функції-параметра від цих двох аргументів і є результатом вихідної функції.

```
[(+1),(*10)] <*> [1,2,3]
-- Результат: [2,3,4,10,20,30]
(Just (+1)) <*> (Just 7)
(Right (+1)) <*> (Right 7)
-- Результат: Just 8; Right 8
Nothing <*> (Just 7)
(Right (+1)) <*> (Left "error")
-- Результат: Nothing; Left "error"
((+) <*> (*2)) 2
-- Результат: 6
-- детально: 2 * 2 + 2
```

Далі розглянемо монади.

- Функція >>= монади списку має сигнатуру [a] -> (a -> [b]) -> [b] та визначена таким чином: для кожного елемента списку [a] застосовується функція (a -> [b]), результати якої конкатенуються за відповідним порядком параметрів у списку [a].
- Для типу Maybe відповідна сигнатура має вигляд Maybe a -> (a -> Maybe b) -> Maybe b і повертає елемент лише у випадку Just a -> (a -> Just b) -> Just b. Якщо або початковий елемент, або результат застосування функції дорівнює Nothing, то і результат буде Nothing.
- Для Either: поведінка аналогічна до поведінки Maybe, як і в попередніх випадках із функтором та аплікативом.
- Для функції з одним параметром функція >>= має сигнатуру (a -> b) -> (b -> a -> c) -> (a -> c) та поведінку, дуже схожу на поведінку функції як аплікатива.

```
[1,2,3] >>= (\x -> [x * 10, x * 100])
-- Результат: [10, 100, 20, 200, 30, 300]
(Just 4) >>= (\x -> Just (x * 5))
(Right 4) >>= (\x -> Right (x * 5))
-- Результат: Just 20; Right 20
(Just 4) >>= (\x -> Nothing)
(Right 4) >>= (\x -> Left "error")
(Left "error") >>= (\x -> Right (x * 5))
-- Результат: Nothing; Left "error"; Left "error"
((+4) >>= (*)) 2
-- Результат: 12
-- детально: (2 + 4) * 2
```

У функторів, аплікативів та монад, що оперують над різними типами, немає чіткої закономірності в задачах, які вони розв'язують. У них спільні лише сигнатури функцій та виконання законів, що розглянуто у [5]. Найчастіше вони реалізовані над типом колекції [4, p. 43] (список, Maybe тощо) або обчислення [4, p. 42] (функція тощо).

Розглянуті елементи з теорії категорій є корисними абстракціями, що буде показано далі на прикладі вільних монад.

Вільні монади

Вільні монади – конструкція, що дозволяє перетворити будь-який функтор на монаду, запозичивши поведінку функції >>= з будь-якої іншої існуючої монади [2]. Зазвичай вільні монади використовуються для задання нових DSL із поведінкою композиції окремих операцій,

схожою на поведінку операції >>=, що запозичується із вже існуючих монад.

Протилежним поняттям до вільних монад, що можуть запозичувати будь-яку нову поведінку оператора >>=, є забудькуваті функтори, що можуть забувати цю поведінку і перетворюватися – навпаки – з монади на функтор [2].

Найчастіше DSL, побудовані на вільних монадах, запозичують свою монадну поведінку у Either. Згадаємо його поведінку функції >>=: Right a повертається лише у випадку (Right a) >>= (\a -> Right b). Якщо початковий елемент або результат функції дорівнює Left b, тоді результат застосування >>= буде дорівнювати Left b.

Такою поведінкою Either дозволяє зручно працювати з обчисленнями, під час яких може статися помилка. У контексті таких обчислень Right a – результат вдалого обчислення, а Left b – помилка, що сталася під час нього.

Якщо в послідовності обчислень із можливою помилкою, що визначені типами Either, на будь-якому етапі виникне помилка, то за визначенням функції >>= для цього типу поведінка буде дуже схожа на оператор throw в імперативних мовах програмування – обчислення припиниться відразу, а невдалий результат першого невдалого обчислення буде результатом усієї послідовності обчислень, визначених у контексті монади Either. Саме тому DSL для алгоритмів з можливими помилками під час обчислень базуються на Either.

Опис імперативних операцій вільними монадами

Веб-сервіс у межах цієї статті написаний мовою Scala із застосуванням функційної бібліотеки Cats. Суть однакова для всіх типізованих функційних мов програмування, зокрема Haskell, F#, Scala тощо.

У межах цієї статті буде наведено реалізацію DSL для точки доступу сервера месенджера, що надсилає повідомлення в приватний чат.

