УДК 004.4'24

ЧИСЛЕННЯ КОНТЕКСТНИХ ТЕРМІВ ДЛЯ СИСТЕМ ПЕРЕПИСУВАННЯ

Р.С. Шевченко

У статті пропонується числення контекстних термів, що доповнює традиційний апарат алгебраїчних сигнатур, котрі використовуються в системах переписування термів, конструкціями роботи з контекстом: утворення контексту, визначення контекстного значення та зв'язку терму з контекстом із перевіркою відповідності. Це дозволяє сформулювати задачі, пов'язані з аналізом та трансформаціями програмного коду, у більш природньому вигляді, оскільки структура вихідного коду в сучасних мовах програмування також має ієрархічну контекстну структуру, що може прямо відповідати структурі терму в переписувальному правилі. Правила виводу типів для мови програмування можуть бути представлені як правила задовільності контексту. Таким чином можна отримати досить простий механізм, що об'єднує переписування термів з аналізом контексту. Застосування підходу описано на прикладі моделювання системи типів сучасних мов програмування: від обмежень на значення до лінійної типізації.
Ключові слова: автоматизація розробки програмного забезпечення, мови програмування, переписування термів, системи типів, алгебра типів, аналіз коду, termware.

В статье предлагается исчисление контекстных термов, которое дополняет традиционный аппарат алгебраических сигнатур, использующихся в системах переписывания термов, конструкциями работы с контекстом: конструирования контекста и операциями определения контекстного значения и связи терма с контекстом с проверкой соответствия. Это позволяет сформулировать задачи, связанные с анализом и трансформацией программного кода в более естественном виде, так как структура исходного кода в современных языках программирования также обладает иерархической контекстной структурой, что может прямо соответствовать структуре терма в переписывающих правилах. Правила вывода типов для языка программирования могут быть отображены в правила соответствия контексту. Таким образом мы получили довольно простой механизм, объединяющий переписывание термов и анализ контекста. Применение подхода описано на примере формулирования с помощью этого механизма распространенных систем типов.
Ключевые слова: автоматизация разработки программного обеспечения, переписывания термов, системы типов, языки программирования.

In this paper, a calculus of context-full terms is proposed. Calculus extends the traditional algebraic signatures, used in term rewriting systems, by so-called constructions, that works with context: context constructor, resolving a context value and binding term to context with compatibility checking. Such extension allows to refining algorithms connected with analysis and transformations of source code in the more natural form, because f the structure of the sense in the modern programming language defined in hierarchical contexts, which can be directly mapped to context-full term in rewriting rule. Type analysis for a programming language can be implemented as checking for context compatibility. In such way, relative simple mechanism, which unites term rewriting and context analysis can be received. The approach is illustrated by defining rules for typing in context-full term formalism.
Key words: software development automation, term rewriting, type systems, programming languages, code analysis, termware.

Вступ

Використання переписувальних правил є широко відомою технологією. Існує багато формалізацій, заснованих на різному підгрунті: від імперативної операційної семантики[1, 2] до екваціональної логіки, ρ-числення [3] та числення зразків (pattern-calculus) [4]. Існує також декілька мов програмування, побудованих на основі переписування термів.

Одним із типових застосувань подібних систем є опис семантик мов програмування. Класичні системи переписувальних правил можуть використовуватися для опису трансформацій стану "в малому", але необхідність роботи з контекстом сильно обмежує можливості декларативного аналізу. Існують різні підходи до вирішення цієї проблеми: аналізатори програм, що засновані на termware [5], використовують так звані 'збагачені' терми, що завжди містять додатковий елемент, що дає доступ до запитів інформації з контексту; Stratego[6] (зараз Spoofax) використовує набори правил, що залежать від контексту.

Ще один підхід – абстрактний синтаксис вищого порядку (high order abstract syntax)[7], де AST з інформацією про змінні представлено у вигляді лямбда-терму, де зв'язані імена представлені як змінні, що зв'язані в лямбда-термах.

