САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

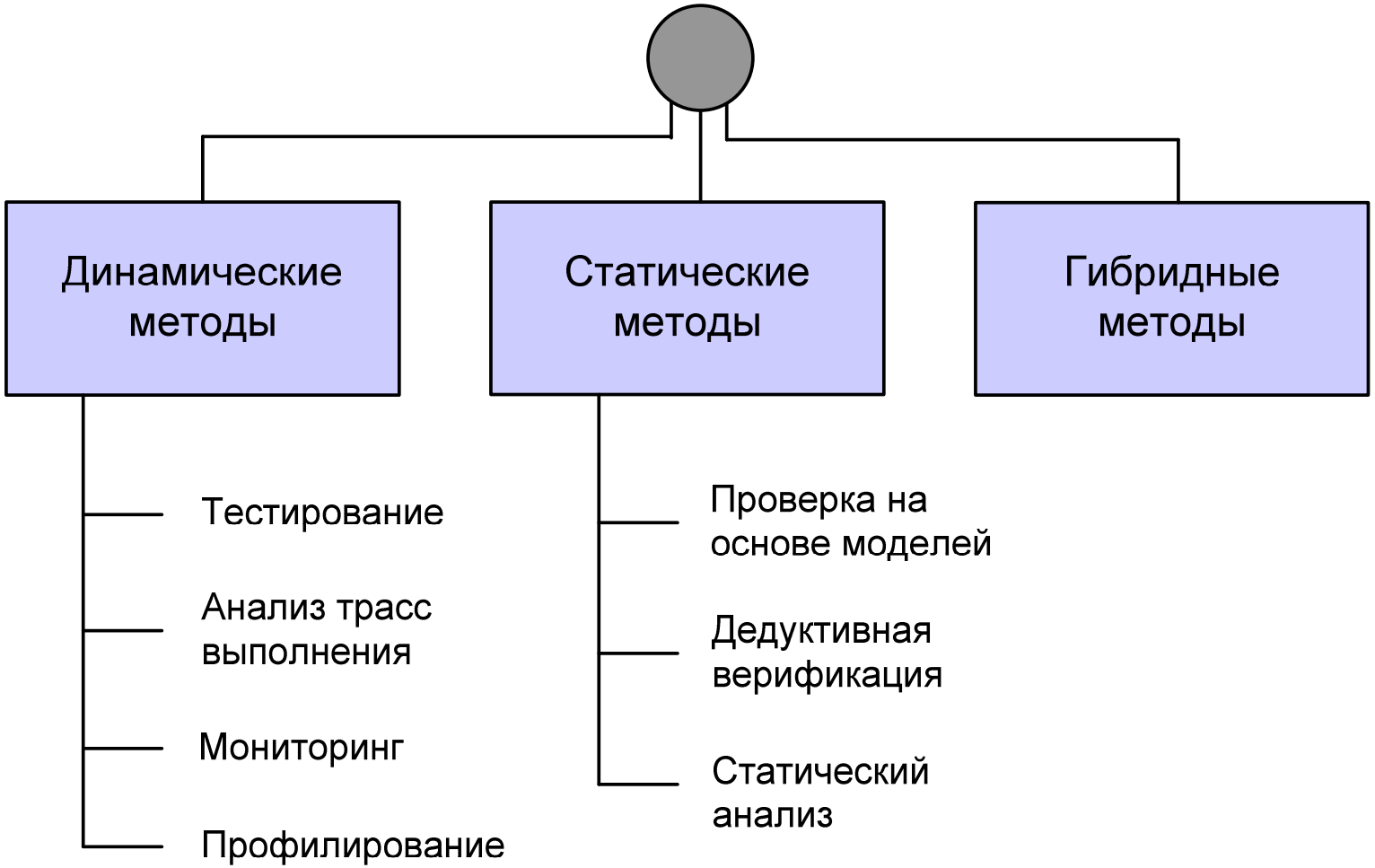
Методы анализа и обеспечения качества ПО

Михаил Моисеев

Введение в статический анализ
Обнаружение ошибок методами статического анализа

Санкт-Петербург
2013

Методы анализа ПО



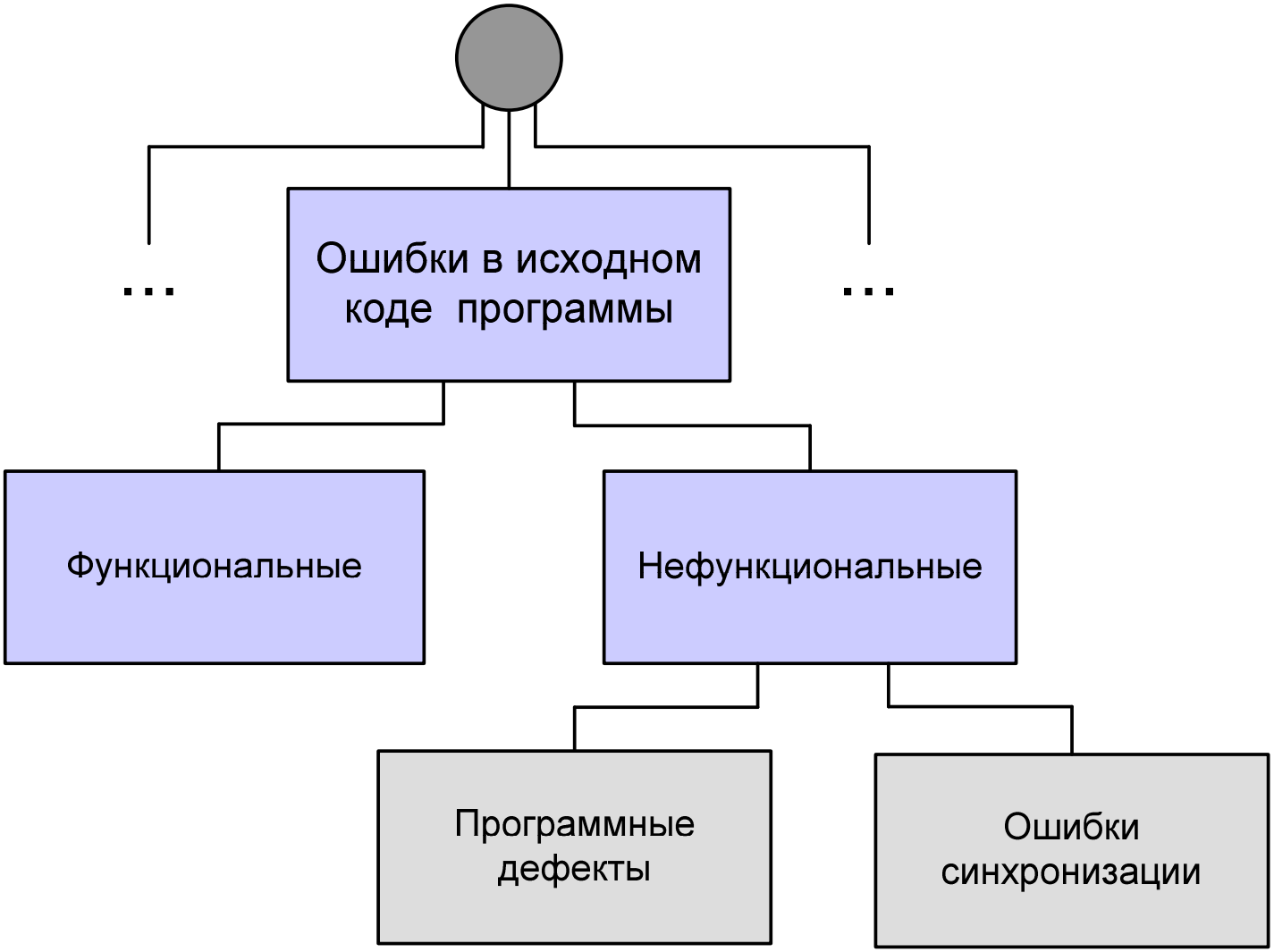
Статический анализ

- **Статический анализ** – группа методов, использующих исходный код программы для извлечения необходимой информации

Задачи статического анализа

- Оптимизация программ
  - вычисление константных выражений
  - обнаружение мертвого кода
  - распараллеливание программы
- Преобразование программ
  - перевод на другие платформы, языки и библиотеки
- Обфускация / деобфускация
- **Обнаружение ошибок**

Программные ошибки



Обнаружение нефункциональных ошибок

- Проверяемые правила определяются стандартами языка и используемых библиотек функций
- Свойства статического анализа
  - обнаружение основных видов нефункциональных ошибок
  - **процесс обнаружения можно полностью автоматизировать**
  - обладает приемлемой ресурсоемкостью
  - обладает достаточно высокой полнотой/точностью
- Статический анализ одна из развитых техник – проверенные на практике правила реализуются в средах разработки и компиляторах
- Обнаружение ошибок с помощью компиляторов

Обнаружение функциональных ошибок

- Проверяется соответствие исходного кода программы спецификациям на некотором формальном языке
- Примерами таких спецификаций являются контракты для функций программы
  - предусловия – требуются для выполнения функции
  - постусловия – должны обеспечиваться после ее выполнения
  - инварианты – должны сохраняться в процессе выполнения функции
- Свойства статического анализа
  - процесс обнаружения кроме создания спецификаций можно полностью автоматизировать
  - обладает приемлемой ресурсоемкостью
  - полнота зависит от количества спецификаций и полноты самих алгоритмов анализа, точность определяется алгоритмами анализа

Виды статического анализа

- Синтаксический анализ (поиск по шаблонам)
- Анализ потока управления
- Анализ потока данных
  - анализ состояний объектов программы
- Анализ параллельного выполнения программы

Анализ программ на основе шаблонов

- Выполняется синтаксический анализ – рассматриваются одна или несколько последовательных конструкций, **без учета контекста программы**
- Поиск в исходном коде конкретных или параметризируемых шаблонов
- Параметризируемый шаблон представляет собой конструкции программы и переменные в качестве параметров

```
gets(x);
```

```
if (x = y) {
   ...;
}
```

```
for (int x; x < C; x++) {
   ...;
}
```

Анализ программ на основе шаблонов

- Шаблоны для обнаружения дефектов
  - дефекты должны представляться 1-2 последовательными конструкциями
  - конструкции представляющие дефект должны быть локализированы
- Достоинства и недостатки
  - простота реализации и низкая ресурсоемкость
  - возможность обнаруживать «тонкие» нарушения стиля кодирования и правил использования библиотечных функций
  - невозможность обнаружения широкого класса дефектов в произвольных ситуациях
- Средство анализа на основе шаблонов Viva64 http://www.viva64.com/

Анализ потока управления

- Control Flow Analysis
- Цель – построение графа потока управления или другого представления программы удобного для проведения анализа потока данных
  - На входе – исходный код программы
  - На выходе – модель программы

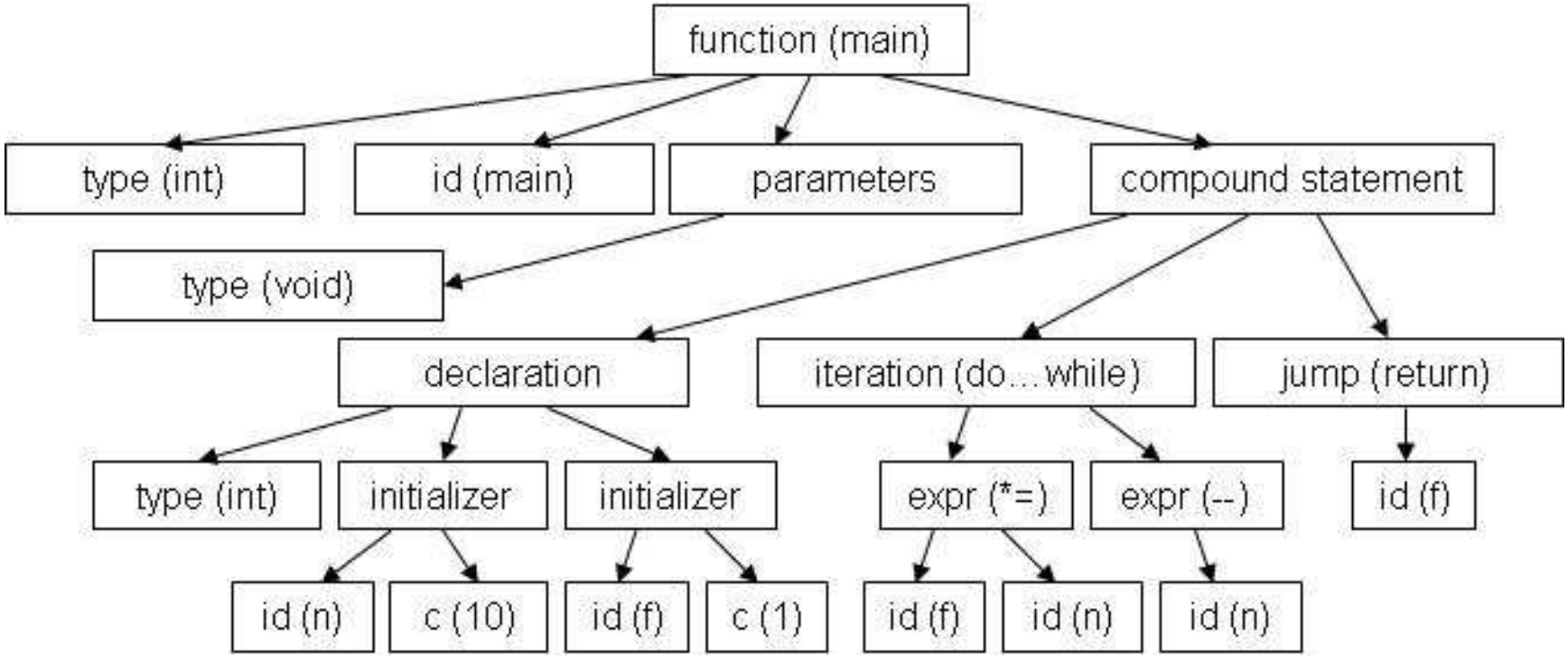
Модели программы

- Модель программы
  - обеспечивать доступ ко всем объектам программного кода и их атрибутам
  - поддерживать эффективный поиск объектов
  - снижать сложность разработки методов статического анализа
- Используемые модели
  - абстрактное синтаксическое дерево (AST)
  - абстрактный семантический граф (ASG)
  - граф потока управления (CFG)
  - граф зависимостей по данным (DDG)
  - представление на основе статического однократного присваивания (SSA)

Абстрактное синтаксическое дерево

- Дерево разбора (Parse tree) – вершины сопоставлены с нетерминалами формальной грамматики, а листья – с терминалами
- **Abstract Syntax Tree (AST)** – упрощенное дерево разбора, в котором нетерминальные узлы с единственным нетерминальным потомком преобразуются в атрибуты узлов
- Могут вводится новые узлы с большей семантической нагрузкой

