УДК 004.4'24 https://doi.org/10.15407/pp2020.02-03.115

ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ КОДУ МОВОЮ PYTHON З ВИКОРИСТАННЯМ ТЕХНІКИ ПЕРЕПИСУВАЛЬНИХ ПРАВИЛ

К.А. Жереб

Запропоновано підхід підвищення продуктивності коду, написаного мовою Python, шляхом перетворення фрагментів коду до більш ефективних мов Cython та C++. Використано високорівневі алгебраїчні моделі та техніку переписувальних правил для автоматизації перетворень програмного коду. Проведено порівняння часу виконання простих програм – початкової версії мовою Python, різних версій перетвореного коду, а також автоматичних засобів Cython та PyPy, що демонструє ефективність запропонованого підходу.

Ключові слова: підвищення продуктивності коду, техніка переписувальних правил, Python, Cython.

Предложен подход повышения производительности кода, написанного на языке Python, с помощью преобразования фрагментов кода в более эффективные языки Cython и C ++. Использованы высокоуровневые алгебраические модели и техника переписывающих правил для автоматизации преобразований кода. Проведено сравнение времени выполнения простых программ – начальной версии на языке Python, различных версий преобразованного кода, а также автоматических средств Cython и PyPy, что демонстрирует эффективность предложенного подхода.

Ключевые слова: повышение производительности кода, техника переписывающих правил, Python, Cython.

Python is a popular programming language used in many areas, but its performance is significantly lower than many compiled languages. We propose an approach to increasing performance of Python code by transforming fragments of code to more efficient languages such as Cython and C++. We use high-level algebraic models and rewriting rules technique for semi-automated code transformation. Performance-critical fragments of code are transformed into a low-level syntax model using Python parser. Then this low-level model is further transformed into a high-level algebraic model that is language-independent and easier to work with. The transformation is automated using rewriting rules implemented in Termware system. We also improve the constructed high-level model by deducing additional information such as data types and constraints. From this enhanced high-level model of code we generate equivalent fragments of code using code generators for Cython and C++ languages. Cython code is seamlessly integrated with Python code, and for C++ code we generate a small utility file in Cython that also integrates this code with Python. This way, the bulk of program code can stay in Python and benefit from its facilities, but performance-critical fragments of code are transformed into more efficient equivalents, improving the performance of resulting program. Comparison of execution times between initial version of Python code, different versions of transformed code and using automatic tools such as Cython compiler and PyPy demonstrates the benefits of our approach – we have achieved performance gains of over 50x compared to the initial version written in Python, and over 2x compared to the best automatic tool we have tested.

Key words: improving code performance, rewriting rules technique, Python, Cython.

Вступ

Мова Python нині стає все більш популярною у різних сферах розробки програмного забезпечення. Зокрема, ця мова застосовується в наукових обчисленнях, штучному інтелекті та машинному навчанні, для збору та аналізу великих обсягів даних, під час розробки веб-застосунків, для автоматизації дій користувача та в інших галузях [1]. Перевагами мови Python є простота вивчення, швидкість написання коду, наявність багатьох бібліотек та фреймворків з підтримкою широких функціональних можливостей. Проте суттєвим недоліком цієї мови є швидкість виконання коду [2]. Оскільки мова Python в багатьох своїх реалізаціях є інтерпретованою (скриптовою), то час виконання коду є значно більшим порівняно з мовами, які компілюються в машинний код (наприклад, C++ або Fortran) або в байткод чи аналогічне проміжне подання (наприклад, Java або C#). Тому для задач, де важлива ефективність під час виконання, мова Python може використовуватись для створення прототипів, які потім переписуються з використанням більш ефективних мов. Інший підхід полягає у використанні розширень мови Python, що дозволяє використати більш ефективну мову (наприклад, C/C++) для реалізації критичних за швидкодією фрагментів коду, які потім викликаються з інших частин коду, написаних мовою Python. Такий підхід дозволяє досягти балансу між ефективністю під час виконання та продуктивністю роботи програміста.

Реалізація розширень для Python з використанням мови C++ вручну може бути досить складною для розробників – оскільки вимагає детальних знань обох мов, використання спеціальних інтерфейсів взаємодії та написання досить великих обсягів службового коду. З метою спрощення розробки більш ефективних програм мовою Python були розроблені інструментальні засоби. Зокрема, мова Cython [3, 4] дозволяє писати ефективний код з синтаксисом, близьким до мови Python, і при цьому ефективністю виконання близькою до мови C++. Також використання мови Cython спрощує підключення до програми мовою Python компонентів чи розширень, написаних мовою C++. Проте розробнику все одно доводиться вивчати нову мову, та враховувати особливості її реалізації. Існують також автоматичні засоби підвищення ефективності виконання коду мовою Python. Зокрема, досить популярним є засіб PyPy [5] – це альтернативна реалізація мови Python, що використовує JIT-компіляцію замість інтерпретації. Використання цього засобу дозволяє виконувати існуючий код більш ефективно, не вимагаючи внесення змін чи використання інших мов [2, 6]. Проте швидкість виконання коду з

використанням PyPy все ж поступається швидкості коду, написаного з використанням більш ефективних мов програмування [7].

В даній роботі запропоновано підхід для автоматизованого (напівавтоматичного) перетворення фрагментів коду мовою Python в еквівалентні за функціональністю фрагменти коду мовою Cython або C++ з метою підвищення ефективності коду за часом виконання. Для цього використовується техніка переписувальних правил, зокрема система Termware, а також підхід побудови високорівневих алгебраїчних моделей коду.

1. Техніка переписувальних правил та система Termware

В даній роботі для автоматизації перетворень програмного коду використано техніку переписувальних правил, реалізовану в системі TermWare [8, 9]. Це дозволяє описувати перетворення програм у декларативному вигляді, що спрощує їх розробку та повторне використання.