Вхідні дані точки доступу:

- поточний користувач (задається через токен авторизації у HTTP-заголовку);
- ідентифікатор користувача, якому надсилається повідомлення;
- текст повідомлення.

Алгоритм:

- переконатися, що токен авторизації, переданий у заголовку, правильний та користувач із таким токеном існує;
- переконатися, що користувач з id отримувача існує;
- перевірити, чи існує діалог між поточним користувачем та отримувачем. Якщо ні, то створити новий;
- надіслати повідомлення.

Вихідні дані:

- якщо хоч один етап алгоритму не був успішним, повернути помилку;
- за успішного виконання повернути оновлений список повідомлень у діалозі.

Типи, що описують імперативні операції, задаються в коді так:

```
// Шаблонний абстрактний тип, від якого породжуються всі операції DSL
// T – результат операції
trait ServerOpA[T]

// Цей тип даних описує операцію, що завжди повертає значення value: T
case class Pure[T](value: T) extends ServerOpA[T]
// Описує операцію, що повертає рядок користувача з БД за id
case class GetUserById(userId: Long) extends ServerOpA[User]
// Описує операцію, що повертає рядок користувача з БД за токеном авторизації
case class GetUserByToken(token: String) extends ServerOpA[User]
// Описує операцію, що повертає рядок приватного діалогу з БД із заданими користувачами
case class GetPrivateDialog(user1: User, user2: User) extends ServerOpA[Option[Dialog]]
// Описує операцію, що створює новий приватний діалог у БД із заданими користувачами
case class CreatePrivateDialog(user1: User, user2: User) extends ServerOpA[Dialog]
// Описує операцію, що створює нове повідомлення у БД
case class SendMessage(dialog: Dialog, sender: User, message: String) extends ServerOpA[Unit]
// Описує операцію, що повертає всі повідомлення заданого діалогу
case class GetMessagesByDialog(dialog: Dialog) extends ServerOpA[Seq[Message]]

// ServerOp[T] – піднесення операції ServerOpA до монади
// T – результат обчислень операції
type ServerOp[T] = cats.Free[ServerOpA, T]
```

Детальну реалізацію конструктора Free наведено у [2].

Було визначено операції типу ServerOpA[T], що є функторами. Але тепер потрібно, щоб ці самі операції мали тип вільної монади ServerOp[T].

```
// Перетворення функтора ServerOpA[T] на вільну монаду ServerOp
def liftServerOp[T](serverOpA: ServerOpA[T]): Free[ServerOpA, T] = liftF[ServerOpA, T](serverOpA)

def pure[T](value: T): ServerOp[T] = liftServerOp(Pure(value))
def getUserById(userId: Long): ServerOp[User] =
 liftServerOp(GetUserById(userId))
def getUserByToken(authToken: String): ServerOp[User] =
 liftServerOp(GetUserByToken(authToken))
// … І так для кожної визначеної операції
// Існує спосіб автоматизації створення конструкторів вільних монад
// Але це не розглядається в цій статті
```

Комбінування операцій та опис точки доступу засобами створеної DSL:

```
// Підпрограма, що повертає рядок БД, відповідний поточному користувачу, використовуючи токен авторизації з HTTP-заголовка
def getUserFromAuthorizationHeader(request: HttpRequest): ServerOp[User] =
 for {
  userToken <- getTokenFromRequest(request)
  user <- getUserByToken(userToken)
 } yield user

// Визначення точки доступу надсилання повідомлення до приватного діалогу
def sendPrivateMessageAction(recipientId: Long, message: String, request: HttpRequest): ServerOp[Seq[Message]] =
 for {
  currentUser <- getUserFromAuthorizationHeader(request)
  recipient <- getUserById(recipientId)
  dialogMaybe <- getPrivateDialog(currentUser, recipient)
 dialog <- dialogMaybe.fold(ifEmpty = createPrivateDialog(currentUser, recipient))(dialog => pure(dialog))
  _ <- sendMessage(dialog, sender = currentUser, message)
  messages <- getMessagesByDialog(dialog)
 } yield messages
```

Варто зауважити, що оператор for у Scala майже не має нічого спільного з імперативним циклом for. У Scala цей оператор є аналогічним до оператора do в Haskell. Значення праворуч від кожної стрілки в тілі оператора мають бути монадами. Значення ліворуч – результат її обчислення. Між кожною парою сусідніх стрілок неявно застосовується комбінування монад функцією >>=. Монада ServerOp базуватиметься на Either[HttpError, _]. Це означає, що якщо станеться помилка (якийсь з операторів поверне Left, а не Right), то послідовність виконання обчислень у тілі for буде перервано, і цю помилку буде повернуто відразу. Ця логіка базується лише на застосуваннях >>= для типу Either та не використовує імперативних операторів throw тощо.