У цій статті для роботи з контекстом пропонується числення контекстних термів, що доповнює традиційний апарат алгебраїчних сигнатур конструкціями роботи з контекстом. Підхід полягає у побудові алгебри контекстних термів, де операції щодо контекстних термів виражені у явному вигляді.

1. Основні конструкції

Нехай C – множина константних символів, 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;

- функціональні терми $f_0(t_1 \ldots t_n) \subset P$. Будемо вважати голову терма також термом, обмеженним видом констант за конструкцією;
- перетворення (стрілки) $a \to b$;
- контекст context(a,b) або $a@b$, що можна читати як *a* в контексті *b*, де *a* та *b* – будь-які терми. (Також в формулах будемо іноді використовувати нотацію a^b);
- множинні мультитерми (або Or – вирази) $\{t_1 \quad \ldots \quad t_n\}$;
- послідовність альтернатив $\tau_1 \mid t_2 \mid \ldots \mid t_{other}$;
- порожній мультитерм $\emptyset$, який можна інтерпретувати як “пустий множинний мультитерм”;
- універсальний мультитерм $*$, який можна інтерпретувати як “сумісний з будь-яким мультитермом;
- мультитерм протиріччя $\perp$.

Основні особливості – замість термів ми оперуємо “мультитермами”, та також відсутні такі об’єкти як вільні змінні. Ми їх промоделюємо за допомогою контексту: нехай в нас є алфавіт змінних $X \quad \{x_1 \ldots x_n\}$ і перетворимо терм з вільнимі змінними $f(x_1 \ldots x_n)$ у контекстний терм $f(x_1 \ldots x_2)@\{x_1 \to *, x_2 \to *\}$ де $x_1 \ldots x_2$ – атоми, що не входять в оригінальний терм. Таким чином, ми можемо репрезентувати традиційні системи переписування, переінтерпритувавши базові операції.

Уніфікація

Почнемо з уніфікації. В нашій інтерпретації вона перетворюється в бінарну операцію над термами, що повертає уніфікатор у контексті підстановок, що перетворюють будь-який аргумент на найбільш конкретний уніфікатор. Будемо позначати уніфікація a та b як unify(a,b) або скорочено $a \circ b$.

Для констант та мультитермів:

$$a \circ a \quad a, \tag{1}$$

$$a \circ b \quad \emptyset \ iff \ a \subset C, \ b \subset C, \ a \neq b, \tag{2}$$

ми можемо замінити при уніфікації символ на його значення в контексті:

$$(a \circ v)@(v \to x) \quad \to a \circ x@(v \to x) \tag{3}$$

два значення мають бути сумісними:

$$(v \circ w)@\{v \to x, w \to y\} \quad \to k@\{v \to k, w \to k, k \to x \circ y\}. \tag{4}$$

Тепер додамо до розгляду структурні терми:

$$f(x_1 \ldots x_n) \circ g(y_1 \ldots y_m) \quad \begin{cases} f(v_1 \ldots v_n)@\{v_1 \to x_1 \circ y_1, \ldots, v_k \to x_n \circ y_k\} \\ \emptyset, \ f \neq g \\ \emptyset, n \neq m \end{cases}. \tag{5}$$

Множинні терми:

$$\{a, b\} \circ c \quad \{a \circ c, b \circ c\}, \tag{6}$$

та симетрично:

$$a \circ \{b, c\} \quad \{a \circ b, a \circ c\}. \tag{7}$$

Послідовна перевірка:

$$(a \mid b) \circ c \quad \begin{cases} a \circ c, \ a \circ c \neq \emptyset \\ b \circ c, otherwise \end{cases} \tag{8}$$

та симетрично:

$$a \lhd (b \mid c) \quad \begin{cases} a \circ b, & a \rhd b \neq \emptyset \\ a \lhd c, & otherwise \end{cases}. \tag{9}$$