Переписувальні правила описуються з використанням синтаксису мови Termware і виконуються за допомогою інструментальних засобів Termware. Основою мови є терми, тобто деревоподібні вирази виду $f(x_1,...,x_n)$. Тут x_i є або атомарними термами (константи або змінні, які записуються у вигляді $var), або іншими термами. Правила Termware мають наступний загальний вигляд:

source [condition] -> destination [action]

Тут використовуються чотири терми:

- *source* – вхідний зразок;
- *destination* – вихідний зразок;
- *condition* – умова, що визначає застосовність правила;
- *action* – дія, виконувана при спрацьовуванні правила.

Виконувані дії і умови, що перевіряються є необов'язковими компонентами правила. Вони можуть викликати імперативний код, що міститься в БД фактів [8]. За рахунок цього базова модель переписування термів може бути розширена довільними додатковими можливостями.

При використанні техніки переписувальних правил для перетворень коду програм необхідно перетворити код з текстового подання в деревоподібну структуру, яка може бути записана у вигляді термів. Для цього спершу використовується синтаксичний аналізатор (парсер) мови програмування. В результаті можна отримати дерево синтаксичного розбору, яке можна розглядати як низькорівневу синтаксичну модель коду. Ця модель є не дуже зручною для використання, до того ж модель залежить від мови програмування та реалізації синтаксичного аналізатора. Тому в даній роботі використано підхід перетворення цієї моделі до високорівневої алгебраїчної моделі коду [10, 11]. Високорівнева модель є меншою за обсягом та більш зручною для обробки. Вона також не залежить від мови програмування, тому може бути використана для переходу до інших мов програмування. Перехід від низькорівневої синтаксичної моделі до високорівневої алгебраїчної моделі автоматизується з використанням переписувальних правил. Після обробки та перетворень високорівневої моделі з неї генерується код цільовою мовою (яка може співпадати або не співпадати з початковою мовою), для чого використовується генератор коду відповідної мови [12, 13].

2. Побудова низькорівневої синтаксичної моделі коду

Як приклад коду мовою Python розглянемо реалізацію простого алгоритму пошуку простих чисел [14]. Початковий код має наступний вигляд:

```
def primes_python(nb_primes):
    p = []
    n = 2
    while len(p) < nb_primes:
        for i in p:
            if n % i == 0:
                break
        else:
            p.append(n)
        n += 1
    return p
```

Код складається з двох вкладених циклів. Зовнішній цикл while відповідає за перевірку послідовних чисел n, поки загальна кількість знайдених простих чисел не перевищить параметр nb_primes. Внутрішній цикл for перевіряє, чи ділиться задане число n на раніше знайдені прості числа (які зберігаються у списку p).

Даний код демонструє можливості мови Python, які відрізняються від базових можливостей інших мов, зокрема C-подібних мов (наприклад, C++, Java, C#, …). Базовий цикл for в Python є не циклом з лічильником, а

циклом типу ”for-each”, що перебирає всі елементи заданого списку. Аналогічні цикли доступні і в інших сучасних мовах програмування, проте там вони є додатковими до циклів з лічильником, тоді як в Python це основна циклічна конструкція. Інша можливість мови Python – це блок else в циклі for. Цей блок виконується в тому випадку, якщо всі ітерації відповідного циклу виконались повністю, а не були завершені оператором break. Подібні можливості відсутні в багатьох інших мов програмування. Подібні особливості мов програмування ускладнюють автоматичне чи автоматизоване перетворення коду з однієї мови на іншу.

На основі початкового коду програми будується низькорівнева синтаксична модель коду. Для цього використовується синтаксичний аналізатор (парсер) мови Python. Для даного прикладу низькорівнева синтаксична модель має наступний вигляд:

```
FunctionDef( name='primes', args=arguments(args=[arg(arg='nb_primes', annotation=None)], vararg=None, kwonlyargs=[], kw_defaults=[], kwarg=None, defaults=[], ),
  body=[Assign( targets=[Name(id='p', ctx=Store())], value=List(elts=[], ctx=Load()), ),
    Assign( targets=[Name(id='n', ctx=Store())], value=Num(n=2), ),
    While( test=Compare( left=Call( func=Name(id='len', ctx=Load()), args=[Name(id='p', ctx=Load())], keywords=[], ), ops=[Lt()], comparators=[Name(id='nb_primes', ctx=Load())], ),
      body=[ For( target=Name(id='i', ctx=Store()), iter=Name(id='p', ctx=Load()), body=[
        If( test=Compare( left=BinOp( left=Name(id='n', ctx=Load()), op=Mod(), right=Name(id='i', ctx=Load()), ), ops=[Eq()], comparators=[Num(n=0)], ), body=[Break()], orelse=[],), ],
        orelse=[Expr(value=Call(func=Attribute( value=Name(id='p', ctx=Load()), attr='append', ctx=Load(), ), args=[Name(id='n', ctx=Load())], keywords=[],),),], ),
      AugAssign( target=Name(id='n', ctx=Store()), op=Add(), value=Num(n=1),),],orelse=[],),
    Return(value=Name(id='p', ctx=Load()),),], decorator_list=[],returns=None,0)
```

Побудована модель – досить велика за обсягом. Вона містить багато деталей, які можуть бути корисними в загальному випадку – проте для даного конкретного алгоритму є не дуже потрібними. Тому з такою моделлю важко працювати.

Для побудови моделі, більш придатної для подальшої обробки та перетворень, можна було б реалізувати спеціалізований синтаксичний аналізатор, який би генерував лише ті дані, які надалі будуть необхідними. Проте це означає, що треба буде розробити спеціалізований синтаксичний аналізатор, що є досить складною задачею. Також будувати новий спеціалізований синтаксичний аналізатор може знадобитись неодноразово, оскільки для вирішення різних задач потрібні різні дані про початковий код.

