SMWare

Межпроцедурные анализы и оптимизации

Павел Ильин

Содержание

Agenda

Межпроцедурные оптимизации:

- Граф вызовов процедур (call graph)
- Подстановка процедур (inline)
- Частичная подстановка (partial inline & outlining)
- Пропагация констант, диапазонов, выравнивая, указателей и алиасов (constant, range, alignment, points-to, aliases propagation)
- Клонирование процедур (cloning)
- Замена стандартных процедур (replacing)
- Хвостовая рекурсия (tail recursion)
- Оптимизации доступа в память (memory optimizations)

Межпроцедурные анализы указателей:

- Результаты анализов: указатели и алиасы, свойства may- и must-point-to
- Неинициализированные указатели и неадресные значения
- Чувствительность к потоку управления
- Чувствительность к вызывающему контексту
- Модель обработки динамической памяти и составных объектов
- Учет языковых типов, требование компиляции всей программы
- Эффективный анализ на основе свойства транзитивности
- Точный анализ на основе ЧТФ

Граф вызовов процедур — Call graph

Граф вызовов – мультиграф, вершины – процедуры. Вершины proc1, proc2 соединены ориентированном ребром (proc1 , proc2), если процедура proc1 вызывается из proc2.

Трудности при построении:

- Раздельная компиляция
- Различные вызовы одной и той же процедуры (пометка точки вызова)
- Вызовы по косвенности и рекурсивные вызовы
- Глобальные метки и нелокальные переходы

Свойства графов:

- Динамический и статический
- Чувствительность к контексту (fully context-sensitive <-> context-insensive)

Оптимизации на графе вызовов:

- Удаление невызывающихся процедур («мертвый код»)
- Пропагация свойства процедур невозврата управления (abort, exit, longjmp) для переноса операции через операцию вызова
- Планирование на межпроцедурном уровне по «холодным» и «горячим» регионам – не просаживать буфер инструкций
- Анализ указателей – уточнение графа вызовов, превращение косвенных вызовов в прямые

Подстановка процедур

Inline

Подстановка процедуры – замена вызова процедуры на копию ее кода, связывание формальных параметров с фактическими, возвращаемых значений с переменными, в который происходит запись этих значений.

Исходный Вариант

```
int f1 (int q)
{   static int A;
    int b = q + 2;
    A++;
    return b;
}
int f2( int p)
{   if ( p) return f1(p + 1);
    else return 0;
}
int f( int a)
{   return f2(a);
}
```

После трансформации

```
static int A;
int f1 (int q)
{   int b = q + 2;
    A++;
    return b;
}
int f2( int p)
{   if ( p) return f1(p + 1);
    else return 0;
}
int f( int a)
{   int virt2, virt1, p, q, b;
    p = a;
    if ( p) {
        q = p + 1;
        b = q + 2;
        A++;
        virt1 = b;
        virt2 = virt1;
    } else {
        virt2 = 0;
    }
    return virt2; }
```

Подстановка процедур

Inline

Трудности:

- Несовпадение числа формальных и фактических параметров (функции с переменным числом параметров, нотация Керниган-Риччи)
- Нелокальные метки в вызывающем контексте – простановка глобальных меток в точках подстановки процедуры
- Коррекция профильной информации и результатов анализов
- Подстановка рекурсивной процедуры

Плюсы:

- Избавляемся от накладных расходов на вызов
- Специфицируем контекст для проведения локальных оптимизаций (оптимизация циклов)

Минусы:

- Рост кода
- Просадка кеша инструкций
- Давление на регистры
- Утяжеление процедуры – увеличение времени работы компилятора – неэффективный анализ

Поддержка в языках программирования - директива inline

Подстановка процедур

Inline

Схемы подстановки – разный обход графа вызовов:

- Начиная с main берем процедуру и подставляем в нее все вызываемые процедуры, которые могут быть подставлены.
- Начиная с процедур, не содержащих вызовы (листовых) подставляем их в процедуры, откуда они вызываются и могут быть подставлены.

Оптимизация листовых процедур (Leaf-Routine Optimization)

- упрощение пролога и эпилога вызова процедуры за счет того, что внутри не содержится вызовов.

Эвристики inline:

- Профильная информация (чем больше процедура вызывается, тем больше она годится для подстановки)
- Размер подставляемого кода (чем меньше процедура, тем больше она годится для подстановки)
- Процедуры внутри цикла надо подставлять для расширение возможностей оптимизации
- Если некоторые параметры процедуры константы, то надо подставлять для повышения эффективности оптимизаций благодаря пропагации констант

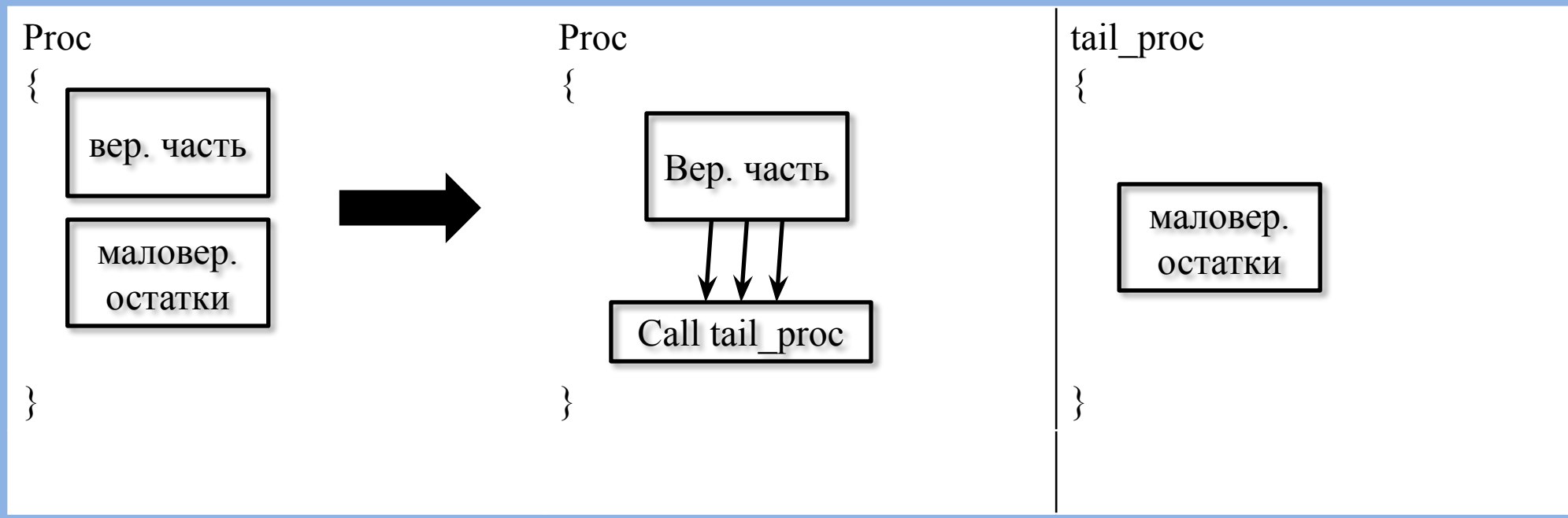
Частичная подстановка Partial inline and outlining

