УДК 004.023
Марченко О.І. к.т.н, доцент, Хоптинець В.А. магістрант
Національний технічний університет України «Київський політехнічний інститут»

ТРАНСЛЯЦІЯ ПРОГРАМ З ПРОЦЕДУРНИХ МОВ ПРОГРАМУВАННЯ У ФУНКЦІОНАЛЬНІ МОВИ З ВИКОРИСТАННЯМ ГРАФУ ЗАЛЕЖНОСТІ ДАНИХ

Марченко О.І., Хоптинець В.А. Трансляція програм з процедурних мов програмування у функціональні мови з використанням графу залежності даних. У статті пропонується спосіб трансляції програм з процедурних мов програмування у функціональні мови з використанням внутрішньої форми, що відповідає функціональній парадигмі програмування – графу залежності даних. На відміну від інших внутрішніх форм, таких як граф потоку виконання, граф залежностей даних, цей граф явно виражає залежність по даним між операторами вхідної мови, що дає змогу виділили обчислення без сторонніх ефектів.
Ключові слова: трансляція, процедурне програмування, функціональне програмування, внутрішня форма, граф залежності даних.

Марченко А.И., Хоптынец В.А. Трансляция программ с процедурных языков программирования в функциональные языки с использованием графа зависимостей данных. В статье предлагается способ трансляции программ с процедурных языков программирования в функциональные языки с использованием внутренней формы, которая соответствует функциональной парадигме программирования – графа зависимостей данных. В отличии от других внутренних форм, таких как граф потока выполнения, граф зависимостей данных явно выражает зависимости по данным между операторами исходного языка, что дает возможность отделить вычисления без побочных эффектов.
Ключевые слова: трансляция, процедурное программирование, функциональное программирование, внутренняя форма, граф зависимостей данных.

Marchenko O.I., Khoptynec V.A. Translation of programs from procedural languages to functional languages using value dependence graph. Technique for programs translation from procedural programming languages to functional languages using functional-oriented intermediate representation – value dependence graph, is proposed in this paper. In contrast to other intermediate representations like control flow graph, value dependence graph exposes dependencies among statements of source languages, that helps to extract evaluations without side effects.
Keywords: translation, procedural programming, functional programming, intermediate representation, value dependence graph.

Вступ. Починаючи з середини XX сторіччя було створено велику кількість програмного забезпечення (ПЗ), яке досі, незважаючи на еволюцію апаратного забезпечення, актуальне і потребує підтримки, оскільки на ньому тримаються критичні системи тих чи інших підприємств. Підтримка таких систем тягне за собою певні додаткові витрати (порівняно із сучасним ПЗ), що пов'язані з використанням застарілих програмних інструментів. Тому, актуальною задачею є автоматична трансляція ПЗ на рівні вихідного коду з одних мов високого рівня у інші. В даному випадку, з мов, що практично вийшли з використання, у більш сучасні мови.

Зараз багато спеціалістів відзначають перспективність декларативної функціональної парадигми для створення нових великих програмних систем. Функціональна парадигма надає ряд переваг з точки зору розпаралелення програм та автоматичної верифікації.

Оскільки застарілі мови програмування високого рівня, як правило, є процедурними, то дослідження способів ефективної трансляції програм з процедурних мов програмування у функціональні мови є перспективним напрямом. Саме дослідженню таких способів трансляції присвячена дана робота.

Постановка наукової проблеми. Метою даної роботи є аналіз існуючих способів трансляції програм з процедурних мов програмування у функціональні з метою отримання способу, що забезпечує автоматичне відокремлення обчислень, що не містять сторонніх ефектів, від обчислень в процедурній парадигмі.

Аналіз існуючих способів.

Процедурне програмування – парадигма програмування, при якій порядок обробки операторів комп'ютером є строго визначеним.

Функціональне програмування – парадигма програмування, в якій процес програмування трактується як обчислення значень функцій в математичному сенсі, і порядок обчислення функцій визначається наявністю даних для їх обчислення, а не порядком їх розташування у тексті програми.

