Ордена Ленина
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В. Келдыша
Академии наук СССР

С.М. Абрамов, С.А. Романенко

ПРЕДСТАВЛЕНИЕ ОБЪЕКТНЫХ ВЫРАЖЕНИЙ МАССИВАМИ ПРИ РЕАЛИЗАЦИИ ЯЗЫКА Р Е Ф А Л

Препринт № 186 за 1988г.

Москва

ОРДЕНА ЛЕНИНА
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
ИМ. М. В. КЕЛДЫША АН СССР

С. М. Абрамов, С. А. Романенко

Представление объектных выражений массивами при реализации языка Рефал

Москва
1988

УДК 681.3.06

При реализации языка Рефал обрабатываемые выражения обычно представляются в виде двусвязных списков. Благодаря этому обеспечивается быстрая конкатенация выражений. В то же время, такое представление приводит к необходимости реально копировать выражения, если значение одной и той же переменной используется несколько раз. Трудоемкость копирования не позволяет, в ряде случаев, использовать функциональный стиль программирования на Рефале. В работе предлагается использовать альтернативное представление выражений, которое позволяет заменить реальное копирование выражений на копирование указателей. При этом, однако, становится более трудоемкой конкатенация выражений.

КЛЮЧЕВЫЕ СЛОВА И ФРАЗЫ: Рефал, Лисп, обработка символьной информации, язык программирования, списковая память, сборка мусора, динамическое распределение памяти.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ. ПРОБЛЕМА КОПИРОВАНИЯ

Почти у всякого, кто учился программированию на Рефале, в душе• запечатлелся полумистический, суеверный ужас перед копированием объектных выражений. Во многих пособиях по программированию на Рефале красочно описаны опасности, сопряженные с копированием и предлагаются различные приемы, позволяющие без этого копирования обойтись [БэР 77], [КлР 86,87], [Кор 86], [Тур 71].

К сожалению, использование этих приемов, не только затемняет рефал-программы, но и, как мы вскоре увидим, вынуждает использовать скорее императивный, чем функциональный стиль программирования на Рефале.

Рассмотрим следующий пример. Допустим, что нам надо определить на Рефале функцию SUBST, обращение к которой должно иметь следующий вид:

<SUBST ($\mathcal{E}_{tab}$) $\mathcal{E}_e$>

где $\mathcal{E}_e$ некоторое объектное выражение, а $\mathcal{E}_{tab}$ "таблица перекодировки", которая представляет собой последовательность упорядоченных пар вида

($\mathcal{S}_1$ $\mathcal{E}_1$) ($\mathcal{S}_2$ $\mathcal{E}_2$) ($\mathcal{S}_n$ $\mathcal{E}_n$)

где $\mathcal{S}_i$ некоторые символы, а $\mathcal{E}_i$ некоторые выражения. Функция SUBST должна просмотреть выражение $\mathcal{E}_e$ и для каждого символа $\mathcal{S}$, входящего в выражение $\mathcal{E}_e$, посмотреть в таблицу $\mathcal{E}_{tab}$. Если таблица содержит пару вида ($\mathcal{S}_i$ $\mathcal{E}_i$), для которой $\mathcal{S} = \mathcal{S}_i$, то символ $\mathcal{S}$ в выражении $\mathcal{E}_e$ должен быть заменен на $\mathcal{E}_i$, а если такой пары не существует, символ $\mathcal{S}$ должен остаться без изменения. Например:

<SUBST ((A X X X)(B Y Y Y)) A B C (A C B)() B> →
X X X Y Y Y C (X X X C Y Y Y) () Y Y Y

Написать на Рефале определение функции SUBST не составляет особого труда:

<SUBST t_t >
<SUBST t_t s_x e_y> <LOOKUP t_t s_x> <SUBST t_t e_y>
<SUBST t_t (e_x) e_y> = (<SUBST t_t e_x>) <SUBST t_t e_y>
<LOOKUP (e_a (s_x e_v) e_b) s_x> e_v
<LOOKUP (e_t) s_x> s_x

Это определение функции SUBST, однако, в процессе работы много раз копирует таблицу перекодировки и поэтому, с точки зрения большинства пособий по Рефалу, никуда не годится. Но его можно "исправить", используя различные приемы программирования. Например, применив трюк "сквозной просмотр выражения", мы получаем следующее определение функции SUBST:

<SUBST t_t e_x> <THRU t_t (()) e_x ()>
<THRU (e_a (s_x e_v) e_b) (e_p (e_q)) s_x e_y (e_r)>
  <THRU (e_a (s_x e_v) e_b) (e_p (e_q e_v)) e_y (e_r)>
<THRU (e_t) (e_p (e_q)) s_x e_y (e_r)>
  <THRU (e_t) (e_p (e_q s_x)) e_y (e_r)>
<THRU (e_t) (e_p) (e_x) e_y (e_r)>
  <THRU (e_t) (e_p ()) e_x ((e_y) e_r)>
<THRU (e_t) (e_p (e_q) (e_x)) ((e_y) e_r)>
  <THRU (e_t) (e_p (e_q (e_x))) e_y (e_r)>
<THRU (e_t) ((e_q)) ()> e_q

В новом определении функции SUBST для сквозного просмотра выражения используются два стека.

Нетрудно заметить, что второе определение функции SUBST является, по существу, императивным: во время работы не порождается ни вложенных, ни "параллельных" функциональных скобок. Таким образом, "связь по управлению" преобладает над "связью по данным" (в результате чего, например, при исполнении программы на многопроцессорной реализации не будет происходить распараллеливание вычислений). В то же время, первое определение функции SUBST вполне функционально по духу и допускает глубокое распараллеливание.

1. ПРЕДСТАВЛЕНИЕ ВЫРАЖЕНИЙ ДВУСВЯЗНЫМИ СПИСКАМИ

Ясно, что отсутствие ограничений на использование копирования, как правило, позволяет получать более короткие и естественные программы. В чем же тогда причина столь широко распространенной и стойкой неприязни к копированию? Чтобы понять это, следует рассмотреть способ представления выражений в памяти компьютера, используемый в распространенных реализациях Рефала.

