Анализ потоков данных

Data flow analysis

Синявин А.В.

Бит-векторный анализ

Достигающие определения (reaching definition = RD)

Def: пусть дан некоторый CFG граф

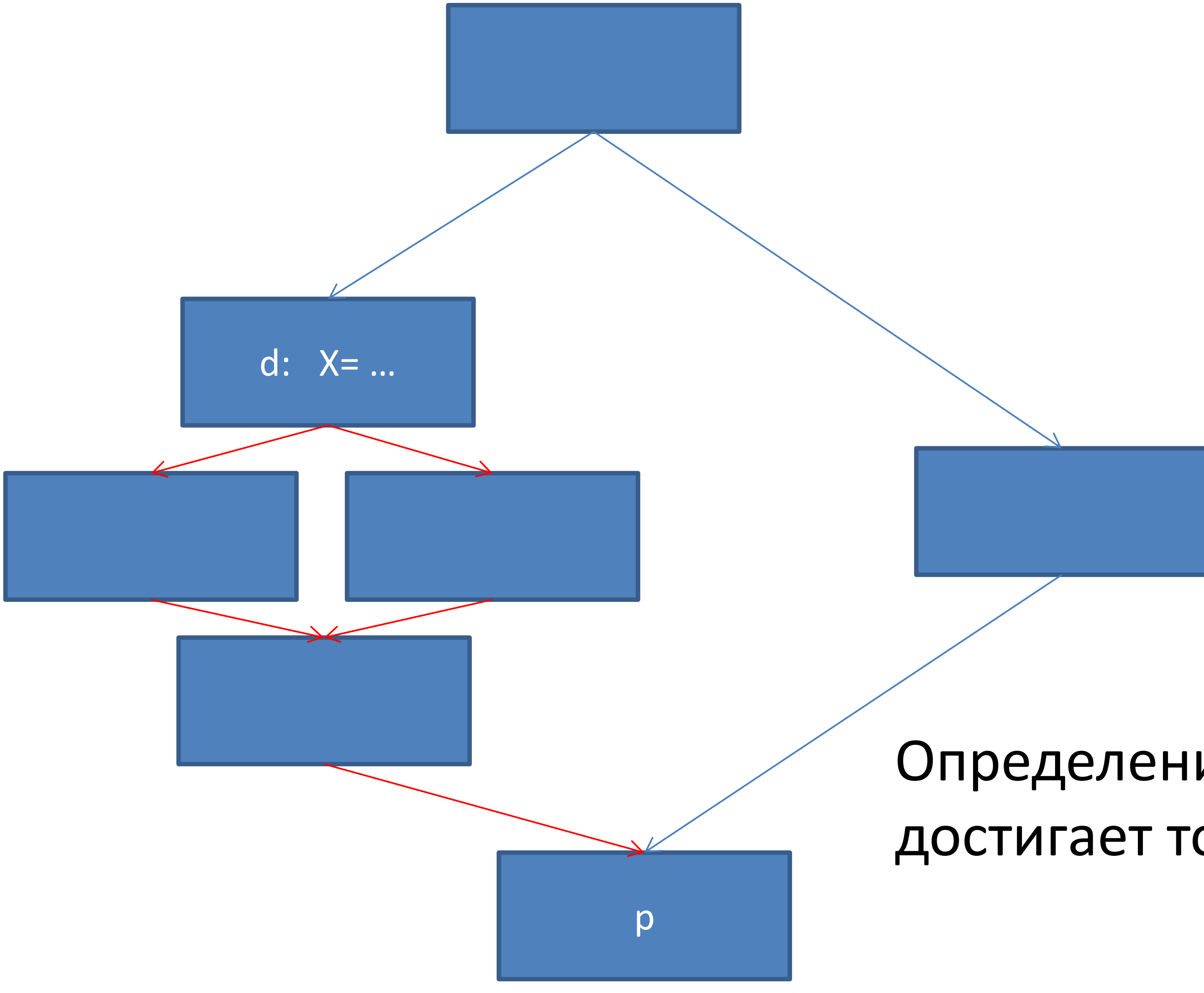
x – некоторая переменная

p – некоторая точка в CFG

- начало некоторого ББ
- конец некоторого ББ
- место между инструкциями

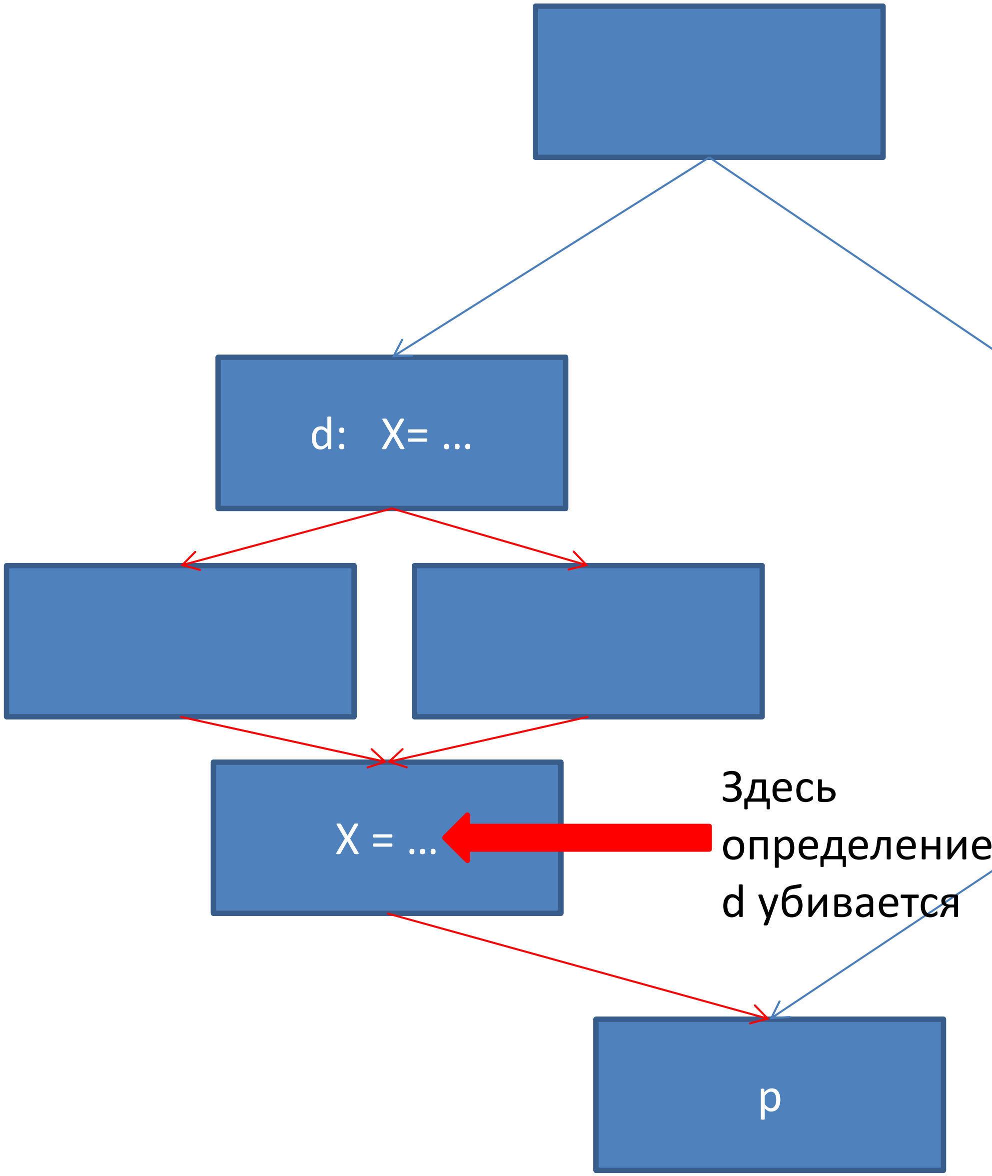
Определение (инструкция) d некоторой переменной x достигает точки p, если ∃ путь от d до p | x не переопределяется вдоль этого пути.

