6. Распространение констант

6.1 Форма статического единственного присваивания (SSA-форма)

6.1.1. Постановка задачи

◊ *Форма статического единственного присваивания* (*SSA*) позволяет в каждой точке программы объединить

- ◊ информацию об имени переменной
- ◊ и информацию о текущем значении этой переменной

(или, что то же самое, информацию о том, ***какое из определений данной переменной определяет ее текущее значение в данной точке***).

Исходная программа **Обычное представление** **Представление в SSA-форме**

```
int bar();
void foo(int a) {
    int b = a + 3;
    b = bar();
    b = a + 3;
    b = b + a;
}
```

→

```
foo:
    param a
    b = a + 3
    b = bar(b)
    b = a + 3
    b = b + a
```

→

```
foo:
    param a.0
    b.0 = a.0 + 3
    b.1 = bar(b.0)
    b.2 = a.0 + 3 // <- b.0
    b.3 = b.2 + a.0
```

Исходная программа

```
int bar();
void foo(int a)
{
    int b = 0;
    b = bar();
    b = a + 3;
    if (a == 0)
    {
        b = b * 4;
    }
    b = b + a;
}
```

→

Обычное представление

```
foo:
    param a
    b = 0
    b = bar()
    b = a + 3
    if (a != 0) goto next
    b = b * 4

next:
    b = b + a
```

→

Представление в SSA-форме

```
foo:
    param a.0
    b.0 = 0
    b.1 = bar()
    b.2 = a.0 + 3
    if (a.0 == 0) goto l1 : l2

l1:
    b.3 = b.2 * 4

l2:

    b.4 = b.? + a.0
```

Исходная программа

```
int bar();
void foo(int a)
{
    int b = 0;
    b = bar();
    b = a + 3;
    if (a == 0)
    {
        b = b * 4;
    }
    b = b + a;
}
```


Обычное представление

```
foo:
    param a
    b = 0
    b = bar()
    b = a + 3
    if (a != 0) goto next
    b = b * 4

next:
    b = b + a
```