Частичная подстановка (partial inline) - процедуры с большим размером кода, не подходящим по эвристике для подстановки, но содержащие значительную вероятную часть.

Подготовка (outlining):

- В соответствии с профилем выделяется наиболее вероятная часть процедуры
- Выходы из вероятной части стягиваются в одну точку, в которой строится вызов новой процедуры называемой хвостовой (tail proc), содержащей маловероятные остатки

Outlining полезен для уменьшения размера процедур



Частичная подстановка Partial inline and outlining

Подстановка:

- Если у процедуры есть разрезанная копия, то подставляем вероятную часть и оставляем вызов хвостовой части

Процедура с большим размером кода, но при вызове она специфицируется параметрами до небольшого размера – необходим пропагатор.

```
Proc ( int args, int cond)
{
    If ( cond )
    {
       /* big code */

    } else
    {
      /* small code */

    }
}
```

Пропагатор

Propagator

- Пропагация констант (constant propagation) – пропагирует статически известные константные значения по представлению.

Исходный Вариант

```
int b = 1;
int g( void){
     return 0;
}
void f( int p)
{   int a;
    if ( a )     {
         a = g( );
     } else      {
         a = p;
     }
      b = a;
}
int main( void)
{
     f( 0);
     f( b);
     return b;
}
```

После трансформации

```
int b = 1;
int g( void) {
    return 0;
}
void f( int p)
{   int a;
    if ( a )  {
        a = 0;
    } else    {
        a = 0;
    }
    b = 0;
}
int main( void)
{
    f( 0);
    f( 0);
    return 0;
}
```

Пропагатор

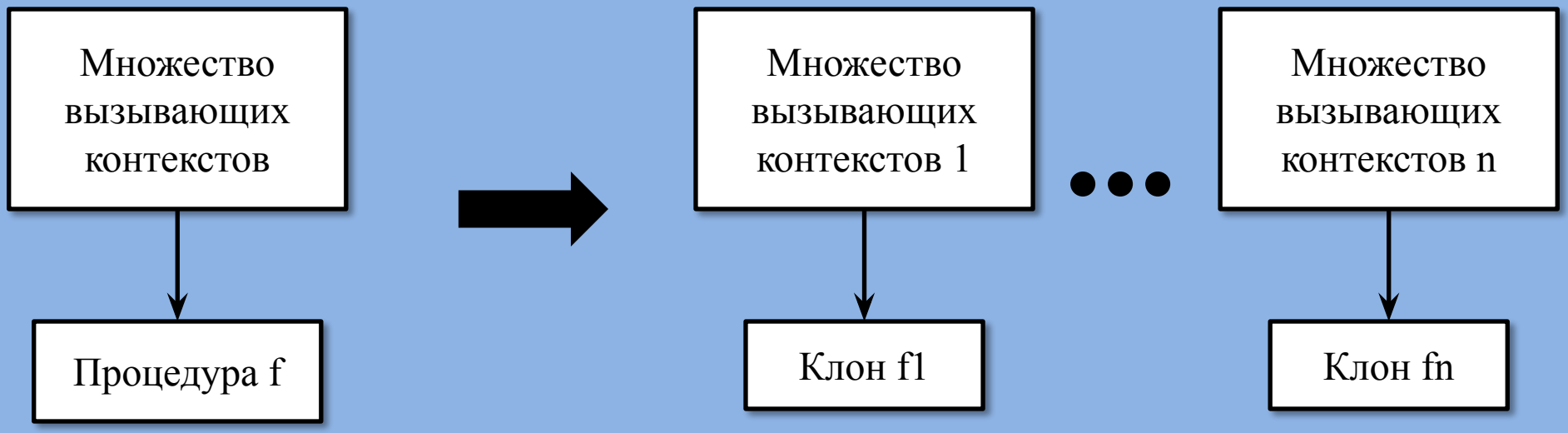
Propagator

- Пропагация выравниваний определяет для каждой операции доступа в память выровненность на определенное значение (фортран).
- Пропагация указателей определяет к каким именно объектам в памяти обращаются операции доступа в память.
- Пропагация алиасов определяет могут ли две операции доступа в память работать с одним фрагментом.
- Пропагация диапазонов определяет интервал или набор возможно принимаемых значений объектами программы

С помощью этих анализов возможно специфициовать поведение процедуры для опеределенных вызывающих контекстов – провести клонирование процедур.

Клонирование процедур

Cloning



Клон – спецификация процедуры для определенного вызывающего контекста

Клонирование подставляет вместо вызова процедуры вызов ее клона, соответствующего приходящим в эту точку вызова контекстам

После клонирования становится возможно подстановка.

