Предикатное программирование

Гиперфункции

Пример. Вычисление суммы чисел в строке

Пример. Решение системы линейных уравнений

Дедуктивная верификация. Особенности

Моделе-ориентированная технология

Платформы дедуктивной верификации

Предикатное программирование

Гиперфункции (прод.)

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*.

Пример. Сумма чисел в строке

СуммаСтроки(**string** s: **nat** n) **post** P(s, n);

Предикат P(s, n) ≅ n – сумма чисел в строке s.

цифра(**char** c) ≅ c ∈ {’0’, ’1’, ’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’};

All(**string** s, H(**char**)) ≅ **if** (s=nil) **true else** H(s.car) & All(s.cdr)

число(**string** s) ≅ s ≠ **nil** & All(s, **lambda char** c. цифра(c))

val(**string** s: **nat** n) **pre** число(s) **post** Val(s, n); значение числа s.

P(**string** s, **nat** n) ≅

(s = **nil** ⇒ n = 0)

& (число(s) ⇒ n = val(s))

& (∀s1∈**string** ∀c∈**char**. ¬цифра(c) & s = c + s1 ⇒ P(s1, n)) &

& (∀s1,s2∈**string** ∀c∈**char**.

¬цифра(c) & число(s1) & s = s1 + c + s2 ⇒

∃m. P(s2, m) & n = val(s1) + m)

string есть **list**(**char**), а “+” обозначает конкатенацию строк.

Обобщение исходной задачи:

Сумма(**string** s, **nat** m**:** **nat** n) **post** ∃x. P(s, x) & n = m + x;

Дополнительный параметр m -- накопитель

Сведение к более общей задаче при m = 0 :

СуммаСтроки(**string** s**:** **nat** n) **post** P(s, n)

{Сумма(s, 0**:** n)};

Алгоритм: Находится первая цифра. Вычисляется число, начинающееся с этой цифры. Для оставшейся части строки действие алгоритма повторяется.

При нахождении первой цифры возможна ситуация, когда строка не содержит цифр. Данную ситуацию обозначим предикатом:

мусор(**string** s) ≅ All(s, **lambda** **char** c. ¬цифра(c))

В частности, предикат истинен для пустой строки.

мусор(**string** s) ≅ All(s, **lambda** **char** c. ¬цифра(c))

Нахождение первой цифры:

первыйЦифра(**string** s: : **char** e, **string** r)

pre 1: мусор(s)

post 2: ∃w. мусор(w) & s = w + e + r & цифра(e);

Отметим, что можно было бы использовать гиперфункцию вида: первыйЦифра1(**string** s: : **string** r), где r начинается с первой цифры, однако в реализации это приведет к повторному доступу к одной и той же литере в строке. В практике типично решение:

первыйЦифра2(**string** s: **char** e, **string** r)

Здесь отсутствие цифр в строке s кодируется некоторым значением e, отличным от цифры. Вред подобных трюков обычно проявляется для сложных программ. Решение в виде гиперфункции дает более короткую и эффективную программу.

Выделение числа в строке и его вычисление:

```
числоВстроке(char e, string r: nat v, string t)
pre цифра(e)
post ∃w. конецЧисла(r, w, t) & v = val(e + w);
конецЧисла(string r, w, t) ≅
            r = w + t  & (w = nil ∨ число(w)) &
            (t = nil ∨ ¬цифра(t.car));

Сумма(string s, nat m : nat n)
   post ∃x. P(s, x) & n = m + x measure len(s)
{   первыйЦифра(s: : char e, string r)
     case  n = m
     case  {  числоВстроке(e, r: nat v, string t);
              Сумма(t, m + v: n)
           }
};
```

```
первая Цифра(string s:  : char e, string r)
   pre 1: мусор(s)
   post 2: ∃w. мусор(w) & s = w + e + r & цифра(e)
   measure len(s);
{  if (s = nil)       #1
   else  if (цифра(s.car)) { r = s.cdr || e = s.car #2}
         else первая Цифра(s.cdr: #1 : e, r  #2)
};                                                  123456
                                                    p
```