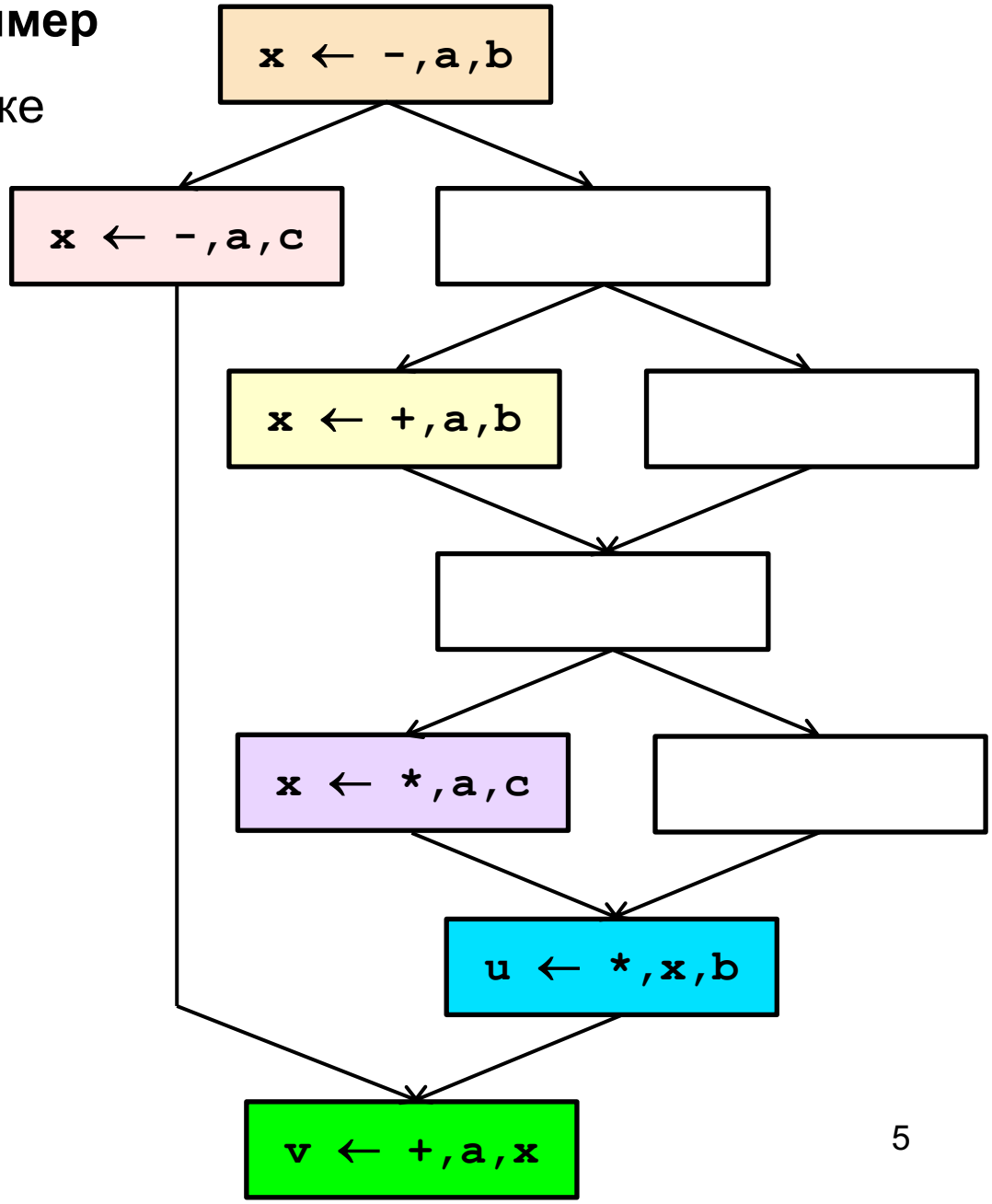

Представление в SSA-форме

```
foo:
    param a.0
    b.0 = 0
    b.1 = bar()
    b.2 = a.0 + 3
    if (a.0 == 0) goto l1 : l2

l1:
    b.3 = b.2 * 4

l2:
    b.4 = phi(b.2, b.3)
    b.5 = b.4 + a.0
```

6.1 SSA-форма

6.1.1. Постановка задачи. Пример

- ◊ Фрагмент ГПУ на рисунке содержит четыре определения **x**
- ◊ Использования в синем блоке достигают три определения **x**, в зеленом – четыре определения **x**
- ◊ Цель же состоит в том, чтобы ***каждого использования достигало только одно определение***