Достигающие определения



Определение d достигает точки p

Достигающие определения

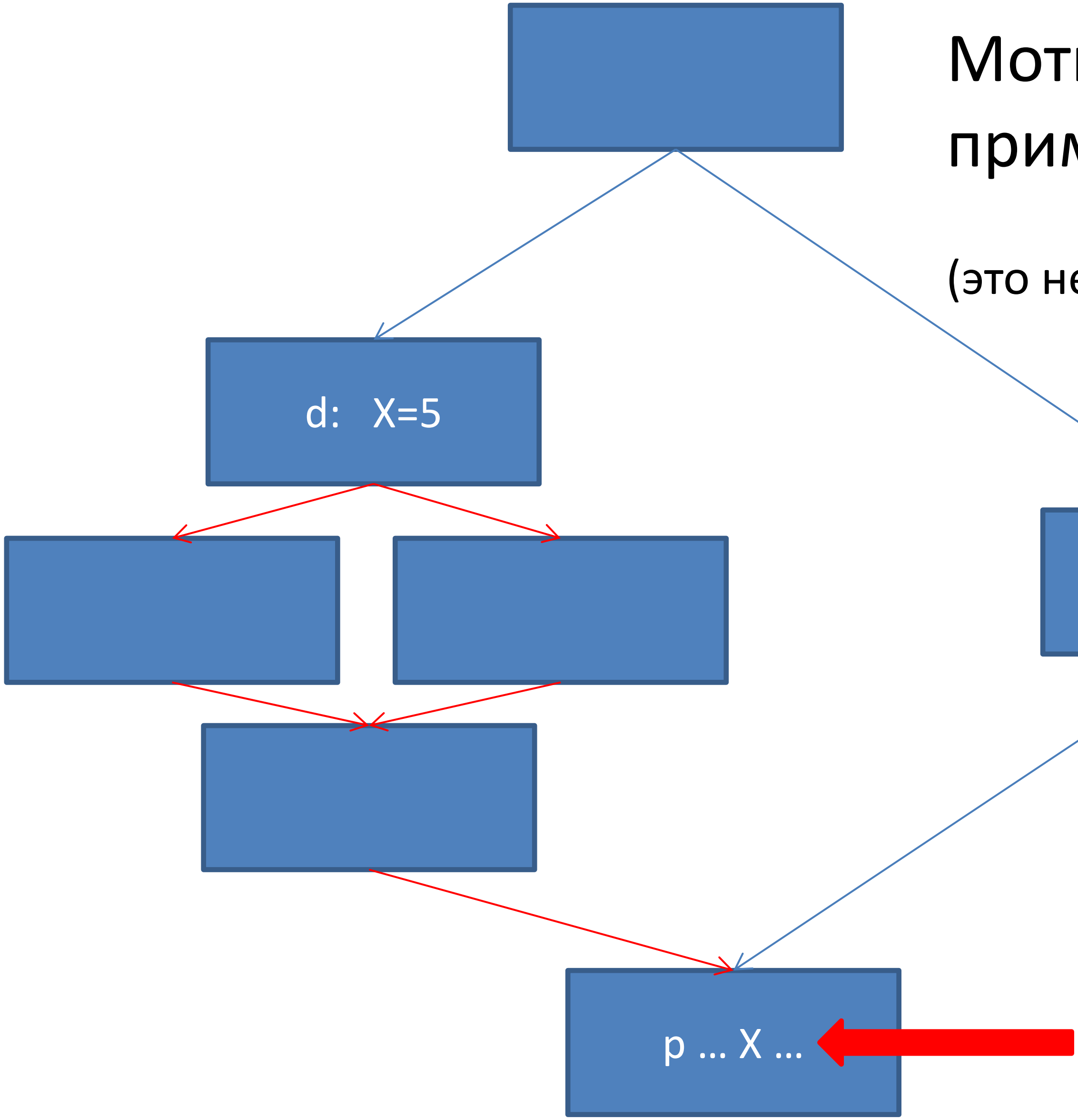


Определение d **не** достигает точки p

Достигающие определения

Мотивация: пример применения

(это не единственный пример)



Если d – единственное присваивание x, которое достигает p, то

X можно заменить на 5

Анализ потоков данных (data-flow analysis = DFA)

Задача вычисления RD – частный случай DFA задачи

Схема анализа потока данных

Задача определения факта возможности выполнения для каждого пути в общем виде неразрешима.

=>

Будем полагать, что каждый путь может выполнятся.

Схема анализа потока данных

- С каждой точкой в CFG мы связываем значение потока данных (data-flow value=DFV)

  Пусть s – инструкция

    IN[s] – DFV до инструкции

    OUT[s] – DFV после инструкции

- Множество значений потока данных – область определения

Схема анализа потока данных

Ограничения на DFV

- семантика инструкций
- основанные на потоке управления

Схема анализа потока данных

Учёт семантики инструкций

передаточные функции для s

прямое направление

OUT[s] = f_s_forward(IN[s])

обратное направление

IN[s] = f_s_backward(OUT[s])

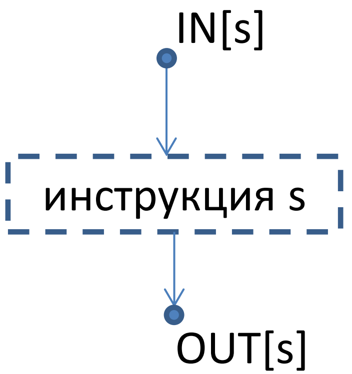


Схема анализа потока данных

Учёт потока управления

в базовом блоке поток управления простой

$$\mathrm{IN}[s_{i+1}]=\mathrm{OUT}[s_i] \text{ для } \forall\ i=1,2,\ldots,n-1$$

Схема анализа потока данных

Поток управления проходит по ББ без ветвлений

=>

Можно рассматривать значения потока при входе в ББ и при выходе из ББ

Схема анализа потока данных

IN[B] – DFV перед базовым блоком B

OUT[B] – DFV после базового блока B

Пусть B состоит из s1, s2, … , sn

=>

IN[B] = IN[s1]

OUT[B] = OUT[sn]

Передаточная функция для B

f_B_forward(IN[B]) =

f_sn_forwrad(… f_s2_forwrad(f_s1_forward(IN[s1])) …)) =

f_sn_forwrad ° … ° f_s1_forwrad(IN[B])

Схема анализа потока данных

Учёт потока управления

в более общем случае

Пусть DFV – информация о множестве констант, которые могут быть присвоены переменным

=>

уравнение потока управление

IN[B] =$\cup_{P-\text{предшественник}} OUT[P]$

Консерватизм DFA

- Все алгоритмы DFA дают приближённые решения
- Решение безопасно (консервативно), если программа не меняет результаты вычисления
- Конкретный DFA должен быть консервативным

Вычисление RD (потоковая функция)

d: u = v op w

op – некоторая операция

Инструкция

- генерирует определение d переменной u
- убивает все другие определения переменной u

f_d_forward(x) = gen_d ∪ (x – kill_d)

Схема анализа потока данных

Итого

прямая задача

OUT[B] = f_B_forward(IN[B])

IN[B] =$\wedge_{P-\text{предшественник}} OUT[P]$

обратная задача

IN[B] = f_B_backward(OUT[B])

OUT[B] =$\wedge_{S-\text{преемник}} IN[S]$

$\wedge$ - оператор сбора (meet operator)

Вычисление RD (потоковая функция)

Пусть базовый блок содержит две инструкции

$f1(x) = gen1 \cup (x - kill1)$ для первой инструкции

$f2(x) = gen2 \cup (x - kill2)$ для второй инструкции

$f2(f1(x)) = (gen2 \cup (gen1 - kill2)) \cup (x - (kill1 \cup kill2))$

Вычисление RD (потоковая функция)

Пусть базовый блок содержит N инструкций

$f1(x) = gen1 \cup (x - kill1)$ для первой инструкции

$f2(x) = gen2 \cup (x - kill2)$ для второй инструкции

…

$fN(x) = genN \cup (x - killN)$ для N-ой инструкции

$fB(x) = genB \cup (x - killB)$, где

$killB = kill1 \cup kill2 \cup kill3 \cup \ldots \cup killN$

$genB = genN \cup$
$(gen[N-1] - killN) \cup$
$(gen[N-2] - kill[N-1] - kill[N]) \cup$
$\ldots$
$(gen1 - kill2 - kill3 - \ldots - killN)$

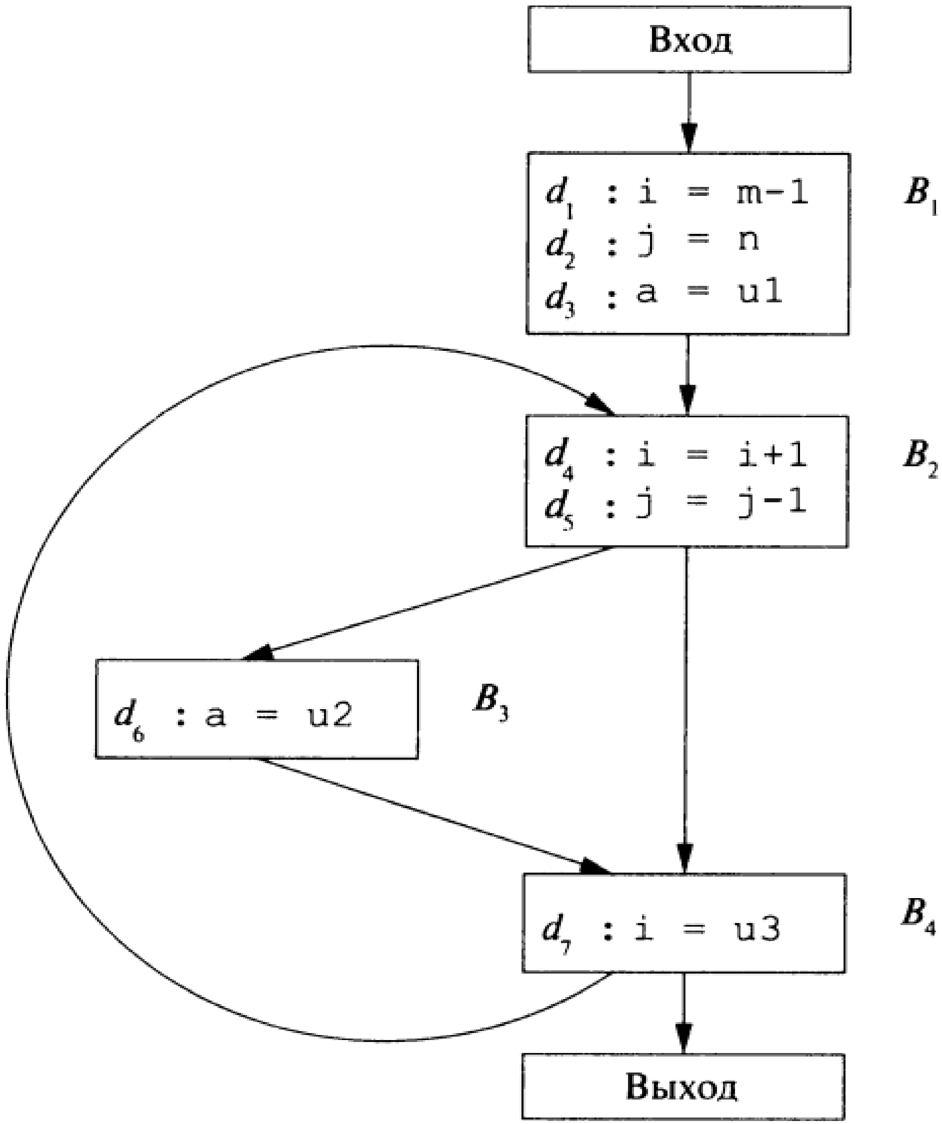
Вычисление RD (потоковая функция)

