УДК 004.43

Реализация оптимизирующих трансформаций в системе предикатного программирования

Каблуков И.В. (Институт систем информатики СО РАН),

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описываются оптимизирующие трансформации склеивания переменных, замены хвостовой рекурсии циклом, открытой подстановки, упрощения и оформления. Результатом трансформаций является эффективная императивная программа. Используется потоковый анализ, включающий построение графа вызовов и определение области жизни переменных программы.

Ключевые слова: *функциональное программирование, трансформации программ, потоковый анализ, склеивание переменных, хвостовая рекурсия, открытая подстановка.*

1. Введение

Программа на языке предикатного программирования Р [6] является предикатом, описывающим алгоритм решения математической задачи в форме исполняемого оператора. Имея общие особенности с функциональным и императивным программированием, предикатное программирование, тем не менее, принципиально отличается от них.

Эффективность предикатных программ достигается применением при трансляции следующих оптимизирующих трансформаций:

- склеивания переменных, реализующие замену нескольких переменных одной;
- замены хвостовой рекурсии циклом;
- подстановки определения предиката на место его вызова;
- кодирования алгебраических типов (списков, деревьев) через массивы и указатели.

Данные трансформации реализуют *оптимизацию среднего уровня* с переводом предикатной программы в эффективную императивную программу. Построение алгоритмов такой оптимизации – новая задача, характерная для предикатного, но не функционального программирования. Оптимизация среднего уровня принципиально отличается от

традиционной оптимизации для императивных языков. Трансформации склеивания переменных и кодирования объектов алгебраических типов аналогов не имеют.

Эффективность программы после применения трансформаций обеспечивается оптимизацией, реализуемой программистом при построении предикатной программы. Фактически, набор применяемых трансформаций планируется при построении программы. И задача трансформатора – «угадать», найти этот набор по программе. Иначе говоря, ставится задача разработки алгоритмов трансформаций не для произвольной программы, а для программы, ориентированной на трансформацию.

В настоящей работе описываются алгоритмы следующих трансформаций: склеивания переменных, замены хвостовой рекурсии циклом, открытой подстановки программ на места их вызовов. Реализация трансформаций использует результаты потокового анализа предикатной программы. Дополнительно осуществляются простые оптимизирующие преобразования, упрощающие программу.

Во втором разделе настоящей работы дается общее представление о системе предикатного программирования. В третьем разделе описывается структура реализуемого блока трансформаций. В последующих разделах описываются составные части этого блока. В четвертом разделе приводятся алгоритмы потокового анализа. В разделах с пятого по восьмой описываются трансформации подстановки определения предиката на место вызова, замены хвостовой рекурсии циклом и склеивания переменных, а также упрощения и оформления. В девятом разделе приводятся примеры трансформаций программ. Десятый раздел – обзор смежных работ по оптимизирующим трансформациям.

2. Система предикатного программирования

Программа на языке Р [6] является предикатом, описывающим алгоритм решения математической задачи в форме исполняемого оператора.

Предикатная программа состоит из набора рекурсивных программ (определений предикатов) на языке Р следующего вида:

<имя программы>(<описания аргументов>**:** <описания результатов>)
pre <предусловие>
{ <оператор> }
post <постусловие>

Необязательные конструкции предусловие и постусловие являются формулами на языке исчисления предикатов; они используются для улучшения понимания программ и для дедуктивной верификации [3, 9, 12, 20]. Ниже представлены основные конструкции языка Р:

оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов и результатов предиката и локальных переменных.

<переменная> = <выражение>

{<оператор1>; <оператор2>}

<оператор1> || <оператор2>

if (<логическое выражение>) <оператор1> **else** <оператор2>

<имя программы>(<список аргументов>**:** <список результатов>)

<тип> <пробел> <список имен переменных>

В предикатном программировании запрещены такие языковые конструкции, как циклы и указатели, серьезно усложняющие программу. Вместо циклов используются рекурсивные функции, а вместо массивов и указателей – списки.

В наборе предикатов программы существует главный предикат, с которого начинается исполнение программы и который вызывает остальные предикаты по цепочкам вызовов.

Трансформации. Получение эффективной программы для функциональных языков проблематично даже с применением изощренной оптимизации в процессе трансляции. После отладки программы на функциональном языке для достижения требуемой эффективности ее приходится переписывать на императивный язык.

К предикатной программе применяется набор трансформаций с получением эффективной программы на императивном расширении языка P, после чего программа может конвертироваться на любой из императивных языков: C, C++, ФОРТРАН и др. Трансформация программ с внутреннего представления на императивное расширение происходит в 4 этапа, на каждом из которых реализуется одна из базовых трансформаций:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование алгебраических типов: списков, деревьев низкоуровневыми структурами с использованием массивов и указателей.

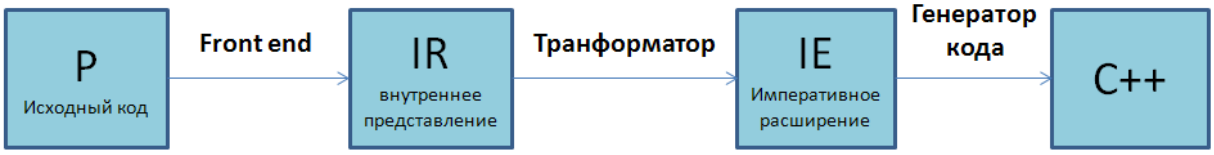

Рис. 2.1. Схема реализации языка P.

Эффективность программы после применения трансформаций обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи [3, 9, 12, 20]. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче [1, 4, 5, 8, 19]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [18]) оптимизация программы полностью возлагается на транслятор, в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду. Разумеется, функциональное программирование существенно уступает в эффективности, поскольку даже применением изощренных методов оптимизации невозможно автоматически воспроизвести серию оптимизаций, совершаемых программистом вручную.

Императивное расширение языка P определяет следующие дополнительные языковые конструкции:

break

for (<тип> <параметр цикла> = <выражение>; <условие завершения>; <пересчет параметра>) { <оператор> }

while (<выражение>) { <оператор> }

loop { <оператор> }

if (<выражение>) { <оператор> }

#<метка>

<метка>**:** <оператор>

Семантика циклов **for** и **while** соответствует языку C++. Оператор вида #<метка> реализует переход на оператор, помеченный указанной меткой.

Приведенные выше конструкции возникают в программе в результате проведения трансформаций предикатной программы. Использование этих конструкций в исходной предикатной программе недопустимо.

3. Структура блока трансформаций

Блок трансформаций реализует оптимизацию среднего уровня, преобразуя программу с языка предикатного программирования на язык императивного расширения. Производятся следующие трансформации:

- склеивание переменных;

- замена хвостовой рекурсии циклом;
- подстановка определения предиката на место его вызова;
- кодирование алгебраических типов.

