ISBN 5-94052-066-0 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ. ПЕРЕСЛАВЛЬ-ЗАЛЕССКИЙ, 2004

УДК 519.68

С. М. Абрамов, А. Ю. Орлов

Компиляция в императивные языки синтаксического отождествления языка Рефал

Аннотация. Синтаксическое отождествление (или сопоставление с образцом) является одним из основных изобразительных средств языка Рефал. Данная работа посвящена новому подходу к компиляции синтаксического отождествления в императивные языки. Подход позволяет во время подбора значений рефал-переменных эффективно (целыми областями) отсечь варианты, на которых заведомо не может быть положительного результата отождествления. Описываемые в данной работе оптимизации достаточно очевидны и, при этом, они дают существенный выигрыш в эффективности. Однако до сих пор они не использовались в реализациях языка Рефал.

От читателя ожидается знакомство с языком Рефал, в работе обсуждаются самые базовые понятия языка, общие для всех диалектов Рефала [1, 2, 5, 10, 11].

Ключевые слова и фразы: Рефал, Рефал Плюс, синтаксическое отождествление, сопоставление с образцом, компиляция в императивный язык.

1. Введение

Язык программирования Рефал разработан В. Ф. Турчиным в шестидесятые годы прошлого столетия как функциональный язык, ориентированный на задачи, связанные с преобразованием символьной информации (symbol manipulation). Одним из базовых понятий языка Рефал является синтаксическое отождествление, сопоставление с образцом. В данной работе излагается новый подход к компиляции синтаксического отождествления в императивные языки, основанный на учете соотношений между длинами выражений и переменных, участвующих в синтаксическом отождествлении.

Новый подход к эффективной реализации синтаксического отождествления будет проиллюстрирован на примерах (раздел 1.1), затем обсужден более детально.

Работа выполнена при поддержке Суперкомпьютерной программы «СКИФ» Союзного государства и при поддержке РФФИ, проект № 02-07-90126.

В статье будет предполагаться, что читатель знаком с языком программирования Рефал. При этом обсуждаться будут самые базовые понятия, общие для всех диалектов [1, 2, 5, 10, 11] Рефала.

В статье предполагается, что в реализации Рефала используется такое представление объектных выражений, что операции вычисления длины ($|e_x|$) выражения и конструирования подвыражения ($e_x\langle i, L\rangle$)[1] выполняются за константное (не зависящее от выражения) время. Это предположение выполнено для большинства современных реализаций диалектов языка Рефал [2, 7, 9, 11].

1.1. Иллюстрация идеи на примерах.

1.1.1. *Пример на тему «кратные длины».* Рассмотрим простую функцию на языке Рефал[2]:

```
IsTriplet e_x = {
    e_x : e_1 e_1 e_1        = True;
    /* otherwise */ = False;
};
```

Во второй строке данного описания стоит *перестройка* $e_x : e_1\, e_1\, e_1$, выполняющая следующую проверку: «верно ли, что аргумент функции e_x может быть представлен как конкатенация трех копий одного и того же выражения e_1?». Если это верно, то в качестве результата вычисления функции возвращается `True`, если не верно — то `False`. В соответствии с данным описанием:

(1) Если вызвать данную функцию, передав ей в качестве аргумента e_x = `'AAA...AAA'` строку длиной $51\,000 = 3 \cdot 17\,000$ символов, содержащую только литеры `'A'`, то в качестве результата функция вернет `True`, так как в данном случае строка e_x может быть представлена как конкатенация из трех копий строки e_1 = `'AAA...AAA'` длиной 17 000 символов, содержащей только литеры `'A'`.

---

[1] В конструкции $e_y = e_x\langle i, L\rangle$ фрагменты i и L могут быть выражениями, вырабатывающими неотрицательные целые значения — номер терма в e_x, с которого будет начинаться подвыражение e_y, и длину подвыражения e_y, соответственно. Термы нумеруются слева направо начиная с 0, номер последнего терма — $(|e_x|-1)$.

[2] Приведена запись функции на языке Рефал Плюс. Аналогичный пример может быть записан на любом ином диалекте языка Рефал, причем запись будет весьма близка к приведенному тексту.

	Refal 6	Refal 6/Java	Refal Plus (32)	Refal Plus (16)	Refal 5 (A)	Refal 5 (B)
t_{True}	2.8	5.0	17.8	59.4	72.7	74.7
t_{False}	8.7	11.2	33.8	133.1	228.9	230.7
$\frac{t_{False}}{t_{True}}$	3.1	2.2	1.9	2.2	3.1	3.1

t_{True} — время (сек.) выполнения вычисления:

`IsTriplet` '$\underbrace{\texttt{AAA}\ldots\texttt{AAA}}_{51\,000}$' = True

t_{False} — время (сек.) выполнения вычисления:

`IsTriplet` '$\underbrace{\texttt{AAA}\ldots\texttt{AAA}}_{50\,999}$' = False

Вычисления выполнялись на ПЭВМ с процессором Intel Pentium IV 2GHz.

Таблица 1. Предыдущие реализации языка рефал: время вычисления функции `IsTriplet`

(2) Если вызвать данную функцию, передав ей в качестве аргумента e_x = `'AAA...AAA'` строку длиной $50\,999 = 3 \cdot 17\,000 - 1$ символов, содержащую только литеры `'A'`, то в качестве результата функция вернет `False`, так как в данном случае ни для каких значений e_1 строка e_x не может быть представлена как конкатенация из трех копий e_1.

Во всех существовавших ранее реализациях диалектов Рефала вычисление такой несложной функции (всего один шаг «проверка–замена») даже на современных ПЭВМ занимает достаточно длительное время (см. Таблицу 1). Причиной этого является неэффективная схема (см. Рис. 1) реализации перестройки $e_x : e_1\, e_1\, e_1$ в функции `IsTriplet`: при такой схеме реализации сложность вычисления функции составляет $O({A_x}^2)$, где $A_x = |e_x|$ — длина входных данных.

Суть реализации данной перестройки в следующем: *в качестве возможных вариантов для значения переменной* e_1 рассматриваются *все возможные префиксы* длины X_1, префиксы рассматриваются поочередно, в следующем порядке: $X_1 = 0, \ldots, A_x$. Для каждого значения e_1 делается проверка, не совпадает ли e_x с конкатенацией $e_1\, e_1\, e_1$ трех копий e_1.

Корни неэффективности в том, что в качестве возможных значений для неизвестной e_1 перебираются *все* префиксы e_x. А этого

```
function IsTriplet(expr e_x) {
  expr e_1
  int X_1
  /* цикл удлинения e_1, перебор вариантов */
  for (X_1 = 0; X_1 ⩽ |e_x|; X_1++) {
    /* в качестве e_1 берем префикс длины X_1 от e_x */
    e_1 = e_x⟨0, X_1⟩
    /* в e_x за 1-ой копией e_1 следует еще копия e_1? */
    if e_1 ≠ e_x⟨X_1, X_1⟩ then goto next
    /* в e_x за 2-ой копией e_1 следует еще копия e_1? */
    if e_1 ≠ e_x⟨2 · X_1, X_1⟩ then goto next
    /* в e_x за 3-ей копией e_1 больше ничего нет? */
    if |e_x| ≠ 3 · X_1 then goto next
    return True
  next:
  }
  return False
}
```

Рис. 1. Неэффективная схема реализации функции `IsTriplet`

можно избежать. Действительно, предположим, нам удалось подобрать такое значение e_1 некоторой длинны X_1, что e_x совпадает с конкатенацией $e_1\,e_1\,e_1$ трех копий e_1. Тогда мы можем утверждать, что:

- выполнено следующее соотношение между длинами e_x и e_1: $A_x = 3 \cdot X_1$, то есть A_x делится на три нацело;
- e_1 является префиксом длины $A_x/3$ выражения e_x.

Тем самым найден новый эффективный способ вычисления функции `IsTriplet` (см. Рис. 2). Способ имеет вычислительную сложность не $O({A_x}^2)$, а $O(A_x)$. В нем не рассматривается (отсекается) множество вариантов значений для e_1, которые не имеют никаких шансов быть решением перестройки, вместо A_x вариантов рассматривается ровно один вариант: $e_1 = e_x\langle 0, |eX|/3\rangle$. Такая реализация функции `IsTriplet` обеспечивает *ускорение примерно в* 8 500 *раз*[3] вычислений, упомянутых в Таблице 1.

[3] Отношение числа сравнений символов: $17\,000 \cdot 17\,001/(2 \cdot 17\,000) = 8\,500.5$.

```
function IsTriplet(expr e_x) {
  expr e_1
  int A_x, X_1
  A_x = |e_x|
  if A_x mod 3 ≠ 0 then goto branch2
  X_1 = A_x / 3
  e_1 = e_x⟨0, X_1⟩
  /* в e_x за 1-ой копией e_1 следует еще копия e_1? */
  if e_1 ≠ e_x⟨X_1, X_1⟩ then goto branch2
  /* в e_x за 2-ой копией e_1 следует еще копия e_1? */
  if e_1 ≠ e_x⟨2 · X_1, X_1⟩ then goto branch2
  return True
branch2:
  return False
}
```

Рис. 2. Новая схема вычисления функции `IsTriplet`

1.1.2. *Пример на тему «учет ограничений на длины».* Рассмотрим пример рефал-функции `IsSubexp`:

```
IsSubexp e_x e_y = {
   e_x : e_1 e_y e_2        = True;
   /* otherwise */ = False;
};
```

Во всех существовавших ранее реализациях диалектов Рефала вычисление такой функции выполняется по следующей схеме:

- в цикле поочередно рассматриваются все префиксы e_1 выражения e_x, с длинами от 0 до $A_x = |e_x|$;
- проверяется, верно ли, что за префиксом e_1 в выражении e_x имеется выражение, равное e_y. Если проверка не успешная, то в качестве e_1 рассматривается следующий префикс e_x (удлинение переменной e_1). Если проверка успешная, то функция вычислена, результат `True`;
- если все префиксы e_1 выражения e_x рассмотрены и невозможно очередное удлинение переменной e_1, то функция вычислена, результат `False`.

```
function IsSubexp(expr e_x, expr e_y) {
  int A_x = |e_x|
  int A_y = |e_y|
  expr e_1, e_2
  int X_1
  /* цикл удлинения e_1, перебор вариантов */
  for (X_1 = 0; X_1 ⩽ (A_x − A_y); X_1++) {
    /* в качестве e_1 берем префикс длиной X_1 от e_x */
    e_1 = e_x⟨0, X_1⟩
    /* в e_x за e_1 следует e_y? */
    if e_y ≠ e_x⟨X_1, A_y⟩ then goto next
    /* закрытая переменная e_2 */
    e_2 = e_x⟨X_1 + A_y, A_x − A_y − X_1⟩
    return True
  next:
  }
  return False
}
```

Рис. 3. Эффективная схема реализации функции `IsSubexp`

Очевидно, что при такой схеме реализации выполняются лишние итерации: ясно, что надо рассматривать *не все* префиксы e_1 выражения e_x, с длинами от 0 до $A_x = |e_x|$, а только префиксы с длинами от 0 до $(A_x - A_y)$, где $A_y = |e_y|$.

Формально это легко получить из следующего анализа: перестройке $e_x : e_1\, e_y\, e_2$ можно поставить в соответствие уравнение

$$A_x = X_1 + A_y + X_2$$

- *с параметрами* A_x и A_y, обозначающими длины переменных e_x и e_y, эти длины будут известны во время исполнения программы в момент начала вычисления перестройки $e_x : e_1\, e_y\, e_2$;
- *с неизвестными* X_1 и X_2, обозначающими длины переменных e_1 и e_2, эти длины будут неизвестны во время исполнения программы в момент начала вычисления перестройки $e_x : e_1\, e_y\, e_2$.

Это уравнение в неотрицательных числах имеет очевидное решение:

$$\begin{cases} X_1 \in \mathbb{Z}[0..A_x - A_y] \\ X_2 = A_x - A_y - X_1 \end{cases}$$

что позволяет выписать более эффективную (чем описанную выше) схему реализации функции `IsSubexp` (см. Рис. 3).

