УДК 004.43

Предикатная программа вставки в АВЛ-дерево

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

Операции с АВЛ-деревьями компактно и элегантно представляются в языках функционального программирования. Однако функциональные программы для операций вставки или удаления вершины заведомо неэффективны, поскольку определяют построение нового дерева, а не модификацию исходного.

Описывается построение двух версий предикатных программ вставки в АВЛ-дерево, допускающих автоматическую трансформацию в эффективные императивные программы. В языке предикатного программирования введена эффективно реализуемая операция доступа вершины по пути в дереве.

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

1. Введение

Принципиальная сложность императивного программирования обнаруживается особенно при работе с указателями. Показателем такой сложности является чрезвычайная трудность дедуктивной верификации программ, оперирующих указателями, например, в алгоритме реверсирования списка [14].

В предикатном программировании [6, 7, 16] нет таких языковых конструкций, как циклы и указатели, серьезно усложняющие программу. Вместо циклов используются рекурсивные программы, а вместо указателей – объекты алгебраических типов (списки и деревья). Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением следующих оптимизирующих преобразований [3], переводящих программу на императивное расширение языка P [4]:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;

- кодирование алгебраических типов (списков и деревьев) с помощью массивов и указателей для всех видов операций с объектами алгебраических типов [9]. Отметим, что для алгоритма реверсирования списка используется одна нетривиальная трансформация.

Такая структура данных как граф непредставима алгебраическими типами данных. В предикатном программировании граф представляется массивом вершин. Индекс вершины в массиве является аналогом указателя.

Операции с АВЛ-деревьями компактно и элегантно представляются в языках функционального программирования [10, 13]. Имеется более десятка разных работ (см. например [16]) по дедуктивной верификации и доказательному построению простейших функциональных программ, реализующих операции с АВЛ-деревьями. Однако функциональные программы для операций вставки или удаления вершины заведомо неэффективны, поскольку определяют построение нового дерева, а не модификацию исходного.

В данной работе делается попытка построения таких предикатных программ вставки в АВЛ-дерево, чтобы применением оптимизирующих трансформаций получить эффективные императивные программы, подобные представленным в [1, 2, 15]. До сих пор в технологии предикатного программирования удавалось воспроизвести любую реализацию, проводимую в императивном программировании, для обширного набора алгоритмов из класса задач дискретной и вычислительной математики. Однако при реализации алгоритмов работы с АВЛ-деревьями, особенно нерекурсивного алгоритма вставки нового элемента, обнаруживается недостаток существующих средств. В настоящей работе в языке P вводятся новые конструкции, в частности, средства доступа вершины по некоторому пути в дереве.

В разделе 2 определяется языковые и технологические особенности предикатного программирования. Представление АВЛ-деревьев описывается в разделе 3. Вводятся дополнительные конструкции языка P для эффективной работы с деревьями. В разделе 4 приведены предикатные программы для классического рекурсивного алгоритма вставки в АВЛ-дерево, а также эффективного нерекурсивного алгоритма. В разделе 5 описываются методы применения оптимизирующих трансформаций с получением эффективных императивных программ для двух версий алгоритма вставки в АВЛ-дерево. В заключении отмечаются особенности реализации и приводятся сравнения с реализациями на форуме [1].

2. Предикатное программирование

Полная предикатная программа состоит из набора рекурсивных *предикатных* программ на языке P [4] 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 [6, 7, 16].

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 [6, 7, 16]. Отметим, что в функциональном программировании (при общеизвестной ориентации на предельную компактность и декларативность [12]) 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.

Гиперфункции. Вызов программы A(x, y) с аргументами x и результатами y записывается в виде A(x**:** y). *Гиперфункция* – программа с несколькими *ветвями* результатов. Гиперфункция A(x**:** y**:** z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются.

Рассмотрим предикатную программу следующего вида:

```
A(x: y, z, c)
pre P(x)
post c = C(x) & (C(x) ⇒ S(x, y)) & (¬C(x) ⇒ R(x, z))
{ … };
```

Здесь x, y и z – непересекающиеся возможно пустые наборы переменных; P(x), C(x), S(x, y) и R(x, z) – логические утверждения. Предположим, что все присваивания вида c = **true** и c = **false** – последние исполняемые операторы в теле программы. Программа A может быть заменена следующей программой в виде *гиперфункции*:

```
hyper A(x: y #1: z #2)
pre P(x)  pre 1: C(x)
post 1: S(x, y) post 2: R(x, z)
{ … };
```

В теле гиперфункции каждое присваивание c = **true** заменено оператором перехода #1, а c = **false** – на #2.

