УДК 628.321

В. Заяць
Національний університет “Львівська політехніка”,
кафедра загальної екології та екоінформаційних систем

ОСНОВНІ ПІДХОДИ ДО ОПТИМІЗАЦІЇ ФУНКЦІОНАЛЬНИХ ПРОГРАМ

Розглянуто основні підходи до оптимізації функціональних програм. Показано особливості та переваги кожного з них. Кожен із підходів ілюструється прикладами реалізації програм у середовищі Ліспу.

Ключові слова: функціональна програма, оптимізація, метод рекурсії, параметри нагромадження.

The basic approaches to optimization of functional programs are considered in the article. Features and advantages of each of them are shown. Each of approaches is illustrated by the examples of realization of the programs in the environment of Lisp.

Key words: functional program, optimization, recursion method, the parameters of accumulation.

Вступ

У роботі розглянуто основні підходи до оптимізації новостворених функціональних програм з метою підвищення ефективності (швидкодія, технічні засоби) та надійності їх роботи.

Доцільність використання описаних підходів ілюструється конкретними прикладними розробками, які реалізовано в середовищі стандартного Ліспу [1], що стосується декларативних мов програмування.

Конструювання сучасних мов програмування сьогодні далеке від досконалості. Кожна з відомих мов має свої недоліки та переваги. Для визначення доцільності використання тієї чи іншої мови доцільно враховувати такі міркування:

1. **Ясність, простота і узгодженість понять мови**. Очевидно, потрібно уникати тонких і каверзних обмежень мови. Вона не повинна бути також двозначною. Семантична ясність мови – це те, що визначає її цінність.

2. **Ясність структури програми.** Ця вимога забезпечується синтаксичною ясністю програм, записаних цією мовою. Мова повинна бути такою, щоб конструкції, які відрізняються семантично, відрізнялись і синтаксичним записом.

Дуже важливо, щоб структура програми відображала структуру самого алгоритму, що дозволяється під час розроблення програми дотримання принципів структурного програмування, ієрархічного конструювання програм – згори донизу. За такого підходу структура програми легкозрозуміла, доступна для діагностики, модифікації та оптимізації.

3. **Природність у використанні.** Мова повинна забезпечувати при розв’язанні задачі найвдаліші структури даних, операції, керуючі структури і легко зрозумілий синтаксис. Все це істотно спрощує створення програмних засобів у заданій галузі знань чи технічних застосувань.

4. **Легкість розширення**. Програмні засоби, які створюються цією мовою, можна розглядати як розширення мови. У принципі більшість мов програмування дає програмісту механізми для створення підпрограм. Тим не менше, властивості самої мови можуть полегшити або ускладнити їх використання. Ця легкість розширення найбільше відчутна у мовах програмування, які мають ідентичне подання даних та програм.

5. **Зовнішнє забезпечення**. Це один із аспектів, який впливає на ефективність використання мови. За наявності потужних засобів тестування, редагування, збереження, модифікації програмних засобів можна зробити слабку мову зручнішою для експлуатації, ніж потужна мова без відповідного технічного забезпечення.

6. **Ефективність** створення, тестування, трансляції, виконання, модифікації та використання програм.

7. **Можливість оптимізації** новостворених програм.

Тим не менше врахування цих чи будь-яких інших міркувань не унеможливлює необхідність освоєння більше ніж однієї мови програмування, як з суто пізнавального погляду, так і з метою досконалого і глибокого вивчення кожної з них. Лише за такого підходу програмування може бути перетворене з мистецтва у строгу наукову дисципліну.

Переваги декларативних функціональних мов над процедурними

Більшість сучасних мов програмування є універсальними, оскільки дають змогу записати будь-який алгоритм цією мовою, якщо не накладати обмежень на час виконання програми та місткість пам'яті. Якщо хто-небудь запропонує нову мову програмування, то вона, очевидно, буде універсальною, якщо ігнорувати обмеження на пам'ять або час. Порівнюючи різні мови програмування, потрібно враховувати не кількісне співвідношення того, що вони дають змогу зробити, а якісні відмінності, що визначають елегантність (короткість і наочність), легкість (прозорість) та ефективність (швидкодія та технічні засоби) програмування на них, здатність до оптимізації. Це порівняння потрібно здійснювати в контексті конкретної сфери застосування.