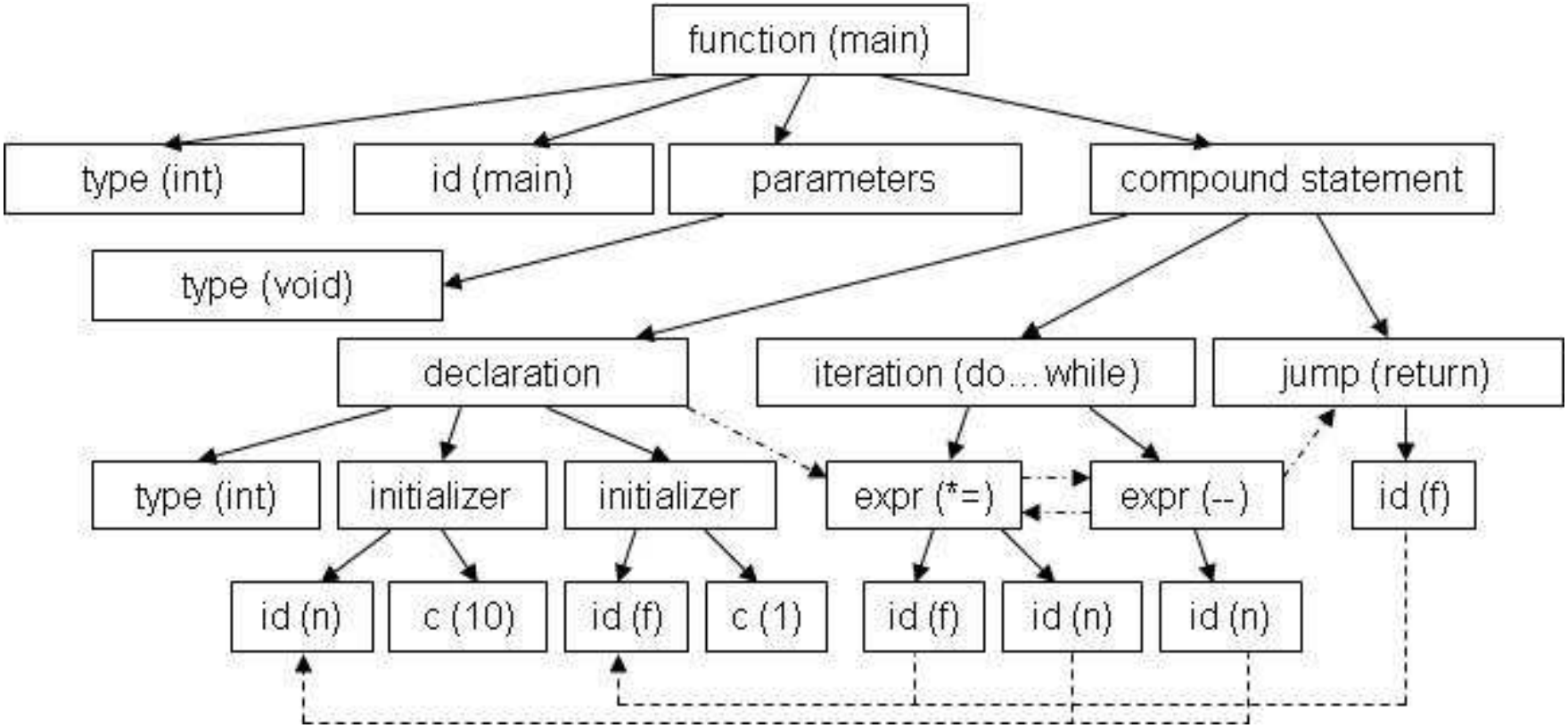
Абстрактное синтаксическое дерево



Абстрактный семантический граф

- **Abstract Semantic Graph (ASG)** – AST дополненное семантической информацией
- Семантическая информация получается в результате проведения предварительного анализа
- Семантическая информация представляется в виде дуг
  - дуги от использования функции к ее определению
  - дуги текущей инструкции к следующей
  - дуги от определения значения переменной к использованию

Абстрактный семантический граф

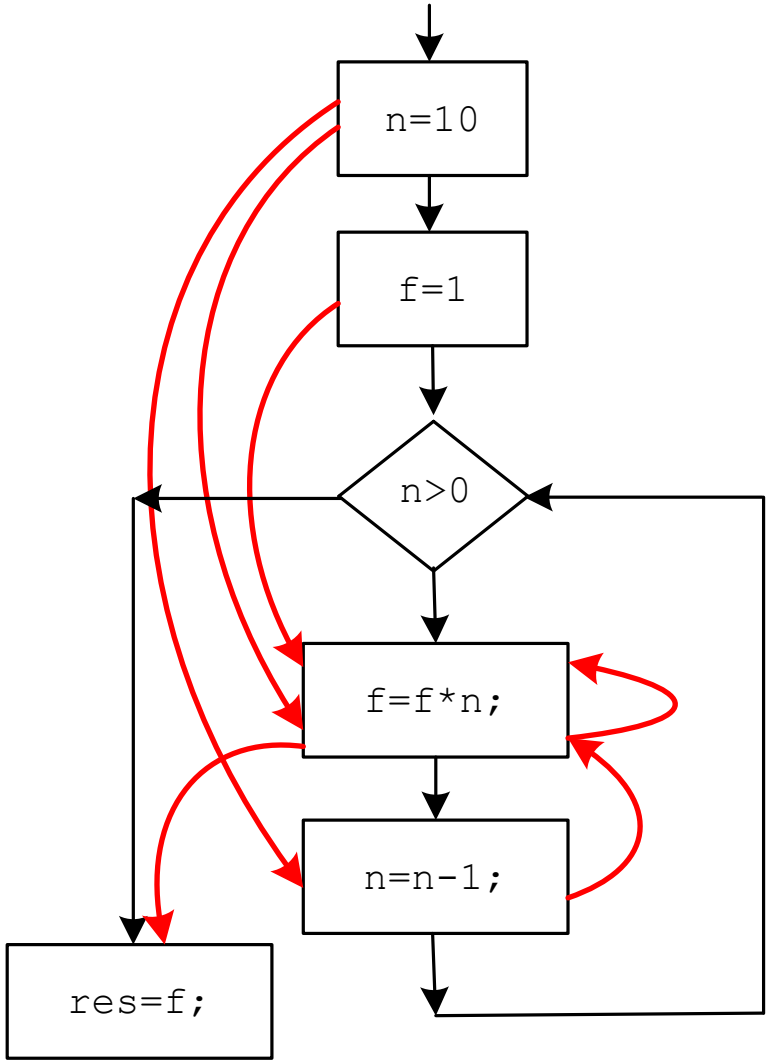


Граф зависимостей по данным

- **Data Dependency Graph (DDG)** – зависимости по данным между узлами-инструкциями представляются в виде направленных дуг
- Дуга связывает два узла тогда и только тогда, когда между ними есть зависимость по данным
  - запись-чтение
  - чтение-запись
  - запись-запись

Пример DDG

```
int n = 10;
int f = 1;
while (n > 0) {
   f = f * n;
   n = n – 1;
}
res = f;
```

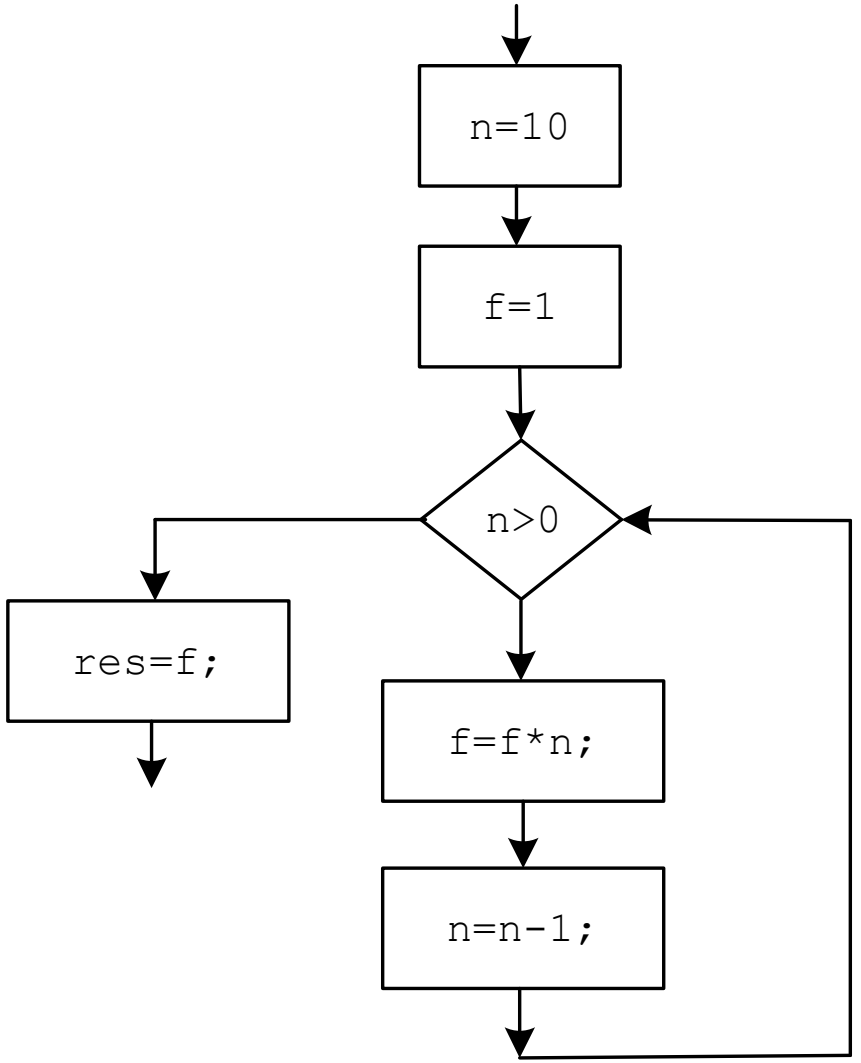


Граф потока управления

- **Control Flow Graph (CFG)** – поток управления представляется в виде ориентированного графа
- Сохраняется информация о конструкциях программы и переходах между ними
- При построении CFG учитываются
  - безусловные переходы
  - ветвления
  - циклы
  - вызовы функций
  - исключения

Пример CFG

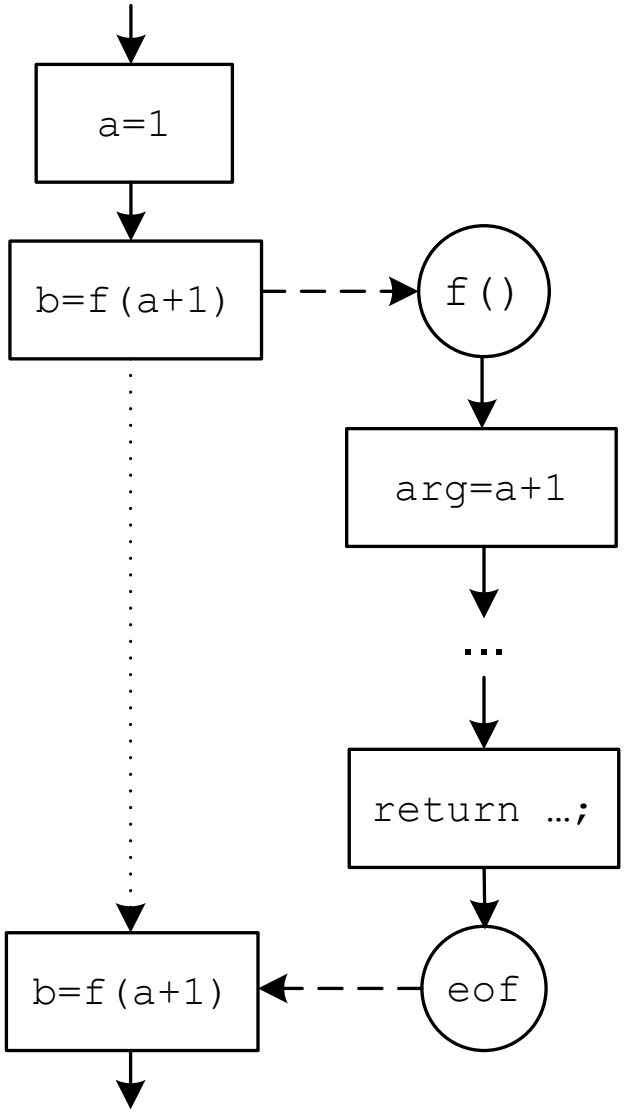
```
int n = 10;
int f = 1;
while (n > 0) {
   f = f * n;
   n = n - 1;
}
res = f;
```



Пример CFG

```
int a = 1;
int b = f(a+1);


float c = g(f(b)*a);
```

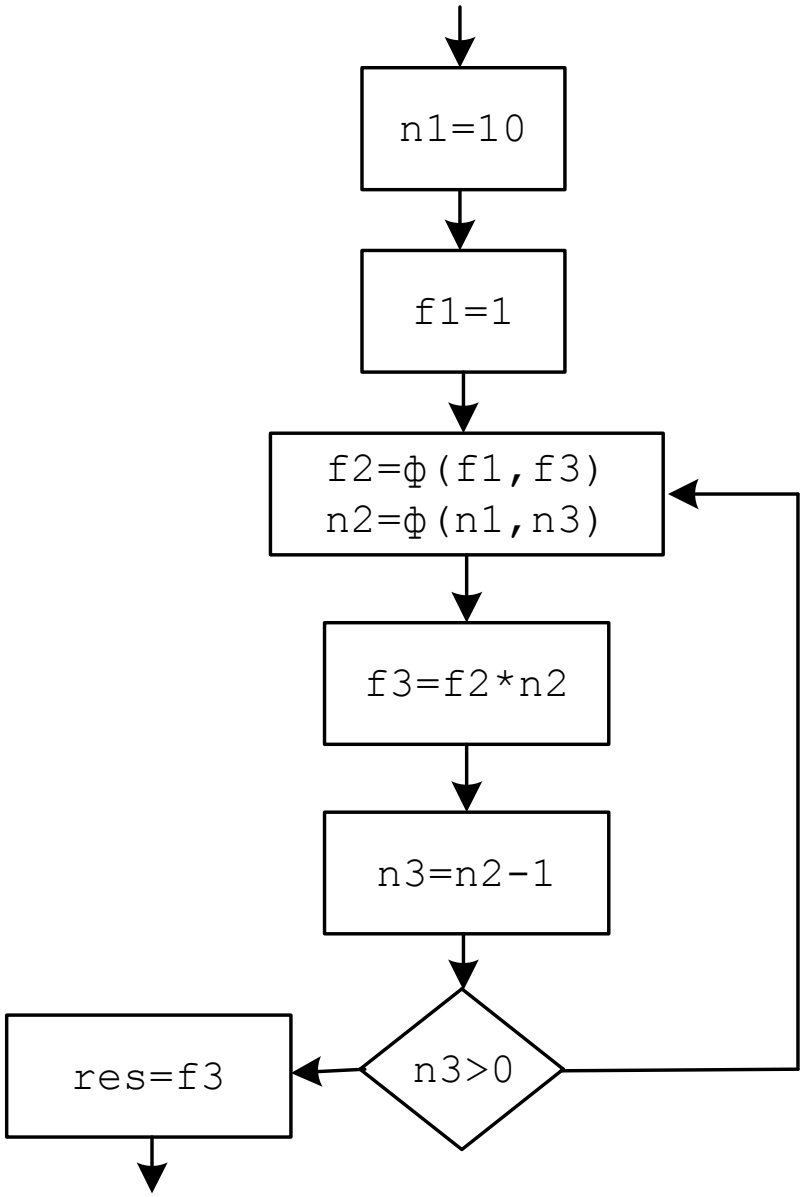


Представление на основе статического однократного присваивания

- Static single assignment form (SSA) – представление программы похожее на CFG
  - каждой переменной значение присваивается только один раз
  - в случае присвоения переменной нескольких значений используется версионирование
  - вводятся ***Φ***-функции, объединяющие ветки программы
  - циклы заменяются инструкциями ветвления и безусловных переходов
  - все выражения находятся в трёхоперандной форме

Пример SSA

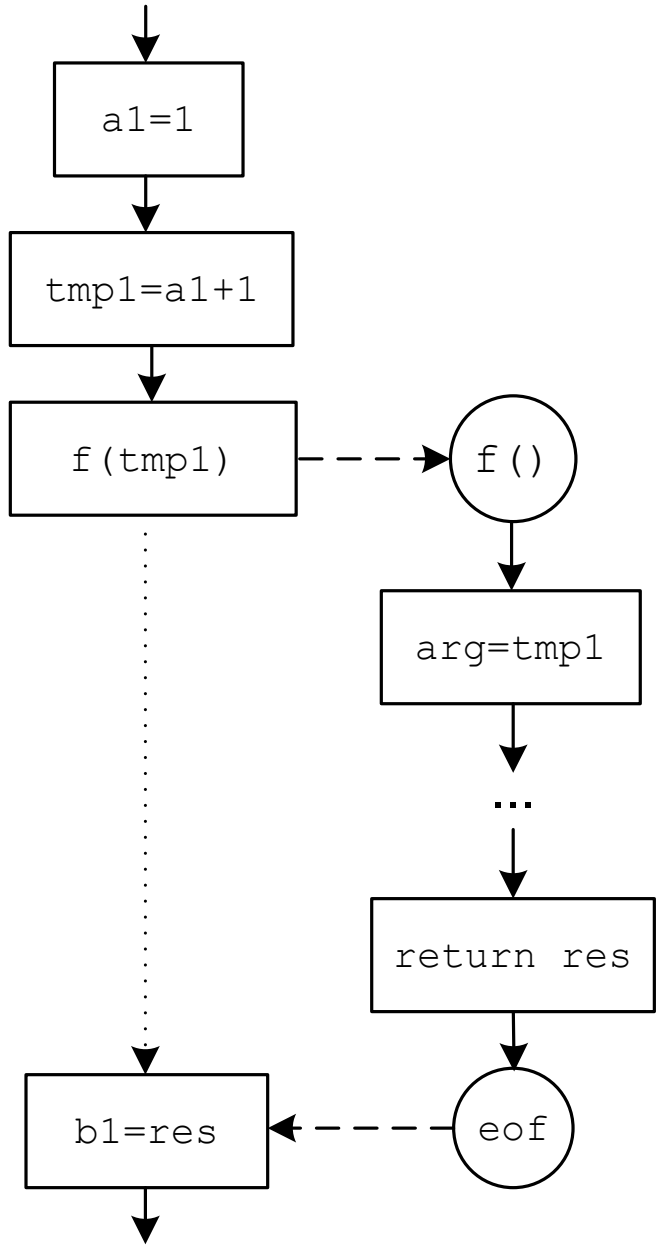
```
int n = 10;
int f = 1;
while (n > 0) {
   f = f * n;
   n = n - 1;
}
res = f;
```