В большинстве реализаций Рефала используется представление объектных выражений в виде двусвязных списков, звенья которых имеют структуру, показанную на рис. 1.

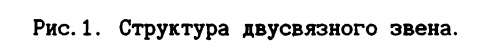

Рис. 1. Структура двусвязного звена.

Поля **prev** и **next** используются для связывания звеньев в линейную последовательность: **prev** содержит ссылку на предшествующее звено, а **next** – на следующее.

Основной смысл использования двусвязных списков в том, что конкатенация двух выражений сводится к установке двух указателей: поля **next** в последнем звене первого выражения и поля **prev** в первом звене второго выражения. Таким образом, время выполнения конкатенации не зависит от размеров выражений.

Поле **tag** содержит признаки, определяющие смысл информации, содержащейся в поле **info**.

Можно использовать поля **tag** и **info** различными способами, при этом получаются различные представления выражений, которые мы условно будем называть "классическим" и "компромиссным".

В случае "классического" представления, каждое звено соответствует символу или скобке. Каждая левая скобка содержит в поле **info** ссылку на парную правую скобку и наоборот (рис. 2 и рис. 3). Списки такой структуры использовались уже в середине 60-х годов при реализации системы AMBIT [Кри 66].

Ясно, что в случае "классического" представления время копирования выражения пропорционально его размерам.

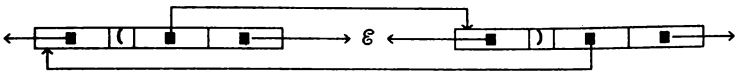

Рис. 2. "Классическое" представление терма (ε).

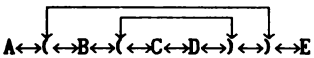

Рис. 3. "Классическое" представление выражения
A (B (C D)) E

"Классическое" представление используется во многих реализациях Рефала [БзР 77], [КРТ 72], [Ром 87а], [ФОТ 69].

В случае "компромиссного" представления, каждое звено соответствует терму (т. е. символу или выражению, заключенному в скобки). Если звено представляет выражение, заключенное в скобки, оно содержит в поле info ссылку на другое звено, именуемое "головой". Начало и конец содержимого скобок присоединены к голове. Таким образом, содержимое скобок вместе с головой образуют кольцевую структуру (рис. 4 и рис. 5).

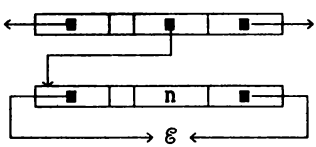

Рис. 4. "Компромиссное" представление терма (ε).

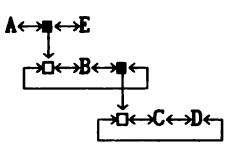

Рис. 5. "Компромиссное" представление выражения
A (B (C D)) E

На одну и ту же головы могут ссылаться несколько звеньев. Таким образом, при копировании выражений достаточно копировать

только нулевой уровень выражений, а вместо копирования содержимого скобок достаточно копировать ссылки на головы. Поле **info** в голове может использоваться в качестве счетчика числа ссылок на эту голову. Списки такой структуры использовались еще в начале 60-х годов в системе SLIP [Вей 63].

2. ПРЕДСТАВЛЕНИЕ ВЫРАЖЕНИЙ МАССИВАМИ

"Компромиссное" представление выражений лишь частично решает проблему копирования. Во-первых, нулевой уровень выражения все равно приходится копировать. Во-вторых, в случае выражения, заключенного в скобки, реального копирования удается избежать только в том случае, если содержимое скобок копируется целиком (тогда достаточно скопировать ссылку на голову). Если же потребовалось скопировать только часть содержимого скобок, реального копирования не избежать.

Пусть, например, функция LR определена следующим образом:

<LR e_x> <LEFT e_x> <RIGHT e_x>

<LEFT t_a e_p> (e_p)

<RIGHT e_q t_b> (e_q)

Теперь вычислим

<LR A B C D> → (B C D) (A B C)

Оказывается, что мы не можем использовать куски выражения A B C D для построения термов (B C D) и (A B C), поскольку невозможно включить подвыражение B C сразу в две линейные последовательности с помощью ссылок в полях **prev** и **next**.

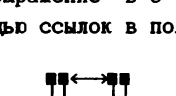

Рис. 6. Представление выражения (B C D)(A B C) посредством двойных ссылок.

Можно, однако, преодолеть эту трудность, изменив представление скобок. А именно, следует представить пару скобок

в виде одного звена, и поместить в это звено два указателя: на начало и на конец содержимого скобок (рис. 6). Таким образом мы избавимся от звеньев-голов и получим возможность повторного использования общих частей выражений.

Возникает, однако, вопрос: а зачем нам теперь, собственно, нужны поля **prev** и **next**? Раньше они использовались для того, чтобы свести конкатенацию выражений к изменению содержимого этих полей. Но как только мы разрешили ссылаться на произвольные части выражения из нескольких мест, оказывается, что при конкатенации выражений все равно необходимо копировать нулевой уровень выражений-операндов. Поэтому теперь лучше вообще отказаться от этих полей. Так мы приходим к "массивному" представлению выражений (рис. 7 и рис. 8).

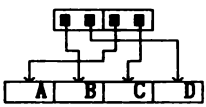

Рис. 7. "Массивное" представление выражения
(B C D)(A B C)

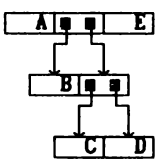

Рис. 8. "Массивное" представление выражения
A (B (C D)) E

В случае "массивного" представления, все термы, составляющие выражение, располагаются в одномерном массиве. (Элементы этого массива мы, по старой памяти, будем называть звеньями.)

Каждое звено (рис. 9) содержит левое информационное поле li, правое информационное поле ri, и однобитовый признак br, который определяет смысл полей li и ri.

