УДК 004.43
Погорелов В.В., Марченко О.І.
Національний технічний університет України "Київський політехнічний інститут"

ОГЛЯД ВНУТРІШНІХ ФОРМ ПРЕДСТАВЛЕННЯ ПРОГРАМИ ДЛЯ ТРАНСЛЯЦІЇ З ПРОЦЕДУРНИХ МОВ ПРОГРАМУВАННЯ У ФУНКЦІОНАЛЬНІ МОВИ

В.В. Погорелов, О.І. Марченко. Огляд внутрішніх форм представлення програми для трансляції з процедурних мов програмування у функціональні мови. У статті, в контексті вирішення задачі трансляції з процедурних мов програмування у функціональні мови, розглянуто ряд внутрішніх форм представлення програми (ВФПП) у вигляді орієнтованих графів: граф потоку управління, форма статичного однократного присвоєння, розширена форма статичного однократного присвоєння та граф залежностей значень і станів. Запропоновано набір критеріїв для вибору ВФПП та проаналізовано відповідність вищезазначених ВФПП набору цих критеріїв.
Ключові слова. Граф потоку управління, форма статичного однократного присвоєння, розширена форма статичного однократного присвоєння, граф залежностей значень, граф залежностей значень і станів, внутрішнє представлення, трансляція програм.**Форм. 0, Таб. 1, Рис. 4, Літ. 21.**

В.В. Погорелов, А.И. Марченко. Обзор внутренних форм представления программы для трансляции из процедурных языков программирования в функциональные языки. В статье, в контексте решения задачи трансляции из процедурных языков программирования в функциональные языки, рассмотрен ряд внутренних форм представления программы (ВФПП) в виде ориентированных графов: граф потока управления, форма статического однократного присвоения, расширенная форма статического однократного присвоения и граф зависимостей значений и состояний. Предложен набор критериев для выбора ВФПП и проанализировано соответствие вышеупомянутых ВФПП набору этих критериев.
Ключевые слова. Граф потока управления, форма статического однократного присвоения, расширенная форма статического однократного присвоения, граф зависимостей значений, граф зависимостей значений и состояний, внутреннее представление, трансляция программ.

V.V Pogorelov, O.I. Marchenko. Survey of intermediate forms of program representation for translation from procedural programming languages into functional languages. The paper deals with the analysis of a range of intermediate representation forms (IRF) as oriented graphs in the context of translation of a program from procedural programming languages into the functional programming languages. In particular, the following graphs are examined: control flow graph, single static assignment, gated single assignment, value state dependence graph. A set of criteria for IRF selection is proposed and correspondence of the aforementioned IRFs to the set of criteria is analyzed.
Keywords. Control flow graph, single static assignment, gated single assignment, value dependence graph, value state dependence graph, internal representation form, translation of programs.

Вступ і постановка проблеми. На сьогоднішній день, в області задач сучасного системного програмування, знаходять застосування мови, що базуються на різних стилях програмування, таких, як об'єктно-орієнтований, аспектно-орієнтований, процедурний, логічний та функціональний. Серед перерахованих категорій мов програмування виділимо процедурні та функціональні мови. Процедурні мови виділяються серед інших за критерієм кількості програмного коду, вже написаного цими мовами, а функціональні мови – за критерієм перспективності їх використання у майбутньому.

Процедурний стиль програмування розглядає програму як опис процедури виконання дій (послідовність дій та умови їх виконання) для досягнення результату, де однією із базових структур є змінна, яка зберігає певне значення і дозволяє змінювати його протягом виконання алгоритму. На відміну від процедурних мов, процес виконання програми, що написана функціональною мовою програмування, ґрунтується на обчисленні результатів функцій від вихідних даних та від результатів інших функцій і не передбачає явного збереження стану програми. Відповідно, не передбачається й зміна цього стану. Під поняттям стану програми будемо розуміти множину значень усіх змінних, що доступні у конкретній точці програми у довільний момент її виконання.

Функціональний стиль передбачає, що процес програмування інтерпретується як обчислення значень функцій у математичному сенсі, причому порядок їх обчислення визначається наявністю даних для обчислення, а не порядком розташування функцій та їх викликів у тексті програми.