Пример SSA

```
int a = 1;
int b = f(a+1);


float c = g(f(b)*a);
```



Построение модели программы

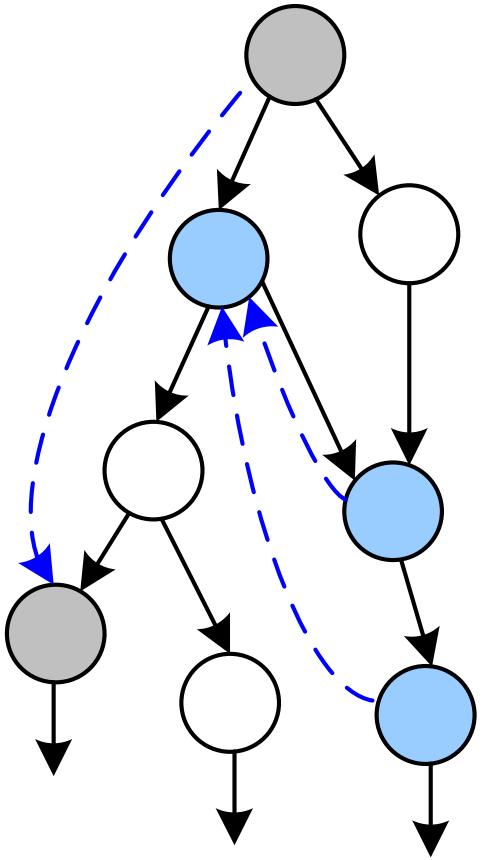
- Средства построения модели программы
  - использование генераторов парсеров (JavaCC, ANTLR, Bison, YACC)
  - использование самостоятельных парсеров (Elsa)
  - использование парсеров в составе средств разработки (NetBeans, CDT)
  - использование средств компиляции (gcc, Clang/LLVM)
  - использование фреймворков для статического анализа (SUIF, CIL, IRE)

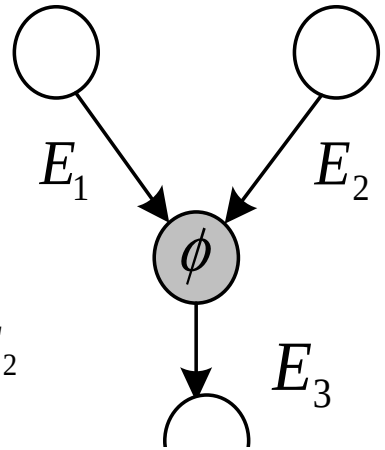
Анализ потока данных

- Анализируются изменения данных в конструкциях программы
- Виды анализа потока данных
  - Reaching Definitions
  - Available Expressions
  - Constant Propagation
  - Very Busy Expressions
  - Live Variables
  - Use-Definition & Definition-Use
  - Анализ состояний объектов программы

Классификация видов анализа

- Направление анализа
  - прямой
  - обратный
- Объединение результатов
  - may-анализ – полные результаты
  - must-анализ – точные результаты





may

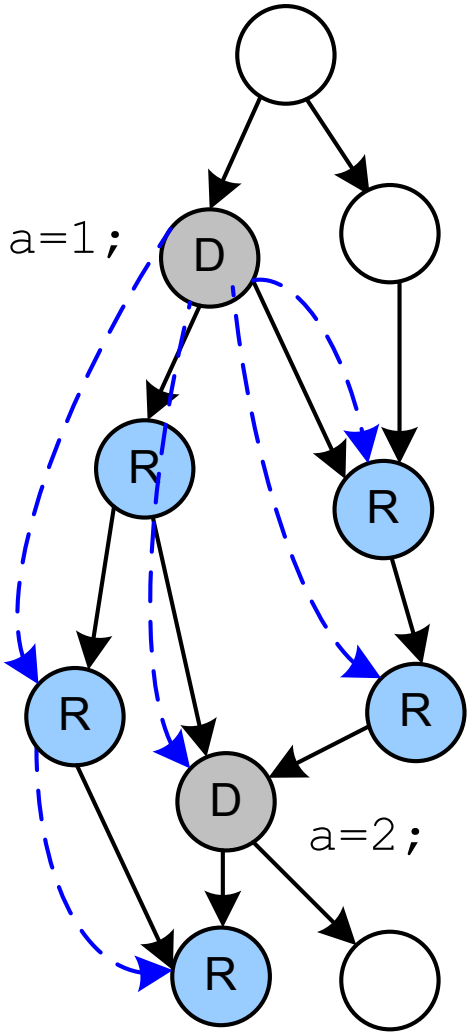
$E_3 = E_1 \cup E_2$

must

$E_3 = E_1 \cap E_2$

Reaching Definitions

- **Reaching Definitions** – определение конструкций, которых может достигать значение переменной присвоенное ей в некоторой конструкции



Reaching Definitions

- RD_j – множество достижимых определений для переменных, с указанием места определения
- $RD_j = (x_{j1}, l_{k1}), \ldots (x_{jn}, l_{kn})$, x – переменная, l – место определения
- Правила анализа

$$[x = a]^k : RD_k = RD_{k-1} \setminus (x, l_j) \cup (x, k)$$

$$[a]^k : RD_k = RD_{k-1}$$

$$[\phi]^k : RD_k = \bigcup_{\forall i \in \Pr ed(k)} RD_i$$

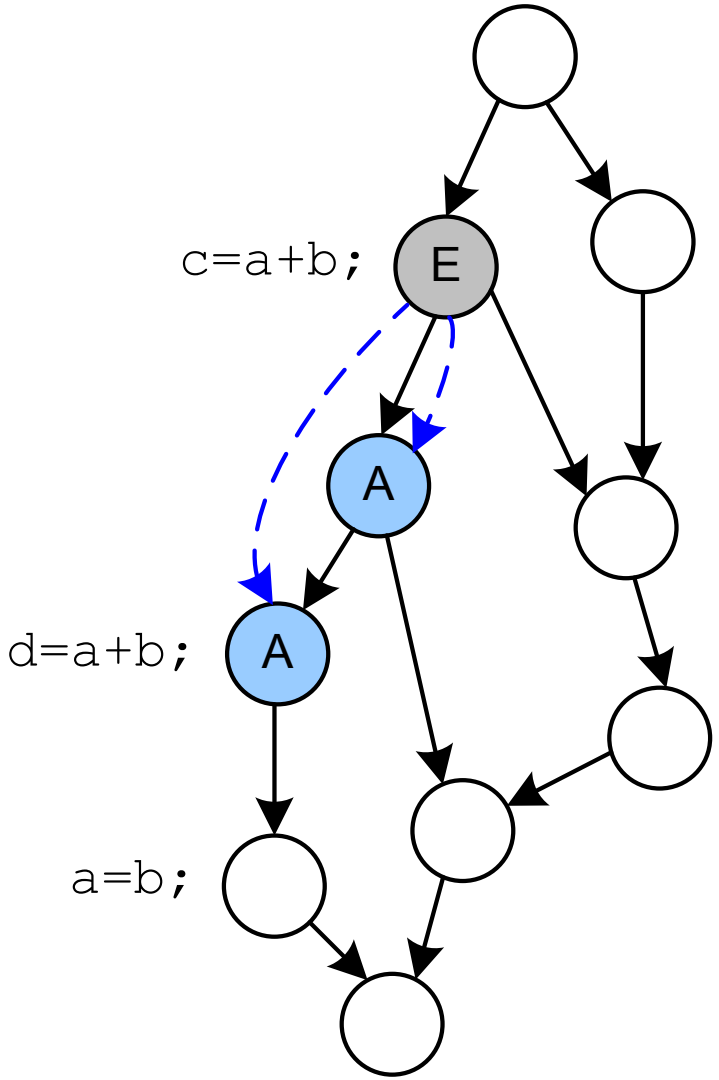
Reaching Definitions

```
0: int a,b;
1: a = 10;
2: b = 20;
3: a = b + 1;
4: if (...) {
5:     a = 3;
6: }
```

```
RD0 = (a,?)(b,?)
RD1 = (a,1)(b,?)
RD2 = (a,1)(b,2)
RD3 = (a,3)(b,2)
RD4 = (a,3)(b,2)
RD5 = (a,5)(b,2)
RD6 = (a,3)(a,5)(b,2)
```

Available Expressions

- **Available Expressions** – анализ ранее предвычисленных выражений, которые не изменились на всех путях до некоторой конструкции программы



Available Expressions

- AE_j – множество предвычисленных выражений
- $AE_j = e_{j1},\ldots e_{jn}$, e – выражение
- Правила анализа

$$[x = a]^l : AE_l = AE_{l-1} \setminus (e : x \in e) \cup (a : x \notin a)$$

$$[a]^l : AE_l = AE_{l-1} \cup (a)$$

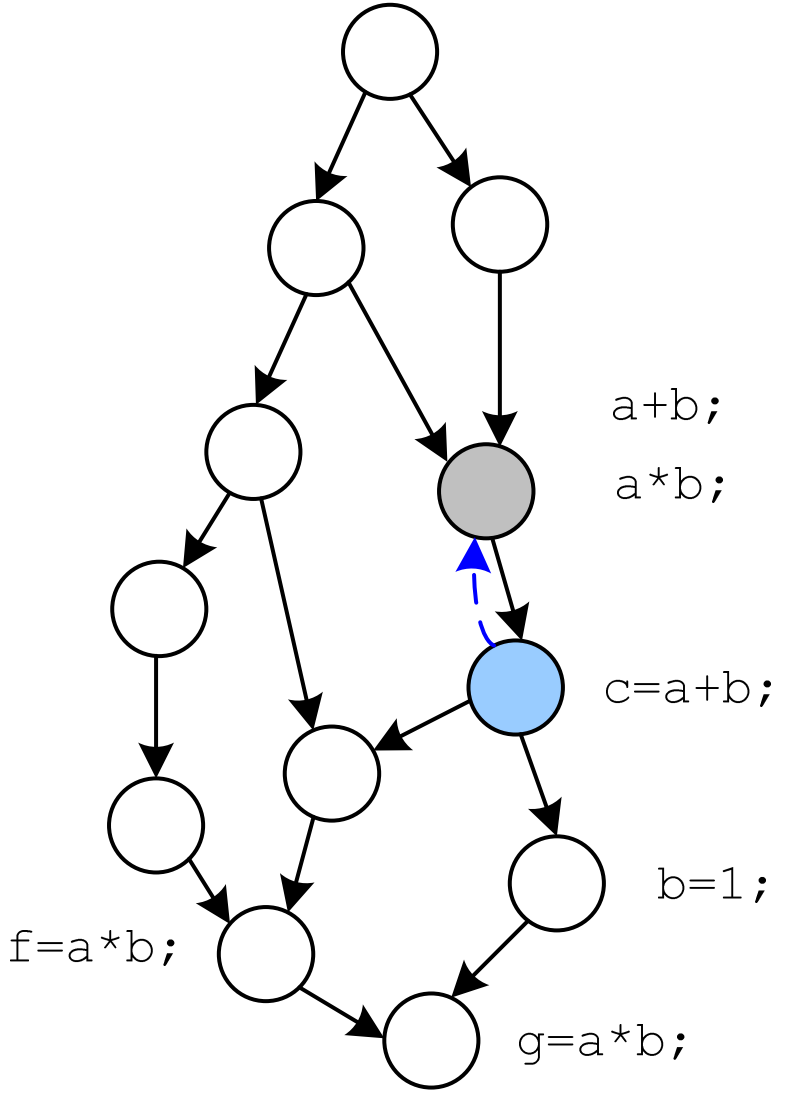
$$[\phi]^l : AE_l = \bigcap_{\forall i \in Pred(l)} AE_i$$

Available Expressions

```
0: int a,b,c;          AE0 = ()
1: a = b + 1;          AE1 = (b+1)
2: c = a * b;          AE2 = (b+1, a*b)
3: b = a + 1;          AE3 = (a+1)
4: if (c < 5){         AE4 = (a+1)
5:     c = a + b;      AE5 = (a+1)(a+b)
6: }                   AE6 = (a+1)
```

Very Busy Expressions

- **Very Busy Expressions** – анализ, определяющий выражения, значения которых гарантированно (на всех путях выполнения) используются до изменения переменных, входящих в это выражение
- Обратный анализ



Very Busy Expressions

- VB_j – множество выражений используемых далее на всех путях выполнения программы
- $VB_j = e_{j1},\ldots e_{jn}$
- Правила анализа

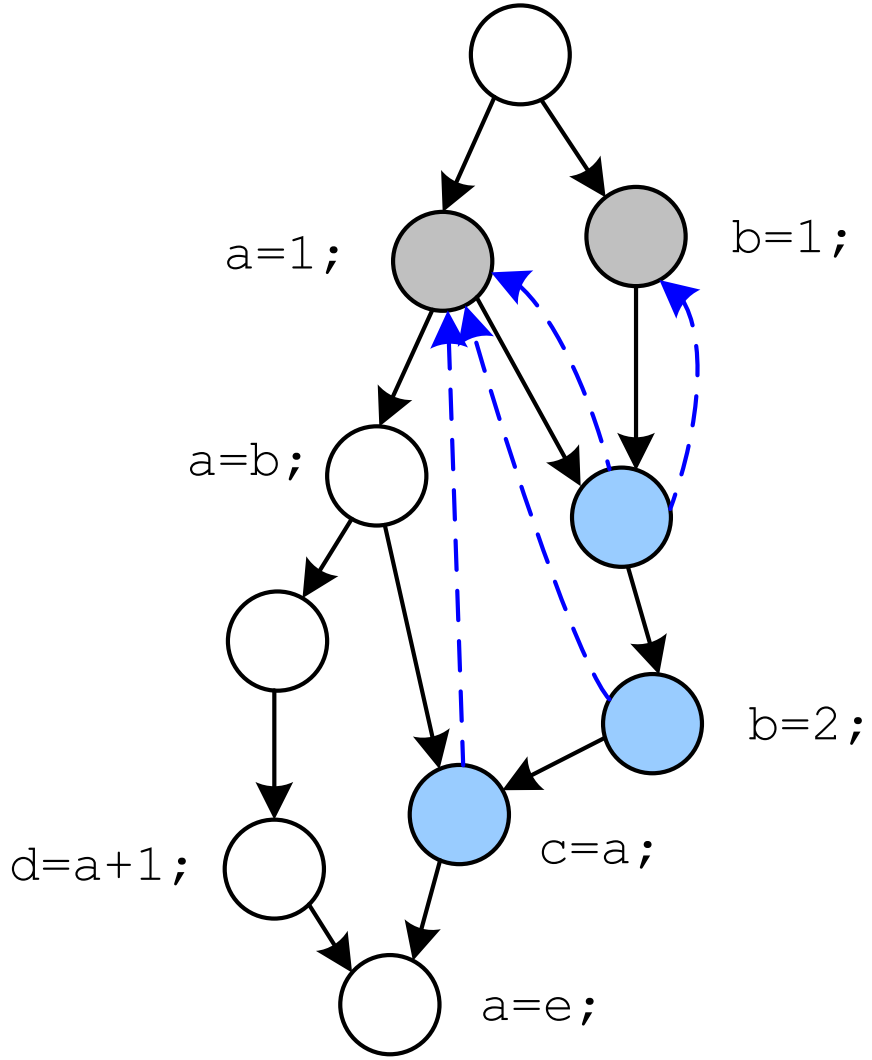
$$[x = a]^l : VB_l = VB_{l+1} \setminus (e : x \in e) \cup (a)$$

$$[a]^l : VB_l = VB_{l+1} \cup (a)$$

$$[if\ ]^l : VB_l = \bigcap_{\forall i \in Succ(l)} VB_i$$

Live Variables

- **Live Variables** – анализ, определяющий переменные в некоторой конструкции, значения которых используется без переопределения хотя бы на одном пути выполнения программы
- Обратный анализ



Live Variables

- LV_j – множество переменных используемых далее хотя бы на одном пути выполнения программы
- $LV_j = v_{j1},\ldots v_{jn}$
- Правила анализа