Тому в даній роботі використано більш універсальний підхід. Цей підхід полягає в автоматизованому перетворенні низькорівневої синтаксичної моделі у високорівневу алгебраїчну модель, придатну для вирішення даної задачі. Для автоматизації перетворень використовується техніка переписувальних правил, зокрема спеціальні правила – патерни [12]. В загальному випадку патерн складається з двох правил r_p та r_g. Правило r_p застосовується до низькорівневої синтаксичної моделі і замінює певні її конструкції на більш високорівневі аналоги. Правило r_g працює у зворотньому напрямку – відтворює елементи низькорівневої моделі з конструкцій високорівневої алгебраїчної моделі. В більшості випадків патерни мають вигляд $t_L \leftrightarrow t_H$, тобто встановлюється безпосередня відповідність між елементами низькорівневої моделі t_L та високорівневої моделі t_H. В цьому випадку відповідні правила мають вигляд $r_p = t_L \rightarrow t_H$ та $r_p = t_H \rightarrow t_L$.

Як приклад використання патернів, розглянемо деякі з конструкцій у наведеному прикладі коду. Почнемо з часто вживаного оператору присвоювання:

1. Assign(targets=[Name(id=$name, ctx=Store())], value=$value) <-> Assign(Var($name), $value)

Цей патерн знаходить спеціальний випадок присвоювання – коли в лівій частині знаходиться змінна, і використовується лише одне присвоювання (тобто вираз на зразок x = a на відміну від x = y = a).

Аналогічним чином описуються патерни для стандартних арифметичних операцій та порівнянь:

2. AugAssign(target=Name(id=$name, ctx=Store()), op=Add(), value=Num(n=1)) <-> Increment(Var($name))
3. BinOp(left=$op1, op=Mod(), right=$op2) <-> Mod($op1, $op2)
4. Compare(left=$op1, ops=[Lt()], comparators=[$op2]) <-> Less($op1, $op2)
5. Compare(left=$op1, ops=[Eq()], comparators=[$op2]) <-> Equal($op1, $op2)

Патерн 2 описує операцію збільшення змінної на 1 (інкремент). В Python немає спеціального оператора для інкремента (на відміну від багатьох мов, що підтримують синтаксис n++). Замість нього використовується звичайне присвоювання n+=1. Проте техніка переписувальних правил дає можливість виділити такі спеціальні випадки. Надалі це дає можливість переходити до іншої мови програмування. Це також може бути корисним для розпізнавання певних стандартних алгоритмів. Патерн 3 описує бінарний оператор остачі від ділення

(ділення по модулю), що в багатьох мовах програмування має синтаксис a%b. Аналогічними патернами описуються й інші арифметичні опе-рації. Патерни 4 та 5 описують операції порівняння a<b та a==b. Низькорівнева модель підтримує порівняння з більш ніж двома операндами, наприклад a<b<c чи a==b==c. Але такі порівняння зустрічаються не дуже часто, тому у високорівневій алгебраїчній моделі вони розглядаються як окремий випадок. Це дозволяє спростити структуру моделі у більш популярних випадках.

Далі розглянемо патерни для змінних та констант базових типів даних:

6. Name(id=$name, ctx=Load()) <-> Var($name)
7. Num(n=$value) [isInteger($value)] <-> Integer($value)

Патерн 6 працює для тих випадків, де змінна використовується для отримання значення, але її значення не змінюється. Ситуації, в яких значення змінної модифікується, описуються патернами на зразок 1 та 2. Патерн 7 описує цілочисельні константи. В низькорівневій моделі коду Python не розрізняються цілі та дійсні числові константи. Тому для перетворення до високорівневої алгебраїчної моделі використано перевірку isInteger($value), яка викликає відповідний метод БД фактів [8]. Аналогічні патерни використовуються і для інших базових типів.

Мова Python також підтримує декілька стандартних типів-контейнерів. Зокрема, в даному прикладі використовуються списки. Використання патернів дозволяє описати основні операції для роботи з такими типами даних:

8. List(elts=$items, ctx=Load()) <-> List($items)
9. Call(func=Var('len'), args= [$list], keywords=[]) <-> Length($list)
10. Expr(value=Call(func=Attribute(value= $list, attr='append', ctx=Load()), args=[$item], keywords=[])) <-> Append($list, $item)

Патерн 8 описує створення списку з фіксованим набором початкових значень. В даному прикладі створюється пустий список, але патерн підтримує і більш загальний випадок. Патерн 9 описує операцію отримання довжини списку. Патерн 10 описує операцію додавання одного елеме-нта в кінець списку. Патерни 9 і 10 демонструють, що деякі операції можуть бути реалізовані по-різному в цільовій мові програмування – довжина списку в Python отримується викликом глобальної функції len($list), а додавання елементу в кінець списку – викликом методу $list.append($item). Високорівнева алгебраїчна модель дозволяє абстрагуватись від таких деталей реалізації, характерних для окремих мов програмування. Це спрощує роботу з високорівневою моделлю, а також дозволяє переходити до інших мов програмування.

Розглянемо патерни для стандартних конструкцій керування, а саме умов та циклів:

11. If(test=$condition, body=$then, orelse=[]) <-> If($condition, $then)
12. While(test=$condition, body=$body, orelse=[]) <-> While($condition, $body)
13. For(target=Name(id=$name, ctx= Store()), iter=$iterable, body=$body, orelse=$else) <-> IfNoBreak(ForEach(Var($name), $iterable, $body),$else)

Патерн 11 описує умовну конструкцію if без блоку else. Патерн 12 описує цикл з умовою типу while. Патерн 13 описує цикл типу for-each, який обходить всі елементи заданого контейнера та виконує тіло циклу для кожного з цих елементів. В цьому патерні реалізована підтримка блоку else, характерної особливості циклів мови Python. Для підтримки такої конструкції у високорівневій алгебраїчній моделі введено нову конструкцію IfNoBreak($loop,$else). Альтернативою цьому підходу було б створення розширених конструкцій для циклів, що підтримують додаткові параметри. Проте запропонований підхід з реалізацією окремої конструкції дозволяє підтримати різні типи циклів (while, for, for-each, …) з використанням лише однієї нової конструкції, а не розширювати кожну з циклічних конструкцій. Також такий підхід спрощує реалізацію цієї конструкції в інших мовах програмування, які явно не підтримують блок типу else в циклах.

