УДК 004.424 http://doi.org/10.15407/pp2024.01.003

Р.С. Шевченко, А.Ю. Дорошенко, О.А. Яценко

ВБУДУВАННЯ СІМЕЙСТВА ЛОГІЧНИХ МОВ ІЗ МОЖЛИВОСТЯМИ ПЕРЕПРОГРАМУВАННЯ МОНАДИЧНОЇ УНІФІКАЦІЇ В SCALA

У статті запропонована структура для вбудовування методів логічного програмування та програмування в обмеженнях у мову Scala шляхом побудови логічної предметно-орієнтованої мови навколо уніфікації типізованої логіки на основі монад. Відмінності в можливостях логічних механізмів можна виразити як підкласи монади уніфікації. Такий спосіб дає змогу генерувати одну реалізацію налаштовуваної уніфікації для вбудовування різних логічних систем у Scala та використовувати вбудовані сторонні проблемно-орієнтовані мови у логічних виразах. Монадичний прикладний програмний інтерфейс надає розробнику програми простий та інтуїтивно зрозумілий інструмент для реалізації власної логіки всередині уніфікації.

Ключові слова: вбудовування, декларативне програмування, логічне програмування, монада, предметно-орієнтована мова, терм, уніфікація, Scala.

Вступ

Декларативне програмування є важливим елементом інженерії програмування. Сьогодні більшість декларативних мов є переважно галузевими техніками, причому в більшості областей логічне програмування загального призначення, як правило, не застосовується. Деякі задачі, такі як взаємодія з користувачами або інтеграція з іншими програмами, завжди будуть поза формалізацією, тому що суть такої роботи — побічний ефект, який краще описати операційно. Проте найпоширенішим підходом є гібридний, коли розробники використовують високорівневі методи для однієї частини системи та операційні — для інших. Для цього потрібна підтримка базової інфраструктури, зокрема:

– незалежні від мови моделі сутностей;

– вбудовування декларативних методів у недекларативні мови.

Перший випадок використовується з архітектурою мікросервісів, коли ми можемо розділити велику систему на грубозернисті підсистеми, де кожна підсистема має бажаний стиль. Однак поділ системи на мікросервіси, як правило, здійснюється не за технологіями, а за бізнес-ознаками, тому використання архітектури на основі мікросервісів часто є більш організаційним, аніж технічним рішенням.

Другий випадок — вбудовування елементів декларативного програмування в операційну семантику. Тут ми можемо побачити в ландшафті нові мультипарадигмальні мови програмування, такі як Flix [1, 2], що поєднують логічне, функціональне та імперативне програмування, або ж Logtalk [3], який додає об’єктну орієнтацію до мови Prolog та більшу тенденцію до вбудовування декларативних методів програмування в існуючі мови. Декларативна програма представлена тут як об’єкт у системі об’єктів основною мовою, зазвичай створений у певного виду вбудованій предметно-орієнтованій мові (Embedded Domain-Specific Language, EDSL). Сьогодні це поширена методика. TermWare [4, 5] вбудовує програмування на основі правил у Java, 2P-Kt [6] вбудовує Prolog у Kotlin, а Byods [7] є прикладом вбудовування системи Datalog у Rust як процедурного макросу.

Однією з поширених проблем вбудовування мови є різниця між логічною та основною моделями об’єктів. У логіці ми зазвичай працюємо з неінтерпретованими функціями, тоді як нетипізоване представлення є рідкістю в об’єктно-орієнтованих мовах зі статистично типізованими функціями. Інша проблема полягає в організації двонаправленого обміну даних під час виконання логічних програм. Логічна части-

на повинна мати можливість викликати імперативну частину і навпаки.

У даній роботі представлено структуру для вбудовування методів логічного програмування та програмування в обмеженнях у мову Scala шляхом побудови логічної DSL навколо уніфікації типізованої логіки.

1. Опис логічної структури

У даному розділі неформально описано частину необхідної логічної структури, яку потрібно вбудувати.

Базовими об'єктами в логічному програмуванні є логічні терми T, які складаються з:

- констант $c \in C$;
- логічних змінних $t \in T_i$;
- неінтерпретованих функцій $f_i(x_1,...,x_n) \in \Theta$.