Для проведения трансформаций требуется предварительный потоковый анализ программы, который строит граф вызовов и определяет аргументы, результаты и живые переменных операторов. После проведения трансформаций потоковая информация о программе может стать недействительной. Поэтому перед каждой трансформацией необходимая потоковая информация обновляется: перед трансформацией подстановки определения предиката граф вызовов строится заново (т.к. замена хвостовой рекурсии циклом сделала некоторые предикаты нерекурсивными, и они становятся пригодны для подстановки на место вызова), а перед трансформацией кодирования алгебраических типов обновляется информация о живых аргументах операторов (т.к. склеивание переменных изменило их время жизни). В конце после применения трансформаций реализуется серия упрощений программы.

Рис. 3.1. Схема блока трансформаций.

Граф вызов предикатной программы — это ориентированный граф (орграф) с предикатами в качестве вершин. Вызов предиката A в теле предиката B определяет дугу в графе B→A.

Понятия живости переменной определяется по отношению к ее вхождению в операторе. Переменная является *живой*, если ее значение используется при исполнении программы после данного оператора.

Обоснование порядка трансформаций. Для рекурсивных предикатов нецелесообразно проводить подстановку тела на место вызова. Поэтому замена хвостовой рекурсии циклом производится перед подстановкой тел предикатов. Трансформацию склеивания можно было бы реализовать после замены хвостовой рекурсии и подстановки определения предиката.

4. Потоковый анализ

Для проведения трансформаций подстановки тела предиката на место вызова и склеивания переменных требуется построение графа вызовов программы.

Построение графа вызовов. Для каждой переменной предикатного типа находятся ее *заместители* — это предикаты, являющиеся возможными значениями переменной. Определим массив S. Его индексы — имена переменных предикатного типа, а значения – множества заместителей. Вначале набор индексов пустой. Алгоритм построения множества заместителей реализуется многократным просмотром программы. На очередном просмотре набор индексов и множества заместителей пополняются. Работа алгоритма завершается, когда на очередном просмотре программы множества заместителей останутся неизменными.

Алгоритм построения графа вызовов следующий. Именами A, B, C будут обозначаться предикаты, именами a, b, c — переменные предикатного типа. В программе ищутся вхождения имен предикатов.

Правило 1**.** Имеется вызов B(…) в теле предиката A. Тогда в граф вызовов добавляется дуга A→B.

Правило 2**.** Пусть в теле предиката A находится вызов B(…, C, …), и есть определение предиката B(…, a, …) {…}. Тогда в множество S[a] добавляется C.

Правило 3**.** Имеется присваивание a = C. Тогда в множество S[a] добавляется C.

Правило 4. Имеется вызов a(…) в теле предиката A. Тогда в граф добавляются дуги A→C для каждого C из множества S[a].

Правило 5**.** Пусть в теле предиката A находится вызов B(…, a, …: …), и есть определение предиката B(…, c, …: …) {…}. Тогда в множество S[c] добавляется множество S[a]**.**

Правило 6**.** Имеется присваивание b = a. Тогда в множество S[b] добавляется множество S[a].

Правило 7**.** Пусть в теле предиката A находится вызов B(…**:** …, a, …), где a — результат вызова. Пусть есть определение предиката B(…**:** …, c, …) {…}. Тогда всякий заместитель c является заместителем a. Поэтому в множество S[a] добавляется множество S[c].

Компоненты сильной связности. Две вершины s и t орграфа *сильно связаны*, если существует ориентированный путь из s в t и ориентированный путь из t в s. Компонента сильной связности есть максимальное множество сильно связанных вершин. Будем называть *рекурсивным слоем* набор предикатов, чьи вершины принадлежат одной компоненте сильной связности графа вызовов. *Рекурсивный вызов* — вызов предиката из того же рекурсивного слоя, в котором находится вызывающий предикат.

Компоненты сильной связности графа вызовов находятся по алгоритму Тарьяна [7, разд. 3.5.3], который имеет линейную сложность. Алгоритм можно считать вариацией алгоритма поиска в глубину. Для каждой вершины строятся пути из нее, и если один из путей вернулся в начальную вершину, то все вершины на пути объединяются в одну компоненту связности.

Пусть в теле предиката A есть вызов B(…, c, …), где c — переменная предикатного типа со значением A. Пусть есть определение B(…, d, …) {… d(…); …}. Тогда A вызывает B, а B неявно вызывает A. Такая форма рекурсии с неявным вызовом через предикатную переменную запрещена в языке Р. Для каждой дуги A→B, порожденной вызовом через переменную проверяется, что A и B не принадлежат одному рекурсивному слою.

Определение порядка предикатов. Необходимо определить порядок предикатов, в котором проводится трансформация склеивания. Склеивания, которые были произведены на аргументах и результатах – формальных параметров предиката, должны быть перенесены на аргументы и результаты, являющиеся фактическими параметрами во всех вызовах данного предиката. Поэтому склеивание в теле предиката должно быть проведено раньше, чем склеивания в теле другого предиката, содержащего вызов первого.

Граф приводится к ациклическому виду стягиванием компонент сильной связности в одну вершину. Орграф становится ациклическим, и на нем можно определить порядок с помощью алгоритма обратной топологической сортировки [7, разд. 3.1.2]. Обратный топологический порядок на орграфе это порядок, при котором для каждой дуги A→B вершина B находится раньше A. Важно отметить, что граф вызовов предикатной программы связный: существует главный предикат, с которого начинается исполнение программы и который вызывает остальные предикаты по цепочкам вызовов. Поэтому все используемые предикаты будут упорядочены.

Нахождение живых переменных операторов. Для трансформации склеивания переменных необходимо определение аргументов, результатов и живых переменных операторов в телах предикатов. Нахождение аргументов и результатов операторов происходит просмотром программы во внутреннем представлении.

Живыми (активными) аргументами оператора являются переменные, чьи значения будут использоваться после завершения исполнения этого оператора. Вхождения живых переменных нельзя склеивать с другими переменными, поскольку значения переменных используются далее. Склеиванию подлежат переменные, не являющиеся живыми. Отметим, что формальные параметры – результаты предиката являются живыми для каждого

оператора его тела, так как они предположительно будут использоваться после завершения исполнения предиката.

Описываемый ниже алгоритм является адаптацией алгоритма [1, разд. 9.2.5]. Для каждого оператора определяются преемники – операторы, исполняющиеся сразу после него. Пусть Op – произвольный исполняемый оператор предиката F. Определим множества In[Op] и Out[Op] – множества переменных, живых перед и после исполнения оператора, соответственно. Пусть args[Op] – множество аргументов, а results[Op] – множество результатов оператора Op; Exit – фиктивный пустой оператор, вставляемый после каждого последнего оператора на каждой ветви тела предиката.