Уніфікація загальних та пустих мультитермів

$$a \lhd * \quad \rhd a, \tag{10}$$

$$* \lhd a \quad \rhd a, \tag{11}$$

$$a \lhd \emptyset \quad \rhd \begin{cases} \emptyset, a \neq \bot \\ \bot, a \quad \bot \end{cases}, \tag{12}$$

$$a \lhd \bot \quad \bot. \tag{13}$$

Уніфікація значень в контексті :

$$(a^b) \lhd (c^d) \quad (a \circ c)^{b \cup d}. \tag{14}$$

Де $b \cup d$ – операція злиття, яку ми введемо трохи пізніше

$$(a \rhd b) \rhd (c \rhd d) \quad \begin{cases} (a \rhd c \rhd b' \circ d')@ctx(a \circ c), & \begin{matrix} b' & subst(b, ctx(a \rhd c)) \\ d' & subst(d, ctx(a \circ c)) \end{matrix} \quad iff\ a \circ c \neq \emptyset, b' \circ d' \neq \emptyset \\ \emptyset, a \circ c \quad \emptyset\ or\ b' \circ d' \quad \emptyset \\ \bot, (any\ of\ arguments \quad \bot) \end{cases}. \tag{15}$$

Де $subst$ та ctx – операції підстановки та взяття контексту з очевидним змістом.

Можна прочитати це перетворення наступним чином: при уніфікації двох стрілок ми визначаємо сумісність лівих та правих частин, та вертаємо лишу сумісну частину. Якщо уніфіковані стрілки дають різний результат, то визнаємо уніфікацію невдалою.

Множинні терми та операції роботи з контекстом

В прикладі з представленням підстановок, контекст підстановки був представлений у вигляді множинного мультитерму $\{a, b, c\}$. У найпростішому випадку, що дає нам перше інтуїтивне бачення, елементи контексту - це набір стрілок типу атом – значення, проте множинний мультитерм може містити об'єкти іншого роду: наприклад, стрілки загального типу, тоді об'єднання стрілок – це система правил, або просто різнорідні об'єкти без протиріч, тоді це просто набір можливостей вибору.

Ми будемо позначати множинний терм як $\{a, b, c\}$, вважаючи що *a*, *b* та *c* – сумісні між собою. Створення множини до перевірки сумісності, будемо позначати як $a \cup b$. Наприклад, твердження, що одне й те ж значення у контексті повинно бути визначено сумісно, можно записати як: $(x \rhd a) \cup (x \rhd b) \quad x \rhd a \lhd b$.

Множина інваріантна до повторень

$$a \cup a \quad a; \tag{16}$$

та комутативна

$$a \cup b \quad b \cup a. \tag{17}$$

При об'єднанні атомів,

$$a \cup b \quad \{a, b\}\ iff\ (a \circ b) \quad \emptyset. \tag{18}$$

У більш загальному вигляді, можна сказати, що $a \cup b \quad (a \rhd b) \cup a \cup b$, тобто можна розглядати об'єднання як фільтр на уніфікацію.

З того, що множина, де є несумісне значення, є протиріччям, випливає:

$$(a \rhd \emptyset) \cup b \quad \bot. \tag{19}$$

Також додамо правила роботи з граничними термами:

$$a \cup = \quad *, \tag{21}$$

$$a \cup \emptyset \quad a; \tag{22}$$

$$a \cup \_ \quad \bot. \tag{23}$$

Та мультитермами:

$$a \cup \{b, c\} \quad (a \cup b) \cup c, \tag{24}$$

$$a \cup (b \mid c) \quad (a \cup b) \mid (a \cup c). \tag{25}$$

Поширення на структурні терми та стріли практично аналогічні (18):

$$(F \quad f(x_1, \ldots x_n)) \cup (G \quad g(y_1, \ldots y_m)) \quad \{F * G, F, G\}, \tag{26}$$

та

$$(a \ \rightarrow b) \cup (c \ \rightarrow d) \quad \{(a \ \cdot b) * (c \ \rightarrow d), a \ \rightarrow b, c \ \rightarrow d\}. \tag{27}$$

Введемо також стандартні позначення для існування та селекції у множинних термах, запозичені з теорії множин: $x \subset \{x_1 \ldots x_n\}$ скорочення для $(x \quad x_1) \vee \ldots \vee (x \quad x_n)$; $v \quad \{x \subset y \mid p(x)\}$ – для $\cup_{x \subset y} \begin{cases} x, p(x) \\ \emptyset, \neg p(x) \end{cases}$ множинного терму, що містить лише ті терми, що задовольняють *p*.