Зважаючи на той факт, що між процедурними та функціональними мовами програмування існує значний семантичний розрив, можна сказати, що проблема трансляції програм з процедурних мов у функціональні є нетривіальною. Коректність та ефективність трансляції великою мірою забезпечується внутрішньою формою представлення програми (ВФПП) у трансляторі. Тому вибір ВФПП, яка дозволить повною мірою відобразити взаємозв'язки вхідної програми на процедурній мові та ефективно

згенерувати суттєво відмінний код вихідної програми на функціональній мові, є актуальною науковою задачею.

Внутрішні форми представлення програм у контексті рішення задач із різних сфер проектування трансляторів в останні роки вивчали, наприклад, такі дослідники, як М. В. Зубов, А. Н. Пустигін, Э. В. Старцев [1, 2], А. А. Рибаков [3], В. А. Битнер, Н. В. Заборовский [4] та ін.

Проблемою побудови оптимізуючих компіляторів, використовуючи різні ВФПП, зокрема такі, як граф потоку управління (ГПУ), форма статичного однократного присвоєння (ФСОП), розширена форма статичного однократного присвоєння (РФСОП) та ін., займався А. Ю. Дроздов. Результати його роботи викладені на сторінках докторської дисертації науковця [5]. Вчений описує процес розробки алгоритмічних основ функціонування компонент оптимізуючих компіляторів, націлених на досягнення граничних рівнів продуктивності для платформ з явним паралелізмом, як на рівні команд, так і на рівні ядер процесора.

Серед зарубіжних вчених особливої уваги заслуговують Даниель Вейс [6], що запровадив поняття графу залежностей значень (ГЗЗ), Джеймс Стен'єр і Дес Ватсон, які запропонували розширений ГЗЗ – граф залежностей значень і станів (ГЗЗС) [7], а також Хельг Бахманн, який запропонував ефективний алгоритм трансформації ГЗЗС у ГПУ [8].

Після проведення детального аналізу літературних джерел не було виявлено комплексного огляду ВФПП, які є ядром ефективного оптимізуючого транслятора, у контексті рішення задачі трансляції з процедурних мов програмування у функціональні.

Метою дослідження є аналіз ефективності використання таких ВФПП, як ГПУ, ФСОП, ГЗП, ГЗЗ та ГЗЗС, у рамках рішення задачі трансляції з процедурних мов програмування у функціональні мови, та визначення критеріїв вибору доцільної ВФПП для цієї задачі.

Виклад основного матеріалу дослідження. У даній роботі важливу роль відіграють такі терміни як процедурна мова програмування (ПМП), функціональна мова програмування (ФМП) та чиста функціональна мова програмування (ЧФМП). У статтях різних авторів вищезгадані терміни часто інтерпретуються не однаково. У контексті даної статі терміни «процедурна (імперативна) мова програмування» та «функціональна мова програмування» будемо розуміти так, як визначено в [9, 10]. У свою чергу, ФМП можна поділити на два взаємовиключні підкласи: чисті (pure) та гібридні (impure) мови.

У літературних джерелах часто посилаються на неформальні визначення ЧФМП. Ці визначення базуються на поняттях прозорості посилань та незалежності порядку обчислення [11]. У даній статті, використовуючи термін ЧФМП, будемо використовувати формальне визначення, що було запропоноване вченим А. Сабри у [12].

Властивість чистоти ФМП тісно пов'язана з відсутністю побічних ефектів (ПЕ). У рамках вирішення задачі трансляції програми з ПМП у ФМП можна виокремити наступні типи ПЕ:

1. Присвоєння локальній змінній нового значення.
2. Читання глобальної змінної.
3. Модифікація глобальної змінної.
4. Виконання операторів вводу/виводу.
5. Виконання процедур/функцій, у яких параметр(и) передаються за адресою.

ПЕ із наведеного вище списку будемо позначати $ПЕ_i$ (i=1..5).

