12. Межпроцедурный анализ

11.1 Введение в межпроцедурный анализ

11.1.1 Вводные замечания

- Неэффективность, связанная с вызовом процедуры может проявляться как:
  - потеря информации при анализе процедуры, из которой происходит вызов, и связанная с этим невозможность выполнить требуемую оптимизацию (консервативность)
  - дополнительные накладные расходы, связанные с вызовом процедуры.
- Межпроцедурный анализ помогает решить первую из указанных проблем.

11.1 Введение в межпроцедурный анализ

11.1.2 Контекстно-нечувствительный межпроцедурный анализ

- Простейший, но очень неточный подход к межпроцедурному анализу, известный как *контекстно-нечувствительный анализ*, заключается в **рассмотрении каждой команды вызова и каждой команды возврата как операции безусловного перехода**.
- Создается надграф ГПУ, в котором, помимо обычных ребер внутрипроцедурных потоков управления, имеются дополнительные ребра, соединяющие:
  - каждую точку вызова с началом вызываемой в ней процедуры;
  - каждую команду возврата с точкой, следующей за точкой вызова.
- Добавляются команды присваивания каждого фактического параметра соответствующему формальному параметру, а также присваивания возвращаемого значения переменной, получающей результат.
- К построенному надграфу ГПУ, который называется межпроцедурным графом потока управления (МГПУ), применяется стандартный глобальный анализ.

11.1 Введение в межпроцедурный анализ

11.1.2 Контекстно-нечувствительный межпроцедурный анализ

- ◊ Простейший, но очень неточный подход к межпроцедурному анализу, известный как *контекстно-нечувствительный анализ*, заключается в **рассмотрении каждой команды вызова и каждой команды возврата как операции безусловного перехода**.
- ◊ Создается надграф ГПУ, в котором, помимо обычных ребер внутри-процедурных потоков управления, имеются дополнительные ребра,

Построение МГПУ похоже на слияние всех процедур программы в одну большую процедуру (похоже на открытую вставку, но вставки не производится, просто вставляются команды перехода – дуги МГПУ)

- ◊ Добавляются команды присваивания каждого фактического параметра соответствующему формальному параметру, а также присваивания возвращаемого значения переменной, получающей результат.
- ◊ К построенному надграфу ГПУ, который называется межпроцедурным графом потока управления (МГПУ), применяется стандартный глобальный анализ.

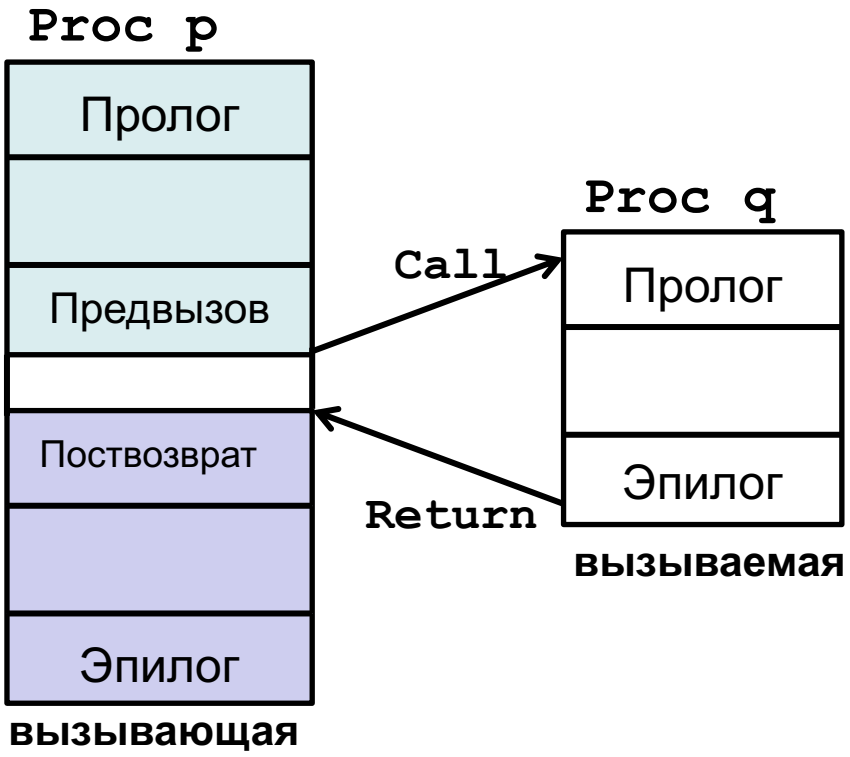
11.1 Введение в межпроцедурный анализ

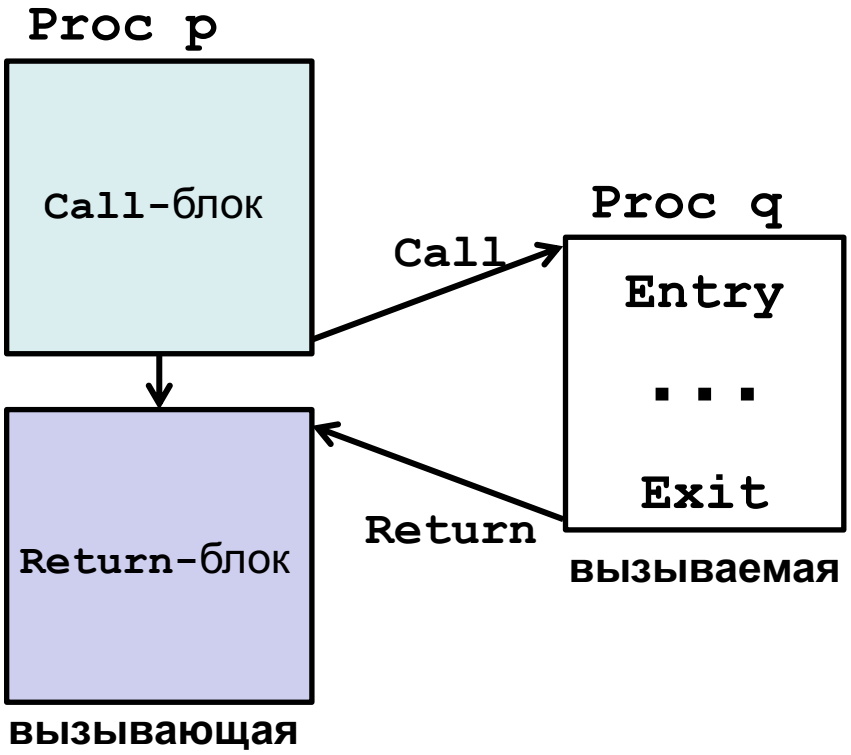
11.1.2 Контекстно-нечувствительный межпроцедурный анализ