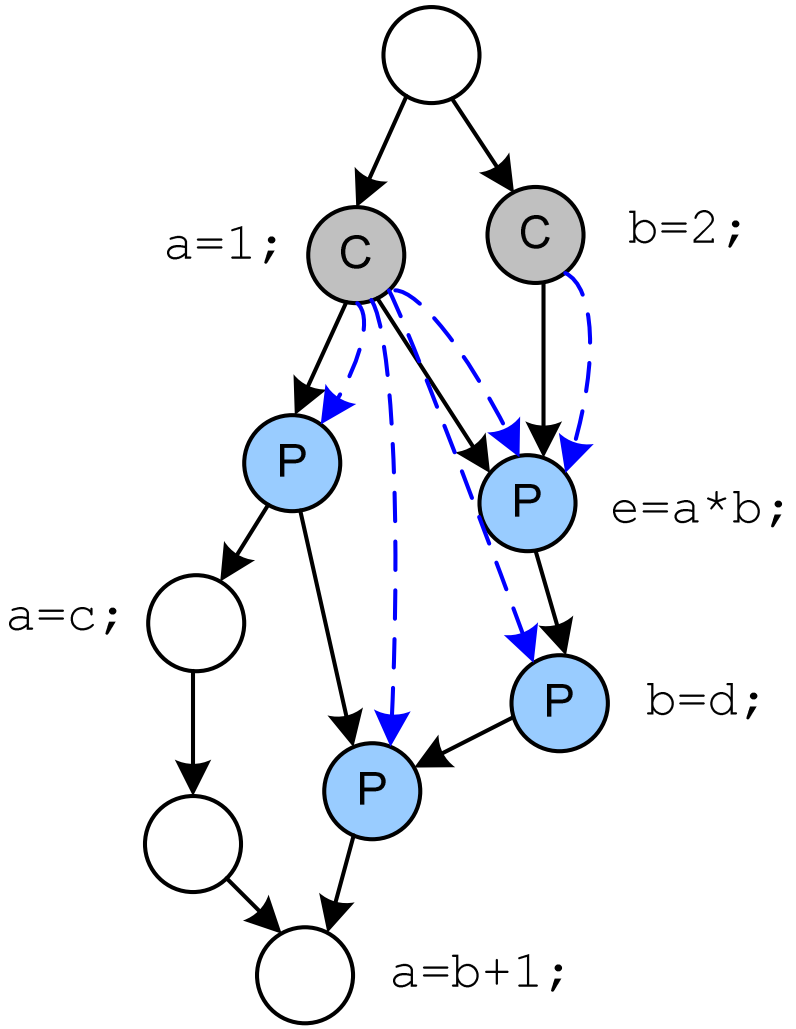
$$[x = a]^l : LV_l = LV_{l+1} \setminus (x) \cup (\forall y \in a)$$

$$[a]^l : LV_l = LV_{l+1} \cup (\forall y \in a)$$

$$[if\ ]^l : VB_l = \bigcup_{\forall i \in Succ(l)} VB_i$$

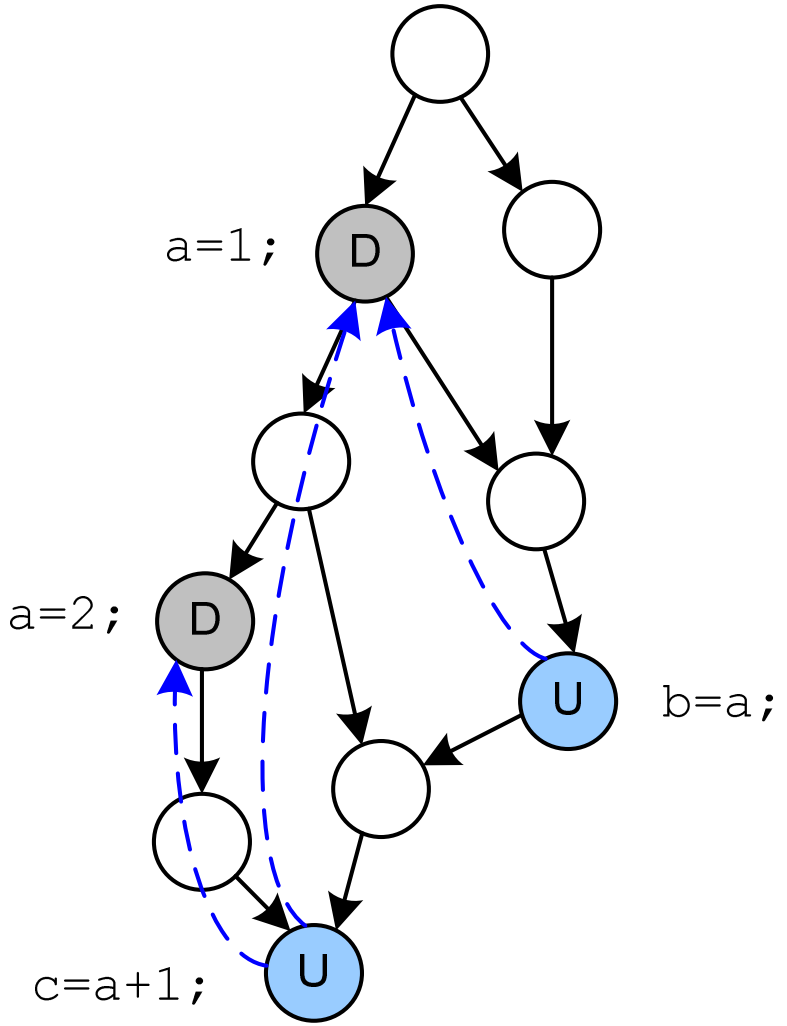
Constant Propagation

- **Constant Propagation** – анализ переменных, которые имеют константные значения в конструкциях программы



Use-Definition

- **Use-Definition** – анализ, определяющий места присваивания значения переменной, если это значение используется в некоторой конструкции программы



Алгоритмы анализа потока данных

- Алгоритмы извлечения необходимой информации
  - подход на основе системы уравнений или ограничений
  - абстрактная интерпретация
  - системы типов и эффектов

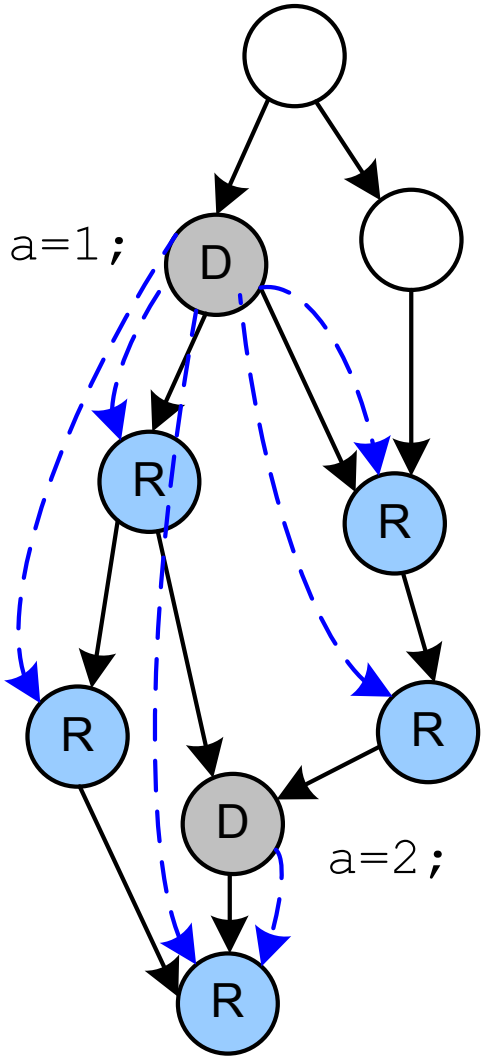
Подход на основе системы уравнений

- На примере RD анализа
- Для каждой конструкции программы строится уравнение
- Решение системы уравнений

$$[x = a]^k : RD_k = RD_{k-1} \setminus (x, l_j) \cup (x, k)$$

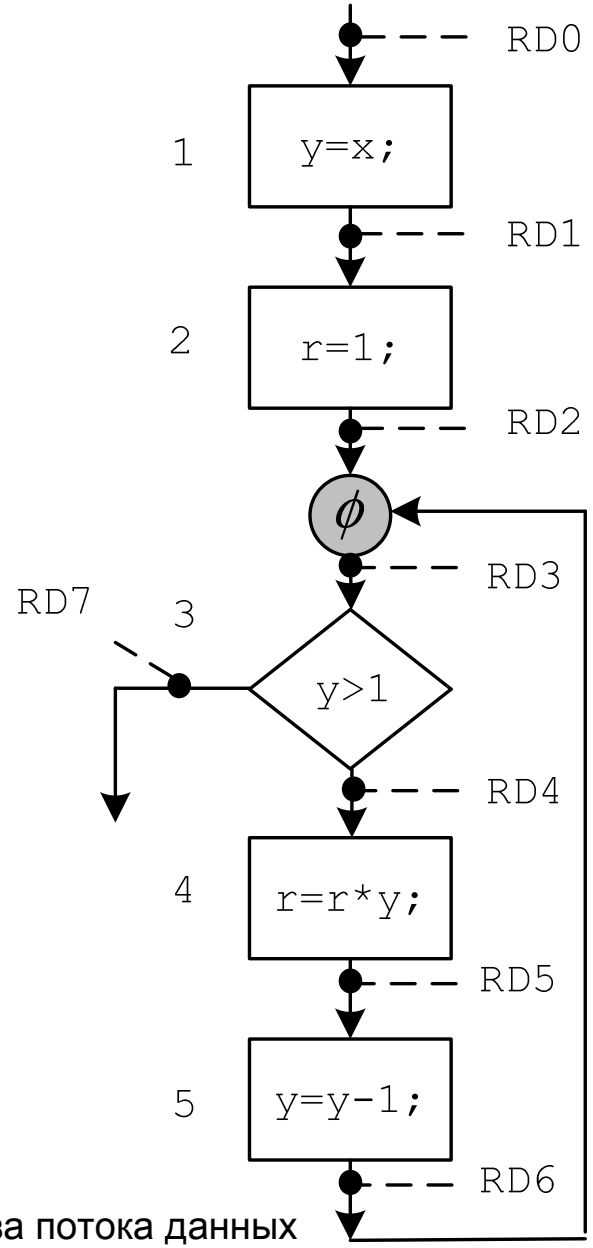
$$[a]^k : RD_k = RD_{k-1}$$

$$[\phi]^k : RD_k = \bigcup_{\forall i \in Pred(k)} RD_i$$

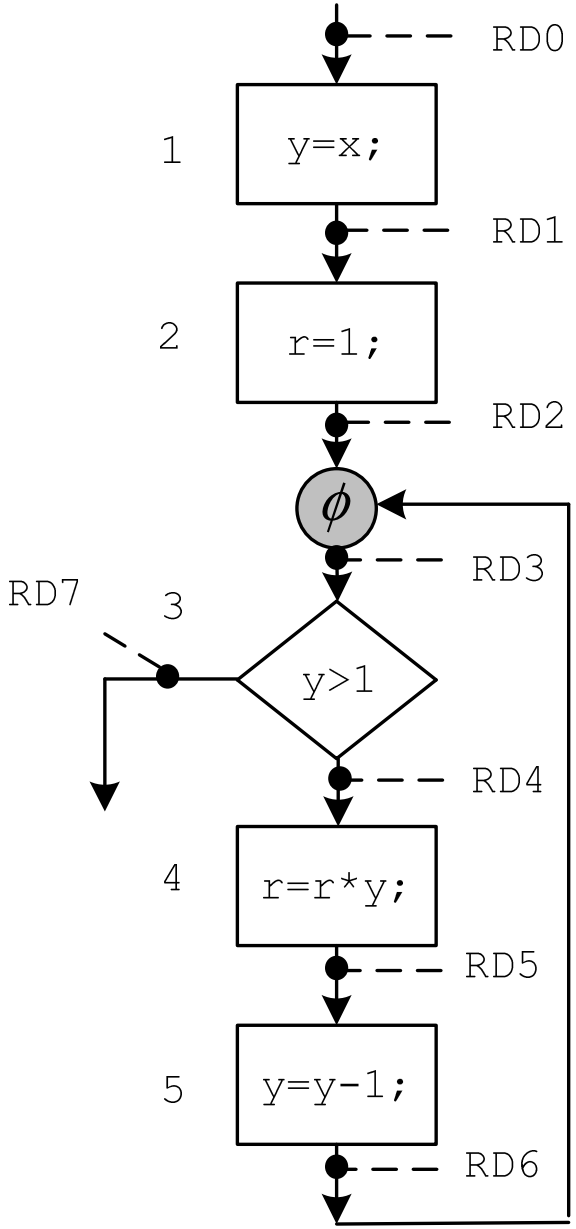


Подход на основе системы уравнений

```
int f(int x) {
   1: int y = x;
   2: int r = 1;
   3: while (y > 1) {
   4:    r = r * y;
   5:    y = y - 1;
   }
   return r;
}
```



Подход на основе системы уравнений



RD0 = (x,?)

RD1 = RD0\(y,i)∪(y,1)

RD2 = RD1\(r,i)∪(r,2)

RD3 = RD2 ∪ RD6

RD4 = RD3

RD5 = RD4\(r,i)∪(r,4)

RD6 = RD5\(y,i)∪(y,5)

RD7 = RD3

RD – множество пар вида (x_{j1}, l_{k1}),

где x_{j1} – переменная программы,

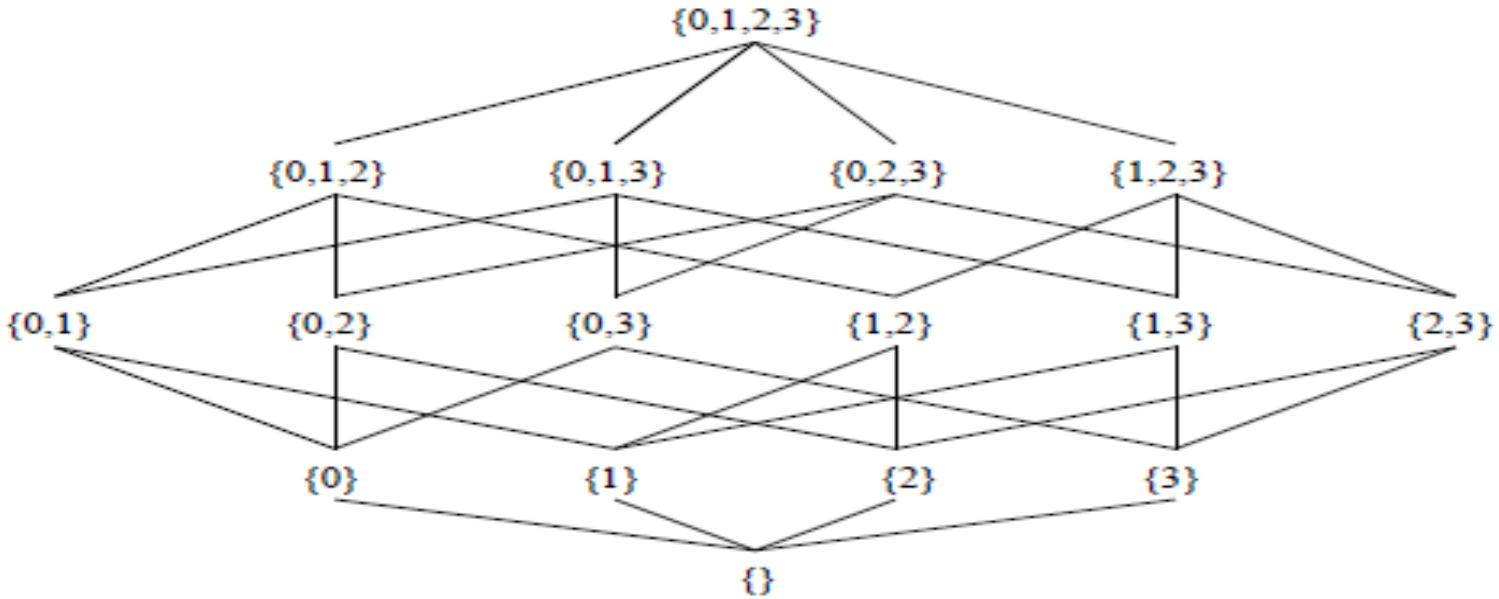
l_{k1} – место последнего определения x_{j1}

Решение системы уравнений

- При отсутствии циклов каждое уравнение достаточно решить один раз
- При наличии циклов используются специальные алгоритмы решения системы уравнений
  - получение решения с требуемыми свойствами (полного решения)
  - получение решения за приемлемое время
- Для описания этих алгоритмов удобно применять математический аппарат теории решеток

Решетка

- Решетка – частично упорядоченное множество, для каждого подмножества которого определена единственная точная верхняя грань и единственная точная нижняя грань