Нарешті, розглянемо патерни для роботи з функціями, а саме для опису функцій в програмному коді:

14. arg(arg=$name, annotation=None) <-> Arg($name)
15. FunctionDef(name=$name, args= arguments(args=$args, vararg=None, kwonlyargs=[], kw_defaults=[], kwarg= None, defaults=[]), body=$body) <-> Function($name,$args, $body)

Патерн 14 описує окремі аргументи функції. Патерн 15 описує визначення функції в цілому, в простому випадку, коли немає спеціальних типів аргументів (підтримка змінної кількості аргументів, значення за замовченням).

Описані патерни застосовуються до низькорівневої синтаксичної моделі коду в напрямку генерації високорівневої моделі – тобто з кожного патерну виділяється відповідне правило r_p, і набір цих правил застосовується до терму, що описує низькорівневу модель. В результаті отримано наступний терм:

```
Function("primes",
  [Arg("nb_primes")], [
  Assign(Var("p"),List([])), Assign(Var("n"),Integer(2)),
  While(Less(Length(Var("p")), Var("nb_primes")),[
    IfNoBreak( ForEach( Var("i"), Var("p"), [
      If(Equal( Mod(Var("n"), Var("i")), Integer(0)), [Break()])
    ]), [Append(Var("p"),Var("n"))]),
    Increment(Var("n"))
  ]) ,
  Return(Var("p))
])
```

Отримана високорівнева алгебраїчна модель значно менша за обсягом порівняно з низькорівневою синтаксичною моделлю. Алгебраїчна модель ближча до початкового коду. При цьому вона описує алгоритм, який не залежить від мови реалізації. Тому цей алгоритм можна буде перетворити в реалізації іншими мовами програмування.

Варто звернути увагу на наступну особливість наведених патернів. У деяких з них з обох боків використовуються терми з однаковими іменами. Наприклад, це стосується патерну 1 (терм Assign), патерну 8 (терм List), патерну 11 (терм If), патерну 12 (терм While). В загальному випадку використання техніки переписувальних правил, наявність термів з однаковими іменами з обох боків правила може викликати проблеми. Після застосування правила, те ж саме правило може знову спрацювати, що призведе до «зациклювання» процесу переписування. Щоб уникнути цієї проблеми, реалізована така модифікація: якщо в патерні з обох боків зустрічаються терми з однаковими іменами та арністю (кількістю аргументів), до перетвореного терма спершу додається модифікатор _pat, який потім прибирається окремим правилом. Таким чином, патерн 1 буде реалізовано у вигляді наступних правил:

16. Assign(targets=[Name(id=$name, ctx=Store())], value=$value) ->
    Assign_pat(Var($name), $value)
17. Assign_pat($op1,$op2) ->Assign($op1, $op2)

Правило 16 використовується під час першого етапу перетворення низькорівневої синтаксичної моделі у високорівневу алгебраїчну модель. Правило 17 запускається окремо, після того як всі правила першого етапу вже відпрацювали. Таким чином, коли працює правило 17, то правило 16 вже не є активним. Це дозволяє уникнути потенційного нескінченного застосування подібних правил.

3. Розширення алгебраїчної моделі додатковими даними

Побудована алгебраїчна модель повністю описує початковий код мовою Python, з якого вона була згенерована. Проте для подальшої роботи з моделлю знадобляться інші дані, які не були явно представлені в початковому коді програми. Зокрема, це стосується типів даних.

Мова Python традиційно використовує динамічну типізацію, тому в початковому коді не описуються типи змінних, аргументів функцій і так далі. В останніх версіях мови Python з’явилась можливість вказувати типи даних для окремих змінних та аргументів функцій [15]. Проте типи даних не є обов’язковими, і їх відсутність не викликає помилок під час розробки. Тому в багатьох проектах мовою Python типи даних не вказані – особливо це стосується коду, написаного певний час назад (підтримка анотацій типів з’явилась у версії Python 3.5, яка була випущена у вересні 2015 року).

Багато інших мов програмування вимагають вказувати типи даних для всіх змінних, аргументів функцій, полів класів і т.д. Зокрема, це стосується мови C++. Мова Cython може працювати з будь-яким кодом мовою Python, тому опис типів даних в цій мові не є обов’язковими. Проте для підвищення продуктивності коду Cython статична типізація має суттєве значення. Тому можливість додати до алгебраїчної моделі інформацію про типи даних окремих її елементів дозволить згенерувати більш ефективний код.

Інформація про типи даних змінних може бути отримана з прикладів їх використання. Для цього використовуються переписувальні правила наступного вигляду:

1. Assign(Var($name), Integer($value)) -> _same_ [saveType($name, Integer)]
2. Assign(Var($name),List([])) -> _same_ [saveType($name, List_partial(Unknown))]

Правило 1 визначає тип змінної, якій присвоюється цілочисельне значення – в цьому випадку використовується метод saveType БД фактів, щоб присвоїти даній змінній тип Integer. Правило 2 працює аналогічно, проте для типу даних список. В даному випадку змінній присвоюється пустий список, тому визначити тип елементів на даному етапі неможливо. Через це записується тип List_partial(Unknown). Тип елементів списку має бути уточнено подальшими правилами. Аналогічні правила існують і для інших типів даних.