Якщо у нас є певна концепція уніфікації термів (яка буде описана пізніше), ми можемо побудувати логічний вираз над термами E. Логічні вирази будуються як:

- функціональні терми $f_i(x_1,...,x_n)$, що представляють логічне значення $\Theta \subset E$;
- $True$ і $False$ є логічними константами, $True \in E$, $False \in E$;
- для $x \in E$, $y \in E$ ми можемо побудувати звичайну логічну операцію: кон'юнкція ($x \wedge y$), диз'юнкція ($x \vee y$);
- заперечення ($\neg x$) відіграє особливу роль і доступне не у всіх інтерпретаціях.

Тепер, коли у нас є логічний вираз, ми можемо побудувати логічну програму, що складається з правил, які описують факти, залежності між правилами та стратегію логічного пошуку.

Програма повинна мати форму, що залежить від основного логічного механізму. Наприклад, якщо ми хочемо емулювати семантику Prolog з алгоритмом інтерпретації SLDNF (Selective Linear Definite clause resolution with Negation as Failure — вибіркова лінійна резолюція з визначеними твердженнями із запереченням як відмовою) [8], кожен оператор має бути диз'юнктом Горна ($x :- y$), де ліва частина має бути функціональним символом або *True*, а права частина має бути кон'юнкцією логічних виразів. Крім того, диз'юнкти Горна в Пролозі можуть містити псевдотерм *cut*:

$$(x :- y_1 ... y_n) : \in R_{Prolog} \; -(x \in \Theta \wedge x = True), \quad y_i \in E \wedge y_i = cut.$$

Для Datalog у нас є диз'юнкт Горна без заперечення:

$$(x :- y_1 ... y_n) : \in R_{Datalog} \; -(x \in \Theta \wedge x = True), \quad y_i \in E \wedge y_i \neq \neg z.$$

Для програмування логіки обмежень наші функціональні символи мають бути простими фактами або виразами із сигнатур підтримуваних теорій.

2. Вбудовування логічних конструкцій у мову Scala

Далі опишемо вбудовування вищерозглянутих конструкцій у Scala. Перше питання полягає в тому, як побудувати універсальне відображення між логічними термами та об'єктами Scala, визначеними безпечним для типів способом.

Логічні терми та вирази матимуть два представлення Scala — для часу компіляції, яке розробник використовує для створення логічних програм, і часу виконання, що застосовується, коли логічний механізм викликає засоби низького рівня.

Почнемо з часу компіляції. Константи та неінтерпретовані функції (факти) відображаються в case-класи Scala. Терми з логічними змінними представлені за допомогою вбудовування HOAS [9]: представлення складного терму $f(x,y)$ є функціональним виразом Scala $(x : A, y : B) \Rightarrow f(x,y)$. Для логічних виразів використовуються відповідні оператори Scala: & або $\wedge$ для кон'юнкції; | або $\vee$ для диз'юнкції; ! для заперечення; |- для диз'юнкта Горна (|- обрано замість :- через нижчий пріоритет оператора).

Нижче наведено приклад такого вбудованого виразу:

```
case class Edge(x: Int, y: Int)
           derives Fact
case class Path(x: Int, y: Int)
           derives Fact

val transitiveClosure = Prolog {
   (x: Int, y: Int, z: Int) => {
     Path(x, y) |-  Edge(x, y)
     Path(x, z) |-  Path(x, y) &
     Edge(y, z)
   }
}
```

Тут Edge і Path — case-класи Scala. Ми можемо визначити набір фактів за допомогою того самого синтаксису,

```
val facts = Prolog{
    Path(1,2) |- true
    Path(2,3) |- true
    Path(3,4) |- true
}
```

або створити диз'юнкти Горна з колекції case-класів Scala:

```
val facts2 = Prolog.asFacts(Path(1,2),
                Path(2,3), Path(3,4))
```

Як бачимо, вбудовування зберігає коректність типізації: введення всередині диз'юнкта Горна перевіряється синтаксичним макросом, а введення Scala-констант — компілятором Scala.

