Реализация оптимизирующих трансформаций предикатных программ

1. ВВЕДЕНИЕ

Типичный способ реализации языка программирования заключается в реализации транслятора с языка программирования на язык команд целевой ЭВМ или на некоторый другой реализованный язык программирования, а также реализации поддержки исполнения. Применение оптимизации в процессе трансляции программ для императивных языков позволяет получать вполне эффективные программы. Однако для функциональных языков не удается достичь приемлемой эффективности даже с помощью изощренной оптимизации. Подобная перспектива неэффективности оттранслированной программы ожидает также и реализацию языка предикатного программирования Р методом прямой автоматической трансляции.

Для языка Р разрабатывается метод трансляции с применением последовательности трансформаций, преобразующих предикатную программу в эффективную императивную программу на императивном расширении языка Р, который добавляет в язык циклы **while** и **for**, и оператор выхода из цикла **break.** Базовыми *трансформациями* являются:

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

Для всех определений предикатов сначала применяется замена хвостовой рекурсии циклом. Тела предикатов с устраненной рекурсией подставляются на место вызовов. Затем производится склеивание переменных. После этого можно провести упрощающие преобразования, в частности константные вычисления. Наконец, структурные объекты (списки, деревья и др.) кодируются посредством массивов и указателей. С помощью данных трансформаций можно получить программу предельной эффективности, однако трудно это сделать автоматически. На втором этапе полученная императивная программа конвертируется на любой из императивных языков: С, С++, ФОРТРАН и др. В настоящее время при программировании на языке Р применяется метод ручной трансформации предикатной программы.

Трансформация замены хвостовой рекурсии циклом является весьма эффективной, поскольку открывает возможность применения серии других оптимизирующих преобразований. Однако рекурсия в наиболее простом решении задачи обычно не является хвостовой. Существует ряд подходов по приведению рекурсии к хвостовому виду, как правило, автоматическими преобразованиями. В технологии предикатного программирования предлагается универсальный метод обобщения исходной задачи для получения решения с хвостовой формой рекурсии. Этот метод нередко оказывается ключевым для эффективной реализации предикатных программ.

В программировании регулярно возникают ситуации, когда традиционными конструкциями языка программирования невозможно адекватно представить некоторый фрагмент алгоритма. Предлагаемый аппарат гиперфункций позволяет гибко декомпозировать программу произвольным образом.

Императивное расширение языка Р определяет дополнительные языковые конструкции, возникающие в программе в результате проведения трансформаций предикатной программы. Использование этих конструкций в исходной программе недопустимо.

2. ПОТОКОВЫЙ АНАЛИЗ

Для проведения трансформаций подстановки предиката на место вызова и склеивания переменных требуется построение графа вызовов программы и разделение программы на слои: слой первого уровня содержит программы, не вызывающие других программ, слой второго уровня содержит программы, вызывающие только программы из слоя первого уровня и т.д. Также существуют рекурсивные слои – набор программ, каждая из которых может быть вызвана по цепочке вызовов программ из этого слоя, начиная с вызова внутри себя. Рекурсивный вызов – это вызов программы из того же рекурсивного слоя, в котором находится вызывающая программа. Для таких вызовов в случае, когда программ в рекурсивном слое больше одной, нужно отдельное рассмотрение. Проведение трансформаций упорядочено по уровням слоев, начиная с первого.

Построение графа вызовов.

Полная программа - это набор определений предикатов. *Граф вызовов программы* - это ориентированный граф. Вершины графа - предикаты. Вызов предиката А в теле предиката В определяет дугу в графе: В−>А. При

наличии вызова предиката A(...) мы говорим, что предикат A используется в *позиции вызова предиката.*

Всякий предикат имеет тип. По набору предикатов программы определяется список предикатных типов (СПТ). Для каждого типа из СПТ строится множество *заместителей* данного типа - это набор предикатов этого типа, встречающихся в позициях, отличных от вызовов предикатов. Среди таких позиций - вхождение имени предиката в качестве аргумента некоторого вызова. Такие предикаты могут стать заместителями, т.е. значениями переменных и выражений предикатного типа.

Допустим, в теле предиката B имеется вызов предиката в виде e(...), где e - выражение предикатного типа, причем e - не имя предиката. Тогда в граф вызовов добавляются дуги вида $B{-}{>}A_i$, где A_i - заместитель для типа выражения e. Если же e - имя предиката, то в граф вызовов добавляется только дуга $B{-}{>}e$.

На первом проходе по программе строятся заместители предикатных типов. На втором проходе строится собственно граф вызовов.

Компоненты сильной связности находятся по алгоритму Тарьяна, который имеет линейную сложность. После преобразования каждой компоненты в одну вершину граф становиться ациклическим и его можно упорядочить в определенном ранее порядке.

