УДК 004.43

Оптимизирующие трансформации списков и деревьев в системе предикатного программирования

Булгаков К.В. (Новосибирский государственный университет),
Каблуков И.В., Тумуров Э.Г. (Институт систем информатики СО РАН),
Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описываются оптимизирующие трансформации для операций над списками и деревьями в системе предикатного программирования. Кодирование операций представлено набором правил, определяющих замену исходной операции на ее образ в императивном языке. Результатом трансформаций является императивная программа по эффективности сравнимая с написанной вручную.

Ключевые слова: *функциональное программирование, трансформации программ, алгебраический тип данных.*

1. Введение

Оперирование указателями является весьма сложной и опасной процедурой в императивном программировании. Показателем такой сложности является чрезвычайная трудность дедуктивной верификации программ, оперирующих указателями, например, для алгоритма реверсирования списка [15].

В языке предикатного программирования P [12] нет указателей, серьезно усложняющих программу. Вместо указателей используются объекты алгебраических типов: списки и деревья. Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением оптимизирующих трансформаций. Они определяют оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу. Эта оптимизация отлична от классической.

Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной [2, 7];
- замена хвостовой рекурсии циклом;

- подстановка определения предиката на место его вызова;
- кодирование списков и деревьев при помощи массивов и указателей.

Цель данной работы – определить эффективные способы кодирования языковых конструкций с объектами алгебраических типов. Итоговая программа по эффективности должна быть сравнима с программами на императивных языках C или C++.

Во втором разделе дается описание алгебраических типов в системе предикатного программирования. В третьем разделе проводится обзор использования алгебраических типов в других языках программирования. В четвертом разделе описывается реализация трансформации конструкций с алгебраическими типами в императивное расширение языка предикатного программирования P [9]. В пятом разделе описаны правила трансформации для конструкций с алгебраическими типами. В шестом разделе приведены примеры кодирования алгебраических структур и анализ производительности сгенерированного кода по сравнению с кодом, написанным вручную.

2. Предикатное программирование

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P следующего вида:

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

Необязательные конструкции предусловия и постусловия являются формулами на языке исчисления предикатов; они используются для улучшения понимания программ и для дедуктивной верификации [8, 10, 18]. Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

```
<переменная> = <выражение>
{<оператор1>; <оператор2>}
if (<логическое выражение>) <оператор1> else <оператор2>
<имя программы>(<список аргументов>: <список результатов>)
<тип> <пробел> <список имен переменных>
```

Эффективность программы также обеспечивается оптимизацией, реализуемой программистом, на уровне предикатной программы. Для приведения рекурсии к хвостовому виду применяется метод обобщения исходной задачи. Далее обычно открывается возможность проведения серии последующих улучшений алгоритма. Итоговая программа по

эффективности не уступает написанной вручную и, как правило, короче [1, 8, 10, 18]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [14]) оптимизация программы полностью возлагается на транслятор, в частности, обеспечивается автоматическое приведение рекурсии к хвостовому виду. Разумеется, функциональное программирование существенно уступает в эффективности, поскольку даже применением изощренных методов оптимизации невозможно автоматически воспроизвести серию оптимизаций, совершаемых программистом вручную.

Описание типа связывает имя типа с его изображением. Тип может быть параметризованным.

```
<описание типа> ::=
    type <имя типа> [( <параметры типа> )] =
          <изображение типа> | <предописание типа>
```

В языке P алгебраические типы определяются следующей конструкцией:

```
<описание типа> ::= union ( <описание конструкторов> )
<описание конструкторов> ::= <описание конструктора>
                                  [, <описание конструкторов>]*
<описание конструктора> ::= <идентификатор> [( <описание полей> )]
<описание полей> ::= <изображение типа> <идентификатор>
                                  [, <описание полей>]
```

Значением типа объединения является значение одного из конструкторов, перечисленных в списке описаний конструкторов, вместе с набором полей конструктора. Алгебраический тип может быть рекурсивно определяемым.

2.1. Списки

Тип «список» – встроенный алгебраический тип со следующим определением [11]:

```
type list (type T) = union (
    nil,
    cons(T car, list(T) cdr)
);
```

Здесь T – тип элемента списка, nil и cons – конструкторы. Канонический способ работы со списками определяется оператором выбора:

```
switch (s) {
    case nil: <оператор1>
    case cons(head, tail): <оператор2>
};
```

Вхождения переменных head и tail являются определяющими, причем head – начальный элемент списка s, а tail – «хвост» списка s. Оператор выбора эквивалентен следующему оператору:

```
if (nil?(s)) <оператор1> else { { T c = s.car || list(T) y = s.cdr}; <оператор2> };
```

Определены следующие операции со списками:

- s.car – первый элемент списка
- s.cdr – список без первого элемента
- last(s) – последний элемент списка
- s[m] – элемент под номером m
- len(s) – длина списка
- nil?(s) – проверка на пустоту списка
- s == nil – проверка на пустоту списка
- cons?(s) – проверка на непустоту списка
- s != nil – проверка на непустоту списка
- prec(s) – список без последнего элемента
- s + t – конкатенация списков
- s + e – добавление элемента в конец списка
- s[m..n] – вырезка списка от номера m до номера n
- s[m..] – вырезка списка от m до конца

Здесь s – выражение типа список, t – терм типа список, e –выражение типа элемента списка, m, n – выражение типа nat. Элементы списка нумеруются с нуля.

Конструкторы списка:

- nil
- cons(head, tail)
- consLeft (list, m) |
- consRight (list) |
- consRight (list, n)

Здесь nil и cons(head, tail), где head – элемент списка и tail – список, являются стандартными конструкторами в соответствии с определением типа list. Описание специальных конструкторов consLeft и consRight дано в разделе 4.2.