Розробник програми може отримати результат, ініціалізувавши логічний інтерпретатор, що розуміє Prolog і виконує запит:

```
val results = Prolog.defaultEndinge(
   rules ++ facts).query(x => Path(1, x))
```

Низькорівневе вбудовування використовується, коли нам потрібно впровадити нашу власну поведінку уніфікації в логічну програму. Наприклад, у реальному житті факти часто потрібно отримувати зі сховищ ключ-значення або якоїсь документо-орієнтованої чи реляційної бази даних. API низького рівня дозволяє написати терм для запиту до бази даних під час інтерпретації та повернення набору даних як набору фактів у процесі інтерпретації.

Щоб написати такі плагіни, розробники повинні працювати з представленням логічних термів під час виконання.

Почнемо з логічної змінної:

```
case class LogicalVariable(id: Long,
           pos: LogicalVariablePosition)
           extends LogicalTerm
```

Ми припускаємо, що логічний механізм підтримує зв'язки між логічними змінними та ідентифікаторами, тому змінні мають лише тип даних long із доданою інформацією для налагодження.

Типізовані терми представлені об'єктом Scala, для якого можна описати алгоритм уніфікації:

```
sealed trait TypedLogicalTerm[T]
             extends LogicalTerm

case class LogicalConstant[
           T: Unificable](value: T)
           extends TypedLogicalTerm[T]

case class LogicalFunctionSymbol[T](
           unificator: Unifiable[T],
           args: IndexedSeq[LogicalTerm])
           extends TypedLogicalTerm[T]
```

Щоб вивести власний тип термів за допомогою власної уніфікації, розробнику потрібно визначити тип Scala та надати для нього реалізацію типокласу Unificable.

Сигнатура уніфікованого типокласу є такою:

```
trait Unifiable[T]  {
   def unify[R[_]:UnificationEnvironment,
             D]
       (value: T,
        term: LogicalTerm, ,
        bindings: Bindings)
        (using EngineInstanceData[D]):
        R[T]
}
```

Тут Bindings — це базована (ground) підстановка, що підтримується логічним механізмом, який надає звичайні методи роботи з підстановками.

```
trait Bindings {

   def get(logicalVariable:
       LogicalVariable):
       Option[LogicalTerm]

   def updated(logicalVariable:
       LogicalVariable,
       term: LogicalTerm): Bindings
```

```
   def bind[R[_],A,D]
       (variable: LogicalVariable,
        a:A)
       using unifiable: Unifiable[A],
       EngineData[D]): R[A]

   def bindM[R[_],A]
       (variable: LogicalVariable,
        a:R[A])
       (using unifiable: Unifiable[A]):
       R[A]

   def newVariable(): LogicalVariable
}
```

Оскільки різні механізми уніфікації мають різні можливості, тип повернення уніфікації загортається в тип R[_].

Це дає нам загальну сигнатуру для уніфікації, яку можна переносити між різними логічними механізмами. Зауважимо, що загалом різні механізми мають різну семантику уніфікації, і тип результату уніфікації також може відрізнятися.

Мінімальна уніфікація над логічними термами x і y, $unify(x, y)$, може бути описана як отримання заміни θ такої, що $x\theta = y\theta = mgu(x, y)$, де є мінімальний основний уніфікатор x і y, якщо $mgu(x, y)$ існує, інакше — невдача.

У логічному програмуванні в обмеженнях уніфікація може повертати набір можливих замін під час обробки логічної змінної, щоб ініціювати пошук у кінцевій області. Наприклад, $unify(x \le 5, x > 3)$ може повернути дві підстановки — $\{x = 4\}$ і $\{x = 5\}$.

ISO Prolog визначає можливість затримки для керування порядком кон'юнктивних тверджень, і одним із можливих результатів уніфікації може бути затримка уніфікації до моменту, коли вирішаться деякі залежності.

Ще одним аспектом є асинхронне виконання проти синхронного. Наприклад, у деяких промислових випадках краще мати уніфікацію, яка повертає набір можливих замін як асинхронний потік. Це дозволяє програмі негайно почати обробку результатів у конвеєрі, не чекаючи закінчення процесу уніфікації. Зауважимо, що цей тип асинхронного результату не можна представити за допомогою типу даних синхронного повернення.