- Для **контекстно-нечувствительного** межпроцедурного анализа МГПУ строится следующим образом:
  - каждый предвызов вызывающей (*caller*) удаляется, а соответствующий вызов выделяется в отдельный базовый блок (*call*-блок)
  - каждый поствозврат вызывающей удаляется и вместо него вставляется дополнительный базовый блок (*return*-блок)
  - каждый *call*-блок соединяется дугой с блоком *Entry* вызываемой
  - блок *Exit* вызываемой (*callee*) соединяется дугой с соответствующим *return*-блоком вызывающей
  - пролог и эпилог вызываемой удаляются
  - если вызывающая содержит несколько вызовов одной и той же вызываемой, все ее *call*-блоки и *return*-блоки соединяются с одним и тем же экземпляром вызываемой

11.1 Введение в межпроцедурный анализ

11.1.2 Контекстно-нечувствительный межпроцедурный анализ



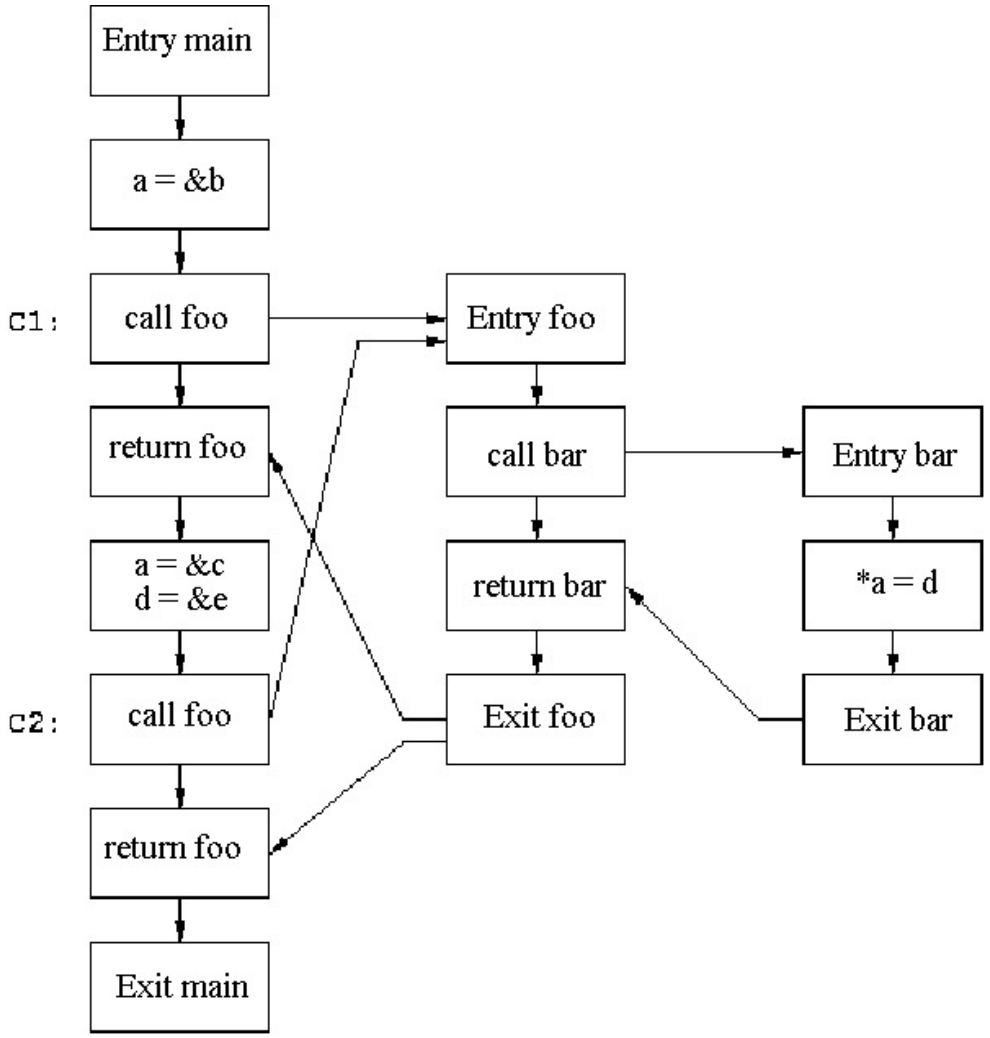


11.1 Введение в межпроцедурный анализ

11.1.2 Контекстно-нечувствительный межпроцедурный анализ

◊ На рисунке представлен МГПУ программы:

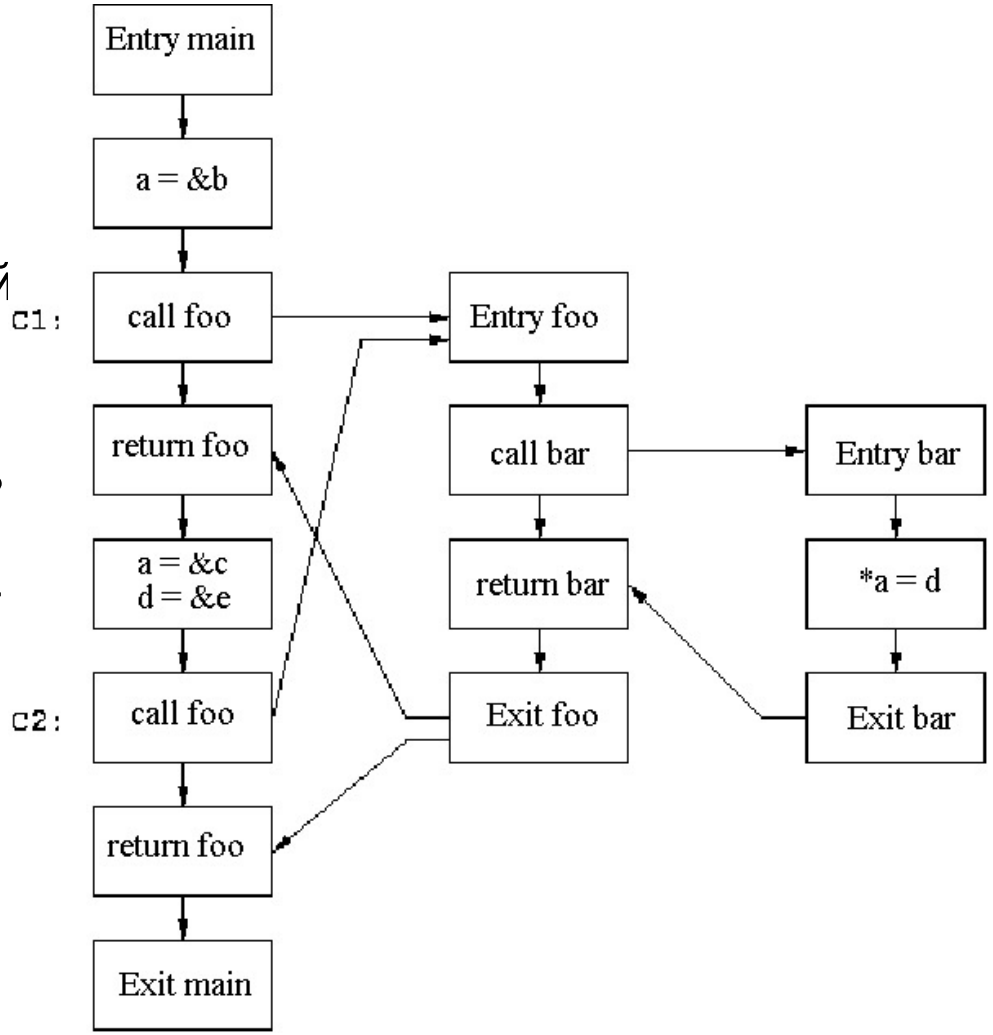
```
int **a, *b, *c, *d, e
foo( ) {
   bar( );
}
bar ( ) {
   *a = d;
}
main( ) {
   a = &b;
   foo( );
   a = &c;
   d = &e;
   foo( );
}
```