Накопитель p для вычисления значения очередного числа. Обобщение для задачи числоВстроке :

```
взять Число(string r, nat p: nat v, string t)
    post ∃w. конец Числа(r, w, t) & val1(w, p, v);
val(s: n)  pre число(s) post val1(s, 0, n);
formula val1(stringw, nat p, n) =
     w = nil ? n = p : val1(w.cdr, p * 10 + valD(w.car), n)
```

где valD(e) — значение цифры e.

```
числоВстроке(char e, string r: nat v, string t)
pre цифра(e)
post ∃w. конецЧисла(r, w, t) & v = Val(e + w)
    { взятьЧисло(r, val(e): v, t) };

взятьЧисло(string r, nat p: nat v, string t)
post ∃w. конецЧисла(r, w, t) & val1(w, p, v) measure len(r)
{  if (r = nil)        {v = p || t = r}
   else
   {    {char b = r.car || string q = r.cdr};
        if (цифра(b)) взятьЧисло(q, p * 10 + val(b): v, t)
        else {v = p || t = q}
   }
};

конецЧисла(string r, w, t) ≅
      r = w + t  & (w = nil ∨ число(w)) & (t = nil ∨ ¬цифра(t.car);
```

Сумма: n ← m; s ← r, t; **Склеивания**

перваяЦифра: s ← r;

числоВстроке: s ← r, t; замена на s для упрощения

взятьЧисло: s ← r, q, t; v ← p;

```
Сумма(string s, nat n : nat n)
{  перваяЦифра(s: : char e, string s)
   case      n = n
   case  { числоВстроке(e, s: nat v, string s);
             Сумма(s, n + v: n)                    }
}
перваяЦифра(string s:  : char e, string s)
{  if (s = nil)      #1
   else   if (цифра(s.car)) { s = s.cdr || e = s.car #2}
          else перваяЦифра(s.cdr: #1 : e, s  #2)
}
```

```
числоВстроке(char e, string s: nat v, string s)
    { взятьЧисло(s, val(e): v, s) }
взятьЧисло(string s, nat v: nat v, string s)
{   if (s = nil) {v = v || s = s}
    else  { {char b = s.car || s = s.cdr };
            if (цифра(b)) взятьЧисло(s, v * 10 + val(b): v, s)
            else {v = v || s = s}                                  } };
```

Замена хвостовой рекурсии циклом. Удаляются операторы вида n = n.

```
СуммаСтроки(string s: nat n){Сумма(s, 0: n)};
Сумма(string s, nat n : nat n)
{   loop{ перваяЦифра(s: #M1 : char e, string s #2)
              case 2: {числоВстроке(e, s: nat v, string s); n = n + v}
    };    M1:
}
```

```
первыйЦифра(string s:  : char e, string s)
{   loop { if (s = nil)       #1
                else { { s = s.cdr || e = s.car};
                        if (цифра(e)) #2 else {}      }
}

числоВстроке(char e, string s: nat v, string s)
   { взятьЧисло(s, val(e): v, s) }
взятьЧисло(string s, nat v: nat v, string s)
{  loop {
        if (s = nil)    break
        else { {char b = s.car || s = s.cdr};
                if (цифра(b)) v = v * 10 + val(b)
                else break
        }
   }
};
```

Подставим программы взятьЧисло и перваяЦифра на место вызовов. Проведем очевидные упрощения.

```
Сумма(string s, nat n : nat n)
{  n = 0;
    loop {
        loop{   if (s = nil)       #M1
                else { e = s.car;  s = s.cdr;
                       if (цифра(e)) #2  else {  }  }
        }
        case 2: { числоВстроке(e, s: nat v, string s);
                  n = n + v }
    }
    M1:
}
```

```
числоВстроке(char e, string s: nat v, string s)
{  v = val(e);
   loop {
        if (s = nil)   break
        {char b = s.car || s = s.cdr};
        if (цифра(b)) v = v * 10 + val(b)
        else break
   }
};
```

Поскольку внутренний цикл в теле предиката Сумма не имеет нормального выхода, можно заменить оператор #2 на **break** и убрать скобки оператора продолжения ветви. Подставим тело программы числоВстроке на место вызова.

```
СуммаСтроки(string s: nat n)
{  n = 0;
   loop {
        loop {  if (s = nil)          return
                { char e = s.car || s = s.cdr };
                 if (цифра(e)) break
        }
        nat v = val(e);
        loop {
                if (s = nil)    break
                { char b = s.car || s =  s.cdr };
                if (цифра(b)) v = v * 10 + val(b)
                else break
        }
        n = n + v
}   }
```