Клонирование процедур

Cloning

Исходный Вариант

```
void f( int p)
 {
     if ( p )
         [часть1];
     else
         [часть2];
 }

 void g( )
 {
     f( 1);
     f( 0);
 }
```

После трансформации

```
void f( int p)
{
    if ( p )
        [часть1];
    else
        [часть2];
}
void f1( )
{
    [часть1];
}

void f2( )
{
    [часть2];
}

void g( )
{
    f1( );
    f2( );
}
```

Замена стандартных процедур

Replacing – замена вызова стандартной функции с некоторыми параметрами на эквивалентный код

Это позволяет сделать затем подстановку или вообще избавиться от вызова.

Для замены требуется условие доверия к стандартным библиотекам, errno и т.д.

Исходный Вариант	После трансформации
`printf("  ");`	`for(i=0; i<5; i++)putch(" ");`
Исходный Вариант	После трансформации
`Pow(2,n), pow(n, k) k=2,3,4;`	`1 << n; n*…*n;`

Хвостовая рекурсия

Tail recursion

Хвостовая рекурсия – преобразование рекурсии в цикл заменой рекурсивного вызова процедуры переходом к первой операции.

Для замены необходимо:

- После рекурсивного вызова сразу следует выход из процедуры
- При наличии профиля вызовов > 0
- Если глубина >= 2, то уменьшаем ее с помощью подстановки.

Исходный Вариант

```
int
 tail( int i)
 {
     if ( i > 32 )
     {
         return (i / 2);
     } else
     {
         return (tail(++i));
     }
 }
```

После трансформации

```
int
 tail( int i)
 {
 start:
     if ( i > 32 )
     {
         return (i / 2);
     } else
     {
         i++;
         goto start;
     }
 }
```

Оптимизации работы с памятью

Memory optimizations

Оптимизация размещения данных в памяти замена выделения многомерного массива на одномерный.

- Разбор шаблона – динамическое выделение многомерного массива
- Замена его на выделение одномерного массива
- Замена работы с многомерным массивом на одномерный
- Расположение исходных данных по профильной информации

Шаблон выделения памяти:

```
a = malloc( n1*sizeof(data**));
for ( i = 0; i < n1; i++ )
{
    a[i] = malloc( n2*sizeof(data*));
    for ( j = 0; j < n2; j++ )
        a[i][j] = malloc( n3*sizeof(data));
}
```

Оптимизации работы с памятью

Memory optimizations

Исходный Вариант

```
static double ***a;
static int n1,n2,n3;
void ALLOC()
{
 int i,j,k;
 a = (double ***)
malloc(n1*sizeof(double**));
 if (a == NULL) {
    printf("No memory\n");
    exit(1);
 }
 for (i=0; i<n1; i++)  {
          a[i] = (double
**)malloc(n2*sizeof(double*));
          if(a[i] == NULL) {
       printf("No memory\n");
              exit(1); }
          for (j=0; j<n2; j++) {
              a[i][j] = (double
*)malloc(n3*sizeof(double));
              if(a[i][j] == NULL) {
printf("No memory\n");
                  exit(1);
              }}}}
```

После трансформации

```
 typedef struct
  {   int* ptr;  int n0;  int n1;  int n2;
int n3;
      int n4;
  } my_ptr_t;

  static my_ptr_t a;
  static int n1,n2,n3;
  void
  ALLOC() {
      int i,j,k;
      a.ptr =
malloc(n1*n2*n3*sizeof(double));
      a.n0 = 1;
      a.n1 = n2;
      a.n2 = n2*n1;
      if (a.ptr == NULL) {
          printf("No memory\n");
          exit(1); }
      for (i=0; i<n1; i++)    {
          if(1 == NULL) {
              printf("No memory\n");
              exit(1); }
          for (j=0; j<n2; j++) {
              if(1 == NULL) {
                  printf("No memory\n");
                  exit(1);  }}}}
```

Оптимизации работы с памятью

Memory optimizations

Исходный Вариант

```
static void ARG(double ***b){
      int i,j,k;
      for (i=0; i<n1; i++) {
          for (j=0; j<n2; j++) {
              for (k=0; k<n3; k++) {
                  b[i][j][k]  = 2.0;
              }
          }
      }
 }
void  REF()
  {
      int i,j,k;

      for (i=0; i<n1; i++) {
          for (j=0; j<n2; j++){
              for (k=0; k<n3; k++) {
                  a[i][j][k] = 0.0;
              }
          }
      }
      ARG(a);
  }
```

После трансформации

```
 static void  ARG(my_ptr_t b){
      int i,j,k;
      for (i=0; i<n1; i++) {
          for (j=0; j<n2; j++) {
              for (k=0; k<n3; k++) {

*((double*)b+i*b.n2+j*b.n1+k*b.n0) = 2.0;
              }
          }
      }
 }
void  REF()
  {
      int i,j,k;

      for (i=0; i<n1; i++) {
          for (j=0; j<n2; j++){
              for (k=0; k<n3; k++) {

*((double*)a+i*b.n2+j*b.n1+k*b.n0) = 0.0;
              }
          }
      }
     ARG(a);
  }
```

Оптимизации работы с памятью

Memory optimizations

Исходный Вариант

```
 int main( )
  {
      n1 = 30;
      n2 = 30;
      n3 = 30;
      ALLOC();
      REF();
      exit(0);
  }
```

После трансформации

```
 int main( )
 {
      n1 = 30;
      n2 = 30;
      n3 = 30;
      ALLOC();
      REF();
      exit(0);
 }
```

Оптимизации работы с памятью

Memory optimizations

Достаточные условия для применения:

- Выделение памяти через стандартные функции malloc, calloc, memalign
- Не было взятия адреса на объекты, содержащие данные (в примере *static double ***a*) или указатели на данные (исключая возможно стандартные безобидные функции scanf(“%d”,&a[i][j][k])
- Отслежены все обращения к данным (для глобальных данных необходим анализ в режиме компиляции всей программы)