Коли значення ServerOp[T] ініціалізується, то задана ним послідовність операцій не виконається, тому що ServerOp лише описує імперативні операції, а за їх виконання відповідає інтерпретатор.

Реалізація інтерпретатора

Інтерпретатор безпосередньо виконує команди, задані типом ServerOp.

Для роботи з базою даних використовується бібліотека ScalikeJDB.

Для реалізації HTTP клієнту використовується Akka HTTP.

Вибір бібліотек для роботи із зовнішнім світом не є принциповим у межах цієї статті.

Далі наведено повну реалізацію інтерпретатора DSL для запитів у базу даних, що використовуються для надсилання повідомлення в діалог.

```
// DSL базується на монаді Either[HttpError, _]
type ServerOpResult[T] = Either[HttpError, T]

def interpreter: ServerOpA ~> ServerOpResult =
 new (ServerOpA ~> ServerOpResult) {
  override def apply[A](fa: ServerOpA[A]): ServerOpResult[A] = {
    case Pure(value) => Right(value)
    case GetUserById(userId: Long) =>
      withSQL {
        select.from(User as u).where.eq(u.id, userId)
      }.map(User(_)).single().apply() match {
        case Some(user) => Right(user)
        case None => Left(HttpError(400, "User with specified id does not exist"))
      }
    case GetUserByToken(token: String) =>
      withSQL {
        select.from(User as u).where.eq(u.authToken, token)
      }.map(User(_)).single().apply() match {
        case Some(user) => Right(user)
        case None => Left(HttpError(403, «UNAUTHORIZED»))
      }
    case GetPrivateDialog(user1: User, user2: User) =>
      Right(
        withSQL {
          select(d.*).from(Dialog as d).where
            .exists(select.from(UserDialog as ud).where.eq(ud.userId, user1.id)).and
            .exists(select.from(UserDialog as ud).where.eq(ud.userId, user2.id))
        }.map(Dialog(_)).single()
      )
    case CreatePrivateDialog(user1: User, user2: User) =>
      val dialogId = withSQL {
        insert.into(Dialog).values(Dialog())
      }.updateAndReturnGeneratedKey().key.asInstanceOf[Long]

      withSQL {
        insert.into(UserDialog).values(UserDialog(dialogId, user1.id), UserDialog(dialogId, user2.id))
      }.update()

      withSQL {
        select.from(Dialog as d).where.eq(d.id, dialogId)
      }.map(Dialog(_)).single().apply()

    case SendMessage(dialog: Dialog, sender: User, message: String) =>
      withSQL {
        insert.into(Message).values(Message(id = None, dialog.id.get, sender.id, message))
      }.update()

    case GetMessagesByDialog(dialog: Dialog) =>
      withSQL {
        select.from(Message as m).where.eq(m.dialogId, dialog.id)
      }.map(Message(_)).list().apply()
  }
}
```

Тепер залишається лише запустити опис операцій в інтерпретаторі.

Ця версія месенджера складається лише з однієї точки доступу, що надсилає повідомлення в діалоги. У тілі цієї точки доступу виконується спочатку генерація змінної типу ServerOp[Seq[Message]], що задає алгоритм. Після цього відбувається послідовно передача кожної команди в інтерпретатор за допомогою методу foldMap, що визначений на монаді Free у бібліотеці Cats.

```
class SendMessageService extends Directives with JsonSupport {

 val route =
  post {
   entity(as[SendPrivateMessageInfo]) { case SendPrivateMessageInfo(recipientId, message, request) =>

    // Опис імперативних операцій
    val program : ServerOp[Seq[Message]] = sendPrivateMessageAction(recipientId, message,
request)

    // Виконання операцій
    program.foldMap(interpreter) match {
     case Right(messagesInDialog) => complete(200, messagesInDialog)
     case Left(HttpError(status, msg)) => complete(status, msg)
    }

   }
  }
}
```

Використовуючи створену DSL та її інтерпретатор, була реалізована точка доступу сервісу обміну повідомленнями, що надсилає повідомлення в діалог.