А. Сабри показав, що відсутність побічних ефектів виступає необхідною умовою чистоти ФМП [12]. Враховуючи дане твердження, можна запропонувати зазначені п'ять типів ПЕ як критерії якості ВФПП, що відображають властивість деякої ВФПП не містити відповідних $ПЕ_i$. Іншими словами, якщо початково код програми містить $ПЕ_i$, а після його перетворення у деяку ВФПП цей $ПЕ_i$ може бути усунений, тоді ця ВФПП відповідає критерію *i*.

Зазначені п'ять критеріїв якості ВФПП можна доповнити ще шостим критерієм, що показує практичну доцільність використання певної ВФПП. В даному контексті практична доцільність (критерій 6) означає можливість генерувати вихідний код одразу з ВФПП без додаткових перетворень.

На практиці, у якості ВФПП для процедурних мов програмування найчастіше виступає ГПУ [13]. ГПУ – це орієнтований граф, вершини якого є одиничними операторами чи базовими блоками. Базовий блок – це послідовність операторів з однією точкою входу та однією точкою виходу, що не містить операторів умовних та безумовних переходів. Дуги графа відображають можливі шляхи виконання програми. Дуга графа *(a, b)* є відображенням того, що потік управління досягне блоку *b* одразу після

виконання останнього оператора у блоці *a*. ГПУ, по суті, є відображенням структури потоку виконання операторів у програмі.

ГПУ будується лише для однієї підпрограми (процедури чи функції). Тобто, цей граф не має засобів для представлення міжпроцедурного потоку управління (МПУ). Ця інформація окремо описується графом викликів підпрограм (ГВП). ГВП – це орієнтований граф, вузли якого відповідають підпрограмам. Між вузлами графа *p* та *q* існує дуга *(p, q)* у випадку, якщо функція *p* викликає функцію *q*. Важливо відзначити, що ГВП не містить інформації про кількість та порядок виклику підпрограм. Цю інформацію можна отримати з ГПУ. Отож, на практиці ГПУ та ГВП найчастіше використовують сумісно.

При побудові ГПУ в обов'язковому порядку враховуються безумовні переходи, умовні переходи, оператори вибору, виклики функцій та процедур, цикли, оператори виходу з циклу (break) та продовження циклу (continue). Приклад побудови ГПУ наведено на рис. 1.

```
int n = READ();
int f = 1;
while(n > 0) {
  n--;
  if((n + 1) % 2 == 0){
        f = f * (n + 1);
  } else {
    f = f / (n + 1);
  }
}
int res = f;
PRINT(res);
```

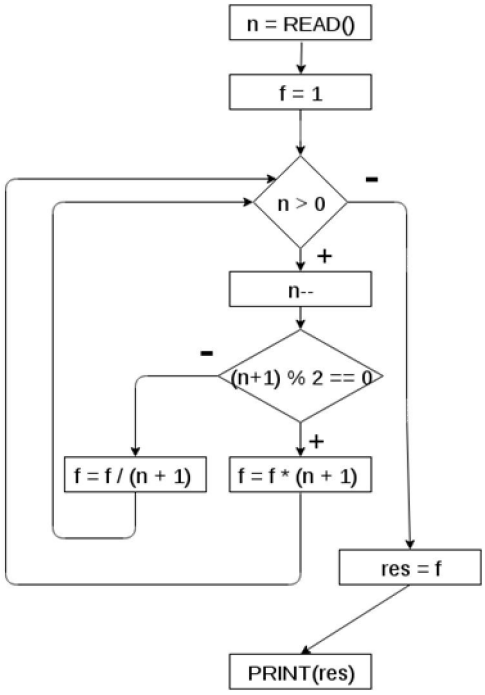

Рис.1. Приклад побудови ГПУ
авторська розробка

Рис. 1 наочно демонструє процедурну природу ГПУ: оператори виконуються покроково, а в результаті виконання операторів модифікуються значення змінних. ГПУ з самого початку створювався як ВФПП для процедурних мов програмування. Тобто цей граф не передбачалося використовувати для вирішення однієї з головних задач трансляції з процедурних мов програмування у функціональні – усунення зміни стану програми. Не важко побачити, що ГПУ не відповідає першим п'яти критеріям якості ВФПП із зазначеного вище переліку. Але, у той же час, ГПУ відповідає критерію 6 і, в принципі, може бути використаний як ВФПП для трансляції програм з ПМП у гібридні ФМП.