- genB содержит все определения, которые видны сразу после блока

- killB – простое объединение kill для всех инструкций

$$fB(x) = genB \cup (x - killB)$$

Достигающие определения



$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

Вычисление RD (уравнение потока управления)

- Определение достигает точки p, если ∃ путь, вдоль которого точка p может быть достигнута

OUT[P] ⊆ IN[B], P – ∀ предшественник B

- Определение не может достигнуть точки p, если ∄ пути до p

Т.о.,

$$\mathrm{IN[B]} = \bigcup_{P-\text{предшественник}\,B} OUT[P]$$

Вычисление RD (система уравнений)

Введём два пустых ББ в CFG. Входной и выходной.

OUT[вход] = $\emptyset$

OUT[B] = genB ∪ (IN[B] - killB)

IN[B] = $\bigcup_{P-\text{предшественник } B} OUT[P]$

Вычисление RD
(решение системы уравнений)

Алгоритм решение системы уравнений для RD

Вход: CFG

Выход: DFV IN[B], OUT[B] для всех ББ

Метод:

```
"OUT[Вход] = ∅;";
for ( "каждый ББ B" ) "OUT[B] = ∅;";
while ("внесены изменения в OUT")
{
    for ( "каждый ББ B" )
    {
        "IN[B] = ⋃_{P−предшественник B} OUT[P]";
        "OUT[B] = genB ∪ (IN[B] - killB)";
    }
}
```

Консерватизм решения задачи поиска RD

Реальность:

мн-во достигающих определений в p
{ d1, d2 }

Решение:

мн-во достигающих определений в p
{ d1, d2, d3 }

безопасно в плане замены на константу

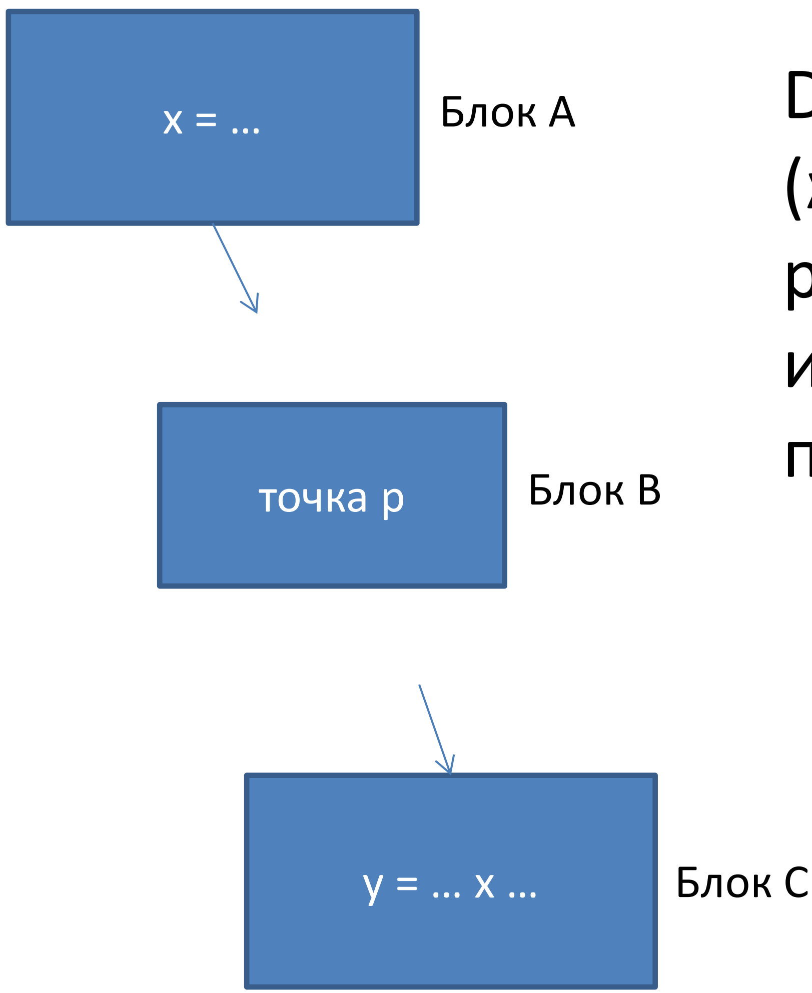
Решение:

мн-во достигающих определений в p
{ d1 }

опасно в плане замены на константу

Наш алгоритм в этом смысле безопасен

Анализ активных переменных live-variable (LV) analysis



Def: Переменная X активна (жива), если ∃ путь в CFG от p до некоторой точки, где используется значение переменной X

Def: В противном случае переменная X мертва

Анализ активных переменных

Мотивация:

Понятие LV важно для распределения регистров.

Пусть x1 и x2 активны в p одновременно, тогда x1 и x2 не могут быть размещены на одном аппаратном регистре.

Анализ активных переменных

IN[B] и OUT[B] – мн-ва LV,
где B – базовый блок

defB – мн-ва переменных, определённых в B до любых их использований в B (если есть такие использования)

useB – мн-ва переменных, значения которых могут использоваться до любых определений в B (если есть такие определения)

Анализ активных переменных (система уравнений)

LV при выходе из программы нет

IN[выход] = $\emptyset$

X активен при входе в B

- X используется до возможного переопределения в B

- X активен на выходе B и не переопределён в B

IN[B] = useB $\cup$ (OUT[B] - defB)

X активен при выходе из B <=> X активен при входе в один из преемников

OUT[B] = $\cup_{S-\text{преемник } B} IN[S]$

Анализ активных переменных (решение системы уравнений)