Плюсы:

- Замена нескольких операций доступа в память на одну (блочную)
- Улучшение работы индексного анализа, разрыв зависимостей
- Дополнительное переупорядочивание данных позволяет оптимизировать работу с памятью

Переупорядочивание данных (Reordering transformations)

- Транспонирование матрицы (разместить в памяти по столбцам)
- Массив структур -> структуру массивов (AoS -> SoA)
- Разрезание и переупорядочивание структур (Structure splitting and reordering)

Анализы указателей и алиасов

Points-to and aliases analysis

Результаты анализов:

- анализ указателей – на какие области памяти указывает каждый указатель в программе
- анализ алиасов – для любых пар указателей ответ указывают ли они на одну область памяти

Анализ указателей => анализ алиасов
Анализ алиасов => анализ указателей ?

p = &x; q = &p; *q = &y;
p ->{x,y}, q->{p}
<*p,x>,<*q,p>,<**q,y>,<**q,x>

Анализы may-point-to и must-point-to

May-Point-to and must-point-to analysis

Свойства результатов:

- анализ may-point-to – указатель может указывать на некоторую область памяти
- анализ must-point-to – указатель обязательно указывает на некоторую область памяти

```
int *p, *q, x, y;
p = &x;
q = &y;
if ( cond ) p = q;

/* q->{y} - may-point-to and must-point-to
   p->{x,y} - may-point-to */
```

Неинициализированные указатели и неадресные значения

Разыменование неинициализированного указателя

1) Некорректная семантика – завершение анализа
2) Разыменование в правой части присваивания (только чтение). Тогда считаем обращения к объекту в левой части (в который пишем) конфликтуют со всеми обращениями в память
3) Разыменование в левоей части присваивания – некорректная семантика и завершение
4) Игнорируем. Считаем, что результат разыменования не будет использован как адрес для обращения в память, иначе ошибка исполнения.

```
int *p, **q, x;

if ( cond )
  p = *q; /* 1,2 - p может указывать на что угодно */
else
  *q = &x; /* 3 - любой указатель может указывать на x */

**q; /* 4 – ошибка исполнения */
```

Неинициализированные указатели и неадресные значения

Неадресные значения (что считать инициализацией указателя)

1) Любое присваивание в указатель – инициализация
2) Присваивание только адресных значений – адреса объектов программы и возвращаемый процедурами выделения памяти

```
/* присваивание в p значения q */
void *p;
while ( p < q ) p+=1;

/* инициализация «неадресным» значением, которое может
оказаться адресом объекта */
q = ( void *) 0x800000cef;
```

Чувствительность к потоку управления

Учитывает или нет поток управления внутри анализируемых процедур (flow-sensitive и flow-insensitive)

- Увеличивает точность анализа
- Существенно увеличивает расходы на память
- Замедляет работу анализа
- Позволяет затирать в точке постдоминирующего присваивания предыдущие значения указателя – свойство сильного обновления (strong update)

```
if(cond){ p = &x;
          foo(p); /* p->{x} во flow-sensitive версии анализа
                     p->{x,y,z} во flow-insensitive версии */
} else{   p = &y;
          foo(p); /* p->{y} flow-sensitive версия анализа
                     p->{x,y,z} flow-insensitive версия */
}
/* p->{x,y} во flow-sensitive версии анализа,
   p->{x,y,z} flow-insensitive версия */
bar(p);
p = &z; /* p->{z} во flow-sensitive версии анализа
           (это свойство сильного обновления);
           p->{x,y,z} во flow-insensitive версии */
```

Чувствительность к вызывающему контексту

Разделение информации (вызывающий контекст), приходящей в процедуру по разным путям исполнения (context-sensitive)

- Нахождение нереализуемых путей (путь, который никогда не может возникнуть в процессе исполнения программы - unrealized paths problem)

```
int* f(int *a) { return g(a); }
int* g(int *b) { return b;
   /* b->{x,y} при context-insensitive анализе */
}
main(){
   int *p, x, y;
   if(cond)
      p = f(&x);
      /* При context-insensitive анализе p->{x,y},
         а при context-sensensitive p->{x} */
   else
      p = f(&y);
      /* При context-insensitive анализе p->{x,y},
         а при context-sensitive p->{y} */
```

Чувствительность к вызывающему контексту

- Вызывающий контекст однозначно определяется путем в графе вызовов
- Разные пути могут давать идентичные контексты, но в общем случае число различных контекстов – число всех путей к процедуре из main – экспонента от числа процедур программы
- Если есть рекурсивные циклы - число путей потенциально бесконечно

Необходимо объединять информацию, приходящую из разных вызывающих контекстов

- Объединять всю информацию – не чувствительный к контексту
- Различать контексты по точкам вызова процедур (на примере для g этого не достаточно)
- Клонирование процедур и граф активаций программы

Модель обработки динамической памяти

1) Вся динамическая память - один объект
2) Создавать уникальный объект динамической памяти по точке выделения
3) Путь в графе вызовов определяет объект – эскпонента
4) Некоторый начальный отрезок пути с эвристический длиной

```
char *p, *q;

p = malloc(num);
q = malloc(num);
...
for( i=0; i < num; i++)
   p[i] = q[i];
```

```
 main(){
   int *p = f1();
   int *q = f2(); ... }

f1(){ return my_malloc();}
f2(){ return my_malloc();}

my_malloc(){
  return malloc(N); }
}
```

Модель обработки динамической памяти

- Эвристически распознавать выделение памяти (newnode – выделение памяти)

```
typedef struct node {
   char type;
   struct node *next;} NODE;
NODE *newseg;
main() {
   NODE *n1=newnode(1);
   NODE *n2=newnode(2); ...
}
void findmem() {
   newseg = calloc(1,NSIZE);
...
}
```

```
 NODE *newnode(int type) {
   NODE *node;
   if(newseg == NIL) {
      findmem();
   node = newseg;
   newseg = node->next;
   node->type = type;
   return (node);
}
```

Обработка составных объектов

Различать элементы составных объектов или рассматривать как единое целое

```
struct
{ type * p1;
  type * p2;
} r;
/* Разные адреса &r.p1; &r.p2; */
```

Множество локализаций – подмножество целочисленной прямой, задается парой

$$\langle f, s \rangle \in \mathrm{Z} \times \mathrm{N}_0, f - шаг, s - смещение, \langle f, s \rangle = \{ f + i \cdot s | i \in \mathrm{Z} \}$$

Для однозначности s != 0 и 0<=f<s.