11.1 Введение в межпроцедурный анализ

11.1.2 Контекстно-нечувствительный межпроцедурный анализ

- Основной недостаток – нечувствительность к контексту: оба вызова передают управление на один и тот же экземпляр вызываемой функции
- К сожалению, этот недостаток неустраним: попытка увеличить количество экземпляров вызываемых функций приводит к экспоненциальному росту накладных расходов по памяти, а затем и по времени выполнения
- Отметим также, что в рассматриваемом примере процедуры не имеют параметров



11.1 Введение в межпроцедурный анализ

11.1.3 Построение графа вызовов

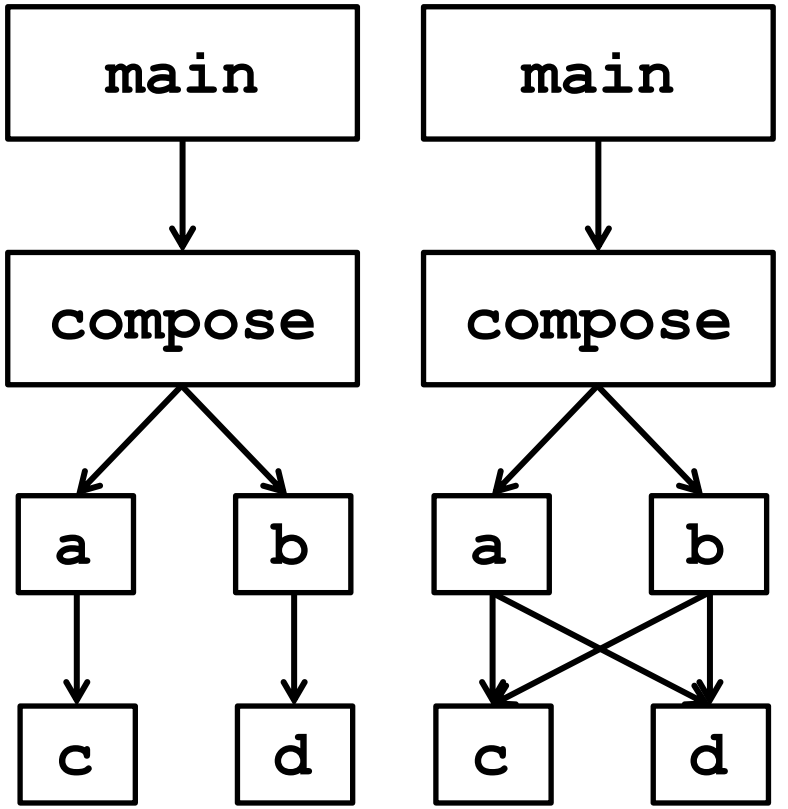
◊ В простейшем случае, когда каждая процедура вызывается по имени, задача построения графа вызовов (ГВ) проста. Компилятор создаёт вершину ГВ для каждой процедуры программы и добавляет дугу ГВ для каждого места вызова. Этот процесс занимает время, пропорциональное числу процедур и числу мест вызова.

11.1 Введение в межпроцедурный анализ

11.1.4 Пример

◊

```
int compose( int f(), int g()) {
  return f(g);
}
int a( int z() ) {
  return z();
}
int b( int z() ) {
  return z();
}
int c( ) {
  return ...;
}
int d( ) {
  return ...;
}
int main(int argc, char *argv[]) {
  return compose(a,c) + compose(b,d);
}
```



Точный ГВ

Приближенный ГВ

Трудно понять, какими могут быть параметры **compose**

11.1 Введение в межпроцедурный анализ

11.1.5 Пример

◊ Рассмотрим программу на языке *C*.

```
int (*pf)(int);
int fun1(int x) {
      if (x < 10)
c1:       return (*pf)(x+1);
      else
          return x;
}
int fun2(int y) {
      pf = &fun1;
c2:       return (*pf)(y);
}
void main() {
      pf = &fun2;
c3:       (*pf)(5);
}
```

Метки *c1*, *c2* и *c3* помечают точки вызова и не являются частью программы.

В программе имеются две функции типа **int(*pf)(int)** – **fun1** и **fun2** (тип функции **main** не соответствует типу указателя **pf)**.

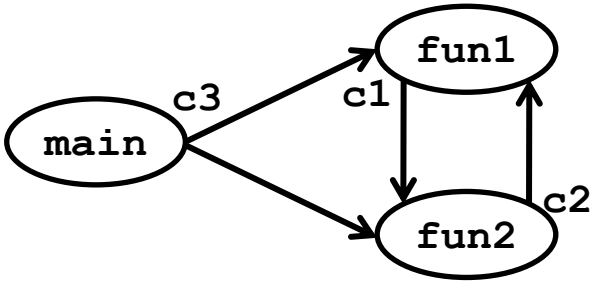
11.2. Межпроцедурный анализ

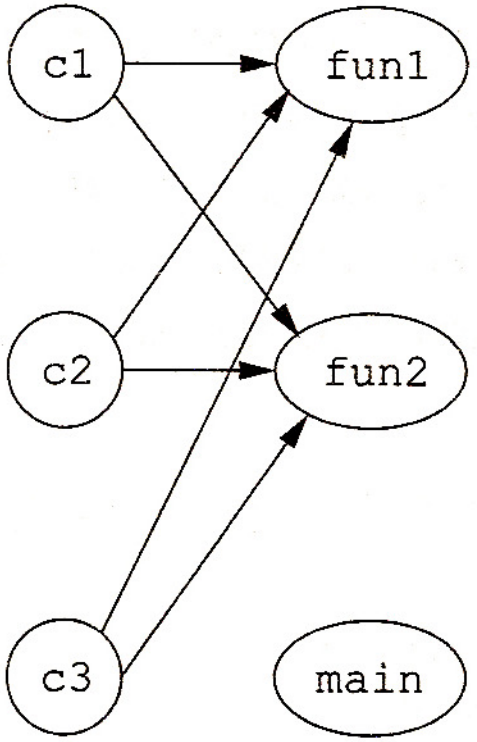
11.2.1 Граф вызовов. Пример

◊ **Простейший анализ** того, на что может указывать **`pf`**, состоит в исследовании типов функций:

- ◊ функции **`fun1`** и **`fun2`** имеют тот же тип, что и указатель **`pf`**,
- ◊ функция **`main`** имеет другой тип.

В результате получится *консервативный граф вызовов*
(граф вызовов, построенный с помощью консервативного анализа)





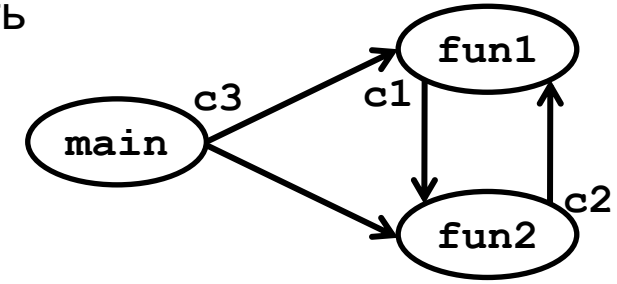
11.2. Межпроцедурный анализ

11.2.1 Граф вызовов. Пример

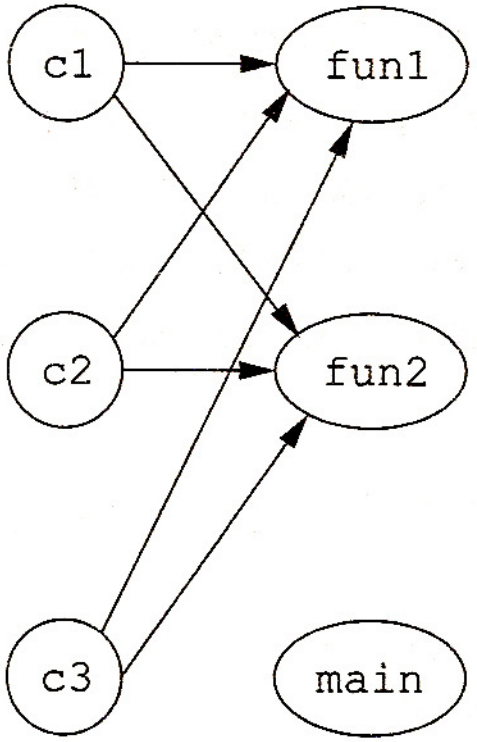
◊ **Простейший анализ** того, на что может указывать **pf**, состоит в исследовании типов функций:

- ◊ функции **fun1** и **fun2** имеют тот же тип, что и указатель **pf**,
- ◊ функция **main** имеет другой тип.

В результате получится *консервативный граф вызовов*




```
int (*pf)(int);
int fun1(int x) {
        if (x < 10)
c1:       return (*pf)(x+1);
        else
          return x;
}
int fun2(int y) {
        pf = &fun1;
c2:     return (*pf)(y);
}
void main() {
        pf = &fun2;
c3:         (*pf)(5);
}
```

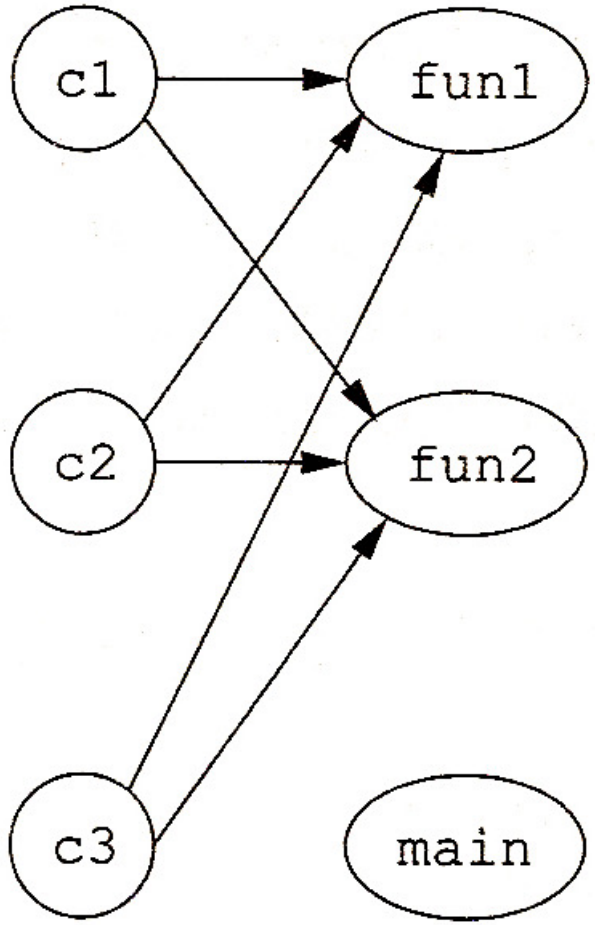


11.2. Межпроцедурный анализ

11.2.1 Граф вызовов. Пример

- **Более точный анализ** позволяет обнаружить:
  - в функции **`main`** указателю **`pf`** присваивается значение **`&fun2`**;
  - в функции **`fun2`** указателю **`pf`** присваивается значение **`&fun1`**;
  - никаких иных присваиваний указателю **`pf`** в программе нет (следовательно, указатель **`pf`** не может указывать на функцию **`main`**).