Недоліки ГПУ можна частково усунути, перетворивши цей граф у ФСОП. Базовий алгоритм перетворення ГПУ у ФСОП був запропонований у [14, 15], а модифікований алгоритм побудови ФСОП було запропоновано в [16]. У порівнянні з базовим модифікований алгоритм працює швидше.

Важливо зазначити, що у різних статтях значення поняття "форма статичного однократного присвоєння" трактується не однаково. У контексті даної статті будемо використовувати визначення ФСОП, що було запропоноване авторами статей [14, 15, 16].

Спрощене визначення ФСОП можна сформулювати наступним чином: ФСОП представляє собою такий ГПУ, у якому кожній змінній в програмі, що використовується у лівій частині

оператора присвоєння (ОП), відповідає рівно один такий оператор. Права частина ОП може включати довільну кількість змінних.

Для приведення ГПУ у ФСОП потрібно виконати наступні кроки:

1. Для кожної змінної програми *x*, для якої зустрічається більше одного ОП (позначимо їх a_i) та *x* використовується у лівій частині a_i, визначаються нові змінні x_i. Ліва частина усіх a_i замінюється відповідними змінними x_i.

2. Вводяться спеціальні функції φ_k, що використовуються у точках конвергенції x_i. Ці функції потрібні для зв'язування різних визначень змінної *x*, що досяжні у точках конвергенції.

На рис. 2 представлено приклад трансформації ГПУ, що заданий на рис. 1 у ФСОП.

```
int n1 = READ();
int f1 = 1;
int n3, f3;
while (true){
  int n2 = φ1(n1, n3);
  if(n2 <= 0){
    break;
  }
  int f2 = φ2(f1, f3);
  int n2_tmp = n2;
  n2_tmp--; // перезапис!
  n3 = n2_tmp;
  int f4, f5;
  if((n3 + 1) % 2 == 0){
    f4 = f2 * (n3 + 1);
  } else {
    f5 = f2 / (n3 + 1);
  }
  f3 = φ3(f4, f5);
}
int f6 = φ4(f1, f3);
int res = f6;
PRINT(res);
```

