DOI: <https://doi.org/10.15407/emodel.43.06.095>
УДК 004.422.639

І.В. Пучко¹, А.М. Примушко², Г.О. Кравцов³, канд. техн. наук

¹ Національний авіаційний університет
Україна, 03680, Київ, пр-т Любомира Гузара, 1
e-mail: ivan.puchko@gmail.com

² Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Україна, 03056, Київ, пр-т Перемоги, 37
e-mail: arsentiy.prymushko@gmail.com

³ Інститут проблем моделювання в енергетиці
ім. Г.Е. Пухова НАН України
Україна, 03164, Київ, вул. Генерала Наумова, 15
e-mail: hryhoriy.kravtsov@gmail.com

Розробка методичних рекомендацій використання функціональної парадигми програмування в мові Scala

Розглянуто різні підходи до визначення структур даних та створення програм в різних парадигмах програмування. Показано залежність імперативної парадигми програмування від алгоритмів та операцій низького рівня. Наведено ідеї щодо основних змінних структур даних та операцій щодо їх мутації. Описано підходи до організації алгоритмів у мові програмування Scala за допомогою цих структур даних. Показано, що у декларативній парадигмі програмування фокус переходить від алгоритмів до математичних виразів, та продемонстровано, як, використовуючи операції високого рівня, створити програму за допомогою алгебраїчних виразів.

К л ю ч о в і с л о в а: Scala, функціональна парадигма програмування, імперативна парадигма програмування, декларативна парадигма програмування, лямбда-функції, анонімні функції, функції вищого порядку.

З часом проблеми, які вирішуються за допомогою програмного забезпечення, стають складнішими, що зумовлює появу нових стандартів та підходів до розробки програмних рішень. В останні роки спостерігається уповільнення швидкості роботи процесора. Тим не менш, системи прискорюються, додаючи більше ядер, які працюють паралельно.

Одним із ефективних рішень сучасних проблем розробки програмного забезпечення є функціональна парадигма програмування (ФПП),

© Пучко І.В., Примушко А.М., Кравцов Г.О., 2021

яка є природною для паралельних обчислень. Scala — це мова програмування, яка підтримує одночасно функціональну та імперативну парадигми програмування.

Розглянемо основні принципи та підходи до реалізації програмного забезпечення з використанням мови програмування Scala.

Функціональна парадигма програмування. У ФПП будь-яка програма розглядається як процес обрахунку математичних функцій. Функціональне програмування (ФП) уникає зміни станів і протиставляється імперативному програмуванню (ІП), що описує процес обрахунку як послідовну зміну станів.

Функціональні мови програмування є декларативними, а саме логіка обчислення подається без опису потоку управління, тобто замість тверджень програмісти використовують вирази, щоб описати необхідний результат виконання програми без додаткового опису отримання цього результату.

Декларативна парадигма програмування. В ФП та ІП розробники оперують примітивами, описаними мовою програмування. Необхідна функціональність є результатом поєднання описаних примітивів. Під примітивом розуміємо програмну абстракцію — структури управління циклом, операції, специфічні для колекцій, тощо. Наприклад, операції створення нової колекції, додавання або видалення елементів з колекції є примітивами.

На відміну від ІП в декларативному програмуванні (ДП) примітиви тісніше пов'язані з їхньою предметною областю, що уможливило створення специфічних доменних мов програмування (DSL) та примітивів в ході виконання програми.

Для розуміння переваг ДП розглянемо наступний приклад циклу, описаного оператором `go-to` (рис. 1). Тут `loop` (цикл) — це шаблон. Шаблоном є розташування двох і більше логічних елементів у коді, які повторюються в різних місцях програми. В даному прикладі шаблоном є цикл, що складається з трьох частин: 1— мітка, що є точкою входу в тіло циклу, де відбувається перехід з кінця циклу, щоб його повторити; 2— умова, яка має бути істинною, щоб цикл повторювався; 3— твердження, що є кінцем тіла циклу.

Потрібно описати цикл `while`. Є змінна `x`, яку треба збільшити в тілі циклу на одиницю, допоки вона менша десяти. В імперативних мовах програмування, таких як Java, це можна зробити за допомогою циклу `while`. Але такого ж результату можна досягти за допомогою оператора `go-to`, якщо мати відповідну мітку на операторі збільшення. Тоді умовний оператор після операції збільшення перевірятиме попередню умову

для змінної x . Можна продовжувати виконання циклу, якщо умова не виконується.

