Предикатное программирование

Метод предикатного программирования
http://persons.iis.nsk.su/files/persons/pages/lbase.pdf

Оптимизирующие трансформации предикатных программ

Подстановка тела программы на место ее вызова

Замена хвостовой рекурсии циклом

Склеивание переменных

Трансформация программы обмена элементов массива

Трансформация кодирования рекурсивных структур

Пример. Обращение списка
http://persons.iis.nsk.su/files/persons/pages/listinvert.pdf

Пример. Сортировка простыми вставками

Гиперфункции

Неудобные ситуации в программировании

Гиперфункция и оператор продолжения ветвей

Свойства гиперфункций

2. Предикатное программирование

Метод предикатного программирования

Метод предикатного программирования

Задача построения предикатной программы, удовлетворяющей спецификации [P(x), Q(x, y)].

Набор x - *исходные данные* задачи.
Набор y - *неизвестные* задачи.

Предусловие P(x) ограничивает допустимый набор исходных данных.
Постусловие Q(x, y) определяет *условие* задачи, связывающее исходные и неизвестные задачи.

Решением задачи является предикатная программа H(x: y), соответствующая спецификации [P(x), Q(x, y)].

Спецификация должна быть тотальной.

Формы декомпозиции программы

1. Независимые подзадачи. Задача H(x: y, z) распадается на две независимые подзадачи B(x: y) и C(x: z).

Программа: H(x: y, z) {B(x: y) || C(x: z)}

2. Разбор случаев. Условие – предикат E(x). Исходная задача H(x: y) делится на две подзадачи.

Программа: H(x: y) { **if (**E(x)) B(x: y) **else** C(x: y)}

3. Сведение к другой задаче. Определяется новый объект z, связанный с исходными данными задачи H(x: y). Задача B(x: z). Далее решается задача C(x, z: y) нахождения искомого объекта y по объекту z.

Программа: H(x: y) { B(x: z); C(x, z: y) }

H(x: y) ≡ C(x, B(x): y) ≡ z = B(x) & C(x, z : y) ≡ B(x: z) & C(x, z : y)

Частный случай: *Сведение к более общей задаче*.
В задаче H(x: y) появляется дополнительный параметр z.
Постусловие остается тем же. Обобщается предусловие.

Др. формы декомпозиции для гиперфункций

Пример. Наибольший общий делитель

Программа great common divisor GCD(a, b: c) - *наибольший общий делитель* положительных a и b.

```
type nat1 = subtype (nat x:  x > 0);
theory Gcd {
 nat1 a, b;    nat c, x;
 formula divisor(a, x) = ∃ nat1 z. x ∗ z = a;
 formula divisor2(a, b, c) = divisor(a, c) & divisor(b, c);
 formula gcd(a, b, c) = divisor2(a, b, c) & ∀x.(divisor2(a, b, x) ⇒ x ≤ c);
 gc1: lemma gcd(a, a, a);
 gc2: lemma gcd(a, b, c) ≡ gcd(b, a, c);
 gc3: lemma gcd(a, b, c) ≡ gcd(a + b, b, c);
 gc4: lemma a < b ⇒ gcd(a, b, c) ≡ gcd(a, b - a, c);
 gc5: lemma a > b ⇒ gcd(a, b, c) ≡ gcd(a - b, b, c);
};
```

Спецификация программы GCD:

```
GCD(nat1 a, b: nat c) post gcd(a, b, c);
```

Спецификация: GCD(nat1 a, b: **nat** c) **post** gcd(a, b, c);

theory Gcd {

formula gcd(a, b, c) = divisor2(a, b, c) & ∀x.(divisor2(a, b, x) ⇒ x ≤ c);

gc1: **lemma** gcd(a, a, a);

gc4: **lemma** a < b ⇒ gcd(a, b, c) ≡ gcd(a, b - a, c);

gc5: **lemma** a > b ⇒ gcd(a, b, c) ≡ gcd(a - b, b, c);

} **Разбор случаев**: a = b, a < b и a > b. **Варианты решения:**

a = b → c = a

a < b → GCD(a, b - a: c)

a > b → GCD(a - b, b: c)

Предикатная программа:

GCD(nat1 a, b: **nat** c) **post** gcd(a, b, c) **measure** a + b

{ **if** (a = b) c = a

else **if** (a < b) GCD(a, b – a: c)

else GCD(a – b, b: c) };

Императивная программа:

while (a != b) { **if** (a < b) b = b – a **else** a = a – b }; c = a;

Пример. Обмен элементов массива

Программа Swap(a: a') обмена элементов массива. $a_1, a_2, \ldots, a_n, a_1=0$. Нулевой элемент a_1 обменивается с любым другим ненулевым элементом, если такой существует.

метод Гаусса-Жордана

nat n; // n>0
type natn = 1..n;
type Arn = **array**(**real**, natn);
formula pSwap(Arn a) = n > 0 & a[1] = 0;
formula qSwap(Arn a, a') =
    ∃ natn j. (a[j] ≠ 0 & a' = a **with** (1: a[j], j: 0)) ∨
    a' = a & ∀ natn j. a[j] = 0;

Спецификация программы:

Swap(Arn a: Arn a') **pre** pSwap(a) **post** qSwap(a, a');

Отметим, что спецификация неоднозначна.