Множество локализаций позволяют производить теоретико-множественные операции над бесконечными множествами за константное время.
Объединение, разность – наименьшее множество, содержащее результат.
Требуется нахождение НОД шагов.
Не учитывает размера области памяти.

Обработка составных объектов

Соотвествие языковых типов и множества локализаций

Языковой тип	Выражение	Множество локализаций объекта
type *p	p	p <0,0>
struct {type *F;}r	r.F	r <f,0>
type *a[]	a[i]	a <0,s>
type *p;	*(p+X)	p <0,X>
struct {type *F[];} r	r.F[i]	r <f(mod s),s>
struct {type *F;} a[]	a[i].F	a <f,s>

Примечание: f – смещение поля F от начала структуры, s – размер элемента массива

Обработка составных объектов

Некорректное использование union

```
typedef union {
   struct { int x,y; } p;
   double z;  /* Поле для быстрой инициализации
                 полей x и y структуры. */
} COMPLEX;

f() {
   COMPLEX a;
   a.z = 0;  /* Инициализируем union как double */
   ...
   length(a.p.x,a.p.y); /* используем как struct */
   ...
```

Учет языковых типов

Учет типа для операций доступа в память

- Область памяти, независимо от типа, может содержат указатель любого типа – слаботипизированные языки C/C++
- Строгое следование языковым типам – не обработать многие реальные программы
- Использовать часть ограничений из стандарта языка – например позволять обращаться к int[] с помощью char *, но запрещать доступ по указателю на процедуру или константную строку.
- Использовать информацию о типе для эвристик – например через формальные параметры типа указатель ждать передачи указателей соответствующего типа.

Требование компиляции всей программы

- Необходимо наличие для анализа всей программы, за исключением возможно стандартных библиотек (флаг доверия к библиотекам)
- Динамически линкуемые библиотеки
- Помодульная компиляция – анализу необходим предполагать, что любая глобальная переменная может быть модифицированна, любая функция может быть вызвана с неизвестными параметрами

Пример эффективного анализа

Анализ на основе свойства транзитивности

Для отношения «алиасить» добавляется свойство транзитивности: если <*p,x> и <*p,y>, то <x,y>.

За один проход по представлению происходит сбор информации об алиасах и объединение объектов в соответствии с введенным отношением.

Транзитивность увеличивает скорость и уменьшает точность.

- Результаты – информация об алиасах
- May-aliased анализ
- Не чувствителен к управлению
- Не чувствителен к контексту
- Динамическую память различает по точкам вызова
- Элементы составных объектов не различаются
- Языковые типа не использует
- Требуется компиляция в режиме вся программа