Повторна реалізація шаблону циклу, що складається з декількох частин (під час виклику при неправильному використанні або їх об'єднанні), може бути причиною неправильного виконання програми. Мітки можуть бути неназвані, або названі неправильно. Логічний оператор, що є умовою переходу до початку циклу, може бути визначений неправильно тощо.

```
#include <iostream>
using namespace std;
int main(){
    int x = 0;
    loop_start:
        x++;
    cout << x << "\n";
    if (x < 10) goto loop_start;
    return 0;
}
```

Рис. 1. Цикл, описаний оператором go-to

Збільшення складності алгоритму призводить до зростання ймовірності виникнення помилок, зокрема, пов'язаних з мітками.

Доброю практикою програмування є дотримання принципу «не повторюватись» (Don't Repeat Yourself), що дозволяє мінімізувати вірогідність виникнення подібних помилок. Одним із фундаментальних правил інженерії є створення абстракцій для повторюваної логіки, за допомогою яких шаблон спрощується до одного компонента — ключового слова з певним синтаксисом.

В прикладі, наведеному на рис. 1, використано шаблон, який викликається на кожній ітерації роботи програми, якщо виконується логічна функція. Цей виклик за ім'ям є основним принципом декларативного програмування, а саме реалізація шаблонів з відповідною назвою, що повторюються лише один раз, та подальший їх виклик в будь-якому місці програми.

Наприклад while, for, loop є абстрактними шаблонами циклів, що реалізуються на мовному рівні. Для повторного виклику шаблону можна звернутись до нього за ключовим словом з відповідним синтаксисом. Це дозволяє зменшити ймовірність виникнення помилок, тому що середовище (надалі компілятор) знає про шаблон. Кожного разу відбувається перевірка умови компіляції, тобто відповідність вхідних даних описаному раніше синтаксису. В цьому випадку компілятор виконує всю логіку переходу самостійно. Це унеможливорює неконтрольований перехід від кінця внутрішнього циклу до початку зовнішнього.

Описана концепція полягає в тому, що мова програмування є підпорядкованою певній закономірності. Компілятор попередньо перевіряє правильність шаблону під час компіляції. Це дозволяє виконати наступне:

- визначити шаблон один раз;
- дати унікальну назву шаблону;

визначити певні обмеження на програміста, який використовує цю назву;

делегувати реалізацію шаблону мові програмування.

Сучасні мови, такі як Java, Scala, Python або Haskell, не підтримують оператор `go-to`. Переважну більшість програм, поданих за допомогою оператора `go-to`, можна реалізувати як набір шаблонів, подібних до циклів, які відрізняються від операторів переходу. Програмістам рекомендовано використовувати ці шаблони вищого рівня за унікальною назвою, а не реалізовувати логіку самостійно за допомогою примітивів нижчого рівня.

Відтак, ДП — це парадигма програмування, відповідно до якої, операції, що потрібно виконати, викликають за унікальною назвою замість того, щоб описувати, як їх виконувати алгоритмічно за допомогою примітивів нижчого рівня, визначених мовою програмування. ДП повністю узгоджується з принципом «не повторюватись».

Функції та лямбди. Сучасне ФП разом з ДП забезпечує кращий рівень абстракції, який можна використовувати для уніфікації повторюваних операцій. ФП має багато спільних рис з ДП, зокрема щодо використання функцій.

Розглянемо детальніше, що таке функції та їх роль в різних парадигмах програмування. Функціями можна назвати будь-які параметризовані частини коду, що мають унікальну сигнатуру — унікальне ім'я, списки параметризованих аргументів.

Функція є своєрідною абстракцією, що дозволяє інкапсулювати повторювану логіку. В об'єктно-орієнтованому програмуванні функції зазвичай розглядають в контексті об'єкта або класу, тобто вони є одиницею взаємодії між об'єктами різних класів. У ІП поведінка об'єктів зазвичай має побічні ефекти, коли функція змінює середовище поза власним тілом. Натомість у ФП функції мають наступні характеристики:

чистота (*pure*) — відсутність побічних ефектів;

прозорість посилань, або відносна прозорість (*referential transparency*).

Відносна прозорість означає, що значення результату, обчислене функцією, можна замінити викликом функції, при цьому семантика виконання програми залишиться незмінною.