Для изображения типа списка допускается использование следующих типовых термов:

```
list(T)
list(T, L)
```

Здесь T – тип элемента списка, L – максимальная длина списка. Размер памяти, отводимой для переменной типа list(T, L), будет достаточным для размещения L элементов списка.

Для значения списковой переменной не допускается выход значения за границы памяти, отведенной для переменной. Соответствующий контроль возлагается на программиста. Для этой цели предусмотрены следующие конструкции.

```
max_len(s)
store(s)
left_store(s)
resize(s, n)
```

Здесь s – переменная типа список. Значением функции max_len(s) является максимальное число элементов списка, которое можно разместить в массиве для переменной s. Функция store(s) определяет число элементов, которое можно разместить справа от значения s в свободной части памяти. Функция left_store(s) определяет число элементов, которое можно разместить слева от значения s в памяти. Оператор resize(s, n) отводит новую память размера n элементов, переписывает туда значение переменной s, освобождая старую память.

В дополнение к основному режиму, в котором программист полностью контролирует распределение памяти для списковых переменных, следует также предусмотреть режим, задаваемый прагмой, в котором контроль выхода за границы и заказ памяти большего размера реализуется автоматически.

2.2. Строковый тип

Строковый тип string является предопределенным в языке предикатного программирования P [11]. Его определение имеет вид:

```
type string = list(char);
```

Набор конструкций, определенный для списков, применим также и для строк с некоторыми ограничениями. В дополнении к этому в языке P определены строковые константы.

Основным представлением строкового объекта является массив литералов, завершающийся терминальным нулем, причем нуль не входит в значение строки. Иначе говоря, для строкового типа фактически действует следующее определение:

```
type string = subtype( list(char) s: s != nil & last(s) == 0 );
```

Проверка строки s на пустоту реализуется оператором s.car == 0, а не s == nil. В принципе, возможна реализация, в которой конструктор nil кодируется значением из единственного нулевого элемента, однако такое решение приведет к потере эффективности. В итоге, типы list и string несовместимы: со строковым объектом нельзя работать как со

списком, в частности, нельзя подставлять строковый объект параметром типа list. Как следствие, библиотеки для списков неприменимы для строковых объектов.

Для строкового типа, как и для списков, существует возможность указать размер памяти выделяемой для строковых объектов. Для этого используется конструкция string(L), где L количество литер, которые вмещает память значения типа string(L). Для строкового типа также возможно использование следующих конструкций:

```
max_len(s)
store(s)
resize(s, n)
```

Исключается возможность сдвига вправо значения строки относительно начала памяти, т.е значение строки всегда размещается с начала памяти.

Вводятся дополнительные конструкции для строковых объектов. Для определения числа элементов строки s вместо функции len(s) используется length(s). Конструктор nil, распознаватель nil?(s), а также отношения s == nil и s != nil не используются для строковых объектов. В качестве пустой строки используется конструктор empty, значением которого является строка из единственного нулевого элемента.

2.3. Двоичные деревья

Двоичное дерево – дерево, в котором каждая вершина имеет не более двух потомков. Двоичное дерево используется для представления табличных данных и хранения множества данных вместе с их ключами, используемыми для поиска. Основные операции двоичного дерева это добавление нового элемента, удаление существующего элемента и поиск элемента по ключу. Ключи и данные представлены следующими типами:

```
type Tkey(<);
type Tinfo;
```

Для типа ключей Tkey определено отношение линейного порядка «<». Типы Tkey и Tinfo – произвольны и являются параметрами модуля, реализующего АВЛ-деревья. [5, 6, 16].

АВЛ-деревья выбраны в языке предикатного программирования Р для реализации сбалансированных деревьев, хотя в большинстве современных реализаций сбалансированных деревьев используются красно-черные деревья. Данный выбор был основан на результатах работы Performance Analysis of BSTs in System Software [17]

Двоичное дерево представляется алгебраическим типом Tree:

```
type Tree = union (
  leaf,
  node (Tkey key, Tinfo info, Tree left, right)
);
```

Лист дерева соответствует конструктору leaf. Вершина дерева, соответствующая листу, не хранит никакой информации. Конструктор node определяет вершину, не являющуюся листом. Полями конструктора являются ключ key и ассоциированное с ним данное info. Левое и правое поддеревья, исходящие из данной вершины, определяются полями left и right.

3. Обзор реализации списков и строк

Для языков программирования C++, Java, Lisp и других имеются библиотеки для поддержки операций со списковыми и строковыми объектами. Стандартными способами реализации списков являются их реализация в виде односвязных и двусвязных списков.

В C++ существует несколько способов работы со строками. Один из них, это способ унаследованный от языка C, когда строки представляются как char-массивы с терминальным нулем в конце. Для работы с таким представлением используется библиотека cstring. Стандартная библиотека C++ предоставляет более удобный формат работы со строками в объектно-ориентированном стиле — класс string. Тип string представляет собой последовательность символов, где каждый символ может быть получен по его позиции в последовательности. Обычно реализуется при помощи динамического массива, который позволяет произвольный доступ к элементам по индексу.

В Java класс для работы со строками — String. Список операций и внутреннее представление этого класса очень схож с классом string в C++, но есть одно важное отличие. Строки в Java неизменяемые. Такие операции, как получение подстроки или конкатенация, всегда будут возвращать новый объект класса String. Для того, чтобы эффективно реализовать комбинации и модификации уже существующих строк, следует использовать классы StringBuilder и StringBuffer.