Традиційні (алгоритмічні) мови програмування є доволі об'ємні і громіздкі, бо не дають змоги:

- максимально використовувати можливості сучасної комп'ютерної техніки для забезпечення ефективності програмних засобів;
- ясно і наочно відобразити алгоритми програми, щоб забезпечити легкість перевірки та модифікації останніх.

Строго функціональні мови програмування такі, як S-Lisp [1], R- Lisp [2], Reduce [3], Common Lisp та AutoLisp [4]є доволі простими, і лише цим забезпечують достатньо високий ступінь виразності програм порівняно з традиційними мовами. Низка функціональних програм можуть ефективно працювати на сучасних комп'ютерах, проте не так ефективно, як відповідні програми з оператором присвоєння. Це пов'язано зі структурою архітектури сучасних комп'ютерів. Окрім того, вибір дещо іншої структури представлення даних, ніж це прийнято в інструментальному Ліспі, забезпечує як більшу ясність подання програм, так і підвищення їх ефективності з використанням сучасних комп'ютерів старої архітектури.

З одного боку, сучасні мови програмування повинні ефективно використовувати сучасні машини, а з іншого, ясно виражати програмні алгоритми, щоб полегшити перевірку останніх. Строго функціональна мова, будучи простою за своєю структурою, демонструє вищий ступінь виразності порівняно з традиційними мовами, де існує оператор присвоєння. Це зв'язано, значною мірою, зі способом вибору структур даних. Дещо інший їх вибір можна забезпечити підвищенням ефективності функційних програм і на сучасних комп'ютерах. Зокрема розглянемо проблему використання функцій зі складовим результатом, оскільки тут є важливий елегантний розв'язок.

Одна з фундаментальних властивостей мови програмування, що дає можливість ясно описати обчислення цією мовою – простота семантики мови. Велика перевага функціональної мови полягає в тому, що тут є кілька основних понять, кожне з яких має просту семантику. Зокрема семантика нашої мови розуміється в термінах значень, які мають вирази, але аж ніяк не в термінах дій і послідовності їх використання. Але з практичного погляду було б справедливіше зробити висновок, що строго функціональна мова є надзвичайно елементарною і деякі її розширення значно б підвищили ефективність і ясність деяких класів вирахувань. Очевидно, необхідно розрізняти суто синтаксичні розширення і розширення, які вимагають зміни семантики мови. Зміна семантики вимагає обережності, оскільки це ускладнює розуміння (ясність) вже налагоджених функціональних програм. Те, що синтаксичне розширення дає змогу підвищити ясність виразів і оптимізувати програму, продемонструємо на прикладі функцій зі складовим результатом.

Строго функціональна мова. Метод рекурсії

Коли йдеться про побудову строго функціональної мови, то нові функції чи функціонали будуються на основі деякого базового набору функцій або примітивних функцій [5]. До них належать функції:

CAR(X) – вибору першого елементу із списку X;

CDR(X) – повернення залишку списку X без першого елементу;

CONS(X, У) – конструювання нового списку, де параметр X є першим елементом списку У;

EQ(X, У) – предикат, який істинний у випадку еквівалентності атомів X і У;

ATOM(X) - предикат, який істинний у випадку, якщо X – атом, а не список.

Наявність цих примітивних функцій, подання арифметичних операцій у вигляді функцій та можливість задання рекурсивних (повторних) викликів функцій чи звертань функцій до самої себе (принцип композиції функцій) дозволяє будувати доволі змістовні функціональні програми [6], [7]. Продемонструємо цю процедуру на прикладі побудови функції **з'єднати(x,y),** яка формує новий список , в якому будуть перераховані всі елементи з x і y. До того ж кожен із аргументів може бути як атомом, так списком довільної довжини. Безпосередній аналіз всіх можливих випадків по x і y приводить до такого функціонального означення:

з'єднати (X,Y) ≡ якщо EQ (X, NIL) то
якщо EQ (Y, NIL) то NIL інакше Y
інакше
якщо EQ (Y, NIL) то X інакше
CONS (CAR(X), з'єднати (CDR(X), Y))

