УДК 004.424 http://doi.org/10.15407/pp2024.02-03.116

Р.С. Шевченко

ПРЕДСТАВЛЕННЯ МОНАДИЧНИХ ЕФЕКТІВ У НЕМОНАДИЧНІЙ ФОРМІ

Сучасне програмування значною мірою покладається на системи ефектів. У контексті розробки мов програмування виокремлюються два підходи до розуміння ефектів: перший визнає ефект як характеристику програми, яка впливає на середовище виконання, відокремлюючи її від простого обчислення (тобто ефекти не враховуються в системі типів мови, що є типовим для імперативних мов програмування); другий розглядає ефект як аспект інтерпретації програми, який впливає на процес її інтерпретації (такі ефекти враховуються в системі типів як типи вищого порядку, і цей підхід характерний для функціонального програмування). В індустріальному програмуванні перевагу надають першому підходу через його ефективність у швидкій розробці та меншій складності концепцій. Проте це призводить до втрати можливості автоматичного аналізу ефектів за допомогою систем типів та збільшує складність знаходження помилок. Монади є зручним інструментом для опису ефектів, оскільки вони мають вбудовану операцію композиції обчислень і можуть занурювати значення у монадичне середовище. Пряме контекстне представлення ефектів може бути корисним для розробників програм, оскільки це дозволяє знизити когнітивне навантаження від синтаксичного шуму і водночас зберігає інформацію про ефекти у типах даних. Таке представлення також дозволяє створювати крос-платформові програми, які можуть використовувати як монадичні, так і немонадичні системи ефектів. У статті розглядається проблематика створення ергономічного інтерфейсу для роботи з монадичними ефектами, які включають логіку обчислень та пов'язані операції у монаді системи ефектів, а також описується пряме контекстне представлення ефектів. Перетворення прямого контекстного представлення на монадичну форму реалізовано через плагін компілятора Scala, що доступний у вигляді відкритого коду. Також розглядається використання умовної компіляції ефектів для створення крос-платформових інтерфейсів, що об'єднують різні методи реалізації ефектів на різних платформах.

Ключові слова: ефекти, монади, Scala, мови програмування, перетворення програм.

R.S. Shevchenko

REPRESENTATION OF MONADIC EFFECTS IN THE NON-MONADIC FORM

Modern programming relies heavily on effects systems. In the context of the development of programming languages, two approaches to understanding effects are distinguished: the first recognizes an effect as a characteristic of a program that affects the execution environment, separating it from a simple calculation (that is, effects are not taken into account in the system of language types, which is typical for imperative programming languages); the second considers an effect as an aspect of program interpretation that affects the process of its interpretation (such effects are considered in the type system as higher-order types, and this approach is characteristic of functional programming). In industrial programming, the first approach is preferred because of its efficiency in rapid development and less complex concepts. However, this leads to the loss of the possibility of automatic analysis of effects using type systems and increases the difficulty of finding errors. Monads are a convenient tool for describing effects because they have a built-in computation composition operation and can sink values into the monadic environment. A direct contextual representation of effects can be useful for application developers because it reduces cognitive load from syntactic noise while preserving information about effects in data types. This representation also allows for cross-platform applications that can use both monadic and non-monadic effect systems. The paper describes the ergonomic programming language interface for working with monadic effects, which encapsulates the logic of computation and associated non-computational operations in the effect system monad, and describes the direct context encoding technique. The translation of the direct context encoding into the monadic form is implemented in the form of a Scala compiler plugin, which is available as an open source. The use of conditional effects compilation to organize cross-platform interfaces that combine different methods of implementing effects on different platforms is also discussed.

Key words: effects, monads, Scala, programing language, program transformation.

Вступ

Системи ефектів зайняли важливе місце у сучасному програмуванні. Водночас у розробці мов програмування існують два погляди на ефекти:

– Ефект - це властивість програми, що продукує зміни у середовище виконання, відмінні від суто обчислень. Ефекти не відображаються у системі типів мови. Цей погляд характерний для імперативних мов програмування;

– Ефект - це властивість інтерпретації програми, що продукує зміни у процесі інтерпретації. Ефекти відображаються у системи типів як типи вищого порядку. Цей погляд характерний для функціонального програмування.