При використанні обох парадигм програмування, як процедурної, так і функціональної, складовими частинами програми є набір операторів або виразів у вигляді підпрограм з певними вхідними параметрами. У випадку процедурного програмування такі підпрограми є або процедурами, або функціями, а у випадку функціонального програмування – функціями. Різниця між функціями процедурного і функціонального стилю полягає у семантиці. У першому випадку функція є підпрограмою, яка може приймати якісь параметри та повертати значення. У функціональному програмуванні, функція має таке ж значення, як і в математиці. Це призводить до того, що функції у функціональних мовах програмування мають бути «чистими», тобто не містити сторонніх ефектів.

Найпростіший спосіб – пряма трансляція. У цьому випадку, вхідна програма транслюється літерально, моделюючи нехарактерні для вихідної мови абстракції вхідної мови.

Сучасні мови програмування, що підтримують як процедурну, так і функціональну парадигми, наприклад Common Lisp, дозволяють використовувати сторонні ефекти, зокрема оператор присвоювання замінюється на виклик макросу setf. Також, в Common Lisp можна використовувати динамічні змінні, які легко замінюють глобальні змінні. Таким чином, трансляція в Common Lisp є достатньо прямою. Приклад програми мовою C показано на рис.1. Відповідна програма мовою Common Lisp – на рис.2.

```
int global_foo;

int foo(int a) {
  int x = 123 + a;
  if (x > global_foo) {
    global_foo = x;
  }
  else {
    global_foo = a;
  }
  return global_foo + a + x;
}
```

Рис.1. Програма мовою C

```
(defvar *global_foo*)

(defun foo (a)
  (let ((x (+ 123 a)))
    (if (> x *global_foo*)
      (setf *global_foo* x)
      (setf *global_foo* a))
   (+ *global_foo* a x))
```

Рис.2. Програма мовою Common Lisp

Натомість в чисто функціональних мовах програмування, таких як Haskell, не можна просто використовувати оператор присвоювання і глобальні змінні. Для емуляції такої семантики у Haskell доводиться використовувати спеціальний тип IORef у зв'язці із монадою вводу-виводу. Попередній приклад мовою Haskell наведено на рис.3.

Отже, можна зробити висновок, що проста пряма трансляція має сенс лише у мовах, які крім основної функціональної парадигми, мають також засоби підтримки процедурної парадигми, бо в іншому випадку результуючий код стає занадто ускладненим. Проблема з цим підходом полягає у тому, що порушуються абстракції, на які спирається функціональна парадигма. Це призводить до ускладнення супроводу таких програм, а також перешкоджає компілятору у виконанні додаткових, більш ефективних, способів оптимізації коду.

```
global_foo :: IORef Int
global_foo = unsafePerformIO $ newIORef
undefined

foo :: Int -> IO Int
foo a = do
  let x = 123 + a
  _global_foo <- readIORef global_foo
  if x > _global_foo then
    writeIORef global_foo x
  else
    writeIOref global_foo a
  _global_foo_return <- readIORef
global_foo
```

Рис. 3. Програма мовою Haskell

Іншим підходом до трансляції процедурних мов у функціональні є використання мово-незалежних внутрішніх (проміжних) форм. Внутрішні форми відрізняються між собою інформацією про вхідну програму, яку вони використовують. Найпростіша внутрішня форма – дерево розбору, яке представляє структуру програми. Для виконання аналізу, оптимізації та трансформації вхідної програми у вихідну програму доцільно попередньо перевести дерево розбору у більш придатну для цього внутрішню форму.

Поширеною внутрішньою формою є граф потоку виконання (CFG) у зв'язці з формою одиничного присвоювання (SSA), на базі якого можна виконувати аналізи потоку даних (DFA). До недоліків цієї форми відноситься сильна впорядкованість операторів, а також відсутність інформації про залежності між операторами. Приклад графу потоку виконання для попередньої програми наведено на рис.4.