6.1 SSA-форма

6.1.1. Постановка задачи. Пример

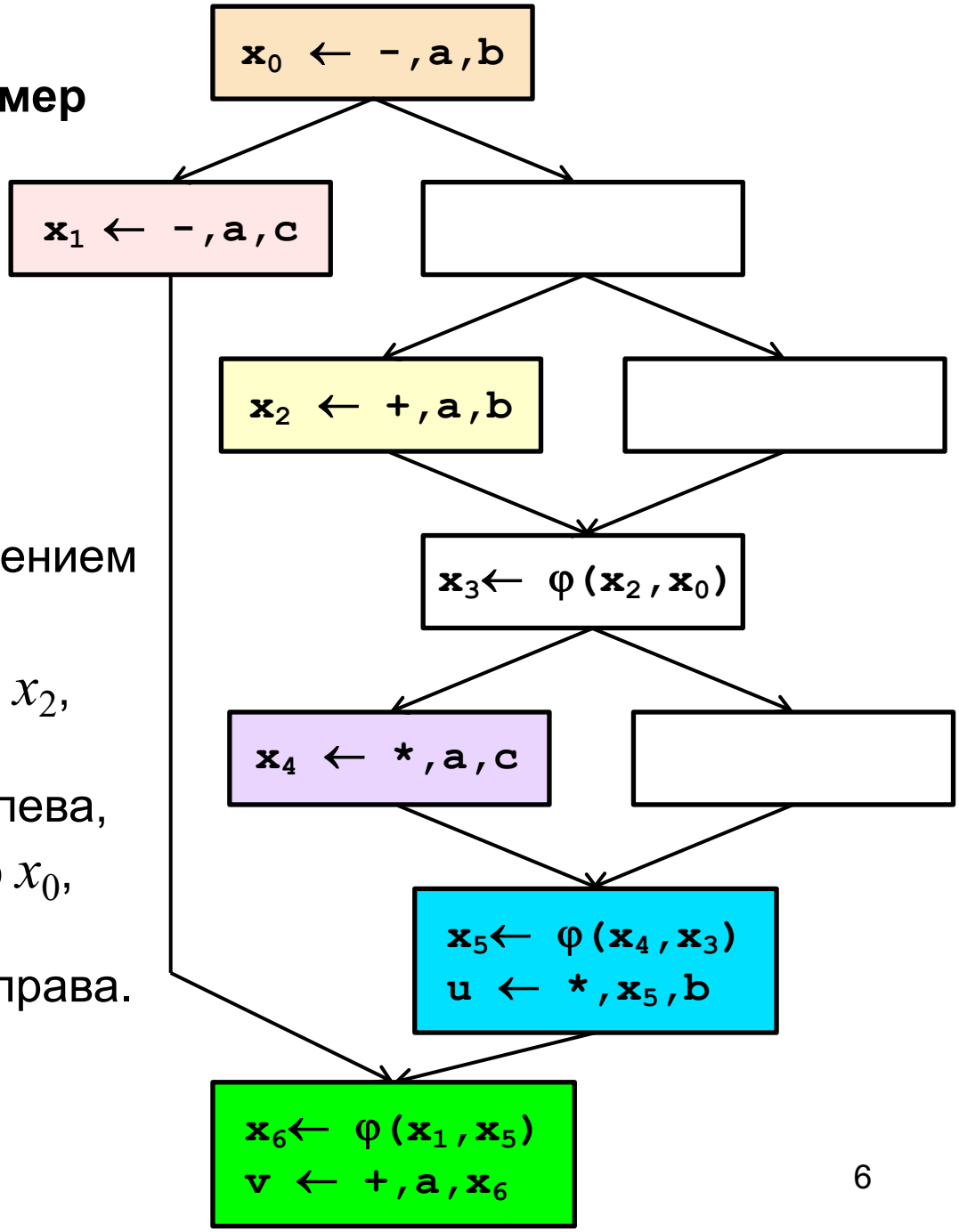
- Введем «функцию» объединения значений, или *φ-функцию*

- По определению

$$x_3 \leftarrow \varphi(x_2, x_0)$$

является новым определением переменной x:

  - значение x_3 равно x_2, когда управление попадает в блок слева,
  - значение x_3 равно x_0, когда управление попадает в блок справа.



6.1 SSA-форма

6.1.2. Определение φ-функции

- ◊ φ-функция определяет *SSA-имя* для значения своего аргумента, соответствующего ребру, по которому управление входит в блок.
- ◊ **При входе в базовый блок все его φ-функции выполняются одновременно и до любого другого оператора, определяя целевые *SSA*-имена.**

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
    x = x + 1
    y = y + x
}
```

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

Каждая из φ-функций *объединяет* значения своих аргументов в новое значение, определением которого она является.

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

Каждая из φ-функций *объединяет* значения своих аргументов в новое значение, определением которого она является.

До цикла –	x_0, y_0
На входе в цикл –	x_1, y_1
Внутри цикла –	x_2, y_2
После цикла –	x_3, y_3

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

◊ **Затруднение**. Первый аргумент $\varphi(x_0, x_2)$ определяется в блоке, который предшествует циклу, второй аргумент определяется позже в блоке, содержащем $\varphi(x_0, x_2)$. Следовательно, при первом выполнении $\varphi(x_0, x_2)$ ее второй аргумент еще не определен.

6.1 SSA-форма

6.1.2. Определение φ-функции. Пример