В правилах 1 та 2 використано спеціальний терм _same_. Він використовується в тих випадках, коли правило не змінює модель напряму, а лише додає інформацію в БД фактів. Під час обробки правила, в правій частині якого знаходиться терм _same_, цей терм автоматично замінюється на терм, аналогічній лівій частині правила, проте з додаванням модифікатора _same. Цей модифікатор потім прибирається на наступному етапі застосування правил. Таким чином, правило 1 перетворюється на наступні правила:

3. Assign(Var($name), Integer($value)) -> Assign_same(Var($name), Integer($value)) [saveType($name, Integer)]
4. Assign_same($op1, $op2) -> Assign($op1, $op2)

Правило 3 застосовується лише один раз до кожного елемента моделі, оскільки в його правій частині стоїть вже модифікований терм. Правило 4 виправляє модифікацію, проте воно працює на наступному етапі, коли правило 3 вже не є активним.

Для того, щоб уточнити тип елементів списку, необхідно використання наступного правила:

5. Append(Var($list), $item) [getType($item,$type)] -> _same_ [saveType($list, List($type))]

Правило 5 знаходить елементи моделі, що описують додавання значень в список. Далі викликається метод БД фактів getType, який записує тип значення в $type. Після цього оновлюється тип списку.

Після того, як визначено тип списку, з'являється можливість визначити тип ще однієї змінної, яка використовується для обходу списку. Для цього використовується наступне правило:

6. ForEach(Var($item),Var($list)) [getType($list, List($type))] -> _same_ [saveType($item, $type)]

Правило 6 знаходить цикли типу "for-each", для яких відомий тип списку, і визначає тип змінної на основі типу списку. Варто звернути увагу, що правило 6 не зможе спрацювати після того, як спрацює правило 2, оскільки повний тип списку ще не є відомим. Саме з цією метою в правилі 2 задається тип List_partial, а не просто List.

Серед змінних, які присутні в моделі, залишається невідомим тип аргументу функції nb_primes. Тип цієї змінної можна визначити двома способами: виходячи з викликів функції, або виходячи з використання цього значення в самій функції. В даній роботі використано другий підхід. Перевагою цього підходу є можливість незалежного аналізу окремих функцій, що значно спрощує перетворення. Можливим недоліком є недостатня точність – оскільки в функцію можуть передаватись дані більш загальних типів, ніж можна вивести з використання. Наприклад, в даному випадку використовується порівняння змінної nb_primes з розміром списку, і з цього можна зробити висновок, що тип змінної має бути таким самим, як тип розміру (тобто беззнакове ціле число). Проте можлива ситуація, коли змінна має інший тип, наприклад дійсне значення – при цьому порівняння все одно буде коректним. Одним з напрямків подальших досліджень є реалізація обох підходів та їх порівняння.

Для реалізації визначення типу змінної виходячи лише з її використання в тілі функції використовується наступне правило:

7. Less(Length($list), Var($len)) -> _same_ [saveType($len, UnsignedInteger)]

Правило 7 шукає порівняння певної змінної з довжиною списку, і присвоює цій змінній тип UnsignedInteger. Аналогічні правила можна застосувати до інших операторів порівняння.

Після застосування всіх правил в БД фактів зберігається інформація про типи змінних в даній функції (табл. 1).

Таблиця 1. Типи змінних в моделі

Ім'я змінної	**Тип в БД фактів**
n	Integer
p	List(Integer)
i	Integer
nb_primes	UnsignedInteger

Окрім визначення типів змінних, переписувальні правила можуть додавати до моделі й інші дані. Зокрема, в даному прикладі корисно знати очікуваний розмір списку p. Знаючи очікуваний розмір списку, можна завчасно зарезервувати достатній обсяг пам'яті для зберігання всіх елементів. Це дозволить уникнути копіювання елементів під час зростання розміру списку. В загальному випадку визначення максимального розміру списку може вимагати побудови складних моделей коду. Але для отримання простих оцінок можна використати прості правила, які працюють в окремих випадках. Оскільки неправильна оцінка розміру списку не робить реалізацію неправильною, а лише впливає на продуктивність коду, то подібні правила можуть бути корисними. Зокрема, в даному випадку використовується наступне правило:

8. Less(Length($list), $val) -> _same_ [saveMaxLength($list, $val)]

Це правило шукає порівняння довжини певного списку $list зі значенням $val і зберігає це значення в БД фактів. Якщо подібні порівняння зустрічаються кілька разів, метод saveMaxLength намагається визначити максимальне значення і використовує саме його.

Таким чином, техніка переписувальних правил дозволяє розширити модель програми, додаючи до неї додаткову інформацію, яку можна знайти в моделі або вивести з вже відомих даних. Це дозволяє в подальшому будувати більш ефективні перетворення програм.

4. Перетворення коду в інші мови програмування: Cython та C++

На основі побудованої високорівневої алгебраїчної моделі програми можна згенерувати код еквівалентної програми іншими мовами. В даній роботі розглядаються мови Cython та C++, оскільки фрагменти коду цими мовами можуть бути легко інтегровані в код мовою Python. Це дає можливість перетворити лише невеликі фрагменти коду, які є найбільш критичними з точки зору продуктивності.

В простих випадках структура високорівневої моделі відповідає структурі коду цільовою мовою, тому для перетворення необхідно лише згенерувати відповідні конструкції цільової мови для кожного елементу моделі. Проте якщо початкова та цільова мова є досить далекими, доводиться вносити певні зміни в структуру моделі. Зокрема, у випадку переходу від Python до Cython або C++ необхідно додати декларацію типів змінних, а також більш детальну початкову ініціалізацію, зокрема для списків. Це можна зробити з використанням наступних переписувальних правил:

1. Function($name, $args, $body) -> Function_todo($name,TypedArgs($args,$name), [DeclareVars($name) : $body])
2. TypedArgs([$arg1: $args],$func) -> [TypedArgs($arg1,$func): TypedArgs($args,$name)]
3. TypedArgs(Arg($name),$func) [getType($name,$func,$type)] -> Args($name,$type)
4. DeclareVars($name) [getTypedVars($name, $declarations)] -> $declarations
5. Assign($list,List([])) [getMaxLength($list,$len)] -> ReserveLength($list, $len)

Правило 1 додає службові терми TypedArgs для заповнення інформації про типи аргументів функції, а також DeclareVars для декларації локальних змінних. Правила 2–3 описують розширення терму TypedArgs. Правило 2 описує, як цей терм послідовно застосовується до елементів списку аргументів. Правило 3 демонструє додавання до аргументу інформації про типи, яка видобувається з БД фактів. Правило 4 описує створення декларації змінних на основі даних з БД фактів. Правило 5 описує додаткову ініціалізацію списків – замість створення порожнього списку, який надалі буде зростати, одразу резервується пам'ять для певної кількості елементів, яка теж береться з БД фактів.