Інше об'єднання термів – послідовний перегляд, можна проінтерпретувати як впорядкований перебіг. Тобто розглядаючи набір $a \ \rightarrow b \mid c \ \rightarrow d$, ми спочатку шукаємо спробу уніфікувати *a*, якщо це не спрацювало – тоді *c*. Це дозволяє відтворити ефект "затінювання" імен (shadowing), коли при переміщенні в контекст, нові імена затінюють старі.

Єдині редукційні правила, що ми можемо додати до послідовного перегляду, – це

- елімінація пустого терму:
  - $\emptyset \mid x \quad x;$ (28)
  - $x \mid \emptyset \quad x;$ (29)
- ескалація протиріччя
  - $\bot \mid x \quad \bot;$ (30)
  - $x \mid \bot \quad \bot;$ (31)
- та збереження універсального терму:

$$* \mid x \quad \begin{cases} *, x \neq \bot \\ \bot, x = \bot \end{cases}. \tag{32}$$

Нарешті можна формулювати правила роботи з контекстом: *x@y*, що визначає *x* у контексті *y*. Почнемо з співвідношення контексту та граничних термів.

Єдина можлива множина у контексті протиріччя – це пустий мультитерм:

$$x@\bot \quad \begin{cases} \bot \ if \ x \quad \bot \\ \emptyset \end{cases}. \tag{33}$$

Протиріччя у контексті залишається протиріччя (ми хочемо зберегти термінальний об'єкт):

$$\bot @ x \quad \bot. \tag{34}$$

Вираз з пустим контекстом еквівалентний виразу без контексту:

$$x@* \quad x. \tag{35}$$

І контекст пустого терма не має змісту: в нас не існує способів його визначити:

$$\emptyset@x \quad \emptyset. \tag{36}$$

Щодо універсального терма: як контекст він не має змісту:

$$x@* \quad \bot, \tag{37}$$

але поміщення універсального терму в контекст має зміст – таким чином можна задавати обмеження для можливого матчінгу. Мультитерми передають контекст своїм компонентам:

$$\{a, b..\}^x \quad \{a^x, b^x\}; \tag{38}$$

$$(a|b)^x \quad a^x|b^x. \tag{39}$$

Залишилась взаємодія між структурними термами та контекстом. Спочатку, наведемо декілька різних прикладів прагматики застосування:

Нехай у нас є структурний терм у контексті: $f(t_1, \ldots t_n)@\{a \to b, c \to d, \ldots\}$. З одного боку, субтерм t_i має мати доступ до загального контексту (тобто $f(t_1, \ldots t_n)^a \quad f(t_1^a \ldots t_n^a)$), якщо a – це щось типу доступу до глобальних елементів; з іншого боку – ми хочемо в контексті зберігати деякі речі, специфічні для $f(\ldots)$ в цілому, скажемо тип f або обмеження на вигляд аргументів. Тому контекст структурного терму складається з двох частин: контекст форми (в коді – head-контекст), що стосується форми f та не передається в аргументи, та контекст змісту (в коді – content-контекст), що передається. Тоді ми зможемо записати передачу контексту як:

$$f(t_1, \ldots t_n)^a \quad f^{head(a)}(t_1^{content(a)} \ldots t_n^{content(a)}). \tag{40}$$