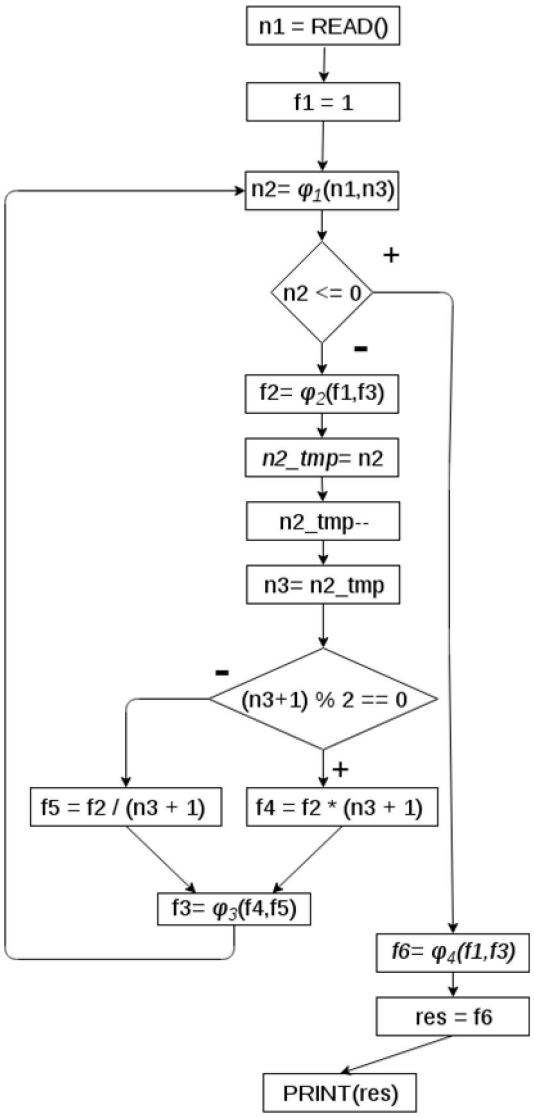

Рис. 2. Приклад трансформації коду програми у ФСОП
авторська розробка

Для того, щоб зберегти однократність присвоєння у програмі, представленій на рис. 2, для змінних *n* та *f*, вводяться нові змінні n_1, n_3 та f_1, f_3, f_4, f_5 відповідно. Для змінної *res* нові змінні не вводяться, оскільки вона використовується у лівій частині лише одного ОП. У точках конвергенції змінних n_1, n_3, f_1, f_3, f_4, f_5 вводяться спеціальні функції φ_1, φ_2, φ_3 та φ_4, що приймають два параметра. Визначення змінних n_2, f_2 та f_6 не є обов'язковим. Ці змінні можна замінити викликом функції φ_k у місцях їх використання.

ФСОП не містить визначення функцій φ_k, тому без додаткових перетворень згенерувати код з ФСОП у вихідну функціональну мову програмування неможливо. Отже критерій 6 для ФСОП не виконується. Базовий та модифікований алгоритми перетворення ГПУ у ФСОП не передбачають, що код вхідної програми містить глобальні змінні, виклики процедур і функцій. Тому ФСОП не задовольняє критеріям 2-5, а відповідає лише критерію 1.

Р.Балланс частково вирішив недоліки ФСОП, запропонувавши розширену форму статичного однократного присвоєння (РФСОП) [17]. У РФСОП вводяться кілька типів функцій φ_k:

1. γ-функція – функція, що відповідає заголовку умовного оператора *if-then-else*. Функція має вигляд $X3 = \gamma (P, X1, X2)$, де P – умова переходу конструкції *if-then-else*, $X1$ – змінна, що відповідає умові P, а $X2$ – змінна, що відповідає інверсній умові $\neg P$.

2. μ-функція визначає значення змінної в тілі циклу. Функція має вигляд $X2 = \mu (X0, X3)$, де $X0$ – змінна, що відповідає початковим установкам циклу, $X3$ – змінна, що використовується в тілі циклу.

3. η-функція визначає значення змінної на виході циклу. Функція має вигляд $X3 = \eta (P, Xfinal)$, де P – умова виконання (завершення) циклу, $Xfinal$ – змінна, що доступна за межами циклу.

На рис. 3 зображено результат перетворення ФСОП (рис. 2) у РФСОП.

```
int n1 = READ();
int f1 = 1;
int n3, f3;
while (true){
 //  μ - функція
 int n2 = is_bound(n3)?n3:n1;
 if(n2 <= 0){
   break;
 }
 //  μ - функція
 int f2 = is_bound(f3)?f3:f1;
 n3 = safe_dec(n2);
 int f4, f5;
 bool P1 = (n3 + 1) % 2 == 0;
 if(P1){
  f4 = f2 * (n3 + 1);
 } else {
  f5 = f2 / (n3 + 1);
 }
 //  γ - функція
 f3 = P1 ? f4 : f5;
}
bool P2 = n1 > 0;
//  η - функція
int f6 = P2 ? f3 : f1;
int res = f6;
PRINT(res);
```