Если br=1, то звено представляет выражение, заключенное в скобки. Если содержимое скобок - непустое выражение, li содержит ссылку на первое звено, а ri - на последнее звено этого

выражения. Если же содержимое скобок пустое выражение, то li=ri=0.

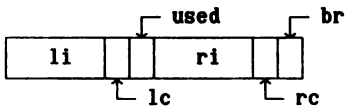

Рис. 9. Структура звена в случае представления массивами.

Если br=0, то звено представляет символ. В этом случае li и ri содержат информацию о символе (которая нас пока не будет интересовать).

Кроме того, в звене есть еще три однобитовых поля: used, lc и rc. Их назначение будет описано позднее.

Если $\mathcal{E}$ объектное выражение, то количество символов и скобок, входящих в $\mathcal{E}$, будем называть размером этого выражения. Если $\mathcal{E} = \mathcal{T}_1\ \mathcal{T}_2 \quad \mathcal{T}_k$, где $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_k$ - объектные термы, то число k будем называть длиной выражения $\mathcal{E}$. Например, выражение "A (B C) ()" имеет размер 7 и длину 3.

Легко видеть, что "массивное" представление позволяет свести копирование выражения к копированию пары указателей. При этом время копирования не зависит от размеров выражения.

В то же время, при конкатенации двух выражений требуется создать новый массив, длина которого равна сумме длин конкатенируемых выражений. Таким образом, время конкатенации пропорционально сумме длин выражений.

3. ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ

В случае использования массивов для представления выражений возникает необходимость динамического захвата связных участков памяти произвольного размера. Поэтому представляется разумным использовать следующие принципы организации памяти.

Под динамическую память следует выделить один связный участок памяти (рис. 10). Часть этого участка должна использоваться для хранения объектных выражений (ее мы будем называть "кучей"), а другая часть - для стека (если он нужен в данной реализации).

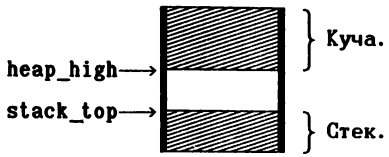

Рис. 10. Организация динамической памяти.

Мы будем считать, что куча расположена в начале динамической памяти (т. е. в младших адресах), а стек в конце (т. е. в старших адресах). При желании, впрочем, можно было бы поместить кучу в старшие адреса, а стек - в младшие.

Указатель heap_high содержит адрес последнего звена кучи, а указатель stack_top - адрес вершины стека.

Память, расположенная между кучей и стеком, не занята. Поэтому, если в процессе работы требуется захватить связный кусок памяти, то используются звенья, расположенные сразу вслед за кучей. При этом указатель heap_high продвигается вперед на нужное число звеньев.

4. СБОРКА МУСОРА

4.1. НЕОБХОДИМОСТЬ СБОРКИ МУСОРА

Если допускаются ссылки на одни и те же выражения из разных мест, необходима сборка мусора [Хен 83], т. е. автоматическое выяснение, какие звенья нужны (из-за того, что на них есть ссылки), а какие не нужны (ибо не них ссылок больше не осталось).

Кроме того, при захвате памяти может возникать ситуация, когда свободная память есть, но она разбита на мелкие куски, и из-за этого нельзя найти связный участок нужного размера. Поэтому, при использовании массивного представления требуется не только сборка мусора, но и уплотнение, при котором все нужные куски памяти плотно придвигаются друг к другу, в результате чего все куски свободной памяти объединяются в один кусок.

При сборке мусора используются однобитовые поля used, lc и gc (в промежутках между сборками мусора они должны быть равны

нулю).

Поле used используется, чтобы помечать звенья, которые нужно сохранить, поскольку они использованы для хранения нужных выражений. Назначение полей lc (left cluster) и rc (right cluster) будет описано ниже.

4.2. ПРИНЦИПЫ РАБОТЫ АЛГОРИТМА СБОРКИ МУСОРА

Алгоритм сборки мусора состоит из двух основных этапов: маркировка и уплотнение.

Маркировка состоит в том, что отыскиваются все звенья, которые еще "нужны", и для всех таких звеньев в поле used устанавливается значение 1. Эти звенья, таким образом, становятся "помеченными". Исходной информацией для маркировки является некоторое множество "заведомо нужных" звеньев. Алгоритм маркировки помечает "заведомо нужные" звенья, а также все звенья, которые прямо или косвенно достижимы из них.

Уплотнение состоит в том, что помеченные звенья переносятся на новое место, при этом корректируются все ссылки на эти звенья из других помеченных звеньев. Уплотнение — самый сложный этап сборки мусора.

Основная трудность состоит в том, что на одно и то же звено может иметься неограниченное количество ссылок из других звеньев, и все эти ссылки требуется откорректировать.

В дальнейшем, для краткости, все помеченные звенья, ссылающиеся на некоторое звено, будем называть "начальниками" этого звена, а само это звено – "подчиненным".

Чтобы исправить адреса, требуется знать в момент корректировки адреса всех звеньев-начальников, а также как старый (до перемещения), так и новый (после перемещения) адрес звена-подчиненного. Поэтому представляется естественным следующее решение проблемы: для каждого звена следует связать всех его начальников в односвязный список (именуемый в дальнейшем "связкой" этого звена). Тогда в момент перенесения этого звена на новое место можно найти всех его начальников и выполнить коррекцию ссылок.

К сожалению, реализация этого плана наталкивается на следующее противоречие: звенья-начальники сами должны быть

перенесены на новое место. А как только будет передвинуто на новое место хотя бы одно звено-начальник - связка "порвется".

Это противоречие можно преодолеть следующим образом: сначала построить связки всех звеньев, а затем провести коррекцию адресов, не передвигая самих звеньев. При этом связки будут использованы и исчезнут, причем для каждого звена придется вычислить его новый адрес и использовать этот адрес для коррекции ссылок. После этого возникнет состояние, когда все звенья будут находиться на старых местах, но все ссылки на звенья будут указывать уже на их новые места. После этого можно будет передвигать звенья на новое место.