Після застосування цих правил модель даної функції виглядає наступним чином:

```
Function("primes",
  [Arg("nb_primes", UnsignedInteger)], [
  Declare(Var("n"), Integer),  Declare(Var("i"), Integer),  Declare(Var("p"), List(Integer)),
  ReserveLength(Var("p"), Var("nb_primes")),
  Assign(Var("n"),Integer(2)),
  While(Less(Length(Var("p")), Var("nb_primes")),[
    IfNoBreak( ForEach( Var("i"), Var("p"), [
      If(Equal( Mod(Var("n"), Var("i")), Integer(0)), [Break()])
    ]), [Append(Var("p"),Var("n"))]),
    Increment(Var("n"))
  ]) ,
  Return(Var("p"))
])
```

Високорівнева модель зазнала незначних змін, пов'язаних з необхідністю підтримки нових мов. При цьому, за необхідності генерації коду мовою Python, ці додаткові можливості можуть бути або ігноровані, або використані для генерації анотацій типів, коментарів чи інших конструкцій коду, які не використовуються під час компіляції та виконання коду, але роблять код більш зрозумілим для розробника. Таким чином, високорівнева алгебраїчна модель все одно залишається незалежною від мови програмування, тобто немає необхідності підтримувати різні види моделі для різних мов (навіть якщо можливості цих мов суттєво відрізняються, зокрема мови з динамічною типізацією на зразок Python, мови з необов'язковою статичною типізацією на зразок Cython та мови з обов'язковою статичною типізацією на зразок C++).

Використовуючи розширену високорівневу модель, може бути згенеровано код відповідної функції мовою Cython. Цей код має наступний вигляд:

```
def primes(unsigned int nb_primes):
    cdef int n, i
    cdef vector[int] p
    p.reserve(nb_primes)
    n = 2
    while p.size() < nb_primes:
        for i in p:
            if n % i == 0:
                break
        else:
            p.push_back(n)
        n += 1
    return p
```

Загальна структура коду аналогічна початковому коду мовою Python. Це робить код більш зрозумілим для розробників, які знають мову Python. Проте в процесі генерації коду деякі конструкції отримали іншу реалізацію (табл. 2).

Таблиця 2. Елементи моделі з різними реалізаціями в мовах Python та Cython

Елемент моделі	**Реалізація в Python**	**Реалізація в Cython**
Length(Var("p"))	`len(p)`	`p.size()`
Append(Var("p"),Var("n"))	`p.append(n)`	`p.push_back(n)`

Для генерації коду мовою C++ використовується та ж високорівнева модель, хоча текстове подання відповідних елементів коду відрізняється. Згенерований код функції виглядає наступним чином:

```
std::vector<int> primes(unsigned int nb_primes){
    int n, i;
    std::vector<int> p;
    p.reserve(nb_primes);
    n = 2;
    while (p.size() < nb_primes) {
        bool found = false;
        for(int i:p) {
            if (n % i == 0) {
                found = true;
                break;
            }
        }
        if (!found) {
            p.push_back(n);
        }
        n += 1;
    }
    return p;
}
```

Структура коду аналогічно реалізації на Cython. Проте є відмінність, викликана відсутністю підтримки блоку else в циклі for. Відповідний елемент моделі IfNoBreak реалізовано мовою C++ з використанням

додаткової логічної змінної `bool found`, яка має значення false після виходу з циклу, лише якщо в тілі циклу не спрацював оператор break.

Крім генерації коду функції відповідними мовами, генератор коду підтримує також додаткові налаштування. Зокрема, для кожної мови налаштовується розширення файлів з кодом, а саме .pyx для Cython та .h для C++. Також для використання коду мовою C++ з програми мовою Python, генерується додатковий файл з розширенням .pyx, що містить опис відповідних функцій:

```
cdef extern from "primes.h":
  vector[int] primes(unsigned int nb_primes)
```

Таким чином, згенерований код мовою Cython або C++ може бути безпосередньо використаний з програм мовою Python. Це дозволяє перетворювати лише найбільш критичні за продуктивністю фрагменти коду, залишаючи більшість коду програми незмінним.

5. Експериментальна перевірка підходу

Для перевірки запропонованого підходу були проведені виміри часу виконання різних версій однієї програми для обчислення простих чисел. Розглядалась початкова версія програми мовою Python (ця версія позначена Python), а також дві перетворені програми – мовою Cython (позначена Cython) та мовою C++ (позначена CPP). Також для порівняння розглянуто два підходи для автоматичного підвищення продуктивності коду мовою Python. Перший з цих підходів полягає в компіляції коду мовою Python з використанням засобів Cython (ця версія позначена Compiled). Другий підхід використовує JIT-компілятор PyPy (ця версія позначена PyPy).

Всі виміри проводились на персональному комп'ютері (Intel Core i7-8550U, 16 Гб RAM, ОС Windows 10). Вимірювався час виконання функції primes залежно від кількості простих чисел, які треба знайти (аргумент функції nb_primes), для значень від 1000 до 16000 з кроком 1000. Результати вимірів показано на рис. 1–3.