Розглянемо наступний приклад. Є програма, яка імітує содовий автомат. Його поведінку описує наступне правило: при отриманні монети, содовий автомат видає пляшечку содової води. Автомат містить такі дані: кількість грошей; кількість банок з газованою водою, наявних у машині. Для того щоб описати поведінку автомата в ІП, треба створити окремий об'єкт та визначити потрібні методи в ньому. Кожен виклик методу супроводжується зміною кількості монет та пляшок води в автоматі.

```
public class ImperativeSodaMachine {
    private int coins = 0;
    private int cans = 0;
    public ImperativeSodaMachine(int initialCans) {
        this.cans = initialCans;
    }
}
```

Рис. 2. Об'єкт автомата з содовою водою

```
public SodaCan insertCoin() {
    if (cans > 0) {
        cans--;
        coins++;
        return new SodaCan();
    }
    else throw new RuntimeException("Out of soda cans!");
}
}
```

Рис. 3. Код автомата, що дозволяє перевірити його внутрішній стан

```
public class SodaCan {
    public void drink() {
        System.out.println("You have drunk a can of soda.");
    }
}
```

Рис. 4. Об'єкт SodaCan

Опишемо автомат з содовою водою, дотримуючись ІІ, кодом, наведеним на рис. 2. Визначимо поведінку автомата наступним чином: якщо в автоматі залишилися пляшечки з газованою водою, то зменшуємо кількість пляшок на одну та збільшуємо кількість монет на одну, після чого повертаємо об'єкт содової води, який містить додатковий функціонал при суперечностях у програмі.

Розглянемо випадок, коли в автоматі не залишилося пляшок з содовою водою, описаний кодом, наведеним на рис. 3.

Опишемо також об'єкт SodaCan, наведений на рис. 4. Кількість пляшок та монет, наявних у автоматі, є змінними для автомата з содою і належать його середовищу. Вони не належать безпосередньо тілу функції, яка змінює не своє внутрішнє середовище, а середовище об'єкта, до якого належить.

Тепер опишемо автомат з содою, дотримуючись ФП, мовою програмування Scala (рис. 5). Наведений на рис. 5 автомат не інкапсулює жодної поведінки. insertCoin — функція, що обчислює новий стан автомата, який, в свою чергу, не містить змінюваних змінних, тобто є статичним по

```
case class SodaMachine(cans: Int, coins: Int = 0)
def insertCoin(sm: SodaMachine): (SodaMachine, SodaCan) =
  if (sm.cans > 0) (SodaMachine(sm.cans - 1, sm.coins + 1), new
SodaCan)
  else throw new RuntimeException("Out of soda cans!")
```

Рис. 5. Автомат, описаний мовою програмування Scala

```
@annotation.tailrec
def whileDie(predicate: => Boolean)(body: => Unit): Unit =
  if (predicate) {
    body
    whileDie(predicate)(body)
  }
}
```

Рис. 6. Визначення структури управління whileDie, що є ФВП, аналогом імперативного циклу

своїй суті. Для ФП характерним також є використання функцій вищого порядку (ФВП) (higher order functions).

Функції вищого порядку — це функції, які приймають інші функції як вхідні аргументи, або продукують нові функції як результат. Наприклад, цикл while може бути подано функціонально як ФВП, що приймає тіло циклу та предикат як аргумент. Тіло циклу можна відобразити як функцію, яка не приймає жодних аргументів, але обчислює деякі побічні ефекти. Принцип роботи полягає в тому, що є функція, яка приймає функцію нульового аргументу та предикат. Далі та сама циклічна функція викликається рекурсивно, поки предикат є істинним.

Можна визначити нову структуру управління whileDie, з дотриманням ФП, через ФВП в мові програмування Scala (рис. 6). Конструкція whileDie приймає предикат та тіло. Предикат буде оцінюватись під час кожного виклику функції і, якщо він істинний, будемо запускати тіло і знову рекурсивно викликати конструкцію whileDie.

Зауважимо, що в анотації @annotation.tailrec вказано, що метод буде викликатися рекурсивно, а це унеможливорює розповсюджену помилку «StackOverflowError». Це пояснюється повторним використанням фрейм стеку (stack frame) [6] свого початкового виклику для всіх наступних рекурсивних викликів.

Нову конструкцію можна використати, як показано на рис. 7, а. Приклад з вбудованим циклом while наведено на рис. 7, б.