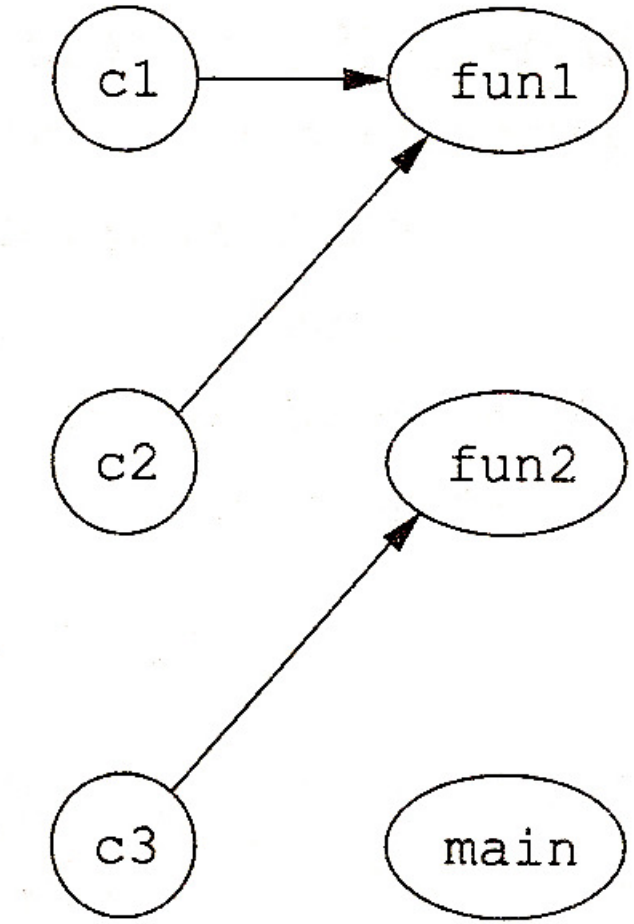
В результате получится тот же самый граф вызовов.



11.2. Межпроцедурный анализ

11.2.1 Граф вызовов. Пример

- **Еще более точный анализ** позволяет установить:
  - **`pf`** ссылается на **`fun2`** только в точке *c3* (этому вызову непосредственно предшествует присваивание **`pf = &fun2`**)
  - в точке *c2* **`pf`** ссылается только на **`fun1`**
  - в функцию **`fun1`** можно попасть только из функции **`fun2`**,
  - в функции **`fun1`** значение **`pf`** не изменяется, следовательно внутри функции **`fun1`** **`pf`** всегда ссылается на **`fun1`** (в частности, в точке *c1* **`pf`** ссылается на **`fun1`**).



11.2. Межпроцедурный анализ

11.2.2 Чувствительность к контексту. Пример

◊ Поведение каждой процедуры, вообще говоря, зависит от контекста, в котором она вызвана.

◊ **Пример**. Рассмотрим фрагмент программы:

```
        for (i = 0; i < n; i++) {
c1:             y1 = 0;
                t1 = f(y1);
c2:             y2 = 243; y1 += y2
                t2 = f(y2);
c3:             t3 = f(y1);
                X[i] = t1 + t2 + t3;
        }

        int f(int v) {
                return (v+1)
        }
```

11.2. Межпроцедурный анализ

11.2.2 Чувствительность к контексту. Пример

◊ Функция **`f`** вызывается в трех точках: *c1*, *c2* и *c3*.

- ◈ вызов в точке *c1*: параметр имеет значение **`0`**, возвращаемое значение равно **`1`**
- ◈ вызовы в точках *c2* и *c3*: значение параметра **`243`**, возвращаемое значение равно **`244`**

Таким образом, значение, возвращаемое функцией **`f`**, зависит от контекста.

◊ Установить, что переменным **`t1`**, **`t2`** и **`t3`** (а значит, и **`X[i]`**) присваиваются константные значения, можно только выяснив, что при вызове в контексте точки *c1* **`f`** возвращает значение **`1`**, а при вызове в контекстах точек *c2* и *c3* – значение **`244`**.

◊ После простейшего анализа можно заключить, что в любом вызове в данной программе **`f`** может возвращать либо **`1`**, либо **`244`**.

11.2. Межпроцедурный анализ

11.2.3 Строка вызовов

◊ Контекст вызова определяется содержимым всего стека вызовов. Последовательность точек вызова в стеке вызовов называется *строкой вызовов.*

◊ **Пример**. Рассмотрим фрагмент кода, полученный модификацией примера 11.2.2: вызовы функции **`f`** заменены вызовами функции **`g`**, которая вызывает **`f`** с тем же аргументом: появляется еще одна точка вызова **`c4`**.

◊ Имеется три строки вызовов функции **`f`**: (*c1, c4*), (*c2, c4*) и (*c3, c4*). При этом в каждой строке вызовов значение **`v`**, передаваемое функции **`f`** зависит не от последней точки вызова **`c4`**, а определяется первым элементом строки вызовов.

◊ Таким образом, важная для анализа информация может определяться ранними элементами строки вызовов: **для получения точного ответа необходимо рассматривать всю строку вызовов**.

11.2. Межпроцедурный анализ

11.2.3 Строка вызовов

```
            for(i = 0; i < n; i++) {
c1:             t1 = g(0);
c2:             t2 = g(243);
c3:             t3 = g(243);
                X[i]= t1 + t2 + t3;
            }

            int g(int v) {
c4:             return f(v);
            }

            int f(int v) {
                return (v+1);
            }
```

11.2. Межпроцедурный анализ

11.2.4 Рекурсивные вызовы

◊ **Пример**. Заменим в предыдущем примере функцию **g**:

пусть для **v > 1 g** рекурсивно вызывается **v** раз,

а для **v ≤ 1 g** такое же, как и раньше.

```
      int g (int v) {
c4:       if (v > 1) {return g(v-1);}
c5:       else {return f(v);}
      }
```

11.2. Межпроцедурный анализ

11.2.4 Рекурсивные вызовы

```
        for (i = 0; i < n; i++) {
c1:          t1 = g(0);
c2:          t2 = g(243);
c3:          t3 = g(243);
             X[i] = t1+t2+t3;
        }
        int g (int v) {
c4:          if (v > 1) {return g(v-1);}
c5:          else {return f(v);}
        }
        int f (int v) {
             return (v+1);
        }
```

11.2. Межпроцедурный анализ

11.2.4 Рекурсивные вызовы

◊ Для функции **`f`** имеются три возможные строки вызовов.

1. После вызова в точке $c1$ функция **`g`** немедленно вызывает функцию **`f`** с параметром **`0`**;
   соответствующая строка вызовов $(c1, c5)$.
2. После вызова в точке $c2$ функция **`g`** вызывается **`243`** раза, после чего вызывается функция **`f`** с параметром **`1`**;
   соответствующая строка вызовов $(c2, c4, ..., c4, c5)$
3. После вызова в точке $c3$ функция **`g`** вызывается **`243`** раза, после чего вызывается функция **`f`** с параметром **`1`**;
   соответствующая строка вызовов $(c3, c4, ..., c4, c5)$

11.2. Межпроцедурный анализ

11.2.5 k-ограниченный контекстно-чувствительный анализ (КЧА)

- В КЧА можно рассматривать не полные строки вызовов, а ограничиться только k последними точками вызовов:
  k-ограниченный КЧА.
- Контекстно-нечувствительный анализ – это 0-ограниченный КЧА.
- В примере 11.2.2 все константы могут быть найдены с помощью 1-ограниченного КЧА,
  в примере 11.2.3 – с помощью 2-ограниченного КЧА.
  в примере 11.2.4 (рекурсивный вызов) никакой k-ограниченный КЧА не в состоянии найти все константы.
- Заметим, что
  при использовании КЧА даже для программ без рекурсии количество разных контекстов вызова может экспоненциально зависеть от количества процедур в программе.