- *M. Schwartzbach* Lecture Notes on Static Analysis

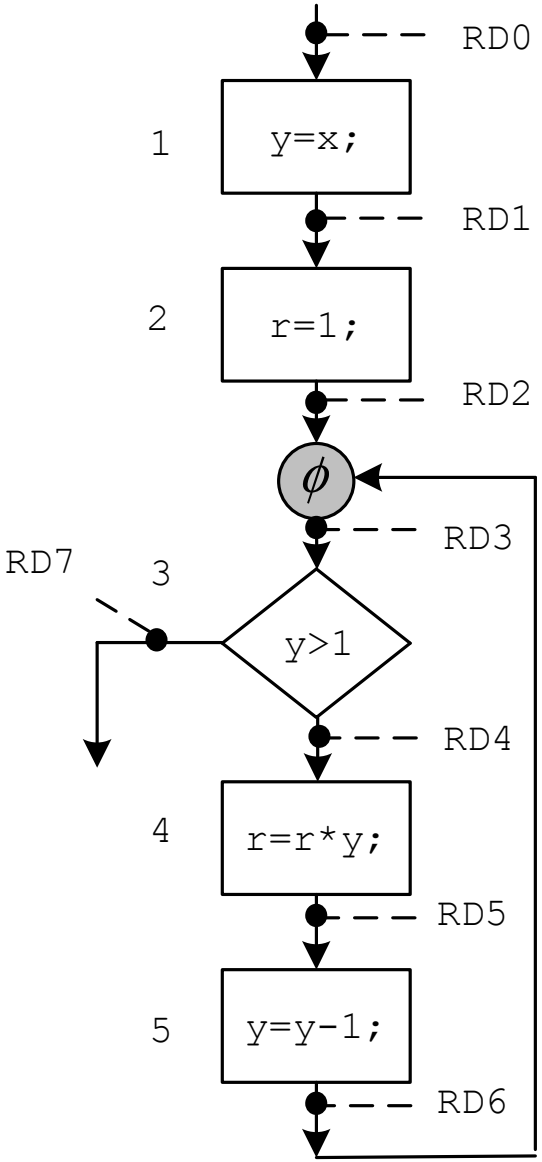
Отношение частичного порядка

- Решетка, состоит из подмножеств множества (x_i, l_j), отношением порядка является отношение включения подмножеств

$$RD_1 \subseteq RD_2 \Leftrightarrow \forall (x_i, l_j) \in RD_1 \Rightarrow (x_i, l_j) \in RD_2$$

- RD_1= {(a,1), (b,2)} ? RD_2= {(a,1)}
- RD_1= {(a,1), (b,2)} ? RD_2= {(a,1), (b,1), (b,2)}
- RD_1= {(b,1), (b,2)} ? RD_2= {(b,1), (b,2)}
- RD_1= {(b,1), (c,2), (c,3)} ? RD_2= {(b,1), (b,2), (c,3)}

Пример решетки для RD анализа



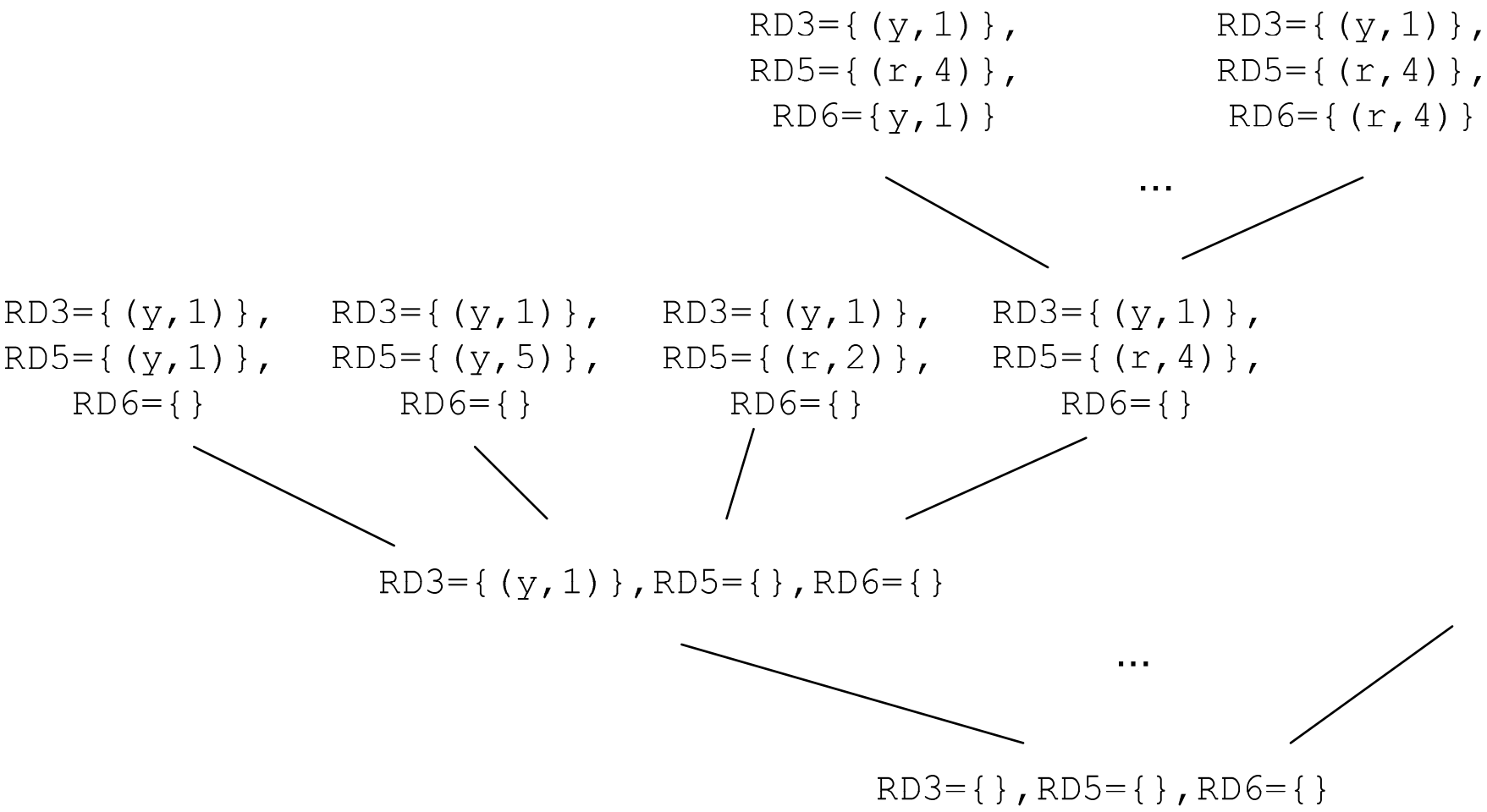
- Построим решетку для `RD3, RD5, RD6`
- Возможные значения

  `(y,1), (y,5), (r,2), (r,4)`
- Низ решетки

  `RD3={},RD5={},RD6={}`
- Верх решетки

  `RD3=RD5=RD6=`
  `{(y,1),(y,5),(r,2),(r,4)}`

Пример решетки для RD анализа



Представление СУ с помощью функции

- Система уравнений состоит из уравнений вида

$$RD_l = f_l(RD_1, \ldots, RD_n)$$

- СУ может быть представлена с помощью функции

$$F(\overline{RD}) = (f_1(\overline{RD}), \ldots, f_n(\overline{RD}))$$

- Функция *F(X)* является монотонной, если справедливо

$$\forall RD_1, RD_2 : RD_1 \subseteq RD_2 \Rightarrow F(RD_1) \subseteq F(RD_2)$$

Наименьшая неподвижная точка (LFP)

- Любая функция *F(X)*, монотонная над элементами конечной решетки, имеет единственную **наименьшую неподвижную точку** (Least Fixed Point), для которой справедливо

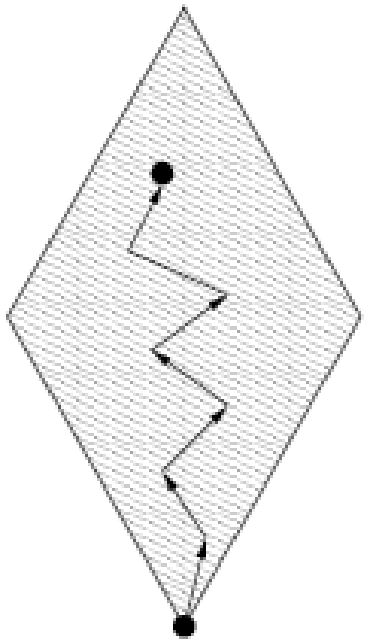
$$F(LFP) = LFP$$

- LFP является минимально возможным решением СУ
- Наименьшая неподвижная точка для заданной монотонной функции может быть получена путем ее многократного применения к нижней грани решетки

$$F^1(\bot) = F(\bot),\ F^2(\bot) = F\left(F^1(\bot)\right), \ldots, F^{n+1}(\bot) = F\left(F^n(\bot)\right)$$

Алгоритмы вычисления LFP

- Алгоритм хаотических итераций (**Chaotic Iteration algorithm**)
- Алгоритм на основе списков зависимых конструкций (**Worklist algorithm**)
- Другие алгоритмы (**Strong Components algorithm**)



Алгоритм хаотических итераций

- Алгоритм хаотических итераций
  1. выполняется решение всех уравнений
  2. если результат решения хотя бы одного уравнения изменился по сравнению с предыдущим решением (итерацией), то п. 1.

$$RD_1 = \varnothing, \ldots, RD_n = \varnothing$$
$$\exists j : RD_j \neq f_j\left(RD_1, \ldots, RD_n\right)$$

- Не учитывает зависимостей между уравнениями программы, в результате решается существенно больше уравнений чем требуется, не применим для реальных программ

Worklist - алгоритм

- Алгоритм на основе списков зависимых конструкций
  1. выполняется решение всех уравнений
  2. для каждого решенного уравнения если результат изменился определяются уравнения, на которые оно влияет, эти уравнения добавляются в список решаемых
  3. если список решаемых уравнений не пуст, то п. 2

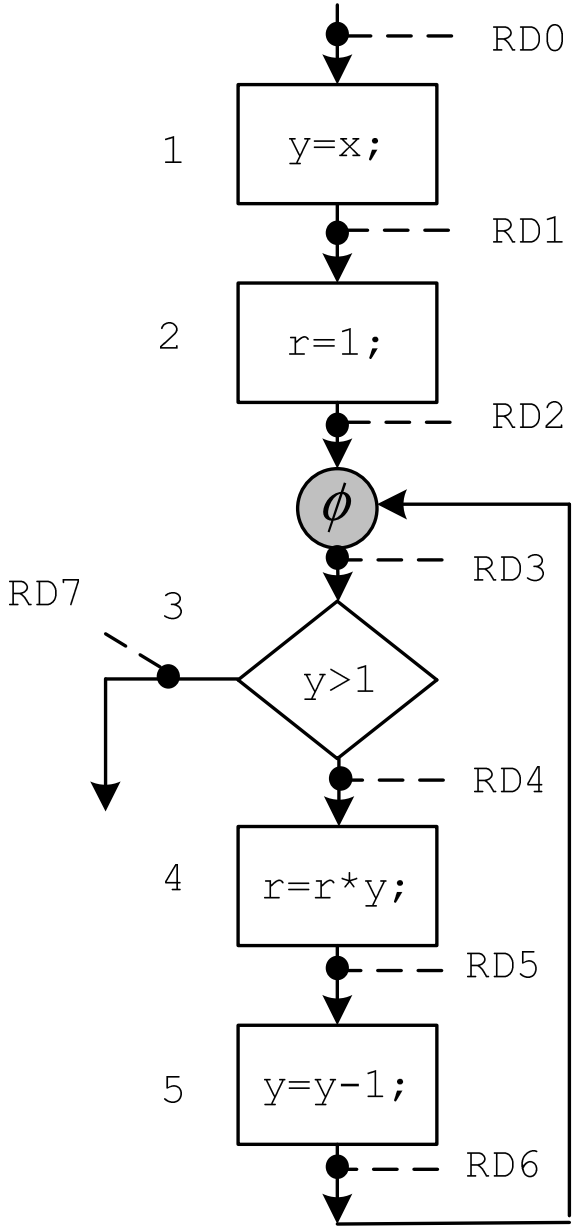
$$RD_1 = \varnothing, \ldots, RD_n = \varnothing, W_k = \varnothing$$
$$W_{k-1} \neq \varnothing : \forall RD_j \in W_{k-1} \Rightarrow$$
$$RD_j \neq f_j(RD_1, \ldots, RD_n), W_k = W_k \cup E_j$$

- На практике бывает сложно определить взаимные влияния уравнений, не обеспечивается оптимальный порядок решения уравнений для вложенных циклов

Strong Components - алгоритмы

- Алгоритмы на основе определения сильносвязанных частей в графе программы (сильносвязанных множеств уравнений)
  1. определяются сильносвязанные множества уравнений
  2. выполняется решение уравнений каждого множества, полученные результаты распространяются на между множествами
  3. процесс повторяется необходимое числа раз
- Многоуровневая иерархия

Решение системы уравнений RD



RD0 = (x,?)

RD1 = RD0\(y,i)∪(y,1)

RD2 = RD1\(r,i)∪(r,2)

RD3 = RD2 ∪ RD6

RD4 = RD3

RD5 = RD4\(r,i)∪(r,4)

RD6 = RD5\(y,i)∪(y,5)

RD7 = RD3

RD1 = (y,1)(r,?)	RD3 = (y,1)(y,5)
RD2 = (y,1)(r,2)	(r,2)(r,4)
RD3 = (y,1)(r,2)	RD5 = (y,1)(y,5)(r,4)
RD5 = (y,1)(r,4)	RD6 = (y,5)(r,4)
RD6 = (y,5)(r,4)	RD7 = (y,1)(y,5)
RD7 = (y,1)(r,2)	(r,2)(r,4)

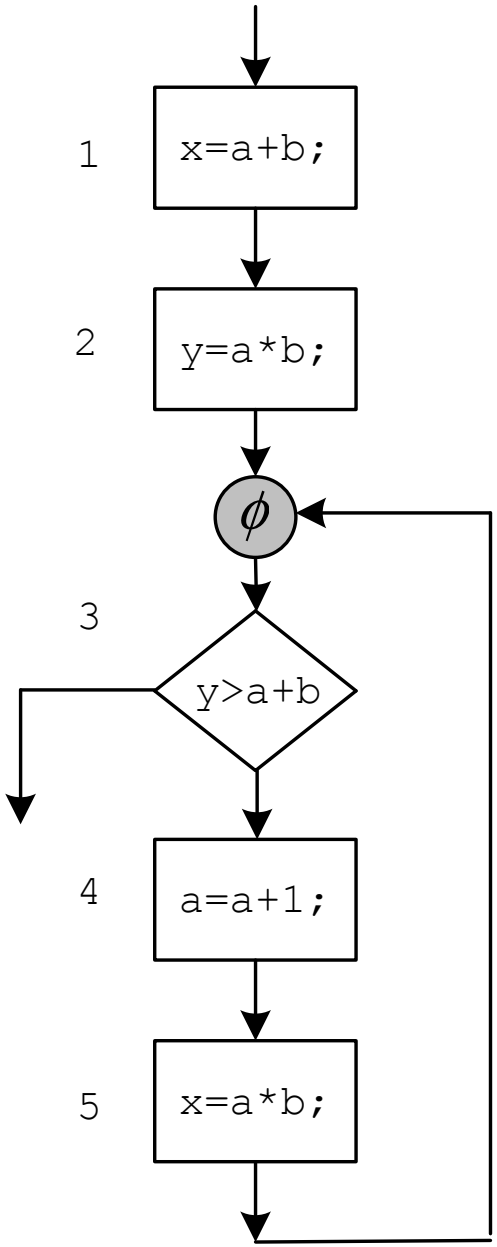
Абстрактная интерпретация

- Abstract Interpretation, P. and R. Cousot
- Общая теория, которая задает способ аппроксимации семантики динамических дискретных систем
- В основе подхода лежит аппроксимация семантики поведения программы для получения вычислимой и при этом достоверной семантики
- Полученная аппроксимация должна обеспечивать проверку частичных свойств программы
- АИ может рассматриваться как частичное выполнение программы с получением требуемой семантической информации

Распространение информации по путям выполнения программы

- Решение СУ путем распространения информации по всем **реализуемым** путям выполнения программы
- Может выполняться как **may**, так и **must** анализ
- Анализ начинается с первой конструкции программы и завершается в после прохождения всех путей
- Для снижения ресурсоемкости используются различные аппроксимации (упрощения)
- Отличия АИ от обычной интерпретации?