Як бачимо, визначення практично ідентичне. Це свідчить про те, що ФВП можна використовувати для визначення структур управління, подібних до вбудованих у мову.

```

var j = 0
whileDie (j < 5) {
  println(s"Printing from custom while loop. Iteration: $j")
  j += 1
}

                                а

var i = 0
while (i < 5) {
  println(s"Printing from built-in while loop. Iteration: $i")
  i += 1
}

                                б

```

Рис. 7. Цикл з використанням whileDie (а) та вбудований цикл while (б)

```

case class Cookie(name: String, gender: String)

def greeting(cookie: Cookie)(modifier: (String, String) =>
String): Unit = {
  val name      = cookie.name
  val gender    = cookie.gender
  val modifiedName = modifier(name, gender)
  print(s"Hello, $modifiedName")
}

def isPhd(name: String): Boolean = name == "Smith"

val cookie = Cookie("Smith", "male")

greeting(cookie) { (name, gender) =>
  if (isPhd(name)) s"Dr $name"
  else gender match {
    case "male"    => s"Mr $name"
    case "female" => s"Mrs $name"
  }
}

```

Рис. 8. Реалізація заданої функції: Cookie — фіктивний клас, що симулює cookie користувача; isPhd — користувацька реалізація логіки модифікатора; greeting — кінцева реалізація заданої функції

Лямбда-функція — це анонімна функція або її формалізація, визначені лямбда-численням [2, с. 53]. Лямбда-функція може бути присвоєна змінній, наприклад функції, що приймає файл cookie з даними сеансу користувача в контексті web-програми. Завданням функції є вивід привітання користувачеві на стандартний вихід. Однак перед друком потрібно певним чином модифікувати ім'я користувача. Є база даних користувачів, які мають наукові ступені докторів наук, і, якщо користувач є доктором, потрібно звертатись до нього з приставкою «доктор». Це можна реалізувати в Scala, як показано на рис. 8, де greeting — спосіб

привітання, що витягає дані з об'єкта Cookie та застосовує модифікатор на ім'я користувача залежно від його статі (цей метод не має точних інструкцій зміни імені).

Функція привітання приймає рядок, а також функцію, яка змінює цей рядок. Під час виклику цієї функції задається модифікатор, який змінює вбудований рядок. Модифікатор не визначається явно до передачі його у функцію привітання. Це і є ідея лямбда-функції. Не потрібно визначати функцію, до першого її виклику в ФВП. Замість цього можна визначити, які функції вбудовано, використовуючи лямбда-синтаксис. Вочевидь, такий підхід корисний у контексті ФВП, тому що дозволяє використувати ФВП без попереднього визначення їх аргументів.

Концепція лямбда-функції присутня у більшості функціональних мов програмування, включаючи Scala, Haskell та Python. У функціональних мовах програмування існує поняття ФВП, анонімних лямбда-функцій та часткових функцій, які з точки зору інженерії програмного забезпечення є корисними для абстрагування будь-якої логіки програми.

Алгебраїчний підхід — це мова математичних виразів, що складається з двох основних елементів: операторів та операндів. Операнди — це інформація, якою маніпулюють. Оператори — це спосіб використання даних, наданих операндом, тобто їх поведінка. ФП та ДП також можна відтворити, дотримуючись алгебраїчного підходу.

Розглянемо вираз $1+2$. Тут числа 1 і 2 є операндами. Вони являють собою деякі числові дані. Символ $+$ є оператором, який зв'язує їх разом. Він має певну пов'язану з цим семантику, тобто додавання одного числа до іншого. Але важливо пам'ятати, що символічна структура виразу та його семантика — це дві окремі речі. Можна взяти той самий вираз і присвоїти інше значення числам 1 і 2 та символу $+$, в результаті чого семантика виразу стане іншою.

Так само алгебраїчний підхід може бути застосований до ДП:

```
val result = numbers.map(n => n * n).
```

Це можна переписати у вигляді

```
val result = numbers map (n => n * n).
```

Scala підтримує синтаксичний цукор, що дозволяє викликати методи як оператори інфіксної нотації. Тут функція `map` є оператором, `numbers` — операнд, який є колекцією, `(n => n*n)` — операнд, який є лямбда-функцією.

Поведінка програми визначається як послідовність операторів, які зв'язують свої операнди разом. В той же час, виконання програми означає розрахунок деякого кінцевого значення, а не безпосереднє виконання алгоритму. Перевагою тут є відсутність зміни стану змінної.