Висновки

Було реалізовано функційну предметно-орієнтовану мову програмування на базі вільної монади в межах точки доступу HTTP-сервера, що виконує багато запитів до бази даних, деякі з яких можуть не виконуватися залежно від результатів попередніх запитів. Без використання вільної монади код був би хоч і написаний на функційній мові програмування, проте за своєю суттю був би імперативним, тому що вся програма була би побудована на нечистих функціях. Використання вільної монади дозволяє спочатку описати імперативні операції функційними структурами даних та зберегти їх як значення, а потім передати їх в інтерпретатор, що власне виконує їх.

Побудована DSL коректно реагує на помилки, використовуючи поведінку, запозичену у монади Either.

На відміну від методу розробки без використання вільних монад, кількість імперативного коду з їхнім використанням залежить не від розміру усього коду, а від кількості операцій, визначених у межах створеної предметно-орієнтованої мови. Різниця може бути великою, особливо якщо одиничні операції перевикористовуються багато разів.

Проведене дослідження підтверджує, що функційні предметно-орієнтовані мови програмування, побудовані на вільних монадах, можуть значно зменшити кількість імперативного коду в контексті задачі розробки HTTP-сервісу на функційній мові програмування.

Список використаної літератури

1. Каширець В. Переваги функціональної парадигми при розробці програмного забезпечення / В. Каширець, Я. Кінах // Матеріали науково-технічної конференції «Інформаційні моделі, системи та технології» (Тернопіль, 24 квіт. 2013 р.). – Тернопіль : ТНТУ, 2013. – С. 41.
2. Free Monad [Electronic resource]. – Mode of access: https://typelevel.org/cats/datatypes/freemonad.html. – Title from the screen.
3. O'Sullivan B. Real World Haskell / B. O'Sullivan, D. Stewart, J. Goerzen. – Sebastopol, USA : O'Reilly, 2008. – 714 p.
4. Petricek T. The F# Computation Expression Zoo / T. Petricek, D. Syme // Materials of 16th International Symposium "Practical Aspects of Declarative Languages" (San Diego, USA, 19–20 January 2014). – Cham, Switzerland : Springer, 2014. – P. 33–48. – ISBN 9783319041322.
5. Typeclassopedia [Electronic resource] / HaskellWiki. – Mode of access: https://wiki.haskell.org/Typeclassopedia. – Title from the screen.

Oleksii Savenkov

IMPLEMENTING HTTP-SERVICE IN A FUNCTIONAL DOMAIN-SPECIFIC LANGUAGE BASED ON FREE MONADS

Implementing web-services in a pure functional way is not a trivial task, because it is common to a query database (or a different type of mutable data store) on all the stages of the server request processing. This complicates separation of the pure functional core and imperative shell of the program, which is the primary principle of the functional software architecture. This article focuses on resolving this problem by using free monads, which makes it possible to implement a new domain-specific language that is defined by purely functional data structures, to describe imperative operations. Each operation is later interpreted into actual data mutation by an interpreter. This allows defining the algorithms as a combination of atomic DSL operations, which may have side-effects, in a purely functional way, while only the interpretation logic of those operations remains imperative.

In contrast to a standard approach, the amount of the imperative code in free monad based algorithms scales depending on the number of atomic DSL operations instead of the size of the whole program. This can bring a big difference, especially if atomic operations are frequently re-used.

As part of the research, an HTTP-based message exchange service was implemented in its own free monad based domain-specific language. As a basis, functional programming language Scala was used. The article shows the implementation of the server endpoint that sends the message to a specific conversation. The endpoint implementation consists of many atomic database queries and contains multiple branches of the control flow.

First, each atomic database query or group of related queries is described as a pure functional record where dynamic query parameters are defined as values of the record fields. Then, using a free monad constructor, these operators are upgraded to monads and get the combination logic from the already existing monadic instances. Usually either one is used as a basis, since it provides a convenient way to handle exceptions. Next, atomic monadic operations are combined into algorithms. Most functional languages have syntactic sugar to write expressions, which combine multiple monads. In case of Scala, it is a for-expression. To interpret a DSL-based expression, an interpreter should be declared and used. The interpreter is a function with a defined behaviour for atomic DSL operations which are defined earlier. This behaviour may not be purely functional and may have side effects.

The conducted research confirms that the usage of free monad based functional domain-specific programming languages is suitable in context of implementing HTTP-services, and it may significantly decrease the amount of the imperative code.

Keywords: functional programming, free monads, domain-specific language, pure functions, functional web-service.

Матеріал надійшов 31.05.2018