1.1.3. *Пример на тему «цепочка перестроек».* На примере следующей функции:

```
AreSymmetric e_x e_y = {
    e_x : e_1 e_2,  e_y : e_3 e_4,  e_1 e_4 : e_3 e_2  = True;
    /* otherwise */                                    = False;
};
```

покажем, как рассмотрение не одиночной перестройки, а *цепочки перестроек*[4], и аккуратная работа с уравнениями и неравенствами над длинами переменных позволяет строить эффективный императивный код.

Длины значений переменных e_x, e_y, e_1, e_2, e_3, e_4 обозначим так: параметры A_x, A_y; неизвестные X_1, X_2, X_3, X_4. По цепочке перестроек ($e_x : e_1\,e_2,\ e_y : e_3\,e_4,\ e_1\,e_4 : e_3\,e_2$) очевидным образом выписывается система из трех уравнений:

$$(1) \qquad \begin{cases} A_x = X_1 + X_2 \\ A_y = X_3 + X_4 \\ X_1 + X_4 = X_3 + X_2 \end{cases}$$

Переносим все члены в левую часть:

$$(2) \qquad \begin{cases} A_x - X_1 - X_2 = 0 \\ A_y - X_3 - X_4 = 0 \\ X_1 + X_4 - X_3 - X_2 = 0 \end{cases}$$

Из первого уравнения выражаем первую переменную (X_1) и подставляем во все остальные уравнения:

$$(3) \qquad \begin{cases} X_1 = A_x - X_2 \\ A_y - X_3 - X_4 = 0 \\ A_x + X_4 - X_3 - 2 \cdot X_2 = 0 \end{cases}$$

[4] Несложно показать, что цепочка перестроек из функции `AreSymmetric` имеет решение только в следующих случаях:

- если $e_x : e_3\,e_0\,e_0\,e_4$ и $e_y : e_3\,e_4$ (тогда $e_1 = e_3\,e_0$, $e_2 = e_0\,e_4$);
- если $e_x : e_1\,e_2$ и $e_y : e_1\,e_0\,e_0\,e_2$ (тогда $e_3 = e_1\,e_0$, $e_4 = e_0\,e_2$).

Теперь мы берем из выражения для X_1 первую по порядку переменную (X_2), находим уравнение, позволяющее ее выразить (третье уравнение), выражаем эту переменную ($2 \cdot X_2 = A_x + X_4 - X_3$) и подставляем во все остальные уравнения, где она содержится (предварительно умножив их на два):

$$\left\{\begin{array}{r} 2 \cdot X_1 = A_x - X_4 + X_3 \\ 2 \cdot X_2 = A_x + X_4 - X_3 \\ A_y - X_3 - X_4 = 0 \end{array}\right. \tag{4}$$

Теперь мы берем из выражения для X_2 первую по порядку переменную (X_4), выражаем ее из последнего[5] уравнения ($X_4 = A_y - X_3$) и подставляем во все остальные уравнения, где она содержится:

$$\left\{\begin{array}{l} 2 \cdot X_1 = A_x - A_y + 2 \cdot X_3 \\ 2 \cdot X_2 = A_x + A_y - 2 \cdot X_3 \\ X_4 = A_y - X_3 \end{array}\right. \tag{5}$$

Мы выразили переменные X_1, X_2 и X_4 через параметры и свободную переменную X_3. Так как все X_i не могут быть отрицательными, то из системы (5) получаем следующие ограничения на значения свободной переменной X_3:

$$\max\left(\lceil (A_y - A_x)/2 \rceil, 0\right) \leqslant X_3 \leqslant \min\left(A_y, \lfloor (A_y + A_x)/2 \rfloor\right) \tag{6}$$

Из ограничения на значения X_3 и из первого уравнения системы (5) получаем уточненные границы для переменной X_1:

$$\max\left(\lceil (A_x - A_y)/2 \rceil, 0\right) \leqslant X_1 \leqslant \min\left(A_x, \lfloor (A_x + A_y)/2 \rfloor\right) \tag{7}$$

Итак, мы получили уточненные границы для цикла по параметру X_1, который является длиной переменной e_1. Внутри данного цикла переменная X_1 будет известной, означенной переменной, то есть параметром—переобозначим его $A_1 = X_1$. И для тела цикла система (2) примет следующий вид:

$$\left\{\begin{array}{r} A_x - A_1 - X_2 = 0 \\ A_y - X_3 - X_4 = 0 \\ A_1 + X_4 - X_3 - X_2 = 0 \end{array}\right. \tag{8}$$

В системе (8) мы разрешаем одну переменную за другой:

$$\left\{\begin{array}{r} X_2 = A_x - A_1 \\ X_3 = A_y - X_4 \\ 2 \cdot X_4 = A_y + A_x - 2 \cdot A_1 \end{array}\right. \tag{9}$$

[5] Первые два уравнения мы уже использовали.

```
function AreSymmetric(expr e_x, expr e_y) {
  int A_x = |e_x|
  int A_y = |e_y|
  expr e_1, e_2, e_3, e_4
  int X_1, X_2, X_3, X_4
  /* цикл удлинения переменой e_1, перебор вариантов */
  int min_1 = max(0, ⌊(A_x − A_y)/2⌋)
  int max_1 = min(A_x, ⌈(A_x + A_y)/2⌉)
  for (X_1 = min_1; X_1 ⩽ max_1; X_1++) {
      X_2 = A_x − X_1
      /* 2 · X_4 = A_y + A_x − 2 · X_1 */
      if (A_y + A_x − 2 · X_1) mod 2 ≠ 0 then goto next
      X_4 = (A_y + A_x − 2 · X_1)/2
      X_3 = A_y − X_4
      e_1 = e_x⟨0, X_1⟩
      e_2 = e_x⟨X_1, X_2⟩
      e_3 = e_y⟨0, X_3⟩
      e_4 = e_y⟨X_3, X_4⟩
      if e_1 ++ e_4 ≠ e_3 ++ e_2 then goto next
      return True
  next:
  }
  return False
}
```

Рис. 4. Эффективная схема реализации функции `AreSymmetric`

После этого легко построить схему вычисления (см. Рис. 4) функции `AreSymmetric`. Отметим, что за счет решения системы уравнений на длины переменных, полученный алгоритм имеет не два вложенных цикла, а один цикл, и работает он в среднем в $|e_y|$ раз быстрее, чем это есть для классических способов компиляции данной функции.

1.2. От идеи к реализации. В предыдущих разделах показаны примеры эффективной компиляции синтаксического отождествления в абстрактный императивный язык, допускающий простую и

эффективную реализацию на современных ЭВМ. При этом предполагалось, что операции вычисления длины выражения и конструирования подвыражения выполняются за константное (не зависящее от выражения) время. Это предположение верно для всех современных реализаций языка Рефал.

За счет дешевизны этих операций в процессе синтаксического отождествления можно вести совместный анализ длин объектных выражений и условий на длины свободных переменных. Зачастую такой анализ позволяет быстро (без посимвольного сравнения выражений) отсечь целые области значений свободных переменных, на которых отождествление заведомо не может дать положительного результата.

Отметим также, что дешевая операция конструирования подвыражения используется для дробления сопоставлений на части, что позволяет вынести за пределы циклов максимальное количество операций.

Рассматриваемые в работе оптимизации достаточно очевидны и, при этом, они дают существенный выигрыш в эффективности. Однако до сих пор они не использовались в реализациях языка Рефал.

2. Синтаксическое отождествление: основные понятия

2.1. Основные понятия и обозначения. Основные понятия языка Рефал и основные обозначения мы заимствуем из работы [2]. В том числе заимствуем следующие понятия и обозначения для них:

- Oe — объектное выражение;
- Os — объектный символ;
- Pe — образцовое (т. е. возможно содержащее переменные) выражение;
- Pt — образцовый терм (т. е. объектный символ, переменная или образцовое выражение в скобках);
- He — жесткое выражение.

Мы расширяем понятие типа рефал-переменной следующим образом: через $V[m,n]$, где $m \in \mathbb{Z}$ — целое число, $n \in \tilde{\mathbb{Z}}$ — целое число или ∞, мы обозначаем рефал-переменную, допустимые значения которой могут иметь длины из отрезка $[m,n]$. Привычные переменные Рефала в такой нотации будут записаны следующим образом:

- s-переменные: $s[1,1]$;
- t-переменные: $t[1,1]$;
- e-переменные: $t[0,\infty]$;

- v-переменные: $t[1, \infty]$;

2.2. Базовый случай: отождествление объектного выражения с образцом. *Базовым клэшем* (от английского clash) будем называть конструкцию вида $Oe : Pe$, где Oe — объектное выражение, а Pe — образцовое выражение.

Решением базового клэша называется упорядоченное множество $[\sigma_1, \ldots, \sigma_n]$ подстановок $\sigma_i = \{V_j \to Oe_{ij} \mid V_j$ — переменная из $Pe\}$, при которых Pe редуцируется к Oe: $Pe/\sigma_i = Oe$.

Во время выполнения рефал-программ необходимо уметь находить решения, удовлетворяющие определенному отношению порядка на подстановках. Существующие диалекты Рефала определяют только два противоположных отношения порядка, называемых [2] сопоставлениями *слева направо* и *справа налево*.

Поскольку варианты всех нижеописанных алгоритмов для этих двух случаев абсолютно аналогичны, мы ограничимся рассмотрением сопоставления слева направо, которое задается следующим образом. Пусть σ_1, σ_2 — подстановки-решения базового клэша $Oe : Pe$, тогда говорят, что подстановка σ_1 предшествует подстановке σ_2, если V/σ_1 короче, чем V/σ_2, где V — первая (слева направо) переменная из Pe, такая, что $V/\sigma_1 \neq V/\sigma_2$.

2.3. Перестройка. В рефал-программе будем называть *перестройкой* конструкцию $Pe_L : Pe_R$, где и левая, и правая части являются образцовыми выражениями, причем Pe_L является *источником*, то есть, Pe_L является выражением, содержащим только *старые переменные*[6] — переменные, имеющие значение к моменту вычисления данного выражения.

2.4. Выполнение и компиляция перестройки. Выражения-источники к моменту выполнения перестройки всегда редуцируются до объектных выражений. Таким образом, в корректно составленной программе перестройка $Pe_L : Pe_R$ в момент своего выполнения редуцируется до базового клэша, и ее выполнение сводится к решению этого базового клэша. А во время компиляции компилятор должен по заданной перестройке $Pe_L : Pe_R$ выписать императивный алгоритм

[6] Старые переменные — означенные переменные, переменные, чьи значения были определены в рефал-программе ранее. Все остальные переменные называются *новыми*.

для решения любых базовых клэшей, которые могут получиться из данной перестройки при разных значениях старых переменных.

2.5. Цепочки перестроек. Как было показано в разделе 1, алгоритм поиска решения можно сделать более эффективным, если вместо одной перестройки рассматривать цепочку подряд идущих перестроек. Это связано с тем, что последующие перестройки могут накладывать ограничения на переменные, встречающиеся в ранее идущих перестройках, сокращая, тем самым, область перебора вариантов.

Поэтому мы везде далее будем работать с цепочкой перестроек

$$\langle Pe_L^1 : Pe_R^1, \ldots, Pe_L^n : Pe_R^n \rangle,$$

в предположении, что компилятор позаботился о нахождении максимально длинной цепочки подряд идущих перестроек.

При фиксированных значениях старых переменных *решение цепочки перестроек* строится следующим образом.

Пусть $[\sigma_1, \ldots, \sigma_k]$ — решение цепочки

$$\langle Pe_L^1 : Pe_R^1, \ldots, Pe_L^{n-1} : Pe_R^{n-1} \rangle.$$

Тогда решение цепочки

$$\langle Pe_L^1 : Pe_R^1, \ldots, Pe_L^n : Pe_R^n \rangle$$

есть объединение (с сохранением порядка элементов) $S_1 \cup \ldots \cup S_k$ упорядоченных наборов подстановок, где каждый набор S_i получается объединением подстановки σ_i поочередно с каждой подстановкой из решения перестройки $Pe_L^n/\sigma_i : Pe_R^n/\sigma_i$ (если решение этой перестройки пустое, то и набор S_i будет пустым).

3. Компиляция цепочки перестроек

В данном разделе мы рассмотрим основные понятия и детали реализации алгоритма компиляции цепочки перестроек, используемого в новой реализации [9, 12] языка Рефал Плюс.