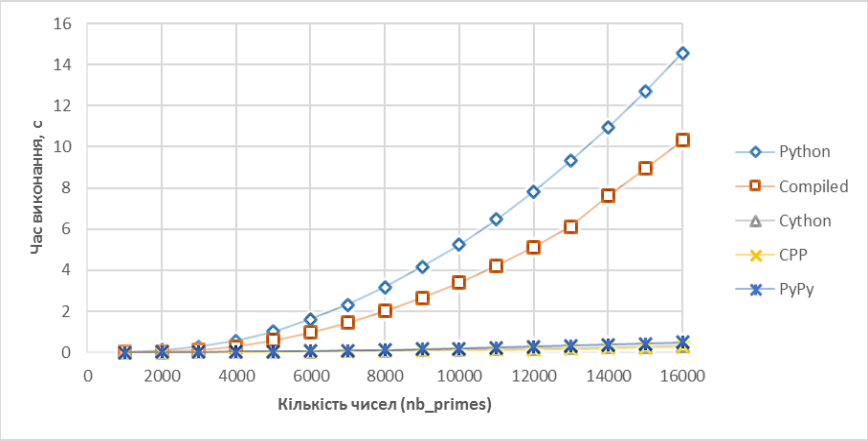

Рис. 1. Порівняння часу виконання різних версій програми

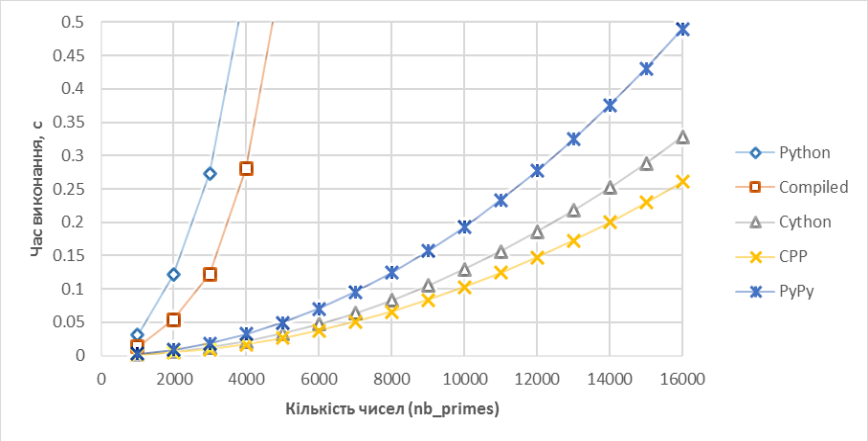

Рис. 2. Деталі порівняння часу виконання різних версій програми

На рис. 1 показано залежність часу виконання (в секундах) від кількості простих чисел для різних версій програми. Як видно з результатів вимірів, початкова версія Python є найбільш повільною. Версія Compiled є трохи швидшою (в 1,5–2 рази). Всі інші версії є набагато швидшими. Тому вони показані на рис. 2, який показує ту саму залежність часу виконання від кількості простих чисел, але з меншим кроком за часом. Як видно з рис. 2, всі три версії PyPy, Cython та CPP є набагато більш ефективними – час їх виконання для 16000 чисел не перевищує 0,5 с, порівняно з часом виконання початкової версії Python (близько 15 с). З цих трьох версій найбільш ефективною в більшості випадків є CPP. Версія Cython є трохи повільнішою (хоча для маленької кількості чисел час її виконання близький та навіть іноді менший за версію CPP). Версія PyPy є менш ефективною, ніж обидві перетворені програми. На рис. 3 зображено коефіцієнт прискорення різних версій порівняно з початковою версією Python. Для великих значень параметру nb_primes коефіцієнт прискорення версії Compiled становить 1,5, для версії PyPy – 30, для версії Cython – 45 і для найбільш ефективної версії CPP – 55 разів.

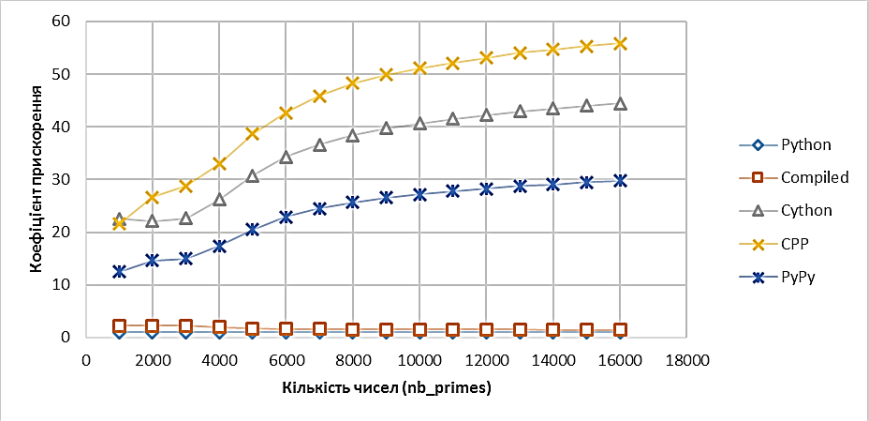

Рис. 3. Коефіцієнти прискорення для різних версій програми

Висновки

В роботі запропоновано підхід до підвищення ефективності коду мовою Python, заснований на використанні техніки переписувальних правил та високорівневих алгебраїчних моделей коду. Критичні за швидкодією фрагменти коду перетворюються до більш ефективних мов Cython та C++, що дозволяє суттєво підвищити продуктивність під час виконання. При цьому перетворені фрагменти викликаються з існуючого коду мовою Python, що спрощує їх використання. Проведені експерименти демонструють ефективність запропонованого підходу, порівняно як з початковим кодом мовою Python, так і з використанням автоматичних засобів – компіляції засобами Cython та використанням альтернативної реалізації мови PyPy. Подальші напрямки досліджень в цій галузі включають розробку перетворень для програм з інших предметних галузей, а також перевірку запропонованого підходу на більш складних програмних системах. Також планується використання аналогічного підходу для роботи з паралельними програмами на сучасних паралельних платформах.