A head та content можна вибирати з контексту, використовуючи контексти лівих частин. Нехай в нас є виділений атом, *htag*, будемо вибирати як head – атоми з ним у лівій частині контексту.
Спочатку визначимо селекцію по тегу:

$$tagSelect(a, tag) \quad \begin{cases} a, & tag \subset left(h) \\ \emptyset, otherwise \end{cases}. \tag{41}$$

Де

$$left(h) \quad \begin{cases} \{left(a_1), \ldots left(a_n)\}, & h \quad \{a_1, \ldots a_n\} \\ left(a_1)|\ldots|left(a_n), & h \quad a_1|\ldots|a_n \\ a, & left \quad a \to b \\ h, otherwise \end{cases}, \tag{42}$$

$$head(x) \quad tagSelect(x, htag), \tag{43}$$

$$content(x) \quad \{a \subset x | head(a) \quad \emptyset\}. \tag{44}$$

Тепер можна сказати, що (40) визначено однозначно.

Залишилось визначити деталізацію уніфікації для контексту. Контекст зразка може визначати додаткові обмеження на уніфікації, що, як правило, можуть бути виражені у вигляді набору правил, представлених мультитермом.

$$x^a \circ y \quad \begin{cases} (x \circ y)^a, & = \{check \to r\} \subset a. \ apply(r, x \circ y) \to true \\ \emptyset & , = (check \to r) \subset a. \ apply(r, x \circ y) \to true. \\ x \circ y & , check \notin left(a) \end{cases} \tag{45}$$

Оскільки уніфікація симетрична, контекст в y розбирається аналогічно. Можна спростити конструкцію, якщо перенести перевірку правил сумісності в правила контексту. Тоді контекстне правило набуде вигляду:

$$x^a \quad \emptyset \ iff = [check \to r] \subset a: apply(r, x) \to true. \tag{46}$$

А уніфікація контексту матимите вигляд: $x^a \circ y \quad (x \circ y)^a$, або для двох контекстів:

$$x^{a} \circ y^{b} \quad (x \circ y)^{a \cup b}. \tag{47}$$

Це всі основні правила конструкції, тепер ми можемо перейти до визначення дії правил переписування.

Основне правило залишається точно таким, як і в традиційних системах:

$$apply(p \to x, y) \quad \begin{cases} subst(x, ctx(p \circ y)), p \circ y \neq \emptyset \\ \emptyset, p \circ y = \emptyset \end{cases}. \tag{48}$$

Множинні терми дозволяють вибирати (паралельно) всі правила, що підходять, а також обробляти множину правил:

$$apply(a \cup b, y) \quad apply(a, y) \cup apply(b, y), \tag{49}$$

$$apply(a, x \cup y) \quad apply(a, x) \cup apply(a, y). \tag{50}$$

А послідовні перетворення дають нам семантику пріоритетного виконання

$$apply(a|b, y) \quad \begin{cases} apply(a, y), apply(a, y) \neq \emptyset \\ apply(b, y), apply(a, y) = \emptyset \end{cases}, \tag{51}$$

$$apply(a, x|y) \quad \begin{cases} apply(a, x), apply(a, x) \neq \emptyset \\ apply(a, y), apply(a, x) = \emptyset \end{cases}. \tag{52}$$

Граничні терми є нижчими елементами решіток (без суперечностей):

$$apply(a, \emptyset) = \emptyset, apply(\emptyset, a) = a. \tag{53}$$

Та з суперечністю:

$$apply(a, \bot) = \bot, \text{ та } apply(\bot, a) \quad \begin{cases} \emptyset, a = \emptyset \\ \bot, a \neq \emptyset \end{cases} \tag{54}$$

$$apply(-, x) = \bot, apply(x, *) = \emptyset. \tag{55}$$

Залишилось визначити співвідношення перетворення і контекста: тут ми можемо, як у випадку з уніфікацією, використовувати значення контексту для перевірки:

$$(a \to b)^{c} \quad a^{c} \to b^{context(c)}, \tag{56}$$

$$apply(a^{c} \to b, x) \quad \begin{cases} subst(b, ctx(a \circ x)^{c}), a \circ x \neq \emptyset \\ \emptyset, a \circ x = \emptyset \end{cases}. \tag{57}$$

Далі, семантику виводу будуємо точно так само, як і у традиційних системах:

Застосування набору правил *r* до терму *t* за допомогою стратегії *s*, що визначає порядок перебору, можна визначити наступним чином:

$$s.traverse(t \to (apply(r, t)|\tau)).$$

Тобто стратегія замінює у термі підтерми на результат apply, або залишає без змін, якщо правила не можуть бути застосовані бути прикладено (*apply* повертає $\emptyset$).