Алгоритм имеет следующий вид:

```
In[Exit] = формальные параметры – результаты предиката F
while (внесены изменения в любое In)
    for (каждый исполняемый оператор Op предиката) {
            Out[Op] = ⋃S – преемник Op In[S];
            In[Op] = args[Op] ∪ (Out[Op] \ results[Op]);
    }
```

Присваивание In[Exit] результатов предиката F необходимо для определения в качестве живых переменных результатов предиката для каждого его оператора по цепочкам преемников. После исполнения оператора $Out[Op] = \bigcup_{S\ –\ преемник\ Op} In[S]$ определяются в качестве живых все переменные, которые живы перед исполнением какого-либо преемника этого оператора. После исполнения оператора In[Op] = args[Op] ∪ (Out[Op] \ results[Op]) определяются в качестве живых все переменные, которые либо используются в качестве аргумента в этом операторе, либо используются после этого оператора и не переопределяются в нем. Таким образом, переменная жива после исполнения оператора, если ее текущее значение используется в одной из цепочек преемников этого оператора или она является результатом предиката.

В трансформации склеивания используются множества переменных Out[Op], живых после исполнения оператора.

5. Подстановка определения предиката на место вызова

Пусть A(x: y) { S } – определение предиката в программе, а A(e: z) – вызов предиката в теле некоторого другого предиката B. Здесь x, y, z обозначают списки переменных, а e – список выражений. В соответствии с операционной семантикой языка предикатного

программирования [13] *подстановка определения предиката на место вызова* A(e: z) есть замена вызова следующей композицией:

| x | = | e |; { S }; | z | = | y | . (5.1)

Здесь конструкции | x |, | z | и | y | являются мультипеременными, а | e | – мультивыражением. Оператор присваивания вида | x | = | e | называется *групповым оператором присваивания.*

Можно убрать присваивание | z | = | y |, заменив вхождения y на z в теле S. Если среди e есть переменные, то их также можно использовать в теле S вместо соответствующих переменных в x при выполнении определенных ниже условий. Тогда вызов заменяется следующей композицией:

| x' | = | e' |; { S' } . (5.2)

S' — тело предиката S с заменой z на y и заменой части параметров x на соответствующие переменные в e. Пусть в списке e встречаются переменная a, которой в списке x соответствует переменная b. Если a не используются после вызова или b не перевычисляется в S, то можно заменить b на a в S. x' и e' — оставшиеся части списков x и e.

Возможно, что локальные имена в теле S могут также встречаться в теле предиката B. Это не вызывает проблем, поскольку эти переменные создаются как разные переменные во внутреннем представлении. Однако при обратной трансляции в текст (посредством инструмента pretty-printer) реализуется переименование переменных.

Гиперфункция - предикат с несколькими ветвями [6, 10]. Рассмотрим определение гиперфункции A(x: y #1 : z #2) { S } с двумя ветвями и вызов этой гиперфункции с последующими обработчиками переходов по ветвям:

A(e: g #M1 : f #M2) case M1: { S1 } case M2: { S2 }; N

Здесь N – следующий за обработчиками оператор. После исполнения оператора вызова управление переходит к оператору S1, если предикат завершился по первой ветке, или S2, если по второй. Далее управление переходит к оператору N.

При открытой подстановке тела предиката A вызов заменяется композицией | x' | = | e' |; { S' } { M1: S1; #M3 } { M2: S2: #M3 } , где M3 - метка оператора N. S' - это оператор S, в котором операторы выхода #1 и #2 заменены на #M1 и #M2 соответственно, результаты предиката y и z заменены на результаты вызова g и f, а также часть параметров x заменены на соответствующие переменные в e как описано выше для обычного предиката.

В соответствии с операционной семантикой [13] вычисление выражений, входящих в набор e в групповом операторе присваивания | x | = | e |, проводится параллельно. Реализация такого параллелизма на процессорах с обычной архитектурой не эффективна. Поэтому проводится *раскрытие* группового оператора присваивания его заменой набором обычных операторов присваивания. В общем случае раскрытие группового оператора возможно при использовании дополнительных промежуточных переменных. Например, раскрытие оператора | a, b | = | b, a | реализуется операторами t = a; a = b; b = t с использованием дополнительной переменной t. В большинстве случаев раскрытие возможно без использования дополнительных переменных; иногда достаточно поменять местами операторы присваивания. Например, раскрытие | a, b | = | c, a | реализуется присваиваниями: b = a; a = c.

6. Замена хвостовой рекурсии циклом

Существует специальный случай рекурсии, называемый *хвостовой рекурсией*, когда можно заменить рекурсию циклом. Рекурсивный вызов предиката определяет хвостовую рекурсию, если:

- имя вызываемого предиката совпадает с именем определяемого предиката, в теле которого находится вызов;
- вызов является последней исполняемой конструкцией в определении предиката, содержащем вызов.
- результаты вызова совпадают с соответствующими формальными параметрами – результатами определяемого предиката.

Последняя исполняемая конструкция оператора S принадлежит множеству last(S), которое определим для различных видов операторов S: last(A; B) = last(B), last(**if** (C) A **else** B) = last(A) ∪ last(B), last(D(e: z)) = {D(e: z)}, last(A || B) = ∅. Однако если параллельный оператор A || B реализуется последовательным исполнением A и B, например, как B; A, то last(A || B) = last(A).

Понятие хвостовой рекурсии проиллюстрируем на примерах. В программе умножения через сложение

Умн(**nat** a, b**:** **nat** c) { **if** (a = 0) c = 0 **else** c = b + Умн(a – 1, b) }

рекурсивный вызов Умн(a – 1, b) не является хвостовым, поскольку после завершения исполнения вызова исполняется операция “+”. Хвостовой является рекурсия для каждого из двух рекурсивных вызовов в программе вычисления наибольшего общего делителя:

```
D(nat a, b: nat c)
{ if (a = b) c = a else if (a < b) D(a, b – a: c) else D(a – b, b: c) }
```

Хвостовая рекурсия может быть заменена циклом, что существенно повышает эффективность программы; в частности, дает возможность использовать открытую подстановку. Определение хвостовой рекурсии представлено далее как специальный случай подстановки определения предиката на место вызова (см. разд. 5).

Пусть имеется рекурсивное определение предиката A(x: y) { S } и хвостовой рекурсивный вызов A(e: y) внутри оператора S. Подставим определение предиката A на место вызова A(e: y). Результатом подстановки является фрагмент: | x | = | e |; { S }. Обозначим через S′ оператор, полученный заменой в S вызова A(e: y) на данный фрагмент. Очевидно, можно заменить в S′ второе вхождение S передачей управления на начало оператора S′. В итоге определение предиката A преобразуется к виду: A(x: y) {M: S″ }, где S″ получается из S′ заменой вызова A(e: y) парой операторов: | x | = | e |; **goto** M. Данное преобразование есть трансформация *замены хвостовой рекурсии циклом*.

