УДК 004.4'24 https://doi.org/10.15407/pp2019.01.048

Р.С. Шевченко, А.Ю. Дорошенко

TERMWARE-3 – СИСТЕМА ПЕРЕПИСУВАННЯ ТЕРМІВ, ЗАСНОВАНА НА КОНТЕКСТНОМУ ЧИСЛЕННІ

У статті описується конструкція системи переписування термів TermWare-3, що побудована на основі рефлексивного числення контекстних термів, коли структура терму включає до себе, окрім дерева, ще й внутрішній контекст та обмеження на співставлення при застосуванні (зовнішній контекст), що дозволяє занурити операції розв'язування імен в математичну семантику переписувальних правил. Описується метод ефективної диспетчеризації вибору правил, а також автоматичне перетворення виразів між системами на основі алгебраїчних типів мови Scala та контекстними термами.

Ключові слова: автоматизація розробки програмного забезпечення, мови програмування, переписування термів, системи типів, алгебра типів, аналіз коду, termware.

Вступ

Системи переписування та логічного виводу давно застосовуються для різних задач. Перша версія системи TermWare описана у статті [1] і з'явилась у 2003 році як система термів, заснована на безтиповому переписуванні термів [2], що доповнена взаємодією з базою фактів. Ця система довела свою ефективність для широкого кола задач [3, 4]. Друга версія, була заснована на тому ж математичному апараті, але на відміну від першої версії мала інший спосіб реалізації. Тоді як у першій версії рекурсивні алгоритми обходу дерева інтерпретувались за допомогою рекурсивних Java методів, то в другій – за допомогою вбудованого нерекурсивного інтерпретатора, що не використовує вбудований стек JVM для обходу дерев. Це дозволило застосування TermWare-2 в індустріальних задачах, де можна було подати кожен файл нетривіальної Java-програми як один великий терм і виконувати перетворення, що потребують обходу всіх файлів проекту. У такому вигляді система TermWare-2 застосовується і нині [3, 4].

Система TermWare-3 (і побудована на її основі мова програмування Vavilon), з одного боку, зберігають основні риси TermWare-2, як вбудованої високорівневої бібліотеки, з іншого – додають нові можливості застосування примітивів контекстного числення [5]. Як і попередня версія, TermWare-3 – це бібліотека, що містить в собі операції роботи з термами та можливість створення та застосування переписувальних правил до об'єктів користувача.

1. Основні конструкції TermWare-3

У цьому розділі ми приведемо основні конструкції числення контекстних термів та наведемо спосіб їх представлення у вигляді Scala-конструкцій TermWare-3.

Нехай T – множина термів, що складається з множини константних символів C, з виділеною підмножиною атомів $A \subset C$ та функціональних термів $F \subset A$. Побудуємо на її основі множину контекстних мультитермів $CT(T)$, що буде включати в себе:

- константи $C_i, i \in I$:

– константні терми, що не є атомами, представлені як трейти (характеристики) PrimitiveTerm[T], де T – це примітивний тип Scala, а PrimitiveTerm[T]; існує неявне перетворення між примітивними типами Scala та PrimitiveTerm;

– атоми, що представлені як трейти AtomTerm; існує неявне перетворення між Scala символами та AtomTerm;

- функціональні терми $f_i(t_1, \ldots, t_n) \in F, i \in I$. TermWare-3 підтримує більш застосовану до способу мислення розробників концепцію *структурних термів* $f_i(n_1 = t_1, \ldots, n_m = t_m)$, де n_1, $n_2 \ldots n_m$ – імена (атоми), t_1, ..., t_m – мультитерми; структурні терми, що є абстракцією виклику функції, що в сучасних мовах може

включати іменовані параметри та значення за замовчуванням. Ці структурні терми представлені як вигляди трейта StructuredTerm. Існує реалізація цього трейта для *case*-классів, головне в цьому трейті – операція іменування;

- перетворення (стрілки) $t_1 \to t_2$:

– реалізація Scala надає трейт ArrowTerm; його основна функціональність – можливість виділення лівої та правої частини перетворення, жодна з яких не є пустою;