```
x =
y = …
while (x < 100){
   x = x + 1
   y = y + x
}
```

```
      x0 = …
      y0 = …
      if (x0 ≥ 100) goto next
loop: x1 = φ(x0, x2)
      y1 = φ(y0, y2)
      x2 = x1 + 1
      y2 = y1 + x2
      if (x2 < 100) goto loop
next: x3 = φ(x0, x2)
      y3 = φ(y0, y2)
```

◊ **Выход из затруднения**: по определению любая φ-функция (а значит и $\varphi(x_0, x_2)$) **читает только один из своих аргументов**, а именно тот аргумент, который соответствует последнем пройденному ребру ГПУ.

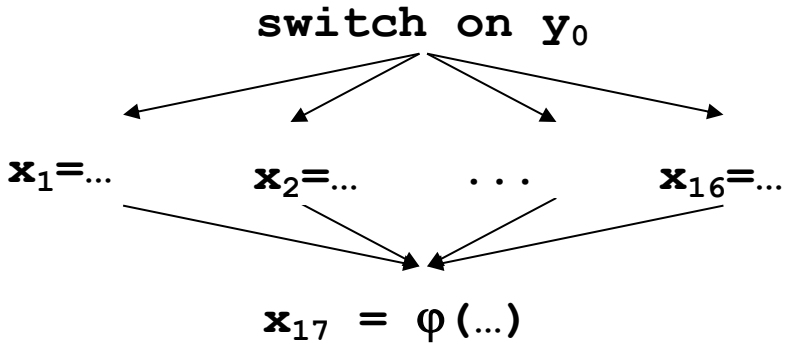
Поэтому φ-функция не прочитает неопределенного значения.

6.1 SSA-форма

6.1.3. Количество аргументов φ -функции

◊ По определению у φ **-функции может быть любое число аргументов**.

Например, на рисунке φ -функция, у которой 16 аргументов:



7.1 Глобальное распространение констант

7.1.1 Постановка задачи

◊ **Распространение констант** – это оптимизирующее преобразование, заменяющее **выражения**, которые при выполнении всякий раз вычисляют одну и ту же константу, **самой** этой **константой**.

◊ Преобразование удобно выполнять над процедурой (программой) в SSA-форме.

7.1 Глобальное распространение констант

7.1.1 Постановка задачи

- ◊ **Распространение констант** – это оптимизирующее преобразование, заменяющее **выражения**, которые при выполнении всякий раз вычисляют одну и ту же константу, **самой** этой **константой**.
- ◊ Преобразование удобно выполнять над процедурой (программой) в SSA-форме.
- ◊ Распространение констант является прямой задачей потока данных.