В системе предикатного программирования строки кодируются массивами с терминальным нулем, что аналогично реализациям в императивных языках C и C++. Для списков имеется два способа реализации: через массивы и через односвязные списки. Для списков и строк, представляемых массивами, используются разные способы реализации в зависимости от контекста, определяемого с помощью потокового анализа.

4. Реализация трансформаций кодирования операций с алгебраическими типами

При преобразовании сложной иерархической конструкции с объектами алгебраических типов преобразование начинается с вложенных подконструкций.

Для оператора присваивания списковой переменной выполняется определение *вида присваивания* для списковой переменной, и на основании этих данных выбирается преобразование.

4.1. Определение вида присваивания для списковой переменной

В соответствии с формальной семантикой языка P вычисление нового значения списка как результата некоторой операции сопровождается выделением памяти для этого значения. Буквальная реализация этого положения оказалась бы весьма расточительной. Например, при исполнении оператора s = x + y + z предполагается отведение памяти для результатов выражений x + y и (x + y) + z, а также для нового значения переменной s. Если заранее подсчитать длину списка x + y + z и отвести достаточную память для переменной s, то исходный оператор заменяется последовательностью "s = x; s = s + y; s = s + z", исполнение которой не требует дополнительной памяти.

В целях эффективного кодирования списков различаются три вида присваивания списковым переменным:

- **Присваивание вида копирование** реализуется копированием результата спискового выражения в переменную слева от знака равенства. При этом переменная s не участвует в выражении e.
- **Присваивание вида модификация** изменяет значение списковой переменной, добавляя к изначальным данным слева или справа дополнительные элементы списка.
- **Сканирование** не модифицирует значение списковой переменной, осуществляя только анализ списка с продвижением по нему без модификации.

Присваивание вида копирования для оператора s = e реализуется копированием списка, вычисленного выражением e, в память для значения переменной s. Для оператора s = e + d в массив для хранения переменной s копируется значение e и вслед за ним копируется значение d.

Присваивание вида модификации[1] реализует изменение значения, размещаемого памяти для переменной s. Итоговое значение помещается в тот же участок памяти. Оператор s = s + e копирует список (значение выражения e) вслед за значением списка s в памяти. Оператор s = e + s копирует список (значение выражения e) перед значением списка s в памяти. Оператор s = s.car реализует отсечение хвоста списка. Операторы s = s[m..n] и s = s[m..] реализуют сужение значения списка к вырезке исходного значения s. Вместо сдвига значения s в начало массива проводится соответствующая корректировка позиции значения списка внутри памяти для списковой переменной. Операторы s = s.cdr и s = prec(s) также реализуются коррекцией позиции в памяти.

Списковая переменная, все действия с которой реализуют лишь анализ списка с продвижением по нему без его модификации[2] в памяти, называется *переменной сканирования.* Для переменной сканирования s присваивание вида s = y реализуется не копированием значения y, а создание *объекта сканирования*, ассоциированного с переменной y, и присваиванию его переменой s. Операторами сканирования являются s = s.cdr, s = prec(s), s = s[m..n] и s = s[m..]. Их отличие от соответствующих операторов присваивания в режиме модификации в том, что они не модифицируют переменную y. Операторы сканирования являются аналогами итераторов в императивных языках.

4.2. Представление алгебраических типов

Основным способом представления списка является массив.

Для представления значения типа list(T) в императивном расширении используется следующая структура.

```
struct list {
 T *data;
 int max_len;
 int m;
 int n;
}
```

Здесь T – тип элемента списка, max_len – максимальная длина списка, m, n – индексы начала и конца текущей вырезки массива, 0 ≤ m ≤ n ≤ max_len, data – массив данных.

[1] Модификация переменных возможна как в исходной предикатной программе, так и в результате склеивания переменных после проведения трансформации склеивания переменных.

[2] Точнее, допускается модификация отдельных элементов списка без изменения их состава.

Другими возможными альтернативами кодирования списка являются: односвязный список, двунаправленный список, кольцевой список. Представление в виде кольцевого буфера следует считать модификацией представления в виде массива.

Представление списка через односвязный список.

```
struct list {
 list *next;
 T data;
}
```

Здесь T – тип элемента списка, next – указатель на следующий элемент.

Для любых видов списковых выражений возможен такой способ реализации, при котором удается отложить отведения памяти до момента присваивания списковой переменной, находящейся в левой части оператора присваивания. Поэтому отведение памяти далее рассматривается по отношению к списковым переменным.

Оптимальной реализацией является использование одного экземпляра памяти для всех присваиваний одной списковой переменной. Такое возможно, если известно верхнее ограничение L числа элементов списка: list(a, L).

Допустим, отведенная для списковой переменной память есть массив A с индексами в диапазоне 0..N. Тогда значение списковой переменной можно представить вырезкой A[m..n], где $0 \le m \le n \le N$, однако в случае пустого списка $m > n$. В большинстве случаев $m = 0$ и свободное место в памяти остается слева в диапазоне m+1..N. Однако бывают случаи, когда свободную часть памяти надо оставить справа для того, чтобы реализовать присваивание вида s = y + s, как, например, в работе [2], где используется присваивание buf = stf + buf.

Для формирования нестандартного размещения значения списка используется специальные конструкторы consLeft, consRight. Конструктор consLeft(list, m), где m – номер элемента, формирует представление списка s в массиве, сдвинутое на m элементов относительно начала массива. Конструктор вида consRight(list) формирует представление списка s в массиве прижатым вправо, т.е. последний элемент списка находится в конце массива. Конструктор вида consRight(list, n) формирует представление списка s в массиве длины n прижатым вправо. При исполнении оператора s = consRight(list, n) отводится новая память для строковой переменной s; если до присваивания переменная s уже имела некоторое значение, то транслятор с языка P должен обеспечить возврат старой памяти переменной s.