Розглянемо реалізацію функції `map` в мовах програмування Java та Scala. Для Java алгоритм може бути такий:

1. Взяти перший елемент колекції.
2. Застосувати деяку функцію до цього елемента.
3. Вставити результуючий елемент в нову колекцію.
4. Повторити 1—3 для наступних елементів колекції.

Для Scala потрібно вказати результат як зв'язку двох операторів, колекції та лямбда-функції за допомогою карти операторів. Для цього достатньо записати бажаний математичний вираз і залишити визначення семантики виразу самій мові програмування Scala.

Програму, записану на Scala, можна розглядати як парадигму символічного визначення того, що потрібно зробити. Ця парадигма відокремлює семантику від структури запропонованої програми, де структура — це символи, колекції представлені символами, лямбда-функції, метод `map` як оператор; семантика — це дії, що виконуються комп'ютером під час обробки цього виразу, тобто те, як комп'ютер інтерпретує вираз.

Таким чином, програми можна розглядати як математичні вирази та працювати з ними як зі структурами даних – символічними структурами виразів. Це основна відмінність від алгоритмічного підходу, що є популярним у ІІІ як у прикладі з кодом, реалізованим за допомогою мови програмування Java. В результаті отримуємо наступне:

1. Легше використовувати символічну логіку для опису програм, які визначаються як математичні вирази, а не як алгоритми.
2. У ДП відсутній часовий вимір як такий, що дозволяє уникнути помилок, пов'язаний з часовою абстракцією.
3. Використовується принцип абстракції.
4. Визначення програми є чистим, тобто методи описують та використовують чисті функції.

Розглянемо такі специфічні структури даних, характерні для зв'язки ФП + ДП + АП, як `Try` та `Option`. Функціональні мови, такі як Scala, Haskell та інші, визначають структури даних, які не мають побічних ефектів. Можна називати їх типами ефектів.

`Try` може містити одне з двох типів значень: 1 — довільне значення деякого типу `A`; 2 — виняток. `Try` гарантує повернення результату обчислень, яке може призвести до помилки. Отже, не потрібно створювати окремий об'єкт винятку, достатньо повернути `Try [A]` у випадку, коли метод може призвести до помилки. `Scala Try[A]` є параметризованим об'єктом типу `A`.

Розглянемо функцію ділення, що є невизначеною, якщо знаменник нульовий (рис. 9). В разі виклику методу з нульовим знаменником резуль-

```
def division(n1: Double, n2: Double): Double =  
    if (n2 == 0) throw new RuntimeException("Division by zero!")  
    else n1 / n2
```

Рис. 9. Функція ділення

```
def pureDivision(n1: Double, n2: Double): Try[Double] =  
    Try { division(n1, n2) }  
println(pureDivision(1, 0)) //  
Failure(java.lang.RuntimeException:  
    Division by zero!)
```

Рис. 10. Реалізація функції ділення з використанням структури даних Try

```
def getUsername(id: Int): Option[String] =  
    if (Set(1, 2, 3).contains(id)) Some(s"User-$id")  
    else None
```

Рис. 11. Визначення функції, що повертає ім'я користувача за унікальним ідентифікатором

```
getUsername(1) match { // "User-1"  
    case Some(x) => println(x)  
    case None => println("User not found")  
}  
  
a  
  
getUsername(10) match { // "User not found"  
    case Some(x) => println(x)  
    case None => println("User not found")  
}  
  
б
```

Рис. 12. Застосування патерну порівняння у результаті обчислення функції getUsername з коректним (а) та некоректним (б) вхідним аргументом

татом буде помилка — побічний ефект. В цей момент програма вийде з ладу, чого можна уникнути, використавши Try (рис. 10).

Результат визначає роботу методу на всьому спектрі вхідних даних, тобто програма в цьому випадку не виходить з ладу. Бачимо, як відтворено різні типи ефектів в контексті структур даних, де явища і є даними. Зображення явищ у вигляді даних називається реїфікацією [2, с.72]. Ця концепція важлива для ФП.

Option може містити деяке значення або бути порожнім. Аналогом Option в ІП на прикладі Java та C++ є поняття нульового посилання. Деякі методи в ході обчислень можуть повертати значення null, чого можна уникнути, використовуючи Option[A] по аналогії з Try[A].