7.1 Глобальное распространение констант

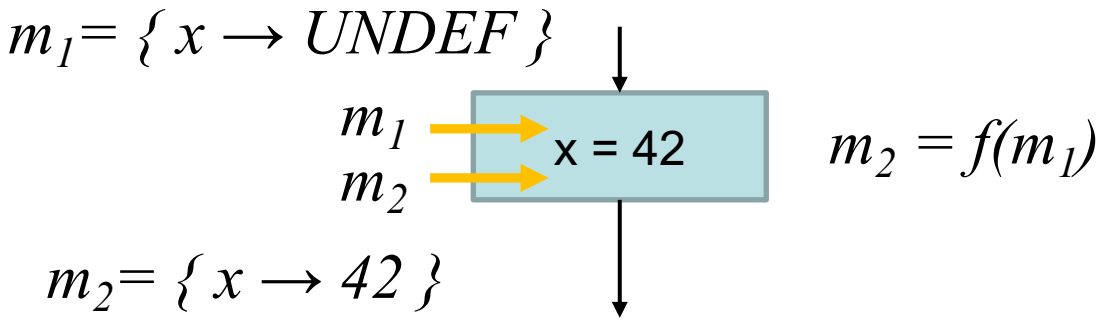
7.1.1 Постановка задачи

- **Распространение констант** – это оптимизирующее преобразование, заменяющее **выражения**, которые при выполнении всякий раз вычисляют одну и ту же константу, **самой** этой **константой**.
- Преобразование удобно выполнять над процедурой (программой) в SSA-форме.
- Распространение констант является прямой задачей потока данных.
- Выполняя глобальный анализ потока данных (методом итераций) рассматриваем в процедуре все SSA-имена v и каждому из них *сопоставляем константу*; это может быть:
  - специальная константа $Undef$ – значение переменной не определено;
  - одна из множества рассматриваемых констант;
  - специальная константа NAC (*not-a-constant*) – переменная не может быть константой.

7.1 Глобальное распространение констант

Основная идея

Пусть m_1 и m_2 – состояния программы до/после некоторой инструкции, f – передаточная функция



Состояние программы в каждой точке – отображение множества *SSA*-имен на полурешетку констант.

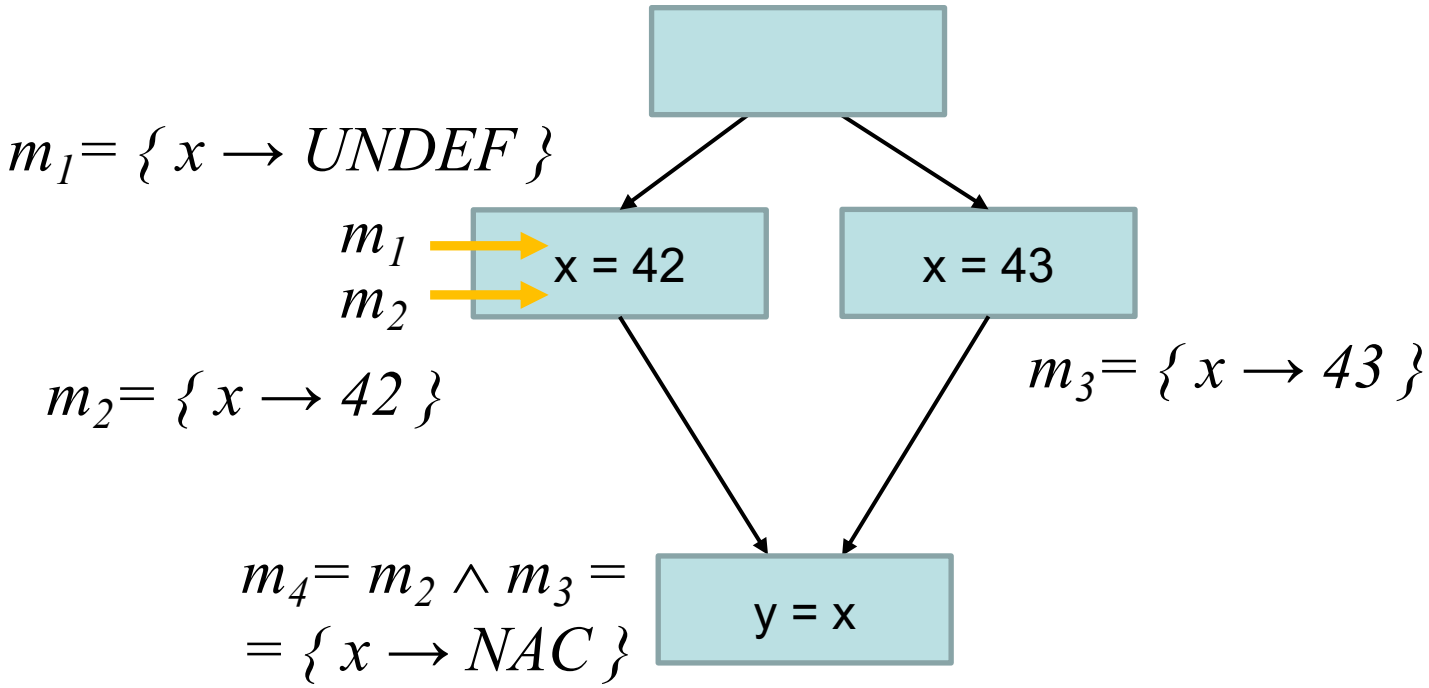
Альтернативная форма записи:

$m_2(x) = 42$

7.1 Глобальное распространение констант

Основная идея

Пусть m_1 и m_2 – состояния программы до/после некоторой инструкции, f – передаточная функция



7.1 Глобальное распространение констант

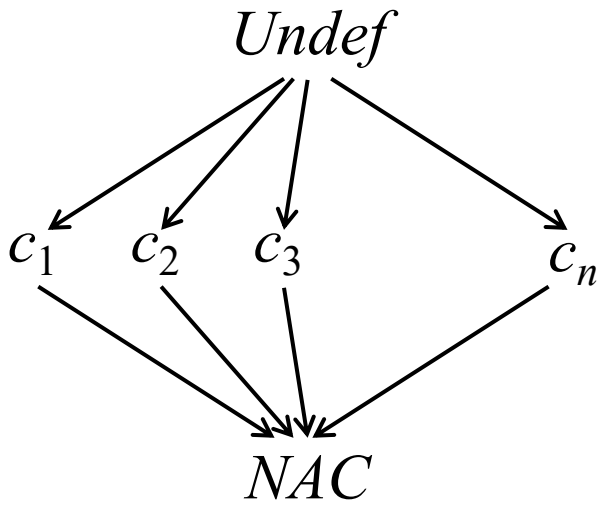
7.1.2 Константы *NAC* и *Undef*

- ***SSA*-имени *v* сопоставляется константа *NAC*** если установлено, что:
  - v может быть присвоено входное значение, т.е.