В языке P нет операторов отведения и освобождение памяти для списковых переменных. Вставка в код соответствующих действий реализуется транслятором. Отведение памяти реализуется при первом присваивании переменной. Размер памяти определяется по значению правой части оператора присваивания, если он явно не указан описанием типа.

Далее рассматривается лишь представление списка в виде массива.

Представление значений типа string.

```
struct string {
 char *data;
 int max_len;
 int len;
}
```

Здесь max_len – максимальная длина строки, len – индекс конца текущей строки, len < max_len, по индексу len значение ноль, data – массив данных.

Для строкового типа используется два вида объектов сканирования: один – для сканирования с начала строки, второй – с конца. В первом случае объект сканирования может быть представлен указателем на начальный элемент строки, во втором – дополнительно требуется длина строки, при этом строка, представленная объектом сканирования, нулем не завершается.

Представление значений типа Tree через указатели.

```
struct Tree {
 Tree *left;
 Tree *right;
 Tkey key;
 Tinfo info;
}
```

Здесь left, right – указатели на левое и правое поддеревья, key – ключ типа Tkey, info – значение типа Tinfo.

Представление значений типа Tree через через массивы [3].

```
struct TreeValue {
    Tkey key;
    Tinfo info;
}
struct Tree {
      TreeValue *data;
 int left;
 int right;
}
```

Здесь data – указатель на массив, где размещается дерево, left, right – индексы левого и правого поддеревьев в массиве.

4.3 Потоковый анализ программы

При кодировании рекурсивных структур используется потоковый анализ программы [4], который включает построение графа вызовов, нахождение аргументов и результатов операторов и нахождение живых переменных операторов. Потоковый анализ определяет время жизни переменных: для каждого вхождения переменной определяется, будет значение этой переменной использоваться при дальнейшем исполнении программы или нет. На основании этих данных определяются эффективные способы преобразования операций алгебраических типов.

Алгоритм определения времени жизни переменных реализуется следующим образом. Для каждого оператора определяются преемники – операторы, исполняющиеся сразу после него. Строятся цепочки преемников для каждого оператора. Переменная жива после исполнения оператора, если ее текущее значение используется в одной из цепочек преемников этого оператора.

Предикат isAlive(variable_name) по отношению к текущему оператору истинен, если переменная с именем variable_name является живой переменной после исполнения оператора. Этот предикат используется в правилах трансформации, чтобы избавиться от избыточного копирования данных неживых переменных. Считается, что результаты программы являются живыми для каждого оператора его тела, так как предположительно будут использоваться после завершения исполнения программы.

5. Правила трансформации

5.1. Описание правил трансформации

Правила трансформации имеют следующую структуру:

if(<условия применимости>)
<конструкция на предикатном языка> → <код в императивном расширении>

Условия применимости это логическое выражение. Трансформация применима лишь при истинном значении условия. Переменные, встречающиеся в левой части правила, являются *параметрами* правила. При использовании правила вместо параметра может быть подставлена любая переменная соответствующего типа.

5.2. Правила трансформации списка при кодировании через массив

Напомним представление списка через массив:

```
struct list {
    T *data;
    nat max_len;
    nat m;
    nat n;
}
```

Приведем правила трансформации для основных операций списка при кодировании через массив.

```
list(T) l;          →   list *l = new list();
                        l→max_len = DEFAULT_MAX_LEN;
                        l→data = new T[l→max_len];
                        l→m = 0;
                        l→n = 0;
list(T, L) l;       →   list *l = new list();
                        l→max_len = L;
                        l→data = new T[l→max_len];
                        l→m = 0;
                        l→n = 0;
l = consLeft(y, m); →   delete[] l→data;
                        l→max_len = y→max_len + m;
                        l→data = new T[l→max_len];
                        l→m = m;
                        l→n = len(y) + m;
                        <копирование массива из y в l со сдвигом на m
                        относительно начала>
l = consRight(y);   →   delete[] l→data;
                        l→max_len = y→max_len;
                        l→data = new T[l→max_len];
                        l→n = l→max_len;
                        l→m = l→n – len(y);
                        <копирование из массива y в l прижатым
                        вправо>
l = consRight(y, L); →  delete[] l→data;
                        l→max_len = L;
                        l→data = new T[l→max_len];
                        l→n = l→max_len;
                        l→m = l→n – len(y);
                        <копирование из массива y в l прижатым
                        вправо>
max_len(s);         →   s→max_len;
store(s);           →   s→max_len – s→n;
```

```
left_store(s); →    s→m;
resize(s, n);  →    T *new_data = new T[n];
                    <копируем данные из s→data в new_data>
                    delete[] s→data;
                    s→data = new_data;
                    s→max_len = n;
len(s);        →    s→n – s→m;
nil?(s);       →    s→data == NULL;
cons?(s);      →    s→data != NULL;
```

Для присваивания вида копирования используются следующие правила.

```
s = s1 + .. + sn;   →   nat result_len = len(s1) + … + len(sn);
                        if (s→max_len < result_len) {
                              list(T, result_len) new_list;
                              delete s;
                              s = new_list;
                        }
                        nat shift = 0;
                        <копирование s1 в s со сдвигом на shift
                        относительно начала>
                        shift += len(s1);
                        …
                        <копирование sn в s со сдвигом на shift
                        относительно начала>
```