– внутрішній контекст *context*(*a*,*b*) або $a@b$, що можна читати як a в контексті b, де a та b – будь-які терми (в формулах іноді використовується нотація a^b); в Scala реалізація будь-якого ординарного терма може мати внутрішній контекст (можливо пустий), для операцій з ним існують методи, описані в трейті ContextBase;

- зовнішній контекст a -: b що можна описати як: "a при умові контексту b". Будемо також інколи використовувати позначення b:-a, тобто, контекст має бути ближче до двокрапки. Ідея в тому, що при використанні a -: b як паттерна, його можна співставляти (як a) лише у випадку, коли контекст співставлення сумісний з b (детальніше описується при опису уніфікації). В реалізації Termware-3 будь-який терм має зовнішній контекст, який за замовченням є '*' – універсальним мультитерм, що сумісний з будь-яким термом;

- множинні мультитерми (або *or*-вирази); $\{t_1 \ \dots \ t_n\}$ (або t_1 *or* t_2 *... or* t_n) – можна розглядати як просто множину ординарних термів. Пуста множина еквівалентна виділеному терму EmptyTerm. В TermWare-3 множинний мультитерм представлений трейтом OrSet, що на додаток до звичайних операцій над термами, дає доступ до вибірки та переліку елементів. Також, над термами визначена операція *or*.

- сумісні множинні мультитерми (або *and*-вирази); $\ll t_1, \dots t_n \gg$ (або $t_1 and\ t_2, and \dots t_n$). Вони відрізняються від or-термів тим, що мають бути сумісні між собою. Формальне визначення сумісності наведемо після переліку термів, неформально – два терми сумісні, якщо це стрілки з різними лівими частинами або один з них є множинним термом або вони однакові. Пустий множинний мультитерм є універсальним термом. TermWare-3 надає трейт AndSet та операцію *and* над мультитермами;

- послідовність альтернатив $t_1 | \ t_2 | \ \dots | \ t_{other}$, або t_1 *orElse* t_2 *orElse* ... t_n , при співставленні результатом є перша можлива альтернатива. TermWare-3 надає операцію *orElse* над термами;

- захищений терм: *if(guard, t)*, де співставлення виконується лише якщо вираз *guard* інтерпретується в *true* у логічній системі, що може задаватись як параметр системи. У TermWare є трейт IfTerm;

- порожній мультитерм $\emptyset$, який можна інтерпретувати як "пустий множинний мультитерм", що у TermWare-3 представлений об'єктом EmptyTerm;

- універсальний мультитерм $*$, який можна інтерпретувати як "сумісний з будь-яким мультитермом", представленим об'єктом StarTerm.

Користуючись базисом цих елементів, можна емулювати конструкції стандартної логіки. Так, наприклад, підстановка може бути представлена як сумісний терм: $\ll x_1 \to t_1, \dots x_n \to t_n \gg$, де x_i – атоми з іменами, що відповідають іменам змінних, а t_n – результати. Додавання ще одного елемента або створить нову сумісну підстановку, або перетворить підстановку у пустий терм, що буде свідчити про неуспішність операції.

Вільні змінні з іменами x_i можуть бути емульовані як атоми з іменами та підстановкою, що перетворює ці імена в обмеження. Вираз

$$f(x_1, x_2)@\{\ x_1 \to *\ , x_2 \to *\ \}$$

відповідає вільним змінним x_1 та x_2 без обмежень. Для змінних з обмеженнями використовуються захищені терми:

$$f(x_1, x_2)@\{x_1 \to if\ (this < 2)\ *, x_2 \to$$

$$if\ \big(isString(this)\big) \to * \}.$$

При співставленні x_1, *this* поміщається у зовнішній контест терму і вираз *this* < *2* інтерпретується у базовій логіці.