Враховуючи, що при **X=NIL з'єднати (X, Y) = Y** незалежно від значення **Y**, означення можна переписати оптимальніше:

з'єднати (X, Y) ≡ якщо EQ (X, NIL) то Y інакше
якщо EQ (Y, NIL) то X інакше
CONS (CAR (Y), з'єднати (CDR (X), Y))

Якщо тепер врахувати, що при **X ≠ NIL** незалежно від **Y** результати функції є **CONS(CAR(X)**, **з'єднати (CDR (X), Y)**, то перевірку по **Y** теж можна опустити і записати визначення функції у вигляді:

з'єднати (X,Y) ≡ якщо EQ (X, NIL) то Y інакше
CONS (CAR (X), з'єднати (CDR (X), Y))

Всі ці три версії означають еквівалентні функції і забезпечують один і той же результат. Першій версії можна віддати перевагу за те, що вона явно перераховує всі можливі випадки. Третій – за її короткість. Але друга і третя версії цієї функції **з'єднати** мають різну ефективність. Друга версія уникає вирахувань у випадку **Y=NIL** за рахунок зайвих перевірок, коли **Y<>NIL**. Третя версія уникає повторних перевірок по **Y**, але вимагає перебудовувати **X** навіть у випадку, коли **Y=NI.** Отже, це означення може бути зроблено ще оптимальніше, зокрема використовуючи додаткові підфункції.

Використання додаткових підфункцій

Часто для побудови оптимальної функції доцільно вводити додаткові функції [8]. Так, для оптимізації попереднього означення функції **з'єднати (X, Y)** можна було б використати проміжну функцію **з'єд (X, Y)**, яка з'єднує **X** і **Y** у припущенні, що **Y<>NIL**. Тоді приходимо до означення:

з'єднати (X, Y) = якщо EQ (Y, NIL) то X інакше з'єд (X, Y)
з'єднати (X, Y) = якщо рівно (X, NIL) то Y
інакше CONS (CAR (X), з'єд (CDR (X), Y))

Відзначимо, що тут об'єднані переваги другої і третьої версій функції **з'єднати** із попереднього розділу.

Це загальноприйнятий прийом у функціональному програмуванні, коли на шляху визначення головної програми визначаються нові функції в термінах старих. Отже, функціональна програма складається з множин підфункцій, які визначені одна через другу. Ця функція чи функціонал, що є метою вирахувань, є головною програмою, першопричиною інших, а всі інші функції слугують для неї підпрограмами.

Проблема вибору підфункцій під час розроблення головної функції є центральною проблемою структурування програми. Іноді стандартні підфункції виявляються самі по собі, але частіші випадки, коли вдалий підбір підфункцій спеціального призначення дозволяє спростити структуру множин функцій, загалом. Для того, щоб побудувати добре структуровану програму, можна дати одну пораду: намагатися постійно покращувати те, що вже зроблено. Власне, це є один із принципів досягнення оптимального результату не лише у функціональному програмуванні, але й у будь-якій іншій галузі знань.

Підхід до оптимізації з використанням параметрів нагромадження

Ідея методу з параметрами нагромадження полягає в тому, щоб визначити допоміжну функцію з додатковим параметром, який використовується для нагромадження бажаного результату [8].

Проілюструємо суть цього методу, програмуючи функцію **обернути(x),** яка відає елементи списку x, можливо й пустого . Для цього введемо додаткову функцію **обр (x,y)**, де **x** – список, який підлягає обертанню, а **y** – додатковий параметр, який нагромаджує обернений список. Дамо таке означення:

обр (x, y) ≡ якщо EQ (x, NIL) то y
інакше обр (CDR (x), CONS (CAR (x), y))

Через цю функцію можна визначити функцію **обернути (x)**:

обернути (x) = обр (x, NIL)

Легше зрозуміти як діє ця функція, ніж описати алгоритм її побудови. Коли викликана функція **обр (x,y)**, то список **y** нагромадив в собі всі розглянуті елементи списку, які підлягають обертанню. Отже, якщо **x** є **NIL**, то в **y** міститься весь обернутий список, а якщо **x** не є **NIL**, то ми можемо в **y** нагромадити **CAR (x)** і знову рекурсивно викликати **обр** для оброблення **CDR (x)**. Наведемо таблицю послідовних викликів функції **обр (x,y)**, якщо перший раз функція **обернути (x)** звертається до списку **(A B C D)**:

X	Y
(A B C D)	NIL
(B C D)	(A)
(C D)	(B A)
(D)	(C B A)
NIL	(D C B A)

По суті справи ми навмання записали означення цієї функції, а потім описали, як вона працює. Для того, щоб застосувати загальноприйняту методику до побудови функцій, необхідно обгрунтувати вигляд результату **обр (x, y)** при довільному **y**. Нехай результатом функції **обр (x,y)**, коли **x** і **y** є списками, можливо пустими, є список всіх елементів **x**, які взяті в зворотному порядку і які доповнені всіма елементами **y** в їх початковому порядку. Тобто формально ми можемо записати:

обр (x, y) = з'єднати (обернути (x), y)

Хоча це визначення і не зовсім підходить, оскільки невідомою є сама функція **обернути**. Тим не менше тепер можна записати алгоритм побудови функції:

випадок (1): **x = NIL обр (x, y) = y**

випадок (2): **x ≠ NIL**

нехай обр (cdr (x),z) = з'єднати (обернути CDR((x),z) тоді
обр (x, y) = з'єднати (обернути (x), y) =
= з'єднати (обернути (CDR (x)), CONS (CAR (x),y)) =
= обр (CDR (x), CONS (CAR (x),y))

Наведений алгоритм швидше є доведенням справедливості вищеозначеної функції, ніж способом її побудови, оскільки в другому випадку перетворення є доволі складні.

Проблему побудови ефективних функційних програм можна успішно вирішити вдалим підбором підфункцій з додатковим параметром. Насправді, якщо підрахувати кількість викликів конструктора в наведеній версії функції **обернути**, то їх буде рівно **n**, якщо довжина списку **x** є **n**.

Якщо порівняти цю кількість викликів конструктора з кількістю викликів **n*(n-1)/2** у разі визначення функції **обернути (x)** без параметрів нагромадження, то маємо значну економію машинних ресурсів.

Розширення строго функціональної мови

Розширимо нашу строго функційну мову, даючи змогу функціям видавати складові результати. Будемо вважати правильними виразами списки правильних виразів, які поміщені в кутові дужки. Так, вираз < e1, e2, ..., ek > є правильним, якщо правильними є вирази e1, e2, ..., ek. Друге розширення те, що у лівій частині локальних форм дозволимо використовувати не лише прості змінні, але й список ідентифікаторів, які поміщені в квадратні дужки. Спосіб використання цього комбінованого розширення демонструє приклад

{ НЕХАЙ f=λ(x,y) <x зал y, x діл y>
{ НЕХАЙ <r, d>=f(u, v)}}

Тобто тут список значень **f** точно буде скопійований в список змінних і внутрішній блок зв'яже з **r** значення **u зал v**, **u** з **d** – значення **u діл v**. Це суто синтаксичне розширення, оскільки цього результату ми могли б досягнути, використовуючи операцію **cons**:

{ НЕХАЙ f=λ(x,y) CONS(x зал y, CONS(x ціл y), NIL
{ НЕХАЙ S=f(u, v)
{ НЕХАЙ r=CAR(S) i d=CAR(CDR(S))}}}

Для демонстрації того, наскільки зросла ясність подання виразів у разі такого розширення, розглянемо функціональне визначення порозрядного складання з переносом одиниці. Нехай маємо функцію, яка підсумовує три числа по модулю m:

сум1(a, в, c)=<(a+b+c) діл m, <(a+в+c) зал m>

Тут **a, в** – цифри, які підсумовують і не перевищують **m, c** – одиниця переносу. Результат функції **сум1** – це пара значень, де на першому місці цифри переносу з цього розряду і цифри суми. Тепер побудуємо функцію **сум N(a, в)**, яка як аргумент видає результатом список цифр суми і кінцеву цифру переносу. Визначення є очевидним і має вигляд:

сумN(x,y)≡якщо CDR(x)=NIL то CDR(y))
інакше сум1(CAR(x), CAR(y)), 0)
інакше {нехай <c_i, S>=сумN(CDR(x), CDR(y))
{нехай <c_0, S>=сум1(CAR(x), CAR(y), c_i)
<c_0, CONS(d, S)>}}