Кодирование строк

Начальное значение строки s — вырезка S[m..p], текущее — S[j..p].

j — переменная, а m и p — константы.

type STR = **array** (**char**, M..N);

STR S;

int j = m;

Значения границ M и N должны быть достаточными, т.е. M ≤ m, p, j ≤ N. Кодирование операций:

s = nil	→	j > p
s.car	→	S[j]
s = s.cdr	→	j = j + 1

```
type STR = array (char, M..N);
STR S; nat n = 0;
for (int j = m; ;) {
   loop  {  if (j > p)        return
            char e = S[j]; j = j + 1;
            if (цифра(e)) break
   }
   nat  v = val(e);
   for (; j <= p; ) { char b = S[j]; j = j + 1;
                      if (цифра(b)) v = v * 10 + val(b)
                      else break
   }
   n = n + v
}
```

если строка s завершается цифрой, то проверка исчерпания строки реализуется дважды.

Имеется недостаток программы: если строка s завершается цифрой, то проверка исчерпания строки реализуется дважды. Чтобы исправить недостаток, следует вместо предиката числоВстроке(e, r: v, t) использовать гиперфункцию числоВстроке1(e, r: v1: v2, t), в которой первая ветвь реализуется при исчерпании входной строки r.

Возможен другой более простой алгоритм. однако менее эффективный.

```
int n=0, j=0;
while (j < N)
{
   e=S[j++];
   w = 0;
   while (j < N && String.IsNumber(e))
   {
        w = int.Parse(e) + w * 10;
        e = S[j++];
   }
   n = n + w;
}
```

Система линейных уравнений

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, i = 1..n \quad \equiv \text{LinEq(a, x, b)}$$

```
nat n; type I1n = 1..n;
type VEC = array (real, I1n);
type MATR = array(real, I1n, I1n);
formula SUM(nat k, predicate(nat: real) f: real) =
                                        k=0? 0: SUM(k-1, f) + f(k);
formula LinEq(MATR a, VEC b, x) =
  forall i=1..n. SUM( n, predicate(nat j: real) {a[i, j] * x[j]} ) = b[i];
Lin(MATR a, VEC b: VEC x :   )
pre 1: det(a) ≠ 0
post 1: LinEq(a, b, x)
{   TriangleMatr(a, b:  MATR c, VEC d :  #2);
    Solution(c, d:  VEC x ) #1
};
```

алгоритм Гаусса (метод Жордано)

formula postLin(MATR a, VEC b, MATR c, VEC d) =
    forall VEC x, y. LinEq(a, b, x) & LinEq(c, d, y) $\Rightarrow$ x = y

formula triangle(MATR a) =
($\forall$i=1..n. a[i, i] = 1) & $\forall$i,j=1..n. (i > j $\Rightarrow$ a[i, j] = 0)

TriangleMatr(MATR a, VEC b**:** MATR a', VEC b' **:**)
pre n > 0
pre 1**:** det(a) != 0
post 1: postLin(a, b, a', b') & triangle(a')
{ Jord(a, b, 1**:** a', b' #1**:** #2) };

Спецификация неоднозначна !