Природним чином емулюється поняття типу: (*x*:*T*) це скорочення для захищеного терму *if* (*T.resolve*(*check*),*this*). Таким чином, тип у TermWare-3 є доведенням властивості безтипового виразу. Тобто останній вираз можна переписати як:

$$f(x_1,x_2)@\{x_1 \to if\ (this < 2) *, x_2 \to if\ (this{:}\ String) \to * \}.$$

Саму систему переписувальних правил також можна представити у вигляді терму: це буде сумісна множина стрілок

$$\ll\ p_1 \to t_1, \dots p_n \to t_n \gg,$$

де $p_1, \dots p_n$ – паттерни, $t_1, \dots t_n$ – відповідні підстановки. Як видно, підстановка є частинним випадком систем правил, і там основним є правило підстановки одного правила:

$$app\big((x \to y), z\big) = \begin{cases} subst\big(y, unify(x,z)\big), \\ unify(x,z) \neq \emptyset, \\ \emptyset, unify(x,z) = \emptyset. \end{cases}$$

Місце в системі правил – визначається операцією вибору. В такому представленні, переписувальні правила мають бути когерентними (тобто не суперечити одне одному).

Ще один варіант представлення – впорядкований набір правил, що може бути представлений за допомогою переліку альтернатив. Такий терм має вигляд:

$$x_1 \to t_1 | \ x_2 \to t_2 | \dots | \ x_n \to t_n.$$

Зазначимо, що якщо терми з більш-специфічним паттерном знаходяться раніше більш загальних, то ці два представлення еквівалентні [5].

За браком місця не будемо наводити повний опис формальної системи. Найбільш близький опис є у [5, 6]. Там подана попередня формалізація, де немає різниці між сумісними та несумісними множинами термів.

2. Імплементація

Стандартний метод представлення числення в Scala – моделювання основних конструкцій у вигляді ієрархії *case*-класів та побудова алгебри операцій над цими case-класами як окремого об'єкта. В TermWare-3 прийнятий інший підхід, що на перший погляд вступає у протиріччя з загальноприйнятим функціональним стилем. А саме, основні конструкції є трейтами, що допускають різні реалізації, а алгебра операцій представлена як методи цих трейтів. Основна причина такого дизайну – можливість ефективної реалізації.

Як приклад, подивимось на проблему диспетчерізації правил. Нехай у нас є терм *x* та система переписувальних правил:

$$\ll\ p_i \to y_i \dots \gg.$$

Як знайти правило з лівою частиною, що сумісне з *x*? Якщо ми використовуємо систему *case*-класів, то нам потрібно або організовувати пошук лівої частини в послідовності правил, що не є ефективним методом, або явно перетворювати систему правил в оптимізоване представлення, що ускладнює зовнішній інтерфейс бібліотеки [7].

Наприклад, розглянемо наступну систему правил:

$$\begin{pmatrix} f(x,g(y),x) \to p_1(x,y) | \\ f(x,x,x) \to p_2(x) \\ f(x,y,x) \to p_3(x,y) \\ g(y) \to y \\ x \to A \end{pmatrix} @ \begin{pmatrix} x \to * \\ y \to * \end{pmatrix}.$$

При співставленні раціонально тримати в пам'яті не список стрілок, а структуру, що застосована для швидкого метчінгу, показану на наступному рисунку.

Тут кожна вершина використовує табличний пошук (за арністю або ім'ям головного терма), далі виконується співставлення з типовим термом, і якщо воно