Гиперфункция A имеет две *ветви* результатов: первая ветвь включает набор переменных y, вторая ветвь – z. *Метки* 1 и 2 – дополнительные параметры, определяющие два различных *выхода* гиперфункции. *Спецификация гиперфункции* состоит из двух частей. Утверждение после “**pre** 1” есть предусловие первой ветви; предусловие второй ветви – отрицание предусловия первой ветви. Утверждения после “**post** 1” и “**post** 2” есть постусловия для первой и второй ветвей, соответственно.

Ветви *вызова гиперфункции* выходят в разные места программы, содержащей вызов. Вызов гиперфункции записывается в виде A(x: y #M1: z #M2). Здесь M1 и M2 – метки программы; операторы перехода #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2. Вызов вида A(x: y #M1: z #M2); M1: **...** может быть представлен в виде A(x: y: z #M2).

Аппарат *гиперфункций* является более общим и гибким по сравнению с известным механизмом обработки исключений, например, в таких языках, как Java и C++. Использование гиперфункций делает программу короче, быстрее и проще для понимания [7, 8].

Императивные конструкции. *Модифицируемой* является переменная, являющаяся аргументом и результатом некоторой предикатной программы. Наряду с оператором вида x′ = x + 1, где подразумевается, что x′ склеивается с x, в предикатной программе допускается оператор вида x = x + 1, а также привычная его форма в виде x++.

На базе операции модификации [4] для значений структурных типов строится *оператор модификации*. Оператор A[i] = x является эквивалентом A′ = A **with** [i: x]. Аналогично, оператор B.f = x эквивалентен B′ = B **with** (f: x). Дополнительно, поля конструктора типа объединения подобны полям структуры, и для них также следует разрешить операцию модификации и эквивалентный оператор модификации. Следующий шаг – это возможность использования переменных вида A[i] и B.f в качестве результатов в вызовах предиката: подобные вызовы нетрудно заменить легальными конструкциями вставкой дополнительного оператора модификации за вызовом. Например, G(…**:** A[i]) заменяется на G(…**:** X x); A′ = A **with** [i**:** x].

Следует предоставить возможность заменить оператор вида b′ = b пустым оператором. Вследствие замены оператора вида b′ = b пустым оператором появляется укороченный условный оператор **if** (E(x)) A(x**:** y).

В функциональном программировании внесение в программу императивных конструкций реализуется неявно через аппарат монад. Без монад функциональные программы потеряли бы свою компактность и привлекательность. В предикатном программировании императивные конструкции определены явно. Программа с императивными конструкциями легко приводима к правильной предикатной программе.

Дополнительные конструкции для работы с деревьями представлены в разделе 3.

3. АВЛ-деревья

Двоичное дерево – дерево, в котором каждая вершина имеет не более двух потомков. Двоичное дерево используется для представления *таблицы* для хранения множества данных вместе с их *ключами*, используемыми для поиска. Основные операции: включение нового данного, исключение данного и поиск данного в таблице. 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.

Элемент таблицы является структурой из двух полей:

```
type ElTab = struct(Tkey key, Tinfo info);
```

Двоичное дерево представляется структурой типа Tree:

```
type BAL = -2..2;
type Tree = union (
  leaf,
  node (Tkey key, Tinfo info, BAL balance, 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. Назначение поля balance будет определено ниже.

Высота heigh дерева N определяется следующей формулой:

```
formula heigh(Tree N: nat) = (N == leaf)? 0 : max(heigh(N.left), heigh(N.right);
```

Совокупность элементов таблицы, хранящихся в двоичном дереве N, характеризуется предикатом isin, определяющим принадлежность элемента (k, x) таблице:

```
formula isin(Tkey k, Tinfo x,  Tree N) =
  (N == leaf)? false : N.key == k & N.info == x ∨ isin(N.left) ∨ isin(N.right);
```

В соответствии с данной формулой для непустого дерева элемент (k, x) либо хранится в корневой вершине, либо принадлежит одному из поддеревьев.

Двоичное дерево поиска – двоичное дерево со следующими свойствами:

- оба поддерева – левое и правое, являются двоичными деревьями поиска;
- у всех вершин левого поддерева произвольной вершины X значения ключей данных меньше, нежели значение ключа данных самой вершины X;
- у всех вершин правого поддерева той же вершины X значения ключей данных больше, нежели значение ключа данных вершины X.

Двоичное дерево поиска N удовлетворяет следующему отношению упорядоченности isord:

```
formula isord(Tree N) =
    (N == leaf)?  true :  isord(N.left) & isord(N.left) &
                          (∀ Tkey k, Tinfo x. isin(k, x, N.left) ⇒ k < N.key) &
                          (∀ Tkey k, Tinfo x. isin(k, x, N.right) ⇒ N.key < k);
```

АВЛ-дерево – сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота двух поддеревьев вершины различается не более чем на 1. Свойство isbal сбалансированности дерева N определяется следующей формулой:

```
formula isbal(Tree N) =
    (N == leaf)?  true :  isbal(N.left) & isbal(N.left) &
                          (heigh(N.left) == heigh(N.right) ∨
                           heigh(N.left) + 1 == heigh(N.right) ∨
                           heigh(N.left) == heigh(N.right) +1 );
```

Поле balance – разница высот правого и левого поддеревьев вершины N: N.balance = heigh(N.right) - heigh(N.left). Дерево, в котором поле balance в каждой вершине равно разнице высот поддеревьев, удовлетворяет предикату, представленному формулой:

```
formula withbal(Tree N) =
(N == leaf)?  true :  N.balance == heigh(N.right) - heigh(N.left) &
                      withbal(N.left) & withbal(N.right);
```

Тип АВЛ-дерева определяется следующим образом:

formula isAVL(Tree N) = isord(N) & isbal(N) & withbal(N)
type AVLtree = **subtype** (Tree N**:** isAVL(N)};

Дополнительные операции с деревьями. С алгебраическим типом Tree считаются ассоциированными следующие типы:

type __Tree = **enum** (left, right);
type _Tree = list(__Tree);

Переменная типа __Tree называется *динамическим полем*. Значение типа _Tree определяет *путь* в дереве в виде последовательности полей, ведущих от корня дерева в некоторую его вершину. Для динамического поля f, принадлежащего типу __Tree, конструкция N.f определяет доступ по чтению и записи определяется следующим образом:

N.f ≡ f == left? N.left **:** N.right;
N.f = x ≡ **if** (f == left) N.left = x **else** N.right = x;

Доступ к вершине, идентифицируемой путем p в дереве N, реализуется конструкцией N.p.

N.p ≡ p == nil? N **:** N.(p.car).(p.cdr);
N.p = x ≡ **if** (p == nil) N = x **else** N.(p.car).(p.cdr) = x;

Конструкция N.p определена лишь при условии корректности пути p. Путь p в дереве N является *корректным*, если он существует в дереве N. Корректность пути определяется предикатом valid:

formula valid(_Tree p, Tree N) = p == nil ? **true :** N != leaf & valid(p.cdr, N.(p.car));

Для пути p операция p.left означает присоединение поля left к пути p. Иначе говоря, значение p.left есть p + left, где «+» понимается как операция конкатенации списков.

В трансформации операций с деревьями конструкция N.p обычно представляется указателем на переменную, соответствующую последнему полю, ссылающемуся на требуемую вершину.

4. Программы вставки в АВЛ-дерево

Описываются два алгоритма вставки элемента в АВЛ-дерево. Первый рекурсивный алгоритм является классическим. Второй, нерекурсивный алгоритм, ранее был представлен лишь в виде императивной программы [2, 15].

4.1. Рекурсивный алгоритм

Гиперфункция AVLinsert реализует вставку значения ainfo с ключом akey в АВЛ-дерево tree. Выход гиперфункции #plus1 реализуется в случае, когда после вставки высота дерева tree увеличивается на 1; выход #same соответствует случаю, когда высота дерева остается прежней. Наличие «*» у аргумента tree означает, что tree является модифицируемой переменной, т.е. является результатом, причем на обеих ветвях гиперфункции AVLinsert.

```
formula Q_insert(Tree tree, tree′, Tkey akey, Tinfo ainfo) =
        ∀ Tkey k, Tinfo x. ( isin(k, x, tree′) ≡ k = akey & x = ainfo ∨ isin(k, x, tree));
hyper AVLinsert(AVLtree tree*, Tkey akey, Tinfo ainfo: #plus1 : #same)
pre plus1: heigh(tree′) == heigh(tree) +1
pre same: heigh(tree′) == heigh(tree)
post  Q_insert(tree, tree′, akey, ainfo)
{  if (tree == leaf) { tree′ = node(akey, ainfo, 0, leaf, leaf) #plus1 }
elsif (tree.key > akey) {
        AVLinsert(tree.left*, akey, ainfo:  : #same);
        switch (tree.balance) {
           case 1:  tree.balance = 0
           case 0:  tree.balance = -1 #plus1
           case -1: RotateRight(tree*)
        }
} elseif (tree.key < akey) {
        AVLinsert(tree.right*, akey, ainfo:  : #same);
        switch (tree.balance) {
           case -1: tree.balance = 0
           case 0:  tree.balance = 1 #plus1
           case 1:  RotateLeft(tree*)
        }
} else  tree.info = ainfo;
#same
};
```

Поясним некоторые правила для гиперфункций. Если исполнение рекурсивного вызова AVLinsert завершается второй ветвью, то и программа AVLinsert завершается второй ветвью. Если исполнение вызова AVLinsert завершается первой ветвью, то далее исполняется следующий оператор после вызова, поскольку в позиции результатов первой ветви нет оператора перехода.

Если высота левого поддерева увеличивается после срабатывания первого рекурсивного вызова AVLinsert (что соответствует первому выходу гиперфункции), поле tree.balance следует уменьшить на единицу. Если при этом получим tree.balance=-2, реализуется ротация дерева вправо, показанная на рис.1а и 1б. В результате получим правильное АВЛ-дерево, содержащее то же множество вершин, что и дерево до ротации.

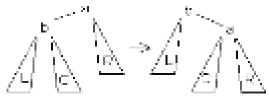

Рис 1а

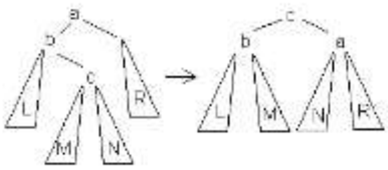

Рис 1б

Ротация на рис1.а реализуется при условии, что высота поддерева L больше, чем высота поддерева C. В противном случае проводится ротация, показанная на рис 1б.

Реализация ротации представлена предикатом:

```
formula eq(Tree tree, tree’) =∀ Tkey k, Tinfo x. isin(k, x, tree) ≡ isin(k, x, tree’);
pred RotateRight(Tree tree: AVLtree tree’)
    pre  tree != leaf & isAVL(tree.left) & isAVL(tree.right) &
         heigh(tree.left) +2 = heigh(tree.right)
    post eq(tree, tree’) & isAVL(tree’)
{   AVLtree L = tree.left;
    if (L.balance == -1)
       tree’ = L with (right: tree with (balance: 0, left: L.right))
    else  {
       AVLtree LR = L.right;
       tree’ = LR with (  left: L with (balance: (LR.balance=-1)? 1 : 0,
                                         right: LR.left),
                          right: tree with (balance: (LR.balance=1)? -1 : 0,
                                            left: LR.right));
    };
    tree.balance = 0
};
```

Алгоритм ротации для случая, когда значение ainfo с ключом akey вставляется в правое поддерево, аналогичен представленному выше алгоритму для левого поддерева. Соответствующие ротации показаны на рис.2а и 2б.

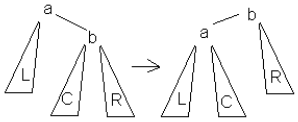

Рис 2а

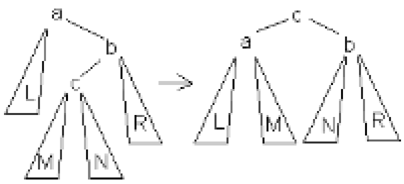

Рис 2б

```
pred RotateLeft(Tree tree: AVLtree tree')
    pre  tree != leaf & isAVL(tree.left) & isAVL(tree.right) &
         heigh(tree.left) = heigh(tree.right) +2
    post eq(tree, tree') & isAVL(tree')
{   AVLtree R = tree.right;
    if (R.balance == 1)
       tree' = R with (left: tree with (balance: 0, right: R.left))
    else  {
       AVLtree RL = R.left;
       tree' = RL with ( left: tree with (balance: (RL.balance=1)? -1 : 0,
                                          right: RL.left),
                         right: R with (balance: (RL.balance=-1)? 1 : 0,
                                          left: RL.right));
    };
    tree.balance = 0
}
```

4.2. Нерекурсивный алгоритм

Алгоритм реализует вставку элемента ainfo с ключом akey в дерево N. Если в дереве присутствует вершина с ключом akey, то существующий элемент заменяется на ainfo, при этом реализуется выход гиперфункции #replace. В противном случае в дерево вставляется новый элемент с выходом #new.

Алгоритм реализуется следующим образом. Находится путь q до листа дерева N, куда надо вставить новую вершину, чтобы сохранить упорядоченность по ключам (отношение isord). Дополнительно определяется путь y, являющийся начальной частью пути q, до вершины с ненулевым значением поля balance при условии, что все вершины далее по пути q имеют нулевой balance. Нетрудно показать, что достаточно провести изменения лишь на отрезке пути от конца y до конца q, а остальная часть дерева останется неизменной. На втором шаге корректируется поле balance на найденном отрезке пути. Наконец, в случае, когда для вершины, идентифицируемой путем y, скорректированное поле balance имеет значение -2 или +2, проводится соответствующая ротация дерева в позиции y.