Как уже было сказано, компилятор будет работать со следующими объектами: рефал-переменные V, объектные выражения Oe, образцовые выражения Pe, перестройки $Pe_L : Pe_R$, цепочки перестроек $S_{cl} = \langle Pe_L^1 : Pe_R^1, \ldots, Pe_L^n : Pe_R^n \rangle$.

Компилятор строит код императивной программы, выполняющей заданную цепочку перестроек S_{cl}. В процессе выполнения этой цепочки все больше новых переменных, входящих в S_{cl}, получают

значения — становятся старыми переменными. Для того чтобы хранить информацию о том, какие переменные будут иметь значения после исполнения кода, построенного на данный момент, компилятор использует список старых переменных T_{old}.

3.1. Генерируемая императивная программа. При компиляции цепочки перестроек в выходной программе используются следующие конструкции императивного языка:

- переменные императивного языка двух типов:
  - **expr**-переменные, значениями которых являются объектные рефал-выражения;
  - **int**-переменные, значениями которых являются неотрицательные целые числа (длины объектных выражений) и особое значение[7] ∞;
- выражения двух типов (**expr** и **int**[8]), построенные над этими переменными и константами при помощи соответствующих наборов операций (см. Рис. 5);
- логические выражения, используемые в условиях операторов **if** и **for** (см. Рис. 5);
- операторы (см. Рис. 5).

Подчеркнем, что здесь рассматривается подмножество императивного языка[9], достаточное для компиляции цепочки перестроек.

Семантика выражений и операторов достаточна очевидна. Поэтому дадим пояснения только по поводу оператора **for**, который в данной работе всегда имеет следующий вид:

```
for (i = a1; i ⩽ a2; i++) {
    op
label:
}
```

[7] Значение ∞ будет встречаться только в таком контексте, что результат операций с ним определяется интуитивно понятным, однозначным образом: $\infty + \infty = \infty$, $\infty + x = \infty$, $\mathbf{min}(\infty, x) = x$ и т. п. (здесь x — неотрицательное целое число).

[8] Значениями выражений типа **int** могут являться любые целые числа, ∞ и $-\infty$.

[9] В работе [12] императивный язык, используемый для компиляции рефал-программ в проекте [9], описан полнее.

где i — **int**-переменная; a_1 и a_2 — **int**-выражения; op — оператор (возможно составной), тело цикла; $label$ — метка, переход на которую (в нашем случае может быть только из тела цикла) вызывает выполнение следующей по очереди итерации цикла. В нашем случае, выражения a_1 и a_2 не будут зависеть от переменных, изменяемых в теле цикла, поэтому значения (i_1 и i_2) этих выражений можно посчитать заранее, до входа в цикл и в нашем случае эти значения будут неотрицательными числами. Выполнение рассматриваемого оператора **for** состоит в выполнении тела цикла для всех целочисленных значений переменных i от i_1 до i_2 включительно. Если $i_1 > i_2$, то тело цикла не будет выполнено ни разу, во всех остальных случаях тело цикла будет выполнено $(i_2 - i_1 + 1)$ раз.

3.2. Состояние компилятора. Компилятор можно рассматривать как рекурсивную функцию со следующим аргументом (состояние компилятора)

$$(S_{cl}, T_{old}, L_{fail}),$$

где

- S_{cl} — цепочка перестроек;
- T_{old} — список старых переменных;
- L_{fail} — метка в ранее построенном коде императивной программы, переход на которую обычно соответствует выполнению следующей попытки удлинения текущей открытой (итерируемой) переменной.

Кроме того, неявно предполагается, что в процессе работы происходит мемоизация полученных состояний и промежуточных вычислений. Это используется для того, чтобы не генерировать в коде императивной программы лишних проверок (см. шаг 4 алгоритма **EqA**, раздел 3.7, и шаг 2 алгоритма **ClA**, раздел 3.15).

В начальный момент времени список T_{old} содержит переменные из S_{cl}, значения которых известны до начала выполнения данной цепочки перестроек.

Метка L_{fail} изначально соответствует фрагменту кода императивной программы, исполняемому в случае, когда выполнение цепочки перестроек заканчивается неуспехом.

	int-*выражения*
$ie ::= iv \mid ic$	— переменные и константы
$\mid \mathbf{min}(ie, ie) \mid \mathbf{max}(ie, ie)$	— минимум и максимум
$\mid \lceil ie/ie \rceil \mid \lfloor ie/ie \rfloor$	— деление с округлением (вверх/вниз)
$\mid ie/ie \mid ie\ \mathbf{mod}\ ie$	— деление нацело, остаток от деления
$\mid ie \cdot ie \mid ie + ie \mid ie - ie$	— умножение, сложение, вычитание
$\mid \lvert ee \rvert$	— длина выражения
	expr-*выражения*
$ee ::= ev \mid ec$	— переменные и константы
$\mid ee +\!+\ ee$	— конкатенация
$\mid (ee) \mid *ee$	— взятие в скобки и снятие скобок
$\mid ee\langle ie, ie\rangle \mid ee[ie]$	— взятие подвыражения и терма
	логические выражения
$be ::=$ `true` $\mid$ `false`	— логические константы
$\mid be$ **or** be	— логическое «или»
$\mid$ **is_symbol**(ee)	— значение ee — символ?
$\mid$ **is_parenth**(ee)	— значение ee — скобочный терм?
$\mid ee = ee \mid ee \neq ee$	— сравнения **expr**-значений
$\mid ie\ \nabla\ ie$	— сравнение **int**-значений, где:
$\nabla ::= < \mid > \mid \leqslant \mid \geqslant \mid \neq \mid =$	— операции сравнения **int**-значений
	операторы
$op ::=$ **expr** $ev, \ldots ev$	— определение **expr**-переменных
$\mid$ **int** $iv, \ldots iv$	— определение **int**-переменных
$\mid op;\ op$	— последовательность операторов
$\mid$ **if** be **then goto** lb	— условный переход на метку lb
$\mid iv = ie \mid ev = ee$	— **int**- и **expr**-присваивания
$\mid$ **for** $(iv = ie;\ be;\ iv{+}{+})$ $\{op;\ lb{:}\}$	— цикл по **int**-переменной

Рис. 5. Конструкции императивного языка, используемые при компиляции цепочек перестроек

3.3. CSC: алгоритм компиляции цепочки перестроек. Алгоритм компиляции **CSC** для текущего состояния $(S_{cl}, T_{old}, L_{fail})$ можно описать следующим образом:

Шаг 1. Если цепочка S_{cl} пустая, то завершить работу алгоритма[10].

Шаг 2. В противном случае выполняется:

(1) Построение системы уравнений S^{Δ}_{eq}, соответствующей цепочке S_{cl} (см. разделы 3.4–3.5, алгоритм **EqB**).

(2) Анализ системы S^{Δ}_{eq} (см. раздел 3.7, алгоритм **EqA**), построение в коде императивной программы линейной последовательности проверок условий на длины:

- проверки равенства длин (см. раздел 3.11, алгоритм **ChkEq**);
- проверки ограничений и кратности для длин новых жестких переменных (см. раздел 3.12, алгоритм **ChkHard**);
- проверки неравенств для границ длин (см. разделы 3.13 и 3.14, алгоритм **ChkBnd**).

(3) Анализ цепочки перестроек S_{cl} (см. раздел 3.15, алгоритм **ClA**), построение фрагмента кода императивной программы, изменение состояния компилятора:
$$(S_{cl}, T_{old}, L_{fail}) \to (S'_{cl}, T'_{old}, L'_{fail}).$$

(4) Рекурсивный вызов алгоритма **CSC** для нового состояния.

3.4. Система уравнений, соответствующая цепочке перестроек. *Система уравнений на длины переменных* — это неупорядоченный набор $S_{eq} = \langle Eq_1, \ldots, Eq_n \rangle$, элементы которого называются *уравнениями на длины переменных* и имеют следующий вид:

$$\begin{array}{rcl} Eq & ::= & \Sigma_L = \Sigma_R \\ \Sigma & ::= & c_1 I_1 + \cdots + c_k I_k \\ I & ::= & 1 \mid A \mid X \end{array}$$

где $c \in \mathbb{Z}$ — целое число, A — конструкция императивного языка, редуцируемая во время исполнения к неотрицательному целому числу, X — переменная.

Систему уравнений на длины переменных S_{eq} можно рассматривать как обычную алгебраическую систему уравнений относительно

[10] Алгоритмы компиляции остатка рефал-предложения, следующего за скомпилированной цепочкой перестроек, будут выводить код императивной программы в позицию, обозначенную ■.

целочисленных переменных X_j, где выражения A_j являются параметрами. Во время выполнения программы эти параметры (A_j) получают конкретные значения, и тогда можно говорить о решении системы.

Решением системы уравнений S_{eq}, при заданных значениях параметров A_j, называется множество $\{\delta_1, \ldots, \delta_m\}$ всех таких подстановок $\delta_i = \{X_j \to n_{ij} \mid X_j - \text{переменная из } S_{eq}\}$, $n_{ij} \in \mathbb{Z}$, $n_{ij} \geqslant 0$, при которых каждое уравнение Eq_k из S_{eq} обращается в верное равенство, то есть для любых k, i, следующий императивный код вычислится в `true`: $\Sigma_L^k/\delta_i = \Sigma_R^k/\delta_i$

Две системы уравнений S_{eq} и S'_{eq} называются *эквивалентными*, если при любых значениях параметров A_j их решения совпадают.

Будем говорить, что система уравнений S_{eq} *соответствует цепочке перестроек* S_{cl}, если она эквивалентна системе S_{eq}^{Δ}, которая строится из S_{cl} алгоритмом **EqB**.

3.5. EqB: алгоритм построения системы уравнений S_{eq}^{Δ}. Алгоритм **EqB** по заданной цепочке перестроек порождает систему S_{eq}^{Δ} уравнений на длины по следующим правилам:

- из каждой перестройки $Pe_L : Pe_R$ получается одно уравнение $\Sigma_L = \Sigma_R$, причем Σ_L строится в результате анализа верхнего уровня выражения Pe_L, а Σ_R — в результате анализа верхнего уровня выражения Pe_R;
- объектный символ Os заменяется на 1;
- выражение в скобках (Pe) заменяется на 1;
- старая переменная V_j заменяется на $A_j = |V_j|$;
- новая переменная $V_j[m, m]$ заменяется на m;
- новая переменная $V_j[m, n]$, где $m \neq n$, заменяется на X_j (в этом случае, так же мы будем использовать для X_j обозначение $X_j[m, n]$).

Пример построения системы уравнений на длины по заданной цепочке перестроек приведен в разделе 1.1.3.

Нас будут интересовать только такие подстановки $\{X_j \to k_j\}$ $(k \in \mathbb{Z})$, решающие систему уравнений на длины, что для всякой переменной X_j, соответствующей рефал-переменной $V_j[m, n]$, выполнено: $m \leqslant k_j \leqslant n$. Поэтому мы везде далее будем говорить, что система уравнений на длины *совместна*, если множество подстановок, обладающих указанным свойством, не пусто. Иначе, мы будем назвать систему *несовместной*.

3.6. Основные свойства систем уравнений, соответствующих цепочкам перестроек. Если система уравнений S_{eq} соответствует цепочке перестроек S_{cl}, то верны следующие утверждения.

Утверждение 3.1. *Каждой новой переменной $V[m,n]$, $m \neq n$, из цепочки S_{cl} соответствует ровно одна переменная X системы S_{eq}.*

Действительно, это верно по построению для системы S^{Δ}_{eq}, а следовательно, и для всех эквивалентных ей систем.

Теорема 3.2. *Пусть $[\sigma_1, \ldots, \sigma_n]$ — решение цепочки S_{cl} при фиксированных значениях старых переменных, $\Delta = \{\delta_1, \ldots, \delta_n\}$ — множество подстановок $\delta_i = \{X_j \to |V_j[m,n]/\sigma_i|\}$, где j пробегает индексы новых переменных из цепочки S_{cl}, для которых $m \neq n$.*

Тогда, Δ является подмножеством решения системы S_{eq} при соответствующих значениях параметров. При этом, для всех δ_i и всех $X_j[m,n]$ выполнено $n \leqslant X_j/\delta_i \leqslant m$.