2. Приклад: формулювання типів

Проілюструємо використання систем переписування реалізацією аналізу типів: побудуємо моделі основних систем типів. Перше питання: як буде виглядати типізована AST у вигляді контекстного терму? Ми будемо представляти тип контекстом, у якому визначено виділений термін check, що визначає набір правил.

Тобто, якщо у мові програмування (x:E) означає, що x має тип E – то на рівні термів в нас є контекст E, у якому є набір правил check. Інтепретація цього набору правил визначає можливість створення терму у цьому контексті.

Непараметризований список буде визначатись наступним чином:

```
List-> {
  check-> x-> (
             (x <> Nil)
              ||
             (x <> Cons(y,z)){y->*,z->*@List}
           )@{ x -> * }@logic
  }
}
```

де logic – якась система, що надає мінімальну інтерпретацію логічних виразів та уніфікації, (будемо називати її L).

Визначення параметризованого списку виглядає подібним чином:

```
{List(e) -> {
  check-> x-> (
             (x <> Nil)
             ||
             (x <> Cons(y,z)){y->*@e,z->*@List(e)}
           )@{ x -> * }@logic
  }
}}@{e->*}
```

Тобто List перетворюється на функціональний терм, де *e* – параметр, що визначається під час уніфікації. E – звичайний терм, що може бути представлений у termware. Отже, "натуральною моделлю" типів буде числення залежних типів [8].

Також зазначимо, що поки e – не визначено, відсутня різниця між екзистенціальними та параметризованими типами. Щоб побудувати модель системи, що підтримує екзистенціальні типи, ми маємо або збагатити інтерпретатор логіки, додавши аналог квантору існування, або збагатити маппінг термів на типи, додавши позначення для невизначених термів.

У першому випадку, визначення екзистенціального типу буде виглядати подібно до:

```
Ex.List -> {
  (List e)(e->exists)@logic
 }
```

Де exists – має бути визначений в логіці як оператор селекції. В залежності від класу логіки, яку ми оберемо, можна отримати багато варіацій можливої семантики. Це досить цікаве питання для подальшого аналізу, але виходить за рамки короткого опису системи у цій статті.

У другому – можна отримати нотацію, подібну до сколемівських типів у Scala (невизначених аліасів):

```
List -> {
  check-> x-> (
             (x <> Nil)
             ||
             (x <> Cons(y,z)){y->*@element,z->*@(List@{`element->element})}
           )@{ x -> * }@logic,
  element -> *
}
```

Де *element* – не аргумент функціонального терму, а елемент, що має бути присутнім у конкретному термі, що використовує контекст List.

Далі, сума та об'єднання типів формулюється очевидним способом (диз'юнкція та кон'юнкція check клауз); алгебраїчні типи – додаванням до check клаузи ще правила визначення підтипу в залежності від дискримінатору.

Як бачимо, елементи систем типів, що зустрічаються у найбільш поширених мовах програмування, прямо моделюються у численні контекстів.

Розглянемо ще менш поширені різновиди – лінійні та афінні типи, що використовуються у мовах з обліком використання ресурсів (такі як Clean [9], Rust [10], Pony [11]).

Тип називається лінійним, якщо один вираз цього типу може бути використан у програмі лише один раз. (Приклад – у мові з тракінгом ресурсів, показчик на мутабельну область пам'яті, що може бути переданий у іншу функцію без копіювання. Функція-отримувач має або передати його далі, або звільнити).