Этот план наталкивается на еще одну трудность: в каждом звене, у которого br=1, находится на одна ссылка, а две. Обе эти ссылки надо корректировать. Кроме того, получается, что одно звено может быть начальником сразу для двух звеньев и должно входить сразу в две связки. От этих трудностей можно избавиться очень простым способом. Достаточно заметить, что длина выражения (понимаемая как количество составляющих его термов) не меняется при перемещении этого выражения в другое место в памяти. Поэтому для любого звена, содержащего в полях li и ri ссылки на начало и конец выражения, разность содержимого ri и li, не меняется в результате перемещения выражений при уплотнении.

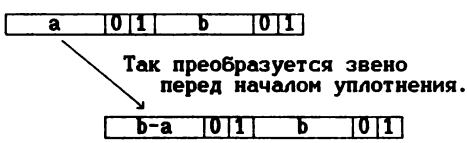

Рис. 11. Замена двойных ссылок на одинарные.

Таким образом, если в каждом звене, представляющем пару скобок, заменить значение поля li на разность значений полей li и ri, окажется, что значение поля li в процессе уплотнения корректировать не нужно, и количество указателей, подлежащих коррекции уменьшится в два раза (рис. 11).

Такое преобразование звеньев будем называть "заменой двойных ссылок на одинарные", а обратное преобразование (которое тоже легко выполнимо) будем называть "заменой одинарных ссылок на двойные".

Теперь мы можем при построении связок использовать поле li для ссылки на первое звено связки, а поле ri для связывания всех звеньев-начальников в цепочку. При этом нам потребуются поля lc и rc: lc=1 будет означать, что связка данного звена не пуста, а rc=1 будет означать, что ri содержит ссылку на другое звено, входящее в ту же связку, что и данное (рис. 12).

Следует обратить внимание, что информация, которая находилась в поле li основного звена до построения связки, должна быть где-то сохранена, ибо она будет использована после перемещения звеньев. Для сохранения этой информации используется поле ri последнего звена связки, поскольку это поле уже не требуется для хранения ссылки на следующее звено связки.

Рис. 12. Структура связки.

Таким образом, у нас получается алгоритм сборки мусора, состоящий из следующих шести фаз.

(1) Маркировка.
(2) Замена двойных ссылок на одинарные.
(3) Построение связок.
(4) Корректировка ссылок.
(5) Передвижение звеньев.
(6) Замена одинарных ссылок на двойные.

4.3. ШЕСТИПРОХОДНЫЙ АЛГОРИТМ СБОРКИ МУСОРА

Теперь приведем описание шестипроходного алгоритма сборки мусора на языке Си [КРФ 85].

При этом изобразим память компьютера в виде массива

звеньев, размера MEMORY_SIZE. Будем считать, что размер адреса памяти составляет ADDR_SIZE битов, а участок памяти, отведенный под хранение выражений, простирается от адреса MEMORY_LOW (включительно) до адреса MEMORY_HIGH (исключительно).

```
#define ADDR_SIZE 14
#define MEMORY_SIZE 1000

typedef unsigned int address;
typedef unsigned int bool;

/* Описание терма. */
struct term
  {
  address  li:   ADDR_SIZE;
  bool     lc:   1;
  bool     used: 1;
  address  ri:   ADDR_SIZE;
  bool     rc:   1;
  bool     br:   1;
  };

struct term t[MEMORY_SIZE];

#define MEMORY_LOW  1
#define MEMORY_HIGH 1000
```

Сборку мусора выполняет функция garb_coll(), которая вызывает функцию mark_term().

Функция mark_term() маркирует все термы, достижимые из терма, адрес которого эта функция получает в качестве параметра.

Чтобы не загромождать описание алгоритма, считаем, что перед началом маркировки имеется единственное заведомо нужное звено: а именно звено, имеющее адрес MEMORY_LOW. В реальной реализации алгоритма, конечно, таких звеньев может быть и несколько.

```
void
mark_term(m)
  address m;
  {
  address k, k_max;
  if (!t[m].used)
    {
    t[m].used = 1;
    if (t[m].br && t[m].ri != 0)
      {
      k_max = t[m].ri;
      for (k=t[m].li; k<=k_max; k++)
        mark_term(k);
      }
```

```
    }
  return;
  }

void
garb_coll()
  {
  address  m, k, new_place;

  /* Маркировка. */
  /* Для простоты считаем, что первоначально нужен */
  /* только терм, находящийся в начале кучи.       */
  mark_term(MEMORY_LOW);

  /* Замена двойных ссылок на одинарные. */
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used && t[m].br && t[m].ri != 0)
      t[m].li = t[m].ri-t[m].li;

  /* Построение связок. */
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used)
      if (t[m].br && t[m].ri != 0)
        {
        /* Нашли пару скобок с непустым содержимым. */
        /* Заносим ее в связку. */
        k = t[m].ri;
        t[m].ri = t[k].li;
        t[m].rc = t[k].lc;
        t[k].li = m;
        t[k].lc = 1;
        }

  /* Корректировка ссылок. */
  new_place = MEMORY_LOW;
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used)
      {
      /* Корректировка всех ссылок на терм. */
      while (t[m].lc)
        {
        /* Очередной элемент связки. */
        k = t[m].li;
        t[m].li = t[k].ri;
        t[m].lc = t[k].rc;
        /* Теперь исправляем ссылку из терма k. */
        t[k].ri = new_place;
        t[k].rc = 0;
        }
      new_place++;
      }

  /* Перемещение термов на новое место. */
  new_place = MEMORY_LOW;
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used)
      {
      /* Сдвиг терма на новое место. */
```

```
      t[m].used = 0;
      t[new_place].li   = t[m].li;
      t[new_place].lc   = t[m].lc;
      t[new_place].used = 1;
      t[new_place].ri   = t[m].ri;
      t[new_place].rc   = t[m].rc;
      t[new_place].br   = t[m].br;
      new_place++;
      }

  /* Сброс пометки и замена одинарных ссылок на двойные. */
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used)
      {
      t[m].used = 0;
      if (t[m].br && t[m].ri != 0)
        t[m].li = t[m].ri - t[m].li;
      }

  return;
  }
```

4.4. УСТРАНЕНИЕ ЛИШНИХ ПРОХОДОВ

Алгоритм сборки мусора, описанный выше, выполняет в процессе уплотнения 5 проходов по памяти, занятой звеньями. В действительности, однако, количество проходов можно сократить. Алгоритм, который получается в результате этого, имеет более запутанную структуру, но работает несколько быстрее.