Заметим, что для любой из подстановок δ_i, для любого уравнения $\Sigma^j_L = \Sigma^j_R$ системы S^{Δ}_{eq}, Σ^j_L/δ_i редуцируется к длине Pe^j_L/σ_i, а Σ^j_R/δ_i редуцируется к длине Pe^j_R/σ_i. Но Pe^j_L/σ_i совпадает с Pe^j_R/σ_i, так как подстановка σ_i принадлежит решению цепочки перестроек S_{cl}. Тем самым, теорема верна для системы S^{Δ}_{eq}. Как следствие, теорема верна и для всех эквивалентных ей систем уравнений.

Теорема 3.2 позволяет утверждать, что в качестве потенциальных решений цепочки перестроек S_{cl} имеет смысл рассматривать только такие подстановки $\{V_j \to Oe_j\}$, что длины выражений Oe_j удовлетворяют системе уравнений S_{eq}, соответствующей цепочке S_{cl}.

3.7. EqA: алгоритм анализа системы уравнений на длины. Анализ системы уравнений S_{eq}, соответствующей цепочке перестроек S_{cl}, производится с целью выписывания условий на параметры A и переменные X, выполнение которых *необходимо*, чтобы система S_{eq} была совместной. Соответственно, если какие-либо из этих условий не выполнены, то система S_{eq} будет несовместной, и, в силу теоремы 3.2, не будет подстановок, решающих цепочку S_{cl}.

Непосредственно из определения эквивалентных систем следует, что все обычные алгебраические преобразования систем уравнений, которые ведут к эквивалентным в алгебраическом смысле системам,

будут вести к эквивалентным системам и в нашем случае. В частности, умножение обеих частей уравнения на число, перенос членов уравнения в другую часть (со сменой знака) и подстановка переменной, выраженной из какого-либо уравнения, в другие уравнения системы не влияют на соответствие системы уравнений S_{eq} цепочке перестроек S_{cl}.

Алгоритм **EqA** принимает на вход следующие параметры (начальное состояние алгоритма):

$$(S_{eq}, S_{cl}, L_{fail}),$$

где S_{cl} и L_{fail} являются параметрами компилятора цепочки перестроек (см. раздел 3.2), а S_{eq} — система уравнений, соответствующая цепочке перестроек S_{cl} (см. раздел 3.5, алгоритм **EqB**).

В результате работы алгоритма **EqA** генерируется фрагмент кода программы на императивном языке, а также следующие выходные параметры:

$$(T_{hard}, Min, Max),$$

где

- T_{hard} — список переменных из цепочки перестроек S_{cl}, длина которых будет известна после исполнения кода императивной программы, сгенерированного к моменту окончания работы алгоритма;
- Min, Max — выражения императивного языка, редуцируемые во время исполнения соответственно к минимальному и максимальному возможным значениям длины переменной, являющейся кандидатом на место текущей открытой (итерируемой) переменной.

Алгоритм **EqA** можно представить в виде (линейной) последовательности следующих шагов:

Шаг 1. Упрощение системы S_{eq} путем приведения подобных членов в ее уравнениях (см. раздел 3.8, алгоритм **EqS**).

Шаг 2. Построение упорядоченных непересекающихся групп уравнений $G_0, G_1, \ldots, G_k$ обладающих следующими свойствами (см. раздел 3.9, алгоритм **EqG**):

- система $S'_{eq} = G_0 \cup G_1 \cup \cdots \cup G_k$ эквивалентна системе S_{eq};
- каждое уравнение из G_0 не содержит переменных X;

- каждое уравнение из G_p (где $p > 0$) имеет вид $c \cdot X = \Sigma$, где Σ не содержит переменных из левых частей уравнений группы G_p, и $c \neq 0$;
- множество переменных, входящих в некоторую группу, не пересекается с множеством переменных, входящих в любую другую группу.

Таким образом, система S_{eq} разбивается на непересекающиеся по множествам содержащихся переменных группы уравнений, и в каждой из таких групп некоторый набор переменных выражается через параметры и остальные переменные, которые мы будем называть *свободными* переменными.

В каждой группе G_p (где $p > 0$) переменную X из первого уравнения $c \cdot X = \Sigma$ такого, что Σ содержит переменные, будем называть *ведущей* переменной группы G_p.

Шаг 3. Формирование списка T_{hard} переменных из цепочки S_{cl}, длина которых может быть вычислена после исполнения кода императивной программы, сгенерированного к моменту окончания работы алгоритма **EqA** (см. раздел 3.10, алгоритм **THard**).

Шаг 4. Анализ системы $S'_{eq} = G_0 \cup G_1 \cup \cdots \cup G_k$ и построение в коде императивной программы линейной последовательности следующих проверок:

(1) для каждого уравнения Eq из группы G_0 выписывается проверка условия, которое Eq накладывает на параметры A (см. раздел 3.11, алгоритм **ChkEq**);

(2) для каждого уравнения $c \cdot X[m, n] = \Sigma$ системы S'_{eq}, где Σ не содержит переменных, выписываются проверки делимости Σ на c нацело и неравенств $c \cdot m \leqslant \Sigma \leqslant c \cdot n$ (см. раздел 3.12, алгоритм **ChkHard**);

(3) для каждой свободной переменной каждой группы G_p, $p > 0$, выписывается[11] проверка условия, необходимого,

[11] При грамотной реализации алгоритма проверки ограничений на свободные переменные должны выписываться только в том случае, когда нет «дешевого» способа эти ограничения улучшить. Например, если, исходя из составленного списка T_{hard}, ясно, что можно (не используя конкатенаций объектных выражений) составить новую перестройку из содержимого скобок, входящих в одну из старых перестроек (см. раздел 3.15, шаг 1 алгоритма **ClA**), и эта новая перестройка будет содержать переменные из группы G_p, то выписывание условий на свободные переменные из группы G_p лучше отложить до следующих вызовов алгоритма **EqA**.

чтобы множество возможных значений этой переменной было не пусто (см. раздел 3.13, алгоритм **ChkBnd**). Вывод некоторой проверки в код императивной программы блокируется, если эта проверка была построена алгоритмом **EqA** ранее.

Шаг 5. Поиск в цепочке перестроек S_{cl} первой (слева) переменной V, не содержащейся в списке T_{hard}.

Если в результате последующего анализа цепочки S_{cl} (раздел 3.15) в коде императивной программы будет построен цикл (см. шаг 8 алгоритма **ClA**), то это будет цикл удлинения именно переменной V, и она станет текущей открытой (итерируемой) переменной. При этом, длина V в цикле будет изменяться в некоторых пределах: от Min до Max.

Можно показать, что переменная X, соответствующая переменной V, является ведущей переменной группы G_1. Ограничения снизу и сверху (Min и Max) на ведущую переменную V строятся при помощи алгоритма **VMinMax** (см. раздел 3.14).

Если указанной переменной V не существует, то параметры Min и Max не определены.

В следующих разделах подробно рассмотрены все алгоритмы, вызываемые из алгоритма **EqA**.

3.8. EqS: алгоритм приведения подобных членов. Каждое уравнение $\Sigma_L = \Sigma_R$ системы S_{eq} заменяется на уравнение $\Sigma = 0$, где Σ получается переносом всех членов Σ_R в левую часть (со знаком минус), и приведением подобных членов. Слагаемые вида $0 \cdot X$ удаляются из суммы Σ.

3.9. EqG: алгоритм разбиения системы на группы. Алгоритм **EqG** разбивает систему уравнений S_{eq} на упорядоченные непересекающиеся группы уравнений $G_0, G_1, \ldots, G_k$ обладающие набором свойств (см. ниже), позволяющих выписать условия, необходимые, чтобы система S_{eq} была совместна.

Состоянием (аргументом) алгоритма **EqG** разбиения системы на группы является четверка (S, T, G_0, Gs), где

- S — система уравнений, изначально равная S_{eq};
- T — список переменных, встречающихся в рассмотренных уравнениях, изначально пустой: $T = \square$.

- G_0 — упорядоченная группа уравнений, изначально пустая.
- Gs — список упорядоченных групп уравнений, изначально пустой.

Шаг 1. Первое уравнение Eq изымается из системы S ($S' = S \setminus Eq$) и анализируется.

(1) Если Eq не содержит переменных X, то уравнение Eq добавляется в группу G_0.
Выполняется рекурсивный вызов алгоритма для полученного состояния.

(2) Иначе изменяется состояние алгоритма:
$$(S, \square, G_0, Gs) \to (S', [X], G_0, Gs'),$$
где X — первая (слева) переменная из Eq, а Gs' получен добавлением к Gs новой пустой группы. Выполняется переход к шагу 2.

Шаг 2. До тех пор, пока список T не пуст, выполняются следующие действия:

(1) Из списка T изымается первая переменная X:
$$T' = T \setminus X.$$

(2) В системе S ищется уравнение Eq, содержащее X (и изымается из системы: $S' = S \setminus Eq$).
Если такого уравнения Eq в системе S нет, то переменная X будет *свободной*. Выполняется изменение состояния алгоритма: $(S, T, G_0, Gs) \to (S, T', G_0, Gs)$, затем рекурсивный переход к шагу 2.
Иначе продолжается выполнение текущего шага.

(3) По уравнению Eq строится уравнение Eq', имеющее вид $c \cdot X = \Sigma$ (все слагаемые, кроме $c \cdot X$, переносятся в правую часть).
Уравнение Eq' добавляется в конец последней группы списка Gs (результат: Gs').

(4) Изменяется состояние алгоритма:
$$(S, T, G_0, Gs) \to (S'', T'', G_0, Gs''),$$
где

- S'' получается из S' заменой $d \cdot c \cdot X$ на $d \cdot \Sigma$ во всех уравнениях (предварительно умноженных на c), содержащих X, и приведением подобных членов в этих уравнениях.

- T'' получается из T' добавлением в конец списка всех переменных, входящих в выражение Σ.
- Gs'' получается из Gs' заменой $d \cdot c \cdot X$ на $d \cdot \Sigma$ во всех правых частях уравнений (предварительно умноженных на c) последней группы списка Gs', содержащих X, и приведением подобных членов в правых частях этих уравнений.

(5) Выполняется рекурсивный переход к шагу 2.

Шаг 3. Если система S не пуста, то выполняется переход к шагу 1, иначе алгоритм заканчивает работу, выдавая в качестве результата список групп: $G_0, G_1, \ldots, G_k$, где $[G_1, \ldots, G_k] = Gs$.

Сформулируем важные для дальнейшего изложения свойства построенного разбиения уравнений по группам.

Утверждение 3.3. *Пусть группы $G_0, G_1, \ldots, G_k$ получены с помощью алгоритма **EqG** из системы S_{eq}. Тогда верны следующие утверждения.*

(1) *Система $S'_{eq} = G_0 \cup G_1 \cup \cdots \cup G_k$ эквивалентна системе S_{eq}.*
(2) *Множество переменных, входящих в некоторую группу, не пересекается с множеством переменных, входящих в любую другую группу.*
(3) *Никакая переменная, входящая в правую часть (Σ) какого-либо уравнения $c \cdot X = \Sigma$ (то есть* свободная *переменная), не встречается в левой части никакого уравнения.*

Следствие 3.4. *Невыполнение любого из равенств группы G_0, или несовместность любой из групп $G_1, \ldots, G_k$ влечет несовместность изначальной системы S_{eq}.*

Следствие 3.5. *Каждая подстановка-решение системы S_{eq} однозначно задается значениями свободных переменных.*

Утверждение 3.6. *Пусть цепочка перестроек S'_{cl} получена из цепочки S_{cl} путем означивания*[12] *новой переменной V, которой соответствует переменная X в системе S_{eq}. Тогда результат применения алгоритма **EqG** к системе S'_{eq}, соответствующей цепочке S'_{cl}, будет содержать все группы $G_1, \ldots, G_k$, построенные по системе S_{eq}, кроме одной группы G, в которую входит переменная X, а*

[12] Такое означивание может произойти в результате выполнения кода, сгенерированного на шагах 4, 5, 6 и 8 алгоритма **ClA**, см. раздел 3.15.

построенная по системе S'_{eq} группа G'_0 будет содержать все уравнения группы G_0.