UnificationEnvironment в Scala описує загальні можливості уніфікації, такі як синхронні чи асинхронні результати, можливість відкладати уніфікацію, доки не буде вирішено певний терм, здатність скорочувати або розгалужуватися після успіху тощо. Ми маємо відповідний клас типу, успадкований від UnificationEnvironment для кожного набору можливостей.

Уніфікація, задана користувачем, може створювати підстановки вручну або отримувати середовище з новими підстановками завдяки операціям прив'язки. Робота середовища уніфікації полягає у збереженні поточної підстановки.

Мінімальне середовище передбачає наступні методи формування результатів:

```
def success[A](a: A, bindings: Bindings):
    R[A]

def failure[A](a: A, right: LogicalTerm,
               reason): R[A]

def error[A](e: Throwable): R[A]
```

Також R[_] утворює монаду, де значення є функціями від поточних прив'язок до набору можливих майбутніх прив'язок. Тобто R[A] можна розуміти як набір усіх можливих зв'язків, на яких логічний вираз, відображений у Scala типу A, є істинним. Тому ми можемо використовувати існуючі методи для роботи з монадичними конструкціями, такі як dotty-cps-async [10].

MultiunificationContext додає можливість повертати більше, ніж один результат уніфікації. Коли підтримується мультиуніфікація, ми можемо використовувати диз'юнкції

```
def or[A](x:R[A]*): R[A]
```

Проілюструємо використання уніфікації, заданої користувачем, написавши уніфікацію для класу, що виконує запит до бази даних документів. Припустимо, що у нас є клас User:

```
case class User(id: Long, name: String,
                email: String,
                phoneNumber: String,
                score: Int)
```

Ми хочемо написати низькорівневий механізм уніфікації так, аби перевірка

фактів користувача запускала запити з бази даних. Для цього визначимо даний Unifiable для нашого користувача як показано на рис. 1. З'єднання з базою даних передається для уніфікації з механізму в дані екземпляра.

Логіка уніфікації проста — якщо ми бачимо незв'язане логічне значення, тоді отримуємо всіх користувачів і надсилаємо в систему нові зв'язки з кожним користувачем за допомогою уніфікатора фактів. (Тобто, коли система запускатиме уніфікацію над екземпляром, повернутим із бази даних, буде викликано userFact unify). На константі здійснюватиметься перевірка, чи присутня ця константа в нашій базі даних, і якщо ми маємо частково заповнений вираз, створюватимуться запит до бази даних і нові зв'язки для кожного результату.

Для виконаня цього запиту, ми повинні встановити для ConnectionProvider значення instanceData

```
val results =
   Prolog.defaultEndinge(rules).
          withData(connectionProvider).
          query(x => ...)
```

Зауважимо, що цей приклад усе ще нереалістичний: отримання всіх баз даних без індексування рідко може бути прийнятним вибором.

Щоб подолати це, функція DelayedUnification дозволяє контролювати порядок розпізнавання логічних змінних у прив'язках.

API виглядає таким чином:

```
trait DelayUnificationContext[R[_]]
      extends UnificationContext[R] {

   def waitResolved(v: LogicalTerm):
       R[LogicalTerm]
}
```

Ця операція перевіряє, чи є поточний терм логічною змінною. Якщо ж так, вона призупиняє виконання поточного уніфікованого терму та перемикає логічну машину на обробку інших виразів, розташованих у тому ж диз'юнкті Горна. Якщо цей терм отримує значення або в поточному диз'юнкті Горна не залишилося цілей, виконання уніфікації відновлюється.

```
  object User {

   val userFact = Fact.derived[User]

   given Unifiable[User] with
     override def unify[R[_] : UnificationEnvironment, ConnectionProvider](
           value: UserInDB, term: LogicalTerm, bindings:Bindings)(
              using EngineInstanceData[ConnectionProvider]): R[User] = {
       val connection = summon[EngineInstanceData[ConnectionProvider]].get.connection
       term match
         case lv:LogicalVariable =>
           // retrieve all the users from the database and
           val users:List[User] = connection.collection[User]("users").find()
           or(users.map(u=>bindings.bind(lv,u)(using userFact)):_*)
         case lc@LogicalConstant(value) =>
           connection.collection[User]("users").findOne({ "id" -> value }) match
             case Some(userInDb) => userFact.unify[R](userInDb, lc, bindings)
             case None => failure(value,term)
         case LogicalFunctionSymbol(unifiable, args) =>
           if (unifiable.checkType[User]) then
             val query = Map.empty ++ args(0).ground.map{"id" -> id }
                                   ++ args(1).ground.map{ "name" -> name }
                                   ++ args(2).ground.map{ "email" -> email }
                                   ++ args(3).ground.map{ "phoneNumber" -> phoneNumber }
             val users = connection.collection[User]("users").find(query)
             or(users.map(u=>bindings.bind(lv,u)(using userFact)):_*)
           else
             failure(value,term)
     }
  }
```