В первую очередь заметим, что замену двойных ссылок на одинарные можно выполнять по ходу дела в процессе маркировки.

Точно так же, замену одинарных ссылок на двойные можно выполнять во время передвижения звеньев.

Таким образом, у нас получается трехпроходный алгоритм уплотнения. Оказывается, однако, что проход, выполняющий корректировку ссылок, тоже можно устранить, хотя это и требует применения более изощренных приемов. Этот проход нельзя целиком совместить ни с построением связок, ни с передвижением звеньев. Тем не менее, оказывается, что можно всю работу по корректировке ссылок выполнять частично при построении связок, частично при передвижении звеньев.

Основная идея состоит в следующем. Рассмотрим некоторое звено. Будем говорить, что все звенья, у которых адреса меньше, чем у этого звена, расположены "слева" от него, и что все

звенья, у которых адреса больше, чем у этого звена, расположены "справа" от этого звена.

Тогда всех начальников некоторого звена можно поделить на две категории: "левых", т.е. находящихся слева от звена, и "правых", т.е. находящихся справа от звена. Кроме того, если звено является своим собственным начальником, мы будем считать его своим левым начальником.

Во время построения связок движение по памяти происходит слева направо. В тот момент, когда алгоритм доходит до некоторого звена, оказывается, что все его левые начальники уже находятся в связке, поэтому можно сразу же выполнить коррекцию ссылок для всех левых начальников (и удалить при этом их из связки).

Затем движение вправо продолжается. При этом алгоритм найдет всех правых начальников звена и занесет их в связку.

В результате, в конце прохода оказывается, что для каждого звена его связка содержит всех его правых начальников, а для левых начальников коррекция ссылок уже выполнена.

Теперь оказывается, что можно совместить коррекцию ссылок для правых начальников с перемещением звеньев, а именно, перед перемещением очередного звена можно корректировать ссылки для всех его правых начальников (с удалением их из связки). Почему же это не приводит к таким неприятностям, как "разрыв" связок?

Основная тонкость состоит в том, что хотя левые начальники звена перемещаются раньше, чем само звено, это ничему не противоречит, ссылки в них уже скорректированы во время предыдущего прохода. В то же время, правые начальники перемещаются только после перемещения их подчиненного, т.е. после того, как они уже удалены из связки.

4.5. ТРЕХПРОХОДНЫЙ АЛГОРИТМ СБОРКИ МУСОРА

Теперь приведем описание трехпроходного алгоритма сборки мусора на языке Си.

```
void
mark_term(m)
  address m;
  {
  address k, k_max;
  if (!t[m].used)
```

```
    {
    t[m].used = 1;
    if (t[m].br && t[m].ri != 0)
      {
      k     = t[m].li;
      k_max = t[m].ri;
      /* Замена двойной ссылки на одинарную. */
      t[m].li = k_max-k;
      for ( ; k<=k_max; k++)
        mark_term(k);
      }
    }
  return;
  }

void
garb_coll()
  {
  address  m, k, new_place;

  /* Маркировка. */
  mark_term(MEMORY_LOW);

  /* Построение связок. */
  new_place = MEMORY_LOW;
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used)
      {
      /* Корректировка всех ссылок слева на терм. */
      while (t[m].lc)
        {
        /* Очередной элемент связки. */
        k = t[m].li;
        t[m].li = t[k].ri;
        t[m].lc = t[k].rc;
        /* Теперь исправляем ссылку из терма k. */
        t[k].ri = new_place;
        t[k].rc = 0;
        }
      if (t[m].br && t[m].ri != 0)
        {
        /* Нашли пару скобок с непустым содержимым. */
        /* Делаем привязку.                         */
        k = t[m].ri;
        t[m].ri = t[k].li;
        t[m].rc = t[k].lc;
        t[k].li = m;
        t[k].lc = 1;
        }
      new_place++;
      }

  /* Корректировка ссылок справа и       */
  /* перемещение термов на новое место.  */
  new_place = MEMORY_LOW;
  for (m=MEMORY_LOW; m<MEMORY_HIGH; m++)
    if (t[m].used)
      {
```

```
    /* Корректировка всех ссылок на терм справа. */
    while (t[m].lc)
      {
      /* Очередной элемент связки. */
      k = t[m].li;
      t[m].li = t[k].ri;
      t[m].lc = t[k].rc;
      /* Теперь исправляем ссылку из терма k. */
      t[k].ri = new_place;
      t[k].rc = 0;
      }
    /* Сброс пометки и сдвиг терма на новое место.  */
    t[m].used = 0;
    t[new_place].li   = t[m].li;
    t[new_place].lc   = t[m].lc;
    t[new_place].used = t[m].used;
    t[new_place].ri   = t[m].ri;
    t[new_place].rc   = t[m].rc;
    t[new_place].br   = t[m].br;
    /* Для скобок замена одинарной ссылки на двойную. */
    if (t[new_place].br && t[new_place].ri != 0)
      t[new_place].li = t[new_place].ri - t[new_place].li;
    new_place++;
    }

  return;
  }
```

5. ЭФФЕКТИВНОСТЬ РАЗЛИЧНЫХ ПРЕДСТАВЛЕНИЙ ВЫРАЖЕНИЙ

Здесь мы имеем дело с принципом "Тришкина кафтана" [Кры 84]. Две основные операции при работе с выражениями копирование и конкатенация выражений. Добиваясь дешевого копирования, удорожаем конкатенацию, и наоборот.