Алгоритм трансформации. Пусть есть рекурсивный предикат D(**nat** a, b**: nat** c) { S }. В теле S ищутся последние исполняемые конструкции по определенным выше правилам для управляющих операторов суперпозиции, условных и параллельных операторов. Если в найденном множестве есть рекурсивные вызовы, проверяется, что результаты этих вызовов совпадают с соответствующими формальными результатами предиката. Для выявленных хвостовых рекурсий производится замена вызова на присваивание аргументам предиката аргументов вызова и переход в начало тела предиката.

Если в рекурсивном определении предиката A все рекурсивные вызовы имеют вид хвостовой рекурсии, то применение трансформации ко всем этим вызовам преобразует тело предиката в цикл, а определении предиката A – в нерекурсивное определение. После этой трансформации становится эффективной постановка определения предиката A на место вызова A в теле другого предиката.

Применение трансформации замены хвостовой рекурсии циклом покажем для программы вычисления наибольшего общего делителя, приведенной выше. Трансформация двух рекурсивных вызовов предиката D дает следующую программу:

```
D(nat a, nat b: nat c) {
M:    if (a = b) c = a
      else  if (a < b) |a, b| = |a, b – a|; goto M
            else |a, b| = |a – b, b|; goto M
```

```
}
```

Раскроем групповые операторы присваивания, а также заменим фрагмент с операторами перехода на цикл **loop**. Получим:

```
D(nat a, nat b: nat c) {
        loop {
                if (a = b) {c = a; break; }
                if (a < b) b = b – a
                else a = a – b
        }
}
```

Замена фрагмента с операторами перехода на цикл **loop** не меняет процесса исполнения. Преобразование такого рода, улучшающее структуру программы, называется *оформлением.*

7. Склеивание переменных

Трансформация *склеивания переменных* a ← x есть замена в предикатной программе всех вхождений каждой переменной из списка переменных x на переменную a. Например, склеивание переменных a ← b, c представляет замену всех вхождений в программе имен b и c на имя a.

В отличие от задачи экономии памяти [4], склеиванию подлежат результаты программы с аргументами или локальные переменные с результатами, между которыми имеется информационная связь. Задача склеивания переменных не актуальна для оптимизации функциональных программ**:** там можно было бы рассматривать лишь склеивание аргументов функций и локальных переменных конструкций **let** и **where**. Эта задача не возникает также для императивных программ, поскольку практически все рассматриваемые здесь склеивания обычно проведены программистом в императивной программе.

В языке P запрещено присваивание вида: x:=op(x, y). Вместо этого используется присваивание x:=op(x1, y) в предположении, что переменная x1 должна быть склеена с переменной x при трансляции на императивное расширение языка P. Например, при склеивании переменных c и d оператор c:=d+1 будет преобразован в оператор присваивания c:=c+1, а оператор a:=b при склеивании a и b превратится в оператор a:=a, удаляемый из программы. Склеивание переменных, массивов или списков, предотвращает копирование больших структур данных.

Склеивание переменных может задаваться в предикатной программе. Если имеется результирующая переменная, имя которой завершается штрихом, при наличии аргумента с тем же именем, то результирующая переменная должна быть склеена с аргументом.

Например, результат a′ при наличии аргумента a неявно определяет трансформацию a ← a′. Набор склеиваний может быть частично задан программистом. Необходимо проверить его корректность и дополнить.

Эквивалентность программы до и после проведения трансформации склеивания достигается при выполнении ряда условий. Одно из них: аргумент a, склеенный с результатом b, не является живой после присваивания переменной b.

В общем случае задача склеивания очень сложна. В действительности, набор склеиваний планируется программистом при написании предикатной программы. Поэтому нет задачи нахождения наилучшего варианта склеивания. Необходимо воспроизвести вариант склеивания, запланированный программистом.

Общая схема реализации. *Регион склеивания* для оператора G есть пара <x: y>, где x — подмножество аргументов оператора, а y — один из его результатов, все переменные имеют одинаковый тип, а переменные x не являются живыми переменными оператора G. Неживую переменную можно склеить с другой, так как она не используется далее в программе. **Пример**. Пусть имеется оператор F с аргументами a, b, c, d, e и результатами f, g, h. Пусть переменные a, b, d и g, h имеют натуральный тип, а переменные c, e и f — массивы. Тогда для оператора F регионы склеивания могут быть такими: <a, b: g>, <d: h> и <c, e: f>.

Регион склеивания вида <a**:** g> , где a и g — переменные, является *командой склеивания*, определяющей замену переменной a на g.

Запрет на склеивание для оператора G есть пара вида {a**:** g} , где a и g — переменные. Он запрещает склеивание переменной g с переменной a.

Алгоритм склеивания для программы в целом реализуется перебором всех предикатов программы в порядке, определенном потоковым анализатором. Он гарантирует, что склеивания для формальных параметров предиката будут перенесены на фактические параметры в вызовах этого предиката, что необходимо в случае открытой подстановки. Для рекурсивных вызовов перенос склеиваний на аргументы вызова необходим лишь для вызовов с хвостовой рекурсией.

Склеивание переменных предиката реализуется с помощью алгоритма уточнения регионов. Дерево тела предиката обходится снизу вверх; строятся начальные регионы склеивания, которые далее уточняются, проверяя выполнение ограничений. По построенным регионам склеивания для предиката выбираются команды склеивания. Применение команд склеивания к программе реализуется независимым просмотром программы. Там же реализуется замена параллельных операторов операторами суперпозиции, о которой сказано ниже.

Построение регионов склеивания. Для построения регионов склеивания дерево тела предиката обходится снизу вверх. В дереве тела предиката нижними операторами являются операторы присваивания и вызовов предикатов. Для этих операторов строятся начальные регионы склеивания определенным ниже образом. Регионы операторов суперпозиции, параллельных и условных операторов строятся на основе уже построенных регионов подоператоров. Построенные таким способом регионы являются корректными, т.е. при проведении склеиваний по этим регионам сохраняется эквивалентность программы.

Для оператора **присваивания** вида a = b строится регион <b: a>, при условии, что b не является живой переменной этого оператора. Склеивание превратит этот оператор в тождественный оператор присваивания b = b, который далее будет удален из тела предиката. Для оператора присваивания с несколькими аргументами $a = b_1 + \ldots + b_n$ регион строится только если тип переменной присваивания — список, для которого операция конкатенации '+' не вычисляет новое значение, а добавляет к левому аргументу операции правый. Для оператора такого вида строиться регион $<b_1: a>$. В таком случае при склеивании исходный оператор заменится на оператор $b_1 = b_1 + \ldots + b_n$, в котором b_1 не будет копироваться.