Решение системы уравнений AE



AE0 = ()

AE1 = AE0\(e:x)∪(a+b)

AE2 = AE1\(e:y)∪(a*b)

AE3 = AE2 ∩ AE6

AE4 = AE3 ∪ (a+b)

AE5 = AE4\(e:a)

AE6 = AE5\(e:x)∪(a+b)

AE7 = AE3

AE1 = (a+b)	AE3 = (a*b)
AE2 = (a+b,a*b)	AE4 = (a*b,a+b)
AE3 = ()	AE5 = ()
AE4 = (a+b)	AE6 = (a*b)
AE5 = ()	AE7 = (a*b)
AE6 = (a*b)	

Нефункциональные ошибки в программах на языках C/C++

- Нефункциональные ошибки
  - В последовательных программах – **программные дефекты**
  - В параллельных программах программные дефекты и ошибки синхронизации
- Классификация программных дефектов
  - CERT https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard
  - Coverity Open Source reports http://www.scan.coverity.com/report/

Классификация программных дефектов

#	Наименование	Краткое описание
RES	Ошибки управления ресурсами и памятью	Множественное освобождение, нарушение протокола работы с ресурсом
LEAK	Утечки ресурсов и динамической памяти	Потеря последней ссылки на выделенный ресурс или память
BUF	Ошибки работы с буферами/ массивами	Выход за границы объекта при чтении или записи, некорректные операции с указателями
INI	Ошибки отсутствия инициализации	Использование неинициализированных переменных или разыменование указателей
EXP	Ошибки в арифметических операциях	Деление на ноль, переполнение
DC	Мертвый код	Наличие недостижимых частей программы
IO	Ошибки ввода/вывода	Ошибки форматной строки, использование опасных функций ввода

Примеры программных дефектов (RES)

- Примеры ошибок управления ресурсами/памятью
  - повторное освобождение памяти
  - нарушение протокола работы с ресурсом

```
int main() {
  int *p = malloc(SIZE);
  …
  free(p);
  …
  free(p);
}
```

```
int main() {
  FILE f = fopen(fname, “r”);
  …
  fwrite(…, f);
}
```

Примеры программных дефектов (LEAK)

- Примеры утечек ресурсов/памяти
  - утечка динамической памяти при выходе из функции
  - утечка динамической памяти при выделении нового объекта

```
int f() {
  int *p = malloc(SIZE);
  …
  return 0;
}
```

```
int main() {
  int *p = malloc(SIZE);
  int *q = malloc(SIZE);
  …
  if (flag){
    p = q;
  }
  …
}
```

Примеры программных дефектов (BUF)

- Примеры ошибок работы с буферами/массивами
  - разыменование указателя за границей объекта
  - операции над указателями на разные объекты

```
int main()
{
  int arr[10];
  int *p;
  …
  p = arr;
  for(int i=0; i<=10; i++)
     printf("%d" , *p++);
}
```

```
int main()
{
  int a[10];
  int b[15];
  int *p = a;
  int *q = b+10;
  …
  int diff = (p - q);
}
```

Примеры программных дефектов (INI)

- Примеры ошибок отсутствия инициализации
  - использование неинициализированной переменной
  - разыменование неинициализированного указателя

```
int main() {
  int i;
  …
  if (i > 0){
    …
  }
  …
}
```

```
int main() {
  int *p;
  …
  if (flag){
    int i =*p;
  }
  …
}
```

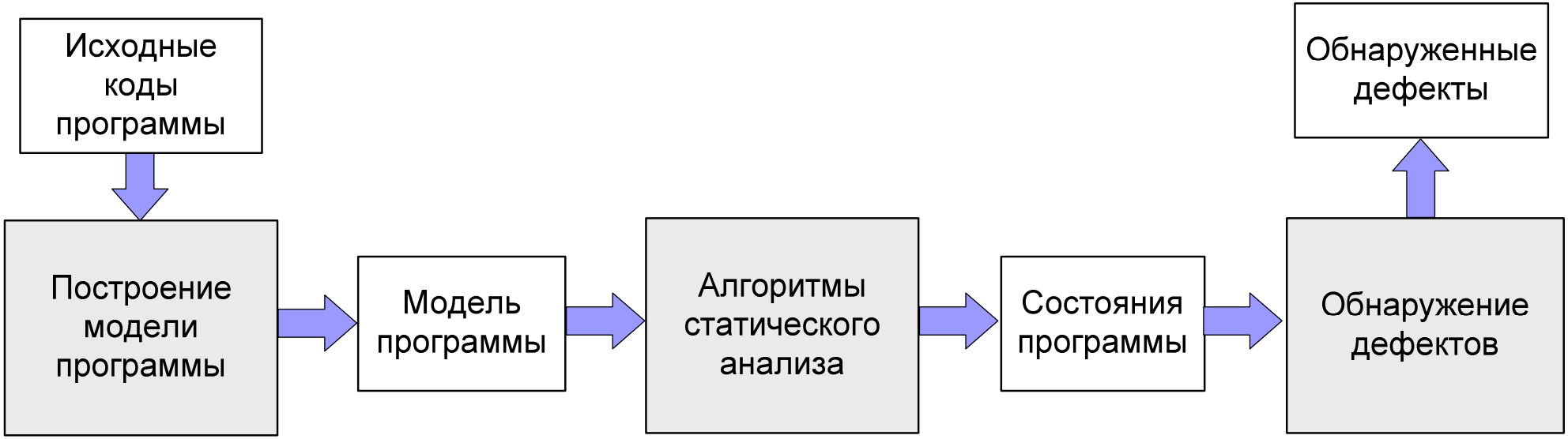
Примеры программных дефектов (IO)

- Примеры ошибок ввода/вывода
  - использование неконтролируемого значения в качестве форматной строки
  - использование опасных функций ввода

```
int main()
{
  char value[50];
  scanf("%49s", value);
  …
  printf(value);
}
```

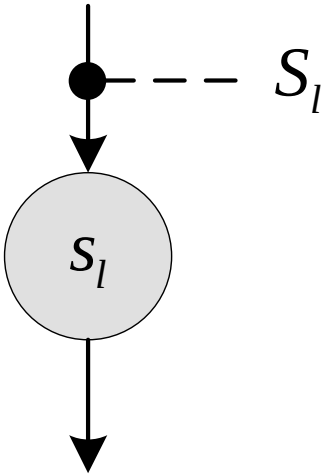
```
int main()
{
  char* s = malloc(10);
  …
  gets(s);
}
```

Общая структура статического анализа



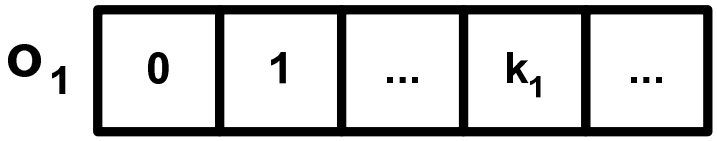
Анализ программ на основе состояний

- Определяются состояния объектов программы в отдельных конструкциях
- Состояние программы – возможные значения всех объектов, видимых в данной точке
- Состояние программы представляет как множество кортежей



$$S_l = \{tuple_i\} \qquad tuple = \langle \text{var}, value \rangle$$

$$\text{var} = (o_1, k_1)$$

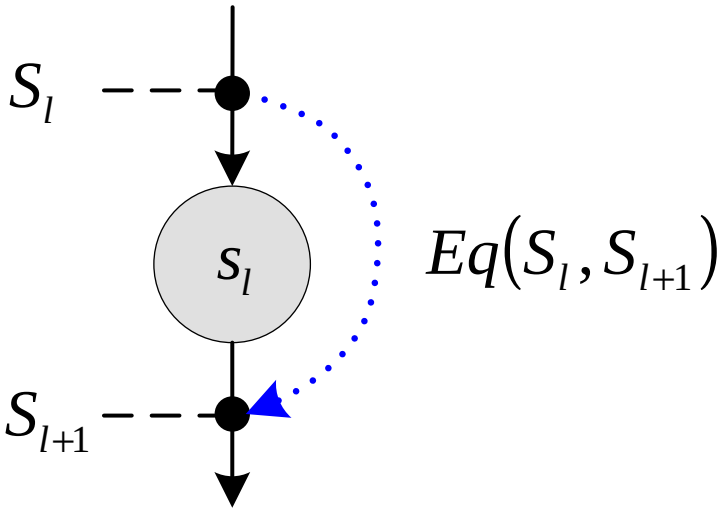


Алгоритмы определения состояний

- Различные типы объектов программы – различные правила и различные значения в правой части кортежей
- Анализ потока данных (анализ конструкций программы внутри процедур/функций)
  - алгоритм интервального анализа
  - алгоритм анализа указателей
  - алгоритм ресурсного анализа
- Межпроцедурный анализ

Общая организация алгоритмов

- Алгоритмы анализа представлены в виде правил для конструкций программы (правила анализа д.б. монотонны)
- Каждое правило формирует уравнение, переменными являются состояния объектов программы



$$Rule(s_l) \rightarrow Eq(S_l, S_{l+1})$$

- Строится система уравнений, в результате решения которой получаются состояния объектов программы

Алгоритм интервального анализа

- Правила для конструкций, результат выполнения которых имеет интервальный тип (`char`, `int`, `float` и т.д.)
- Значения для интервальных переменных

$$value = i, i = (i_{\min}, i_{\max})$$

$$\langle\langle o_1, k_1 \rangle (i_{\min}, i_{\max}) \rangle\rangle$$

- Для учета неинициализированных переменных используется специальный интервал $i_{noninit}$

Правила интервального анализа

$$[declare(T\ a)]^l: \quad S_{l+1} = S_l \cup \left\langle \left\langle a,0 \right\rangle, i_{noninit} \right\rangle$$

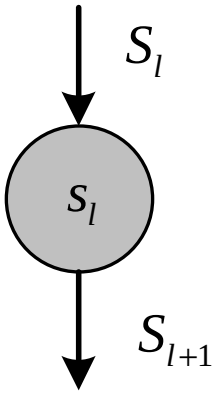
$$\left\langle a,0 \right\rangle \Leftrightarrow a$$

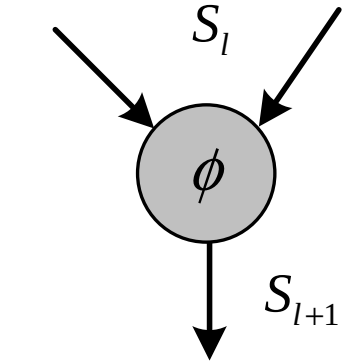
$$[undeclare(a)]^l: \quad S_{l+1} = S_l \setminus \bigcup_{\forall \left\langle x,i \right\rangle \in S_l : x=a} \left\langle a,i \right\rangle$$

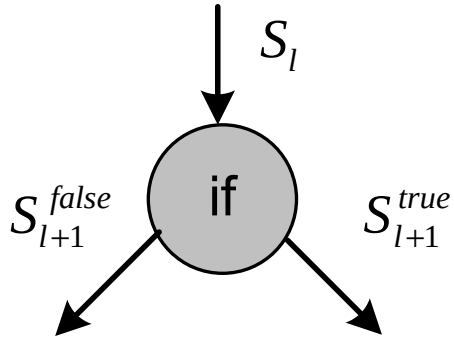
$$[\phi(...)]^l : S_{l+1} = \cup S_l$$

$$[if\ (cond)\ s_1; s_2]^l: \quad S_{l+1}^{true} = S_l, \ S_{l+1}^{false} = S_l$$

$$S_{l+1}^{true} = \eta(S_l, cond), S_{l+1}^{false} = \eta\left(S_l, \overline{cond}\right)$$







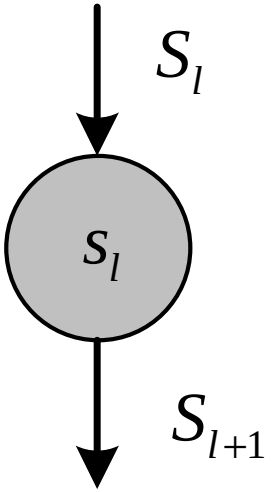
Правила интервального анализа

$$[a = C]^l : \quad S_{l+1} = S_l \setminus \bigcup_{\forall \langle x,i \rangle \in S_l : x = a} \langle a, i \rangle \cup \langle a, C \rangle \ \ (C - const)$$

$$[a = b]^l : \quad S_{l+1} = S_l \setminus \bigcup_{\forall \langle x,i \rangle \in S_l : x = a} \langle a, i \rangle \cup \bigcup_{\forall \langle x,j \rangle \in S_l : x = b} \langle a, j \rangle$$

$$[a = b + d]^l : \quad S_{l+1} = S_l \setminus \bigcup_{\forall \langle x,i \rangle \in S_l : x = a} \langle a, i \rangle \cup$$

$$\bigcup_{\forall \langle b,k \rangle, \langle d,j \rangle \in S_l : k, j \neq i_{noninit}} \langle a, \gamma(a, k, j) \rangle \cup \bigcup_{\exists \langle b,k \rangle, \langle d,j \rangle \in S_l : k \vee j = i_{noninit}} \langle a, i_{noninit} \rangle$$



$$\forall \langle a, (i_{\min}, i_{\max}) \rangle$$

$$\gamma(a, k, j) = \left(\min(k, j) + i_{\min}, \max(k, j) + i_{\max} \right)$$

Алгоритм анализа указателей

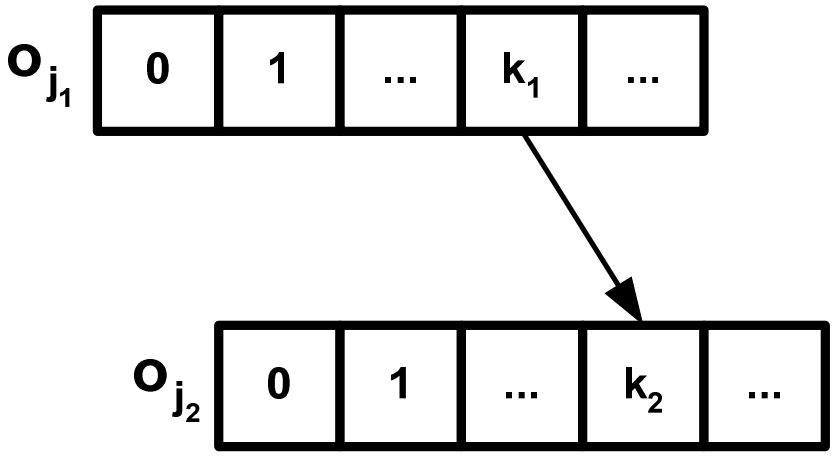
- Правила для конструкций, результат выполнения которых является указателем на объект или функцию
- Значения для переменных-указателей на объекты

$$value = \langle o, k \rangle$$

- Для учета неинициализированных переменных используется специальный объект $o_{invalid}$
- Значение NULL представляется с помощью специального объекта o_{null}

Представление состояния памяти

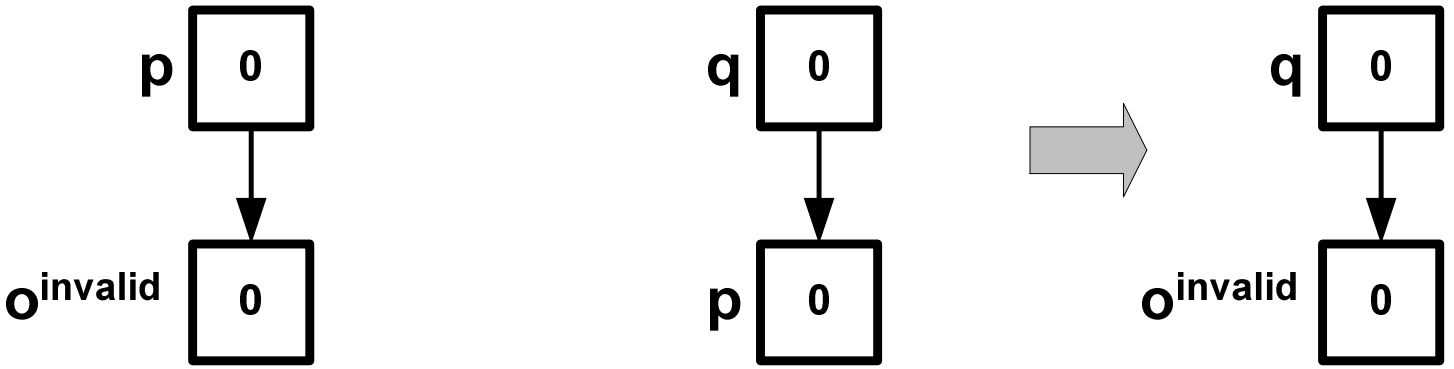
- Состояния переменных-указателей представляются в виде кортежей $\langle\langle o_1, k_1\rangle, \langle o_2, k_2\rangle\rangle$
- Представление произвольных отношений между объектами



Правила алгоритма анализа указателей

$$[declare(T * p)]^l : \quad S_{l+1} = S_l \cup \left\langle \left\langle p,0 \right\rangle, o_{invalid} \right\rangle$$