Для присваивания вида модификация используются следующие правила.

```
s = s + e;     →    if (store(s) >= len(e)) {
                          <копировать e вслед за значением s>
                    } else {
                          resize(s, len(s) + len(e));
                          <копировать e вслед за значением s>
                    }
s = e + s;     →    if (left_store(s) >= len(e)) {
                          <копировать e перед значением s>
                    } else {
                          s = consRight(s, len(s) + len(e));
                          <копировать e перед значением s>
                    }
s = s.car;     →    s→n = s→m + 1;
s = s[m..n];   →    s→m = m;
                    s→n =n;
s = s[m..];    →    s→m = m;
s = s.cdr;     →    s→m = s→m + 1;
s = prec(s);   →    s→n = s→n - 1;
s = last(s);   →    s→m= s→n – 1;
```

Для присваивания вида сканирование используются следующие правила.

```
s = s.cdr;      →    s→m = s→m + 1;
s = prec(s);    →    s→n = s→n - 1;
s = s[m..n];    →    s→m = m;
                     s→n = n;
s = s[m..];     →    s→m = m;
```

5.3. Правила трансформации списка при кодировании через указатели

Напомним представление списка через указатели:

```
struct list {
    list *next;
    T data;
}
```

Приведем правила трансформации для основных операций списка при кодировании через список.

```
list(T) l;        →    list *l = new list();
                       l→next = NULL;
max_len(s);       →    sizeof(nat);
store(s);         →    max_len(s) – len(s);
left_store(s);    →    max_len(s) – len(s);
len(s);           →    nat len = 0;
                       list *tmp = s;
                       while (tmp != NULL) {
                            ++len;
                            tmp = tmp→next;
                       }
nil?(s);          →    s == NULL;
cons?(s);         →    s != NULL;
```

Для присваивания вида копирования используются следующие правила.

```
if(!isAlive(s1) && … && !isAlive(sn))
s = s1 + .. + sn;    →    s = s1;
                          while (s1→next != NULL) {
                                s1 = s1→next;
                          }
                          s1→next = s2;
                          while (s2→next != NULL) {
                                s2 = s2→next;
                          }
                          ….
```

Для присваивания вида модификация используются следующие правила.

```
if(!isAlive(e))
s = s + e;      →   list *tmp = s;
                    while (tmp→next != NULL) {
                        tmp = tmp→next;
                    }
                    tmp→next = e;
if(!isAlive(e))
s = e + s;      →   list *tmp = e;
                    while (tmp→next != NULL) {
                        tmp = tmp→next;
                    }
                    tmp→next = s;
                    s = e;
s = s.car;      →   <освободить память начиная с s→next>
                    s→next = NULL;
s = s[m..n];    →   list *tmp;
                    for (nat j = 0; j < m; ++j) {
                        tmp = s→next;
                        delete s;
                        s = tmp;
                    }
                    nat size = n – m + 1;
                    for (nat j = 0; j < size; ++j) {
                        tmp = tmp→next
                    }
                    <удалить элементы после tmp>
                    tmp→next = NULL;
s = s[m..];     →   list *tmp;
                    for (nat j = 0; j < m; ++j) {
                        tmp = s→next;
                        delete s;
                        s = tmp;
                    }
s = s.cdr;      →   list *tmp = s;
                    s = s→next;
                    delete tmp;
s = prec(s);    →   list *tmp = s;
                    while (tmp→next→next != NULL) {
                        tmp = tmp→next;
                    }
                    delete tmp→next;
                    tmp→next = NULL;
s = last(s);    →   list *tmp = s;
                    while (tmp→next != NULL) {
                        tmp = tmp→next;
                        delete s;
                        s = tmp;
```

```
}
```

Для присваивания вида сканирование используются следующие правила.

```
s = s[m..];   →   for (nat j = 0; j < m; ++j) {
                      s = s→next;
                  }
s = s.cdr;    →   s = s→next;
```

При кодировании списка через указатели существуют дополнительные правила трансформации для парных операций.

```
s = u.car + s; u = u.cdr;  →   list *a = u→next;
                               u→next = s;
                               s = u;
                               u = a;
```

5.4. Правила трансформации для строкового типа

Напомним представление строки:

```
struct string {
    char *data;
    nat max_len;
    nat len;
}
```

Приведем правила трансформации для строкового типа.

```
string(L) s;   →   string *s = new string();
                   s→max_len = L;
                   s→data = new char[s→max_len];
                   s→len = 0;
                   s→data[s→len] = 0;
s = empty;     →   string *s = new string();
                   s→max_len = 1;
                   s→data = new char[s→max_len];
                   s→len = 0;
                   s→data[s→len] = 0;
max_len(s);    →   s→max_len;
store(s);      →   max_len(s) – len(s) - 1;
length(s);     →   s→len;
resize(s, n);  →   char *new_data = new char[n];
                   <копируем данные из s→data в new_data>
                   delete[] s→data;
                   s→data = new_data;
                   s→max_len = n;
s.car == 0;    →   s→data[0] == 0;
```

Для присваивания вида копирование используется следующее правило.

```
s = s1 + .. + sn;   →   nat result_len = length(s1) + … + length(sn);
                        if (s→max_len < result_len) {
                                string(result_len) new_string;
                                        delete s;
                                s = new_string;
                        }
                        nat shift = 0;
                        <копирование s1 в s со сдвигом на shift
                        относительно начала>
                        shift += length(s1);
                        …
                        <копирование sn в s со сдвигом на shift
                        относительно начала>
```

Для присваивания вида модификация используются следующие правила.

```
s = s + e;      →   if (store(s) >= len(e)) {
                            <копировать e вслед за значением s>
                    } else {
                            resize(s, len(s) + len(e));
                            <копировать e вслед за значением s>
                    }
s = s.car;      →   s→len = 1;
                    s→data[s→len] = 0;
s = prec(s);    →   s→len = s→len – 1;
                    s→data[s→len] = 0;
```