Для определения предиката, вызываемого нерекурсивным оператором **вызова**, команды склеивания уже построены, что достигается выбранным порядком обработки предикатов по графу вызовов. Пусть есть определение предиката F(…, a, …**:** …, b, …) {…}, в котором переменные a и b склеиваются, и есть нерекурсивный вызов F(…, d, …**:** …, e, …). При последующей трансформации подстановки определения предиката на место вызова переменные a и b будут заменены на, соответственно, d и e. Так как в предикате F переменные a и b склеились, то переменные d и e тоже можно склеить. Поэтому строится регион <d: e>.

Если соответствующий аргумент вызова – не переменная, то регионов не строится, за исключением случая конкатенации строк $b_1 + \ldots + b_n$, описанного выше для оператора присваивания.

Для хвостовых рекурсий регионы строятся после просмотра тела рекурсивного предиката и построения команд склеивания. Пусть имеется определение предиката F(…, a, …**:** …, b, …) {…}, для которого была построена команда склеивания <a: b>. Пусть в теле предиката есть хвостовой рекурсивный вызов F(…, d, …: …, b, …). Поскольку замена хвостовой рекурсии циклом интерпретируется как открытая подстановка тела на место вызова, соответствующие переменные d и b, позиции которых в вызове соответствуют позициям a и b в списке формальных параметров, должны быть склеены. Поэтому набор команд склеивания предиката F дополняется командой <d: b>.

Для нехвостовых рекурсивных вызовов регионов склеивания не строится, так как такие вызовы не будут заменяться определениями предикатов.

В подоператорах **параллельного оператора** все результаты различны, а аргументы делятся на уникальные — участвующие только в одном подоператоре, и общие — участвующие более чем в одном подоператоре. Уникальные аргументы можно склеить с результатами, а общие аргументы склеивать нельзя, т.к. они одновременно используются в других подоператорах.

В предикатном программировании возможны две реализации параллельного оператора: через последовательное исполнение и через параллельное. Параллелизм на уровне простых операторов допускает эффективную реализации лишь для архитектуры широких команд Эльбрус-3. Для прочих архитектур параллельный оператор в подавляющем большинстве случаев целесообразно реализовать последовательным исполнением. Алгоритм склеивания зависит от способа реализации параллельного оператора. При параллельной реализации оператора G(d, e: a) || F(d, e: b) склеивание d с a невозможно, так как d может поменять свое значение до того, как будет использована в операторе F; тогда как при последовательной реализации оператор преобразуется в F(d, e: b); G(d, e: a) и конфликта не возникнет.

Регионы для параллельного оператора строятся по регионам его подоператоров. Цель алгоритма построения регионов – стремится найти наибольшее количество склеиваний за приемлемое время. Если регион содержит и уникальные и общие аргументы, то все общие удаляются. Далее для регионов только с общими аргументами выбирается аргумент, встречающийся в наименьшем количестве регионов. Выбираются регионы, содержащие его, и удаляются все, кроме одного. В оставшемся регионе из левой части удаляются все аргументы, кроме данного общего. Таким способом обрабатываются все регионы подоператоров; получившиеся регионы становятся регионами параллельного оператора. Отметим, что, следуя такому алгоритму, мы можем потерять возможные варианты склеивания.

Регионы **условного оператора** строятся из регионов его ветвей. Сначала регионы ветвей корректируются с учетом запретов на склеивание. Если есть регион <…, a, …: b> и запрет на склеивание {a: b}, то аргумент a удаляется из региона.

Пусть есть регионы <…, a, …: b> и <…, a, …: d>, где a — аргумент оператора, a b и d — результаты разных ветвей. Тогда a удаляется из этих регионов, так как нельзя склеить одну переменную с двумя живыми, и создаются запреты на склеивание {a: b} и {a: d}. Они являются живыми, так как являются результатами оператора, следовательно, либо являются результатами предиката, либо используются далее для вычисления других переменных.

Данное правило не относится к регионам с локалами ветвей в правых частях. Если b локал ветви, то оба этих склеивания можно провести, так как локал, в отличие от результата ветвей, не будет использоваться после условного оператора.

Пусть есть регионы <x: z> и <y: z>, где z — результат условного оператора, а x и y — наборы аргументов и локалов. Такие регионы объединяются в один <x, y: z>, который означает, что z можно склеить только с одной из переменных наборов x и y.

Обработанные регионы ветвей становятся регионами условного оператора.

Для **оператора суперпозиции** регионы строятся как совокупность регионов подоператоров.

Запреты на склеивание условного, параллельного оператора и оператора суперпозиции строятся как совокупность запретов подоператоров.

Построение команд склеивания. Команды склеивания для предиката строятся уточнением регионов склеивания тела этого предиката. Общая цель – по региону склеивания <x: y> с несколькими аргументами построить команду склеивания <z: y>, определяющую замену y на z. Сначала регионы уточняются набором *априорных склеиваний* и *запретов*, заданных в исходной программе. Программист может задать в заголовке предиката склеивание результата с аргументом. В этом случае имя результата есть имя аргумента с добавлением штриха в конце (<a’: a>). Команда склеивания (или запрета склеивания) может быть также задана прагмой. Проверяется, что априорные склеивания содержатся в регионах, т. е. для каждого априорного склеивания есть такой регион склеивания, в правой части которого содержится результат априорного склеивания, а одна из переменных левой части — аргумент априорного склеивания. В противном случае выдаются сообщения о некорректном задании априорных склеиваний. Запрет на склеивание — пара переменных, означающая, что вторую переменную нельзя склеить с первой. Проверяется, что данная пара переменных не принадлежит ни одному региону. Если есть такой регион склеивания, в правой части которого содержится вторая переменная запрета склеивания, а одна из переменных левой части региона — первая переменная запрета склеивания, то эта первая переменная удаляется из региона.

Все получившиеся регионы с одним результатом и аргументом считаются командами склеивания. Для регионов с несколькими аргументами произвольным образом выбираются команды склеивания с одним результатом и одним аргументом. Все возможные варианты выбора равноценны, так как склеивание результата с любым из аргументов уберет одно копирование в предикате.

Далее обрабатываются команды, содержащие локалы предиката. Если команда имеет вид <аргумент: локал>, то во все команды, кроме текущей, аргумент подставляется на место локала.

После построения полного набора команд склеивания программа обходится сверху вниз, склеивая переменные исходя из команд склеивания. Там же реализуется замена параллельных операторов, в которых происходит склеивание общих аргументов, операторами последовательного исполнения.