Также для трансформации склеивания переменных необходимо опередение аргументов, результатов и пост-аргументов операторов в телах предикатов. Пост-аргументами оператора являются переменные предиката, используемые после завершения исполнения текущего оператора.

Построение пост-аргументов и результатов.

Для произвольного оператора D определим рекурсивную программу $DF(D, ArgP_D{:}\ Arg_D, R_D)$, вычисляющую аргументы Arg_D и результаты R_D оператора D, а также аргументы, результаты и пост-аргументы для всех вложенных операторов; $ArgP_D$ обозначает ранее вычисленные пост-аргументы оператора D.

Для оператора суперпозиции A = B; C программа $DF(A, ArgP_A{:}\ Arg_A, R_A)$ представляется следующей последовательностью операторов:

$ArgP_C = ArgP_A$;
$DF(C, ArgP_C{:}\ Arg_C)$
$ArgP_B = ArgP_A \cup Arg_C$;
$DF(B, ArgP_B{:}\ Arg_B)$
$Arg_A = Arg_B \cup Arg_C$;
$R_A = \{R_B \setminus Loc_B\} \cup R_C$;

Для условного оператора A = **if** (E) B **else** C и параллельного оператора A = B || C программа $DF(A, ArgP_A{:}\ Arg_A, R_A)$ представляется следующей последовательностью операторов:

$ArgP_B = ArgP_A$;
$ArgP_C = ArgP_A$;
$DF(B, ArgP_B{:}\ Arg_B)$
$DF(C, ArgP_C{:}\ Arg_C)$
$Arg_A = Arg_B \cup Arg_C$;
$R_A = R_B \cup R_C$;

3. ПОДСТАНОВКА ОПРЕДЕЛЕНИЯ ПРЕДИКАТА НА МЕСТО ВЫЗОВА

Пусть A(x: y) { S } – определение предиката на императивном расширении языка P, а A(e: z) – вызов предиката в теле некоторого другого предиката B. Здесь x, y, z обозначают списки переменных, а e – список выражений. *Подстановка определения предиката на место вызова* A(e: z) есть замена вызова следующей композицией:

| x | = | e |; { S }; | z | = | y | . (3.1)

Конструкции | x |, | z | и | y | являются мультипеременными, а | e | – мультивыражением, которые определяются таким образом:

МУЛЬТИПЕРЕМЕННАЯ ::= **|** СПИСОК-ПЕРЕМЕННЫХ **|** | СПИСОК-ПЕРЕМЕННЫХ

МУЛЬТИВЫРАЖЕНИЕ ::= | СПИСОК-ВЫРАЖЕНИЙ | | СПИСОК-ВЫРАЖЕНИЙ

В результате подстановки переменные наборов x и y становятся локальными переменными определения предиката B, в котором находился вызов A(e: z). В результате подстановки не должно возникать коллизий с именами переменных предиката B. В общем случае проведению подстановки предшествует предварительное систематическое переименование переменных. Нетрудно показать, что исполнение последовательности (3.1) эквивалентно исполнению вызова A(e: z) в соответствии с операционной семантикой ([1] 4.14) и семантикой выражений как аргументов вызова, определенной в [1] 5.4.

Оператор присваивания вида | x | = | e | называется *групповым оператором присваивания*. Это преобразование называется *раскрытием* группового оператора 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.

Пусть набор x состоит из переменных $x_1, x_2, \ldots, x_n$; а e определяет набор выражений $e_1, e_2, \ldots, e_n$. Допустим, e_j является переменной для некоторого j. В соответствии с соглашением об отсутствии коллизий переменная e_j должна быть отлична от x_j. В подавляющем большинстве случаев замена x_j на e_j дает эквивалентную программу. Данная замена (склеивание переменной x_j с e_j) уменьшает число переменных на единицу и позволяет удалить копирование $x_j = e_j$ в составе группового оператора | x | = | e |. Склеивание переменных x_j и e_j корректно за исключением случая, когда x_j перевычисляется внутри S (что возможно как результат склеивания x_j с другими переменными (см. главу. 5)), а переменная e_j используется не только в вызове A(e: z), но и после него.

Аналогичным образом проводится склеивание результирующих переменных набора y с переменными набора z. Поскольку переменные, входящие в набор z и набор y, должны быть различными, то склеивание результирующих переменных всегда возможно. При склеивании результатов устраняется групповой оператор | z | = | y |. Отмеченные склеивания (замену x_j на e_j и y_j на z_j) можно не проводить, если склеиваемые переменные обозначены одинаковыми идентификаторами в исходной программе. Поэтому естественной является следующая стратегия именования в процессе программирования на языке P. Если имеется вызов A(e: z) и требуется запрограммировать определение предиката A, то в качестве результирующих параметров y в определении A выбираются переменные набора z. Переменные в составе набора e становятся соответствующими аргументами определения A с учетом ограничений, указанных выше.