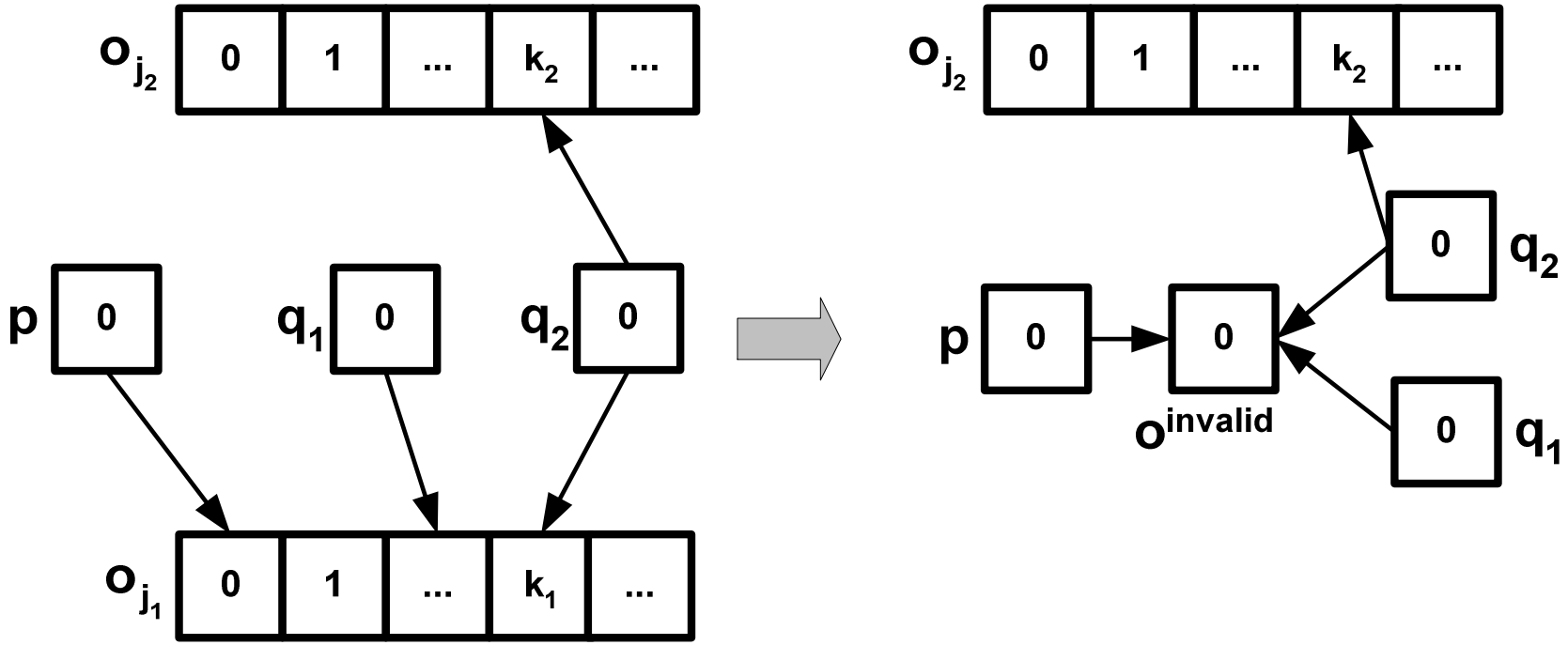
$$[undeclare(p)]^l : \quad S_{l+1} = S_l \setminus \bigcup_{\forall \left\langle \left\langle q,k_1 \right\rangle \left\langle p,0 \right\rangle \right\rangle \in S_l} \left\langle \left\langle q,k_1 \right\rangle \left\langle p,0 \right\rangle \right\rangle \cup \bigcup_{\forall \left\langle \left\langle q,k_1 \right\rangle \left\langle p,0 \right\rangle \right\rangle \in S_l} \left\langle \left\langle q,k_1 \right\rangle, o_{invalid} \right\rangle$$



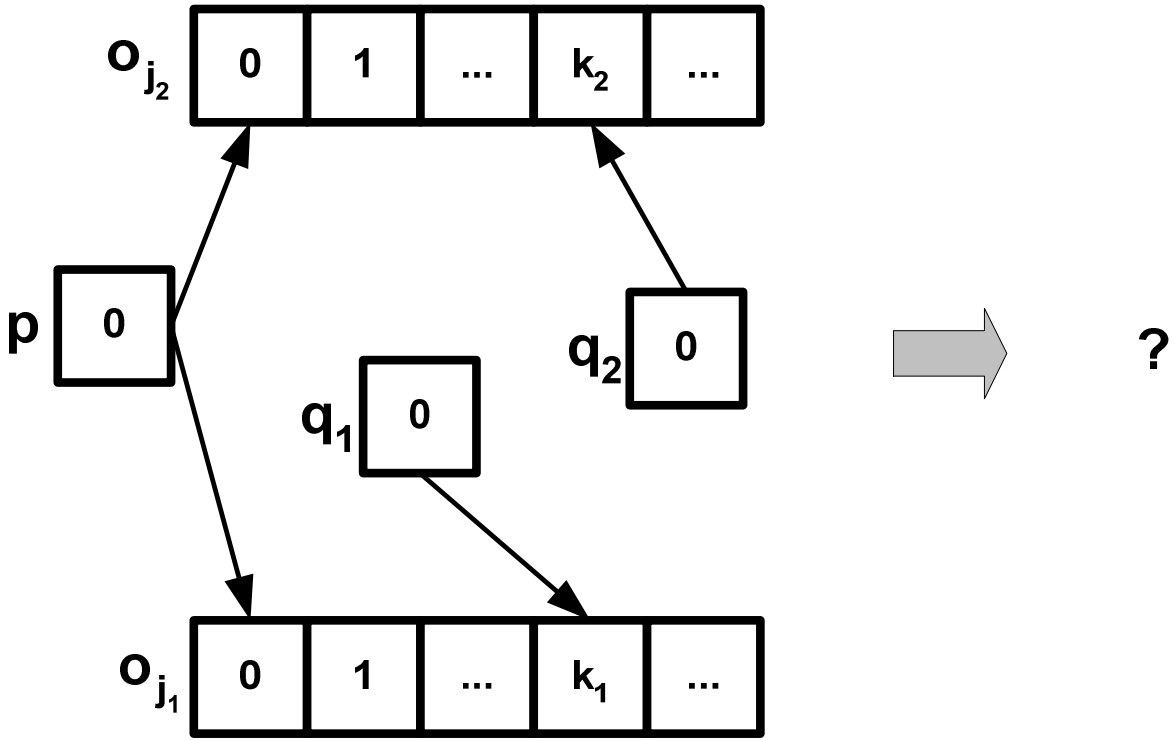
Примеры правил анализа указателей

$$[\,free(p)]^l: \quad S_{l+1} = S_l \setminus \bigcup_{\forall\langle\langle p,0\rangle\langle o_{j1},k_1\rangle\rangle\in S_l} \langle\langle p,0\rangle\langle o_{j_1},k_1\rangle\rangle \setminus \bigcup_{\forall\langle\langle p,0\rangle\langle o_{j_1},k_1\rangle\rangle,\langle\langle q,k_2\rangle\langle o_{j1},k_3\rangle\rangle\in S_l} \langle\langle q,k_2\rangle\langle o_{j1},k_3\rangle\rangle \cup$$

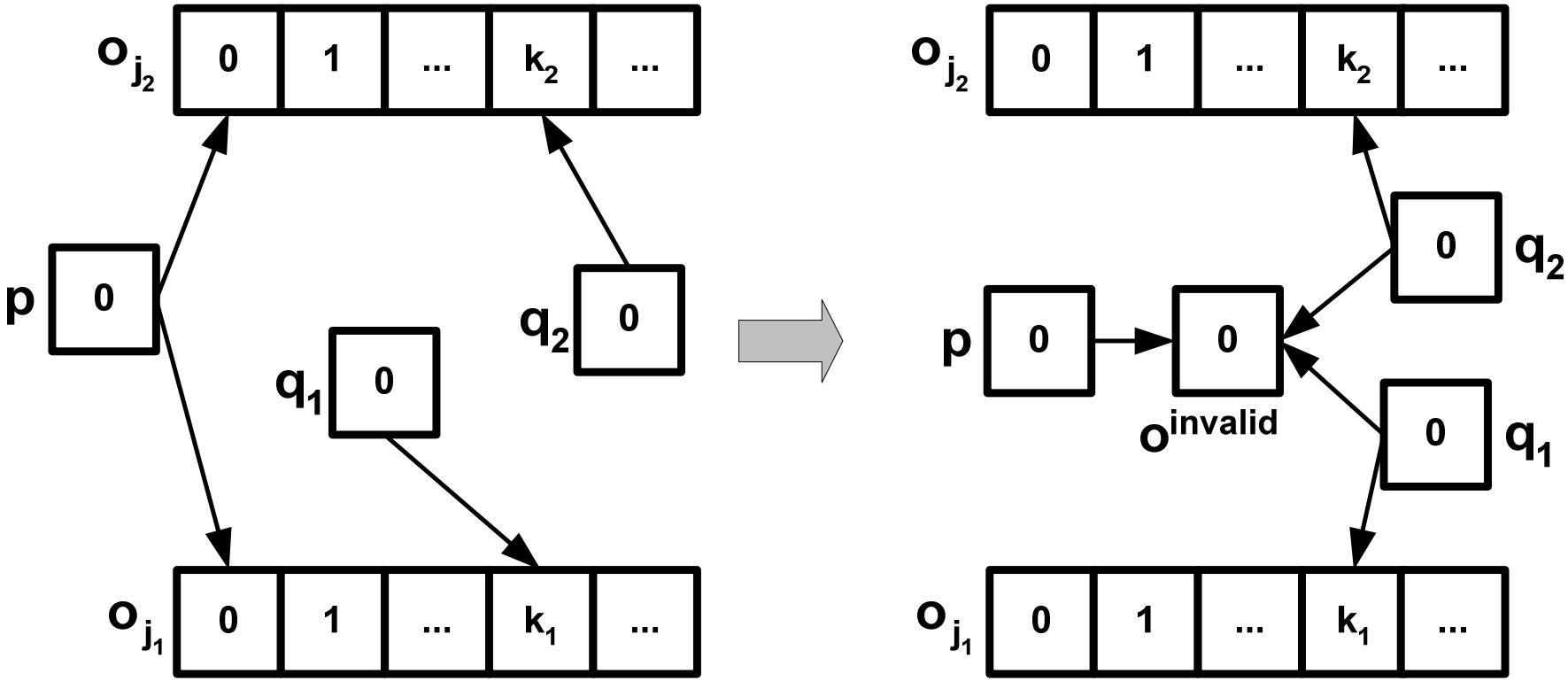
$$\langle\langle p,0\rangle, o_{invalid}\rangle \cup \bigcup_{\forall\langle\langle p,0\rangle\langle o_{j_1},k_1\rangle\rangle,\langle\langle q,k_2\rangle\langle o_{j1},k_3\rangle\rangle\in S_l} \langle\langle q,k_2\rangle, o_{invalid}\rangle$$



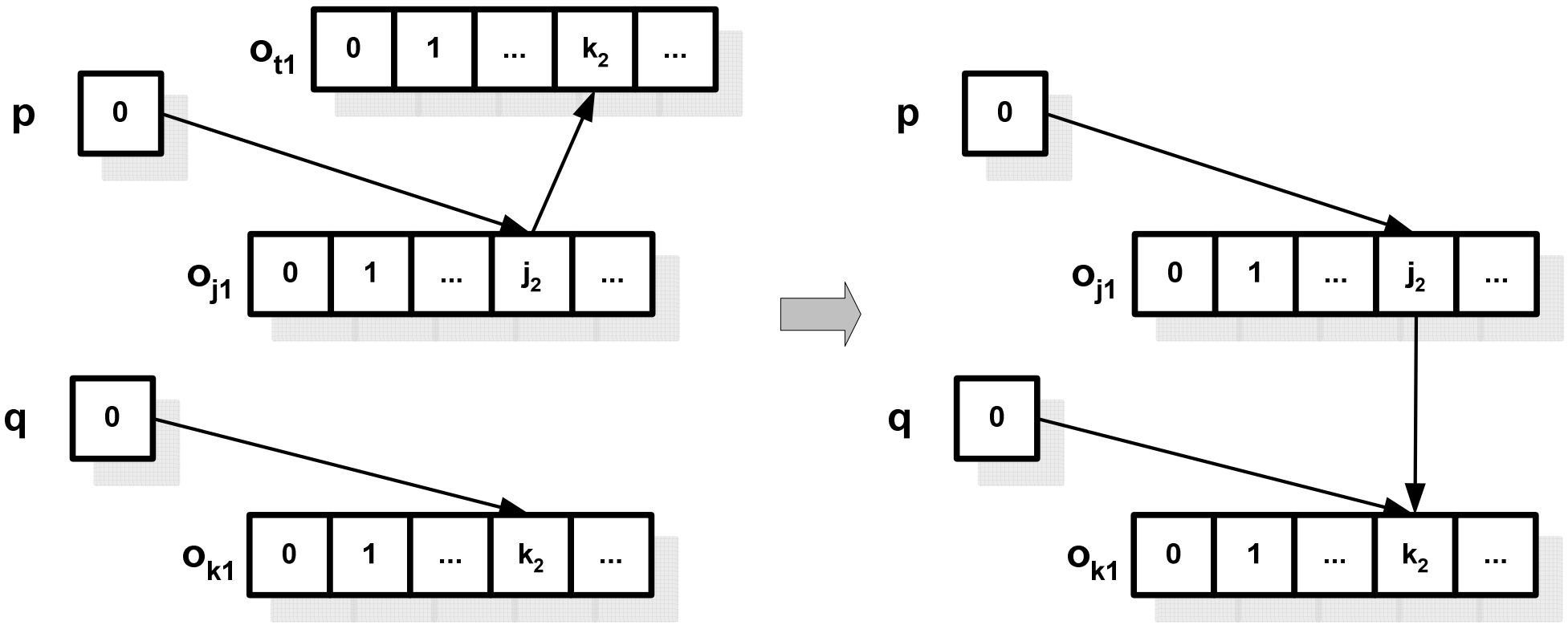
Примеры правил анализа указателей



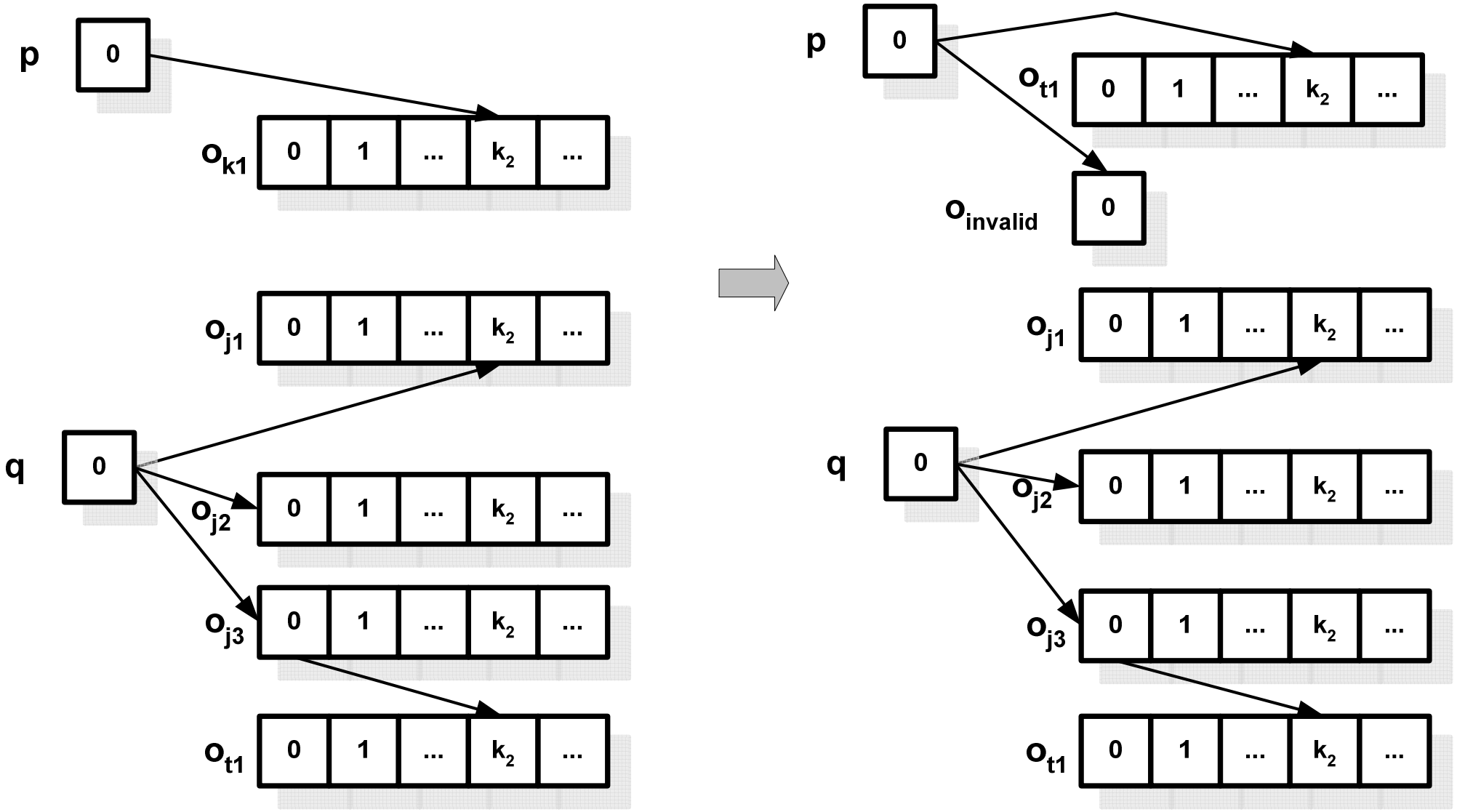
Примеры правил анализа указателей



Примеры правил анализа указателей



Примеры правил анализа указателей



Алгоритм ресурсного анализа

- Ресурсный анализ применяется для конструкций, оперирующих с ресурсами
- Примеры ресурсов: файлы, сокеты, потоки, мьютексы
- Правила ресурсного анализа определяют состояние ресурса

$$value = state,\ state = [s_1, \ldots s_n]$$

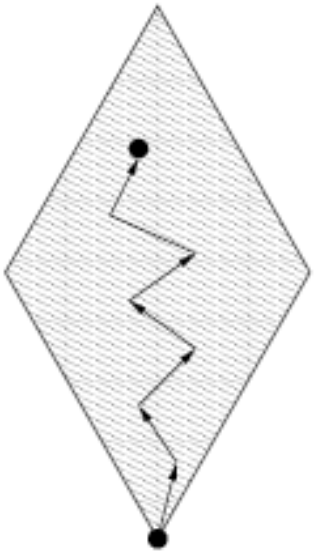
- Обнаружение нарушения протокола работы с ресурсами – нечто среднее между нефункциональными и функциональными ошибками