Группы (и уравнения, не содержащие переменных), входящие в $\mathbf{EqG}(S'_{eq})$ и не входящие в $\mathbf{EqG}(S_{eq})$, мы будем называть *конкретизирующими* (группами и уравнениями) для группы G.

Легко показать, что группы (и уравнения), конкретизирующие группу G, получаются из тех же уравнений системы S_{eq}, из которых получена группа G, после замены в них переменной X соответствующим параметром A.

Утверждение 3.7. *Пусть цепочка перестроек S'_{cl} получена из цепочки S_{cl} путем отщепления*[13] *жесткой (и не содержащей новых переменных) части некоторой перестройки $Oe : Pe$, причем перестройка не оказалась полностью вычисленной в результате этой операции.*

*Тогда в процессе применения алгоритма **EqA** к системе уравнений $\boldsymbol{EqB}(S'_{cl})$ будут построены ровно те же самые группы уравнений, что и для системы $\boldsymbol{EqB}(S_{cl})$.*

Это свойство следует из того факта, что после приведения подобных членов, системы будут полностью совпадать:

$$\mathbf{EqS}(\mathbf{EqB}(S'_{cl})) = \mathbf{EqS}(\mathbf{EqB}(S_{cl})).$$

3.10. THard: алгоритм построения списка переменных с вычисляемыми длинами. Алгоритм формирует список T_{hard}, исходя из следующего определения.

Элементами списка являются:

(1) все переменные $V[m, m]$ из цепочки S_{cl};
(2) все переменные V из цепочки S_{cl}, которым соответствуют параметры A системы S_{eq};
(3) все переменные V из цепочки S_{cl}, которым соответствуют переменные X системы S_{eq}, такие, что в одной из групп G присутствует уравнение $c \cdot X = \Sigma$, где Σ не содержит переменных.

[13] Отщепление жестких частей перестроек происходит в результате выполнения кода, сгенерированного на шагах 1 и 5 алгоритма **ClA**, см. раздел 3.15.

3.11. ChkEq: алгоритм генерации проверок на длины старых переменных и жестких выражений. В группе G_0 каждое уравнение $\Sigma = 0$ не содержит переменных, оно накладывает условие на параметры.

Алгоритм **ChkEq** для каждого уравнения из группы G_0, не конкретизирующего никакую из ранее построенных групп и не совпадающего ни с каким из ранее построенных уравнений, генерирует следующий фрагмент кода императивной программы:

if $\Sigma \neq 0$ **then goto** L_{fail}; ■

3.12. ChkHard: алгоритм генерации проверок на длины новых жестких переменных. Если в системе есть уравнение вида $c \cdot X[m,n] = \Sigma$, где Σ не содержит переменных, то система будет совместной только при таких значениях параметров, входящих в выражение Σ, что Σ делится на c, и $c \cdot m \leqslant \Sigma \leqslant c \cdot n$.

Алгоритм **ChkHard** в группах G_p $(p > 0)$ для каждого такого уравнения $c \cdot X[m,n] = \Sigma$ (из тех уравнений, для которых это не было сделано раньше), генерирует следующую проверку в коде императивной программы:

if Σ **mod** $c \neq 0$ **then goto** L_{fail}; ■

Далее, для каждого такого уравнения $c \cdot X[m,n] = \Sigma$, не конкретизирующего никакую из ранее построенных групп и не совпадающих ни с одной из них, генерируется следующая проверка:

if $\Sigma < c \cdot m$ **or** $\Sigma > c \cdot n$ **then goto** L_{fail}; ■

Если переменная $X[m,n]$ не ограничена сверху $(n = \infty)$, то последняя часть проверки $(\Sigma > c \cdot n)$ опускается.

Для каждого уравнения $c \cdot X = \Sigma$ (где Σ не содержит переменных, X соответствует рефал-переменной V), для которого это не было сделано раньше, в коде императивной программы генерируется уникальная целочисленная переменная len_V и ее инициализация:

int $len_V = \Sigma/c$; ■

Таким образом, для всякой новой переменной V из списка T_{hard} выражение len_V редуцируется во время исполнения сгенерированной программы к длине, которую обязано иметь значение V.

3.13. ChkBnd: алгоритм генерации проверок граничных условий на длины новых переменных. В группе G_p, где $p > 0$, каждое уравнение $c \cdot X = \Sigma$ (за исключением тех, у которых Σ не содержит переменных) выражает переменную X через свободные переменные. Алгоритм **ChkBnd** генерирует необходимые условия, которым должны удовлетворять значения свободных переменных, чтобы система была совместна.

Для каждой свободной переменной X_j, содержащейся в уравнениях группы G, следующий алгоритм порождает выражения императивного языка Min_j и Max_j, редуцируемые после исполнения кода императивной программы, построенного на данный момент, к ограничениям на длину переменной V_j соответственно снизу и сверху.

Шаг 1. Из группы G изымается последнее уравнение Eq вида
$$c \cdot X[m, n] = \Sigma,$$
где Σ содержит переменные, изменяется состояние алгоритма: $G \to G \setminus Eq$. Если таких уравнений в группе G нет, то алгоритм заканчивает работу.

Шаг 2. Для каждого слагаемого вида $d \cdot X_j[k, l]$ из Σ выполняется следующее:

(1) Определяется выражение Σ':
- $\Sigma' = \Sigma - d \cdot X_j - c \cdot X$, если $d < 0$;
- $\Sigma' = c \cdot X - \Sigma + d \cdot X_j$, если $d > 0$.

(2) Генерируется новое выражение для минимума переменной X_j: $Min_j = \mathbf{max}\big(Min_j, \lceil \Sigma_{min}/d \rceil\big)$, где
- Min_j справа от знака равенства — ранее полученное выражение для минимума переменной X_j, либо k, если оно еще не было определено;
- выражение Σ_{min} получается из Σ' заменой переменных $X_p[s, t]$, входящих в Σ' с положительными коэффициентами, на Min_p (либо на s, если Min_p не определено), а переменных $X_q[s, t]$, входящих в Σ' с отрицательными коэффициентами, на Max_q (либо на t, если Max_q не определено).

(3) Генерируется новое выражение для максимума переменной X_j: $Max_j = \mathbf{min}\big(Max_j, \lfloor \Sigma_{max}/d \rfloor\big)$, где
- Max_j справа от знака равенства — ранее полученное выражение для максимума переменной X_j, либо l, если оно еще не было определено;

- выражение Σ_{max} получается из Σ' заменой переменных $X_p[s,t]$, входящих в Σ' с положительными коэффициентами, на Max_p (либо на t, если Max_p не определено), а переменных $X_q[s,t]$, входящих в Σ' с отрицательными коэффициентами, на Min_q (либо на s, если Min_q не определено).

Шаг 3. Сгенерированные новые выражения для минимумов и максимумов свободных переменных запоминаются вместо выражений, полученных ранее. Выполняется переход к шагу 1.

В алгоритме **ChkBnd** выполняется вышеприведенная последовательность шагов для всех групп G_p (где $p > 0$). Далее, для каждой свободной переменной X_j, входящей в группу G_p ($p > 0$), не конкретизирующую никакую из ранее построенных групп и не совпадающую ни с одной из них, генерируется следующий фрагмент кода императивной программы:

if $Min_j > Max_j$ **then goto** L_{fail}; ■

3.14. VMinMax: алгоритм построения ограничений на ведущую переменную группы. Алгоритм **VMinMax** генерирует выражения *Min* и *Max* императивного языка, редуцируемые во время исполнения программы к ограничениям соответственно снизу и сверху на значение ведущей переменной X заданной группы G.

Ведущая переменная X входит в уравнение $c \cdot X[m,n] = \Sigma$, где Σ содержит некоторое множество $\{X_j\}$ свободных переменных. Для каждой из переменных X_j в результате работы алгоритма **ChkBnd** (см. раздел 3.13) построены выражения императивного языка Min_j и Max_j, редуцируемые во время исполнения программы к ограничениям соответственно снизу и сверху на значение X_j.

Выражения *Min* и *Max* строятся следующим образом:

- $Min = \mathbf{max}\big(m, \lceil \Sigma_{min}/c \rceil\big)$;
- $Max = \mathbf{min}\big(n, \lfloor \Sigma_{max}/c \rfloor\big)$;
- Σ_{min} получается из Σ заменой каждой переменной X_j, входящей в Σ с положительным коэффициентом, на Min_j, а каждой переменной X_j, входящей в Σ с отрицательным коэффициентом, на Max_j.
- Σ_{max} получается из Σ заменой каждой переменной X_j, входящей в Σ с положительным коэффициентом, на Max_j, а

каждой переменной X_j, входящей в Σ с отрицательным коэффициентом, на Min_j.

Можно показать, что выражение Σ_{max} редуцируется к конечному (не равному ∞) значению. Таким образом, множество возможных значений ведущей переменной конечно.

3.15. ClA: алгоритм анализа цепочки перестроек. В результате анализа цепочки перестроек S_{cl} строится фрагмент кода императивной программы, осуществляющий некоторый набор из следующих этапов вычисления решения цепочки:

- построение объектного выражения Oe_L, соответствующего левой части Pe_L некоторой перестройки $Pe_L : Pe_R$ (см. раздел 3.15.1). Везде далее, в случае, если существует объектное выражение[14] Oe_L, соответствующее левой части Pe_L некоторой перестройки $Pe_L : Pe_R$, мы будем говорить, что перестройка имеет вид $Oe_L : Pe_R$;
- упрощение перестройки $Oe_L : Pe_1(Pe_2)Pe_3$, в которой длина одного из выражений Pe_1 и Pe_3 может быть вычислена путем снятия скобок с соответствующего терма выражения Oe_L и создания новой вспомогательной перестройки (см. раздел 3.15.3);
- сравнение на равенство старых переменных и объектных подвыражений, встречающихся в правой части Pe_R некоторой перестройки $Oe_L : Pe_R$, соответствующим подвыражениям выражения Oe_L (см. раздел 3.15.5);
- означивание новых переменных, встречающихся в правой части Pe_R некоторой перестройки $Oe_L : Pe_R$, путем взятия соответствующих подвыражений выражения Oe_L (см. раздел 3.15.5);
- итеративный подбор значения одной из новых переменных (см. раздел 3.15.6).

Кроме того, как результат анализа, изменяется состояние компилятора $(S_{cl}, T_{old}, L_{fail})$ (см. раздел 3.2):

- цепочка перестроек S_{cl} упрощается в результате действий, описанных в разделах 3.15.2–3.15.6;

[14] Для определения такого объектного выражения Oe_L компилятор может хранить некую дополнительную информацию о созданных выражениях. В любом случае, если Pe_L является старой переменной V, то значение V во время исполнения программы как раз и будет нужным объектным выражением Oe_L.

- список старых переменных T_{old} пополняется за счет означивания новых переменных и заведения вспомогательных выражений (см. разделы 3.15.1, 3.15.2, 3.15.5 и 3.15.6);
- метка L_{fail}, соответствующая следующей попытке удлинения значения текущей открытой (итерируемой) переменной, изменяется в результате образования цикла подбора значения одной из новых переменных, которая становится текущей открытой (итерируемой) переменной (см. раздел 3.15.6).

На вход алгоритма **ClA** анализа цепочки перестроек S_{cl}, помимо состояния компилятора $(S_{cl}, T_{old}, L_{fail})$, подается упорядоченная тройка (T_{hard}, Min, Max), полученная с помощью алгоритма **EqA** из системы уравнений S_{eq}, соответствующей цепочке S_{cl} (см. раздел 3.7):

- T_{hard} — список переменных, длина значений которых может быть вычислена во время исполнения программы. Если длина некоторого выражения Pe может быть вычислена (то есть это выражение является конкатенацией объектных символов Os, выражений в скобках (Pe') и переменных V, принадлежащих списку T_{hard}), то мы будем называть его *жестким* выражением и обозначать He;
- Min, Max — выражения, редуцируемые во время исполнения к ограничениям соответственно снизу и сверху на длину первой (слева) переменной V из цепочки перестроек S_{cl}, которая не входит в список T_{hard}. Эта переменная станет текущей открытой (итерируемой) переменной, если будет выполнено построение нового цикла (см. раздел 3.15.6).