Література

1. Gries P., Campbell J. and Montojo J. 2017. Practical programming: an introduction to computer science using Python 3.6. Pragmatic Bookshelf.
2. Roghult A. 2016. Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy.
3. Herron P. 2016. Learning Cython Programming. Packt Publishing Ltd.
4. Cython: C-Extensions for Python [Електронний ресурс]. Режим доступу: https://cython.org/. 25.02.2020 р.
5. PyPy: a fast, compliant alternative implementation of Python [Електронний ресурс]. Режим доступу: https://www.pypy.org/. 25.02.2020 р.
6. Marowka A. 2018. Python accelerators for high-performance computing. *The Journal of Supercomputing*, 74(4), P. 1449–1460.
7. Fua P. and Lis K. 2020. Comparing Python, Go, and C++ on the N-Queens Problem. arXiv preprint arXiv:2001.02491.
8. Doroshenko A., Shevchenko R. A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae. 2006. Vol. 72, N 1–3. P. 95–108.
9. Termware. [Електронний ресурс]. Режим доступу: http://www.gradsoft.com.ua/products/termware_eng.html. 25.02.2020.
10. Андон Ф.И., Дорошенко А.Е., Жереб К.А., Шевченко Р.С., Яценко Е.А. Методы алгебраического программирования. Формальные методы разработки паралелльных программ. Киев, "Наукова думка". 2017. 440 с.
11. Дорошенко А.Е., Жереб К.А. Алгебро-динамические модели для распараллеливания программ. *Проблемы программирования*. 2010. № 1. С. 39–55.
12. Жереб К.А. Программный инструментарий, основанный на правилах, для автоматизации разработки приложений на платформе Microsoft .NET. *Управляющие системы и машины*. 2009. № 4. С. 51–59.
13. Дорошенко А.Ю., Хаврюченко В.Д., Туліка Є.М., Жереб К.А. Перетворення успадкованого коду на Fortran до масштабованого паралелізму і хмарних обчислень. *Проблеми програмування* (матеріали конференції УкрПРОГ'2016). 2016 № 2–3. С. 133–140.

14. Cython Tutorial. [Електронний ресурс]. Режим доступу: https://cython.readthedocs.io/en/latest/src/ tutorial/cython_tutorial.html. 25.02.2020 р.
15. Rossum van G., Lehtosalo J., Langa Ł. PEP 484 Type Hints [Електронний ресурс]. Режим доступу: https://www.python.org/dev/peps/pep-0484/. 25.02.2020 р.

References

1. Gries, P., Campbell, J. and Montojo, J., 2017. Practical programming: an introduction to computer science using Python 3.6. Pragmatic Bookshelf.
2. Roghult, A., 2016. Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy.
3. Herron, P., 2016. Learning Cython Programming. Packt Publishing Ltd.
4. Cython: C-Extensions for Python [Online] Available from: https://cython.org/. [Accessed: 25th February 2020]
5. PyPy: a fast, compliant alternative implementation of Python [Online] Available from: https://www.pypy.org/. [Accessed: 25th February 2020]
6. Marowka, A., 2018. Python accelerators for high-performance computing. The Journal of Supercomputing, 74(4). P. 1449–1460.
7. Fua, P. and Lis, K., 2020. Comparing Python, Go, and C++ on the N-Queens Problem. arXiv preprint arXiv:2001.02491.
8. Doroshenko A., Shevchenko R. A Rewriting Framework for Rule-Based Programming Dynamic Applications. Fundamenta Informaticae. 2006. Vol. 72, N 1–3. P. 95–108.
9. Termware. [Online] Available from: http://www.gradsoft.com.ua/products/termware_eng.html. [Accessed: 25th February 2020]
10. Andon P.I., Doroshenko A.Yu., Zhereb K.A., Shevchenko R.S., Yatsenko O.A., Methods of Algebraic Programming. Formal methods of parallel program development. Kyiv, "Naukova dumka". 2017. 440 p.
11. Doroshenko A.Yu., Zhereb K.A. Algebra-dynamic models for program parallelization. Problems in Programming. 2010. N 1. P. 39–55.
12. Zhereb K.A. Rule-based software toolset for automating application development on Microsoft .NET Platform. Control systems and machines. 2009. N 4. P. 51–59.
13. Doroshenko A.Yu., Khavryuchenko V.D., Tulika Ye.M., Zhereb K.A. Transformation of the legacy code on Fortran for scalability and cloud computing. Problems in Programming. 2016. N 2–3. P. 133–140.
14. Cython Tutorial. [Online] Available from: https://cython.readthedocs.io/en/latest/src/ tutorial/cython_tutorial.html. [Accessed: 25th February 2020]
15. Rossum van G., Lehtosalo J., Langa Ł. PEP 484 Type Hints [Online] Available from: https://www.python.org/dev/peps/pep-0484/. [Accessed: 25th February 2020]

Одержано 10.03.2020

***Про автора*:**

Жереб Костянтин Анатолійович,
кандидат фізико-математичних наук, асистент,
старший науковий співробітник відділу теорії комп'ютерних обчислень
Інституту програмних систем НАН України.
Кількість наукових публікацій в українських виданнях – понад 40.
Кількість наукових публікацій в зарубіжних виданнях – понад 10.
Індекс Хірша – 3.
http://orcid.org/0000-0003-0881-2284.

Місце роботи автора:

Київський національний університет імені Тараса Шевченка,
кафедра інтелектуальних програмних систем,
03022, Україна, м. Київ,
проспект Академіка Глушкова, 4д.
Тел.: +380 (44) 259-05-11.
E-mail: zhereb@gmail.com.