Алгоритм решение системы уравнений для LV

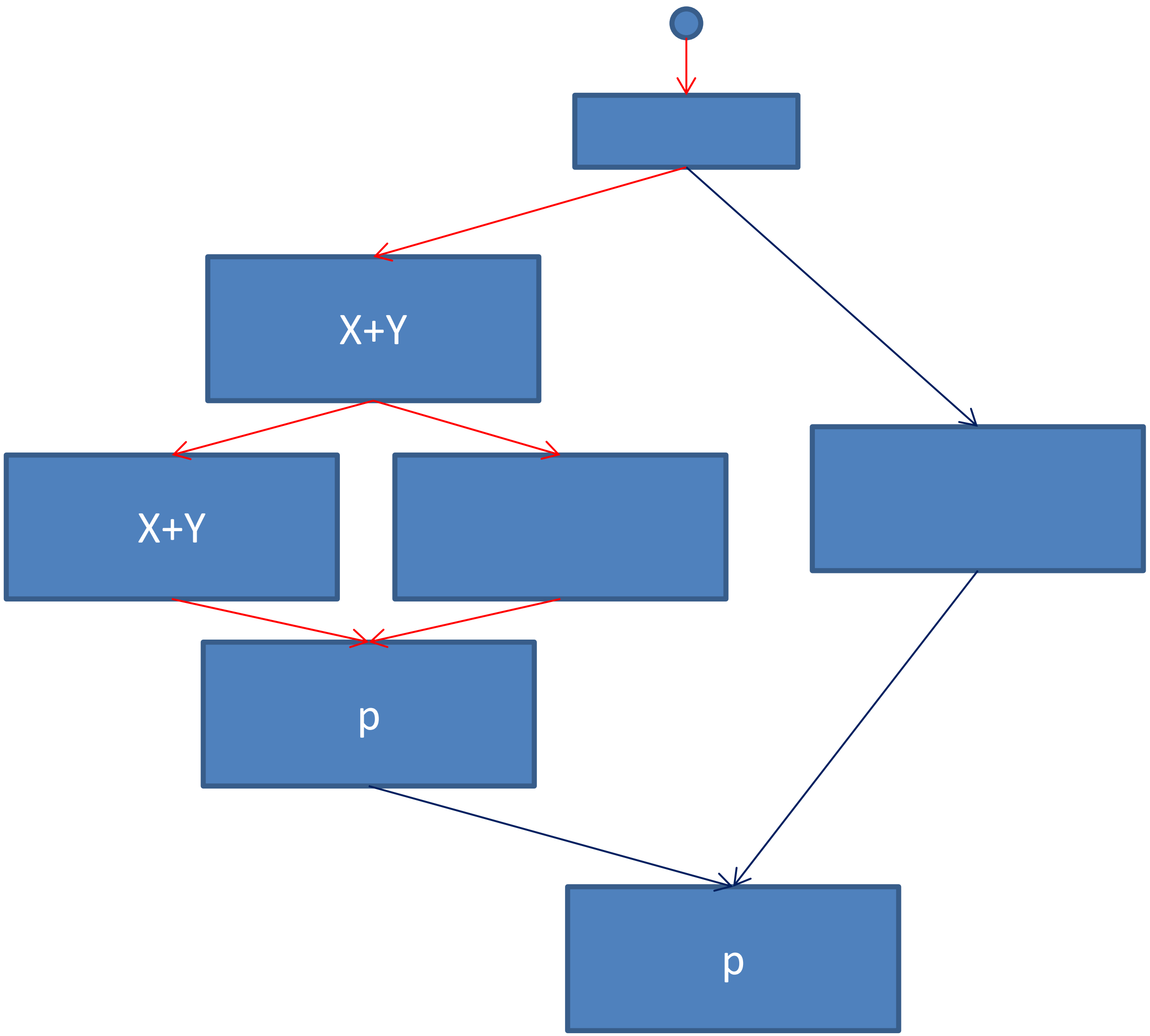
Вход: CFG

Выход: DFV IN[B], OUT[B] для всех ББ

Метод:

```
"IN[Выход] = ∅;";
for ( "каждый ББ B" ) "IN[B] = ∅;";
while ("внесены изменения в IN")
{
     for ( "каждый ББ B" )
     {
          "OUT[B] = ∪_{S−преемник B} IN[S]";
          "IN[B] = useB ∪ (OUT[B] - defB)";
     }
}
```

Анализ доступных выражений (available expression =AE analysis)



Def: выражение x+y доступно в точке p, если ∀ путь от входного ББ до p вычисляет x+y и после последнего вычисления до достижения p нет присваиваний X и Y

Анализ доступных выражений

Мотивация:

Понятие AE удобно при поиске общих подвыражений

Анализ доступных выражений

U – мно-во всех выражений

IN[B] – мно-во AE на входе в B
OUT[B] – мно-во AE на выходе в B

IN[B], OUT[B] $\subseteq$ U

e_kill[B] – мн-во AE, убиваемых (х.б. один аргумент переопределяется) в B

e_gen[B] – мн-во AE, генерируемых (выражение вычисляется и далее в B не убиваются) B

Анализ доступных выражений (система уравнений)

На входе AE нет

OUT[вход] = ∅

AE на выходе из B есть, если

- генерируется в B

- AE есть на входе в B и не убивается в B

OUT[B]=e_gen[B] ∪ (IN[B] – e_kill[B])

AE есть на входе в B, если AE есть в конце всех предшественников

IN[B] = $\cap_{P-\text{предшественник}} OUT[P]$

Анализ доступных выражений (решение системы уравнений)

Алгоритм: решение системы уравнений для AE
Вход: CFG
Выход: DFV IN[B], OUT[B] для всех ББ
Метод:

```
“OUT[Вход] = ∅;”;
for ( “каждый ББ B, отличный от входного” ) “OUT[B] = U”;
while (“внесены изменения в OUT”)
{
    for ( “каждый ББ B , отличный от входного” )
    {
        “IN[B] = ∩_{P−предшественник} OUT[P]”;
        “OUT[B]=e_gen[B] ∪ (IN[B] – e_kill[B])”;
    }
}
```

Формализация DFA

Def: Полурешётка –

множество $(V, \wedge) \mid \forall\ x, y, z \in V$

- $x \wedge x = x$ (идемпотентность)
- $x \wedge y = y \wedge x$ (коммутативность)
- $x \wedge (y \wedge z) = (x \wedge y) \wedge z$

$\wedge$ - оператор сбора (meet)

Формализация DFA

Def: $(V, \wedge)$ – полурешётка. Верхний элемент T | $\forall x \in V$

$$T \wedge x = x$$

Def: $(V, \wedge)$ – полурешётка. Нижний элемент $\perp$ | $\forall x \in V$

$$\perp \wedge x = \perp$$

Частичный порядок

Def: Мн-во V – частично упорядоченное мн-во, если $\exists$ отношение $\leq$ |

$\forall$ x, y, z $\in$ V, верно

- $x \leq x$ (рефлексивность)
- Если $x \leq y$ и $y \leq x$, то $x = y$ (антисимметричность)
- Если $x \leq y$ и $y \leq z$, то $x \leq z$ (транзитивность)

Частичный порядок в полурешётке

В полурешётке можно определить частичный порядок

$$x \leq y \iff x \wedge y = x$$

Частичный порядок в полурешётке

Области определения , которые были в DFA (RD, LV, AE) , - полурешётки

Операторы ∩, ∪ - операторы сбора

$$\text{IN[B]} = \bigcup_{P-\text{предшественник } B} OUT[P]$$

$$\text{IN[B]} = \bigcap_{P-\text{предшественник } B} OUT[P]$$

Наибольшая нижняя граница

Def: Наибольшая нижняя граница
(greatest lower bound - GLB)

Пусть $(V, \wedge)$ – полурешётка
g – GLB для $x, y \in V$, если

- $g \leq x$
- $g \leq y$
- $\forall z \in V \mid z \leq x$ и $z \leq y \Longrightarrow z \leq g$

Наибольшая нижняя граница

Легко доказать, что

$\forall$ x, y $\in$ V (полурешётка)

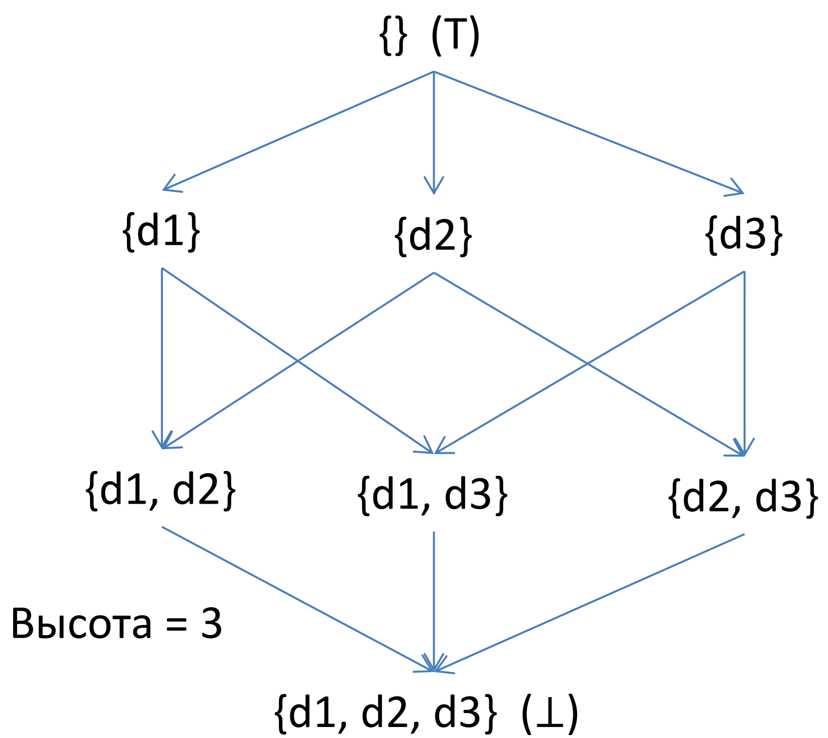
x $\wedge$ y является GLB для элементов x, y

Диаграммы решёток

Пример для RD

Пусть даны определения d1, d2, d3

$$x \le y \Leftrightarrow x \supseteq y$$



Def: восходящая цепочка – последовательность вида

$x1 < x2 < x3 < \ldots < xn$

Def: высота полурешётки - наибольшее кол-во отношений < в восходящей цепочке

Обобщённый алгоритм (прямая версия)

Алгоритм: решение прямой задачи DFA
Вход: CFG
Выход: DFV IN[B], OUT[B] для всех ББ
Метод:

```
“OUT[Вход] = v0”;
for ( “каждый ББ B, отличный от входного” ) “OUT[B] = T”;
while (“внесены изменения в OUT”)
{
    for ( “каждый ББ B , отличный от входного” )
    {
        “IN[B] = ∧_{P−предшественник} OUT[P]”;
        “OUT[B]= f_B(IN[B])”;
    }
}
```

Обобщённый алгоритм (обратная версия)