Общая схема работы алгоритма **ClA** может быть представлена следующим образом:

Шаг 1. Если в цепочке S_{cl} есть перестройка вида

$$Oe_L : Pe_1(Pe_2)Pe_3,$$

где одно из выражений Pe_1 или Pe_3 является жестким, то из содержимого соответствующего скобочного терма выражения Oe_L и выражения Pe_2 (после выписывания необходимых проверок) составляется новая перестройка, которая помещается в цепочку S_{cl} (см. раздел 3.15.3). Изменив таким образом состояние компилятора, алгоритм **ClA** заканчивает работу.

Шаг 2. Для всех перестроек вида $Oe_L : Pe_R$ из цепочки S_{cl}, для всех представлений выражения Pe_R в виде конкатенации выражений $Pe_1 V Pe_2$, где одно из выражений Pe_1 и Pe_2 является жестким, а переменная V является новой s-переменной, выполняется построение проверки того, что терм выражения Oe_L, находящийся в позиции, соответствующей позиции переменной V в выражении Pe_R, является символом (см. раздел 3.15.4). Построенная проверка выводится в код императивной программы, если для данного терма выражения Oe_L это не было сделано раньше.

Шаг 3. Если в цепочке S_{cl} есть перестройка вида

$$Pe_L : Pe_1(Pe_2)Pe_3,$$

где одно из выражений Pe_1 и Pe_3 является жестким, а все переменные, входящие в выражение Pe_L, являются старыми, то формируется объектное выражение Oe_L, соответствующее выражению Pe_L (см. раздел 3.15.1). Перестройка принимает вид $Oe_L : Pe_1(Pe_2)Pe_3$. Выполняется переход к шагу 1.

Шаг 4. Если в цепочке S_{cl} есть перестройка вида

$$Pe_L : Pe_1(Pe_2)Pe_3,$$

где одно из выражений Pe_1 и Pe_3 является жестким, а значения новых переменных выражения Pe_L однозначно определяются, исходя из значений старых переменных, то выполняется означивание новых переменных выражения Pe_L (возможно, требующее построения новых объектных выражений, см. раздел 3.15.2). Означенные переменные добавляются в список T_{old}. Выполняется переход к шагу 1.

Шаг 5. Если в цепочке S_{cl} есть перестройка вида $Oe_L : He_1 Pe_2 He_3$, где одно из выражений He_1 и He_3 не пусто, то в коде императивной программы строятся проверки на равенство старых переменных и объектных подвыражений выражений He_1 и He_3 соответствующим подвыражениям выражения Oe_L. Новые переменные, содержащиеся в выражениях He_1 и He_3, получают значения и добавляются в список T_{old}. Исходная перестройка заменяется на перестройку $Oe'_L : Pe_2$, где Oe'_L получается из Oe_L отбрасыванием слева и справа выражений, равных соответственно He_1 и He_3 (см. раздел 3.15.5). Выполняется рекурсивный переход к шагу 5.

Шаг 6. Если в цепочке S_{cl} есть перестройка вида $Pe_L : He_1 Pe_2 He_3$, где одно из выражений He_1 и He_3 не пусто, а значения новых переменных выражения Pe_L однозначно определяются, исходя из значений старых переменных, то выполняется означивание новых переменных выражения Pe_L (см. раздел 3.15.2) и формирование объектного выражения Oe_L, соответствующего выражению Pe_L (см. раздел 3.15.1). Перестройка принимает вид $Oe_L : He_1 Pe_2 He_3$. Выполняется переход к шагу 5.

Шаг 7. Если в цепочке S_{cl} есть перестройка $Pe_L : Pe_R$, где все переменные, входящие в выражение Pe_L, являются старыми, то формируется объектное выражение Oe_L, соответствующее выражению Pe_L (см. раздел 3.15.1). Перестройка принимает вид $Oe_L : Pe_R$. Выполняется рекурсивный переход к шагу 7.

Шаг 8. Если цепочка S_{cl} не пуста, то выполняется построение оператора цикла (**for**) в императивной программе, соответствующего циклу подбора значения первой (слева) переменной первой перестройки цепочки (см. раздел 3.15.6). Эта переменная добавляется в список T_{old}, метка L_{fail} получает новое значение. Изменив таким образом состояние компилятора, алгоритм заканчивает работу.

Шаг 9. Цепочка перестроек S_{cl} пуста. Алгоритм заканчивает работу.

В следующих разделах подробнее рассматриваются упомянутые выше фрагменты алгоритма **ClA**.

3.15.1. *Построение объектного выражения, соответствующего левой части одной из перестроек.* Если левая часть Pe_L перестройки $Pe_L : Pe_R$ состоит только из объектных выражений и старых переменных, то можно построить новое объектное выражение Oe_L, соответствующее выражению Pe_L.

С помощью следующего рекурсивного алгоритма по образцовому выражению Pe, не содержащему новых переменных, выписывается выражение E императивного языка, редуцируемое во время исполнения программы к объектному выражению Oe, которое соответствует выражению Pe.

Шаг 1. Если выражение Pe имеет вид $Pe_1 Pe_2$, следует выполнить построение выражений императивного языка E_1 и E_2 для выражений соответственно Pe_1 и Pe_2, и вернуть их конкатенацию: $E_1 ++ E_2$.

Шаг 2. Если выражение Pe имеет вид (Pe'), следует выполнить построение выражения императивного языка E' для выражения Pe' и взять его в скобки: (E').

Шаг 3. Если выражение Pe является объектным символом или старой переменной, следует просто вернуть выражение Pe.

После того, как с помощью данного алгоритма построено выражение императивного языка E_L, соответствующее выражению Pe_L, генерируется фрагмент кода императивной программы, обеспечивающий заведение новой вспомогательной переменной, служащей для обращения к объектному выражению Oe_L: $\boxed{\textbf{expr}\ V_\diamond = E_L\,;\ \blacksquare}$, где $V_\diamond$ — новое, неиспользуемое имя переменной.

Переменная $V_\diamond$ заносится в список старых переменных T_{old}.

Перестройка $Pe_L : Pe_R$ заменяется на следующую: $V_\diamond : Pe_R$.

3.15.2. *Формирование значений новых переменных, принадлежащих левой части какой-либо перестройки.* Построение объектного выражения, соответствующего левой части перестройки $Pe_L : Pe_R$, возможно лишь в том случае, когда выражение Pe_L не содержит новых переменных. Данная часть алгоритма служит для ответа на вопрос о том, можно ли, исходя из текущего набора старых переменных, сформировать значения всех новых переменных выражения Pe_L, и в случае, когда ответ положительный, для построения фрагмента кода императивной программы, вычисляющего эти значения.

Следующий рекурсивный алгоритм генерирует фрагмент кода императивной программы, осуществляющий построение значения новой переменной V, а также всех объектных выражений и значений других новых переменных, необходимых для этого. Если, исходя из текущего набора старых переменных T_{old} и имеющихся перестроек S_{cl}, формирование значения переменной V невозможно, то работа алгоритма заканчивается неудачей.

Шаг 1. Если переменная V не принадлежит списку жестких переменных T_{hard}, то работа алгоритма заканчивается неудачей.

Шаг 2. Рассматриваются *одна за другой* все перестройки вида

$$Pe_L : Pe_1\ V\ Pe_2,$$

где по крайней мере одно из выражений Pe_1 и Pe_2 является жестким[15].

[15] Напомним, для жестких выражений He через $|He|$ обозначают выражение императивного языка, которое во время исполнения программы редуцируется к значению длины He. Компилятор строит это выражение, опираясь на тот факт,

(1) Если уже построено объектное выражение Oe_L, соответствующее выражению Pe_L, то генерируется один из следующих фрагментов императивного кода, означивающих переменную V:
   - $\boxed{V = Oe_L\langle|Pe_1|, |V|\rangle;\ \blacksquare}$,
     если Pe_1 является жестким выражением;
   - $\boxed{V = Oe_L\langle|Oe_L| - |Pe_2| - |V|, |V|\rangle;\ \blacksquare}$,
     если Pe_2 является жестким выражением.

   Переменная V заносится в список старых переменных T_{old}. Алгоритм (успешно) заканчивает работу.

(2) Если выражение Pe_L не содержит новых переменных, то формируется соответствующее ему объектное выражение Oe_L (см. раздел 3.15.1), затем выполняются все те же действия, что и в пункте 1.

(3) Если выражение Pe_L содержит новые переменные V_j, то для каждой из этих переменных запускается алгоритм формирования значения.
   - Если удалось сформировать значения всех переменных V_j, то далее выполняются действия пункта 2.
   - Если же хотя бы для одной из переменных V_j работа алгоритма заканчивается неудачей, то весь императивный код, построенный в процессе формирования значения переменной V, забывается. Переменные, получившие значения в процессе работы алгоритма, вновь удаляются из списка старых переменных T_{old}.

     После этого выполняются действия шага 2 для *следующей* перестройки вида $Pe_L : Pe_1 V Pe_2$, где по крайней мере одно из выражений Pe_1 и Pe_2 является жестким.

Шаг 3. Если ни одна из имеющихся перестроек вида $Pe_L : Pe_1 V Pe_2$ не позволила сформировать значение переменной V, то работа алгоритма заканчивается неудачей.

что для всякой жесткой переменной V ранее была построена целочисленная переменная len_V, значение которой равно длине, которую обязано иметь значение переменной V (см. раздел 3.12).

Для того чтобы выражение Pe_L не содержало новых переменных, достаточно запустить приведенный алгоритм для всех новых переменных, содержащихся в Pe_L. Если для какой-то из этих переменных работа алгоритма закончится неудачей, то сформировать ее значение, исходя из текущего состояния компилятора, невозможно. В этом случае следует удалить фрагмент кода императивной программы, полученной во время работы алгоритма, и отменить произведенные изменения состояния компилятора.

3.15.3. *Построение новой перестройки из содержимого скобок одной из старых перестроек.* Если в цепочке S_{cl} есть перестройка $Oe_L : Pe_R$, в которой Pe_R имеет вид:

- $He_1(Pe_2)Pe_3$, то
  (1) генерируется фрагмент кода императивной программы, проверяющий, что соответствующий терм Oe_L является выражением в скобках:

```
if is_symbol(Oe_L[|He_1|])
then goto L_fail
■
```

  (2) перестройка заменяется на три (идущие в указанном порядке):

$$Oe_L : He_1(Pe_2)Pe_3 \quad \to \quad \begin{aligned} Oe_L\langle 0, |He_1|\rangle &: He_1, \\ *Oe_L\big[|He_1|\big] &: Pe_2, \\ Oe_L\langle |He_1|+1, |Oe_L|-|He_1|-1\rangle &: Pe_3. \end{aligned}$$

- $Pe_1(Pe_2)He_3$, то
  (1) генерируется фрагмент кода императивной программы, проверяющий, что соответствующий терм Oe_L является выражением в скобках:

```
if is_symbol(Oe_L[|Oe_L| - |He_3| - 1])
then goto L_fail
■
```

  (2) перестройка заменяется на три (идущие в указанном порядке):

$$Oe_L : Pe_1(Pe_2)He_3 \ \to \ \begin{aligned} Oe_L\langle 0, |Oe_L|-|He_3|-1\rangle &: Pe_1, \\ *Oe_L\big[|Oe_L|-|He_3|-1\big] &: Pe_2, \\ Oe_L\langle |Oe_L|-|He_3|, |He_3|\rangle &: He_3. \end{aligned}$$

Заметим, что в обоих случаях разбиение перестройки не меняет порядок следования новых переменных в цепочке S_{cl}. Поэтому такое

преобразование не меняет порядок вхождения подстановок, вычисляемых генерируемым кодом, в решение цепочки S_{cl} (см. теорему 4.3).

3.15.4. *Построение проверок того, что объектные термы, соответствующие новым s-переменным, являются символами.* При выполнении цепочки перестроек, необходимо проверять, что новым s-переменным присваиваются только объектные термы, являющиеся символами.