В індустріальному програмуванні перший погляд виграє з точки зору швидкості розробки та меншої кількості опанування концепцій, проте ціною є неможливість автоматичного аналізу ефектів за допомогою системи типів і більша трудомісткість знаходження помилок. Зрештою, ці поняття все одно з'являються у вигляді опису exception-free коду та засобів аналізу, але вивчати їх доводиться пізніше.

Функціональний підхід зазвичай представляється виразом з ефектами, як монадичний вираз F[T], де T — тип виразу, який був би без явних ефектів у імперативній мові програмування.

Монади є зручним інструментом для опису ефектів завдяки тому, що на них за визначенням існує операція композиції обчислень та занурення значення у монадичне середовище. Сигнатура цих операцій мовою Scala, яку ми будемо використовувати у прикладах, виглядає як:

```
flatMap[A,B](fa: F[A])(f: A=>F[B]):F[B]
```

та

```
pure[A](a:A): F[A].
```

У літературі з функціонального програмування часто використовується стандартний монадичний тайпклас для мови Haskell, де ця операція композиції обчислень виглядає як

```
bind: F A -> (A->F B) -> F B,
```

а занурення у середовище як

```
unit: A -> F A.
```

Для детального опису властивостей монадичних інтерфейсів рекомендуємо звернутись до статей Moddy [1] та Wadler [2, 3].

Класичними прикладами монадичних ефектів є операції вводу/виводу, що як правило, відображаються як IO[X], або аварійне виключення Try [X].

Інтерпретатор монади ефекту це функція, що перетворює монадичний вираз на його значення та виконує ефекти. Записати тип можливо у метанотації, що включає до себе також значення середовища.

Тепер перейдемо від одиночних ефектів до систем ефектів. Нехай ми розробляємо програму, що робить одночасно і ввід/вивід, і може аварійно завершитися. Який тип вона може мати? Це не IO[T] і не Try [T]. Це якась монада, яка включає в себе обидва ефекти. Існує багато способів побудови таких монад, наведемо найбільш відомі.

Рішення, яке ще не можна назвати повноцінною системою ефектів, але часто використовується на практиці — це так звана "монада-бог" ("god monad"). Тобто можна побудувати монаду, що буде поєднувати в собі і обробку вводу/виводу, і генерацію помилок. Якщо кількість ефектів, що використовуються, невелика, то такий підхід досить зручний. Проте, на відміну від повноцінних систем ефектів, там неможливо додати новий ефект або замінити імплементацію існуючого без переписування інтерпретатора монади.

Історично перший повноцінний стек ефектів був побудований на основі монадних трансформерів [4, 5]. Основна ідея досить проста — логічним типом для результату програми, що має ввід/вивід та може генерувати виключення є IO[Try[T]]. Ми не можемо використовувати IO[T]

прямо, але можемо переписати IO у вигляді IOT[F[_],T]. Інтерпретатор цієї монади перетворює IOT[F[_],T] у T, використовуючи інтерпретатор F. (Або у деяких випадках можна побудувати частковий інтерпретатор, який можна вже називати обробником ефекту, що перетворить IO[F[_],T] у F[T]. А далі ми, вже маючи інтерпретатор F[T], можемо побудувати повний інтерпретатор складної монади.) Така схема проста та інтуїтивно зрозуміла. Цей підхід використовується у індустріальному функціональному програмуванні, коли в нас є один або два ефекти, що є досить поширеним випадком. Але в роботі з великим списком ефектів виникають проблеми продуктивності, оскільки кожна операція має пройти через низку викликів для кожної монади у стеку.

Наступним кроком є використання горизонтальної композиції замість вкладення, тобто розділення "монади, що містить ефекти" та суто ефектів. Тобто, наприклад, є типи ефектів $E_1[*], ..., E_n[*]$ і можливості мови програмування дозволяють нам сконструювати список рівня типів $L = (E_1, ..., E_n)$. Тоді можна побудувати монаду $Eff(X) = F[(E_1, ..., E_n), X]$ разом з інтерпретаторами ефектів, що мають вид:

$$IE_k : F[L, X] \Rightarrow F[L - E_k, X],$$

де $L - E_k = (E_1 \ldots E_{k-1}, E_{k+1} \ldots E_n)$ — список типів ефектів L без E_k. Самі типи ефектів E_i не обов'язково повинні бути монадами.