11.2. Межпроцедурный анализ

11.2.6 Контекстно-чувствительный анализ на основе клонирования

- ◊ Сначала контекстно-зависимые процедуры клонируются по одной для каждого уникального контекста.
  Затем к клонированному графу вызовов применяется контекстно-нечувствительный анализ.
- ◊ Отметим, что в реальности клонировать (размножать) код не требуется – можно просто использовать эффективное внутреннее представление для отслеживания результатов анализа каждого клона.
- ◊ Применим клонирование к рассмотренным примерам (11.2.3 и 11.2.4).
- ◊ **Пример**. Клонированная версия кода примера 11.2.3 показана на следующем слайде
  Поскольку каждый контекст вызова работает со своим клоном, никакой путаницы не возникает:
  клон `g1` получает в качестве аргумента `0` и возвращает `1`;
  клоны `g2` и `g3` получают в качестве аргументов `243` и возвращают `244`.

11.2. Межпроцедурный анализ

Исходная программа

```
    for(i = 0; i < n; i++) {
c1:     t1 =g(0);
c2:     t2 =g(243);
c3:     t3 =g(243);
        X[i]= t1 + t2 + t3;
    }
        int g(int v) {
c4:     return f(v);
    }




   int f (int v) {
        return (v+1);
   }
```

Клонированная программа

```
      for (i = 0; i < n;
c1:    t1 = g1(0);
c2:    t2 = g2(243);
c3:    t3 = g3(243);
        X[i] = t1 + t2 + t3;
       }
       int g1 (int v) {
c4.1:     return f1(v);
       }
       int g2 (int v) {
c4.2:     return f2(v);
       }
       int g3 (int v) {
c4.3:      return f3(v);
       }
       int f1 (int v) {
            return (v+1);
       }
       int f2 (int v) {
            return (v+1);
       }
       int f3 (int v) {
            return (v+1);
       }
```

11.2. Межпроцедурный анализ

11.2.6 Контекстно-чувствительный анализ на основе клонирования

◊ **Пример**. Клонированная версия кода примера 11.2.4 показана на следующем слайде

Для процедуры **`g`** создается клон для представления всех экземпляров **`g`**, впервые вызываемых из точек вызова **`c1`**, **`c2`** и **`c3`**.

В этом случае анализ может определить, что вызов в точке **`c1`** возвращает **`1`**

(в предположении, что этот анализ в состоянии вывести, что при **`v = 0`** результат проверки **`v > 1`** отрицательный).

Однако этот анализ недостаточно хорош, чтобы вычислить константы для точек вызова **`c2`** и **`c3`**.

11.2. Межпроцедурный анализ

Исходная программа

```
    for (i = 0; i < n; i++)
{
c1:     t1 = g(0);
c2:     t2 = g(243);
c3:     t3 = g(243);
        X[i] = t1 + t2 + t3;
          }
          int g (int v) {
        if (v > 1) {
c4:        return g(v-1);
        } else {
c5:        return f(v);
        }
          }
     int f (int v) {
          return (v+1);
  }
```

Клонированная программа

```
      for (i =0; i < n; i++) {
c1:       t1 = g1(0);
c2:       t2 = g2(243);
c3:       t3 = g3(243);
        X[i] = t1+t2+t3; }
     int g1 (int v) {
        if (v > 1) {
c4.1:       return g1(v-1);
        } else {
c5.1:       return f1(v); } }
         int g2 (int v) {
        if (v > 1) {
c4.2:       return g2(v-1);
         } else {
c5.2:       return f2(v); } }
         int g3 (int v) {
         if (v > 1) {
c4.3:       return g3(v-1);
         } else {
c5.3:       return f3(v); } }
         int f1 (int v) {return (v+1); }
         int f2 (int v) {return (v+1); }
         int f3 (int v) {return (v+1); }
```

11.2. Межпроцедурный анализ

11.2.6 Контекстно-чувствительный анализ на основе аннотаций

◊ Межпроцедурный анализ на основе аннотаций представляет собой обобщение анализа на основе областей.
По сути, в анализе на основе аннотаций каждая область представлена кратким описанием («*аннотацией*»), которое концентрирует некоторое наблюдаемое поведение области.
В рассматриваемом случае роль областей играют процедуры.
Основная цель аннотации – избежать повторного анализа тела процедуры в каждой точке, где эта процедура может быть вызвана.
Это позволяет сделать учет особенностей контекста вызова не слишком дорогим.

◊ Каждая процедура моделируется как область с единой точкой входа, при этом все пары «вызывающая/вызываемая процедуры» описываются отношением «внешняя/внутренняя области».
Единственное отличие от внутрипроцедурной версии заключается в том, что в межпроцедурном случае область процедуры может быть вложена внутрь нескольких различных внешних областей.
(В этом случае может помочь клонирование: в каждую область будет вложен свой экземпляр процедуры).

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

- Анализ (как и в случае анализа на основе областей) состоит из двух фаз:
  - *восходящая фаза*, на которой вычисляются передаточные функции для аннотирования действий процедуры («*распространение аннотаций*»);
  - *нисходящая фаза*, на которой информация вызывающей процедуры передается вызываемым процедурам для вычисления результатов их выполнения. .
- Для полного учета контекстной чувствительности информация от разных контекстов вызова должна распространяться вниз к отдельным вызываемым функциям.
- Для повышения эффективности (с понижением точности) можно сначала объединять информацию от всех вызывающих функций с использованием оператора сбора, а затем передавать ее вниз вызываемым функциям.

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

◊ **Пример 1.** Для распространения констант каждая процедура аннотируется передаточной функцией, определяющей, как константы распространяются через тело этой процедуры.

◊ Во фрагменте программы

```
for (i = 0; i < n; i++) {
   c1:   t1 = f(0);
   c2:   t2 = f(243);
   c3:   t3 = f(243);
         X[i] = t1 + t2 + t3;
}

int f(int v) {return (v+1)}
```

для функции **f** можно составить аннотацию:

"v == c ⇒ f(v)== c + 1"

(если фактическим параметром функции **f** является константа **c**, то возвращаемым значением будет константа **c + 1**)

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

◊ **Пример 1.** Для распространения констант каждая процедура аннотируется передаточной функцией, определяющей, как константы распространяются через тело этой процедуры.