```
hyper AVLinsert1(AVLtree N*, Tkey akey, Tinfo ainfo: #new : #replace)
     pre new: ∀ Tinfo x. ¬ isin(akey, x, tree)
     post Q_insert(tree, tree′, akey, ainfo)
{  Search(N, akey, ainfo: _Tree y, q: N′ #replace);
   N.q = node(akey, ainfo, 0, leaf, leaf);
   updateBalance(N*, y, q);
   if (N.y.balance == -2) RotateRight (N.y*)
   elseif (N.y.balance == 2) RotateLeft(N.y*);
   #new
};
```

Гиперфункция Search определяет путь q до листа для вставки новой вершины и подпуть y до минимального поддерева, в котором надо провести балансировку, если только не обнаружится вершина с ключом akey, в случае чего реализуется выход #replace.

```
hyper Search(AVLtree N, Tkey akey, Tinfo ainfo: _Tree y, q #new : N′ #replace)
     pre new: ∀ Tinfo x. ¬ isin(akey, x, tree)
     post new: Qsearch(N, akey, y, q)
     post replace: ∃ _Tree r. N.r.key = akey & N′ = N with (r: N.r with (info: ainfo))
{  Search1(N, akey, ainfo, nil, nil: y, q #new: N′#replace) }
```

Приведенное определение есть сведение к более общей программе Search1, в которой дополнительные два параметра фиксируют начальные значения пустых путей для y и q. Отметим, что при y = nil поле balance для корневой вершины N может оказаться нулевым.

Постусловие для выхода replace фиксирует, что в дереве N есть вершина с ключом akey и в итоговом дереве N′ отличается от N заменой поля info.

Постусловие для выхода new определяет условия на пути q и y.

```
formula Qsearch(Tree N, Tkey akey, _Tree y, q) =
        PSearch(N, akey, y, q) & N.q == leaf;
```

Предикат PSearch используется в качестве предусловия для программы Search1. Второй конъюнкт постулирует, что путь q достигает листа дерева N.

```
formula Psearch(Tree N, Tkey akey, _Tree y, q) =
        valid(q, N) & valid(y, N) &
        (N.y.balance !=0 ∨ y == nil) & ordered(N, akey, q) &
        ∃ _Tree r. q == y + r & ZeroBal(N.y, r);
```

Утверждается, что пути q и y являются корректными, путь q соответствует порядку ключей (предикат ordered), путь y либо пустой, либо заканчивается на вершине с ненулевым полем balance, путь y является начальной частью пути q, причем ниже находятся вершины с нулевым полем balance.

```
formula ordered(Tree N, Tkey akey, _Tree q) =
        q == nil? true : fiord(q.car, N.key, akey) & ordered(N.(q.car), akey, q.cdr)′
formula fiord(__Tree d, Tkey k, akey) = d == left? akey < k : k < akey;
```

В предикате ordered утверждается, что путь q реализуется движением по дереву N в соответствии с порядком ключей, что гарантирует правильную позицию в дереве для вставки новой вершины.

```
formula ZeroBal(Tree B, _Tree r) = B == leaf ∨ ZeroBal1(B.(r.car), r.cdr);
formula ZeroBal1(Tree B, _Tree r) =
     B == leaf ∨ r = nil ∨ ∃ Tree B1 == B.(r.car). B1.balance == 0 & ZeroBal1(B1, r.cdr);
```

В предикате ZeroBal утверждается, что все вершины на пути r, кроме, возможно, начальной, имеют поле balance = 0.

Программа Search1 строит пути q′ и y′ в предположении, что их начальная часть (q и y) уже построены. Алгоритм реализуется разбором случаев для вершины N.q на пути q.

```
hyper Search1(AVLtree N, Tkey akey, Tinfo ainfo, _Tree y, q:
                                              _Tree y′, q′ #new : N′ #replace)
     pre PSearch(N, akey, y, q)
     pre new: ∀ Tinfo x. ¬ isin(akey, x, tree)
     post new: Qsearch(N, akey, y, q)
{  if (N.q == leaf) #new;
if (N.q.balance !=0 ) y = q;
if (N.q.key > akey) Search1(N, akey, ainfo, y, q.left: y′, q′ #new: N′ #replace)
elsif (N.q.key < akey) Search1(N, akey, ainfo, y, q.right: y′, q′ #new: N′ #replace)
else { N.q.info = ainfo #replace}
};
```

Программа updateBalance модифицирует поле balance для всех вершин на пути от y до q исключая лист в конце пути q. В итоговом дереве поле balance является корректным, т.е. соответствует предикату withbal.