Класичний приклад конструкції такого типу — це Freerer монада [6]. Зараз у контексті індустріального програмування, коли кажуть про монадичну систему ефектів, то загалом мають на увазі подібну конструкцію з низкою оптимізацій. Для Haskell існують близько десяти реалізацій, для Scala зараз у розробці декілька систем, такі як Eff [7], Kryo [8], Turbolift [9].

Ще слід згадати так званий безтеговий кінцевий (tagless final) стиль опису [11], де до ефектів звертаються не як конкретних типів, а як до тайпкласів, що реалізують деяке API [10]. Дійсно, єдине, що характеризує ефект, — це набір методів, як от, гіпотетичний ефект вводу/виводу може характеризуватись парою функцій

```
trait StdIO[X] {

 def putStrLn(line: String): StdIO[X]

 def getStrLn: StdIO[String]

}
```

Якщо ми можемо "проштовхнути" ці функції у основну монаду, то це API можна записати як

```
trait StdIO[F[_]] {

 def putStrLn(line: String): F[X]

 def getStrLn: F[String]

}
```

А потім використовувати цей тайпклас як характеристику монади ефектів, незалежно від того, як ця монада побудована — як самостійна монада ефекту або елемент монадного трансформера або елемент системи ефектів.

Тепер можемо сформулювати задачу. В нас є два методи організації ефектів: чи можемо ми сумістити ці дві форми так, щоб було можливо використовувати монадичні ефекти без ускладнення синтаксису, водночас залишаючи можливість аналізу виразів з ефектами за допомогою систем типів?

Ця задача має два прикладних застосування. Перше — це полегшення роботи з монадними інтерфейсами для розробників, друге — мультиплатформові системи з різними можливостями рантайму на різних платформах. Тоді ефекти, що можуть бути представлені у прямій формі на одній платформі (наприклад, асинхронність на post-Loom JVM), мають бути монадичними на платформі без підтримки продовжень (continuations), таких як Javascript.

Прямий синтаксис

Методи полегшення синтаксичного навантаження з'явились одночасно з монадичними інтерфейсами. Підходи можна розбити на дві групи:

– спеціальні конструкції, що створюють “підмову” мови програмування для монадних DSL;

– встановлення псевдооператорів, що перетворюють звичайні конструкції мови у монадичну форму.

Прикладом першого підходу є do-notation у Haskell [11], Computation Expressions у C# [12] та for-comprehansions у Scala [13]. З одного боку це зручно у реалізації, адже ці синтаксичні конструкції компілятор може перетворювати до аналізу типів, з іншого — користувач бачить дві схожі, але різні мови, що спричиняє додаткові складнощі під час вивчення та застосування.

Прикладом другого підходу є bing notation в Idris [14] та async/await (щоправда обмежено асинхронністю) у C# та більшості сучасних мов програмування [15]. Автором було розроблено подібну систему для Scala: dotty-cps-async [16–18], що підтримує пару псевдооператорів reify/reflect (або async/await як традиційні синоніми) для будь-якої монади як макроси. Практично це рішення вирішує питання зменшення когнітивного навантаження у застосуванні асинхронних API та “пом’якшення” кривої навчання. Проте, у процесі роботи з монадичними ефектами виникають наступні складності.

Оскільки у системах ефектів більшість операцій монадичні, вихідний код все ж таки перевантажений операторами reflect (або await), що не несуть нетехнічного змісту. Так, наприклад, на Рис. 1 ми бачимо “await” майже у кожному рядку.

Друга проблема, специфічна для систем ефектів. У наведеному вище прикладі ми працюємо з однією монадою IO, яку вказуємо як параметр типу async[IO]. Якщо ми працюємо з системами ефектів, то там тип монади складний і async вираз буде мати вигляд

```
async[[X]=>

      Eff[IO::*Error::*Config,X]]
```

Програміст мусить написати цю повну тайп-сигнатуру, що інколи змінюється залежно від того, які ефекти використовуються. Але макрос може зібрати типи ефектів, що використовуються, і на основі цього вивести повний тип ефекту.