Swap(Arn a: Arn a') **pre** pSwap(a) **post** qSwap(a, a');

pSwap(Arn a) = n > 0 & a[1] = 0;

qSwap(Arn a, a') = ∃ natn j. (a[j] ≠ 0 & a' = a **with** (1: a[j], j: 0)) ∨
a' = a & ∀ natn j. a[j] = 0;

Сведение к более общей задаче Swap1:

p – индекс очередного элемента массива a. Предполагается, что первые p элементов массива a – нулевые, и требуется обменять первый элемент с ненулевым элементом массива a.

formula pSwap1(natn p, Arn a) = n > 0 & ∀ j=1..p. a[j] = 0;

Swap1(natn p, Arn a: Arn a') **pre** pSwap1(p, a) **post** qSwap(a, a');

Сведение при p = 1 : pSwap(a) ≡ pSwap1(1, a),

Swap(Arn a: Arn a') **pre** pSwap(a) **post** qSwap(a, a')
{ Swap1(1, a: a') };

Swap1(natn p, Arn a**:** Arn a') **pre** pSwap1(p, a) **post** qSwap(a, a');

pSwap1(natn p, Arn a) ≅ n > 0 & p > 0 & p ≤ n & ∀ j=1..p. a[j] = 0;

qSwap(Arn a, a') = ∃ natn j. (a[j] ≠ 0 & a' = a **with** (1**:** a[j], j**:** 0)) ∨
a' = a & ∀ natn j. a[j] = 0;

Разбор случаев p = n и p < n.

1. Случай p = n. Массив a полностью нулевой, и единственно возможным решением является a' = a.

pSwap1(p, a) & p = n → a' = a

2. Случай p < n. **Разбор случаев** a[p+1] ≠ 0 и a[p+1] = 0.

2.1. Случай p < n и a[p+1] ≠ 0. Элемент a[p+1] можно обменять с a[1]. Получаем другой вариант решения:

pSwap1(p, a) & p < n & a[p+1] ≠ 0 →
a' = a **with** (1**:** a[p+1], p+1**:** 0) .

2.2. Случай p < n & a[p+1] = 0.

pSwap1(p, a) & p < n & a[p+1] = 0 → Swap1(p+1, a**:** a')

pSwap1(p, a) & p = n $\rightarrow$ a’ = a

pSwap1(p, a) & p < n & a[p+1] $\neq$ 0 $\rightarrow$

a’ = a **with** (1**:** a[p+1], p+1**:** 0) .

pSwap1(p, a) & p < n & a[p+1] = 0 $\rightarrow$ Swap1(p+1, a**:** a’)

Объединение трех вариантов решения дает программу:

```
Swap1(nat p, Arn a: Arn a’)
pre pSwap1(p ,a) post qSwap(a, a’) measure n - p
{           if (p = n) a’ = a
            else  if (a[p+1] = 0) Swap1(p+1, a: a’)
                  else a’ = a with (1: a[p+1], p+1: 0)
};
```

Программный синтез

H(x: y) **corr** [P(x), Q(x, y)] ≅ (1)

$\forall x.\ P(x) \Rightarrow [\forall y.\ \mathrm{R}(H)(x, y) \Rightarrow Q(x, y)\ ] \ \&\ \exists y.\ \mathrm{R}(H)(x, y)$

Программа однозначна, спецификация тотальна.

T0:

$\forall x,y.\ P(x)\ \&\ H(x: y) \Rightarrow Q(x, y)$;

$\forall x.\ P(x) \Rightarrow \exists y.\ H(x: y)$

H(x: y) corr [P(x), Q(x, y)]

Поскольку $\mathrm{R}(H) = H$, то достаточно найти тотальный вычислимый предикат H(x: y), удовлетворяющий

$$P(x)\ \&\ H(x: y) \Rightarrow Q(x, y) \qquad (2)$$

Предикат H(x: y) является результатом *абдукции*, т.е. *обратного вывода* из постусловия Q(x, y).

2. Предикатное программирование

Оптимизирующие трансформации предикатных программ

Предикатная программа транслируется на язык $\mathbf{P_{imp}}$ – императивное расширение языка **P** применением оптимизирующих трансформаций.

Базовые *трансформации*:

- замена хвостовой рекурсии циклом;
- подстановка тела программы на место ее вызова;
- склеивание переменных, реализующее замену нескольких переменных одной;
- кодирование алгебраических типов (списков, деревьев) с помощью массивов и указателей.

Оптимизация среднего уровня

Итоговая программа по эффективности не уступает написанной вручную и, как правило, короче.

Для функциональных языков не удалось достичь приемлемой эффективности даже с применением изощренных методов оптимизации.

Подстановка тела программы на место ее вызова

A(x: y) { S } — предикатная программа на императивном расширении языка P, а

A(e: z) — вызов предиката в теле программы B.

x, y, z — списки переменных, а e — список выражений. *Подстановка тела программы предиката на место вызова* A(e: z) есть замена вызова композицией:

| x | = | e |; { S }; | z | = | y | (5)

x и y становятся локальными переменными

Необходимо предварительное систематическое переименование переменных.

Исполнение (5) реализует исполнение вызова A(e: z).

|x| = |e| - *групповой оператор присваивания*

раскрытие группового оператора

|a, b| = |c, a| реализуется присваиваниями: b = a; a = c

| x | = | e |; { S }; | z | = | y | (5)

| z | = | y | можно устранить, если провести замену в операторе S всех переменных из списка y на соответствующие переменные списка z.

Замена не всегда корректна.

Стратегия именования переменных предикатной программы должна быть такой, чтобы обеспечить минимальное изменение оператора S при его подстановке на место вызова A(e: z). С этой целью полезно использовать те же самые имена для заменяемых переменных.