    - v – имя формального параметра процедуры или
    - v – имя целевой переменной функции ввода;
  - при вычислении v используется NAC
  - на различных путях вычисления v могут быть присвоены разные значения.
- ***SSA*-имени *v* сопоставляется константа *Undef***, если не найдено ни одного определения v, достигающего рассматриваемой точки. На самом деле всем SSA-именам сначала сопоставляется константа $Undef$.

7.1 Глобальное распространение констант

7.1.2 Полурешетка значений констант

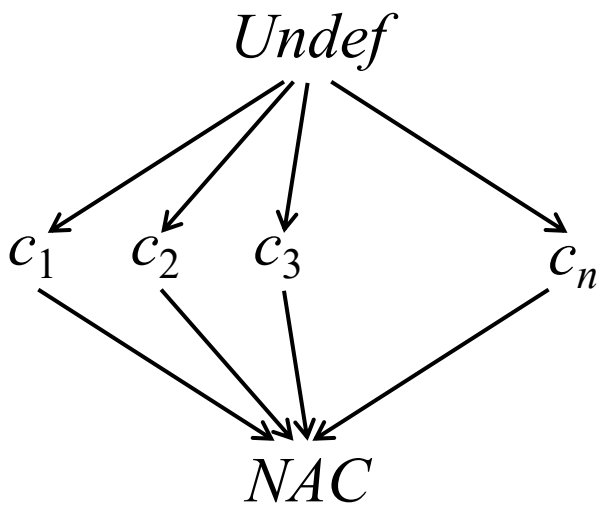
◊ Множество значений, сопоставляемых каждой переменной, (включая специальные значения $Undef$ и NAC) можно представить как полурешетку (см. рисунок).



7.1 Глобальное распространение констант

7.1.2 Полурешетка значений констант

◊ Множество значений, сопоставляемых каждой переменной, (включая специальные значения $Undef$ и NAC) можно представить как полурешетку (см. рисунок).

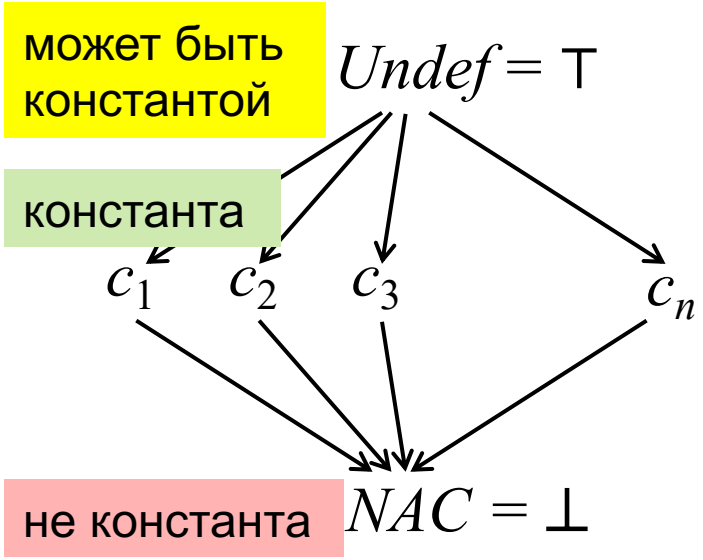


- верхний элемент $\top = Undef$
- нижний элемент $\bot = NAC$
- Значения констант (c_i) не сравнимы между собой, но все они «меньше» $Undef$ и «больше» NAC: $\forall i \;\; NAC \le c_i \le Undef$

7.1 Глобальное распространение констант

7.1.2 Полурешетка значений констант

◊ Множество значений, сопоставляемых каждой переменной, (включая специальные значения $Undef$ и NAC) можно представить как полурешетку (см. рисунок).

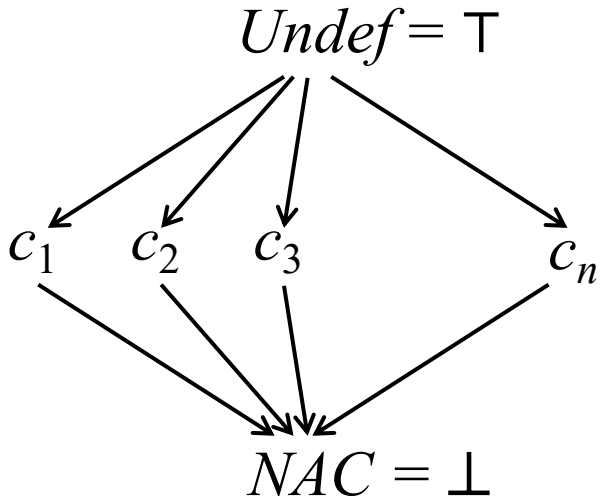


- верхний элемент $\top = Undef$
- нижний элемент $\bot = NAC$