Для вирішення питання перевантаження async-виразами, в перших версіях dotty-cps-async був запропонований режим автоматичного розфарбовування, що заснований на неявних перетвореннях. Ми дозволяємо неявне перетворення між IO[T] та T, якщо єдине використання асинхронного виразу це взяття або ігнорування значення. Якщо вираз використовується по-іншому (наприклад, передається в іншу функцію як IO), то ми повідомляємо про помилку. Приклад використання автоматичного розфарбовування показано на Рис. 2.

```
def myFun(using RunContext): IO[HttpReply] =
async[IO] {
 val results1 = await(talkToServer("request1",
None))
 await(IO.sleep(100.millis))
 val results2 = await(talkToServer("request2",
Some(results1.data)))
 if results2.isOk then
    await(writeToFile(results2.data))
    await(IO.println("done"))
 else
    await(IO.println("abort"))
 results2
}
```

Рис. 1. Використання dotty-cps-async без автоматичного розфарбовування

```
def myFun(using RunContext): IO[HttpReply] =
async[IO] {
 val results1 = talkToServer("request1", None)
 IO.sleep(100.millis)
 val results2 = await(talkToServer("request2",
Some(results1.data)))
 if results2.isOk then
    writeToFile(results2.data)
    IO.println("done")
 else
    IO.println("abort")
 results2
}
```

Рис. 2. Використання dotty-cps-async з автоматичним розфарбовуванням

Як бачимо, візуально деяке покращення є, але це рішення не набуло широкого розповсюдження, оскільки аналіз такого коду дещо складніший (скажімо, чому біля result1 немає await а біля result2 є ?), і для того, щоб самостійно розуміти, які типи виводяться, потрібно крім правил мови ще враховувати контекст. Незважаючи на те, що макрос проводить аналіз типів і генерує помилку під час (можливо) некоректного використання, такі перетворення сприймалися розробниками як небезпечні.

Наразі на зміну режиму автоматичного розфарбовування розроблений плагін компілятора, що дозволяє описувати монадичні обчислення у прямому стилі за допомогою контекстних аргументів і контекстних функцій.

Контекстний аргумент (у Scala 2 він називався неявним аргументом) забезпечує можливість автоматично сконструювати і передати аргумент функції, не відображаючи це в коді програми. Списки аргументів функцій та методів можуть маркуватись як using і компілятор у процесі генерування виклику функції буде шукати значення цих параметрів у контексті. Наприклад, наступне визначення:

```
def write[A](a:A)
       (using Writer[A]): Unit
```

визначає функцію з контекстним параметром Writer[A], і ми можемо викликати її як write(10).

Контекстні функції (A ?=> B) були введені у Scala 3 [13] і означають обчислення, що залежать від контексту A та повертають B. Корисною для нас особливістю є короткий синтаксис контекстних лямбда-функцій: в метод, що приймає на вхід контекстну функцію A ?=> B передати вираз типу B, який містить виклики функцій з контекстним аргументом, що можуть бути виведені з A. Наприклад, нехай у нас є якийсь об'єкт, з якого ми можемо вивести Writer для елементарних типів:

```
trait Writers {
 given Writer[Int] ...
 given Writer[String] ...
}
```

і функція

```
def withJsonWriters[A]
        (f: Writes ?=> A): A
```

Тоді виклик такої функції може мати форму:

```
  withJsonWriters {
    write(1)
    write("abc")
  }
```

Повертаючись до монадичних ефектів, ми можемо представити монадичне обчислення F[X] як контекстну функцію,

що повертає X: CpsDirect[F] ?=> X. Використовуючи описані вище властивості контекстних функцій, ми можемо вільно використовувати результати асинхронних викликів у межах видимості CpsDirect[F] як значення типу X.

Плагін компілятору перетворює функції та методи, що містять контекстний параметр CpsDirect[F] та повертають значення типу X, у методи, що повертають F[X] і конвертують тіло цих методів у монадичну форму. Попередній приклад з Рис. 1 та Рис. 2 тепер можна записати, як показано на Рис. 3.

```
def myFun(using RunContext, CpsDirect[IO]):
HttpReply = {
 val results1 = talkToServer("request1", None)
 sleep(100.millis)
 val results2 = talkToServer("request2",
Some(results1.data))
 if results2.isOk then
    writeToFile(results2.data)
    println("done")
 else
    println("abort")
 results2
}
```

Рис. 3. Використання dotty-cps-async: пряме контекстне представлення

Тут ми неявно вважаємо, що API, яке використовується, також конвертоване у пряме контекстне представлення.