formula triaCol(**nat** k, MATR a) =

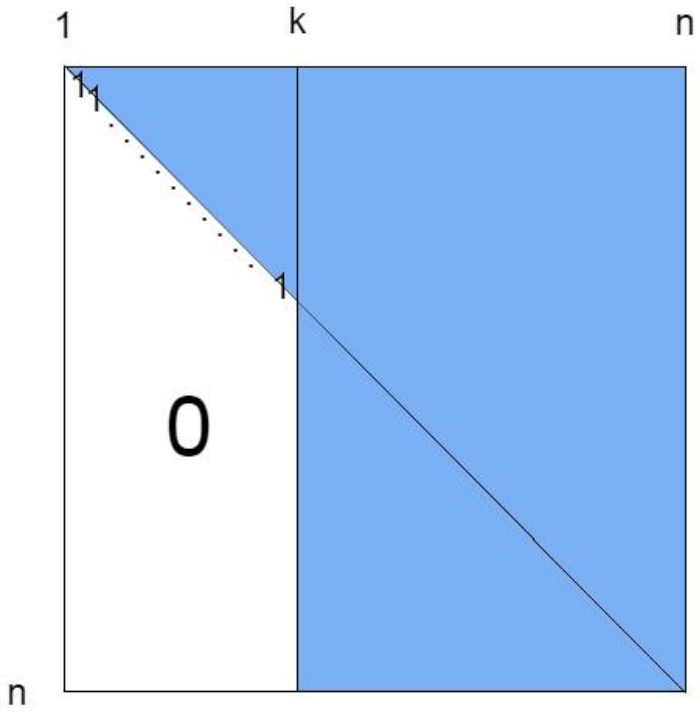
$\forall i=1..k-1.\ (a[i, i] = 1)\ \&\ \forall i = 1..n, j=1..k-1.\ (i > j \Rightarrow a[i, j] = 0)$

Jord(MATR a, VEC b, **nat** k**:** MATR a', VEC b' **:**)

pre $1 \leq k \leq n$ & triaCol(k, a)

pre 1**:** det(a) != 0

post 1: postLin(a, b, a', b') & triangle(a')



```
formula triaCol(nat k, MATR a) =
    ∀i=1..k-1. (a[i, i] = 1) & ∀i = 1..n,j=1..k-1. (i > j ⇒ a[i, j] = 0)

Jord( MATR a, VEC b, nat k: MATR a', VEC b' : )
pre 1 ≤ k ≤ n & triaCol(k, a)
pre 1: det(a) != 0
post 1: postLin(a, b, a', b') & triangle(a')
{  diagEl(a, b, k: MATR c, VEC d : #2);
   norm(k, c, d: MATR(n) e, VEC(n) f);
   subtrLines(k, e, f: MATR(n) g, VEC(n) h);
   if (k = n) { a' = g || b' = h #1 }
   else       Jord(g, h, k + 1:  a', b' #1 : #2)
};
```

```
diagEl( MATR a, VEC b, nat k: MATR a', VEC b' : )
pre  1 ≤ k ≤ n & triaCol(k, a)
pre   1: det(a) != 0
post 1: a'[k, k] != 0 & postLin(a, b, a', b')
{       if (a[k, k] != 0) {a' = a || b' = b #1}
        else perm(a, b, k, k+1: a', b' #1 :  #2)
}
```

```
perm( MATR a, VEC b, nat k, m:       MATR a', VEC b' : )
pre  1 ≤ k ≤ n & k<m ≤ n+1 &
          ∀i = k..m-1. a[i, k] = 0 &  triaCol(k, a)
pre 1: det(a) != 0
post 1: a'[k, k] != 0 & postLin(a, b, a', b')
{       if (m>n) #2
        else  if (a[m, k] != 0) {
              b' = b for (I1n j) {case k: b[m] case m: b[k]} ||
              a' = a for (I1n i, j)
                        {case (k, k..n): a[m, j] case (m, k..n): a[k, j]}
              #1
        } else  perm(a, b, k, m + 1: a', b' #1 : #2)
}
```

```
norm(nat k, MATR a, VEC b: MATR a', VEC b')
pre  1 ≤ k ≤ n & a[k, k] != 0 &  triaCol(k, a)
post a'[k, k] = 1 & postLin(a, b, a', b')
{  b' = b for (j) {case k: b[k] / a[k, k] } ||
    a' = a for (i, j) {case (k, k): 1
                       case (k, k+1..n): a[k, j] / a[k, k]}
}
subtrLines(nat k, MATR a, VEC b: MATR a', VEC b')
pre  1 ≤ k ≤ n & a[k, k] = 1 &  triaCol(k, a)
post triaCol(k+1, a) & postLin(a, b, a', b')
{  b' = b for (i) {case k+1..n: b[i] - b[k] × a[i, k]} ||
   a' = a for (i, j) { case (k+1..n, k): 0
                       case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k]  }
}
```

```
Solution( MATR a, VEC b:  VEC x )
pre  n > 0 & triangle(a)
post LinEq(a, b, x)
{   uniMat(a, b, n:  x ) }

formula nulColl(nat k, MATR a) =
                                        ∀j=k+1..n. ∀i=1..j-1. a[i, j] = 0
uniMat( MATR a, VEC b, nat k:  VEC x )
pre  n > 0 & triangle(a) & nulCol(k, a)
post LinEq(a, b, x)
{   if (k = 1) x = b
    else  {   subtrCol( a, b, k : MATR c, VEC d);
              uniMat(c, d, k-1:  x )
          }
}
```