◊ Во фрагменте программы

```
for (i = 0; i < n; i++) {
   c1:   t1 = f(0);
   c2:   t2 = f(243);
   c3:   t3 = f(243);
         X[i] = t1 + t2 + t3;
}

int f(int v) {return (v+1)}
```

для функции **f** можно составить аннотацию:

$$\text{"v == c} \Rightarrow \text{f(v)== c + 1"}$$

(если фактическим параметром функции **f** является константа **c**

На основе этой аннотации анализ может определить, что переменные **t1**, **t2** и **t3** получат значения **1**, **244** и **244** соответственно.

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

◊ **Пример 2.** Рассмотрим фрагмент программы:

```
          for(i = 0; i < n; i++) {
c1:                    t1 =g(0);
c2:                    t2 =g(243);
c3:                    t3 =g(243);
                 X[i]= t1 + t2 + t3;
          }

          int g(int v) {
c4:              return f(v);
          }

          int f (int v) {
                 return (v+1);
          }
```

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

◊ **Пример 2.** Рассмотрим фрагмент программы:

```
   for(i = 0; i < n; i++) {
c1:      t1 =g(0);
c2:      t2 =g(243);
c3:      t3 =g(243);
         X[i]= t1 + t2 + t3;
         }

         int g(int v) {
c4:            return f(v);
         }

         int f (int v) {
               return (v+1);
         }
```

Этот фрагмент отличается от фрагмента из предыдущего примера тем, что в нем добавлена функция **g**, которая вызывает функцию **f**.
Следовательно, **передаточная функция для g такая же, как и передаточная функция для f**, что и будет указано в аннотации.
Прочитав во время анализа эту аннотацию, можно будет сделать вывод: переменные **t1**, **t2** и **t3** получают значения **1**, **244** и **244** соответственно.

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

◊ **Пример 1. (продолжение)**

Анализ позволяет понять, что **`v == NAC`**

(в точке **`c1 v == 0`**, в точке **`c2 v == 243`**)

Можно создать два специализированных клона функции **`f`** для входных значений **`0`** и **`243`** соответственно

```
                for (i =0; i < n; i++) {
c1:                   t1 = f0(0);
c2:                   t2 = f243(243);
c3:                   t3 = f243(243);
                      X[i] = t1+t2+t3;
                }
                int f0 (int v) {
                      return (1);
                }
                int f243 (int v) {
                      return (244);
                }
```

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

◊ **Пример 1. (продолжение)**

Анализ позволяет понять, что **`v == NAC`**

(в точке **`c1 v == 0`**, в точке **`c2 v == 243`**)

Можно создать два специализированных клона функции **`f`** для входных значений **`0`** и **`243`** соответственно

Отличие от КЧА на основе клонирования:
в случае КЧА на основе клонирования клонирование выполняется до анализа, на основе строк вызовов;
в случае КЧА на основе аннотаций клонирование выполняется после анализа, на основе его результатов.

```
        for (i =0; i < n; i++) {
c1:       t1 = f0(0);
c2:       t2 = f243(243);
c3:       t3 = f243(243);
          X[i] = t1+t2+t3;
        }
        int f0 (int v) {
          return (1);
        }
        int f243 (int v) {
          return (244);
        }
```

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

◊ **Пример 1. (продолжение)**

Анализ позволяет понять, что **`v == NAC`**

(в точке **`c1 v == 0`**, в точке **`c2 v == 243`**)

Можно создать два специализированных клона функции **`f`** для входных значений **`0`** и **`243`** соответственно

Вывод: Только контекстно-чувствительный анализ дает значимые результаты, но он требует большого времени выполнения и существенных расходов памяти, так как предполагает новый анализ функции при каждом ее вызове. Это связано с зависимостью фактических параметров вызова от контекста в точке вызова. Методы, рассмотренные выше позволяют сократить количество анализов функции, учитывая особенности каждого её вызова.

```
        for (i =0; i < n; i++) {
c1:       t1 = f0(0);
c2:       t2 = f243(243);
c3:       t3 = f243(243);
          X[i] = t1+t2+t3;
        }
        int f0 (int v) {
          return (1);
        }
        int f243 (int v) {
          return (244);
        }
```

11.2. Межпроцедурный анализ

11.2.7 Контекстно-чувствительный анализ на основе аннотаций

- ◊ Отличие от предыдущего варианта КЧА заключается в том, что в случае КЧА на основе клонирования клонирование выполняется до анализа, на основе строк вызовов,
  а в случае КЧА на основе аннотаций клонирование выполняется после анализа, на основе его результатов.
- ◊ Даже если клонирование не применяется, выводы КЧА на основе аннотаций о результатах выполнения вызываемой процедуры делаются точно, без проблемы нереализуемых путей.
- ◊ Клонирование функций можно заменить открытой вставкой их кода. Открытая вставка дает дополнительный эффект, устраняя накладные расходы, связанные с вызовами процедур.
- ◊ Справиться с рекурсией можно с помощью итераций.

Вывод: Только контекстно-чувствительный анализ дает значимые результаты, но он требует значительного времени выполнения и существенных расходов памяти, так как предполагает новый анализ функции при каждом ее вызове. Это связано с зависимостью фактических параметров вызова от контекста в точке вызова. Методы, рассмотренные выше позволяют сократить количество анализов функции, учитывая особенности каждого её вызова.

11.3. Межпроцедурное распространение констант

11.3.1. Введение

- Межпроцедурное распространение констант отслеживает **известные константные значения** глобальных переменных и параметров процедур при их распространении по графу вызовов как через тела процедур, так и вдоль ребер графа вызовов.
- Цель межпроцедурного распространения констант – выявить ситуации, когда процедура всегда получает известное константное значение или когда она всегда возвращает такое значение. Когда обнаруживается такое значение (константа), анализ может преобразовать код с учетом обнаруженного значения.
- Концептуально межпроцедурное распространение констант сводится к решению следующих трех подзадач:
  - Выявление исходного множества константных значений
  - Распространение известных константных значений по графу вызовов
  - Моделирование передачи значений через процедуры.

11.3. Межпроцедурное распространение констант

11.3.2. Выявление исходного множества констант

◊ В каждой точке вызова анализатор должен определить, какие параметры и (или) глобальные переменные имеют известные константные значения.
Для решения этой подзадачи существует много методов от выявления фактических параметров, являющихся литеральными константами, до многократного выполнения процедуры глобального распространения констант (что, конечно, делает анализ намного более затратным).

11.3. Межпроцедурное распространение констант

11.3.3. Распространение известных константных значений по графу вызовов

◊ Получив начальное множество констант, анализатор распространяет константные значения вдоль ребер графа вызовов и через каждую процедуру от ее входа (*Entry*) до каждой точки вызова других процедур, имеющейся в рассматриваемой процедуре.