Розглянемо функцію, що повертає ім'я користувача за його ідентифікатором. Можливий сценарій, коли повернути ім'я не можна через відсутність ідентифікатора (рис. 11). Якщо ідентифікатор користувача відповідає числу 1, 2 або 3, вважаємо, що користувач присутній у базі даних. Задана функція явно включає інформацію про те, чи присутній користувач як результуючий об'єкт типу `Option[String]`, тобто повертається ім'я користувача та інформація про те, чи він присутній в базі даних. Таке відтворення даних уможливорює використання патернів порівняння у результаті обчислення (рис. 12).

`Option` спочатку аналізується з оператором, що відповідає деякому шаблону, перед фіналізацією обчислення. На рис. 12, а, отримано очікуваний результат «User-1», а рис. 12, б, свідчить про відсутність даного користувача, тому що результатом попереднього обчислення є відсутність користувача у базі даних. Таким чином, можна розділити логіку виконання програми та унеможливити будь які помилки, пов'язані з побічними ефектами.

Висновки

Розглянуті характерні для ФПП структури даних `Try` та `Option` дозволяють зберегти прозорість посилань та запобігти побічним ефектам. Наведені ідеї, рекомендації та підходи до функціонального програмування є корисними при створенні надійних та гнучких програмних продуктів та підвищенні ефективності роботи програмістів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Chiusano P., Bjarnason R. Functional Programming in Scala. [Електронний ресурс]. Режим доступу: <https://www.manning.com/books/functional-programming-in-scala>.
2. Kmetiuk A. Mastering Functional Programming. Birmingham: Packt Publishing Ltd, 2018, 501 p.
3. Volpe G. Practical FP in Scala. [Електронний ресурс]. Режим доступу: <https://leanpub.com/pfp-scala>
4. Adam R. Essential Effects. [Електронний ресурс]. Режим доступу: <https://essentialeffects.dev/>
5. Halliday S. Functional Programming for Mortals with Scalaz. [Електронний ресурс]. Режим доступу: <https://leanpub.com/fpmortals>
6. The Java® Virtual Machine Specification. [Електронний ресурс]. Режим доступу: <https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>.

Отримано 01.10.21

REFERENCES

1. Chiusano, P. and Bjarnason, R. (2014), Functional Programming in Scala, Manning, available at: <https://www.manning.com/books/functional-programming-in-scala>.
2. Kmetiuk, A. (2018), Mastering Functional Programming, Packt Publishing, Birmingham, England.
3. Volpe, G. (2020), Practical FP in Scala, available at: <https://leanpub.com/pfp-scala>.
4. Adam, R. (2021), Essential Effects, available at: <https://essentialeffects.dev/>
5. Halliday, S. (2018), Functional Programming for Mortals with Scalaz, available at: <https://leanpub.com/fpmortals>.
6. The Java® Virtual Machine Specification, available at: <https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>.

Received 01.10.21

I.V. Puchko, A.M. Prymushko, H.O. Kravtsov

DEVELOPMENT OF METHODOICAL RECOMMENDATIONS USAGE OF FUNCTIONAL PARADIGM PROGRAMMING IN SCALA

Different approaches to defining data structures and creating programs in different ones are considered programming paradigms. The dependence of the imperative programming paradigm on low-level algorithms and operations. Ideas for the main variables of data structures are given and operations on their mutation. Approaches to the organization of algorithms in the programming language are described in Scala using these data structures. It is shown that in the declarative paradigm programming focus shifts from algorithms to mathematical expressions, and demonstrates how to use high – level operations to create a program for using algebraic expressions.

Keywords: Scala, functional programming paradigm, imperative programming paradigm, declarative programming paradigm, lambda functions, anonymous functions, higher order functions.

ПУЧКО Іван Вікторович, провідний інженер програмного забезпечення в компанії IN-SCALE. В 2008 р. закінчив Національний авіаційний університет. Область наукових досліджень — масштабування розподілених обчислень у високонавантажених системах, програмування.

ПРИМУШКО Арсендій Миколайович, інженер програмного забезпечення в компанії Intellias. В 2020 р. закінчив Національний технічний університет України «Київський політехнічний інститут ім. Ігоря Сікорського». Область наукових досліджень — штучний інтелект, розподілені обчислювальні системи, програмування, системи орієнтації і навігації.

КРАВЦОВ Григорій Олексійович, канд. тех. наук, докторант Інституту проблем моделювання в енергетиці ім. Г.Є. Пухова НАН України. В 2000 р. закінчив Севастопольський військово-морський інститут ім. П.С. Нахімова. Область наукових досліджень — кібербезпека смарт-грид, криптографія, програмування, розробка розподілених гетерогенних обчислювальних систем.