Замена хвостовой рекурсии циклом

Это специальный случаем подстановки тела программы на место вызова

Рекурсивный вызов определяет *хвостовую рекурсию*, если:

- имя вызываемого предиката совпадает с именем предикатной программы, в теле которого находится вызов; результаты вызова совпадают с результатами – формальными параметрами
- вызов является последней исполняемой конструкцией в предикатной программе, содержащем вызов.

last(S) - множество последних исполняемых конструкций в операторе S. Тогда: last(A; B) = last(B), last(**if** (C) A **else** B) = last(A) ∪ last(B), last(D(e: z)) = {D(e: z)}, last(A || B) = ∅. Однако если A || B реализуется последовательным исполнением A и B, например, как B; A, то last(A || B) = last(A).

Умн(**nat** a, b**:** **nat** c) { **if** (a = 0) c = 0 **else** c = b + Умн(a – 1, b) }

не хвостовая рекурсия!!

GCD(**nat** a, b**:** **nat** c) { **if** (a = b) c = a **else if** (a < b)

GCD(a, b - a**:** c) **else** GCD(a - b, b**:** c) }

это хвостовая рекурсия

Рекурсивная программа B(x: y) { K } и вызов B(g: y) с хвостовой рекурсией внутри оператора K.

Подставим тело программы B на место вызова B(g: y). Результатом подстановки является композиция: | x | = | g |; { K }. Обозначим через K’ оператор, полученный заменой в K вызова B(g: y). Можно заменить в K’ второе вхождение K передачей управления на начало оператора K’. В итоге программа B преобразуется к виду: B(x: y) {M: K’’ }, где K’’ получается из K заменой вызова B(g: y) парой операторов: | x | = | g |; **goto** M. Данное преобразование есть трансформация *замены хвостовой рекурсии циклом*.

Предикатную программу B будем представлять в виде B(x: y) {**loop** { K’’’ }}, где K’’’ получается из K заменой всякого хвостового вызова B(g: y) оператором | x | = | g |, а в конце каждой ветви оператора K, не завершающейся рекурсивным вызовом, вставляется оператор выхода из цикла **break**.

```
GCD(nat a, nat b: nat c) {
M:    if (a = b) c = a
      else  if (a < b) {|a, b| = |a, b - a|; goto M}
      else  {|a, b| = |a - b, b|; goto M}
}
```

Раскроем групповые операторы присваивания, а также заменим фрагмент с операторами перехода на цикл **loop**. Получим:

```
GCD(nat a, nat b: nat c) {
   loop {
      if (a = b) {c = a; break; }
      if (a < b) b = b – a
      else a = a – b
   }
}
```

```
while (a != b) { if (a < b) b = b – a else a = a – b };   c = a;
```

Склеивание переменных

Задачи экономии памяти в классических работах А.П. Ершова, С.С. Лаврова, В.В. Мартынюка.

Трансформация *склеивания переменных* - замена всех вхождений одной переменной на другую.

$a \leftarrow b, c$ - замена всех вхождений имен b и c на имя a.

Склеиваются переменные одного типа, между которыми имеется информационная связь. В каждом операторе программы определяются аргументы и результаты оператора.

Условия корректности, гарантирующих эквивалентность предикатной программы до и после проведения склеивания: Результат оператора a может быть склеен с любым аргументом b соответствующего типа при условии, что аргумент b не используется далее после завершения исполнения оператора, т.е не является живым после выполнения оператора, в котором проводятся склеивания.

Умолчания: $c \leftarrow c'$;

Пример.Трансформация программы обмена элементов массива

```
Swap(a:  a') { Swap1(1, a: a') };
Swap1(p, a: a')
{       if (p = n) a' = a
        else    if (a[p+1] = 0) Swap1(p+1, a: a')
                else a' = a with (1: a[p+1], p+1: 0)            };
```

Трансформация склеивания a ← a'

```
Swap(a: a) { Swap1(1, a: a) };
Swap1(p, a: a)
{       if (p = n) a = a
        else  if (a[p+1] = 0) Swap1(p+1, a: a)
                else a = a with (1: a[p+1], p+1: 0)
};
```

Замены хвостовой рекурсии циклом с раскрытием группового оператора присваивания:

```
Swap1(p, a:  a)
{ loop {
      if (p=n) { a = a; break;}
      else      if (a[p+1] = 0) p = p+1
                else {a = a with (1: a[p+1], p+1: 0); break;}
  }                                          };
```

Трансформация упрощения: замена a = a пустым оператором. Условие p = n втягивается в заголовок цикла.

```
Swap1(p, a:  a)
{   while (p!=n) {
          if (a[p+1] = 0) p = p+1
          else { a = a with (1: a[p+1], p+1: 0) ; break; }
    }                          };
```

```
Swap1(p, a: a)
{  while (p!=n) {  if (a[p+1] = 0) p = p+1
                   else {a = a with (1: a[p+1], p+1: 0) ; break;}
   }              };
```

Вынесение оператора p = p+1 перед условным оператором. Модифицируется условного оператора – трансформация оформления. Раскрытие операции **with**

```
Swap1(p, a: a)
{  while (p!=n) { p = p+1; if (a[p]!=0) { a[1] = a[p]; a[p] = 0; break }
   }   };
```