```
pred updateBalance(Tree N, Tkey akey, _Tree y, q : Tree N′)
pre isAVL(N with (q: leaf)) & isord(N)
post withbal(N′)
{  if (y != q) {
        if (N.y.key > akey) {N.y.balance--; updateBalance(N, akey, y.left, q)}
        else { N.y.balance++; updateBalance(N, akey, y.right, q)}
}
};
```

5. Трансформация операций с деревьями

Определим сначала трансформацию рекурсивного алгоритма. Сначала проводятся очевидные склеивания переменных типа tree′ → tree. В программах RotateRight и

RotateLeft декомпозируются иерархические операции модификации: каждая вложенная операция модификации выносится перед оператором в форме X = X **with** (...). Подобное вынесение корректно, если X далее нигде не используется, т.е. не является живой [3]; в противном случае необходимо будет сохранить значение X в дополнительной рабочей переменной. Декомпозируем модификации для программы RotateRight.

```
pred RotateRight(Tree tree: AVLtree tree)
{   Tree L = tree.left;
    if (L.balance == -1) {
       tree = tree with (balance: 0, left: L.right);
       L = L with (right: tree);
       tree = L
    } else {
       Tree LR = L.right;
       tree = tree with (balance: (LR.balance=1)? -1 : 0, left: LR.right);
       L = L with (balance: (LR.balance=-1)? 1 : 0, right: LR.left);
       LR = LR with (  left: L, right: tree);
       tree = LR;
    };
    tree.balance = 0
};
```

Далее реализуется замена операторов вида X = X **with** (...) на присваивания отдельным полям.

```
pred RotateRight(Tree tree: AVLtree tree)
{   Tree L = tree.left;
    if (L.balance == -1) {
       tree.balance = 0; tree.left =  L.right;
       L.right = tree;
       tree = L
    } else {
       Tree LR = L.right;
       tree.balance = (LR.balance=1)? -1 : 0;   tree.left = LR.right;
       L.balance = (LR.balance=-1)? 1 : 0;    L.right = LR.left;
       LR.left = L; LR.right = tree;
       tree = LR;
    };
    tree.balance = 0
};
```

Кодирование алгебраического типа Tree реализуется следующим образом. Значением типа дерево является указатель (типа TREE) на корневую вершину дерева. Лист дерева кодируется нулевым указателем. Тип вершины кодируется структурой типа Tree, определяющей поля конструктора node. Правое и левое поддеревья вершины представляются указателями на поддеревья.

```
type TREE = Tree*;
type Tree = struct (Tkey key, Tinfo info, BAL balance, TREE left, right);
```

Определим трансформации типов и конструкций в соответствии с данным способом кодирования алгебраического типа дерева:

```
Tree → TREE
leaf → null
N == leaf → N == null;
N.right →   N->right
```

Переменная tree в программе AVLinsert является аргументом и результатом. Вместо подстановки результатом используется подстановка через указатель. Поэтому используется переменная trEE типа TREE*. Предполагается, что программы RotateRight и RotateLeft открыто подставляются на место вызовов. При этом присваивания tree = L и tree = LR в RotateRight должны быть заменены на trEE = &L и trEE = &LR.

```
hyper AVLinsert(TREE* trEE, Tkey akey, Tinfo ainfo: #plus1 : #same)
{  TREE tree = trEE*;
   if (tree == null) { tree = node(akey, ainfo, 0, null, null) #plus1 }
   elseif (tree->key > akey) {
        AVLinsert(&(tree->left), akey, ainfo:  : #same);
        switch (tree->balance) {
           case 1:  tree->balance = 0
           case 0:  tree->balance = -1 #plus1
           case -1: RotateRight(tree)
        }
   } elseif (tree->key < akey) {
        AVLinsert(&(tree->right), akey, ainfo:  : #same);
        switch (tree->balance) {
           case -1: tree->balance = 0
           case 0:  tree->balance = 1 #plus1
           case 1:  RotateLeft(tree)
        }
   } else  tree->info = ainfo;
   #same
};
```

Поскольку вызовы AVLinsert нельзя подставить открыто, применяется общий способ реализации выходов гиперфункции через аргумент – переменную типа LABEL. Один из выходов гиперфункции, в нашем случае, это выход #plus1, можно реализовать как обычный возврат из процедуры. Самый внешний вызов вида AVLinsert(N, ke, inf : N′: N′), определяющий выход на следующий оператор после вызова для обеих ветвей гиперфункции, реализуется следующим образом:

```
AVLinsert(&N, ke, inf , SAME); SAME: ;
```

Отметим, что оператор перехода #same реализует переход непосредственно на метку SAME, минуя всю иерархию рекурсивных вызовов. Очевидно, что использование гиперфункции вместо результата типа **bool** дает выигрыш в эффективности. Процедура AVLinsert, реализующая выходы гиперфункции, представлена ниже.