Алгоритм преобразования параллельного оператора. Для параллельного оператора из построенных команд склеивания выбираются те, что содержат его результаты в правых частях. Рассматривается каждая выбранная команда склеивания <x: y>. В подоператорах параллельного оператора аргументы делятся на уникальные — участвующие только в одном подоператоре, и общие — участвующие более чем в одном подоператоре. Если x — уникальный аргумент параллельного оператора, то склеивание можно произвести без конфликтов с другими подоператорами. Если x — общий аргумент параллельного оператора, а y — результат параллельного оператора, то необходимо трансформировать параллельный оператор. Подоператор **B**, в результатах которого содержится y, необходимо удалить из параллельного оператора, заменив последний на **A; B** , где **A** — исходный параллельный оператор без подоператора **B**. В операторе **B** этот аргумент станет уникальным, и с ним можно будет склеить результат.

Пример 1.

Работу алгоритма склеивания переменных покажем на примере предикатной программы нахождения целочисленного квадратного корня:

```
sq1(nat x, k, n : nat m) {
        nat p = n + 2* k + 1;
        if (x < p) m = k else  sq1(x, k + 1, p: m)
}
```

Построение регионов реализуется, начиная с нижних, простых операторов. Для оператора присваивания `m = k` строится регион <k: m>, так как k – неживая переменная, т.е. ее значение не используется после этого оператора. Для хвостового рекурсивного оператора вызова `sq1(x, k + 1, p: m)` регионов при первом просмотре программы не строится. Для условного оператора регионы его ветвей объединяются: <k: m>. Для оператора присваивания `nat p = n + 2* k + 1` регионов не строится, т.к. это не оператор вида a = b, где b - переменная. Тело программы — суперпозиция оператора присваивания с условным оператором. Регионы оператора суперпозиции строятся объединением регионов его

подоператоров. В результате для тела программы и для программы sq1 в целом получим регионы <k: m>. Этот регион является командой склеивания. Далее по этой команде склеивания строятся регионы на хвостовых рекурсивных вызовах. Но для оператора вызова sq1(x, k + 1, p: m) регионов не строится, так как соответствующий аргумент k + 1 — не переменная.

Процесс склеивания реализуется обходом дерева программы. Вхождения в программе переменных из правых частей команд склеивания заменяются соответствующими переменными из левых частей, т.е. переменная переменная m заменяется на k.

Итоговая программа, в которой произведено склеивание <k: m>:

```
sq1(nat x, k, n) {
        nat p = n + 2* k + 1;
        if (x < n) k = k else  sq1(x, k + 1, p: k)
}
```

При упрощениях программы оператор k = k будет удален.

Пример 2.

Предикат sort1 реализует сортировку массива простыми вставками.

```
type T;
nat n;
type natn = 0 .. n;
type Arn = array (natn, T);

pop_into(Arn a, natn k, m, T e: Arn a') {...}

sort1(Arn a, natn m: Arn a') {
      if (m = n) a' = a
      else { T e = a[m+1];
            if (a[m] <= e) sort1(a, m+1: a')
            else { pop_into(a, m+1, m, e: Arn c);
                  sort1(c, m+1: a')
            }
      }
}
```

Построение регионов реализуется, начиная с простых операторов. Для оператора присваивания a' = a строится регион <a: a'>, так как a – неживая переменная. Для оператора присваивания T e = a[m+1] регионов не строится, т.к. это не оператор вида a = b, где b - переменная. Для хвостовых рекурсивных операторов вызова sort1(a, m+1: a') и sort1(c, m+1: a'), регионов на первом просмотре программы не строится. В вызываемом предикате pop_into первый аргумент склеивается с результатом. Поэтому для оператора

вызова pop_into(a, m+1, m, e**:** Arn c) строится регион <a: c>. В операторах суперпозиции и условных операторах регионы подоператоров объединяются. В результате для тела программы и для программы sort1 в целом получим регионы <a: c> и <a: a'>. Эти регионы являются командами склеивания. Команда <a: a'> склеивает параметры предиката и поэтому по ней строятся регионы на хвостовых рекурсивных вызовах. Для оператора вызова sort1(a, m+1**:** a′) строится регион <a: a'>, для оператора вызова sort1(c, m+1**:** a′) строится регион <c: a'>. Оба региона уже присутствуют в командах склеивания.

Итоговая программа, в которой произведены склеивания <a: c> и <a: a'>:

```
sort1(Arn a, natn m: a) {
        if (m = n) a = a
        else { T e = a[m+1];
                if (a[m] <= e) sort1(a, m+1: a)
                else { pop_into(a, m+1, m, e: Arn a);
                        sort1(a, m+1: a)
                }
        }
}
```

8. Упрощения и оформления

В результате склеивания переменных a c b в операторах вида a = b возникает тождественный оператор a = a, который удаляется из программы. Также производятся другие изменения в программе, относящиеся к упрощениям:

- Упрощаются параллельный оператор и оператор суперпозиции, содержащие пустой оператор.
- Упрощаются пустые ветви условного оператора и сам оператор, если обе ветви пусты.
- Упрощается оператор switch, в котором все ветви пусты.
- Замена оператора модификации вида a = a with [k: e] оператором присваивания a[k] = e.
- Удаление оператора перехода на следующий оператор. Реализуется после открытой подстановки вызова гиперфункции.

9. Примеры трансформации программ

Приведем на примерах этапы трансформации программ.

9.1. Программа нахождения целочисленного квадратного корня:

```
sq1(nat x, k, n : nat m) {
        nat p = n + 2* k + 1;
        if (x < p) m = k else  sq1(x, k + 1, p: m)
}
```

Склеивания в sq1: k ← m;

```
sq1(nat x, k, n : nat k) {
        nat p = n + 2* k + 1;
              if (x < p) k = k else  sq1(x, k + 1, p: k)
}
```

Замена хвостовой рекурсии циклом:

```
sq1(nat x, k, n : nat k) {
        M: nat p = n + 2* k + 1;
        if (x < n) k = k else { |x, k, p| = |x, k + 1, n|; goto M }
}
```

Упрощения и оформления:

```
sq1(nat x, k, n : nat k) {
        M: nat p = n + 2* k + 1;
        if (x >= n) { |k, p| = |k + 1, n|; goto M }
}
```

9.2. Программа умножения через сложение:

```
Умн(nat a, b:  nat c) {
         Умн1(a, b, 0: c)
}
Умн1(nat a, b, d:  nat c) {
         if (a = 0)  c = d
         else Умн1(a – 1, b, d + b: c)
}
```

Склеивание в Умн1: c ← d.

```
Умн1(nat a, b, c:  nat c) {
         if (a = 0)  c = c
         else  Умн1(a – 1, b, c + b: c)
}
```