Афінний тип – тип, що може бути використаний у програмі один або нуль раз. Приклад – мутабельна змінна, що може бути передана до іншого потоку виконання.

Перше питання – як ми взагалі можемо промоделювати програми з лінійними типами: вочевидь, нам потрібно визначити поняття блоку коду та зв'язати сам тип з входженням у цей блок.

```
Linear{
  check -> (x -> LinearStatements(definedIn@x))@(x->*)
  definedIn -> (x -> *@LinearStatements)@(x->*)
}
```

Тут ми можемо визначити лінейний тип як такий, що визначений у LinearStatements. *definedIn* ми повинні визначити при відображенні AST на терми.

LinearStatements можна розглядати як список виразів. А що таке вираз з точки зору нашого аналізу? Це щось, що може або створювати або використовувати лінійну змінну:

```
LinearStatement {
   usage -> (x -> boolean)@(x->*@Linear),
   created -> List *@Linear
}
```

Тепер, можна визначити типизацію блоку коду:

```
LinearStatements = {
  check -> {x -> checkLinear(Nil, Nil, x)}@(x->*),
  checkLinear -> {
     (nonUsed, used, statements) ->
            {
              if (statements = Nil) -> used = Nil and nonUsed=Nil,
              if (statements = Cons(statement,rest)@(statement->*@LinearStatement,
                                        rest -> (List *@LinearStatement))
                  ->  checkLinear(
                        created(statement) + nonUsed.filter(!statement@usage),
                        used + nonUsed.filter(statement@usage),
                        rest
                      )
                      and checkNonUsed(nonUsed,statement)
            }
  },
  checkNonUsed -> {
    (Nil, statement) -> true
    (Cons(x,y),statement) -> !statement@usage(x) and checkNonUsed(y,statement)
  }
}
```

(Тут +, filter – операція додавання списків, що може бути визначена очевидним чином).

Таким чином, маємо таке формулювання: лінійний тип – це такий тип, що вираз цього типу, визначений у блоці, використовується там лише один раз, декларативно виражено мовою переписуючих систем.

Висновки

Отже, числення контексту дозволяє декларативно промоделювати сучасні системи типів та здійснювати семантичний аналіз та перетворення програм, залишаючись у рамках декларативної парадигми переписувальних правил. Поєднуючи різну логічну семантику з відображенням мов програмування, можна отримати різні версії систем типизації.

Наступні кроки:

- верифікація і побудова верифікованої моделі у coq;
- імплементація наступної версії termware на основі числення контекстних термів;

- імплементація швидкого алгоритму співставлення.

Заначимо, що множинний терм з набором правил можна вважати стандартною моделлю як для контекста, так і для системи переписуючих правил без пріоритетів. Застосування системи таких правил є процесом резолвінгу терму в контексті.

Ще однією широкою темою для досліджень є перетворення термів між різними базами контекстів. Тоді, представляючи контекстом операції середовища виконання, можна переформулювати задачу компіляції або інтерпретації терму як перенос терму програми в інший контекст. Якщо маркувати черговість елімінації контексту при перетворенні терму, отримаємо можливість моделювання процесу "постановки"(staging) на кшталт описаного в [12], де рівень мета-змінних відповідає відповідним міткам контексту.

Задачі оптимізації на основі додаткових знань або результатів профілюючого виконання програм також можуть бути описані у цьому фреймворку, якщо розглядати метрики ефективності як складові цільового базового контексту.