4. ЗАМЕНА ХВОСТОВОЙ РЕКУРСИИ ЦИКЛОМ

Подстановка определения нерекурсивного предиката на место вызова является эффективной при наличии в программе одного вызова. Подстановка определения рекурсивного предиката усложняет программу и поэтому сомнительна, хотя иногда может принести пользу; в любом случае такое преобразование считается экзотическим. Для рекурсивно определяемого предиката эффективным является другое преобразование − замена хвостовой рекурсии циклом. Тем не менее, оказывается, это преобразование является специальным случаем подстановки определения предиката на место вызова.

Существует специальный случай рекурсии, называемый *хвостовой рекурсией* (tail-рекурсией в языке Лисп), 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 результатами определяемого предиката.

Через last(S) обозначим множество последних исполняемых конструкций в операторе S. Тогда: 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) }
```

Допустим, имеется рекурсивное определение предиката A(x: y) { S } и вызов A(e: y) с хвостовой рекурсией внутри оператора S. Подставим определение предиката A на место вызова A(e: y). В соответствии с изложенным в конце главы 3 результатом подстановки определения предиката A является | x | = | e |; { S }. Обозначим через S′ оператор, полученный заменой в S подстановкой определения на место вызова A(e: y). 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*.

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. А после проведения подстановки вызова A становится возможной серия других преобразований.

Применение трансформации замены хвостовой рекурсии циклом покажем для программы вычисления наибольшего общего делителя. 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 **for**. Получим:

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

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

5. СКЛЕИВАНИЕ ПЕРЕМЕННЫХ

Склеивание переменных [3] - это замена (сохраняющая эквивалентность) в тексте программы всех вхождений одной переменной на существующую другую. Значительный эффект достигается при склеивании структурных переменных, таких как массивы и списки, поскольку склеивание обычно позволяет избежать копирования структур.

В отличие от задачи экономии памяти [5], склеиванию в программе подлежат результаты с аргументами, аргументы с локалами и локалы с результатами, причем типы склеиваемых переменных должны совпадать. Набор склеиваний может быть частично задан пользователем. Необходимо проверить его корректность и дополнить. 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. Типы склеиваемых переменных должны совпадать. В дополнение к этому переменная параметрического типа T(n+1) может быть склеена с переменной типа T(n) в случае возрастания n на 1.

Общая схема реализации

По слоям программы строится порядок предикатов программы, в котором необходимо производить трансформацию. В каждом предикате строятся регионы склеивания для каждого оператора. Склеивание переменных реализуется в оптимизирующем трансформаторе с помощью алгоритмов уточнения регионов.

Регион склеивания для оператора G есть набор аргументов и результатов одного типа <x: y>, где x - список аргументов и y - один из результатов оператора. **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.

Рекурсивный вызов - это вызов программы из той же компоненты сильной связности, в которой находится вызывающая программа. Построение регионов упорядочено по уровням слоев, начиная с первого, и реализуется обходами дерева программы сверху вниз и снизу вверх.

Для вызова функции, у которой результат склеивается с одним из аргументов, оператор, включающий вызов, при анализе представляется в виде оператора суперпозиции. Например, x = 1+foo(b) заменяется на foo(b: t); x = 1+t. Здесь t склеивается с b, x склеивается с t, и в итоге x склеивается с b.

Построение регионов склеивания

Для построения регионов склеивания программа обходится снизу вверх. В дереве предикатной программы нижними операторами являются операторы присваивания и вызовов предикатов. Для этих операторов строятся регионы склеивания определенным ниже образом. После обходятся объемлющие операторы суперпозиции, параллельные и условные операторы, регионы которых строятся на основе уже определенных регионов подоператоров.

Для оператора **присваивания** регион строится с набором его аргументов в левой части и переменной в правой части.

Предполагается, что для предиката, вызываемого в операторе **вызова** предиката, уже построены итоговые регионы склеивания результатов с аргументами. Подставляя соответствующие аргументы и результаты вызова в эти регионы, получаются регионы данного оператора вызова.

В рекурсивном предикате для хвостового рекурсивного вызова регионов не строится, т. к. при замене хвостовой рекурсии вызов заменяется мультиприсваиванием аргументам предиката новых значений, и поэтому соответствующий набор регионов пуст.

В случае рекурсивного кольца с числом предикатов больше одного регионы для рекурсивных вызовов строятся с использованием лишь априорных склеиваний вызываемых предикатов. Фактически это означает требование к пользователю явным образом задавать все склеивания результатов с аргументами для таких предикатов.

Пример

Foo (**int** a, b**: int** a’, b’)

{ ... }

В соответствии с правилами языка P использование имен a’ и b’ означает, что переменная a’ должна быть склеена с a, а переменная b’ - с b. Подстановка тела предиката Foo на место вызова Foo (p1+p2, p3+p4**:** r1, r2) с раскрытием мультиприсваиваний дает последовательность: a = p1+p2; b = p3+p4; {…} r1 = a; r2 = b. Переменные a и b становятся локалами. Строятся следующие регионы: <p1, p2**:** a>, <p3, p4**:** b>, <a**:** r1> и <b**:** r2>.

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

Регионы для параллельного оператора строятся по регионам его подоператоров. Алгоритм построения следующий. 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.

В **условном операторе,** если есть регионы с разными результатами и одинаковым аргументом, то этот аргумент удаляется из этих регионов. Данное правило не относится к регионам с локалами ветвей в правых частях. Например, у одной ветви есть регион <a: x>, у другой - <a: d>, где a - аргумент условного оператора, x - результат, а d - локал второй ветви. Оба этих склеивания можно провести, т. к. локал, в отличие от результата ветвей, не будет использоваться после условного оператора, поэтому две переменные с одинаковым именем не будут использоваться одновременно. Оставшиеся регионы с одинаковым результатами объединяются, образуя один регион с данным результатом в правой части и объединением аргументов в левой.

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

Построение команд склеивания

Команды склеивания строятся в оптимизирующем трансформаторе уточнением регионов склеивания. Сначала регионы уточняются набором *априорных склеиваний* и *запретов*, заданных в исходной программе. Пользователь может задать в заголовке предиката склеивание результата с аргументом. В этом случае имя результата есть имя аргумента с добавлением штриха в конце. 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 и одним или несколькими результатами произвольным образом выбираются команды склеивания с одним результатом и одним аргументом.

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 регионов склеивания. Замена параллельных операторов на последовательные дает дополнительные возможности для склеивания.

Алгоритм. Для параллельного оператора из построенных команд склеивания выбираются те, что содержат его результаты в правых частях.

Рассматривается каждая выбранная команда склеивания. Если слева - уникальный аргумент параллельного оператора, то склеивание можно провести без конфликтов с другими подоператорами. Если слева - общий аргумент параллельного оператора, а справа - результат параллельного оператора, то необходимо трансформировать параллельный оператор. Выбираются все подоператоры с этим аргументом, кроме того, в результатах которого содержится результат из этой команды, и исходный параллельный оператор заменяется суперпозицией **A;B** , где **A** - параллельный оператор из выбранных подоператоров, а **B** - исходный параллельный оператор без выбранных подоператоров. В операторе **B** этот аргумент станет уникальным, и с ним можно будет склеить результат.

Пример

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 простых операторов. Для операторов присваивания строятся регионы <n: p> и <k: m>. В первом операторе k - пост-аргумент, поэтому с ним склеивать нельзя, и он не может находиться в регионе. Для рекурсивного вызова регионы не строятся. Далее на условном операторе объединяются регионы его ветвей. В результате единственным регионом условного оператора станет <k: m>. Тело программы - оператор суперпозиции 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 <n: p> и <k: m>. Эти регионы являются итоговыми командами склеивания. Отметим, что предварительно надо было бы заменить вхождения локала p в других командах, однако таковые отсутствуют.

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

Итоговая программа, в которой произведены склеивания <n: p> и <k: m>:

```
sq1(nat x, k, n)
{       n = n + 2* k + 1;
```

if (x < n) k = k **else** sq1(x, k + 1, n**:** k)

}

В дальнейшем оператор k = k удаляется из программы.

6. ТРАНСФОРМАЦИИ УПРОЩЕНИЯ ПРОГРАММЫ

После проведения трансформации склеивания переменных необходимо провести некоторые упрощения программы. Удаляются тождественные присваивания вида a = a. Также, если в условном операторе в одной из ветвей не осталось операторов, то надо заменить условный оператор на короткий условный оператор с одной ветвью. Если в обеих ветвях не осталось операторов, то такой условный оператор удаляется. Также удаляются опустевшие подоператоры параллельного оператора и оператора мультиприсваивания. Удаляются неиспользуемые вычисления (присваивания переменным, далее не использующиеся в программе).

7. ЛИТЕРАТУРА

1. Шелехов В.И. Введение в предикатное программирование. - Новосибирск, 2002. - 82с. - (Препр. / ИСИ СО РАН; N 100).

2. Шелехов В.И. Язык предикатного программирования Р. - Новосибирск, 2002. - 40с. - (Препр. / ИСИ СО РАН; N 101).

3. Каблуков И. В., Шелехов В.И. Реализация склеивания переменных в предикатной программе. - Новосибирск, 2012. - 6с. - (Препр. / ИСИ СО РАН; N 167).

4. Петров Э.Ю. Склеивание переменных в предикатной программе // Методы предикатного программирования. Новосибирск, 2003. С. 48-61.

5. Ершов А.П. Введение в теоретическое программирование. М.: Наука, 1977. 288с.