Пример точного анализа

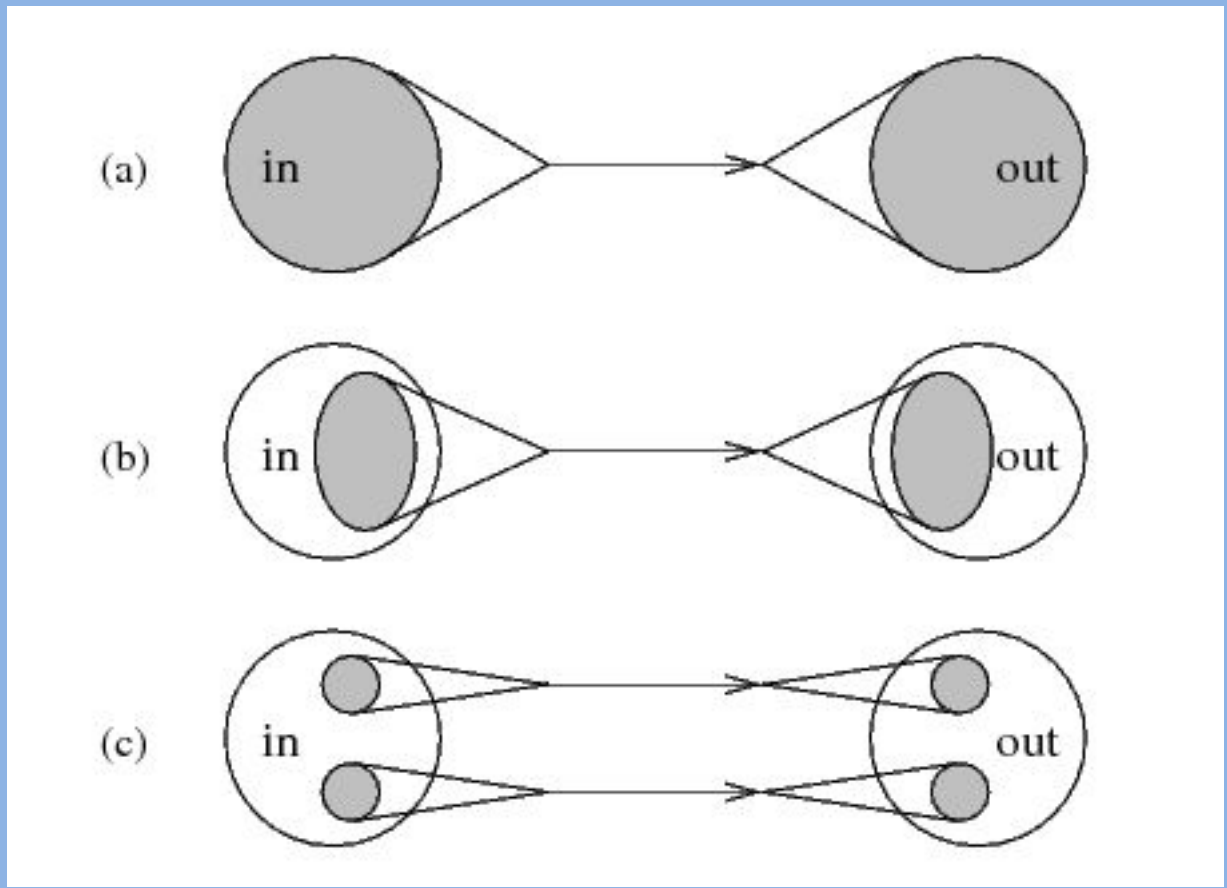
Анализ на основе ЧТФ

Трасферная функция процедуры определена для всех вызывающих контекстов, результат - измененный процедурой вызывающий контекст. (a)

Частичная трасферная функция определена не для всех возможных вызывающих контекстов.

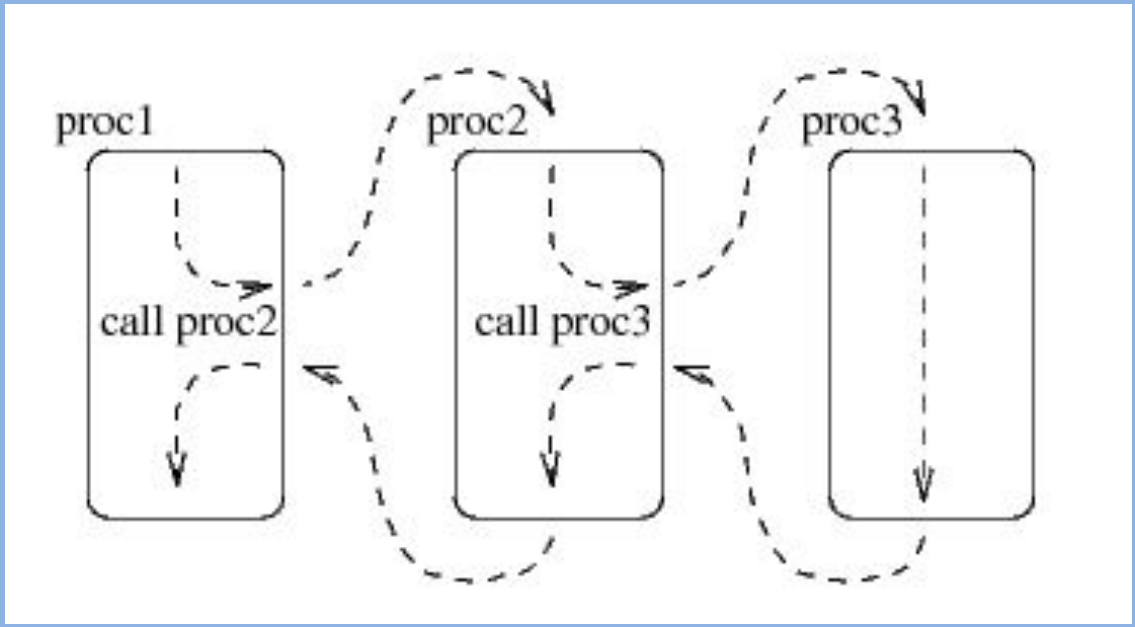
Обобщение одной ЧТФ на все вызывающие контектсты и получение таким образом ТФ - моновариантный анализ (b)

Построение нескольких ЧТФ, определенных в совокупности на всех вызывающих контекстах - поливариантный анализ (c)



Анализ на основе ЧТФ

Идея абстрактной интерпритации – начиная с main итерационно собирается информация о значении указателей и завершается, когда значения указателей в main перестают меняться.



Встречаем вызов, построение ЧТФ текущей процедуры приостанаваливается и переходим к анализу вызвынной процедуры (построению новой или обобщению уже имеющейся ЧТФ), затем продолжается построение ЧТФ текущей процедуры.

Эвристики – число ЧТФ, алгоритм сравнения контекстов, обощение области определения ЧТФ. Эффективность □□ точность.

Анализ на основе ЧТФ

Пример

```
f(int **p, int **q, int **r){
   *p = *q;
   *q = *r;
}
int x, y, z;
int *x0 = &x;
int *y0 = &y;
int *z0 = &z;

main( ){
   if(cond)
S1:      f(&x0, &y0, &z0); // 1-й вызывающий контекст f
   else if (cond')
S2:      f(&z0, &x0, &y0); // 2-й вызывающий контекст f
   else
S3:      f(&x0, &y0, &x0); // 3-й вызывающий контекст f
}
```

Как формализовать контекст с точки зрения интересующего нас свойства указывать?

Анализ на основе ЧТФ

Чтобы задать область определения ТФ используем points-to функции

```
p->{x,y}
p->{y}
```

```
p=&x;
q=&y;
if ( cond ) p=q;
```

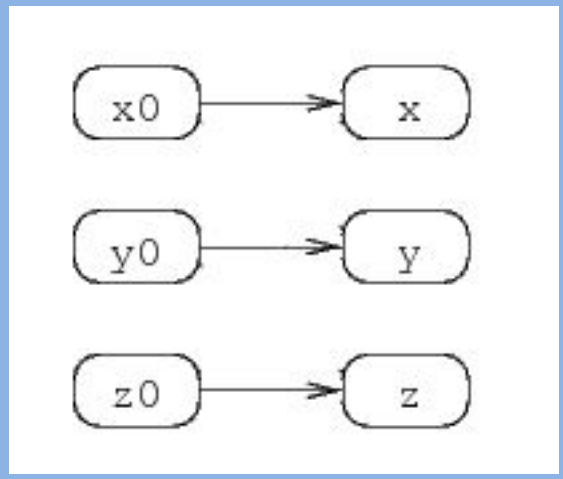
1-й и 2-й контекст одинаковые, как это отразить?