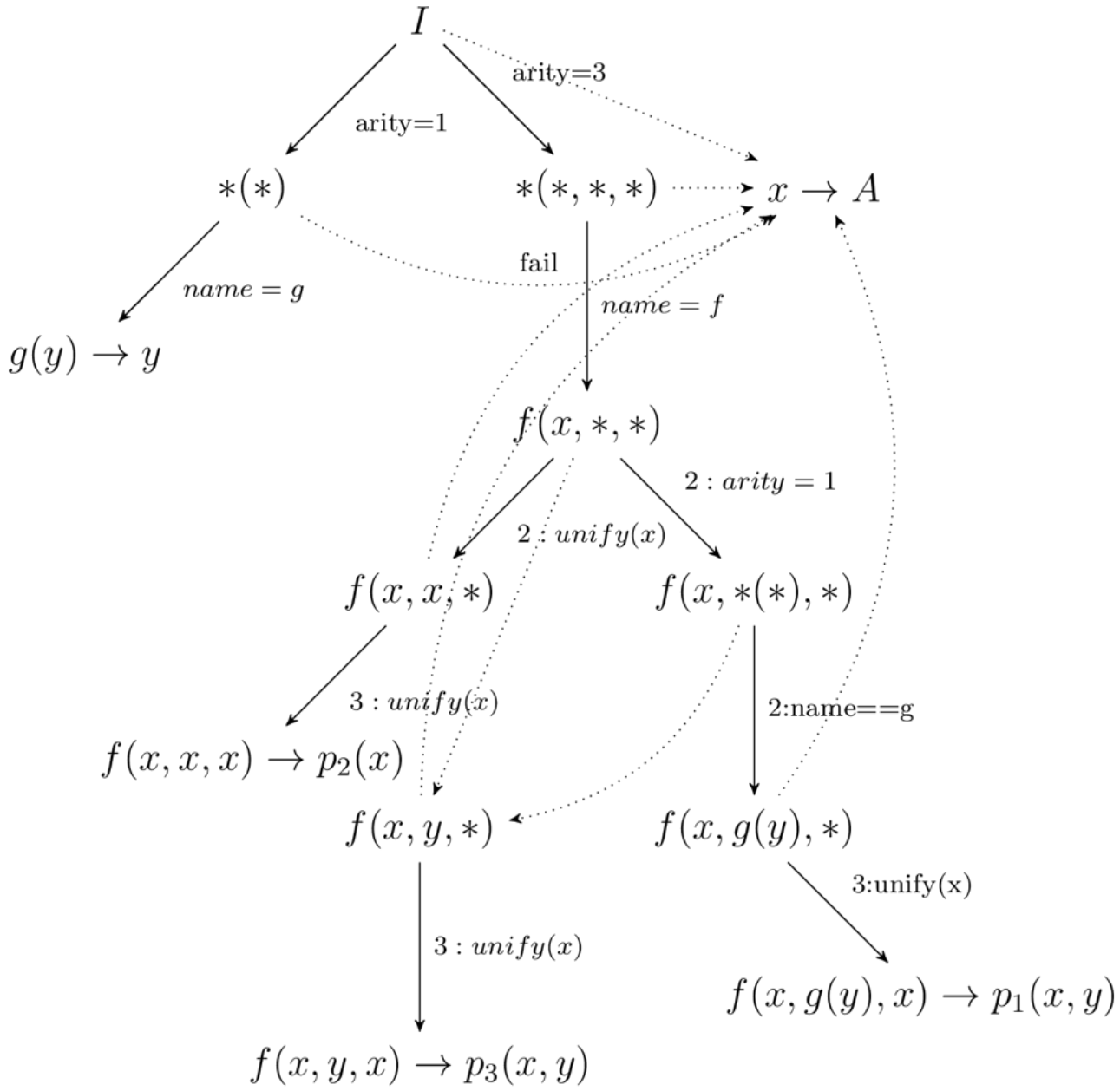

Рисунок. Структура для швидкого метчінгу

правильне – то проводиться остаточно перевірка, якщо ні – співставлення продовжується з “fallback”-вершини, перехід до якої показано пунктирною стрілкою.

Якщо взяти, наприклад, набір даних $p(1,1,1)$, то для визначення потрібного правила досить двох вибірок з таблиці: перший за арністю, другий – за іменем, а потім можна відразу переходити до правила $x \to A$; $f(1,1,1)$, що потребує трьох співставлень. У існуючій реалізації, відбувається спочатку табличний пошук за арністю, далі – табличний пошук за іменами і лише на третьому (передостанньому) рівні – повне співставлення. Теоретично ще можливо ввести кількісну оцінку складності співставлення та будувати структуру, що буде мінімізувати цю оцінку на наборі вхідних даних.

3. Приклад застосування

Наведемо приклад застосування аналізу контексту, на прикладі імплементації перевірки контрактів.