subtrCol(MATR a, VEC b, **nat** k**:** MATR a’, VEC b’)

pre n > 0 & triangle(a) & nulCol(k, a)

post nulCol(k-1, a) & postLin(a, b, a’, b’)

{ b’ = b **for** (I1n i) {**case** 1..k-1**:** b[i] - b[k] × a[i, k]} ||

a' = a **for** (I1n i, j) {**case** (1..k-1, k): 0 } }

На первом этапе трансформации реализуются склеивания.

Lin: a <- c; b <- d, x;

TriangleMatr: a <- a’; b <- b’;

Jord: a <- c, e, g, a’; b <- d, f, h, b’;

diagEl: a <- a’; b <- b’;

perm: a <- a’; b <- b’;

norm: a <- a’; b <- b’;

subtrLines: a <- a’; b <- b’;

Solution: b <- x;

uniMat: a <- c; b <- d, x;

subtrCol: a <- a’; b <- b’;

На втором этапе реализуется замена хвостовой рекурсии циклами. Результатом проведения двух этапов трансформации является следующая программа:

```
Lin( MATR a, VEC b: VEC b : )
{  TriangleMatr(a, b:  a, b :  #2);
    Solution(a, b:  VEC b ) #1
}

TriangleMatr( MATR a, VEC b:  a,  b :  );
{  Jord(a, b, 1:  a, b #1:  #2) }
```

```
Jord( MATR a, VEC b, nat k:  a, b : )
{   for(;;) {
                diagEl( a, b, k:  a, b : #2);
                norm(k, a, b: a, b);
                subtrLines(a, b:  a, b);
                if (k = n) #1
                else        k = k + 1
    }
}
diagEl( MATR a, VEC b, nat k:  a,  b : )
{   if (a[k, k] != 0) #1
    else perm(a, b, k, k+1: a, b #1 : #2)
}
```

```
perm( MATR a, VEC b, nat k, m: a, b : )
{   for(;;) {
      if (m>n) #2
      else if (a[m, k] != 0) {
           b = b for (j) {case k: b[m] case m: b[k]} ||
           a = a for (i, j) {case (k, k..n): a[m, j] case (m, k..n): a[k, j]}
           #1
       } else  m = m + 1
    }
}

norm(nat k, MATR a, VEC b:  a,  b)
{   b = b for (j) {case k: b[k] / c[k, k] } ||
    a = a for (i, j) {case (k, k): 1 case (k, k+1..n): a[k, j] / a[k, k]}
}
```

```
subtrLines(nat k, MATR a, VEC b:  a, b)
{   b = b for (i) {case k+1..n: b[i] - b[k] × a[i, k]} ||
    a = a for (i, j) {case (k+1..n, k): 0
                      case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k] }
}
Solution( MATR a, VEC b:  b )
{   uniMat( a, b, n:  b ) }

uniMat( MATR a, VEC b, nat k:  b )
{   for(;;) {
        if (k = 1) return;
        subtrCol(a, b, k :  a, b);
        k = k - 1
    }
}
subtrCol( MATR a, VEC b, nat k:  a, b)
{   b = b for (I1n i) {case 1..k-1: b[i] - b[k] × a[i, k]} ||
    a = a for (I1n i, j) {case (1..k-1, k): 0 }
}
```