Рис. 1. Визначення Unifiable для користувача

Тому, якщо ми хочемо уніфікувати роботу користувача лише тоді, коли деякі з індексованих властивостей вирішено, ми можемо змінити наш приклад на

```
override def unify[R[_] :
  (UnificationEnvironment &&
   DelayUnificationEnvironment),
  ConnectionProvider](
    value: UserInDB,
    term: LogicalTerm,
    bindings:Bindings)(
      using EngineInstanceData
           [ConnectionProvider]): R[User]
      = reify[R]{
        waitResolved(term) match
        case lv:LogicalVariable =>
          error(RuntimeException(
                "Attempt to query all
                 users"))
        case lc@LogicalConstant(value) =>
          ... // the same logic as in
              // the previous example
     }
```

Тепер запит, який вимагатиме отримання всіх об'єктів із бази даних, не вдасться виконати. Розглянемо також складену сигнатуру класу типів для unify, && визначений як

```
type &&[A[_[_]],B[_[_]]] =
              [R[_]] =>> A[R] & B[R]

R[_]: MultyUnificationEnvironment &&
      DelayUnificationEnvironment
```

Це дозволить використовувати unify у всіх середовищах, які підтримують як multi, так і delay.

API DelayUnificationEnvironment підходить для випадків, коли факти користувача потрібні для доступу до деталей про інші події. Але що, як ми дійсно хочемо отримати довгий список результатів із бази даних?

AsyncUnificationEnvironment додає до можливостей взаємодію з асинхронними обчисленнями в монадній упаковці. Це додавання наступних методів:

```
def adoptAsync[A](value: F[A]): R[A]

def asyncResult[[A](output:
                    AsyncList[F,A]): R[A]
```

Тип F[_] — це наша асинхронна монада, яка визначається як псевдонім абстрактного типу в AsyncUnificationEnvironment, для уникнення додавання F[_] до сигнатури unify.

Робота першого методу полягає в прийнятті результату асинхронного обчислення в поточну монаду. Наявність цього методу дозволяє нам реалізувати клас типів для перетворення між монадами та використовувати reflect[F] в дужках reify[R].

Другий метод — це вихід асинхронного потоку результатів, який буде асинхронно оброблений логічною системою як набір можливих альтернатив. Розробнику слід бути обережним із отриманням потоків у базі даних, оскільки на стороні клієнта легко виконувати неефективні об'єднання. У більшості випадків краще показати, що факт витягується з баз даних основною мовою. Код, який визначає факт QueryLanguage, може виглядати, як показано на рис. 2.

```
case class User(id: Long, name: String, email: String, phoneNumber: String,
                score: Int) derives Fact

case class QueryAllUsers(u: User)

object QueryAllUsers {
 given Unifiable[QueryAllUsers] with
   override def unify[R[_] : AsyncUnificationEnvironment.Aux[IO],ConnectionProvider](
     value: UserInDB, term: LogicalTerm, bindings: Bindings,
     instanceData: InstanceData[ConnectionProvider]): R[User] = reify[R] {
     val collection = instanceData.get.connection.collection[User]("users")
     term match
       case lv: LogicalVariable =>
         reflect(collection.asyncFind().map(u => bindings.bind(lv, QueryUser(u)))
       case lc@LogicalConstant(value) =>
         ... // logic similar to the previous example
   }
 }
}
```

Рис. 2. Визначення факту QueryLanguage

Тепер ми можемо використовувати QueryAllUsers у логічній програмі та писати такі логічні правила:

```
AllUsersAreScored |- QueryAllUsers(u) &
                     AssignScore(u)
```

Приклад усе ще спрощений, наступним природним кроком є розширення нашої здатності вбудовування для перекладу логічного виразу в запити до бази даних. Цього можна досягти напрочуд легко, оскільки побудова DSL-запитів у Scala, яка відображається в набір класів, є популярною технікою. Усе, що нам потрібно зробити, — це реалізувати Unificable для класу запитів обраної бібліотеки доступу до бази даних.