Нехай у нас є опис контракту на мові Solidity [8], що є типовим представником об’єктно-орієнтованих мов з вкладеними областями визначення. Для прос-

тоти викладу, ми не розглядатимемо фазу синтаксичного аналізу та припустимо, що об'єктом нашого розгляду є представлення Solidity AST у вигляді глибоко вбудованого (deep embedding) [9] виразу Scala.

Контракт на Solidity фактично є описом актора, що активується при надходженні транзакції, де може бути вказано додаткове повідомлення, що декодується у виклик методу. Він включає у себе опис стану актора, набір можливих функцій та вихідних випадків (events), що доступні для спостереження, разом з визначенням необхідних структур даних та модифікаторів.

На Scala це описується наступною ієрархією:

```
case class Contract(
      name: String,
     contractMember: Seq[ContractMember])

sealed trait ContractMember

case class VarDef[T](
  name:String,
  dataType: DataType[T],
  scope: Scope,
  location: DataLocation) extends ContractMember

sealed trait FunctionLikeMember extends ContractMember

case class ConstructorMember(
    params: Map[String,VarDef[_]],
    statements: IndexedSeq[Statement]
  ) extends FunctionLikeMember

case class FunctionMember(
  params: Map[String,VarDef[_]],
  statement: IndexedSeq[Statement])
```

Тут DataType – це тип даних, що може бути або вбудованим, або побудованим користувачем:

```
sealed trait DataType[T]

sealed trait Primitive[T] extends DataType[T]

case class IntPrimitive(value: Int) extends Primitive[Int]

case class AddressPrimitive(value: Long)
                    extends Primitive[Long]

 ……..
case class Struct(fields:
Map[String,DataType[_]])
              extends
DataType[Map[String,DataType[_]]]

case class Mapping[
           K:Primitive,
V:DataType](value:Map[K,V])
                          extends
DataType[Map[K,V]]

……
case class SArray[T:DataType](
              value: IndexedSeq[T]
       ) extends DataType[IndexedSeq[T]]
```

і нарешті Statement – це може бути виклик методу, присвоювання значення, ініціалізація локальної змінної тощо. Скорочено це можно проілюструвати наступним фрагментом:

```
case class MethodCall[O,T](obj: Expr[O],
params: LExpr[_]) extends Expr[T]

case class VarName[T](name:String)  extends
LExpr[T]

case class Assignment[T](left: LExpr[T] ,expr:
Expr[T]) extends Statement

case class VarInit[T](name:String,
        dateType:DataType[T],
        initExpr: RExpr[T]) extends Statement

case class IfStatement(condition:RExpr[Boolean],
```

```
        ifTrue: Statement,
        ifFalse: Statement) extends Statement
```

Як бачимо, вся програма представлена як алгебраїчна структура даних. В TermWare-3 реалізовано автоматичне перетворення екземплярів алгебраїчних типів у терми: все що потрібно, це проімпортувати DSL та викликати розширений метод перетворення:

```
val programTerm = ast.toTerm()
```

Але за замовчуванням, структура AST буде перенесена в таку ж структуру терму без побудови контекста. Для того, щоб показати, що деякі частини належать до контексту, потрібно вказати відображення поля в розташування підтерму (що може бути або в підтермі, або в контексті):

```
implicit object ContractContextPlacement
                extends AsContext[

Seq[ContractMember],
                              Symbol,
                              ContractMember]{

 override def map(in: Seq[ContractMember]):
     Try[Map[Symbol, ContractMember]] = {
  Try {
   in.map(m => (Symbol(m.name), m)).
                 groupBy(_._1).
                mapValues {
    s =>
     if (s.tail.nonEmpty) {
      throw new IllegalArgumentException(
              "duplicate name ${s.head._1}")
     } else {
      s.head._2
     }
   }
  }
 }

 override def unmap(in: Map[Symbol, Con-
tractMember]): Try[Seq[ContractMember]] = {
  Success(in.values.toSeq)
 }
}
```

І схоже перетворення потрібно вказати для VarDef, щоб визначення змінної потрапляло у контекст блоку. Як бачимо, задача цього перетворення – перетворення терму на маппінг, що згодом перетворюється на набір впорядкованих даних. Також зазначимо використання типу Symbol як ключа відображення, замість String. Це зроблено тому, що String за замовченням відображається у PrimitiveTerm[String] а Symbol – в AtomTerm.

