УДК 628.321
© В. Заяць[1], 2012

ПІДХОДИ ДО ПОБУДОВИ ТА ОПТИМІЗАЦІЇ ФУНКЦІОНАЛЬНИХ ПРОГРАМ

Розглянуто основні підходи до побудови функціональних програм. Визначено основні способи оптимізації новостворених функцій та функціоналів. Кожен із способів ілюструється прикладами реалізації у середовищі Ліспу.

The basic approaches for constructions of the functional programs are considered in the article. The main methods of optimization of the new functions and of the functional are determined. Each of methods is illustrated by the examples of realization in the environment of Lisp.

1. ВСТУП

У роботі розглядаються підходи до побудови та оптимізації новостворених функціональних програм з метою підвищення ефективності та надійності їх роботи.

Доцільність використання описаних підходів ілюструється конкретними прикладними розробками, які реалізовано в середовищі стандартного Ліспу [1], який відноситься до 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 слід враховувати такі міркування:

1. **Ясність, простота і узгодженість понять мови**. Очевидно, слід 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 при розробці програми дотримання принципів структурного програмування, ієрархічного конструювання програм – зверху вниз. При такому підході структура програми легкозрозуміла, доступна для діагностики, модифікації та оптимізації.

[1] Національний університет “ Львівська політехніка”

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 на цій мові, можна розглядати як розширення мови. В 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 найбільш 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.

2. ПЕРЕВАГИ 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, слід виходити не з кількісного співвідношення того, що вони дозволяють зробити, а з якісних відмінностей, що визначають елегантність (короткість і наочність), легкість (прозорість) та ефективність (швидкодія та технічні засоби) програмування на них, здатність до оптимізації. Це порівняння слід 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. Ряд функціональних програм можуть ефективно працювати на сучасних комп'ютерах, тим не менше 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 представлення 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, великою мірою, зі способом вибору структур даних. Дещо інший їх вибір можна забезпечити підвищенням ефективності функціональних 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 на цій мові – простота семантики мови. Велика перевага функціональної мови 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 відлагоджених функціональних програм.

3. СТРОГО ФУНКЦІПНАЛЬНА МОВА

Коли мова йде про побудову строго функціональної мови, то нові функції чи функціонали будуються на основі деякого базового набору функцій або примітивних функцій [5]. До них відносяться 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. При цьому 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 більш оптимальним чином:

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 лишніх перевірок, коли **Y<>NIL**. Третя версія уникає повторних перевірок по **Y**, але вимагає перебудовувати **X** навіть у випадку, коли **Y=NI.** Таким чином, дане означення може бути зроблено ще оптимальніше, зокрема використовуючи додаткові підфункції.

4. 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. Таким чином, 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 служать для неї підпрограмами.

Проблема вибору підфункцій при розробки головної функції є центральною проблемою структурування програми. Іноді стандартні підфункції виявляються самі по собі, але більш часті випадки, коли вдалий підбір підфункцій спеціального призначення дозволяє спростити структуру множин функцій в цілому. 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.

5. 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 накопичує обернений список. Дамо наступне 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)

Пояснимо як працює ця функція, а потім опишемо 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 обернений список, а якщо **x** не є **NIL**, то ми можемо в **y** нагромадити **CAR (x)** і знову рекурсивно викликати **обр** для обробки **CDR (x)**. Приведемо 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 навгад записали означення цієї функції, а потім описали, як вона працює. Для того, щоб застосувати загальноприйняту методику до побудови функцій, необхідно обґрунтувати 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 вище означеної функції, ніж способом її побудови, оскільки в другому випадку перетворення є доволі складні.

Проблема побудови ефективних функціональних програм може бути успішно розв'язана шляхом вдалого підбору підфункцій з додатковим параметром. Дійсно, якщо підрахувати кількість викликів конструктора в приведеній версії функції **обернути**, то їх буде рівно **n**,

якщо довжина списку **x** є **n**. Якщо порівняти це число викликів конструктора з числом викликів **n×(n–1)/2** у випадку визначення функції **обернути (x)** без параметрів нагромадження, то маємо значну економію машинних ресурсів.

5. НАПРЯМИ ОПТИМІЗАЦІЇ ФУНКЦІОНАЛЬНПХ 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. В 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:

– при неготовності змиритися з деякою втратою ефективності (при виграші в якості і елегантності програми) відмовитися від функціональних програм;

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

– перебудувати структуру сучасних комп'ютерів так, щоб вони відповідали цілям інтерпретації функціональних програм.

З певного погляду кожен з цих підходів має право на 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 функціональні 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** – вої 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 при врахуванні 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 представляти реалізацію значень масивів, оскільки втрачається основна перевага сучасних комп'ютерів – можливість індексування. Дійсно, при послідовному доступі в лінійних списках час доступу до **i**–го елемента пропорційний до кількості елементів **i**. Водночас при розташуванні 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 при елементарному оновленні 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.

Підвищення швидкості комп'ютерів та місткості їх пам'яті на сучасному етапі робить доступними ряд важливих функціональних програм прикладного характеру. Але якісного стрибка у використанні функціональних мов можна очікувати лише з появою комп'ютерів нової архітектури, яка орієнтована не на використання технічних засобів, а на особливості самої мови, яка будується, виходячи з природних потреб користувача.

6. ВИСНОВКИ

Застосування розглянутих способів та підходів до побудови програм дозволяє оптимізувати складно структуровану функціональну

програму по швидкодії, технічних засобах та формі запису, уникнути повторюваності логічних вислідів, що актуально при роботі з великими обсягами даних та базами даних, обробці символьної інформації та створенні 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.–192с. 3. Еднерал В.Ф., Крюков А.П., Родионов А.Я. Язык аналитических вычислений Reduce. – М.: Изд–во МГУ, 1984. – 176с. 4. Бадаєв Ю.І. Теорія функціонального програмування. Мови Common Lisp та Auto Lisp. – Київ, 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., Заяць М.М. Логічне та функційне програмування: Навч. посібник. – Львів.: Видавництво Бескид Біт, 2006.– 352 с.