представление	цена конкатенации	цена копирования
классическое	низкая	высокая
компромиссное	низкая	компромиссная
массивное	высокая	низкая

Рис. 13. Особенности различных представлений выражений.

Конечно, различные представления выражений можно сравнивать, замеряя скорость работы программ на различных реализациях Рефала, использующих различные представления, однако не вполне ясно, как истолковывать результаты таких измерений.

Ведь различные представления подразумевают различный стиль программирования на Рефале. Поэтому, если какая-то программа работает медленно, это может свидетельствовать только о том, что она "плохо написана", а не о том, что используется "плохое" представление выражений.

Измеряя время на искусно подобранных тестах можно получить произвольные результаты как в пользу массивного представления, так и против него. Рассмотрим, например, функцию TT:

```
<TT () t.X>              t.X
<TT s.* e.A) t.X> =      <TT (e.A) (t.X t.X)>
```

Ясно, что если использовать классическое представление, то время работы функции TT зависит от количества звездочек экспоненциально, а если использовать массивное или компромиссное представление линейно. Нетрудно, впрочем, подобрать и противоположные примеры.

6. МЕРЫ ПО СНИЖЕНИЮ ЗАТРАТ НА КОНКАТЕНАЦИЮ

Поскольку "массивное" представление выражений приводит к увеличению стоимости конкатенации выражений, возникает вопрос: а не будут ли все выгоды, полученные за счет удешевления копирования потеряны из-за дополнительных затрат на конкатенацию?

К счастью, оказывается, что в значительном числе случаев конкатенация по сути дела не нужна и ее можно избежать. В следующих разделах мы рассмотрим этот вопрос более подробно.

6.1. РАСПОЗНАВАНИЕ ЧАСТНЫХ СЛУЧАЕВ КОНКАТЕНАЦИИ

Если используется представление выражений массивами, конкатенация двух выражений $\mathcal{E}_1$ и $\mathcal{E}_2$ требует копирования этих выражений на нулевом уровне скобок. При этом, если $\mathcal{E}_1$ имеет длину L_1, а $\mathcal{E}_2$ – длину L_2, то захватывается связный кусок памяти длиной L_1+L_2, и на нем создается массив термов, представляющий выражение $\mathcal{E}_1\mathcal{E}_2$. Те части выражений $\mathcal{E}_1$ и $\mathcal{E}_2$, которые находятся внутри скобок, при этом копировать не надо: достаточно скопировать указатели.

В некоторых случаях, однако, можно уменьшить объем работы, необходимой для выполнения конкатенации.

(1) Если ε_1 пустое выражение, то результатом конкатенации является - ε_2.

(2) Если ε_2 пустое выражение, то результатом конкатенации является - ε_1.

(3) Если массив, представляющий ε_1, уже находится в памяти непосредственно перед массивом, представляющим ε_2, то эти два массива уже фактически являются результатом конкатенации.

(4) Если массив, представляющий ε_1, находится в самом конце кучи, достаточно скопировать только массив, представляющий ε_2, приписав копию к массиву, представляющему ε_1.

На первый взгляд может показаться, что случай 3 очень редок и его не стоит рассматривать особо. В действительности, однако, часто встречаются программы, которые выполняют просмотры выражений, перекладывая выражение терм за темом из одной переменной в другую. В таких случаях конкатенация как раз и сведется к исправлению указателей.

6.2. ФУНКЦИИ С НЕСКОЛЬКИМИ ВХОДАМИ И ВЫХОДАМИ

Довольно часто в программах приходится иметь дело с функциями, имеющими несколько аргументов и порождающими несколько результатов (в дальнейшем, для краткости, количество аргументов функции мы будем называть ее арностью, а количество результатов - ее размерностью).

Между тем, в реализованных версиях Рефала все функции, определяемые и используемые в рефал-программах, должны иметь арность и размерность 1 (т.е. получать ровно один аргумент и выдавать ровно один результат).

При программировании это ограничение приходится преодолевать следующим образом.

Чтобы подать на вход функции FUNC m выражений ε_1, ε_2, ε_m, эти выражения заключают в скобки, конкатенируют и подают

функции в виде одного аргумента следующим образом:

<FUNC ($\mathcal{E}_1$)($\mathcal{E}_2$) ... ($\mathcal{E}_m$)>

Если же функция должна выдать n результатов $\mathcal{R}_1$, $\mathcal{R}_2$, ... $\mathcal{R}_n$, она заключает эти выражения в скобки, конкатенирует и выдает в качестве одного результата:

($\mathcal{R}_1$)($\mathcal{R}_2$) ... ($\mathcal{R}_n$)

Ясно, что ни в том, ни в другом случае, с содержательной точки зрения конкатенация не нужна, и используется только из-за того, что в Рефале нет средств определять функции с несколькими аргументами или результатами. Если же эти средства ввести, необходимость в таких конкатенациях отпадет.

Впрочем, понятия арности и размерности вводить во входной язык рефал-системы не обязательно, ибо существуют методы автоматического повышения арности и размерности функций [Ром 87г], [Ром 88]. Вполне достаточно, чтобы функции имели арность и размерность на уровне промежуточного языка рефал-системы.

6.3. ИСПОЛЬЗОВАНИЕ СКОБОК ДЛЯ СТРУКТУРИЗАЦИИ ДАННЫХ

Конкатенация становится трудоемкой операцией только если она применяется к длинным выражениям (напоминаем, что длина это количество термов в выражении на нулевом уровне скобок). Поэтому, разумно используя скобки, во многих случаях можно избежать возникновения длинных выражений.