5.5. Правила трансформации для дерева

Напомним представление дерева через указатели:

```
struct Tree {
    Tree *left;
    Tree *right;
    BAL balance;
    Tkey key;
    Tinfo info;
}
```

Приведем правила трансформации для деревьев при кодировании дерева через указатели.

```
Tree tree(leaf);                    →   Tree *tree = NULL;
Tree tree(key, info, bal, leaf, leaf);
                                    →   Tree *tree = new Tree();
                                        tree→key = key;
                                        tree→info = info;
                                        tree→balance = bal;
                                        tree→left = NULL;
                                        tree→right = NULL;
if(!isAlive(subtree1) && !isAlive(subtree2))
Tree tree(key, info, bal, subtree1, subtree2);
                                    →   Tree *tree = new Tree();
                                        tree→key = key;
                                        tree→info = info;
                                        tree→balance = bal;
                                        tree→left = subtree1;
                                        tree→right = subtree2;
if(!isAlive(subtree1) && isAlive(subtree2))
Tree tree(key, info, bal, subtree1, subtree2);
                                    →   Tree *tree = new Tree();
                                        tree→key = key;
                                        tree→info = info;
                                        tree→balance = bal;
                                        tree→left = subtree1;
                                        tree→right = copy_tree(subtree2);
if(isAlive(subtree1) && !isAlive(subtree2))
Tree tree(key, info, bal, subtree1, subtree2);
                                    →   Tree *tree = new Tree();
                                        tree→key = key;
                                        tree→info = info;
                                        tree→balance = bal;
                                        tree→left = copy_tree(subtree1);
                                        tree→right = subtree2;
if(isAlive(subtree1) && isAlive(subtree2))
Tree tree(key, info, bal, subtree1, subtree2);
                                    →   Tree *tree = new Tree();
                                        tree→key = key;
                                        tree→info = info;
                                        tree→balance = bal;
                                        tree→left = copy_tree(subtree1);
                                        tree→right = copy_tree(subtree2);
```

6. Примеры кодирования рекурсивных структур

6.1. Кодирование списка через указатели

6.1.1. Вычисление суммы значений всех элементов списка

Дан список элементов, для которых определена операция сложения. Нужно вычислить сумму значений всех элементов списка. Код программы на исходном языке:

```
type T;
type ListT = list(T);
addItems(ListT l, T t : T s) {
    if ( l = nil )
        s = t;
    else
        addItems(l.cdr, t + l.car : s);
}
```

Код после замены хвостовой рекурсии циклом, открытой подстановки и склеивания переменных:

```
addItems(ListT l : T s) {
    s = 0;
    while ( l != nil ) {
        s = s + l.car;
        l = l.cdr;
    }
}
```

Процесс преобразования предикатной программы в императивное расширение осуществляется путем последовательного обхода инструкций в исходном тексте программы и их преобразовании в императивное расширение. Рассмотрим этот процесс применительно к текущему примеру.

В программе выделяются три конструкции со списками: `l != nil`, `l.car`, `l = l.cdr`. Для первых двух конструкций процесс кодирования реализуется единственным образом без каких-либо условий применимости. Более подробно рассмотрим конструкцию `l = l.cdr`. По результатам потокового анализа все действия с переменной `l` реализуют лишь анализ списка с продвижением по нему, без его модификации в памяти, точнее изменяется указатель на текущий элемент, но не изменяются значения элементов списка. Поэтому в данном случае применяется преобразование для присваивания вида сканирование.

В результате применения найденных преобразований получим следующую программу на императивном расширении.

```
addItems (ListT *l : T s) {
    s = 0;
    while ( l != NULL ) {
        s = s + l→data;
        l = l→next;
    }
}
```

Код, написанный вручную:

```
int64_t calc_sum_manuall(List *list) {
    int64_t result = 0;
    while (list != nullptr) {
        result += list→value;
        list = list→next;
    }
    return result;
}
```

6.1.2. Обращение списка

Дан список, необходимо его инвертировать, т.е. обратить порядок элементов так, чтобы первый элемент стал последним, а последний первым. Пусть список представлен последовательностью элементов: $s_1,s_2,\ldots,s_n$. Тогда результатом инвертирования будет список $s_n,\ldots s_2,s_1$. Проблема дедуктивной верификации программы инвертирования списка чрезвычайно сложна и относится к категории Verification Grand Challenges. Графически задачу можно представить следующим образом (рис. 3).

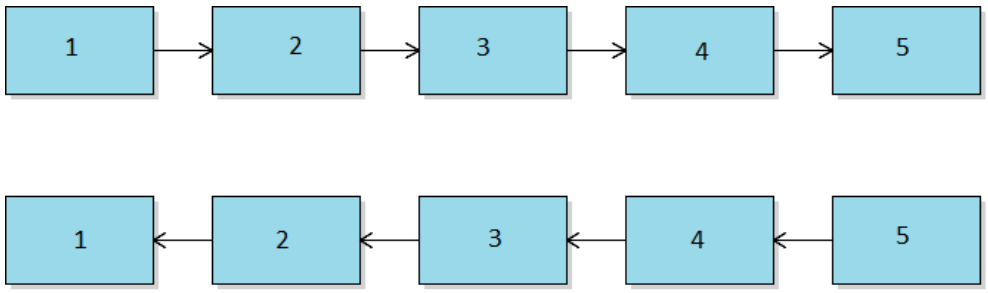

Рисунок 1

Код программы на исходном языке:

```
type T;
type ListT = list(T);
reverse(ListT s : ListT s') pre s != nil {
    reverseIn([s.car], s.cdr : s');
}
reverseIn(ListT s, u : s') {
    if (u = nil)
        s' = s;
    else
        reverseIn(u.car + s, u.cdr : s');
}
```

В программе reverse аргумент s – исходный список, результат s' – инвертированный список. Программа reverseIn является обобщением программы reverse. Исходный список представлен в reverseIn из двух частей. Аргумент s есть начальная часть списка в инвертированном виде, аргумент u – оставшаяся часть списка.. Графически аргументы программы reverseIn можно представить следующим образом (рис. 4).