Замена хвостовой рекурсии циклом:

```
Умн1(nat a, b, c:  nat c) {
         M: if (a = 0) c = c
         else {
               |a, b, c| = |a – 1, b, c + b|;
                goto M
         }
}
```

Подстановка определения Умн1 на место вызова в Умн:

```
Умн(nat a, b:  nat c) {
        c = 0;
        M: if (a = 0) c = c
        else {
                |a, b, c| = |a – 1, b, c + b|;
                goto M
        }
}
```

Упрощения и оформления:

```
Умн(nat a, b:  nat c) {
        c = 0;
        M: if (a != 0) {
                |a, c| = |a – 1, c + b|;
                goto M
        }
}
```

9.3. Программа сортировки простыми вставками:

```
type T; // произвольный тип с линейным  порядком
nat n;  // n – 1 – число элементов сортируемого массива
type natn = 0 .. n;
type Arn = array (natn, T);
sort(Arn a: Arn a') { sort1(a, 0: a') }
```

В предикате sort1 предполагается, что первые m элементов массива a – отсортированы. Итоговый массив a′ полностью отсортирован.

```
sort1(Arn a, natn m: Arn a') {
      if (m = n) a' = a
      else { T e = a[m+1];
                if (a[m] <= e) sort1(a, m+1: a')
                else { pop_into(a, m+1, m, e: Arn c);
                sort1(c, m+1: a')
                }
      }
}
```

Предикат pop_into вставляет элемент e, который изначально был элементом a[m]. При этом элементы от a[k] до a[m-1] сдвинуты на одну позицию вправо и все они больше e. При этом в позиции k – «дыра». Итоговый массив a′ – отсортирован.

```
pop_into(Arn a, natn k, m, T e: Arn a') {
      Arn b = a with [k: a[k-1]];
      if (k = 1) a' = b with [0: e]
      else if (b[k-2] <= e) a' = b with [k-1: e]
            else pop_into(b, k-1, m, e: a')
}
```

Склеивания в sort: a ← a'

```
sort(Arn a: a) { sort1(a, 0: a) };
```

Склеивания в sort1: a ← a', c

```
sort1(Arn a, natn m: a) {
      if (m = n) a = a
      else { T e = a[m+1];
            if (a[m] <= e) sort1(a, m+1: a)
            else { pop_into(a, m+1, m, e: Arn a);
            sort1(a, m+1: a)
            }
      }
}
```

Склеивания в pop_into: a ← a', b

```
pop_into(Arn a, natn k, m, T e: a) {
      a = a with [k: a[k-1]];
      if (k = 1) a = a with [0: e]
            else if (a[k-2] <= e) a = a with [k-1: e]
            else pop_into(a, k-1, m, e: a)
};
```

Замена хвостовой рекурсии циклом в sort1 и pop_into:

```
sort1(Arn a, natn m: a) {
       M: if (m = n) a = a
       else { T e = a[m+1];
            if (a[m] <= e) { |a, m| = |a, m+1|; goto M; }
            else { pop_into(a, m+1, m, e: Arn a);
                  |a, m| = |a, m+1|; goto M;
            }
       }
}
pop_into(Arn a, natn k, m, T e: a) {
       M: a = a with [k: a[k-1]];
       if (k = 1) a = a with [0: e]
            else if (a[k-2] <= e) a = a with [k-1: e]
                  else { |a, k, m| = |a, k - 1, m|; goto M; }
};
```

Подстановка определения pop_into на место вызова в sort1:

```
sort1(Arn a, natn m: a) {
        M: if (m = n) a = a
        else { T e = a[m+1];
                if (a[m] <= e) { |a, m| = |a, m+1|; goto M; }
                else {
                        natn k =  m+1;
                        M1: a = a with [k: a[k-1]];
                        if (k = 1) a = a with [0: e]
                        else if (a[k-2] <= e) a = a with [k-1: e]
                                else { |a, k, m| = |a, k - 1, m|; goto M1; }
                        |a, m| = |a, m+1|; goto M;
                }
        }
}
```

Подстановка определения sort1 на место вызова в sort:

```
sort(Arn a: a) {
        natn m = 0;
        M: if (m = n) a = a
        else { T e = a[m+1];
                if (a[m] <= e) { |a, m| = |a, m+1|; goto M; }
                else {
                        natn k =  m+1;
                        M1: a = a with [k: a[k-1]];
                        if (k = 1) a = a with [0: e]
                            else if (a[k-2] <= e) a = a with [k-1: e]
                                    else { |a, k, m| = |a, k - 1, m|; goto M1; }
                        |a, m| = |a, m+1|; goto M;
                }
        }
 };
```

Упрощения и оформления:

```
sort(Arn a: a) {
        natn m = 0;
        M: if (m != n) {
                T e = a[m+1];
                if (a[m] <= e) { m = m+1; goto M; }
                else {
                        natn k = m+1;
                        M1: a[k] = a[k-1];
                        if (k = 1) a[0] = e
                        else if (a[k-2] <= e) a[k-1] = e
                                else { k = k - 1; goto M1; }
                        m = m+1; goto M;
                }
        }
```

};

При завершении оптимизации и генерации программы на C++ реализуются косметические преобразования вставки циклов **while** и **for** вместо операторов перехода.

10. Обзор работ

Наиболее известным проектом в области трансформационного программирования является проект CIP [15-17], реализованный в Техническом университете Мюнхена. В проекте CIP определяется язык CIP-L, называемый также The wide spectrum language 84, содержащий следующие подъязыки:

- язык спецификаций, аналогичный языку исчисления предикатов;
- аппликативный язык – язык функционального программирования;
- диалекты императивных языков Pascal и Algol**.**

Различные уровни языка CIP-L интегрируются формальным описанием трансформационной семантики. Трансформационное правило для каждой языковой конструкции определяет построение математической функции из функций для подконструкций данной конструкции. Построение программы на языке CIP-L реализуется применением набора трансформаций для начальной более простой версии программы. Трансформации реализуются в автоматическом режиме с проверкой условий их корректности. Аппарат трансформаций является универсальным в CIP. Трансформации могут применяться для доказательства утверждений, для преобразования программы в рамках аппликативного языка, для преобразования с аппликативного языка на императивный, а также для оптимизации программы на императивном языке. Имеются трансформации линейной рекурсии в хвостовую, а хвостовой – в цикл типа **while**. Отметим, что наши трансформации склеивания переменных и кодирования алгебраических типов значительно сложнее определяемых в проекте CIP. Они определяются в рамках формальной операционной семантики, а не денотационной.