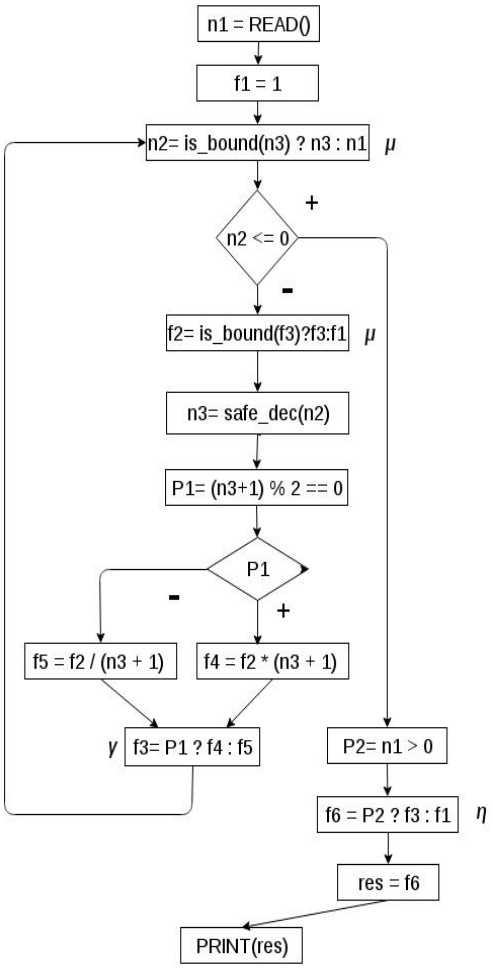

Рис. 3. Перетворення ФСОП у РФСОП
авторська розробка

РФСОП має ті ж властивості, що й ФСОП, але окрім того, у ній присутня уся необхідна інформація для генерації коду вихідної програми. Тому РФСОП відповідає не тільки критерію 1, а також критерію 6. РФСОП так само як і ФСОП не відповідає критеріям 2-5.

Хоча ФСОП та РФСОП надають деякі переваги у вирішенні поставленої проблеми у порівнянні з ГПУ, але вони є його модифікаціями. А ГПУ, як відомо, не відповідає жодному з п'яти критеріїв якості ВФПП. Тому наступним логічно обґрунтованим кроком є розгляд такої ВФПП, що не буде базуватися на ГПУ. Такою формою виступає ГЗЗ [18] та його модифікація ГЗЗС [19]. Властивості ГЗЗС не залежать від значень змінних, від порядку, в якому ці значення з'являються та від місця визначення змінних [6, 20]. ГЗЗС є паралельним представленням, у якому відношення порядку між операціями задається частково і базується лише на залежностях за даними та станами. Тобто, на відміну від ГПУ, ГЗЗС передбачає обчислення значень лише у разі потреби. Звідси слідує, що обчислення ГЗЗС може бути таким, що завершилось, навіть якщо виконання вхідної програми буде нескінченним. Ця особливість графу пояснюється тим, що значення, обчислення яких відбувається під час виконання програми але які не потрібні для побудови ГЗЗС, не виконуються.

ГЗЗС – це дводольний граф, що може бути визначений наступним чином [21]: $G = (N, E_V, E_S, L, N_0, N_\infty)$. Складовими частинами цього графа є:

1. N – множина вершин графа. Граф включає три типи вершин: операторні вершини, вершини-значення, вершини стану.
2. E_V – множина дуг графа ($E_V \subseteq N \times N$), що відповідають залежностям значень.
3. E_S – множина дуг графа ($E_S \subseteq N \times N$), що відповідають залежностям стану.
4. L – множина функцій, що асоціюють кожну вершину графа з відповідним оператором.
5. N_0 – початкова вершина графа.
6. N_∞ – кінцева вершина графа.