Для этого на шаге 2 алгоритма **ClA** для каждой новой s-переменной V из правой части Pe_R каждой перестройки $Oe_L : Pe_R$, для каждого представления выражения Pe_R в виде конкатенации трех выражений $Pe_1 V Pe_2$, где одно из выражений Pe_1 и Pe_2 является жестким, выполняется одно из следующих действий:

- если выражение Pe_1 является жестким, то генерируется следующий фрагмент кода императивной программы:

  if is_parenth$(Oe_L\big[|Pe_1|\big])$ **then goto** L_{fail}; ■

- если выражение Pe_2 является жестким, то генерируется следующий фрагмент кода императивной программы:

  if is_parenth$(Oe_L\big[|Oe_L| - |Pe_2| - 1\big])$
  then goto L_{fail}
  ■

Каждая проверка выводится в код императивной программы лишь в том случае, если она не была построена алгоритмом **ClA** ранее.

3.15.5. *Отщепление жестких выражений.* Если в цепочке S_{cl} есть перестройка вида $Oe_L : He_1 Pe_2 He_3$, то известные части выражений He_1 и He_3 необходимо проверить на равенство соответствующим подвыражениям Oe_L.

Прежде всего, перестройка разбивается на три новые:

$$Oe_L : He_1 Pe_2 He_3 \quad \rightarrow \quad \begin{cases} Oe_L\big\langle 0, |He_1|\big\rangle : He_1, \\ Oe_L\big\langle |He_1|, |Oe_L| - |He_1| - |He_3|\big\rangle : Pe_2, \\ Oe_L\big\langle |Oe_L| - |He_1| - |He_3|, |He_3|\big\rangle : He_3. \end{cases}$$

Далее, над первой и третьей полученными перестройками поочередно производятся следующие действия:

Шаг 1. Если перестройка пуста, то алгоритм заканчивает работу.

Шаг 2. Иначе перестройка имеет вид $Oe : Pt\, He$.

- Если образцовый терм Pt является объектным символом или старой переменной, то строится фрагмент кода

императивной программы, осуществляющий сравнение Pt с соответствующим подвыражением Oe:

$$\textbf{if } Pt \neq Oe\langle 0, |Pt|\rangle \textbf{ then goto } L_{fail}\textbf{;} \ \blacksquare$$

- Если же образцовый терм Pt является новой переменной, то она получает значение:

$$Pt = Oe\langle 0, |Pt|\rangle\textbf{;} \ \blacksquare$$

и добавляется в список старых переменных T_{old}.

Шаг 3. Перестройка заменяется на следующую: $Oe\langle |Pt|, |He|\rangle : He$, после чего выполняется переход к шагу 1.

Исходная перестройка в цепочке S_{cl} заменяется на оставшуюся, непроверенную часть: $Oe_L\langle |He_1|, |Oe_L| - |He_1| - |He_3|\rangle : Pe_2$.

3.15.6. *Построение цикла подбора значения новой переменной.* Из цепочки S_{cl} выбирается первая перестройка. В результате работы шага 7, она имеет вид $Oe_L : Pe_R$.

В результате работы шага 5, выражение Pe_R имеет вид $V\,Pe'_R$, где V — новая переменная, не входящая в список T_{hard}, ее длину требуется подбирать.

Ограничения на длину V дают выражения Min и Max, построенные на шаге 5 алгоритма **EqA** анализа системы уравнений S_{eq}, соответствующей цепочке перестроек S_{cl} (см. раздел 3.7).

Генерируется следующий фрагмент кода императивной программы:

```
int len_V
for (len_V = Min; len_V ⩽ Max; len_V++) {
  V = Oe_L⟨0, len_V⟩
  ■
L_next:
}
```

Здесь L_{next} — это новая метка. Переменная V заносится в список T_{old} старых переменных, она становится текущей открытой (итерируемой) переменной. Метка L_{fail} принимает значение L_{next}.

4. Обоснование алгоритма компиляции цепочки перестроек

4.1. Корректность. Следующие теоремы показывают, что построенный с помощью алгоритма **CSC** фрагмент кода императивной

программы вычисляет решение заданной цепочки перестроек в порядке следования подстановок слева направо.

Теорема 4.1. *Построенный фрагмент кода императивной программы вычисляет множество подстановок σ_i, каждая из которых принадлежит решению цепочки S_{cl}.*

Предположим, что в результате работы построенной императивной программы была получена подстановка σ, не принадлежащая решению цепочки S_{cl}.

Это значит, что в цепочке S_{cl} есть перестройка $Pe_L : Pe_R$, такая, что Pe_L/σ не совпадает с Pe_R/σ.

Однако этого не может быть. Совпадение каждого терма правой части Pe_R с соответствующим термом левой части Pe_L каждой перестройки $Pe_L : Pe_R$ из цепочки S_{cl} обеспечивается на шаге 5 алгоритма **ClA** анализа цепочки перестроек (см. раздел 3.15).

Теорема 4.2. *Любая подстановка σ, принадлежащая решению цепочки S_{cl}, будет вычислена построенным фрагментом кода императивной программы.*

Изначально, любая подстановка

$$\sigma = \{V_j \to Oe_j \mid V_j - \text{новая переменная цепочки } S_{cl}\}$$

рассматривается как потенциально принадлежащая решению цепочки S_{cl}.

Алгоритм **EqA** анализа системы уравнений S_{eq}, соответствующей цепочке S_{cl}, строит только условия на подстановки σ_i *необходимые* для того, чтобы σ_i принадлежали решению S_{cl} (см. теорему 3.2 и алгоритм **EqA**, раздел 3.7).

Все условия, которые строит алгоритм **ClA** анализа цепочки перестроек S_{cl}, опираются на прямое сравнение левых и правых частей перестроек (см. раздел 3.15). Соответственно, выполнение этих условий для подстановки σ также *необходимо*, чтобы σ принадлежала решению S_{cl}.

Таким образом, подстановка, принадлежащая решению цепочки S_{cl}, обязана удовлетворять всем выписанным условиям. Однако построенный фрагмент кода императивной программы осуществляет полный перебор всех подстановок, удовлетворяющих выписанным условиям. Следовательно, любая подстановка из решения цепочки S_{cl} будет в какой-то момент вычислена.

Теорема 4.3. *Подстановки σ_i вычисляются построенным фрагментом кода императивной программы в порядке следования* слева направо.

Сформулируем и докажем вспомогательную лемму.

Лемма 4.4. *Если $Pe_L : Pe_R$ — перестройка из цепочки S_{cl}, такая, что выражение Pe_L не содержит новых переменных, а длина любой новой переменной выражения Pe_R однозначно определяется, исходя из текущих значений длин старых переменных, то решение цепочки S_{cl} не изменится, если данную перестройку $Pe_L : Pe_R$ перенести на первое место цепочки.*

Из условия леммы следует, что значение каждой новой переменной выражения Pe_R однозначно определяется, исходя из текущих значений старых переменных.

Это значит, что (неупорядоченное) множество подстановок, составляющих решение цепочки S_{cl}, не изменится при переносе перестройки $Pe_L : Pe_R$ в начало цепочки.

Предположим, что поменяется порядок следования подстановок в решении. То есть существуют две такие подстановки σ_1 и σ_2, что $\sigma_1 < \sigma_2$ в решении цепочки S_{cl}, и $\sigma_2 < \sigma_1$ в решении цепочки, полученной переносом перестройки $Pe_L : Pe_R$ в начало.

Значит, существуют две такие переменные V_1 и V_2, что:

- $V_1/\sigma_1 < V_1/\sigma_2$,
- $V_2/\sigma_1 > V_2/\sigma_2$,
- V_1 находится левее, чем V_2 в цепочке S_{cl},
- V_1 находится правее, чем V_2 в цепочке S_{cl}, с перенесенной в начало перестройкой $Pe_L : Pe_R$.

Как мы видим, переменная V_2 входит в выражение Pe_R — только так она может оказаться левее V_1 после переноса перестройки.

Однако, мы уже знаем, что значение каждой новой переменной V_j выражения Pe_R определяется однозначно, то есть не может существовать двух подстановок σ_1 и σ_2, присваивающих какой-либо из переменных V_j разные значения.

Итак, порядок следования подстановок в решении цепочки S_{cl} также не поменяется при переносе перестройки $Pe_L : Pe_R$ в начало цепочки. Лемма доказана.

Приведенная лемма утверждает, что решение цепочки перестроек не зависит от того, в каком порядке вычисляются значения переменных, которые определяются параметрами (старыми переменными) цепочки однозначно.

Иными словами, если не меняется порядок подбора значений открытых переменных, то не поменяется и решение цепочки.

Заметим теперь, что построенный фрагмент кода императивной программы осуществляет перебор значений открытых переменных в порядке *слева направо*: цикл всегда строится по самой левой открытой переменной, причем ее значение изменяется от меньшего к большему (см. раздел 3.15.6).

Следовательно, предложенный алгоритм компиляции действительно строит фрагмент кода императивной программы, вычисляющий решение цепочки перестроек, соответствующее сопоставлению слева направо.

4.2. Эффективность. Пусть дана цепочка перестроек

$$S_{cl} = \langle Pe_L^1 : Pe_R^1, \ldots, Pe_L^n : Pe_R^n \rangle ,$$

такая, что выражения Pe_R^i не содержат скобочных термов[16]. В каждой из перестроек сумма длин объектных выражений и переменных на верхнем уровне выражения Pe_L^i, должна быть равна сумме длин объектных выражений и переменных, входящих в выражение Pe_R^i. Зафиксируем значения старых переменных, входящих в S_{cl}, и рассмотрим эти равенства как систему R_{eq} уравнений относительно *действительных* переменных Y (т. е. $Y \in \mathbb{R}$), соответствующих длинам новых переменных $V[m,n]$, где $m \neq n$ (будем обозначать такие переменные через $Y[m,n]$).

Теорема 4.5. *Пусть цепочка S_{cl} такова, что при некоторых значениях старых переменных выполнено одно из следующих условий:*

- *система R_{eq} несовместна;*
- *система R_{eq} имеет единственное решение, но оно не целочисленно;*

[16] Все обсуждаемые теоремы легко обобщаются на цепочки перестроек любого вида, которые либо сводятся к цепочкам без скобок в правой части, либо к цепочкам, где скобки не влияют на вычисление части решения, относящейся к переменным верхнего уровня.

- *система R_{eq} имеет единственное решение, которое присваивает некоторой переменной $Y[m,n]$ значение, лежащее вне отрезка (луча) $[m,n]$.*

Тогда построенный фрагмент кода императивной программы определит, что при данных значениях старых переменных цепочка S_{cl} несовместна, не сравнивая на равенство никакие объектные выражения.

Пусть $[G_0, G_1, \ldots, G_k]$ — группы уравнений, построенные по цепочке S_{cl} с помощью алгоритмов **EqB** и **EqG** (разделы 3.5 и 3.9, соответственно).

Если все равенства группы G_0 выполнены, то система R_{eq} будет иметь по крайней мере одно решение (достаточно зафиксировать произвольные значения свободных переменных в группах $G_1, \ldots, G_k$). Таким образом, несовместность системы R_{eq} (а следовательно, и цепочки S_{cl}) будет определена при проверке выполнения равенств группы G_0 (см. алгоритм **ChkEq**, раздел 3.11).

Если какая-то из групп $G_1, \ldots, G_k$ содержит хотя бы одну свободную переменную, то система R_{eq} либо несовместна, либо имеет более одного решения (действительно, если есть какой-то набор значений переменных, удовлетворяющий системе R_{eq}, то ей будет удовлетворять и любой другой набор значений, отличающийся от первого только на свободных переменных групп $G_1, \ldots, G_k$). Поэтому, если система R_{eq} имеет единственное решение, то его целочисленность и удовлетворение граничным условиям будут проверены при выполнении кода, построенного с помощью алгоритма **ChkHard** (раздел 3.12).

Теорема 4.6. *Если цепочка S_{cl} такова, что при некоторых значениях старых переменных система R_{eq} имеет единственное решение, это решение целочисленно, и значение каждой переменной $Y[m,n]$ принадлежит отрезку (лучу) $[m,n]$, то построенный фрагмент кода императивной программы вычислит решение цепочки S_{cl} (при данных значениях старых переменных), сравнив при этом не более, чем $C \cdot \Sigma$ объектных термов. Здесь Σ — сумма длин значений всех старых переменных из S_{cl}, а C — константа, не зависящая от значений старых переменных из S_{cl}.*

Пусть $[G_0, G_1, \ldots, G_k]$ — группы уравнений, построенные по цепочке S_{cl} с помощью алгоритмов **EqB** и **EqG** (разделы 3.5 и 3.9, соответственно).