Ми можемо викликати функції із CpsDirect[F] параметром з async блоків і інших прямих контекстних функцій, а також вільно використовувати псевдооператор await/reflect для перетворення F[T] у T за необхідності. Для забезпечення коректності перетворень ми накладаємо ряд обмежень на вирази типу CpsDirect[F] — їх не можна використовувати в операторах присвоювання значення, вихідних значень функцій та зіставляти з параметрами типів. Можливо, у майбутньому ці обмеження можна буде виразити за допомогою відстежування захоплень (capture tracking), що зараз розробляється як експериментальне розширення Scala [19].

Отримана модель асинхронності на практиці досить близька до моделі призупинених функцій Kotlin [20].

Зазначимо, що на відміну від режиму “автоматичного розфарбовування” функції, тут типи F[X] та CpsDirect[F] ?=> X є різними типами для програміста, і для кожної функції нам потрібно обирати, як краще представити асинхронне значення — у контекстній чи монадичній формі? Тобто проблема розфарбовування нікуди не зникла: просто з’явилась можливість змінити основний метод виклику на квазісинхронний.

Монадичні інтерфейси можуть використовуватись поряд із прямими. Розглянемо приклад, коли монадичний інтерфейс дійсно потрібен. Наприклад, нехай нам потрібно паралельно прочитати декілька потоків даних. Якщо припустити, що fetch також записана у прямій формі, то наступний блок коду -

```
def readFirstN(urls: Seq[String])
 (using CpsDirect[Future]):Seq[String
 =
   urls.map(url => fetch(url))
```

- прочитає усі дані послідовно, один за одним. Якщо нам потрібна паралельна

обробка, то ми маємо перевести пряме API у монадичну форму:

```
def readFirstN(urls: Seq[String])
 (using CpsDirect[Future]:Seq[String]
 =
   urls.map(url =>
       asynchronized(fetch(url))
       ).map{
        future => await(future)
       }
```

Тому для прямого синтаксису введений псевдооператор asynchronized (або reifeied), що перетворює псевдосинхронний вираз на монадичний, дуальний до await.

Для випадку, коли F[_] — складна монада ефектів, можна розробити короткий синтаксис вказання набору ефектів у F як набору контекстів або обмежень типу F. Одним з варіантів є абстракція від конкретної системи ефектів F у формі tagless final Оптимальний вигляд такого запису — відкрите питання, що є темою подальших досліджень.

Сумісність з немонадичними системами ефектів

Ще одна типова проблема розробки — це підтримка міжплатформових бібліотек, де середовище виконання має різні властивості на різних платформах. Зокрема, на платформі Java, починаючи з версії 21, стало можливо розробляти системи ефектів [21], засновані на продовженнях (continuations), тоді як подібне API в немонадичному вигляді неможливо у scala-js. Чи можливо побудувати такий режим компіляції, щоб один і той же код був сумісний з бібліотекою асинхронного програмування в JVM, заснованій на віртуальних потоках, та системі, заснованій на асинхронній моделі конкурентності Javascript? Основною проблемою є те, що немонадичні ефекти зараз ніяк не відображаються у системи типів, тому зі сторони системи з продовженнями, в нас немає інформації для коректної трансляції. Отже, ми не можемо перенести код з системи, що ґрунтується на продовженнях, у монадичну. Але можемо навпаки: у монадичній системі типи мають всю необхідну інформацію. Отже, можна записати API системи з продовженнями як пряме представлення монадичної форми, потім відкомпілювати його у Javascript як монадичну форму, а в JVM — просто стираючи контекстний параметр (тобто плагін повинен перетворювати CpsDirect[F] ?=> X в F[X] на Javascript, та в X на JVM для ефектів, що в JVM реалізовані через продовження). Це дозволить програмам використовувати одне й те ж базове API на обох платформах.

Висновки

Пряме контекстне представлення ефектів може бути корисним інструментом при розробці програм, оскільки знижує когнітивне навантаження на програміста від синтаксичного шуму і в той же час зберігає в типах інформацію про ефекти. Також за допомогою такого представлення можна будувати крос-платформові програми, що можуть використовувати як монадичні, так і немонадичні системи ефектів.