Література

1. Jan Willem Klop and Vincent van Oostrom and Femke van Raamsdonk. Combinatory Reduction Systems: introduction and survey. Theoretical Computer Science. Vol. 121. 1993. P. 279–308.
2. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications. Fundamenta Informaticae. 2006. Vol. 72. N 1–3. P. 95–108.
3. Dan Dougherty, Pierre Lescanne, Luigi Liquori, Frédéric Lang. Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics. Electronic Notes in Theoretical Computer Science. Vol. 127, Issue 5, 27 May 2005. P. 57–82.
4. Cirstea, Horatiu and Kirchner, Claude and Liquori, Luigi. Rewriting Calculus with(out) Types. Proceedings of the fourth workshop on rewriting logic and applications. Electronic Notes in Theoretical Computer Science, Sep. 2002. .
5. C. Barry Jay and Delia Kesner. Pure Pattern Calculus. ACM Trans. Program. Lang. Syst. 2005. P. 263–274.
6. Martin Bravenboer and Karl Trygve Kalleberg and Rob Vermaas and Eelco Visser. {Stratego/XT 0.17}. {A} language and toolset for program transformation. Science of Computer Programming. Vol. 72. 2008. P. 52–70.
7. Pfenning, F. and Elliott, C., Higher-order Abstract Syntax. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. PLDI-88. 1998. P. 199–208.
8. Bove, Ana; Peter Dybjer . Dependent Types at Work. LerNet ALFA Summer School. 2008: P. 57–99.
9. Rinus Plasmeijer and Marko van Eekelen, Keep it Clean: A unique approach to functional programming. ACM Sigplan Notices, June 1999.
10. Eric Reed. Patina: A Formalization of the Rust Programming Language. University of Washington. Technical Report. 2015.
11. Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, Andy McNeil, Deny Capabilities for Safe, Fast Actors. Causality Ltd., Imperial College London, 2015.
12. Nada Amin and Tiark Rompf. Collapsing Towers of Interpreters. Proc. ACM Program. Lang. 2, POPL, Article 33 (January 2018). 33 p.

References

1. Jan Willem Klop and Vincent van Oostrom and Femke van Raamsdonk. Combinatory Reduction Systems: introduction and survey. Theoretical Computer Science. Vol. 121. 1993. P. 279–308.
2. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications. Fundamenta Informaticae. 2006. Vol. 72. N 1–3. P. 95–108.
3. Dan Dougherty, Pierre Lescanne, Luigi Liquori, Frédéric Lang. Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics. Electronic Notes in Theoretical Computer Science. Vol. 127, Issue 5, 27 May 2005. P. 57–82.
4. Cirstea, Horatiu and Kirchner, Claude and Liquori, Luigi. Rewriting Calculus with(out) Types. Proceedings of the fourth workshop on rewriting logic and applications. Electronic Notes in Theoretical Computer Science, Sep. 2002. .
5. C. Barry Jay and Delia Kesner. Pure Pattern Calculus. ACM Trans. Program. Lang. Syst. 2005. P. 263–274.
6. Martin Bravenboer and Karl Trygve Kalleberg and Rob Vermaas and Eelco Visser. {Stratego/XT 0.17}. {A} language and toolset for program transformation. Science of Computer Programming. Vol. 72. 2008. P. 52–70.
7. Pfenning, F. and Elliott, C., Higher-order Abstract Syntax. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. PLDI-88. 1998. P. 199–208.
8. Bove, Ana; Peter Dybjer . Dependent Types at Work. LerNet ALFA Summer School. 2008: P. 57–99.
9. Rinus Plasmeijer and Marko van Eekelen, Keep it Clean: A unique approach to functional programming. ACM Sigplan Notices, June 1999.
10. Eric Reed. Patina: A Formalization of the Rust Programming Language. University of Washington. Technical Report. 2015.
11. Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, Andy McNeil, Deny Capabilities for Safe, Fast Actors. Causality Ltd., Imperial College London, 2015.
12. Nada Amin and Tiark Rompf. Collapsing Towers of Interpreters. Proc. ACM Program. Lang. 2, POPL, Article 33 (January 2018). 33 p.

***Про автора*:**

Шевченко Руслан Сергійович,
підприємець.
Кількість наукових публікацій в українських виданнях – 11.
Кількість наукових публікацій в зарубіжних виданнях – 5.

http://orcid.org/0000-0002-1554-2019.

Місце роботи автора:

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