Отметим, что вынесение оператора p = p+1 можно было бы реализовать в виде следующей предикатной программы:

```
Swap1(p, a: a')
{  if (p = n) a' = a
   else { nat p1 = p+1;
          if (a[p1] = 0) Swap1(p1, a: a') else a' = a with (1: a[p1], p1: 0)
 }  };
```

Поскольку программа Swap1 более не рекурсивна, подставим ее внутрь программы Swap.

```
Swap(a:  a)
{    nat p; | p, a | = | 1, a|;
   while (p!=n) { p = p+1; if (a[p]!=0) { a[1] = a[p]; a[p] = 0; break }
    }
}
```

Раскроем групповой оператор присваивания и втянем оператор p = 1 в заголовок цикла. Получим итоговую программу.

```
Swap(a: a)
{ for ( nat p = 1; p!= n; )
        { p = p+1; if (a[p]!=0) { a[1] = a[p]; a[p] = 0; break } }
}
```

Трансформация кодирования рекурсивных структур

Пример. Программа суммирования sum(s: c) эл-тов посл-ти s.

```
type listR = list (real);
formula SUM(listR s:  real) =  s = nil ?  0 : s.car + SUM(s.cdr);
sum(listR s: real c) post c = SUM(s)
{ switch (s){ case nil:  c = 0
              case cons(h, t): c = h + sum(t)   }
};
```

nil?(s) **or** cons?(s) = **true**

```
sum(listR s: real c) post c = SUM(s)
{  if (nil?(s))  c = 0
   else {{h = s.car || t = s.cdr}; c = h + sum(t)}
};

sum(listR s: real c) post c = SUM(s)
{  if (s = nil)  c = 0  else c = s.car + sum(s.cdr) };
```

```
type listR = list (real);
formula SUM(listR s:  real) =  s = nil ?  0 : s.car + SUM(s.cdr);
sum(listR s: real c) post c = SUM(s)
{  if (s = nil)  c = 0 else c = s.car + sum(s.cdr) };
```

Обобщение задачи SUM

```
formula SUMG(listR s, real d: real c) = c = SUM(s) + d;
sumG(listR s, real d: real c)  post c = SUMG(s, d);

sum(listR s: real c) post c = SUM(s)
{ sumG(s, 0: c) };

sumG(listR s, real d: real c) post c = SUMG(s, d)
    measure len(s)
{ if (s = nil ) c = d else sumG(s.cdr, d + s.car : c) };
```

```
sumG(listR s, real d: real c) post SUMG(s, d, c)
    measure len(s)
{ if (s = nil ) c = d else sumG(s.cdr, d + s.car : c) };
```

Склеивание c ← d.

```
sumG(listR s, real c:  c)
{ if (s = nil ) c = c else sumG(s.cdr, c + s.car : c) }
```

Замена хвостовой рекурсии циклом.

```
sumG(listR s, real c: real c)
{ loop { if (s = nil ) {c = c; break; }
              else | s, c | = |s.cdr, c + s.car |       } }
```

При раскрытии группового оператора меняется порядок присваивания параметрам s и c.

```
sumG(listR s, real c: real c)
{ while (s != nil) { c = c + s.car; s = s.cdr }
```

```
sum(listR s: real c) { sumG(s, 0: c) };

sumG(listR s, real c: real c)
{ while (s != nil) { c = c + s.car; s = s.cdr }
```

Подстановка тела программы на место вызова

```
sum(listR s: real c) {
  real c = 0; while (s != nil) { c = c + s.car; s = s.cdr }
}
```

Кодирование списка вырезкой массива

Кодирование начального и текущего состояния списка S

type SEQR = **array** (**real**, 0..n);

SEQR S;

Начальное — S = S[0..n], текущее — S[j..n].

int j = 0;

Кодирование операций:

s = nil	→	j > n	s.car	→	S[j]
s != nil	→	j <= n	s = s.cdr	→	j = j + 1

Итоговая программа

real c = 0; **for** (**int** j = 0; j <= n;) { c = c + S[j]; j = j + 1 }

Исходная программа:

real c = 0; **while** (s != nil) { c = c + s.car; s = s.cdr }

Кодирование списка через указатели

type LTP = **struct** (**real** car, LTP *cdr);

LTP *S;

s → S

Кодирование операций:

s = nil → S = null

s != nil → S != null

s.car → S->car

s = s.cdr → S = S->cdr

Итоговая программа

real c = 0; **while** (S != null) { c = c + S->car; S = S->cdr }

Исходная программа:

real c = 0; **while** (s != nil) { c = c + s.car; s = s.cdr }

Пример. Обращение списка

Результат обращения списка:

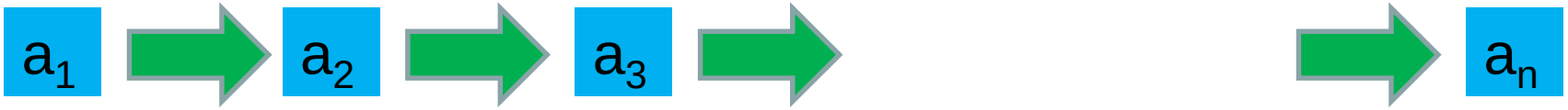
Односвязный список**:** **type** LTP = **struct** (T car, LTP *cdr);

type T; **type** LT = list(T);

formula reverse(LT s**:** LT) =

(s = nil)? s **:** reverse(s.cdr) + s.car;

ReverseIn(LT s**:** LT s') **pre** s ≠ nil **post** s' = reverse(s)



Обобщение задачи reverseIn:

$a_k, a_{k-1}, \ldots a_1$

type T; **type** LT = list(T);

formula reverse(LT s**:** LT) = (s = nil)? s **:** reverse(s.cdr) + s.car;

ReverseIn(LT s**:** LT s') **pre** s ≠ nil **post** s' = reverse(s)

ReverseG(LT s, u**:** LT s')

post s' = reverse(u) + s;

Здесь s = [a_k, a_{k-1}, … a_1]

ReverseIn(LT s**:** LT s') **pre** s ≠ nil **post** s' = reverse(s)

{ ReverseG([s.car], s.cdr**:** s') };

```
type T;     type LT = list(T);
formula reverse(LT s: LT) = (s = nil)? s : reverse(s.cdr) + s.car;
ReverseIn(LT s: LT s') pre s ≠ nil post s' = reverse(s)
{ ReverseG([s.car], s.cdr: s') };
ReverseG(LT s, u: LT s') post s' = reverse(u) + s {
    if (u = nil) s' = s
    else ReverseG(u.car + s, u.cdr: s')
};
```

Склеивание и замена хвостовой рекурсии циклом:

```
ReverseIn(LT s: LT s){ ReverseG([s.car], s.cdr: s) };
ReverseG(LT s, u: LT s){
    loop  { if (u = nil) { s = s; break }
            else |s, u| = |u.car + s, u.cdr|
     }                  };
```

Оформление цикла и подстановка на место вызова:

```
ReverseIn(LT s: LT s){
   LT u; |s, u| = |[s.car], s.cdr|;
   while (u != nil) { s = u.car + s;  u = u.cdr }
};
```

Раскрытие мультиприсваивания:

```
ReverseIn(LT s: LT s){
   LT u = s.cdr; s = [s.car];
   while (u != nil) { s = u.car + s;  u = u.cdr }
};
type LTP = struct (T car, LTP *cdr);
LTP *S;
```

s → S; u → U

Кодирование операций:

u != nil	→	U != null
s = [s.car]	→	S->cdr := null
u = s.cdr	→	U = S->cdr

Дополнительная переменная для устранения конфликта

```
ReverseIn(LT s: LT s){
   LT u = s.cdr; s = [s.car];
   while (u != nil) { z = u;  u = u.cdr; s = z.car + s}
};
```

Кодирование операций:

z.car + s → Z->cdr = S;

s = z.car + s → Z->cdr = S; S = Z

```
ReverseIn(LTP* S) {
   LTP* U = S->cdr;
   S->cdr = null;
   while (U != null)
         { LTP* Z = U; U = U->cdr; Z->cdr = S; S = Z }
};
```

2. Предикатное программирование

Пример. Сортировка простыми вставками

Пример. Сортировка простыми вставками

```
program SORT (type T, nat n) {  // T – тип элементов с "≤"
  import Total_order(T, "≤")
  type natn = 0 .. n;
  type Arn = array (T, natn);
.....
}
```

Массивы вместо списков

Спецификация программы сортировки:

```
sort(Arn a: a') post perm(a, a') & sorted(a')

theory Sort {
   formula sorted(Arn a) =  ∀ natn i, j. i < j ⇒  a[i] ≤ a[j];
};
```

theory Perm {

type F = **subtype** (**predicate** (natn**:** natn) f**:** bijective(f));

formula perm(Arn a, b) = ∃ F f. ∀ natn j. a[j] = b[f(j)];

Arn a, b, c;

pe1: lemma perm(a, a);

pe2: lemma perm(a, b) & perm(b, c) ⇒ perm(a, c);

};

Сведение к более общей задаче sort1:

theory Sort { ……..

formula sorted(Arn a, natn m) = ∀ i, j = 0..m. i < j ⇒ a[i] ≤ a[j]

so1: lemma sorted(a, n) ⇒ sorted(a);

};

Более общая задача **:**

sort1(Arn a, natn m**:** Arn a’)

pre sorted(a, m) **post** perm(a, a’) & sorted(a’);

sort1(Arn a, natn m**:** Arn a’)

pre sorted(a, m) **post** perm(a, a’) & sorted(a’);

formula sorted(Arn a, natn m) = $\forall$ i, j = 0..m. i < j $\Rightarrow$ a[i] $\leq$ a[j]

sorted(a, 0) истинна !!!

sort(Arn a: Arn a’) **post** perm(a, a’) & sorted(a’)

{ sort1(a, 0**:** a') };

sort1(Arn a, natn m: Arn a’)

pre sorted(a, m) **post** perm(a, a’) & sorted(a’)

{ **if** (m = n) a’ = a

else { pop_into(a, m, a[m+1]**:** Arn c); sort1(c, m+1**:** a’)}

};

pop_into(Arn a, natn m, T e**:** Arn a’)

pre m < n & sorted(a, m) & e = a[m+1]

post perm(a, a’) & sorted(a’, m+1);

Необходимо обобщение pop_into

```
sort1(Arn a, natn m: Arn a')
   pre sorted(a, m) post perm(a, a') & sorted(a')
{  if (m = n) a' = a
   else { pop_into(a, m+1, m, a[m+1]: Arn c);
          sort1(c, m+1: a')
        }                              //  для обновленной pop_into
};
```

Схема работы алгоритма pop_into

Спецификация обобщенной программы pop_into :