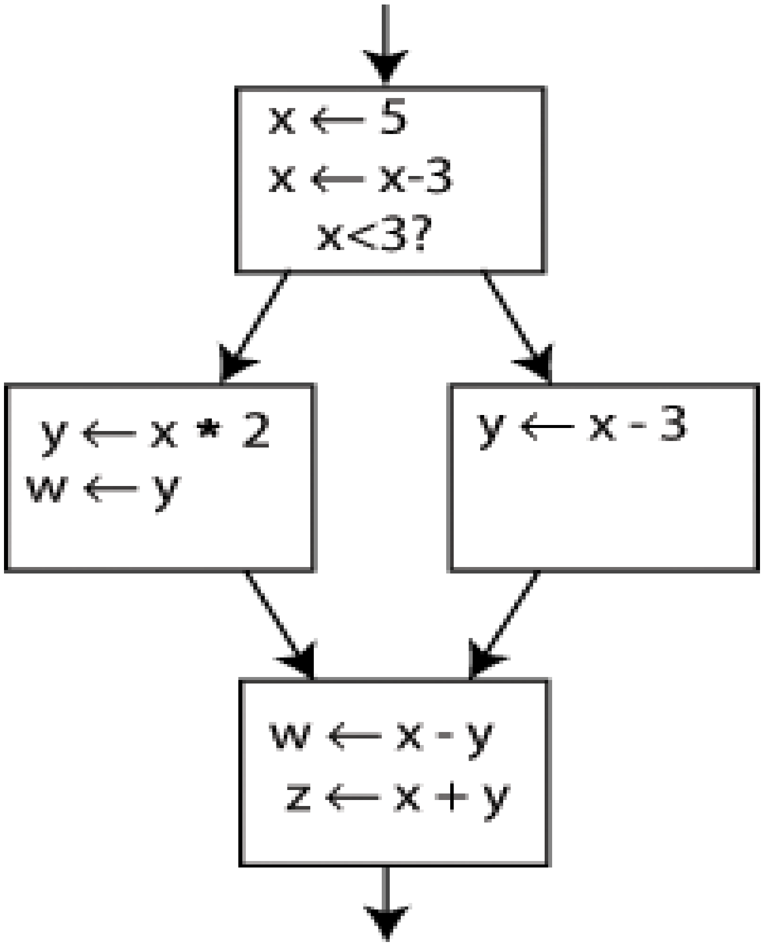
Алгоритм: решение обратной задачи DFA
Вход: CFG
Выход: DFV IN[B], OUT[B] для всех ББ
Метод:

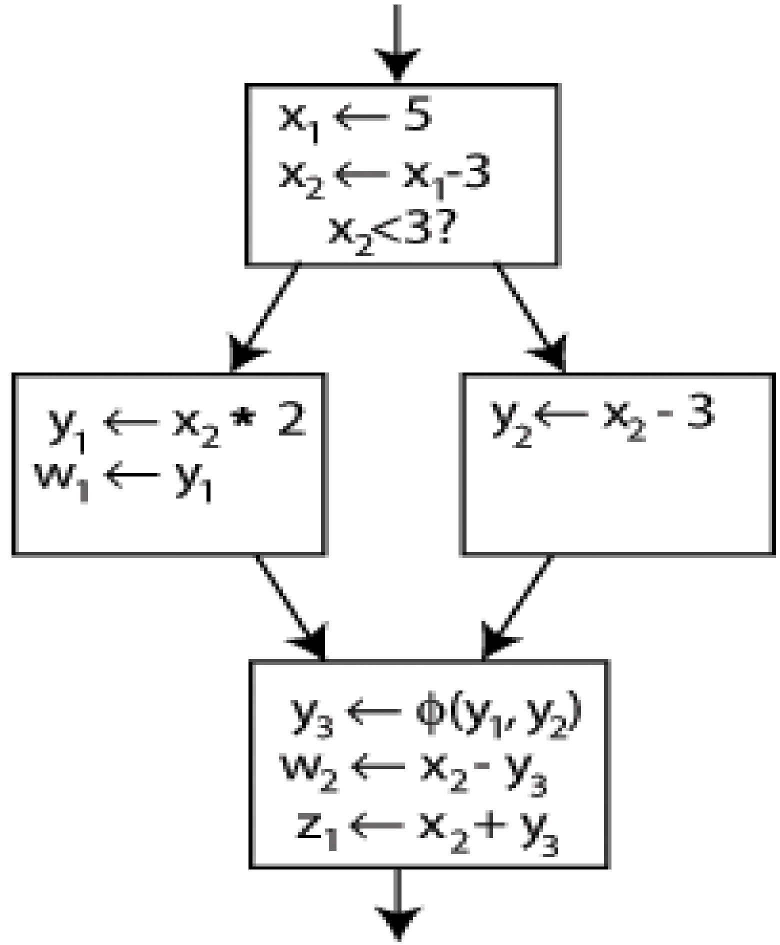
```
“IN[Выход] = v0”;
for ( “каждый ББ B, отличный от выходного” ) “IN[B] = T”;
while (“внесены изменения в IN”)
{
    for ( “каждый ББ B , отличный от выходного” )
    {
        “OUT[B] = ∧_{S−преемник} IN[S]”;
        “IN[B]= f_B(OUT[B])”;
    }
}
```

Static Single Assignment

Понятие и построение SSA

Понятие SSA





Каждая переменная имеет только одно присваивание

Зачем?

В терминах SSA описываются многие оптимизирующие алгоритмы

Алгоритм построения

Делится на две части:

- размещение ф-функций
- переименование переменных (определение версий переменных)

Алгоритм построения (размещение ф-функций)

Def: Соединение некоторого подмножества $S \subseteq V$ множества вершин

$$J(S) = \{x | \exists p_1, p_2 \in S, \quad p_1 \neq p_2, \qquad x \in V$$
$$\exists \text{ непустые непересекающиеся пути из } p_1 \text{ в } x \quad \text{и} \quad \text{из } p_2 \text{ в } x\}$$

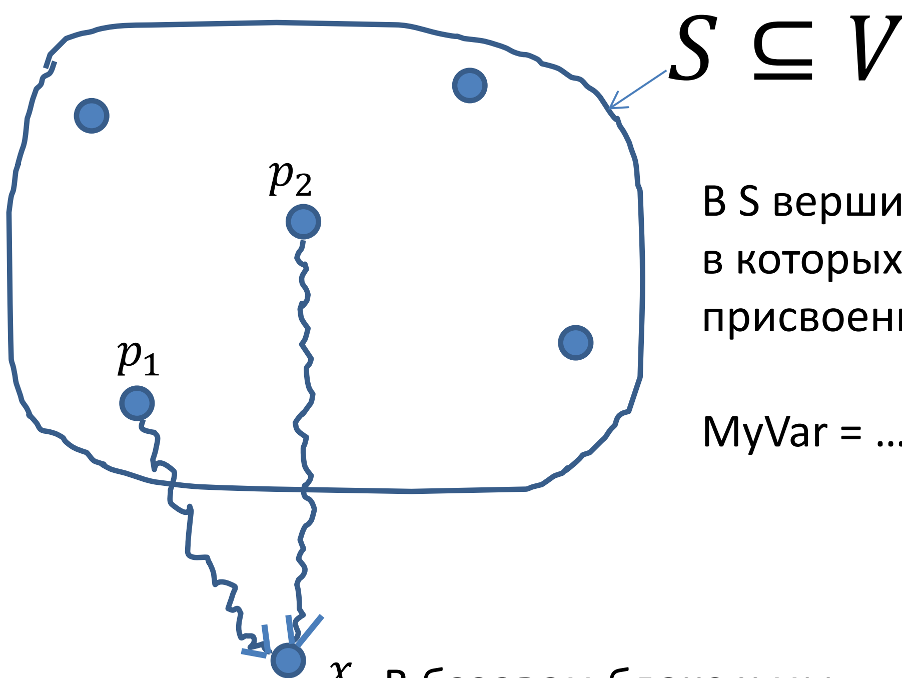
Def: Итерированное соединение $J^+(S)$ некоторого подмножества $S \subseteq V$ множества вершин

$$J_1 = J(S)$$
$$J_2 = J(S \cup J_1)$$
$$J_3 = J(S \cup J_2)$$
$$\dots$$
$$J_{i+1} = J(S \cup J_i)$$

Процесс делается до тех пор, пока Jk не перестанет меняться

Алгоритм построения (размещение ф-функций)