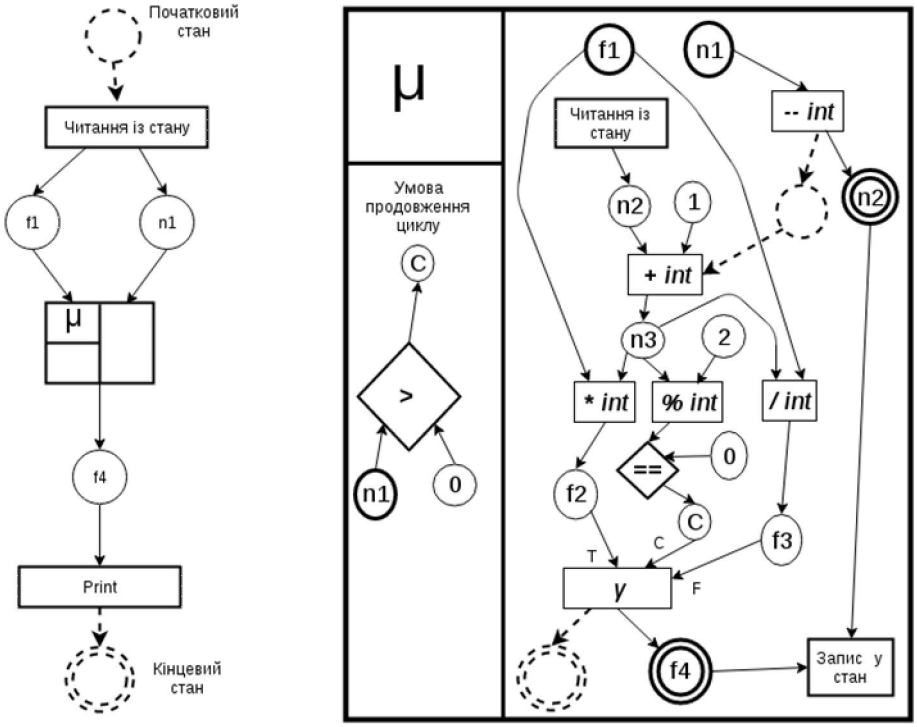

Рис. 4. Приклад побудови ГЗЗС
авторська розробка

Цикли та умовні оператори представляються у ГЗЗС за допомогою спеціальних вершин μ та γ.

На рис. 4 наведено приклад побудови ГЗЗС для програми, заданої на рис. 2.

У ГЗЗС непрямим чином забезпечується властивість однократного присвоєння, оскільки існує однозначна відповідність між операторними вершинами та вершинами-значеннями. Кожній вершині-значенню можна присвоїти ім'я, що буде унікальним і відповідатиме змінній програми у ФСОП.

Усі значення у ГЗЗС зчитуються із стану або продукуються у результаті виконання операторних вершин. Тому глобальні змінні при побудові ГЗЗС оброблюються аналогічно локальним. Таким чином ГЗЗС відповідає не тільки критерію 1, а й критеріям 2, 3, 5.

Кожна вершина, що відповідає оператору вводу/виводу продукує новий стан. Тому ГЗЗС відповідає критерію 4.

ГЗЗС містить усю необхідну інформацію для генерації вихідної програми, а отже відповідає критерію 6.

У рамках статті було проаналізовано відповідність ВФПП критеріям 1 – 6. Підсумок результатів аналізу наведено у таб. 1.

Таблиця 1. Підсумкова таблиця відповідності ВФПП критеріям 1 – 6
авторська розробка

ВФПП / Критерій	**1**	**2**	**3**	**4**	**5**	**6**
ГПУ	-	-	-	-	-	+
ФСОП	+	-	-	-	-	-
РФСОП	+	-	-	-	-	+
ГЗЗС	+	+	+	+	+	+

Висновки. Проведений аналіз ВФПП показав, що ГПУ не придатний для ефективного вирішення задачі трансляції програм із процедурних мов програмування у функціональні, оскільки ГПУ не відповідає жодному із п'яти критеріїв якості ВФПП. У той же час ГПУ відповідає критерію 6, тому ГПУ, теоретично, можна використовувати як ВФПП для трансляції програм у гібридні функціональні мови програмування.

На відміну від ГПУ ФСОП та РФСОП відповідають критерію 1. ВФПП, що відповідають критерію 1, можна використовувати для трансляції такої підмножини програм, які містять побічні ефекти лише типу 1. ФСОП не відповідає критерію 6, тому на практиці цю ВФПП використовувати недоцільно.

Найбільш придатним для вирішення вищезгаданої задачі є ГЗЗС, оскільки ця ВФПП відповідає усім запропонованим критеріям якості ВФПП. Крім цього, ГЗЗС відповідає критерію 6, тому його доцільно використовувати не тільки як теоретичний спосіб внутрішнього представлення програми у трансляторі, а і як основу для побудови реальної системи.

Серед актуальних проблем для проведення подальших досліджень можна виокремити задачу ефективної побудови ГЗЗС для програм, що містять оператори безумовного переходу (*goto*), завершення циклу (*break*) та продовження циклу (*continue*).