```
AVLinsert(TREE* trEE, Tkey akey, Tinfo ainfo, LABEL same)
{  TREE tree = trEE*;
   if (tree == null) { tree = node(akey, ainfo, 0, null, null); return }
   elseif (tree->key > akey) {
        AVLinsert(&(tree->left), akey, ainfo, same);
        switch (tree->balance) {
           case 1:  tree->balance = 0
           case 0:  { tree->balance = -1 ; return }
           case -1: RotateRight(tree)
        }
   } elseif (tree->key < akey) {
        AVLinsert(&(tree->right), akey, ainfo, same);
        switch (tree->balance) {
           case -1: tree->balance = 0
           case 0:  { tree->balance = 1 ; return }
           case 1:  RotateLeft(tree)
        }
   } else  tree->info = ainfo;
   #same
};
```

Трансформация программы RotateRight, подставляемой в AVLinsert, представлена ниже.

```
pred RotateRight(TREE tree: TREE tree)
{    TREE L = tree->left;
     if (L->balance == -1) {
        tree->balance = 0; tree->left =  L->right;
        L->right = tree;
        trEE = &L
     } else  {
        AVLtree LR = L->right;
        tree->balance = (LR->balance=1)? -1 : 0;   tree->left = LR->right;
        L->balance = (LR->balance=-1)? 1 : 0;    L->right = LR->left;
        LR->left = L; LR->right = tree;
        trEE = &LR;
     };
     tree->balance = 0
};
```

Далее представим трансформацию нерекурсивного алгоритма AVLinsert1. Сначала заменим хвостовую рекурсию циклом в программах Search1 и updateBalance.

```
hyper Search1(AVLtree N, Tkey akey, Tinfo ainfo, _Tree y, q:
                                        _Tree y’, q’ #new : N’ #replace) {
   for(;;) {
        if (N.q == leaf) #new;
        if (N.q.balance !=0 ) y = q;
        if (N.q.key > akey) q = q.left
        elsif (N.q.key < akey) q = q.right
        else { N.q.info = ainfo #replace}
   }
};

pred updateBalance(Tree N, Tkey akey, _Tree y, q : Tree N’) {
   for(;;) {
        if (y != q) {
              if (N.y.key > akey) {N.y.balance--; y = y.left}
              else { N.y.balance++; y = y.right}
        }
   }
};
```

Подставим программу Search1 в Search.

```
hyper Search(AVLtree N, Tkey akey, Tinfo ainfo: _Tree y, q #new : N’ #replace)
{    _Tree y = nil, q = nil;
     for(;;) {
        if (N.q == leaf) #new;
        if (N.q.balance !=0 ) y = q;
        if (N.q.key > akey) q = q.left
        elsif (N.q.key < akey) q = q.right
        else { N.q.info = ainfo #replace}
     }
};
```

Подставим программы Search и updateBalance в AVLinsert1.

```
hyper AVLinsert1(AVLtree N*, Tkey akey, Tinfo ainfo: #new : #replace)
{  _Tree y = nil, q = nil;
   for(;;) {
      if (N.q == leaf) #new1;
      if (N.q.balance !=0 ) y = q;
      if (N.q.key > akey) q = q.left
      elsif (N.q.key < akey) q = q.right
      else { N.q.info = ainfo #replace}
   };
   new1:
   N.q = node(akey, ainfo, 0, leaf, leaf);
   for( ;y != q; ) {
         if (N.y.key > akey) {N.y.balance--; y = y.left}
         else { N.y.balance++; y = y.right}
   }
   if (N.y.balance == -2) RotateRight (N.y*)
   elsif (N.y.balance == 2) RotateLeft(N.y*);
   #new
};
```

Переход по метке #new1 можно заменить на **break**.

Будем считать, что любой путь, значение типа _Tree, строится только для одного объекта типа Tree. Путь кодируется указателем на поле вершины дерева. Это поле соответствует концу пути в дереве. Пустой путь кодируется указателем на переменную, значением которого является дерево. Путь кодируется значением типа PTREE.

```
type PTREE = TREE*;  // тип переменных q и y
```

Реализуются трансформации:

```
_Tree → PTREE;
N.q → q*;
N.y → y*;
N.q == leaf → q* == null
N.q.key → q*->key;
q = q.left → q = &(q*->left);
```

Применение трансформаций дает следующую программу.