Решение системы уравнений

- В результате применения правил строится система уравнений
- Монотонность уравнений обеспечивается правила анализа, при каждом следующем решении уравнения результат на выходе по крайнем мере не уменьшается
- Решение системы уравнений
  - с использованием теории решеток
  - с использованием абстрактной интерпретации

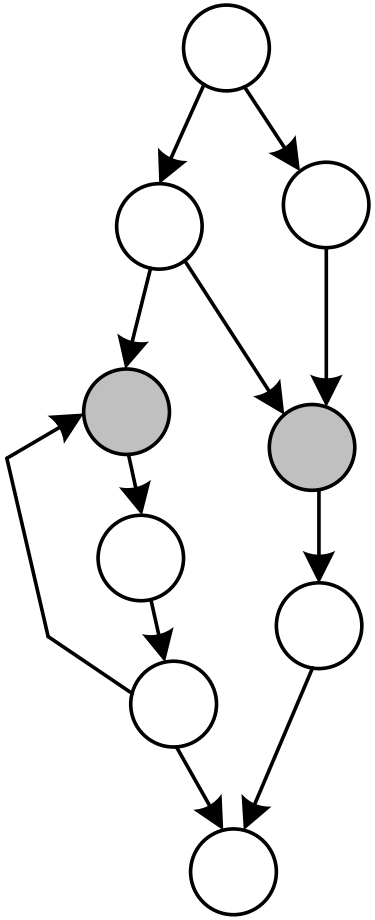
Использование теории решеток

- В узлах решетки – множества кортежей для переменных программы
- Начальное состояние ($\perp$ решетки) – соответствует пустым множествам кортежей или неинициализированным значениям переменных
- В результате решения СУ определяются множества кортежей в каждой точке программы
- Алгоритмы решения СУ
- Анализ циклов
- Техника расширения



Использование абстрактной интерпретации

- Выполняется решение уравнений в соответствии с последовательностью выполнения конструкций программы
- Последовательность решения влияет на ресурсоемкость алгоритма
- Анализ ветвлений
- Анализ циклов
- Анализ рекурсивных вызовов



Правила обнаружения дефектов

- Использование неинициализированной переменной

$$\frac{\langle a, i_{noninit} \rangle \in S_l}{[x = a]^l : defect}$$

- Повторное освобождение памяти

$$\frac{\langle \langle p,0 \rangle, \langle o_{invalid},0 \rangle \rangle \in S_l}{[\,free(p)]^l : defect}$$

- Разыменование некорректного указателя

$$\frac{\langle \langle p,0 \rangle, \langle o_{invalid},0 \rangle \rangle \in S_l}{[a = *p]^l : defect}$$

Правила обнаружения дефектов

- Выход за границы объекта

$$\frac{\langle\langle p,0\rangle,\langle o_j,k\rangle\rangle \in S_l : (k<0)\vee(k\geq o_j.size)}{[a=*p]^l : defect}$$

- Арифметические операции с указателями на разные объекты

$$\frac{\langle\langle p,0\rangle,\langle o_{j1},k_1\rangle\rangle,\langle\langle q,0\rangle,\langle o_{j2},k_2\rangle\rangle \in S_l : o_{j1}\neq o_{j2}}{[a=p-q]^l : defect}$$

- Утечка динамической памяти

Способы снижение ресурсоемкости

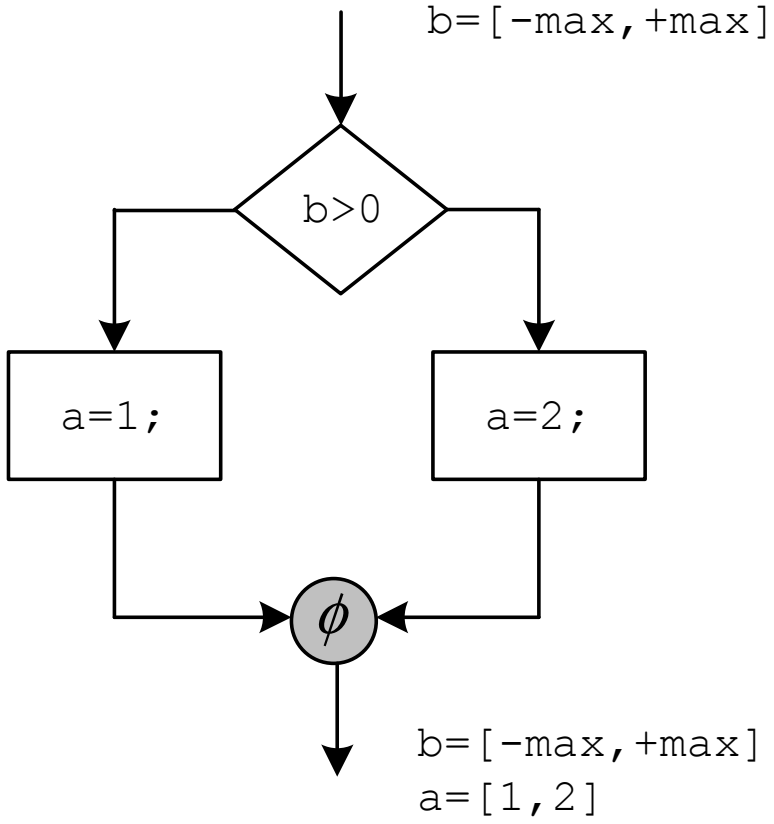
- СА при высокой точности и полноте имеет высокую ресурсоёмкость
  - большое число анализируемых путей
  - большая длина путей
  - большой объем данных, связанных с каждым путём

Способы снижение ресурсоемкости

- Сокращение числа путей
  - объединение состояний объектов программы в ϕ-функциях и далее проведение общего анализа
- Сокращение длины путей
  - ограничение числа итераций циклов
  - ограничение глубины стека вызова функций
  - замена некоторых функций их аппроксимациями
- Уменьшение количества данных
  - ограничение размера выделяемых и анализируемых объектов
  - компактное представления информации

Способы повышения точности

- Частичное неслияние путей, необходимо определить эффективный критерий неслияния
  - ограничение на число отдельных путей
  - время жизни пути
- Анализ зависимостей между значениями переменных

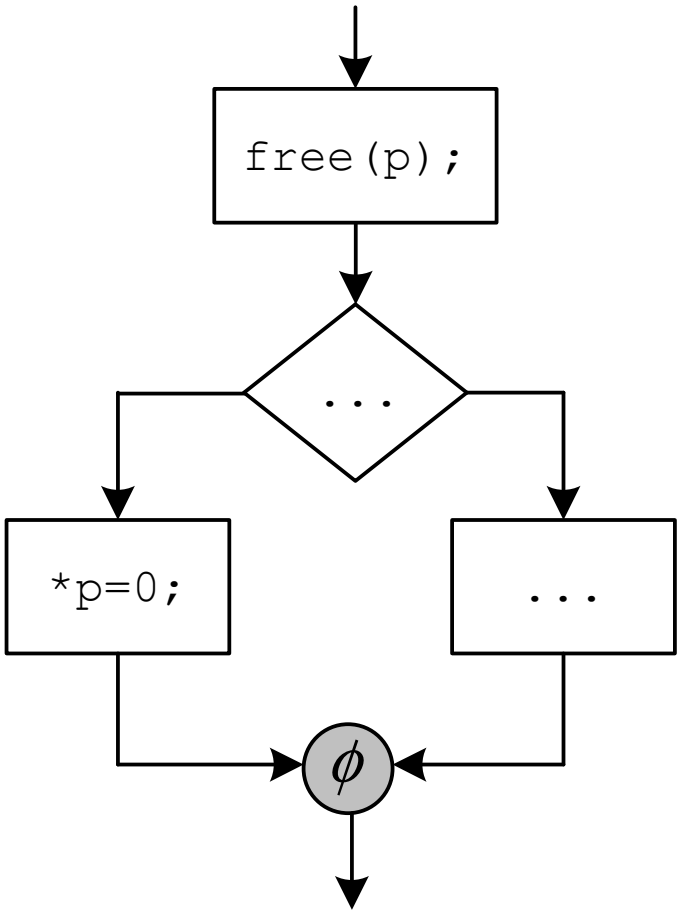


Пример работы зависимостей

```
int fd;
int f() {
    if (...) {
        return -1;
    fd = fopen(fname, “rw”);
    return 0;
}
int main() {
    ...
    int result = f();    // fd == file, result == 0
    if (result < 0) {
         exit;
    }
    fprintf(fd, ...);   // fd =file
```

Обнаружение множественных дефектов

- Программа может содержать несколько дефектов
- Возникновение наведенных дефектов
- Необходимо устранить влияние обнаруженных дефектов – в общем случае невозможно
- Удаление путей, на которых обнаружен дефект



Анализ программ с неполными исходными кодами

- Библиотечные функции
- **Аннотации** – упрощенных описания поведения отсутствующих функций, результаты работы функции
  - возвращаемое значение
  - переменные, изменяемые через указатели
  - изменяемые глобальные переменных
- Применение аннотация для определения значений входных данных, значений внутренних переменных и т.п.

Эффективность статического анализа

- **Эффективность СА** – комплексное свойство, отражающее качество получаемых результатов, степень автоматизации процесса анализа и сложность его организации, ресурсоемкость, применимость для программ различного размера и класса
- Показатели эффективности:
  - полнота результатов
  - точность результатов
  - ресурсоемкость
  - возможность автоматизации

Показатели эффективности

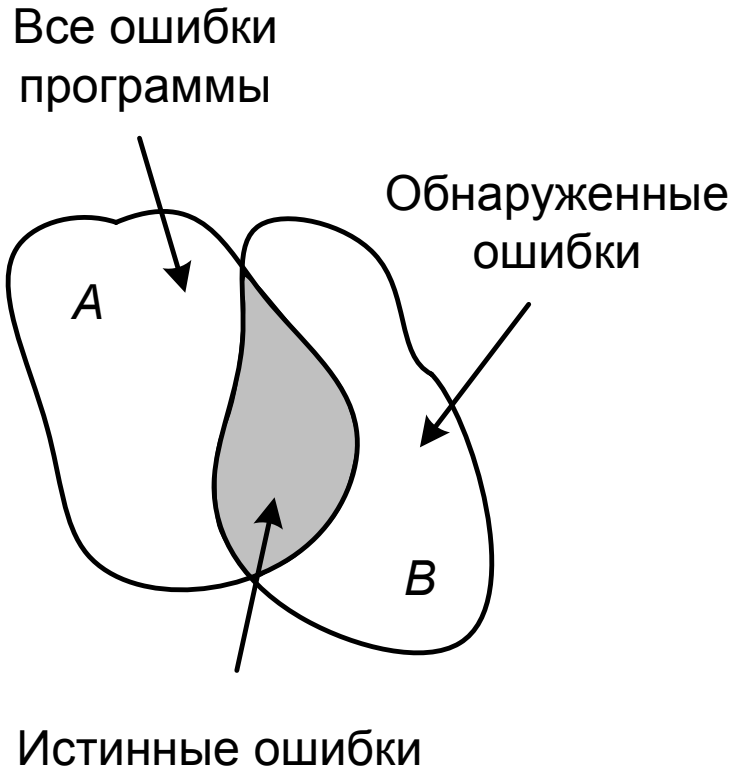
- **Полнота** – доля истинных ошибок среди всех ошибок в программе

$$\frac{|A \cap B|}{|A|}$$

- **Точность** – доля истинных ошибок среди всех обнаруженных ошибок

$$\frac{|A \cap B|}{|B|}$$

- **Ресурсоемкость**
- **Степень автоматизации**



Эффективность статического анализа

- Эффективность СА зависит от
  - вида извлекаемой информации
  - свойств языка программирования
  - особенностей алгоритма программы и стиля кодирования
  - степени влияния окружения и внешних воздействий
  - используемых алгоритмов анализа

Извлекаемая информация

- Определяется задачей статического анализа
  - типами обнаруживаемых ошибок
- Для обнаружения нефункциональных ошибок в программах на языке C необходимо определять
  - значения интервальных переменных
  - объекты, на которые указывают указатели
  - функции, на которые указывают указатели
  - состояния ресурсов
- Извлекаемая информация для обнаружения функциональных ошибок определяется проверяемыми спецификациями

Свойства языка программирования

- Наиболее сложные конструкции языка C
  - арифметика указателей
  - сложные типы данных
  - указатели на функции
  - приведение типов
  - рекурсивные функции
- В языке C++ дополнительно
  - анализ исключений
  - полиморфизм и виртуальные функции
  - шаблоны

Особенности алгоритма и стиля кодирования

- Сложность алгоритма и способ его реализации
- Обычно используется подмножество языка программирования
- Архитектура программы
- Стиль кодирования

Входные данные и внешние воздействия

- Виды внешних воздействий
  - входные параметры программы
  - данные, полученные из потока ввода и прочитанные из файлов
  - команды пользователя
  - внешнее окружение программы
- Информация о внешних воздействиях отсутствует в исходном коде программы
- Чем больше влияние входных данных информации, тем ниже эффективность статического анализа

Свойства алгоритмов СА

- Любой алгоритм статического анализа использует те или иные аппроксимации
  - для снижения ресурсоемкости
  - при этом снижается полнота/точность получаемых результатов
- Примеры используемых аппроксимаций
  - совместный анализ путей выполнения программы
  - ограничение на глубину стека вызовов
  - ограничение на число итераций при анализе циклов
  - ограничение на размер анализируемых объектов

Средства статического анализа

- Средства для обнаружения широкого класса дефектов
  - Coverity Prevent SA
  - Klockwork Insight
  - Mathworks PolySpace
  - IBM RSA
  - Fortify SCA
  - **Microsoft CA**
  - Frama-C
  - SPlint
- Отечественные разработки
  - SVaCE Detector
  - Viva64, VivaMP
  - **AEGIS**

Открытый сервис www.digiteklabs.ru/aegis

Результаты анализа

Длительность анализа: 6 seconds and 882 milliseconds

Путь	Код	Дефекты, контексты
	01: // Several for-cycles with function call	
	02:	
	03: #include <stdio.h>	
	04:	
	05: int arr[2];	
	06:	
	07: int f1(float par) {	
	08: int tmp = par;	
	09: printf("%d \n",tmp);	
*(3)	10: return arr[tmp]=tmp; // BUF-02	Разыменование указателя int * arr, tmp выведенного за границу объекта (BUF-02)