Якщо в списках **x** і **y** лише по одній цифрі, то можна використати **сум1** з цифрою переносу, що дорівнює 0. Інакше використовується **сумN(x)**, щоб скинути всі цифри, крім старших, і отримати цифри переносу $\mathbf{c_i}$ і суму **S**. Потім використовується **сум1,** щоб одержати суму цифр найстаршого розряду і цифру переносу в старший розряд $\mathbf{c_i}$ і отримати цифру переносу зі старшого розряду $\mathbf{c_0}$. Старша цифра суми є **d**, тому весь список складання цифр **x** і **y** отримується шляхом використання операції **cons** до **d** і **S**.

Безсумнівно, можна написати цю програму, не використовуючи запропонованого розширення мови. Другою альтернативою може бути реалізація двох функцій, одна з яких вираховує переноси, а друга – суми від списків задання цифр по модулю m. Хоч ясність такого подання можливо не зменшується, але ефективність значно погіршиться, оскільки потрібно буде багатократно повторити вирахування одних і тих самих переносів у молодших розрядах для кожної з підфункцій.

Розглянемо третій варіант, коли не потрібно розширювати мову, який є доволі конкурентоспроможний в розумінні ясності подання.

Кожну з функцій **сум1** і **сумN** розширюємо введенням її як параметра функції **f**, яку назвемо продовженням. Так, виклик **сумN(x, y, f)** означає складання списків **x** і **y** і використання функції **f** до цього результату.

Природно, що функція продовження **f** вимагає двох параметрів, відповідно значення цифри переносу і списку цифр суми. Нове визначення матиме вигляд:

сум1(а, в, с, f)≡f((а+в+с) діл m, (а+в+с) зал m)

сума(x, y, f)≡якщо CDR(x)=NIL то сум1(CAR(x)), CAR(y), 0, f)

інакше сумN(CDR(x), CDR(y),

{λ(c_i, S) сум1(CAR(x), CAR(y), c_i, {λ(c_0, d) f(c_0, CONS(d, S))})})

Відзначимо, що фактичне продовження, яке замінює вкладені виклики **сум1** і **сумN,** записане як **λ**-вирази з формальними параметрами, які мають ті самі імена (і виконують ту саму функцію), що відповідні локальні визначення в попередній програмі з кутовими дужками.

Шляхи вдосконалення функціональних мов

Принциповою характеристикою сучасних комп'ютерів є те, що вони зберігають вирахувані значення в комірках пам'яті, час від часу замінюючи їх вміст або перезаписуючи їх. Ця властивість відображена в конструкції присвоєння, яка характерна для традиційних мов програмування. Є ще один аспект реальних машин, завдяки якому традиційні мови ефективніші порівняно з функціональними мовами програмування. Це поняття доступу до структур даних за допомогою індексування. У зв'язку з цим функціональні програми не можуть досягнути такої самої ефективності на сучасних комп'ютерах, як програми алгоритмічними мовами програмування. Можна вказати, принаймні, три виходи з такої ситуації:

1) у разі неготовності до деякої втрати ефективності (під час виграшу в однозначному розумінні та елегантності програми) відмовитися від функціональних програм;

2) розробляти методи організації функціональних програм так, щоб їх виразність і ясність поєднувалися з ефективністю, зіставимою з програмами, написаними традиційними мовами програмування;

3) перебудувати структуру сучасних комп'ютерів таким способом, щоб вони відповідали цілям інтерпретації функційних програм.

З певного погляду кожен з цих підходів має право на своє існування. Операція присвоєння є тісно пов'язана зі зв'язуванням значення зі змінною у функціональній мові. Можна показати, що будь-яка програма, де є операція присвоєння змінній певного значення, може бути трансформована у функціональну програму. Тоді така функціональна програма з певними обмеженнями на її структуру може бути скомпільована в програму з присвоєнням. Отже, можна стверджувати, що функційні мови не виключають можливості ефективного використання сучасних комп'ютерів. Але це лише частина проблеми, оскільки операція присвоєння значення елемента масиву виду

a[i]:=1

a[i+1]:=a[i]+1

істотно відрізняється від операції присвоєння значення простій змінній. У строго функціональній мові відсутнє поняття вирахування послідовністю дій і з масивами поводяться як з цілими масивами значень. Основними операціями над масивами є операція індексування для отримання значення компоненти і конструювання нового значення масиву із старого. Якщо **a** – значення масиву, **i** – його індекс, **x** – значення, то запис

обновити (a, i, x)

означає збереження значення масиву **a** зі змінною його **i**-ї компоненти на значення **x**. А вираз

обновити (a, i, a[j]), j, a[i])

означає заміну місцями значень **i**-го і **j**-го елементів масиву **a**.