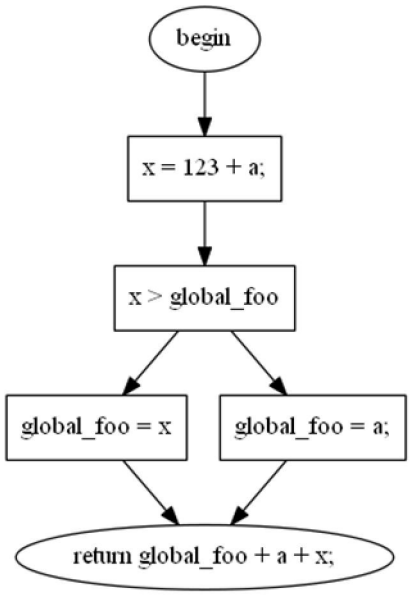

Рис. 4. Граф потоку виконання

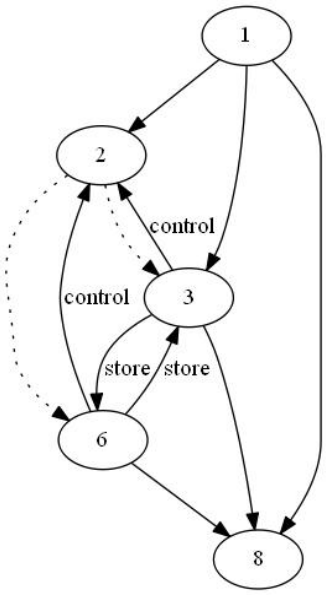

Рис. 5. Граф залежностей

Альтернативою CFG є граф залежностей (PDG) [1]. В цьому графі вершинами також є оператори, а дуги - залежності між ними. Така форма дозволяє «послабити» впорядкованість, роблячи її неявною, що дозволяє виконувати деякі специфічні трансформації, а також векторизуючі оптимізації. Проблема PDG полягає в тому, що структурною одиницею є оператор,

зокрема, оператор присвоювання. Тобто, така форма добре підходить для подальшої генерації асемблерного коду або коду на процедурній мові високого рівня. Граф залежностей для наведеного вище прикладу зображено на рис.5. Кожен вузол відповідає оператору вхідної програми (номер вузла відповідає номеру рядка у вхідній програмі). 1 – визначення змінної x; 2 – порівняння x та global_foo; 3 – запис x у global_foo; 6 – запис a у global_foo.

Виклад основного матеріалу й обґрунтування отриманих результатів дослідження.

Пропонується спосіб трансляції програм із процедурних мов програмування у функціональні мови, що базується на використанні графу залежності даних (VDG) як внутрішньої форми представлення програми [2]. Граф залежності даних дозволяє позбавитись від інформації про оператор, залишаючи натомість інформацію про залежність даних у цих операторах. У цьому графі вершинами є дані і операції над ними, а як дуги використовуються залежності. Таким чином, граф залежності даних добре відповідає функціональній парадигмі і тому придатна для ефективної генерації коду функціональної мови програмування. При такому підході компілятор зможе автоматично виявити чисто функціональні фрагменти коду, а для інших фрагментів використати монаду стану [3]. До недоліків цієї форми відноситься складність побудови графу залежності даних за програмою на вхідній процедурній мові. Граф залежності даних для наведеного вище прикладу зображено на рис. 6.

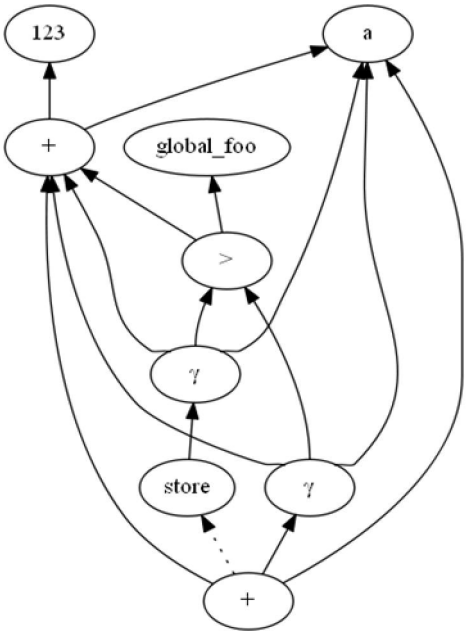

Рис. 6. Граф залежності даних

У цьому графі вершини відображають обчислення, а дуги – значення, від яких залежить обчислення. Якщо операція E1 залежить від результату обчислення E2, то граф містить дугу від E1 до E2. Вершини можуть бути кількох типів:

1. Примітивна операція. До цього типу відносяться арифметичні операції, константи тощо.