- Значения констант (c_i) не сравнимы между собой, но все они «меньше» $Undef$ и «больше» NAC: $\forall i \ \ NAC \le c_i \le Undef$

7.1 Глобальное распространение констант

7.1.2 Полурешетка значений констант

◊ Множество значений, сопоставляемых каждой переменной, (включая специальные значения $Undef$ и NAC) можно представить как полурешетку (см. рисунок).



◈ для всех значений v:

$$Undef \wedge v = v$$

$$NAC \wedge v = NAC$$

◈ в частности

$$NAC \wedge Undef = NAC$$

◈ по определению

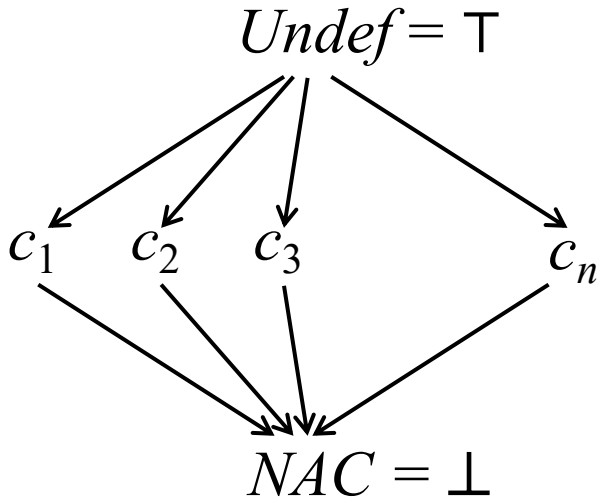
для любых двух констант c_1 и c_2

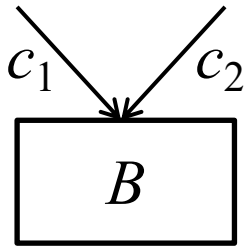
$$c_1 \wedge c_2 = \begin{cases} c_1 & \text{если } c_1 = c_2 \\ NAC & \text{если } c_1 \neq c_2 \end{cases}$$

7.1 Глобальное распространение констант

7.1.2 Полурешетка значений констант

◊ Множество значений, сопоставляемых каждой переменной, (включая специальные значения $Undef$ и NAC) можно представить как полурешетку (см. рисунок).





◈ для всех значений v:

$$Undef \wedge v = v$$

$$NAC \wedge v = NAC$$

◈ в частности

$$NAC \wedge Undef = NAC$$

◈ по определению

для любых двух констант c_1 и c_2

$$c_1 \wedge c_2 = \begin{cases} c_1 & \text{если } c_1 = c_2 \\ NAC & \text{если } c_1 \neq c_2 \end{cases}$$

7.1 Глобальное распространение констант

7.1.3 Операция сбора в структуре распространения констант

◊ Значение потока данных в структуре распространения констант представляет собой отображение

$$m: \mathcal{N} \to \mathcal{C},$$

где $\mathcal{N}$ – множество SSA-имен