На третьем этапе проведем подстановку следующую серию подстановок тел определений на место вызовов:

perm -> diagEl;

subtrLines, norm, diagEl -> Jord -> TriangleMatr -> Lin;

subtrCol -> uniMat -> Solution -> Lin;

```
Lin( MATR a, VEC b: VEC b : )
{   for(nat k = 1; ; k=k+1) {
        if (a[k, k] != 0) #M1;
        nat m;
        for(m = k +1;;m = m + 1) {
            if (m>n) #2;
            if (a[m, k] != 0) break
        };
        {   b = b for (j) {case k: b[m] case m: b[k]} ||
            a = a for (i, j) {case (k, k..n): a[m, j]  case (m, k..n): a[k, j]}  };
M1:     {   b = b for (j) {case  k: b[k] / a[k, k] } ||
            a = a for (i, j) {case (k, k): 1 case  (k+1, k..n): a[k, j] / a[k, k]}    };
        {   b = b for (i) {case  k+1..n: b[i] - b[k] × a[i, k]} ||
            a = a for (i, j) {case (k+1..n, k): 0
                              case (k+1..n, k+1..n): a[i, j] - a[k, j] × a[i, k] }   };
        if (k = n) break
    }
    for(nat k = n; k != 1; k = k - 1) {
        b = b for (i) {case 1..k-1: b[i] - b[k] × a[i, k]} ||
        a = a for (i, j) {case (1..k-1, k): 0 }                          }
   #1
}
```

```
Lin( MATR a, VEC b:  b : )
{   for(nat k = 1; ; k=k+1) {
          if (a[k, k] != 0) #M1;
          nat m;
          for(m = k +1;;m = m + 1) {
               if (m>n) #2;
               if (a[m, k] != 0) break
          }
          {  real t = b[k]; b[k] = b[m]; b[m] = t ||
             for (nat j=k..n) {real u = a[k, j]; a[k, j] = a[m, j]; a[m, j] = u }       };
M1:       {  b[k] = b[k] / c[k, k] || a[k, k] = 1 ||
             for (nat j=k+1..n) a[k, j] = a[k, j] / a[k, k]                            };
          {  for (nat i=k+1..n) b[i] = b[i] - b[k] × a[i, k] ||
             for (nat i=k+1..n, k) a[k, j] = 0 ||
             for (nat i=k+1..n, nat j=k+1..n) a[i, j] = a[i, j] - a[k, j] × a[i, k]       };
          if (k = n) break
     }
     for(nat k = n; k != 1; k = k - 1) {
          for (nat i=1..k-1) b[i] = b[i] - b[k] × a[i, k] ||
          for (nat i=1..k-1) a[i, k] = 0
     }
    #1
}
```

На четвертом этапе раскроем операции с массивами

Предикатное программирование Платформы верификации. Сертификация

Особенности дедуктивной верификации

Дедуктивная верификация: 100%-ая гарантия корректности программы относительно спецификации

Применение – критические фрагменты программы в приложениях с высокой ценой ошибки.

Метод доказательства теорем. Почему 100% ?

Ошибка, если $\not\vdash$ **H(x: y) corr [P(x), Q(x, y)]**

100% при условии 100%-ой корректности сопутствующих инструментов и используемой технологии

Формальная семантика, правила доказательства корректности, транслятор, ОС, система команд, компьютер. Проблемы информационной безопасности

Сертифицированные трансляторы CompSert

Дедуктивная верификация объектного кода

Моделе-ориентированная технология

Разработка модели программы, ее анализ, верификация, построение программы на базе модели.
Симуляция (моделирование), если модель исполнима.

1. Model-based design
2. Model-driven engineering
3. Co-modeling & co-simulation

кибер-физические системы

Формальная модель eventB

Дедуктивная верификация свойств модели

Платформы дедуктивной верификации

Why3, Boogie, …

Язык спецификаций и программирования:
WhyML

Решатели: **Alt-Ergo, Z3, CVC4, Vampire…**

Системы интерактивного доказательства:
Coq, Isabelle/HOL, PVS, …

Системы моделирования и верификации:
Event-B , TLA+ …

Назначение предикатного программирования

Назначение: разработка и дедуктивная верификация программ высокой надежности и безопасности

Затраты: в 5 раз больше (против 20 в императивном пр-и)

Быстрая сортировка с двумя опорными элементами - основной алгоритм в библиотеке JDK языка Java