$J(S)$



В S вершины (ББ) CFG,
в которых есть
присвоение переменной

MyVar = …

Волнистые линии –
непустые
непересекающиеся
пути

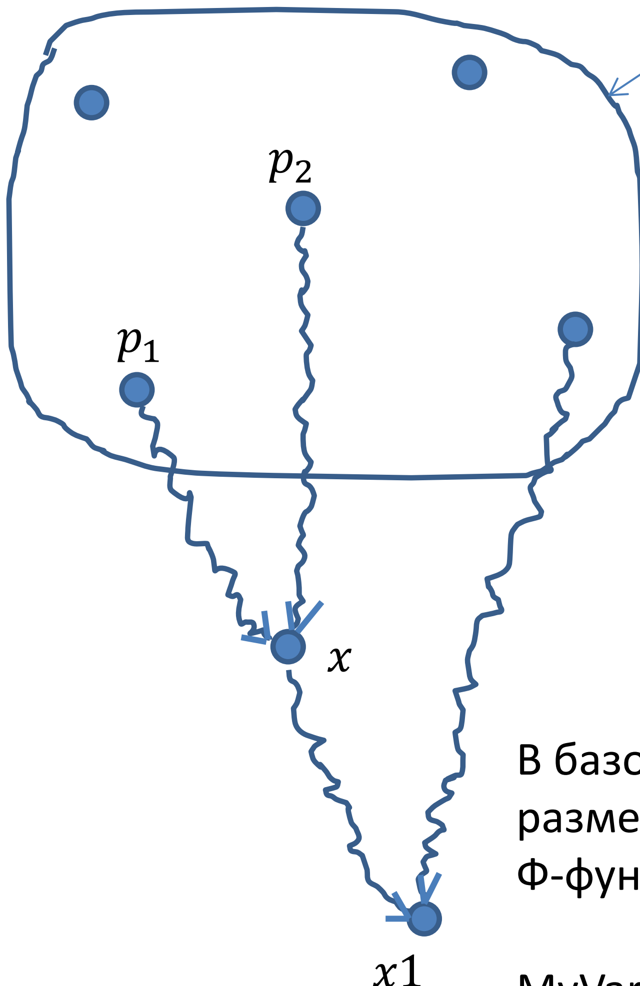
В базовом блоке x мы
размещаем Ф-функцию

MyVar = Ф(…)

Алгоритм построения (размещение ф-функций)

$J^{+}(S)$

$S \subseteq V$



В S вершины (ББ) CFG, в которых есть присвоение переменной

MyVar = …

Волнистые линии – непустые непересекающиеся пути

В базовом блоке x1 мы размещаем **ещё одну** Ф-функцию

MyVar = Ф(…)

Алгоритм построения (размещение ф-функций)

S –вершины, в которых делается присвоение переменной MyVar

$J^+(S)$- где следует расположить ф-функции для переменной MyVar

Алгоритм построения (размещение ф-функций)

Непосредственное вычисление $J^{+}(S)$ - слишком дорого

Оказывается $J^{+}(S)$ можно вычислить через доминаторы

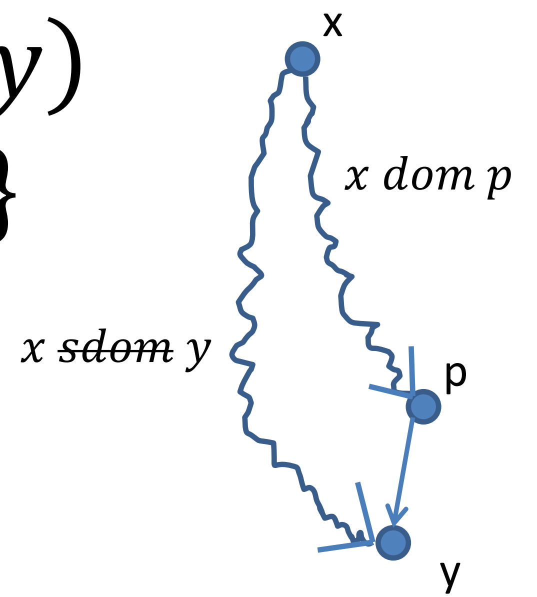
Алгоритм построения (размещение ф-функций)

Def: Dominance frontier DF(x) для одной вершины

Пусть дан некий CFG

x – вершина CFG графа

$$DF(x) = \{y | \exists p \in Pred(y)$$
$$x\ dom\ p\ \&\&\ x\ \cancel{sdom}\ y\}$$

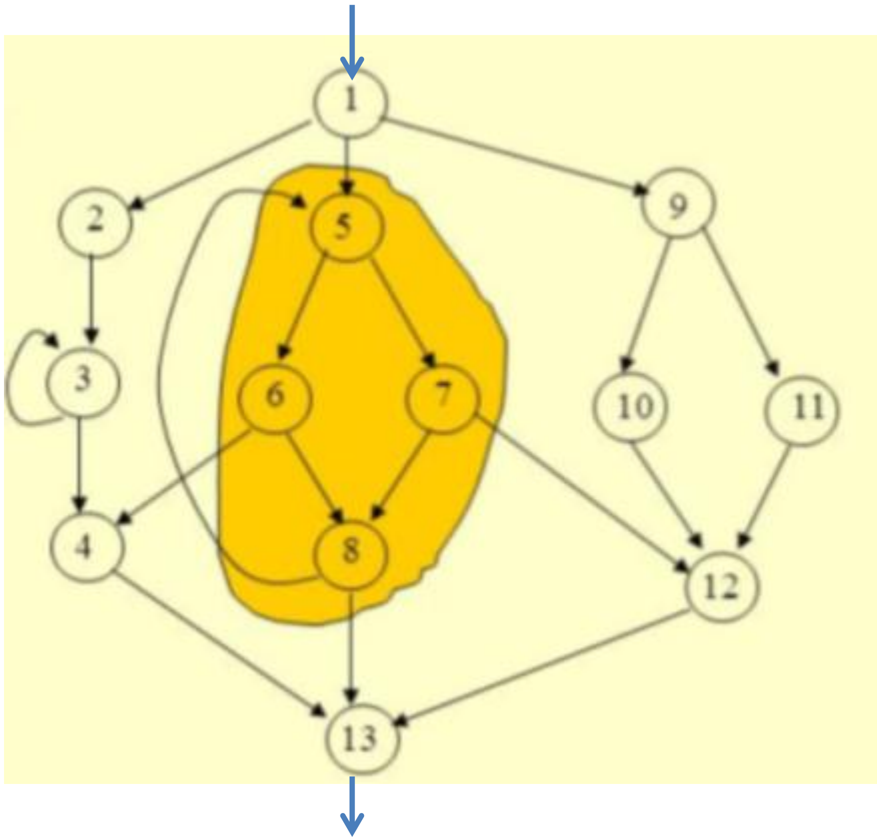


$x\ sdom\ y$ означает $x\ dom\ y$ **и** $x \neq y$

=>

$x\ \cancel{sdom}\ y$ означает $x\ \cancel{dom}\ y$ **или** $x = y$

Алгоритм построения (размещение ф-функций)



DF(5) = { 4, 5, 12, 13}

Узлы 4, 13, 12 представляют собой блоки, над которыми блок 5 не доминирует и они такие встречаются первыми на путях из 5.
Т.е. это граница, где исчезает доминирование блока 5.

Узел 5 – особый случай

Алгоритм построения (размещение ф-функций)

Def: Dominance frontier DF(S) для нескольких вершин

Пусть дан некий CFG, V – множество вершин

$S \subseteq V$

$$DF(S) = \bigcup_{x \in S} DF(x)$$

Алгоритм построения (размещение ф-функций)

Def: Итерированное dominance frontier $DF^{+}(S)$ для нескольких вершин

Пусть дан некий CFG, V – множество вершин

$$S \subseteq V$$

$$DF_1 = DF(S)$$
$$DF_2 = DF(S \cup DF_1)$$
$$DF_3 = DF(S \cup DF_2)$$
$$\dots$$
$$DF_{i+1} = DF(S \cup DF_i)$$

Процесс делается до тех пор, пока DFk не перестанет меняться

Алгоритм построения (размещение ф-функций)

Оказывается, что

Теорема #1:

$$DF^{+}(S) = J^{+}(S)$$

Алгоритм построения (размещение ф-функций)

Т.о., для размещения ф-функций нам нужно

- вычислить DF(S)
- “проитерировать” вычисленное выше множество

Алгоритм построения (размещение ϕ-функций)

Прямое вычисление DF(x) имеет квадратичную сложность от числа вершин

Теорема #2:

Children в доминаторном дереве

$$DF(x) = DF_{local}(x) \cup \bigcup_{z \in children(x)} DF_{up}(z)$$

$$DF_{local}(x) = \{\mathrm{y} \mid \mathrm{y} \in Succ(x) \text{ и } \mathrm{x} \cancel{\mathrm{dom}} \mathrm{ y}\}$$

$$DF_{up}(z) = \{\mathrm{y} \mid \mathrm{y} \in \mathrm{DF(z)} \text{ и } \mathrm{idom(z)} \cancel{\mathrm{dom}} \mathrm{ y}\}$$

Алгоритм построения (размещение ф-функций)

Алгоритм построения DF для вершины

Вход: CFG

Выход: для каждой вершины готовый DF(x)

Метод:

```
list<vertex> post_order;
map<vertex, set<vertex>> DF;
foreach(vertex x in post_order)
{
    DF[x] = new set<vertex>;
    foreach(vertex y in succ(x))
        if ( idom(y) != x ) DF[x] += y;        /* local */
    foreach(vertex z in children(x))
        foreach(vertex y in DF(z))
            if ( idom(y) != x ) DF(x) += y;  /* up */
}
return DF;
```

Алгоритм построения (размещение ф-функций)

Алгоритм построения DF для мн-ва вершин

Вход: CFG, $S \subseteq V$

Выход: DF(S)

Метод: DF_Set(S)

```
set<vertex> res;            /* результат */
foreach ( vertex v in S )
{
    res += DF(v);
}
return res;
```

Алгоритм построения (размещение φ-функций)