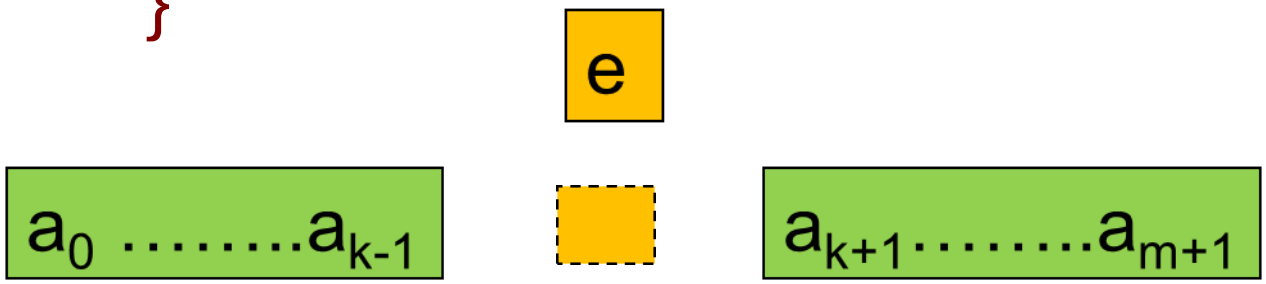
```
pop_into(Arn a, natn k, m, T e: Arn a')// k – пустая позиция
pre m < n & k > 0 & sorted(a, k-1) & sorted(a, k+1, m+1) &
          (k > m  or e < a[k+1] & a[k-1] <= a[k+1])
post perm(a with (k: e), a') & sorted(a', m+1);

formula sorted(Arn a, natn k, m) = ∀ i, j = k..m. i < j ⇒  a[i] ≤ a[j];
```

formula sorted(Arn a, natn k, m) =

$$\forall\ i, j = k..m.\ i < j \Rightarrow\ a[i] \le a[j];$$

pop_into(Arn a, natn k, m, T e**:** Arn a') // k – позиция дырки
pre m < n & k > 0 & sorted(a, k-1) & sorted(a, k+1, m+1) &
(k > m **or** e < a[k+1] & a[k-1] <= a[k+1])
post perm(a **with** (k**:** e), a') & sorted(a', m+1);

```
{ if (a[k-1] <= e) a' = a with (k: e)
  else { Arn b = a with (k: a[k-1]);
         if (k = 1) a' = b with (0: e)
         else pop_into(b, k-1, m, e: a')
       }
}
```



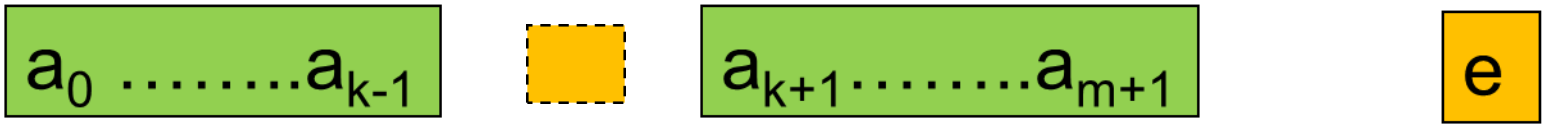
Недостаток алгоритма: если a[m] <= a[m+1] при вызове pop_into, то элемент a[m+1] сначала считывается в переменную e, а затем записывается назад в массив a

```
sort1(Arn a, natn m: Arn a')
   pre sorted(a, m) post perm(a, a') & sorted(a')
{ if (m = n) a' = a
  else { T e = a[m+1];
         if (a[m] <= e) sort1(a, m+1: a')
         else { pop_into(a, m+1, m, e: Arn c);
                sort1(c, m+1: a')
              }
       }
};
```

Изменился интерфейс pop_into

Улучшенная версия pop_into

```
pop_into(Arn a, natn k, m, T e: Arn a') // k – позиция дырки
  pre m < n & k > 0 & sorted(a, k-1) & sorted(a, k+1, m+1) &
      (k > m  or e < a[k+1] & a[k-1] <= a[k+1]) & a[k-1] > e
  post perm(a with (k: e), a') & sorted(a', m+1)
{  Arn b = a with (k: a[k-1]);
   if (k = 1) a' = b with (0: e)
   else if (b[k-2] <= e) a' = b with (k-1: e)
        else pop_into(b, k-1, m, e: a')
};
```



Склеивания

a ← a'

```
sort(Arn a: a) { sort1(a, 0: a) };
```

```
a ← a', c
sort1(Arn a, natn m: a) pre sorted(a, m)
{ if (m = n) a = a
  else { T e = a[m+1];
         if (a[m] <= e) sort1(a, m+1: a)
         else { pop_into(a, m+1, m, e: Arn a);
                sort1(a, m+1: a)    }
       }
}
a ← a', b
pop_into(Arn a, natn k, m, T e: a) // k – позиция дырки
{  a = a with (k: a[k-1]);
   if (k = 1) a = a with (0: e)
   else if (a[k-2] <= e) a = a with (k-1: e)
        else pop_into(a, k-1, m, e: a)
};
```

Замена хвостовой рекурсии циклом. Раскрытие групповых операторов и модификаторов.