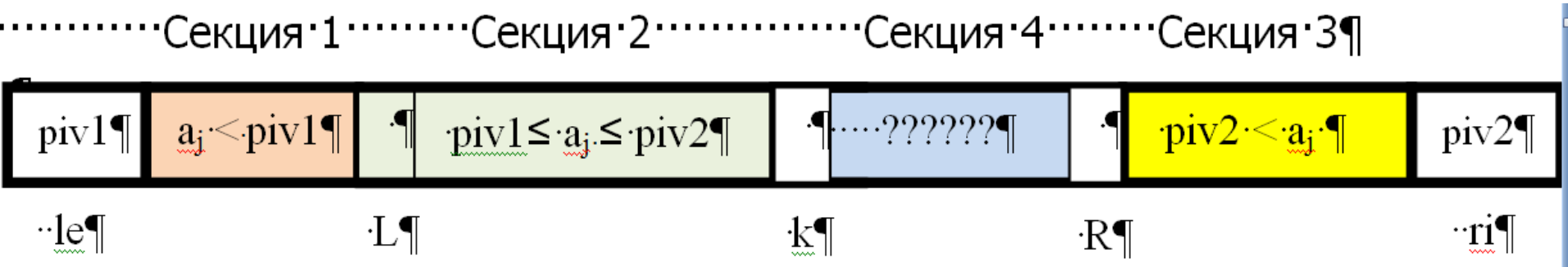
Дедуктивная верификация предикатной программы проведена в 10 раз быстрее верификации Java-программы

Универсальность: Для всякой императивной программы, принадлежащей классу программ-функций и решающей некоторую математическую задачу, можно построить эквивалентную предикатную программу. При этом императивная программа получается из предикатной программы применением некоторого набора оптимизирующих трансформаций.

Иллюстрации

```
formula partG(int le, ri, T piv1, piv2)(Arl a, LR L0, R0, k) =
        L0 > le & R0 < ri & L0 <= R0 +1 & L0 <= k &
        (∀int j. le<j<L0 ⇒ a[j]<piv1) & (∀int j. R0<j<ri ⇒ piv2<a[j]);
```

```
split(int le, ri, T piv1, piv2)(Arl a, LR L0, R0, k:   Arl a', LR L, R)
pre Psplit(a, L0, R0, k) post Qpart(a, a', R, L) measure R0 - L0 + ri - k {
    if (k > R0) Fin(L0, R0, a: a', L, R)
    else {  T ak = a[k];
            if (ak<piv1) split(a with(k: a[L0], L0: ak), L0+1, R0, k+1:  a', L, R)
            else if (ak>piv2) {
                    scanR(a, R0 : LR R1);
                    if (k > R1) Fin(L0, R1, a: a', L, R)
```



Java-программа – итог трансформаций

```
sort(int[] a, int le, ri) { < Сортировка малых массивов. Вычисление piv1 и piv2  >
  int L; int R = ri-1;            for(L = le+1; a[L]<piv1; L = L+1);
  for(int k = L ; ; k = k+1) {
      if (k > R) break;
      int ak = a[k];
      if (ak<piv1) { a[k] = a[L]; a[L] = ak; L=L+1 }
      else if (ak>piv2) {   for( ; a[R]>piv2; R=R–1);
                            if (k > R)  break;
                            int aR = a[R];
                            if (aR >= piv1) { a[k] = aR; a[R] = ak; R=R-1 }
                            else  { a[k] = a[L]; a[L] = aR; a[R] = ak; L=L+1; R=R-1 }     }
  };
  a[le] = a[L-1]; a[L-1] = piv1; a[ri] = a[R+1]; a[R+1] = piv2;
  {  sort(a, le, L-2); sort(a, R+2, ri); sort(a, L, R)  }                                  };
```

==========

```
Sp5: LEMMA Psplit(a, L0, R0, k, piv1, piv2)& k<=R0 & ak = a(k)& ak >= piv1 & ak>piv2 &
        QscanR(a, R0, R1, piv2) & k > R1 & L = L0 & R = R1 &
            a9 = a WITH [le:=a(L0-1),(L0-1) := piv1, ri:=a(R1+1), (R1+1) := piv2]
                IMPLIES Qpart(a, a9, L, R, piv1, piv2);
```