Так как система R_{eq} имеет единственное решение, то ни одна из этих групп не содержит свободных переменных. Это значит, что все переменные из цепочки S_{cl} попадут на шаге 3 алгоритма **EqA** (см. раздел 3.7) в список **THard** жестких переменных. Поэтому вызванный далее алгоритм **ClA** (см. раздел 3.15) завершит компиляцию цепочки перестроек S_{cl}, построив код для сравнения не более, чем Σ' объектных термов, где Σ' — сумма длин верхних уровней всех перестроек: $\Sigma' = \sum_{i=1}^{q} |Pe_L^i|$, где q — число перестроек в цепочке S_{cl}.

Осталось заметить, что длины новых жестких переменных выражаются через длины старых переменных с помощью линейных уравнений. Поэтому и длины перестроек выражаются через длины старых переменных линейно. А это значит, что существует такая константа C, что $\Sigma' \leqslant C \cdot \Sigma$.

Теорема 4.7. *Пусть цепочка S_{cl} такова, что при некоторых значениях старых переменных все решения системы R_{eq} параметризуются значением ровно одной свободной неизвестной. Тогда построенный фрагмент кода императивной программы будет в цикле подбирать значение Oe некоторой переменной V из S_{cl}, причем будут рассматриваться только такие значения Oe, что система R'_{eq}, соответствующая (при данных значениях старых переменных) цепочке $S_{cl}/\{V \to Oe\}$, будет иметь решение, которое каждой переменной $Y[m,n]$ присваивает значение из отрезка (луча) $[m,n]$.*

Пусть $[G_0, G_1, \ldots, G_k]$ — группы уравнений, построенные по цепочке S_{cl} с помощью алгоритмов **EqB** и **EqG** (разделы 3.5 и 3.9, соответственно).

Поскольку решения системы R_{eq} параметризуются значением одной свободной неизвестной, то одна из групп G_p ($p > 0$) содержит одну свободную переменную X. Значения всех переменных во всех остальных группах определяются однозначно.

Все (кроме X) переменные X_j группы G_p выражаются через X: $X_j = \Sigma_j(X)$. Алгоритм **ChkBnd** (раздел 3.13) строит такие ограничения на значение Oe' переменной X, что соответствующие значения всех переменных X_j лежат в допустимых пределах:

$$\forall j \ \min(X_j) \leqslant \Sigma_j(Oe') \leqslant \max(X_j)\,,$$

где через $\min(X_j)$ и $\max(X_j)$ для переменной $X_j[m,n]$ обозначены соответственно числа m и n.

Алгоритм **VMinMax** (раздел 3.14) выбирает такие пределы изменения ведущей переменной группы G_p, чтобы выполнялись полученные ограничения для свободной переменной X. Таким образом, при каждом рассматриваемом значении ведущей переменной, граничные условия для всех остальных переменных будут автоматически выполнены.

5. Возможные пути улучшения алгоритма компиляции

Рассмотренный в данной работе алгоритм компиляции цепочки перестроек использует анализ длин для сужения области перебора при поиске решения цепочки. Однако, используется не вся информация, которую можно извлечь из соотношений на длины.

Если R_{eq} — система уравнений (соответствующая цепочке перестроек S_{cl}) относительно переменных $Y \in \mathbb{R}$, как описано в разделе 4.2, то использованию всей информации о соотношениях длин соответствует поиск всех целочисленных решений системы R_{eq}, таких, что каждой переменной $Y[m, n]$ присваивается значение из отрезка (луча) $[m, n]$.

5.1. Уточнение границ перебора. Если решения системы R_{eq} параметризуются более, чем одной свободной неизвестной, то построенный алгоритмом **CSC** фрагмент кода императивной программы будет перебирать не только такие решения системы R_{eq}, которые присваивают каждой переменной $Y[m, n]$ значение из отрезка (луча) $[m, n]$. Это связано с тем, что свободные переменные, входящие совместно в выражение для некоторой связанной переменной, могут ограничивать значения друг друга. Алгоритм **EqA** не генерирует код для соответствующего анализа.

Рассмотрим процесс компиляции следующей цепочки:

$$
\begin{aligned}
&e_x : e_1\, e_2\, e_3\, e_4\, e_5\, e_6,\\
&e_5 : e_2\, e_y,\\
&e_6 : e_3\, e_y,\\
&e_z : e_4\, e_5\, e_6
\end{aligned}
$$

Здесь e_x, e_y и e_z — старые переменные, а $e_1, \ldots, e_6$ — новые. По такой цепочке перестроек будет построена следующая (единственная)

группа G_1 уравнений на длины (см. алгоритм **EqG**, раздел 3.9):

$$\begin{cases} X_1 = A_x + 2 \cdot A_y - A_z - X_5 - X_6 \\ X_2 = X_5 - A_y \\ X_3 = X_6 - A_y \\ X_4 = A_z - X_5 - X_6 \end{cases}$$

Анализируя последовательно (снизу вверх) полученные уравнения, строятся (см. алгоритм **ChkBnd**, раздел 3.13) следующие ограничения на длины свободных переменных:

$$(1)\ \begin{cases} 0 \leqslant X_5 \leqslant A_z \\ 0 \leqslant X_6 \leqslant A_z \end{cases}$$

$$(2)\ \begin{cases} 0 \leqslant X_5 \leqslant A_z \\ A_y \leqslant X_6 \leqslant A_z \end{cases}$$

$$(3)\ \begin{cases} A_y \leqslant X_5 \leqslant A_z \\ A_y \leqslant X_6 \leqslant A_z \end{cases}$$

$$(4)\ \begin{cases} A_y \leqslant X_5 \leqslant \min(A_z,\ A_x + A_y - A_z) \\ A_y \leqslant X_6 \leqslant \min(A_z,\ A_x + A_y - A_z) \end{cases}$$

Таким образом, в код императивной программы будет выведена проверка того, что $A_y \leqslant \min(A_z,\ A_x + A_y - A_z)$, и затем будет построен цикл по переменной e_1, длина значения которой будет изменяться от $\max(0,\ A_x + 2 \cdot A_y - A_z - 2 \cdot \min(A_z,\ A_x + A_y - A_z))$ до $A_x - A_z$ (см. алгоритм **VMinMax**, раздел 3.14).

Если предположить, что $A_x = 200$, $A_y = 100$, и $A_z = 150$, то длина значения переменной e_1 будет изменяться от 0 до 50. Но ясно, что при любых значениях длины переменной e_1, получающаяся цепочка будет несовместна, так как длина каждой из переменных e_5 и e_6 ограничена снизу числом 100, а сумма этих длин ограничена сверху числом 150.

Проблема в том, что свободные переменные X_5 и X_6 входят в последнее уравнение группы одновременно, однако ограничения на них генерируются независимо. В данном конкретном случае достаточно было бы после получения на свободные переменные нетривиальных ограничений снизу, еще раз вычислить ограничения на них сверху, исходя из последнего уравнения.

5.2. Аккуратный перебор целочисленных решений. Алгоритм **EqA** не делает никаких попыток выкинуть из рассмотрения те области значений переменных, в которых не могут находиться целочисленные решения системы R_{eq}.

Между тем, даже несложный анализ в этом направлении может дать значительное сокращение области перебора. Например, рассмотрим реализацию функции `AreSymmetric` из раздела 1.1.3, изображенную на рисунке 4. Легко видеть, что выражение

$$\boxed{(A_y + A_x - 2 \cdot X_1) \ \textbf{mod} \ 2 \neq 0}$$

справедливо тогда и только тогда, когда справедливо выражение

$$\boxed{(A_y + A_x) \ \textbf{mod} \ 2 \neq 0},$$

которое не зависит от значения текущей открытой (итерируемой) переменной. Это условие достаточно проверить один раз перед исполнением цикла. Если оно окажется ложным, то цикл не должен исполняться вообще.

Далее, можно заметить, что условие целочисленности позволяет подбирать значения длин итерируемых переменных не подряд, а с определенным шагом.

Например, рассмотрим перестройку $e_x : e_1\, e_2\, e_2$, где e_x — старая, а e_1 и e_2 — новые переменные. Ясно, что длину переменной e_1 следует подбирать с шагом 2.

Ограничения на очередное значение длины итерируемой переменной могут быть получены и другими способами (например, с помощью анализа объектных выражений, входящих в перестройку), и здесь возникает интересная возможность для комбинации метода анализа длин с другими методами сужения области перебора при поиске решений перестроек.

Благодарности. За плодотворные обсуждения по теме данной работы авторы благодарны участникам проекта реализации системы программирования Рефал Плюс: Л. В. Парменовой, С. М. Пономаревой и А. Ф. Слепухину. Особые благодарности рефал-активистам, без чьих усилий не удалось бы заполнить данными Таблицу 1: А. А. Владимирову, Ан. В. Климову, Арк. В. Климову, А. П. Немытых и М. Ю. Потанину.

Список литературы

[1] Turchin V. F. Refal-5: Programming Guide and Reference Manual. — Holyoke: New England Publishing Co., 1989, http://shura.botik.ru/refal/book/html/, на русском языке: http://www.refal.ru/rf5_frm.htm. ↑(document), 1, 5.2

[2] Гурин Р. Ф., Романенко С. А. Язык программирования Рефал Плюс. — М.: Интертех, 1991, с. 183. ↑(document), 1, 2.1, 2.2, 5.2

[3] Климов Анд. В., Романенко С. А., Турчин В. Ф. Компилятор с языка Рефал. — М.: ИПМ им. М.В.Келдыша, 1972, с. 74. ↑

[4] Климов Анд. В., Романенко С. А., Турчин В. Ф. Теоретические основы синтаксического отождествления в языке Рефал: Препринт 13. — М.: ИПМ им. М.В.Келдыша, 1973, с. 65. ↑

[5] Романенко С. А. Реализация Рефала-2. — М.: ИПМ им. М.В.Келдыша, 1987, с. 191. ↑(document), 1, 5.2

[6] Романенко С. А.. 1978. *Машинно-независимый компилятор с языка рекурсивных функций*, Диссертация на соискание уч.степени к.ф.-м.н., М.. ↑

[7] Абрамов С. М., Романенко С. А. Представление объектных выражений массивами при реализации языка Рефал: Препринт 186. — М.: ИПМ им. М.В.Келдыша, 1988, с. 27. ↑1

[8] Романенко С. А. Компиляция Рефал-программ в виртуальный код, 1999, http://shade.msu.ru/refal.msu.ru/docs/rbvcomp.ps.gz. ↑

[9] *Реализация языка программирования Рефал Плюс за счет прямой компиляции в императивный язык*: Веб-сайт проекта, ИПС РАН, 2003, http://skif.pereslavl.ru/refal. ↑1, 3, 9

[10] Klimov Ark. V. Refal-6, 2003, http://www.refal.org/~arklimov/refal6. ↑(document), 1, 5.2

[11] Klimov Ark. V., Klimov And. V. REFAL — JAVA. The programming language Refal implemented on top of the Java2 Platform, 2003, http://www.refal.org/~arklimov/refal6/refal-j.htm. ↑(document), 1, 5.2

[12] Абрамов С. М., Орлов А. Ю., Парменова Л. В., Пономарева С. М., Слепухин А. Ф. *Новый подход к реализации системы программирования Рефал Плюс* // Международная конференция «Программные системы: теория и приложения». — Переславль-Залесский: Физматлит, 2004. ↑3, 9

Институт программных систем Российской академии наук

Исследовательский центр мультипроцессорных систем ИПС РАН

S. M. Abramov, A. Y. Orlov. *Compilation of Refal pattern matching to imperative languages.* (in russian.)

Abstract. Pattern matching is one of the main operation in Refal language. This paper describes new approach in compilation of refal pattern matching into an imperative language. The approach allows to essentially reduce search space of variants during pattern matching by cutting a lot of variants for which positive result of pattern matching is impossible. Optimizations described in this paper are clear enough but produce essential win in efficiency. However they were not used in Refal implementations up to now.

The authors are supposed that the reader is familiar to Refal. Only basic concepts of Refal common for all Refal dialects [1, 2, 5, 10, 11] are discussed in the paper.