У проєкті dotty-cps-async розроблено плагін компілятора Scala, що перетворює пряме контекстне представлення на монадичну форму. Подальші напрямки досліджень включають пошук оптимальної форми підтримки складних систем ефектів типу Eff, продовження експериментів у напрямку крос-платформової трансляції ефектів та вдосконалення генерації монадичного представлення програми.

References

1. E. Moggi, Notions of computation and monads, in: Information and Computation, 93 1 (1991) 55–92. doi:10.1016/0890-5401(91)90052-4.
2. P. Wadler, Monads for Functional Programming, in: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial, Berlin, Heidelberg, Springer-Verlag, 1995, pp. 24–52.
3. P. Wadler, The essence of functional programming, in: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages

(POPL'92), 1992, New York, ACM, pp. 1–14. https://doi.org/10.1145/143165.143169

4. S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'95), 1995, New York, ACM, pp. 333–343. https://doi.org/10.1145/199448.199528
5. T. Schrijvers, M. Piróg, N. Wu, M. Jaskelioff, Monad transformers and modular algebraic effects: what binds them together, in: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019), 2019, New York, ACM, pp. 98–113. https://doi.org/10.1145/3331545.334259
6. O. Kiselyov, H. Ishii, Freer monads, more extensible effects. SIGPLAN Not. 50, 12 (2015) 94–105. https://doi.org/10.1145/2887747.2804319
7. eff: [cited 08.04.2024] https://github.com/atnos-org/eff
8. kyo: [cited 08.04.2024] https://github.com/getkyo/kyo
9. turbolift [cited 08.04.2024] https://marcinzh.github.io/turbolift/
10. Carette, J., Kiselyov, O., Shan, C.-C., Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages, in: Journal of Functional Programming, 19 5 (2009) 509–543. doi:10.1017/S0956796809007205.
11. P. Hudak, J. Hughes, S. P. Jones, P. Wadler, A history of Haskell: being lazy with class, in: Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III), New York, ACM, 2007, pp. 12–1–12–55. https://doi.org/10.1145/1238844.1238856
12. T. Petricek, D. Syme, The F# Computation Expression Zoo, in: Flatt, M., Guo, HF. (eds) Practical Aspects of Declarative Languages. PADL 2014. Lecture Notes in Computer Science, vol 8324. Cham, Springer, 2014, https://doi.org/10.1007/978-3-319-04132-2_3
13. Scala 3 Language Reference. cited 08.04.2024 https://docs.scala-lang.org/scala3/reference/
14. Idris Language Reference. cited 08.04.2024 https://docs.idris-lang.org/en/latest/reference/index.html
15. C# Asyncronious Programming Scenarious. cided 08.04.2024. https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios
16. R. Shevchenko, Project Paper: Embedding Generic Monadic Transformer into Scala, in: Lecture Notes in Computer Science, 3401 (2022). https://doi.org/10.1007/978-3-031-21314-4_1
17. dotty-cps-async. Retrieved 09.04.2024 https://github.com/rssh/dotty-cps-async
18. R.S. Shevchenko, A.Yu. Doroshenko, O.A. Yatsenko. Embedding a family of logic languages with custom monadic unification in Scala, in: Problems in programming 1 (2024) 3–11. [in Ukrainian]
19. A. Boruch-Gruszecki, M. Odersky, E. Lee, O. Lhoták, J. Brachthäuser, Capturing Types, in: ACM Trans. Program. Lang. Syst. 45, 4, Article 21 (2023) 52 p. https://doi.org/10.1145/3618003
20. Koltin coroutines guide. Retrieved 09.04.2024 https://kotlinlang.org/docs/coroutines-guide.html
21. Gears. An Async library for Scala. (cited 09.04.2024). https://github.com/lampepfl/gears

Одержано: 10.04.2024
Внутрішня рецензія отримана: 18.04.2024
Зовнішня рецензія отримана: 27.04.2024

Про автора:

[1]*Шевченко Руслан Сергійович*,
кандидат технічних наук
старший науковий співробітник.
https://orcid.org/0000-0002-1554-2019

Місце роботи автора:

[1]Інститут програмних систем
НАН України,
тел. +38-067-407-32-33
E-mail: ruslan@shevchenko.kiev.ua
www.iss.nas.gov.ua