Тому проблема корекції інтерпретації правильності функції **обновити** є доволі складною під час врахування того, що одне і те саме значення масиву може мати різні імена. Операція оновлення значень масиву вимагає багатократного копіювання цих значень. Виникає запитання: чи можна вибрати таку структуру даних, щоб цілком або частково уникнути цих копіювань? Проблеми не виникає, якщо для цих цілей використовувати операцію **cons**, оскільки структури, які є компонентами **cons** не копіюються, а зберігаються лише вказівники на них в структурі, яка є результатом **cons**. Але не хотілося б так зображати реалізацію значень масивів, оскільки

втрачається основна перевага сучасних комп'ютерів – можливість індексування. Насправді у разі послідовного доступу в лінійних списках час доступу до **i**-го елемента пропорційний кількості елементів **i**. Водночас у разі розташування масиву в комірках пам'яті одночасно доступний будь-який елемент незалежно від позиції. Можна масиви подати у вигляді бінарного дерева; час доступу до елемента пропорційний $\mathbf{ln_2 n}$. Тим не менше це є непорівнянним з одиницею часу доступу, яка необхідна у разі елементарного оновлення в операторі **a[i]:=x**.

Функціональні мови за ефективністю на сучасних комп'ютерах не можуть конкурувати з традиційними мовами, якщо ефективність є абсолютною вимогою. Однак програми, написані з елементним присвоєнням структурованих значень, є програмами низького рівня. Вимагають значних зусиль для їх побудови, а тому є важкими для правильної інтерпретації.

Висновки

Списокова форма подання даних і програм, яка характерна для функціональних мов програмування, дає змогу успішно оперувати зі значною кількістю арифметичних, логічних та синтаксичних умов і додаткових обмежень під час застосування кожного із наведених підходів.

Відсутність поняття змінної, операторів присвоєння, циклів, умовних та безумовних переходів в явному вигляді, які характерні для алгоритмічних мов програмування, дає можливість елегантного і легкозрозумілого запису функціональної програми та її доступного тестування.

Застосування розглянутих методів та підходів до побудови програм дає змогу оптимізувати складно структуровану функціональну програму по швидкодії та формі запису, уникнути зациклювань і повторюваності логічних вислідів, що актуально під час роботи з великими обсягами даних, опрацювання символьної інформації та створення інтерпретаторів інших мов програмування.

1. Mccarthy J. Recursive functions of symbolic expressions and their computation by machine // Comm. ACM.: – 1960.– Vol.3. – P.184–195. 2. Крюков А.П., Родионов А.Я., Таранов А.Ю., Шаблыгин Е.М. Программирование на языке R-Лисп. – М.: Радио и связь, 1991. – 192 с. 3. Еднерал В.Ф., Крюков А.П., Родионов А.Я. Язык аналитических вычислений Reduce. – М.: Изд-во МГУ, 1984. – 176 с. 4. Бадаєв Ю.І. Теорія функціонального програмування. Мови Common Lisp та Auto Lisp. – К., 1999. – 150 с. 5. Заяць В.М. Конспект лекцій з курсу "Функційне програмування". – Львів, 1999. – 55 с. 6. Маурер У. Введение в программирование на языке Лисп. – М.: Мир, 1976. – 104 с. 7. Хювенен Э., Слеппян Й. Мир Лиспа. В 2-х т. Т.1. Введение в язык Лисп и функциональное программирование. Пер. с финск. – М.: Мир, 1990. – 447 с. 8. Заяць В.М., Заєць М.М. Логічне та функційне програмування: Навч. посібник. – Львів: Бескид Біт, 2006. – 352 с.