Пространстро имен (name space) для каждой ЧТФ:

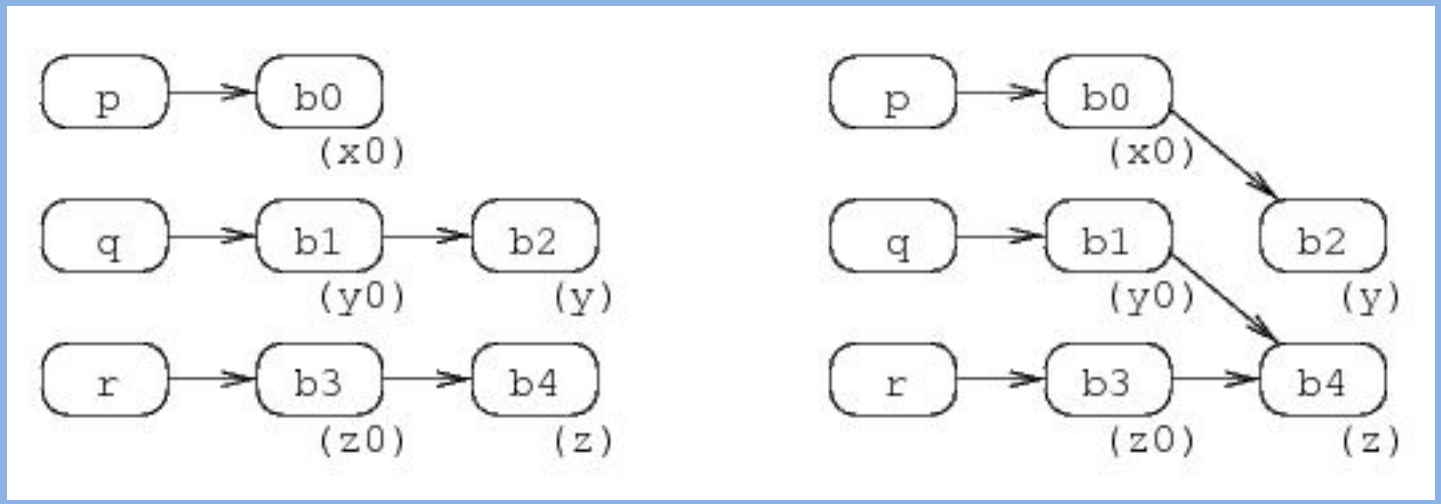
- Локальные блоки – не зависящие от вызывающего контекста (формальные параметры, локальные переменные, динамически выделяемая память)
- Нелокальные блоки – существуют в вызвающем контексте (глобальные переменные, объекты вызывающей процедуры)
- Block binding function – связывает нелокальные блоки в вызывающем и вызываемом контексте

Анализ на основе ЧТФ

Начальная points-to функция для main



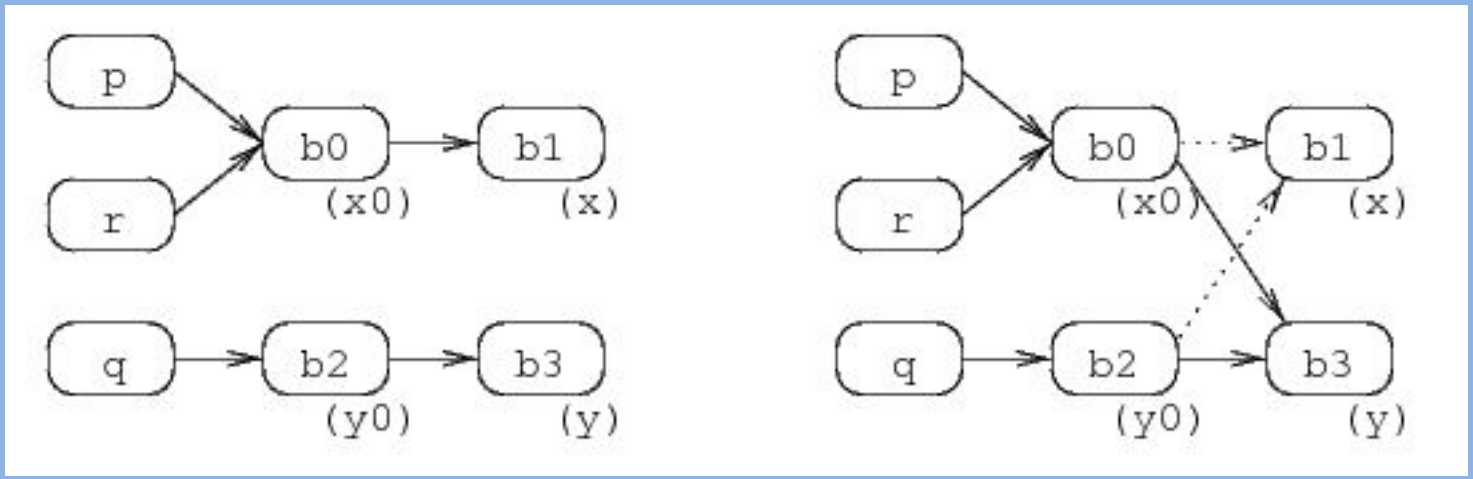
Начальная и конечная points-to функция для f в точке вызова S1



*В начале процедуры перезаписываем значение *p, потому на что указыает b0 нас не интересует*