Також має сенс імена змінних та частини селектори теж представляти як атоми:

```
implicit def varNameFieldMapping[T] =
  new AsFieldMapping[VarName[T],Symbol] {
 override def map(in: VarName[T]):
Try[Symbol] = Success(Symbol(in.name))

 override def unmap(in: Symbol):
Try[VarName[T]] = Suc-
cess(VarName(in.toString()))
}
```

Тепер *programTerm*, представлений як терм, в контексті якого знаходяться визначення (тут ми для простоти оминули додавання стандартних елементів програми, що визначені в специфікації Solidity). Ми можемо використовувати навігацію за символами: *x.resolve(name)* видасть нам визначення імені.

Розглянемо, для прикладу, програму перевірки (чекер) так званої «reentrance» [10] – помилки в контрактах, коли контракт робить транзакцію, не змінюючи свого стану. Саме ця вразливість була використана у відомому зломі DAO [11].

Приклад проблеми такого роду можна побачити в наступному коді:

```
contract Example {

  mapping (address=>uint) balances;

  ….

 function add() {
   balances[sender.address] += msg.valuel
```

```
 }

 function withdraw(uint amount) {
   if (balances[sender_address] > amount) {
      sender.address.send(amount);
      balances[sender.address] -= amount;
   }
}
```

Тут у функції *withdraw*, метод *send* викликається до модифікації *balances[sender.address]*. Оскільки send – це фактично виклик функції прийому платежа в іншому контракті, то інший контракт може викликати функцію *withdraw* ще раз, до того як поверне керування, і отримати ще один трансфер на свою адресу.

Система правил для знайдення цієї вразливості може виглядати так:

```
StateVars(statements, history) -> {

 statements . (seq(head,tail) ->
    IfStatement(expr,ifTrue,ifFalse) ->
                                Stat-
eVars(seq(ifTrue,tail)) &&
                                Stat-
eVars(seq(ifFalse,tail)) head
    |
     StateVars(tail, StateVars(head) + history)
   )

 Assignment(left,expr) -> StateVars(left)

 MethodCall(account,'send,params)
  if (resolve(account).type == address  &&
                                    history.isEmpty)
      -> Violation("reentrance-bug")
    |
   MethodCall(obj,metod,params) ->
     let v = re-
solve(obj).resolce(method).statements
      StateVars(tail,StateVars(v) + history)
```

Тут пропущені деталі, які не впливають на алгоритм, такі як вказання набору змінних. Цю систему правил ми маємо викликати до кожного методу. Тобто, ми проводимо рекурсивний спуск за послідовністю виразів, введемо в *history* множину і коли потрапляємо на виклик методу *send*, перевіряємо, що множина змін не є пустою.

Якщо ми зустріли виклик об'єкта в програмі, то додаємо до множини змін його слід (трейс).

Висновки

Отже, в даній роботі представлена система TermWare-3 на основі імплементації контекстного числення термів. Це дозволяє застосувати методи переписувальних правил до великих систем і подавати зв'язки між різними частинами системи у явному вигляді за допомогою розв'язання імен (резолвінгу).

Подальші кроки, що планується здійснити в цьому напрямку, це:

- адаптація застосування TermWare-3 у прикладних проектах;
- інтеграція з аналізаторами мов програмування Java та C, таким чином, щоб існувала можливість міграції систем на основі TermWare-2 на нову платформу;
- імплементація відображення системи типів Scala на TermWare-3 з тим щоб отримати типобезпечне вбудовування в Scala;
- побудова та аналіз ефективних імплементацій базових операцій.

Література

1. Shevchenko R., Doroshenko A. Managing Business Logic with Symbolic Computation. Information Systems Technology and Applications: Proc. 2-nd Intern. Conf. ISTA'2003, June 19–21, 2003. Kharkiv, Ukraine. P. 143–152.
2. Marc Bezem, Jan Willem Klop, Roel de Vrijer ("Terese"), Term Rewriting Systems ("TeReSe"), Cambridge University Press, 2003.

3. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications. *Fundamenta Informaticae*. 2006. Vol. 72, N 1–3. P. 95–108.
4. Eugene Tulika, Anatoliy Doroshenko, Kostiantyn Zhereb. Using Choreography of Actors and Rewriting Rules to Adapt Legacy Fortran Programs to Cloud Computing. June 2017. Communications in Computer and Information Science.
5. Шевченко Р.С., Дорошенко А.Е. η-исчисление – реалистичная формализация класса переписывающих систем. Проблеми програмування. 2011. № 2. С. 3–11.
6. Шевченко Р.С. Числення контекстних термів для систем переписування. *Проблеми програмування.* 2018. № 2–3. С. 21–30.
7. Шевченко Р. Про імплементацію систем контекстних термів. *Winter InfoCom Advanced Solutions*. 2018. 41 с.
8. Ethereum Foundation. The solidity contract-oriented programming language. https://github.com/ethereum/solidity.
9. Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). In Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP '14). 2014. ACM, New York, NY, USA. P. 339–347.
10. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16). ACM, New York, NY, USA. P. 254–269.
11. "Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO" (PDF). Securities and Exchange Commission. July 25, 2017.

References

1. Shevchenko R., Doroshenko A. Managing Business Logic with Symbolic Computation. Information Systems Technology and Applications: Proc. 2-nd Intern. Conf. ISTA'2003, June 19–21, 2003, Kharkiv, Ukraine. P. 143–152.
2. Marc Bezem, Jan Willem Klop, Roel de Vrijer ("Terese"), Term Rewriting Systems ("TeReSe"), Cambridge University Press, 2003, ISBN 0-521-39115-6.
3. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications. Fundamenta Informaticae. 2006. Vol. 72, N 1–3. P. 95–108.
4. Eugene Tulika, Anatoliy Doroshenko, Kostiantyn Zhereb. Using Choreography of Actors and Rewriting Rules to Adapt Legacy Fortran Programs to Cloud Computing. // June 2017. Communications in Computer and Information Science, DOI: 10.1007/978-3-319-69965-3_5
5. Shevchenko R.S., Doroshenko A.Y. η-calculus – Realistic Formalization of a Class of Rewriting Systems. Problems of Programming. 2011. N 2. P. 3–11.
6. Shevchenko R.S. A Calculus of Context Terms for Rewriting Systems. Problems of Programming. 2018. N 2–3. P. 21–30.
7. Shevchenko R. On Implementation of Term Systems. Winter InfoCom Advanced Solutions, 2018. 41 p.
8. Ethereum Foundation. The solidity contract-oriented programming language. https://github.com/ethereum/solidity.
9. Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). //In Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP '14). 2014. ACM, New York, NY, USA, 339-347. DOI: https://doi.org/10.1145/2628136.2628138
10. Loi uu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, nd Aquinas Hobor. Making Smart Contracts Sarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). ACM, New York, NY, USA, 254-269. DOI:https://doi.org/10.1145/2976749.2978309
11. "Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO" (PDF). Securities and Exchange Commission. July 25, 2017.

Одержано 08.02.2019

***Про авторів*:**

Шевченко Руслан Сергійович.
Підприємець.
Кількість наукових публікацій в українських виданнях – 13.
Кількість наукових публікацій в зарубіжних виданнях – 5.
http://orcid.org/0000-0002-1554-2019,

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук,
професор, завідувач відділу
теорії комп'ютерних обчислень,
професор кафедри автоматики та
управління в технічних системах
НТУУ "КПІ імені Ігоря Сікорського ".
Кількість наукових публікацій в українських виданнях – понад 150.
Кількість наукових публікацій в зарубіжних виданнях – понад 50.
Індекс Хірша – 5.
http://orcid.org/0000-0002-8435-1451,

Місце роботи авторів:

ПП "Руслан Шевченко".
E-mail: ruslan@shevchenko.kiev.ua

Інститут програмних систем
НАН України,
03187, м. Київ-187,
проспект Академіка Глушкова, 40.
Тел.: (044) 526 3559.
E-mail: doroshenkoanatoliy2@gmail.com