Например, предположим, что в программе требуется завести стек, содержащий объектные выражения. Если представить пустой стек в виде пустого выражения, а результат вталкивания выражения $\mathcal{E}_{top}$ в стек $\mathcal{E}_{stack}$ представить в виде

$\mathcal{E}_{stack}$ ($\mathcal{E}_{top}$)

то стек, содержащий выражения $\mathcal{E}_1$, $\mathcal{E}_2$, ... $\mathcal{E}_k$ будет иметь вид

($\mathcal{E}_1$)($\mathcal{E}_2$) ... ($\mathcal{E}_k$)

где вершина стека находится справа. Ясно, что такое

представление может привести к большим затратам на конкатенацию, ибо вталкивание нового элемента в стек реализуется через конкатенацию всего стека с одним термом справа. Однако, если представить результат вталкивания выражения ε_{top} в стек ε_{stack} в виде

$$(\varepsilon_{stack})\ (\varepsilon_{top})$$

то все проблемы, связанные с конкатенацией, отпадут, ибо каждый раз придется конкатенировать не более двух термов.

7. ПРЕДСТАВЛЕНИЕ МАССИВАМИ В УСЛОВИЯХ ВИРТУАЛЬНОЙ ПАМЯТИ

Системы виртуальной памяти имеет смысл использовать только в тех случаях, когда обращения к памяти обладают свойством локальности, т. е. если в течение сравнительно больших интервалов времени адреса, по которым производятся обращения к памяти, сконцентрированы в нескольких небольших областях адресного пространства (причем этих областей должно быть не слишком много).

Свойство локальности обычно не выполняется для систем обработки списковых структур данных, поскольку со временем происходит "перемешивание" элементов списка, вследствие которого логически соседние элементы списка перестают быть соседними в адресном пространстве. Кроме того, в системах со сборкой мусора происходит "расползание" списков по обширному адресному пространству, при котором реально используемые звенья списка разделены большими неиспользованными промежутками. В результате, если, например, в основной памяти размещена 1/10 списковой структуры, то из 10 обращений к данным 9 вызывают страничный обмен [БоМ 67].

Были разработаны многочисленные алгоритны сборки мусора с линеаризацией и уплотнением списков [Мар 82], [АбТ 83], назначение которых противодействовать "перемешиванию" и "расползанию" списков.

"Массивное" представление выражений должно лучше соответствовать требованиям работы в условиях виртуальной памяти, чем "классическое" и "компромиссное", поскольку в случае "массивного" представления звенья, представляющие логически

соседние термы выражения, автоматически оказываются соседними и в адресном пространстве. Кроме того, сборка мусора с уплотнением создает тенденцию к уменьшению используемого адресного пространства.

8. ПРОБЛЕМА КОПИРОВАНИЯ И РЕАЛИЗАЦИЯ РАСШИРЕНИЙ РЕФАЛА

Высокая стоимость копирования помешала реализовать "Полный Рефал" [Тур 71], Рефал-4 [Ром 87б, 87в] и другие подобные расширения Рефала. Рассмотрим, например, такую программу на Рефале-4:

```
#func LR
  #  e_X, e_X: t_a e_p, e_X   e_q t_b    (e_p)(e_q)
#end
```

Из этого примера хорошо видно, что в Рефале-4, в отличие от Рефала-2, значение одной и той же переменной может быть проанализировано несколько раз с помощью сопоставления с разными образцами. В данном случае, если вызвать функцию следующим образом:

```
<LR A B C D>
```

окажется, переменная e_X принимает значение "A B C D" Затем это значение анализируется дважды: сопоставляется с образцами $t_a\ e_p$ и $e_q\ t_b$. Получается, что выражение "A B C D" первый раз рассекается на части "A" и "B C D", а второй раз на части "A B C" и "D".

В результате (рис. 14) значения различных переменных "налезают" друг на друга. Поэтому, когда наступает момент, когда требуется построить новое выражение, используя значения переменных, оказывается, что мы не можем попросту разобрать исходное выражение на куски и слепить их в новое выражение: требуется, вообще говоря, предварительно скопировать эти куски, поскольку они могут входить в значения нескольких переменных.

Заметим, что в Базисном Рефале [БзР 77], [Тур 71] или в Рефале-2 [КлР 87, 86], [Ром 87а] такая проблема никогда не возникала, поскольку в этих вариантах Рефала аргумент функции может быть успешно сопоставлен только с одним образцом. Это и

позволяло успешно использовать представление выражений двусвязными списками.

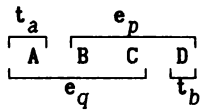

Рис. 14. Взаимное перекрытие значений переменных.

"Язык сборки", который используется в качестве промежуточного языка в реализациях Базисного Рефала и Рефала-2 [БзР 77], [КРТ 72], [Ром 87а], рассчитан на "классическое" представление выражений и мало пригоден для реализации расширений Рефала. Поэтому при реализации Полного Рефала или Рефала-4 более подходящим промежуточным языком может оказаться язык RL [Ром 87г], [Ром 88]. Язык RL может служить промежуточным языком и при реализации других языков, подобных Рефалу, например языка FLAC [Кис 87].

ЗАКЛЮЧЕНИЕ

Предложенное нами "массивное" представление объектных выражений позволяет свести копирование выражений к копированию указателей, что дает возможность более свободно использовать функциональный стиль программирования на Рефале. Кроме того, это представление может служить основой для реализации расширений Рефала, подобных Рефалу-4.

"Массивное" представление более экономно использует память, чем представления, основанные на использовании двусвязных списков, поскольку не требуются поля для ссылок вперед и назад. В этом смысле переход к "массивному" представлению является более радикальным шагом, чем предложенное в работе [МаЭ 87] устранение только одной из двух ссылок: ссылки назад.

В условиях виртуальной памяти "массивное" представление является более удовлетворительным, чем основанное на двусвязных списках, поскольку логически соседние термы оказываются соседними и в адресном пространстве, а сборка мусора с уплотнением способствует уменьшению используемого адресного пространства.