Рисунок 2

Код после замены хвостовой рекурсии циклом, открытой подстановки и склеивания переменных:

```
reverse(ListT s : ListT s) {
    ListT u = s.cdr;
    s = [s.car];
    while ( u != nil ) {
        s = u.car + s;
        u = u.cdr;
    }
}
```

В программе выделяются пять конструкций со списками: u = s.cdr, s = [s.car], u != nil, s = u.car + s, u = u.cdr. Для конструкций s = [s.car] и u != nil процесс кодирования реализуется единственным образом без каких-либо условий применимости. Рассмотрим ListT u = s.cdr. По результатам потокового анализа все действия с переменной u реализуют лишь анализ списка с продвижением по нему, без его модификации в памяти, точнее изменяется указатель на текущий элемент, но не изменяются значения элементов списка. Поэтому в данном случае применяется преобразование для присваивания вида сканирование. Дополнительных условий применимости не требуется. Рассмотрим конструкцию s = u.car + s; В данном случае изменяется значение, размещаемое в памяти s, поэтому для кодирования данной конструкции используется присваивание вида модификация. Переменная u.car не является живой, поэтому выбирается преобразование без копирования переменной u.car. Рассмотрим конструкцию u = u.cdr. Как говорилось ранее, по результатам потокового анализа все действия с переменной u реализуют лишь анализ списка с продвижением по нему, поэтому в данном случае имеет место присваивание вида сканирование. В результате, для данных конструкций используется правило трансформации для парного кодирования. Т.е. применяется следующее правилом

```
s = u.car + s; u = u.cdr; →   list *a = u→next;
                              u→next = s;
                              s = u;
                              u = a;
```

В результате применения соответствующих правил трансформации получаем следующий код.

```
reverse(list *s) {
    list *u = s→next;
    s→next = NULL;
    while ( u != NULL ) {
        list *a = u→next;
        u→next = s;
        s = u;
        u = a;
    }
}
```

Код, написанный вручную:

```
List *revert_manuall(List *list) {
    List *result = nullptr;
    while (list != nullptr) {
        List *tmp = list;
        list = list→next;
        tmp→next = result;
        result = tmp;
    }
    return result;
}
```

6.2. Кодирование списка через массив

6.2.1. Вычисление суммы значений всех элементов списка

Дан список элементов, для которых определенны операция сложения. Нужно вычислить сумму значений всех элементов списка. Код программы на исходном языке:

```
type T;
type ListT = list(T);
addItems(ListT l, T t : T s) {
    if (len(l) = 0)
        s = t;
    else
        addItems(l.cdr, t + l.car : s);
}
```

Код после замены хвостовой рекурсии циклом, открытой подстановки и склеивания переменных:

```
addItems(ListT l : T s) {
    s = 0;
    while (len(l) != 0) {
        s = s + l.car;
        l = l.cdr;
    }
}
```

Процесс преобразования предикатной программы в императивное расширение осуществляется путем последовательного обхода инструкций в исходном тексте программы и их преобразовании в императивное расширение. Рассмотрим этот процесс применительно к текущему примеру.

В программе выделяются три конструкции со списками: len(l), l.car, l = l.cdr. Для первых двух конструкций процесс кодирования реализуется единственным образом без каких-либо условий применимости. Более подробно рассмотрим конструкцию l = l.cdr. По результатам потокового анализа все действия с переменной l реализуют лишь анализ списка с продвижением по нему, без его модификации в памяти, точнее изменяется указатель на текущий элемент, но не изменяются значения элементов списка. Поэтому в данном случае применяется преобразование для присваивания вида сканирование.

```
addItems (ListT *l : T s) {
    s = 0;
    while ( (l→n – l→m) != 0) {
        s = s + l→data[l→m];
        l→m += 1;
    }
}
```

Код, написанный вручную:

```
int64_t calc_sum_manuall(ArrayList *list) {
    int64_t result = 0;
    for (int64_t i = list→m; i < list→n; ++i) {
        result += list→data[i];
    }
    return result;
}
```

6.3. Анализ производительности примеров

В приведенных выше примерах показаны коды программ на императивном расширенные, которые получены при помощи оптимизирующих преобразований, в том числе применением правил трансформации. Необходимо оценить производительность программ, которые получаются в результате компиляции исходных кодов, полученных при помощи оптимизирующих преобразований. А также сравнить их производительность, с программами полученными при помощи компиляции программ, которые изначально были написаны на императивном языке C++.