1. Зубов М. В. Применение универсальных промежуточных представлений для статического анализа исходного программного кода / М. В. Зубов, А. Н. Пустыгин, Е. В. Старцев. // Доклады Томского государственного университета систем управления и радиоэлектроники. – март 2013. – №1 (27). – С. 64–68.
2. Зубов М. В. Математическое моделирование универсальных многоуровневых промежуточных представлений для статического анализа исходного кода / М. В. Зубов, А. Н. Пустыгин, Е. В. Старцев. // Доклады Томского государственного университета систем управления и радиоэлектроники. – сентябрь 2014. – №3 (33). – С. 94–99.
3. Рыбаков А. А. Алгоритм создания случайных графов потока управления для анализа глобальных оптимизаций в компиляторе / Алексей Анатольевич Рыбаков. // Международная конференция по параллельным и распределенным компьютерным системам. – Харьков, 13-14 марта 2013. – С. 269–275.
4. Битнер В. А. Построение универсального линеаризованного графа потока управления для использования в статическом анализе кода алгоритмов / В. А. Битнер, Н. В. Заборовский. // Модел. и анализ информ. систем. – 2013. – Т. 20, №2. – С. 166–177.
5. Дроздов А. Ю. Компонентный подход к построению оптимизирующих компиляторов: дис. докт. техн. наук : 05.13.11 / Дроздов Александр Юльевич – Москва, 2010. – 307 с.
6. Value dependence graphs: representation without taxation / D.Weise, R. F. Crew, M. Ernst, B. Steensgaard. // POPL '94 Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 1994. – Pp. 297–310.

7. Stanier J. The value state dependence graph revisited / J. Stanier, L. Alan. // In Proceedings of the Workshop on Intermediate Representations. – 2011. – Pp. 53–60.
8. Bahmann H. Perfect Reconstructability of Control Flow from Demand Dependence Graphs / H.Bahmann, N. Reissmann, J. Magnus, J. C. Meyer. // ACM Transactions on Architecture and Code Optimization. – January 2015. – Volume 11 Issue 4. – Article No. 66.
9. Reinhard W. Compiler Design: Virtual Machines / W. Reinhard, S. Helmut., 2011. – Springer. – Pp. 7–55.
10. Reinhard W. Compiler Design: Virtual Machines / W. Reinhard, S. Helmut., 2011. – Springer. – Pp. 57–104.
11. Launchbury J. State in Haskell / J. Launchbury, J. Peyton, L. Simon. // Lisp and Symbolic Computation. – 1995. – №8. – Pp. 293–341.
12. Sabry A. What is a purely functional language? / Amr Sabry. // Journal of Functional Programming. – Volume 8 Issue 1. – January 1998. – Pp. 1 – 22.
13. Compilers: Principles, Techniques, and Tools / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. – 2nd Ed. – Boston: Addison-Wesley Longman Publishing Co., 2006.
14. Rosen B. K. Detecting equality of variables in programs / B. K. Rosen, M. N. Wegman, F. K. Zadeck. // Proceeding POPL '88 Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 1988. – Pp. 1–11.
15. Rosen B. K. Global value numbers and redundant computations / B. K. Rosen, M. N. Wegman, F. K. Zadeck. // Proceeding POPL '88 Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 1988. – Pp. 12–27.
16. An efficient method of computing static single assignment form / [R. Cytron, J. Ferrante, B. K. Rosen та ін.]. // Proceeding POPL '89 Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – 1989. – Pp. 25–35.
17. Ballance R. A. The program dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. / R. A. Ballance, A. B. Maccabe, K. J. Ottenstein. // In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90). – 1990. – Pp. 257–271.
18. Byers D. Syntax-Directed Construction of Value Dependence Graphs / D. Byers, M. Kamkar, T. Palsson. // ICSM '01 Proceedings of the IEEE International Conference on Software Maintenance. – 2001. – Pp. 692 – 703.
19. Lawrence A. C. Optimizing compilation with the Value State Dependence Graph / Lawrence Alan C. – 2007. – 183 p.
20. Колчин А. В. Метод редукции анализируемого пространства поведения при верификации формальных моделей распределенных программных систем // Искусственный интеллект. –2013. – №4. –С. 113–126.
21. Johnson N. E. Code size optimization for embedded processors / Neil E. Johnson. – Technical Report UCAM-CL-TR-607. – November 2004. – 257 p.