```
hyper AVLinsert1(AVLtree N*, Tkey akey, Tinfo ainfo: #new : #replace)
{  PTREE y = &N, q = &N;
   for(;;) {
      if (q* == null) break;
      if (q*->balance !=0 ) y = q;
      if (q*->key > akey) q = &(q*->left)
      elsif (q*->key < akey) q = &(q*->right)
      else { q*->info = ainfo #replace}
   };
   q* = node(akey, ainfo, 0, leaf, leaf);
   for( ;y != q; ) {
         if (y*->key > akey) {y*->balance--; y = &(y*->left) }
         else { y*->balance++; y = &(y*->right }
   }
   if (y*->balance == -2) RotateRight (y*)
   elsif (y*->balance == 2) RotateLeft(y*);
   #new
};
```

Вместо двойного указателя (q или y) можно использовать одинарный. В использующих позициях переменных q и y применим трансформации:

$$q^* \rightarrow Nq;$$
$$y^* \rightarrow Ny;$$

Однако после присваивания переменной q или y необходим синхронный пересчет значения переменной. Итоговая программа представлена ниже.

```
hyper AVLinsert1(AVLtree N*, Tkey akey, Tinfo ainfo: #new : #replace)
{  PTREE y = &N, q = &N;
   TREE Nq = N; // = q*
   for(;;) {
      if (Nq == null) break;
      if (Nq->balance !=0 ) y = q;
      if (Nq->key > akey) q = &(Nq->left)
      elsif (Nq->key < akey) q = &(Nq ->right)
      else { Nq->info = ainfo #replace};
      Nq = q*;
   };
   q* = node(akey, ainfo, 0, leaf, leaf);
   TREE Ny = y*;
   for( ;y != q; ) {
         if (Ny->key > akey) { Ny->balance--; y = &(Ny->left) }
         else { Ny->balance++; y = &(Ny->right };
         Ny = y*;
   }
   if (Ny->balance == -2) RotateRight (y*)
   elsif (Ny->balance == 2) RotateLeft(y*);
   #new
};
```

Заключение

Предпосылкой появления данной работы стала дискуссия на форуме [1] о том, какая из программ вставки в АВЛ-дерево лучше: на языке Оберон или графическом языке Дракон [5]. Эргономические методы, применяемые в языке Дракон, существенно улучшают восприятие программы. Тем не менее, программа не выглядит проще. Причина – исходная сложность императивной программы. Методы предикатного программирования: использование рекурсивных программ вместо циклов, алгебраических типов вместо указателей и др. позволяют существенно снизить сложность программы по сравнению с аналогичной императивной программой, в частности с программами на форуме [1].

Доступ к вершине дерева реализован через указатель на поле (в некоторой вершине), в котором хранится ссылка на требуемую вершину. При передаче через параметр программы возникает двойной указатель. В описании библиотеки libavl [15] используется однократный указатель, однако при этом дополнительно поддерживается указатель на предыдущую вершину-отца. Как следствие, алгоритм получается более громоздким и менее эффективным в сравнении с приведенным в настоящей работе. В нашей версии, тем не менее, для каждого двойного указателя заводится соответствующий одинарный. Реализация такой техники в трансформациях может оказаться нетривиальной. Поэтому в начальном релизе следует ограничиться только двойным указателем.

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

Список литературы

1. АВЛ-дерево. Алгоритм добавления вершины. [Электронный ресурс]. URL: **http://forum.oberoncore.ru/viewtopic.php?f=78&t=4003**
2. Википедия. АВЛ-дерево. [Электронный ресурс]. URL: http://ru.wikipedia.org/wiki/%D0%90%D0%92%D0%9B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE
3. Каблуков И. В. Реализация оптимизирующих трансформаций предикатных программ // XIV Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям. Томск, 2013. 7с. URL: http://conf.nsc.ru/files/conferences/ym2013/fulltext/175069/177104/Опт.%20трансформации.pdf
4. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.12. Новосибирск, 2013. 52с. URL: http://persons.iis.nsk.su/files/persons/pages/plang12.pdf
5. Паронджанов В. Д. Язык ДРАКОН. Краткое описание. М., 2009. 124 с.
6. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.

7. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164).
8. Шелехов В.И. Разработка автоматных программ на базе определения требований // Системная информатика, №4, 2014. ИСИ СО РАН, Новосибирск. С. 1-29. URL: http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf
9. Шелехов В.И. Предикатное программирование. Учебное пособие. Новосибирск: НГУ, 2009. 109с.
10. AVL Tree in Haskell. [Электронный ресурс]. URL: **https://gist.github.com/gerard/109729**
11. Clochard M. Automatically Verified Implementation of Data Structures Based on AVL Trees // 6th Working Conference on Verified Software: Theories, Tools, and Experiments, 2014. P. 167-180.
12. 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.
13. Hettler R., Nazareth D., Regensburger F., Slotosch O. AVL trees revisited: A case study in Spectrum. LCNS, vol. 1009, 1995. P. 128-147.
14. 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.
15. Pfaff B. GNU libavl 2012. An Introduction to Binary Search Trees and Balanced Trees. URL: ftp://ftp.gnu.org/pub/gnu/avl/avl-2.0.2.pdf.gz
16. 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.