Для анализа и сравнения производительности сгенерированного и написанного вручную кода использовалась библиотека Celero [13]. Программы компилировались при помощи стандартного компилятора IDE Visual Studio 2013 со стандартными настройками. В ходе тестирования выполнялось несколько итераций программ с разным размером входных данных. Результаты тестирования следующие (Таблица 1):

Таблица 1. Тест производительности

Timer resolution: 0.410529 us

Group	Experiment	Prob. Space	Baseline	us/Iteration	Iterations/sec
ElemSumList	Manual	16	1.00000	0.03337	29964451.05
ElemSumList	Manual	128	1.00000	0.25748	3883852.16
ElemSumList	Manual	1024	1.00000	1.92963	518235.21
Revert	Manual	16	1.00000	0.02475	40408331.57
Revert	Manual	128	1.00000	0.21030	4755128.63
Revert	Manual	1024	1.00000	1.58906	629301.42
ElemSumArrayLis	Manual	16	1.00000	0.04778	20929556.24
ElemSumArrayLis	Manual	128	1.00000	0.25167	3973508.08
ElemSumArrayLis	Manual	1024	1.00000	1.72695	579054.14
ElemSumList	Generated	16	0.99491	0.03320	30117863.32
ElemSumList	Generated	128	0.99054	0.25504	3920961.98
ElemSumList	Generated	1024	1.02137	1.97086	507391.48
Revert	Generated	16	1.23557	0.03058	32704124.76
Revert	Generated	128	1.23033	0.25874	3864920.05
Revert	Generated	1024	1.25464	1.99371	501578.47
ElemSumArrayLis	Generated	16	0.99901	0.04773	20950360.14
ElemSumArrayLis	Generated	128	1.00148	0.25204	3967648.13
ElemSumArrayLis	Generated	1024	1.00366	1.73328	576939.74

Столбец Group содержит названия решаемых задач, где ElemSumList — нахождение суммы элементов при кодировании списка через указатели, Revert — обращение списка, ElemSumArrayLis - нахождение суммы элементов при кодировании списка через массив. Столбец Experiment содержит обозначение способа, каким был получен код: Manual — написан вручную на императивном языке, Generated — получен при помощи оптимизирующих преобразований. Столбец Prob. Space содержит информацию о размере входных данных, на которых проводилось тестирование. Столбец Baseline отображает соотношение времени, затраченного на выполнения кода, для программы написанной вручную и для программы сгенерированной при помощи оптимизирующих трансформаций. us/Iteration — время одной итерации программы в миллисекундах. Iterations/sec — количество итераций в секунду.

Для того чтобы сравнить производительность программ, написанных вручную и сгенерированных при помощи оптимизирующих трансформаций, рассмотрим столбец Baseline. По результатам измерений производительности можно увидеть, что в одних случаях быстрее выполняется код, написанный в ручную, а в других — сгенерированный код, при этом колебания производительности достаточно малы. Это говорит о том, что производительность кода, написанного вручную и сгенерированного при помощи оптимизирующих преобразований практически идентична.

По результатам тестирования можно сказать, что поставленная в начале работы задача выполнена. Определены эффективные способы кодирования языковых конструкций использующих алгебраические структуры, которые позволяют добиться для оттранслированной программы на языке P производительности, сравнимой с кодом, изначально написанным на императивном языке программирования, таком как C++.

7. Заключение

В данной работе описывается трансформация кодирования объектов алгебраических типов через массивы и указатели. Кодирование операций над объектами алгебраических типов представлено набором правил, определяющих замену исходной операции на ее образ в языке императивного расширения. Кодирование эффективно для простых типовых случаев и позволяет получить программы по эффективности сравнимые с написанными вручную. С этой целью проводится потоковый анализ программы, в частности определяется время жизни переменных.

В дальнейшем планируется реализовать полный набор кодирования списков: через односвязный список, двусвязный список, очередь, массив и деку. По набору операций, используемых в программе, для конкретного объекта можно было бы автоматически определять один из указанных способов кодирования, наиболее подходящий для набора используемых операций.

Список литературы

1. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т. 4 (33) , 2008. С. 79-94.

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

3. Каррано Ф. М. Абстракция данных и решение задач на C++. Стены и зеркала. Вильямс, 2003. 848 с.

4. Касьянов В. П. Оптимизирующие преобразования программ. М.: Наука, 1988. 336 с.

5. Кнут Д. Э. Искусство программирования. Том 1. Основные алгоритмы. Вильямс, 2010. 720 с.

6. Кормен Т. Х. Алгоритмы. Построение и анализ. Вильямс, 2013. 1328 с.

7. Петров Э. Ю. Склеивание переменных в предикатной программе // Методы предикатного программирования. ИСИ СО РАН, Новосибирск, 2003. С. 48–61.

8. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.

9. Шелехов В.И. Предикатное программирование. Учебное пособие. НГУ, Новосибирск, 2009. 109 с.

10. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).

11. Шелехов В.И. Списки и строки в предикатном программировании // Системная информатика, №3, 2014. ИСИ СО РАН, Новосибирск. С. 25-43. [Электронный ресурс] URL: http://persons.iis.nsk.su/files/persons/pages/String.pdf

12. Шелехов В.И. Язык предикатного программирования Р. Описание языка. - Новосибирск, 2013, [Электронный ресурс] URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf.

13. Celero - A C++ Benchmark Authoring Library. [Электронный ресурс] URL: http://www.codeproject.com/Articles/525576/Celero-A-Cplusplus-Benchmark-Authoring-Library

14. Cooke D. E., Rushton J. N. Taking Parnas's Principles to the Next Level: Declarative Language Design. *Computer*, 2009, vol. 42, no. 9, P. 56-63.

15. Meyer B. Towards a Calculus of Object Programs // Patterns, Programming and Everything, Judith Bishop Festschrift, eds. Karin Breitman and Nigel Horspool, Springer-Verlag, 2012. P. 91-128.

16. Pfaff B. An Introduction to Binary Search Trees and Balanced Trees. [Электронный ресурс] URL: https://ftp.gnu.org/gnu/avl/avl-2.0.2.pdf.gz

17. Pfaff B. Performance Analysis of BSTs in System Software. [Электронный ресурс] URL: http://benpfaff.org/papers/libavl.pdf

18. Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. Vol. 45, No. 7, P. 421–427.