2. Булевий селектор (γ), що відповідає умовним виразам. Ці вершини видають одне з вхідних значень в залежності від умовного виразу.

3. Замикання (λ), що видає значення-функцію.

4. Виклик функції. Ці вершини залежать від значення-функції та параметрів (що відповідають формальним параметрам вхідної функції). Результатом цих вершин є значення із вершини результату вхідної функції.

5. Формальний параметр. Ці вершини не мають залежностей, але мають значення, що було передано у дану функцію через даний формальний параметр.

6. Вільна змінна. Такі вершини містять значення зі зовнішнього контексту.

7. Вершина результату. Від цих вершин інші вершини не залежать явно. Самі вершини залежать від значення, яке є значенням всієї функції. Будь-який граф залежності даних містить не менше однієї вершини результату (по одній вершині для кожного значення, що повертається).

8. Читання із глобального стану.

9. Запис у глобальний стан.

Побудова графу залежності даних із дерева розбору складається зі п'яти основних етапів. По-перше, необхідно розбити дерево на базові блоки і побудувати граф потоку виконання.

На другому етапі необхідно побудувати граф залежності записів (store dependency graph – SDG). Цей граф замість інформації про залежності у виконанні відображає потік зміни значення запису. Запис містить інформацію про значення усіх змінних. При обчисленні нового значення певної змінної використовуються поточні значенні запису. Вузлам розгалуження у графі потоку виконання відповідає розгалуження запису з подальшим вибором за допомогою γ-вузла. Тіла циклів замінюються функція, а самі цикли на виклики цих функцій.

Наступним кроком необхідно виконати вбудовування (inlining) нерекурсивних викликів. Після чого можна виконувати символьну інтерпретацію графу залежності записів. При цьому створюються вершини, що відповідають операціям, що виконуються над значеннями. Під час інтерпретації використовуються таблиці пошуку, в яких міститься інформація з якої вершини можна отримати поточне значення певної змінної. Якщо в таблиці відсутній вказівник на відповідну вершину (наприклад, значення було неявно змінено у функції), то використовуються спеціальні вершини читання із глобального стану.

На цьому етапі граф залежності даних вважається створеним, але для подальшої генерації варто виконати аналіз циклів та видалення зайвих операцій запису і читання.

Варто зазначити, що в результаті побудови графу залежності даних, над вхідною програмою автоматично виконуються нумерація значень і розповсюдження копій та констант.

Висновки

Проаналізовані способи трансляції програм з процедурних мов програмування у функціональні можуть бути закладені в основу відповідних спеціалізованих трансляторів. Як спосіб прямої трансляції, так і спосіб, що базуються на використанні внутрішніх форм, мають свої переваги і недоліки.

Спосіб прямої трансляції є простішим у реалізації, але має обмежене застосування. Його можна використати лише тоді, коли вихідна мова підтримує імперативну парадигму, або коли вхідна програма не містить сторонніх ефектів, тобто семантично є функціонально чистою, що майже не зустрічається на практиці.

Спосіб, що базується на використанні графу залежності даних є більш складним у реалізації, але він надає змогу автоматично виділити функціонально чисті фрагменти коду від наявних імперативних фрагментів, відокремивши їх за допомогою монади стану. Крім того, у порівнянні з іншими внутрішніми формами, граф залежності даних дозволяє ефективніше виконувати деякі подальші оптимізації, таких як slicing [4].

Напрямом подальшого дослідження є розробка комбінованих способів трансляції, що поєднують описані вище способи трансляції.

1. Jeanne Ferrante, Karl J. Ottenstein, Joe D. Warren "The Program Dependence Graph and Its Use in Optimization" - ACM Transactions on Programming Languages and Systems, том 9, №3, Липень 1987, с. 319-349.
2. Daniel Weise, Roger F. Crew, Michael Ernst, Bjarne Steensgaard "Value Dependence Graphs: Representation Without Taxation", Principles Of Programming Languages 1994.
3. Philip Wadler "Monads for functional programming" – University of Glasgow G12 8QQ
4. Mark Weiser. "Program slicing". Proceedings of the 5th International Conference on Software Engineering, pages 439–449, IEEE Computer Society Press, March 1981.