Оформление циклов. Упрощения.

```
sort1(Arn a, natn m: a)
{  for( ; m != n; m = m+1) {
      T e = a[m+1];
       if (a[m] > e)  pop_into(a, m+1, m, e: Arn a);
       }
}

pop_into(Arn a, natn k, m, T e: a) // k – позиция дырки
{ for( ; ; k = k  - 1) {
     a[k] = a[k-1];
     if (k = 1) { a[0] = e; break; }
     else if (a[k-2] <= e) { a[k-1] = e; break; }
     }
};
```

Подстановка pop_into на место вызова в sort1 и sort1 в sort.

```
sort1(Arn a, natn m: a)
{  for( ; m != n; m = m+1) {
      T e = a[m+1];
       if (a[m] > e)          //pop_into(a, m+1, m, e: Arn a);
         |a, k, m, e| = |a, m+1, m, e|;
         for( ; ; k = k  - 1) {
           a[k] = a[k-1];
           if (k = 1) { a[0] = e; break; }
           else if (a[k-2] <= e) { a[k-1] = e; break; }
         }
    }
}
```

Подстановка sort1 на место вызова в sort. Упрощения.

```
sort(Arn a: a) {   //  sort1(a, 0: a)      //   |a, m| = |a, 0|;
   for( m = 0 ; !m = n; m = m+1) {
      T e = a[m+1];
       if (a[m] > e)
          for( k = m+1; ; k = k  - 1) {
            a[k] = a[k-1];
            if (k = 1) { a[0] = e; break; }
            else if (a[k-2] <= e) { a[k-1] = e; break; }
          }
    }
}
```

Возможны улучшения алгоритма

Предикатное программирование

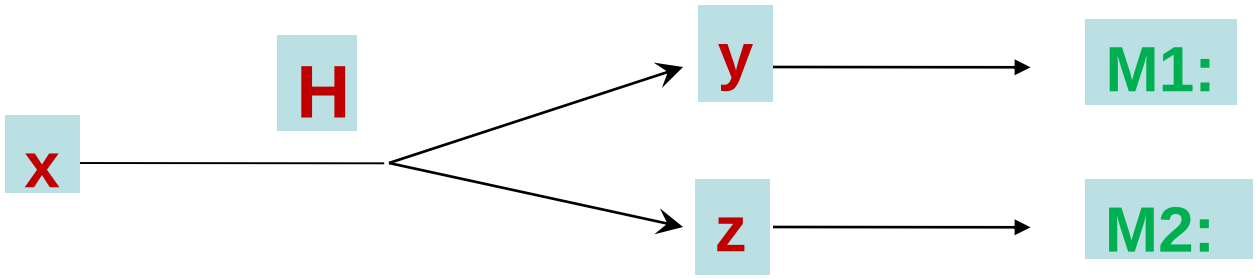
Гиперфункции

Гиперфункции

H(x: y) – программа H с аргументами x и результатами y

H(x: y : z) – *гиперфункция* с двумя *ветвями* результатов y и z

H(x: y #M1: z #M2) – *вызов гиперфункции* с разными переходами по ветвям



Исключения Java

elemTwo(list(int) s: int e, bool exists) ≡

post exists = (s ≠ nil & s.cdr ≠ nil) & (exists ⇒ e = s.cdr.car)

{ if (s = nil or s.cdr = nil) exsts = false

else { e = s.cdr.car || exists = true }

}

Взять 2-й элемент списка и выполнить программу B:

elemTwo(s: e, exists); if (exists) B(e…) else D(…)

Подставим тело elemTwo на место вызова.

if (s = nil or s.cdr = nil)

{ exist = false; if (exist) B(e…) else D(…) }

else {{e = s.cdr.car || exist = true };

if (exist) B(e…) else D(…)} Переменная exist – лишняя.

if (s = nil or s.cdr = nil) D(…) else { e = s.cdr.car; B(e…) }

```
type Choice = union (elem(int e), empty);
elemTwo(list(int) s: Choice ch)
{   if (s = nil or s.cdr = nil) ch = empty
    else   ch = elem(s.cdr.car)
}

elemTwo(s: ch);
       switch (ch) { case elem(e): B(e…) case empty: D(…) }


hyper elemTwo(list (int) s : int e #1: #2 )
pre 1: s ≠ nil & s.cdr ≠  nil post 1: e = s.cdr.car
{   if (s = nil  or  s.cdr = nil)   #2
    else   { e = s.cdr.car   #1  }
};
elemTwo(s: e #L1: #L2)  case L1: B(e…) case L2: D(…);
```

Определение гиперфункции :

A(x**:** y #1**:** z #2) **pre** P(x) **pre** 1: C(x)

post 1: S(x, y); **post** 2: R(x, z)

{ K(x: y, z) };

Блок {K(x: y, z)} — тело гиперфункции, P(x) — общее предусловие, C(x) — *предусловие первой ветви* гиперфункции, S(x, y) — *постусловие первой ветви*, R(x, z) — *постусловие второй ветви*. Метки 1 и 2 являются *метками ветвей*.

Метки ветвей в определении предиката можно опустить:

A(x**:** y**:** z) **pre** P(x); **pre** 1: C(x) … { … };

A(x**:** y #1**:** z #2) **case** 1: B(y**:** u) **case** 2: D(z**:** u)

case 1**:** B(y**:** u) и **case** 2**:** D(z**:** u) — *операторы продолжения ветви*, а вместе — *обработчик ветвей*.