ЛИТЕРАТУРА

[АбТ 83] С. М. Абрамов, С. Б. Трубицына. Алгоритм линеаризующей сборки мусора, минимизирующей число страничных обменов // Материалы Всесоюзного семинара "Оптимизация и преобразование программ". Новосибирск: ВЦ СО АН СССР, 1983, т. 2, с. 5-12.

[БзР 77] Базисный Рефал и его реализация на вычислительных машинах. - М.: ЦНИПИАСС, 1977. - 258 с.

[БоМ 67] D. Bobrow, D. Murphy. Structure of a Lisp system using two-level storage. Comm. ACM, Vol. 10, No. 3, (March 1967), pp. 155-159.

[Вей 63] J. Weizenbaum. Symmetric list processor. Comm. ACM, Vol. 6, No. 9 (September 1963), pp. 524-536.

[Кис 87] В. Л. Кистлеров. Принципы построения языка алгебраических вычислений FLAC. - Препринт, М.: Институт проблем управления, 1987. - 39 с.

[КлР 86] Ан. В. Климов, С. А. Романенко. Система программирования Рефал-2 для ЕС ЭВМ. Описание библиотеки функций. М.: ИПМ им. М. В. Келдыша АН СССР, 1986, препринт N 200. - 38 с

[КлР 87] Ан. В. Климов, С. А. Романенко. Система программирования Рефал-2 для ЕС ЭВМ. Описание входного языка. М.: ИПМ им. М. В. Келдыша АН СССР, 1987. - 52 с.

[Кор 86] А. В. Корлюков. Методические указания по спецкурсу "Прикладная алгебра" для студентов специальности 2013. Гродно: Гродненский государственный университет, 1986. 57 с.

[Кри 66] C. Christensen. On the implementation of AMBIT, a language for symbol manipulation. Comm. ACM, Vol. 9, No. 8 (August 1966), pp. 570-573.

[КРТ 72] Ан. В. Климов, С. А. Романенко, В. Ф. Турчин. Компилятор с языка Рефал. - М.: ИПМ АН СССР, 1972. - 74 с.

[КРФ 85] Б. Керниган, Д. Ритчи, А. Фьюэр. Язык программирования Си. Задачи по языку Си: Пер. с англ. М.: Финансы и статистика, 1985. - 279 с.

[Кры 84] И. А. Крылов. Сочинения в двух томах. Том II. М.: Правда, 1984. - с. 82.

[Мар 82] J. Martin. An efficient garbage compaction algorithm. Comm. ACM, Vol. 25, No. 8 (August 1982), pp. 571-572.

[МаЭ 87] Н. Н. Мансуров, Л. К. Эйсымонт. Реализация расширенного языка Рефал на односвязных списках с кольцевыми цепочками. М.: ИПМ им. М. В. Келдыша АН СССР, 1987, препринт N 20. 32 с.

[Ром 87а] С. А. Романенко. Реализация Рефала-2. М.: ИПМ им. М. В. Келдыша АН СССР, 1977. - 191 с.

[Ром 87б] С. А. Романенко. Рефал-4 расширение Рефала-2, обеспечивающее выразимость результатов прогонки. М.: ИПМ им. М. В. Келдыша АН СССР, 1987, препринт N 147. - 27 с.

[Ром 87в] С. А. Романенко. Прогонка для программ на Рефале-4. М.: ИПМ им. М. В. Келдыша АН СССР, 1987, препринт N 211. 19 с.

[Ром 87г] С. А. Романенко. Генератор компиляторов, порожденный самоприменением специализатора, может иметь ясную и естественную структуру. М.: ИПМ им. М. В. Келдыша АН СССР, 1987, препринт N 26. - 35 с.

[Ром 88] С. А. Романенко. Мета-мета-вычисления и специализация программ // Семиотические аспекты формализации интеллектуальной деятельности. Всесоюзная школа-семинар г. Борхоми, 22-30 апреля 1988 г. Тезисы докладов и сообщений. - М.: ВИНИТИ, 1988. - с. 65-68.

[Тур 71] В. Ф. Турчин. Программирование на языке Рефал. М.: ИПМ АН СССР, 1971, препринты N 41, N 43, N 44, N 48, N 49.

[ФОТ 69] С. Н. Флоренцев, В. Ю. Олунин, В. Ф. Турчин. Эффективный интерпретатор языка Рефал. - М.: ИПМ АН СССР, 1969, препринт N 29. - 103 с.

[Хен 83] П. Хендерсон. Функциональное программирование. Применение и реализация: Пер. с англ. - М.: Мир, 1983. - 349 с.

Ссылки на издание рекомендуется делать по следующей форме: и.о., фамилия, название, препринт Ин. прикл. матем. им. М.В. Келдыша АН СССР, год, №.

Распространение: препринты института продаются в магазинах Академкниги г. Москвы, а также распространяются через Библиотеку АН СССР в порядке обмена.

Адрес: СССР, 125047, Москва-47, Миусская пл. 4, Институт прикладной математики им. М.В. Келдыша АН СССР, ОНТИ.

The references should be typed by the following form: initials, name, title, preprint, Inst.Appl.Mathem., the USSR Academy of Sciences, year, N(number).

Distribution. The preprints of the Keldysh Institute of Applied Mathematics, the USSR Academy of Sciences are sold in the bookstores "Academkniga", Moscow and are distributed by the USSR Academy of Sciences Library as an exchange.

Adress: USSR, I25047, Moscow A-47, Miusskaya Sq.4, the Keldysh Institute of Applied Mathematics, Ac.of Sc., the USSR, Information Bureau.

С.М. Абрамов, С.А. Романенко " Представление объектных выражений массивами при реализации языка Рефал."
Редактор В.С. Штаркман. Корректор Е.Ю. Шведова.

Подписано в печать 24.11.88 г. № Т-19176. Заказ № 452.
Формат бумаги 60X90 1/16. Тираж 250 экз.
Объём 1,2 уч.-изд.л. Цена 15 коп.

055 (02)2

Отпечатано на ротапринтах в Институте прикладной математики АН СССР

Москва, Миусская пл. 4.

Цена 15 коп.