Висновки

Запропонована структура для вбудовування сімейства мов декларативної логіки в Scala із уніфікацією на основі монад, що може бути перепрограмована користувачем. Відмінності в можливостях логічних механізмів можна виразити як підтипи монад уніфікації. Такий спосіб дає змогу генерувати одну реалізацію налаштовуваної уніфікації для вбудовування різних логічних систем у Scala та використовувати вбудовані сторонні проблемно-орієнтовані мови у логічних виразах. Монадичний API, можливо, з використанням прямого стилю поверх нього, надає розробнику програми простий та інтуїтивно зрозумілий інструмент для реалізації власної логіки всередині уніфікації.

Монадичне представлення логічного обчислення є добре відомою технікою. Стандартна бібліотека Haskell включає трансформатор монад LogKit [11] на основі [12]. У світі функціонального програмування засоби логічного програмування часто вважаються невеликими доповненнями до існуючих засобів мови. Як бачимо, більшість логічних фреймворків у функціональній мові реалізують представлення алгоритмів «генерування та фільтрування», які мають обмежене практичне застосування, тому що справжня сила логічного програмування часто розкривається з можливістю складної оптимізації, такої як індексування термів затримки цілі або розв'язувачі, що задаються користувачем [8, 13].

Для Scala існує низка різних реалізацій вбудовування логічного програмування, серед яких перенесення монади Haskell LogKit [11], Scalano [14], які представляють логічний зв'язок як об'єкт Scala над типізованими термами зі зворотним обчислювачем, Fusemate [15], який поєднує логіку першого порядку зі спеціалізованою мовою для аналізу та моделювання систем, заснованих на подіях.

Наш підхід забезпечує чітке відділення логічної дедукції: код Scala використовується лише для уніфікації, що задана користувачем, але загальне виконання цілі є зовнішньою інтерпретацією, яка може реалізувати різні стратегії. Такий спосіб надає модульність і гарну інтеграцію з рештою екосистеми, але водночас існують недоліки. В ситуації, коли API є синхронним і дані знаходяться в пам'яті, монадичний інтерфейс може бути вузьким місцем, що буде перешкоджати досягненню високої продуктивності.

Подальша робота полягає в розширенні набору підтримуваних алгоритмів і розв'язувачів для природного інтегрованого програмування в обмеженнях. Користувацькі уніфікації також можуть бути елементом системи переписування правил (в TermWare також використовується уніфікація, що може бути перепрограмована користувачем), тому додавання підтримки програмування на основі правил також може бути вектором розширення.

Література

1. The Flix Programming Language [Електронний ресурс]. Доступ до ресурсу: https://flix.dev (дата звернення: 27.11.2023).
2. Magnus M., Starup J.L., Lhoták O. Flix: a meta programming language for Datalog. Datalog. *4th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0 2022)*. 2022. P. 202–206. [Електронний ресурс]. Доступ до ресурсу: https://ceur-ws.org/Vol-3203/short8.pdf (дата звернення: 27.11.2023).
3. Logtalk [Електронний ресурс]. Доступ до ресурсу: https://logtalk.org (дата звернення: 27.11.2023).

4. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications. *Fundamenta Informaticae*. 2006. Vol. 72, N 1–3. P. 95–108.
5. Мамедов Т.А., Дорошенко А.Ю., Шевченко Р.С. Засіб статичного аналізу .NET програм за допомогою переписувальних правил. *Проблеми програмування*. 2020. № 2–3. С. 157–163.
6. Ciatto G., Calegari R., Omicini A. 2P-Kt: a logic-based ecosystem for symbolic AI. *SoftwareX*. 2021. Vol. 16, 100817. P. 1–7. [Електронний ресурс]. Доступ до ресурсу: https://doi.org/10.1016/j.softx.2021.100817 (дата звернення: 27.11.2023).
7. Sahebolamri A., Barrett L., Moore S., Micinski K. Bring your own data structures to Datalog. *Proceedings of the ACM on Programming Languages*. 2023. Vol. 7, OOPSLA2, Article 264. P. 1198–1223. [Електронний ресурс]. Доступ до ресурсу: https://doi.org/10.1145/3622840 (дата звернення: 27.11.2023).
8. Sterling L., Shapiro E. The art of Prolog advanced programming techniques. 2nd ed. Cambridge, MA, USA: MIT Press, 1994. 560 p.
9. Pfenning F., Elliott C. Higher-order abstract syntax. *ACM SIGPLAN 1988 conference on Programming language design and implementation (PLDI'88). ACM SIGPLAN Notices*. 1988. Vol. 23, N 7. P. 199–208. [Електронний ресурс]. Доступ до ресурсу: https://doi.org/10.1145/960116.54010 (дата звернення: 27.11.2023).
10. Shevchenko R. Project paper: embedding generic monadic transformer into Scala. *In Swierstra W., Wu N. (eds) Trends in Functional Programming. TFP 2022. Lecture Notes in Computer Science*. Vol. 13401. Cham: Springer, 2022. P. 1–17. [Електронний ресурс]. Доступ до ресурсу: https://doi.org/10.1007/978-3-031-21314-4_1 (дата звернення: 27.11.2023).
11. LogKit [Електронний ресурс]. Доступ до ресурсу: https://github.com/logkit/logkit (дата звернення: 27.11.2023).
12. Kiselyov O., Shan C., Friedman D.P., Sabry A. Backtracking, interleaving, and terminating monad transformers (functional pearl). *10th ACM SIGPLAN international conference on Functional programming (ICFP'05)*. 2005. New York: ACM, 2005. P. 192–203. [Електронний ресурс]. Доступ до ресурсу: https://doi.org/10.1145/1086365.1086390 (дата звернення: 27.11.2023).
13. Körner P., Leuschel M., Barbosa J., Costa V. et al. Fifty years of Prolog and beyond. *Theory and Practice of Logic Programming*. 2022. Vol. 22, N 6. P. 776–858. [Електронний ресурс]. Доступ до ресурсу: https://doi.org/10.48550/ arXiv.2201.10816 (дата звернення: 27.11.2023).
14. Amin N., Byrd W.E., Rompf T. Lightweight functional logic meta-programming. *In Lin A. (eds) Programming Languages and Systems. APLAS 2019. Lecture Notes in Computer Science*. Vol. 11893. Cham: Springer, 2019. [Електронний ресурс]. Доступ до ресурсу: https://doi.org/10.1007/978-3-030-34175-6_12 (дата звернення: 27.11.2023).
15. Baumgartner P. The Fusemate logic programming system. *28th International Conference on Automated Deduction (Automated Deduction – CADE 28)*. Berlin: Springer, 2021. P. 589–601. [Електронний ресурс]. Доступ до ресурсу: https://doi.org/10.1007/978-3-030-79876-5_34 (дата звернення: 27.11.2023).

References

1. The Flix Programming Language [Online]. Available from: https://flix.dev [Accessed 27/11/2023].
2. Magnus, M., Starup, J.L. & Lhoták, O. (2022) Flix: a meta programming language for Datalog. Datalog. *4th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0 2022)*. p. 202-206. [Online]. Available from: https://ceur-ws.org/Vol-3203/short8.pdf [Accessed 27/11/2023].
3. Logtalk [Online]. Available from: https://logtalk.org [Accessed 27/11/2023].
4. Doroshenko, A. & Shevchenko, R. (2006) A rewriting framework for rule-based programming dynamic applications. *Fundamenta Informaticae*. 72 (1-3). p. 95-108.
5. Mamedov, T.A., Doroshenko, A.Yu. & Shevchenko, R.S. (2020) Static analysis of .NET programs using rewriting rules. *Problems in programming*. (2–3). p. 157-163. (in Ukrainian)
6. Ciatto, G., Calegari, R. & Omicini, A. (2021) 2P-Kt: a logic-based ecosystem for symbolic AI. *SoftwareX*. 16, 100817. p. 1-7. [Online]. Available from: https://doi.org/10.1016/j.softx.2021.100817 [Accessed 27/11/2023].
7. Sahebolamri, A. et al. (2023) Bring your own data structures to Datalog. *Proceedings of the*