По существу это – усложненный итеративный анализ потока данных, который требует намного больше итераций, чем в случае анализа достигающих определений или живых переменных.

11.3. Межпроцедурное распространение констант

11.3.4. Моделирование передачи значений через процедуры

◊ Во время обработки каждой вершины графа вызовов анализатор должен определять **как константные значения, известные на входе в процедуру, влияют на множество константных значений в каждой точке вызова**.

Это позволяют сделать небольшие модели, называемые *функциями скачка*.

Для каждого фактического параметра строится своя функция скачка, так что точке вызова s с n параметрами соответствует вектор функций скачка $\mathcal{J}_s = \left(\mathcal{J}_s^{p_1}, \mathcal{J}_s^{p_2}, ..., \mathcal{J}_s^{p_n}\right)$, где p_i (i = 1, …, n) – фактические параметры вызова в точке s.

Каждая функция скачка $\mathcal{J}_s^{p_i}$ может зависеть не только от p_i, но и от некоторого подмножества других формальных параметров вызова. Это подмножество будем обозначать $Support\,(\mathcal{J}_s^{p_i})$.

Требуется, чтобы функция $\mathcal{J}_s^{p_i}$ возвращала значение *Undef*, если хотя бы один параметр из $Support\,(\mathcal{J}_s^{p_i})$ имеет значение *Undef*.

11.3. Межпроцедурное распространение констант

11.3.5. Алгоритм

- ◊ Рассмотрим простой алгоритм межпроцедурного распространения констант. Он похож на алгоритм глобального распространения констант, рассмотренный в разделе 7.2.
- ◊ На фазе инициализации все значения фактических параметров полагаются равными *Undef*.
- ◊ После этого выполняются итерации для каждого фактического параметра *a* в каждой точке вызова *s*, в результате чего значения фактических параметров уточняются, а уточненные параметры снова помещаются в *Worklist*.
- ◊ В конце фазы инициализации образуется начальное множество констант, представленных функциями скачка, а в *Worklist* записывается список всех формальных параметров.
- ◊ Псевдокод первой фазы на следующем слайде.

11.3. Межпроцедурное распространение констант

11.3.5. Алгоритм

```
// Фаза 1: Инициализация
// Построение всех функций скачка и поддержка отображений

Worklist ← ∅;
for each procedure p in the program
  for each formal parameter f to p
    Value(f) ← ⊤          // ⊤ = Undef (начальное значение)
    Worklist ← Worklist ∪ {f}
for each call site s in the program
  for each formal parameter f that receives value at s
    Value(f) ← Value(f) ∧ J_s^f  // Учет J_s^f в начальных значениях
```

call site – точка вызова

11.3. Межпроцедурное распространение констант

11.3.5. Алгоритм

◊ **Вторая фаза** многократно выбирает формальные параметры из *Worklist* и распространяет их.

Для того, чтобы распространить формальный параметр f процедуры p анализатор находит каждую точку вызова s процедуры p и каждый формальный параметр x (который соответствует фактическому параметру точки вызова s), такой что $f \in Support(\mathcal{J}_s^x)$. Затем вычисляется $\mathcal{J}_s^x$ и полученное значение сравнивается с текущим значением x.

Если сравниваемые значения не совпадают, значение x заменяется и x добавляется в *Worklist*.

◊ Вторая фаза завершается, так как каждый параметр может принимать всего три значения: ⊤, *константа* и ⊥

◊ Псевдокод второй фазы на следующем слайде.

11.3. Межпроцедурное распространение констант

11.3.5. Алгоритм

```
// Фаза 2: Выполнение итераций
while (Worklist ≠ ∅;)
    pick parameter f from Worklist // выбрать параметр
    let p be the procedure declaring f
   // Обновить значение каждого параметра f функции p
    for each call site s in p and parameter x such that
```

$f \in Support(\mathcal{J}_s^x)$

```
       t ← Value(x)
       Value(x) ← Value(x) ∧ J_s^x        // Новое значение
       if (Value(x) < t)
             then Worklist ← Worklist ∪ {x}
```

11.3. Межпроцедурное распространение констант

11.3.5. Алгоритм

```
// Post-process Val формирует множество CONSTANTS
for each procedure p
CONSTANTS(p) ← ∅;
      for each formal parameter f to p
            if (Value(f) = Undef)
                  then Value(f) ← NAC;
            if (Value(f) ≠ NAC)
                  then CONSTANTS(p) ←
                        CONSTANTS(p) ∪ { (f, Value(f)) }
```

11.3. Межпроцедурное распространение констант

11.3.6. Реализация функций скачка

- Реализации функций скачка варьируются от простых статических приближений, которые не изменяются во время анализа (например, аннотации), либо небольших параметризованных моделей (например, клоны, или строки вызова), до более сложных схем, которые выполняют глубокий всесторонний анализ при каждом вычислении функции скачка.
- Для каждой из таких схем выполняется несколько простых принципов.
  - Если анализатор выясняет, что параметр **x** в точке вызова **s** равен известной константе **c**, то $\mathcal{J}_s^x = c$ и $Support\left(\mathcal{J}_s^x\right) = \varnothing$
  - Если $y \in Support\left(\mathcal{J}_s^x\right)$ и $Value(y) = Undef$, то $\mathcal{J}_s^x = Undef$
  - Если анализатор выясняет, что значение $\mathcal{J}_s^x$ невозможно определить, то $\mathcal{J}_s^x = NAC$

11.3. Межпроцедурное распространение констант

11.3.6. Реализация функций скачка

- Анализатор может реализовать $\mathcal{J}_s^x$ многими способами.
  - Самая простая реализация может распространять константу только тогда, когда x является SSA-именем формального параметра процедуры, вызываемой в точке s.
  - Более сложная реализация может строить выражения из SSA-имен формальных параметров и литеральных констант.
  - Наиболее точная, но дорогая реализация может для обновления значений функций скачка вызывать программу, выполняющую алгоритм глобального распространения констант.

11.3. Межпроцедурное распространение констант

11.3.7. Учет возвращаемых значений

- ◊ Рассмотренный алгоритм межпроцедурного распространения констант учитывает только фактические параметры вызываемых процедур.
- ◊ Его можно расширить очевидным образом, чтобы он учитывал и возвращаемые значения, а также значения глобальных переменных.
- ◊ Для этого необходимо определить *обратные функции скачка*, моделирующие передачу значений от вызываемой функции к вызывающей.
- ◊ Обратные функции скачка важны, когда анализируются функции, используемые для инициализации значений, а не для заполнения полей *Java*-класса