В функциональном программировании оптимизация программы полностью возлагается на транслятор. Определенный прогресс в эффективности достигается разработкой сверхмощных интерпретаторов и генераторов кода для языка SequenceL [18], определяющего предельно декларативный стиль программирования. Функциональное программирование существенно уступает в эффективности, поскольку невозможно автоматически воспроизвести серию оптимизаций, совершаемых вручную в предикатном программировании.

Замена хвостовой рекурсии циклом и подстановка тела программы на место вызова – типичные, хорошо известные оптимизирующие преобразования. Замена хвостовой рекурсии циклом проводится лишь для функциональных языков. Трансформация склеивания переменных реализуется только для предикатного программирования. Склеиваются аргумент и результат некоторого фрагмента программы, включая также используемые промежуточные переменные между аргументом и результатом. Стиль программирования, при котором такое склеивание подразумевается, характерен лишь для предикатного программирования. В императивном программировании склеивание проводит программист. В функциональном программировании склеивание возможно лишь для конструкций типа **let** и **where**, однако в публикациях по функциональным языкам подобной трансформации не обнаружено.

Алгоритм склеивания переменных разработан Э. Петровым [8] в 2003г. в рамках первой версии системы предикатного программирования. Склеивание переменных проводилось на базе кандидатных множеств, по структуре отличных от регионов склеивания. Алгоритм Э. Петрова существенно сложнее нашего. Для параллельного оператора алгоритм имеет временную сложность O(n!), тогда как наш алгоритм – $O(n^2)$, однако последний может пропустить некоторые варианты склеивания.

Термин «склеивание переменных» впервые появился в рамках задачи экономии памяти в классических работах А. П. Ершова, С. С. Лаврова, В. В. Мартынюка. Склеивание переменных определялось как выбор переобозначения аргументов и результатов из заданного множества корректных переобозначений, которое позволяет в наибольшей степени уменьшить объем необходимой памяти [4]. В нашем подходе, в отличие от задачи экономии памяти, склеиванию подлежат только те переменные одного типа, между которыми имеется информационная связь.

Оптимизация памяти для современных компьютеров актуальна при распределении регистров. В этой задаче большое число используемых переменных необходимо разместить на небольшом количестве регистров. Используется метод раскраски графа несовместимостей [19], по цветам распределяя регистры для хранения переменных.

11. Заключение

В настоящей работе описывается оптимизация предикатных программ с трансляцией на язык C++, названная оптимизацией среднего уровня, существенно отличающаяся от классической оптимизации императивных программ. Оптимизация реализует следующий

набор трансформаций: склеивание переменных, замену хвостовой рекурсии циклом, подстановку определения предиката на место вызова, упрощения и оформления.

В целях оптимизации проводится потоковый анализ предикатной программы. Строится граф вызовов программы с использованием достаточно точной аппроксимации значений переменных предикатного типа. Определяются аргументы и результаты операторов программы и области жизни переменных.

Описываемые оптимизирующие трансформации иллюстрируются на различных предикатных программах в работах [3, 6, 9, 10, 12, 20]. Трансформации частично реализованы в рамках экспериментальной системы предикатного программирования. Трансформация кодирования алгебраических типов (списков, строк и деревьев) через массивы и указатели описана в работах [2, 11, 14].

В дальнейшем планируется разработка трансформаций для автоматического распараллеливания программ. Планируется также расширить возможности склеивания переменных реализацией склеивания переменных с отдельными компонентами структур.

Работа выполнена при поддержке РФФИ, грант № 16-01-00498.

Список литературы

1. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман Компиляторы: принципы, технологии и инструментарий = Compilers: Principles, Techniques, and Tools. — 2 изд. — М.:Вильямс, 2008. — ISBN 978-5-8459-1349-4
2. Булгаков К.В., Каблуков И.В., Тумуров Э.Г., Шелехов В.И. Оптимизирующие трансформации списков и деревьев в системе предикатного программирования // Системная информатика, № 9. — Новосибирск, 2017. — С. 63-92. [Электронный ресурс]. URL: http://www.system-informatics.ru/files/article/105.pdf
3. В.А. Вшивков, Т.В. Маркелова, В.И. Шелехов. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т.4(33), с. 79-94, 2008.
4. Ершов А.П. Введение в теоретическое программирование. М.: Наука, 1977. 288с.
5. Каблуков И. В., Шелехов В.И. Реализация склеивания переменных в предикатной программе. – Новосибирск, 2012. – 6с. – (Препр. / ИСИ СО РАН; N 167).
6. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Версия 0.12 — Новосибирск, 2013. — 52с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
7. Касьянов В. Н., Евстигнеев В. А. - Графы в программировании: обработка, визуализация и применение – БХВ – Петербург, 2003 – ISBN 5-94157-184-4

8. Петров Э.Ю. Склеивание переменных в предикатной программе // Методы предикатного программирования. Новосибирск, 2003. С. 48-61.
9. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.
10. Шелехов В.И. Доказательное построение, верификация и синтез предикатных программ // Знания-Онтологии-Теории (ЗОНТ-2017), Том 2. — Институт Математики СО РАН, Новосибирск, 2017. — С. 156-165. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/lbase.pdf
11. Шелехов В.И. Предикатная программа вставки в АВЛ-дерево // Системная информатика, № 9. — Новосибирск, 2017. — С. 23-42. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/avl_insert.pdf
12. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. – Новосибирск, 2012. – 30с. – (Препр. / ИСИ СО РАН. N 164).
13. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск, 2015. 13с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf
14. Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, №3, 2014. ИСИ СО РАН, Новосибирск. С. 25-43. [Электронный ресурс] URL: http://persons.iis.nsk.su/files/persons/pages/String.pdf
15. Bauer F.L., Broy M., et.al. The Wide Spectrum Language 84 / Inst. For Informatik. Technische Universitat Munchen, 1983. – 157p.
16. Bauer F.L., Broy M., et.al. Wide Spectrum Language for Program Specification and Development (Tentative Version). – Munchen, 1981. – 236p. – (Prepr: Inst. For Informatik / Technische Universitat Munchen / TUM-18104)
17. Brass B., Erhard F., Horsch A., Riethmayer H.-O., Steinbruggen R. CIP-S: An instrument for program transformation and rule generation. – Munchen, 1982. – P. 44-62. – (Prepr: Inst. For Informatik / Technische Universitat Munchen / TUM-18211)
18. Cooke D. E., Rushton J. N. Taking Parnas's Principles to the Next Level: Declarative Language Design // Computer, Vol. 42, no. 9. 2009. P. 56-63.
19. George, Lal; Appel, Andrew W. (May 1996). "Iterated Register Coalescing". ACM Trans. Program. Lang. Syst. 18 (3): 300–324 doi:10.1145/229542.229546. ISSN 0164-0925.
20. Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements. Automatic Control and Computer Sciences. Vol. 45, No. 7, 421–427.