Алгоритм построения $DF^{+}(S)$ для мн-ва вершин
Вход: CFG, $S \subseteq V$
Выход: DF(S)
Метод: DFP_Set(S)

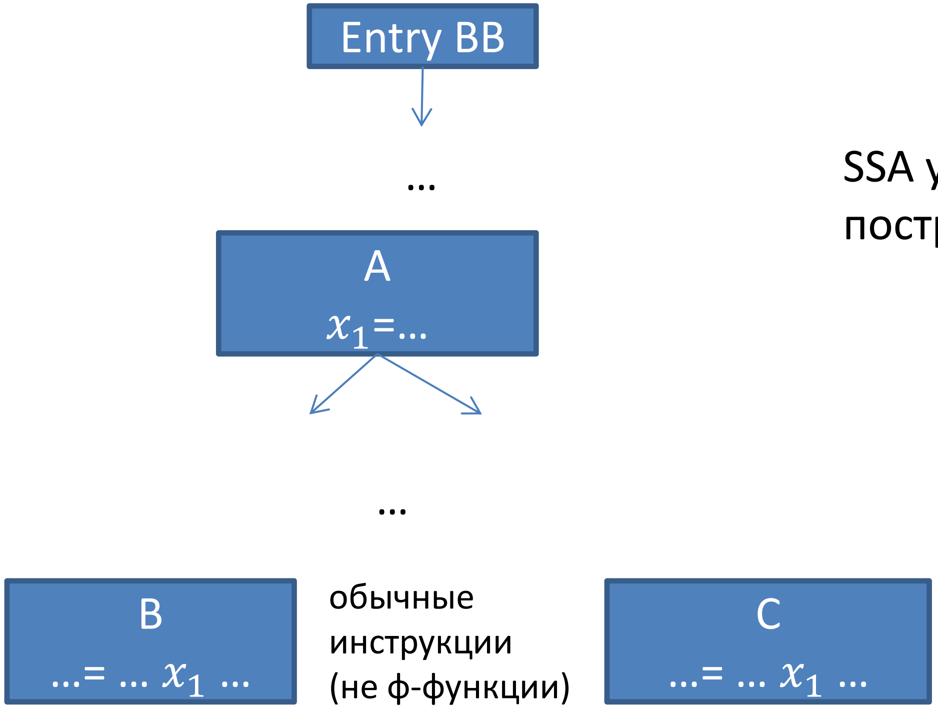
```
set<vertex> res;
set<vertex> DFP;
bool change = true;
DFP = DF_Set(S);
do
{
    change = false;
    DFP = DF_Set(S+DFP);
    if ( DFP != res )
    {
        DFP = D;
        change = true;
    }
}
while( change );
return res;
```

Алгоритм построения (определение версий переменных)

Св-во SSA-формы

Определения переменных доминируют над их использованиями в обычных инструкциях (не ф-функции)

Алгоритм построения (определение версий переменных)

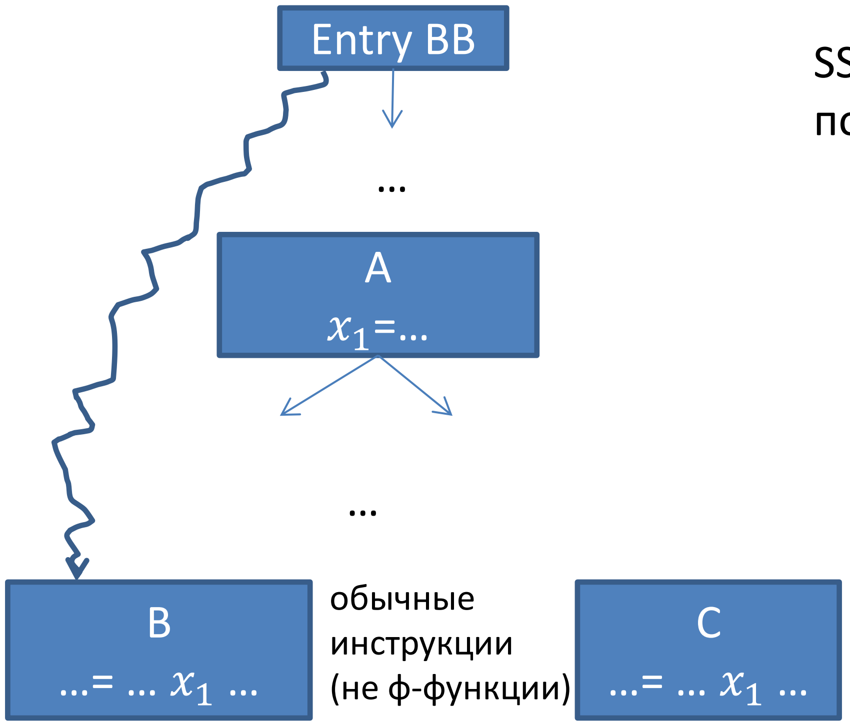


SSA уже здесь построено

A dom B

A dom C

Алгоритм построения (определение версий переменных)



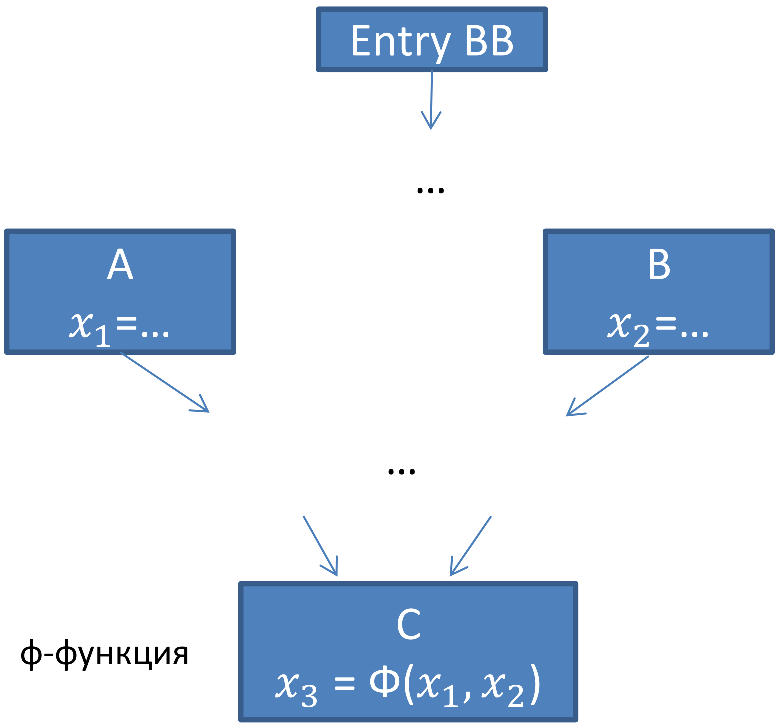
SSA уже здесь построено

Если

A ~~dom~~ B,

то в B можно попасть, минуя присваивание

Алгоритм построения (определение версий переменных)



SSA уже здесь построено

A ~~dom~~ B

B ~~dom~~ C

Алгоритм построения (определение версий переменных)

Алгоритм: переименование переменной p
Вход: CFG, вставлены ф-функции для переменной p
Выход: CFG с переименованными переменными
Метод: RenameVar(p)

```
int counter=0;
list<int> stack;
Traverse(entry);

/* продолжение на след. слайде */
```

```
void Traverse(vertex v)
{
  foreach(stmt s in v.stmts)
  {
     if ( !s.is_phi )
       заменить p на pi в s.rhs, где i = stack.Top();

     заменить p на pi в s.lhs, где i=counter;
     stack.push(i);
     counter = i + 1;
  } /* first loop */
  foreach(vertex v1 in Succ(v)) /* - последователи в CFG */
  {
      int j = WhichPred(v1, v); /* порядковый номер v в
                                   массиве предшественников v1 */
      foreach(stmt phi in v1.phis)
        заменить j-ый операнд p в phi.rhs на pi, где i=stack.top();
  } /* second loop */
  /* продолжение на следующем слайде */
```