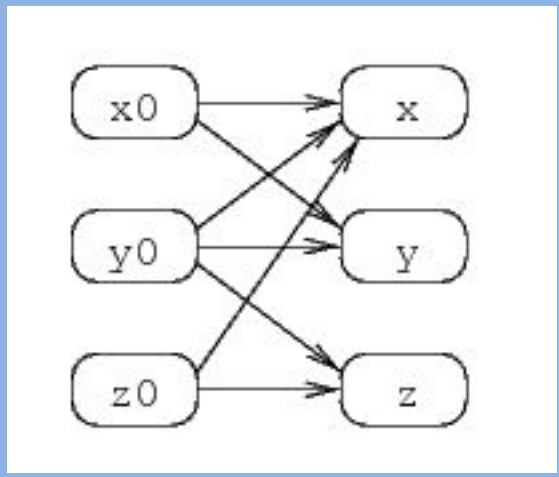
Анализ на основе ЧТФ

Начальная и конечная points-to функция для f в точках вызова S3



*пунктиром анализ определяет, что при записи в *p, записали в *r*

Конечная points-to функция для main



Анализ на основе ЧТФ

- Блоки создаем только когда встречаем реальное разыменование указателя, т.е. записываем информацию не в виде “p есть указатель на b0”, а “p с начальным значением”

```
int *f ( int *p)
{
   int *q = p; /* нет
разыменования *p */
   return q;
}
```

```
void x() /* 3. обновляем PTF */
{
 int *p1 = &q;
 y(&p1);
}
void y( int **p) /* 2. обновляем PTF */
{
 z(p);
}
void z( int **p)
{
  r = *p; /* 1. Встретили разыменование.
Надо найти нелок. блок для *p */
}
```

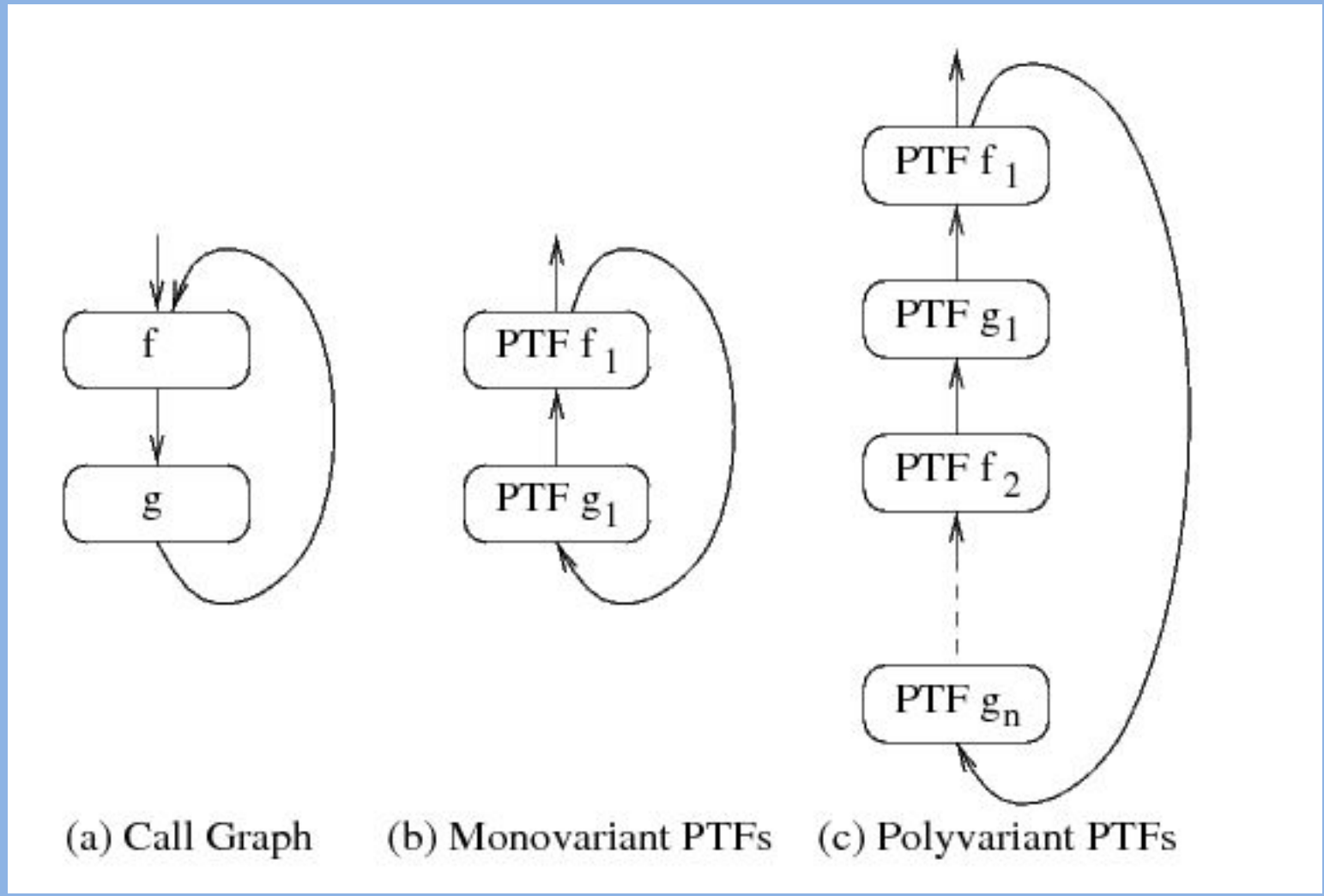
Анализ на основе ЧТФ

- Неявные вызовы – включаем в область определения ЧТФ таблицу с указателями на функции, которые потенциально могут быть вызваны. Добавляем только в случае реального вызова.

```
void x() /* 3. обновляем PTF */
{
 y(&proc1);
}
void y( funcPtr p) /* 2. обновляем PTF */
{
 z(p);
}
void z( funcPtr p)
{
  funcPtr q = &proc2;
  p(); /* 1. Встретили вызов – ищем значение p */
  q(); /* Локальное присваивание, не нужно обновлять PTF
вызывающих функций */
}
```

Анализ на основе ЧТФ

- Обработка рекурсии



Общая схема анализов