ACM on Programming Languages. 7 (OOPSLA2), Article 264. p. 1198-1223. [Online]. Available from: https://doi.org/10.1145/3622840 [Accessed 27/11/2023].
8. Sterling, L. & Shapiro, E. (1994) The art of Prolog advanced programming techniques. 2nd ed. Cambridge, MA, USA: MIT Press.
9. Pfenning, F. & Elliott, C. (1988) Higher-order abstract syntax. *ACM SIGPLAN 1988 conference on Programming language design and implementation (PLDI'88). ACM SIGPLAN Notices*. 23 (7). p. 199-208. [Online]. Available from: https://doi.org/10.1145/960116.54010 [Accessed 27/11/2023].
10. Shevchenko, R. (2022) Project paper: Embedding generic monadic transformer into Scala. *In Swierstra, W., Wu, N. (eds.) Trends in Functional Programming. TFP 2022. Lecture Notes in Computer Science*. 13401. p. 1-17. Cham: Springer. [Online]. Available from: https://doi.org/10.1007/978-3-031-21314-4_1 [Accessed 27/11/2023].
11. LogKit [Online]. Available from: https://github.com/logkit/logkit [Accessed 27/11/2023].
12. Kiselyov, O. at el. (2005) Backtracking, interleaving, and terminating monad transformers (functional pearl). *10th ACM SIGPLAN international conference on Functional programming (ICFP'05)*. New York: ACM. p. 192-203. [Online]. Available from: https://doi.org/10.1145/1086365. 1086390 [Accessed 27/11/2023].
13. Körner, P. et al. (2022) Fifty years of Prolog and beyond. *Theory and Practice of Logic Programming*. 22 (6). p. 776-858. [Online]. Available from: https://doi.org/10.48550/arXiv.2201.10816 [Accessed 27/11/2023].
14. Amin, N., Byrd, W.E. & Rompf, T. (2019) Lightweight functional logic meta-programming. *In Lin, A. (eds.) Programming Languages and Systems. APLAS 2019. Lecture Notes in Computer Science*. 11893. Cham: Springer. [Online]. Available from: https://doi.org/10.1007/978-3-030-34175-6_12 [Accessed 27/11/2023].
15. Baumgartner, P. (2021) The Fusemate logic programming system. *28th International Conference on Automated Deduction (Automated Deduction – CADE 28)*. Berlin: Springer. p. 589-601. [Online]. Available from: https://doi.org/10.1007/978-3-030-79876-5_34 [Accessed 27/11/2023].

Одержано: 30.11.2023

***Про авторів*:**

Шевченко Руслан Сергійович,
кандидат фізико-математичних наук,
старший науковий співробітник.
Кількість наукових публікацій
в українських виданнях – 11.
Кількість наукових публікацій
в зарубіжних виданнях – 8.
http://orcid.org/0000-0002-1554-2019,

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук,
професор, завідувач відділу теорії
комп'ютерних обчислень, професор
кафедри інформаційних систем
та технологій Національного Технічного
Університету України
«КПІ імені Ігоря Сікорського».
Кількість наукових публікацій
в українських виданнях – понад 200.
Кількість наукових публікацій
в зарубіжних виданнях – понад 80.
Індекс Гірша – 7.
http://orcid.org/0000-0002-8435-1451,

Яценко Олена Анатоліївна,
кандидат фізико-математичних наук,
старший науковий співробітник.
Кількість публікацій
в українських виданнях – 60.
Кількість зарубіжних публікацій – 21.
http://orcid.org/0000-0002-4700-6704.

Місце роботи авторів:

Інститут програмних систем
НАН України,
03187, м. Київ,
проспект Академіка Глушкова, 40.
Тел.: (044) 526 60 33.
E-mail: ruslan@shevchenko.kiev.ua,
doroshenkoanatoliy2@gmail.com,
oayat@ukr.net