Кодирование гиперфункций

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

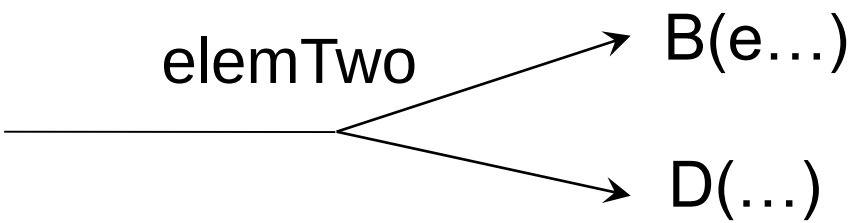
Определение гиперфункции A может быть представлено предикатной программой:

A(x**:** y, z, c) **pre** P(x) **post** c = C(x) & Q(x, y, z) { … };

$Q(x, y, z) \cong (C(x) \Rightarrow S(x, y)) \;\&\; (\neg C(x) \Rightarrow R(x, z))$

Операторы перехода #1 и #2 в теле K заменятся, соответственно, операторами c = **true** и c = **false**.

```
elemTwo(list (int) s : int e : )
pre 1: s ≠ nil & s.cdr ≠ nil
post 1: e = s.cdr.car
{  if (s = nil  ∨  s.cdr = nil)   #2
   else        { e = s.cdr.car   #1  }
};

elemTwo(list (int) s : int e  #1 : #2)

      elemTwo(s: e: )  case B(e…) case D(…)
```



Пример. Определим гиперфункцию Comp следующей спецификацией:

```
Comp(list (int) s : : int d, list (int) r)
pre 1: s = nil
post 2: d = s.car & r = s.cdr;
```

Построим определение предиката elemTwo через гиперфункцию Comp без полей car, cdr и конструктора nil.

```
elemTwo(list (int) s: int e #1: #2)
pre 1: s != nil & s.cdr != nil  post 1: e = s.cdr.car
{  int e1; list (int) s1, s2;
   Comp(s: : e1, s1)
    case  #2
    case {
            Comp(s1: : e, s2)
            case #2
            case #1
    }
};
```

Если оператор продолжения ветви содержит только оператор перехода, то оператор перехода переносится в конец соответствующей ветви вызова гиперфункции.

```
elemTwo(list (int) s: int e #1: #2)
pre 1: s != nil & s.cdr != nil post 1: e = s.cdr.car
{   Comp(s: #2: int e1, list (int) s1 #M2)
     case   M2:  Comp(s1: #2: e, list(int) s2 #1);
};
```

Реализован перенос описаний локальных переменных e1, s1 и s2 в позиции их *определения*.

Если обработчик ветвей состоит из единственного оператора продолжения ветви, а предшествующий оператор не имеет нормального завершения исполнения, то обработчик ветвей вырождается. В этом случае оператор продолжения ветви устраняется, а находящийся в нем оператор композируется с предыдущим оператором в виде оператора суперпозиции.

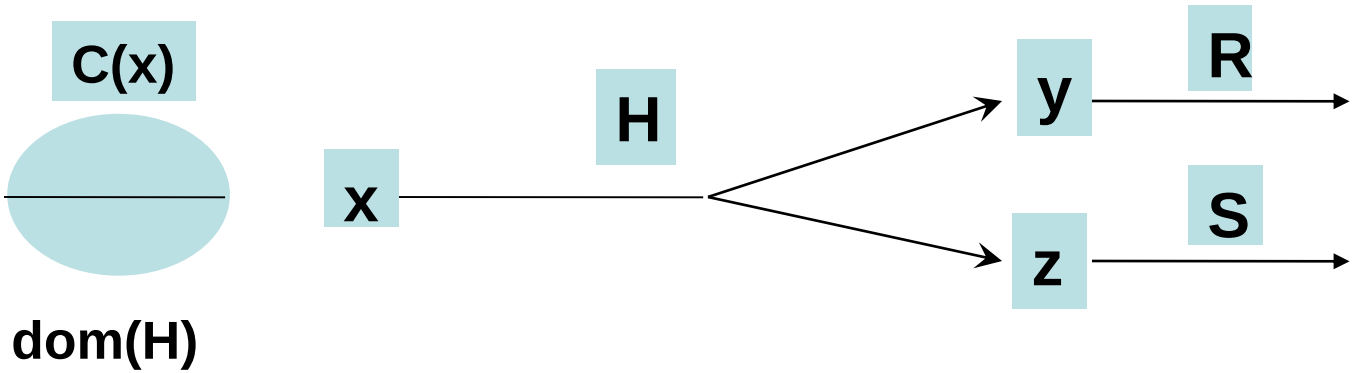
```
elemTwo(list (int) s: int e #1: #2)
pre 1: s != nil & s.cdr != nil post 1: e = s.cdr.car
{  Comp(s: #2:  _, list (int) s1);
   Comp(s1: #2: e, _ #1);
};
```

Вхождения локальных переменных e1 и s2 заменены стандартным именем “_”, означающим, что результат игнорируется при исполнении.

$$H(x\colon Y\ y,\ Z\ z) \equiv (C(x) \Rightarrow P(x, y)) \ \&\ (\neg C(x) \Rightarrow Q(x, z))$$

Гиперфункция и оператор *продолжения ветвей*

$H(x\colon y : z) \equiv (C(x) \Rightarrow P(x, y)) \ \&\ (\neg C(x) \Rightarrow Q(x, z))$
H(x: y : z) case R(y: …) case S(z: …)



ветвь гиперфункции

Свойства гиперфункций

```
if (φ(x))                          H(x…: … : …)
    { A;  B }        ⇒                 case  B
else { C;  D }                         case  D
```

H(X x…: … : …) ≡ { if (φ(x)) {A #1} else {C #2} }

Аналогов гиперфункций нет. Ближе к ним:

- **Метки в качестве параметров процедур в Алгол-60**
- **Механизм исключений в языках Java, C#**

Использование гиперфункций дает возможность более гибкой декомпозиции программы