vertex - базовый блок

v.stmts – инструкции в ББ

s.is_phi – True, если инструкция является Ф-функцией

s.rhs – правая часть присваивания

s.lhs – левая часть присваивания

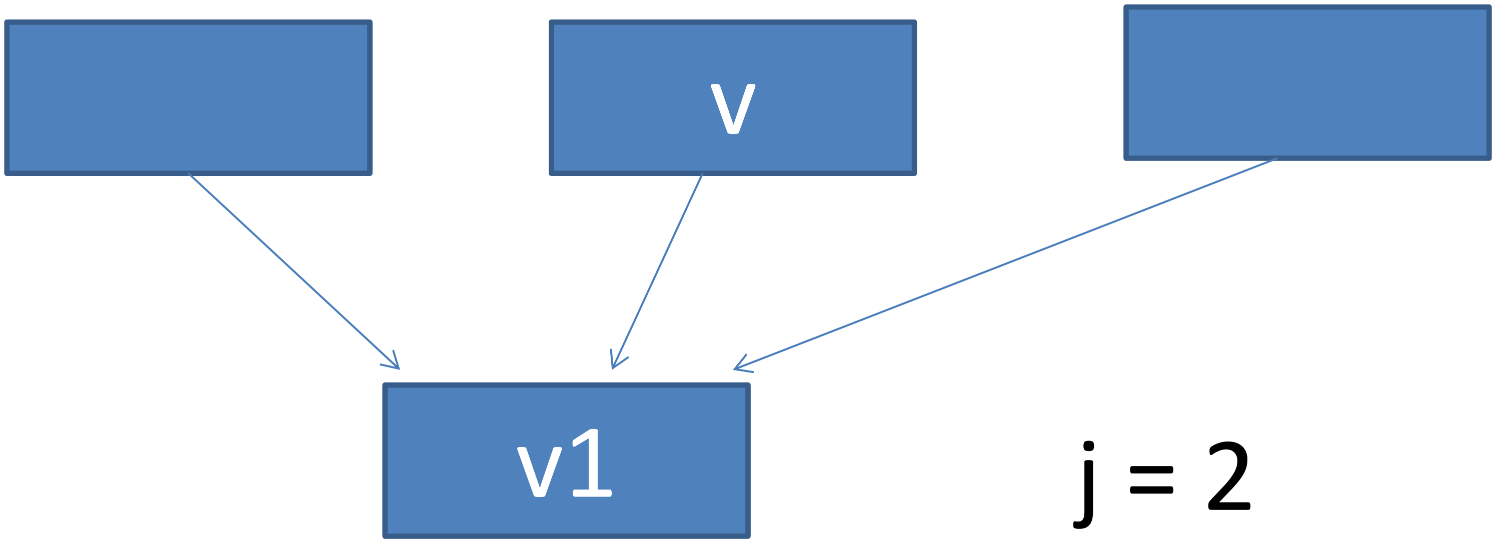
```
  foreach(vertex v1 in Child(v)) /*
                                  child – последователи в
                                  дереве доминаторов */
  {
     Traverse(v1);   /* рекурсивный вызов */
  }

  foreach(stmt s in v.stmts)
  {
     if ( в s.lhs переменная p любой версии )
        stack.pop();
  }

} /* конец Traverse(v) */
```

Алгоритм построения (определение версий переменных)

int j = WhichPred(v1, v); /* порядковый номер v в массиве предшественников v1 */



j = 2

Список литературы

- Компиляторы. Принципы, технологии и инструментарий. 2-е изд. А. Ахо (бит-векторный анализ)
- Компиляторы. Принципы, технологии и инструментарий. 1-е изд. А. Ахо (бит-векторный анализ)
- Advanced compiler design & implementation. S. Muchnik (SSA)
- Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. R. Cytron

Дополнение / факультатив

на зачёте спрашиваться НЕ будет

Передаточные функции

Def: структура DFA (D, F, V, $\wedge$) монотонна, если

$$\forall\, f \in F \text{ и } \forall\, x, y \in V \mid x \leq y$$

$$=>$$

$$f(x) \leq f(y)$$

Передаточные функции

мн-во передаточных функций F $f: V \to V$ обладает св-вами

- F содержит тождественную функцию I |

  $\forall x \in V \;\; I(x)=x$
- F замкнуто относительно композиции функций

Легко видеть, что передаточные функции в DFA (RD, LV, AE) удовлетворяют этим условиям

Передаточные функции

Def: структура DFA $(D, F, V, \wedge)$ дистрибутивна, если

$$\forall\, x, y \in V$$

$$\Rightarrow$$

$$f(x \wedge y) = f(x) \wedge f(y)$$

Обобщённый алгоритм (св-ва)

Если алгоритм сходится, то получающийся результат является решением уравнений потоков данных.

⊲ Если при выходе из некоторой итерации цикла while уравнения не удовлетворяются, то в OUT (IN) будет внесено изменение => потребуется вернуться в цикл while ⊳

Обобщённый алгоритм (св-ва)

Если структура монотонна, то решение (MFP = Maximum Fixed Point) обладает тем св-ом, что в любом другом решении значения IN/OUT не превышают соответствующих значений MFP.

$\lhd$

1) Докажем по индукции, что значения IN[B] и OUT[B] могут только не увеличиваться

Базис: Тривиален, т.к. все начальные значения устанавливаются в T

Индукция: Пусть $\text{OUT}[P]^k \leqslant \text{OUT}[P]^{k-1}$

$$\text{IN}[B] = \wedge_{P-\text{предшественник } B} \text{OUT}[P]$$

$\text{OUT}[P]^k \leqslant \text{OUT}[P]^{k-1}$ => $\text{IN}[B]^{k+1} \leqslant \text{IN}[B]^k$ (в силу св-в оператора сбора)

$\text{OUT}[B] = f_B(\text{IN}[B])$ в силу монотонности передаточной функции $\text{IN}[B]^{k+1} \leqslant \text{IN}[B]^k$ => $\text{OUT}[B]^{k+1} \leqslant \text{OUT}[B]^k$

2) Оператор сбора возвращают GLB своих аргументов. Передаточные функции возвращают только решения, согласующиеся с самим блоком. => результат алгоритма должен быть не меньше любого другого решения ур-ий

$\rhd$

Обобщённый алгоритм (св-ва)

Если полурешётка структуры монотонна и имеет конечную высоту, то алгоритм сходится.

⊲

Значения IN[B] / OUT[B] не увеличивается при при каждом изменении. Алгоритм завершает свою работу при отсутствии изменений. =>

Алгоритм сойдётся не позже чем N*M итераций

N – кол-во узлов графа, M – высота полурешётки

⊳

Идеальное решение

Рассмотрим **возможный** путь

(“возможный” – означает, что некоторое вычисление программы следует по этому пути)

$$P = \text{Вход} \rightarrow B_1 \rightarrow B_2 \rightarrow \cdots \rightarrow B_{k-1} \rightarrow B_k$$

передаточная функция для этого пути

$$f_P \qquad\qquad f_{B_1}, f_{B_2}, \ldots, f_{B_{k-1}}$$

Определим идеальное решение как

$$\mathrm{IDEAL}\,[B] = \bigwedge_{P-\text{возможный путь от входа к } B} f_P\left(v_{\text{Вход}}\right)$$

Идеальное решение (св-ва)

- любое решение, большее, чем IDEAL не верен
- любое решение, меньшее или равное, консервативно (безопасно)
- значение, более близкое к идеальному, является более точным

Решение сбором по путям (MOP - решение)

$$\mathrm{MOP}[B] = \bigwedge_{P \text{ - путь от входа к } B} f_P(v_{\text{вход}})$$

Здесь включаются все пути в графе до вершины B даже те, которые реально не могут быть пройдены в процессе вычисления.

Очевидно, что

$$\mathrm{MOP}[B] \leqslant \mathrm{IDEAL}[B]$$

MFP и MOP-решения

MOP-решение в общем случае получить нельзя из-за циклов

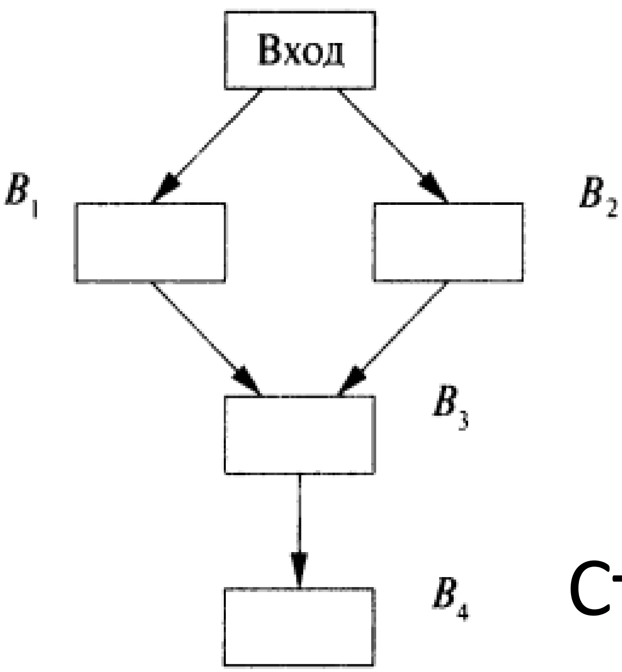
Обобщённый алгоритм посещает ББ не обязательно в порядке их выполнения

Как соотносятся MFP и MOP решения?

MFP и MOP-решения

Оказывается можно показать, что

$$\mathrm{MFP} \leqslant \mathrm{MOP};$$



Для MOP-решения

$$\mathrm{MOP}[B_4] = ((f_{B_3} \circ f_{B_1}) \wedge (f_{B_3} \circ f_{B_2}))(v_{\text{Вход}})$$

Для обобщённого алгоритма

$$\mathrm{IN}[B_4] = f_{B_3}((f_{B_1}(v_{\text{Вход}}) \wedge f_{B_2}(v_{\text{Вход}})))$$

Структура монотонна =>

$$\mathrm{IN}[B_4] \leqslant \mathrm{MOP}[B_4]$$

Итого

$$\text{MOP} \leqslant \text{IDEAL},$$

$$\text{MFP} \leqslant \text{MOP},$$

=>

$$\text{MFP} \leqslant \text{IDEAL},$$

Результат работы обобщённого алгоритма консервативен (т.е. безопасен)