$\mathcal{C}$ – полурешетка констант (для соответствующего типа переменной)

Значение, соответствующее SSA-имени n в отображении m, обозначается как $m(n)$

◊ Полурешетка значений потока данных представляет собой декартово произведение k полурешеток $\mathcal{C}$, где k – количество переменных в анализируемой процедуре:

$$m \wedge m' = m'' \quad \Leftrightarrow \quad \forall n : m''(n) = m(n) \wedge m'(n).$$

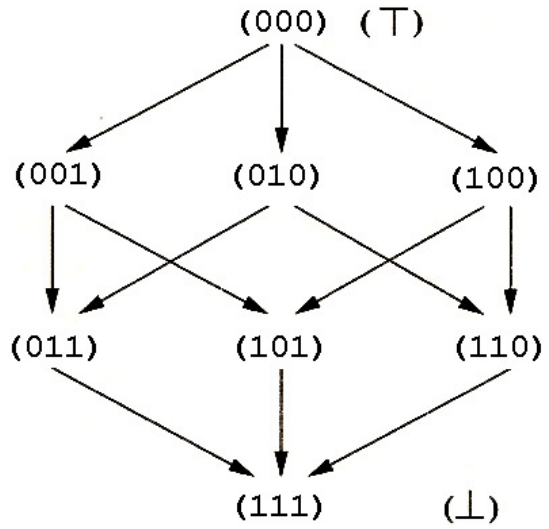
$$m \le m' \quad \Leftrightarrow \quad \forall n : m(n) \le m'(n)$$

(Повторение: полурешетки)

Диаграммы полурешеток

◊ *Диаграмма полурешетки* $\langle L, \wedge \rangle$ представляет собой граф, узлами которого являются элементы L, а ребра направлены от x к y, если $y \leq x$.

◊ **Пример**. На рисунке – диаграмма полурешетки $\langle U, \cup \rangle$, $|U| = 8$: элемент множества U представляется битовым 3-вектором.



(определение $\leq$ *:*
$x \leq y <=> x \wedge y = x$ *)*

7.1 Глобальное распространение констант

7.1.4 Передаточные функции структуры распространения констант

◊ Множество передаточных функций $\mathcal{F}_{\mathcal{CP}}$ состоит из некоторых передаточных функций, которые принимают отображения переменных на значения в решетке констант и возвращают другие отображения данного вида.

- ⧫ $\mathcal{F}_{\mathcal{CP}}$ содержит тождественную функцию i, которая получает в качестве аргумента и возвращает одно и то же отображение: $\forall m \;\; i(m) = m$
- ⧫ $\mathcal{F}_{\mathcal{CP}}$ содержит константную передаточную функцию c для входного узла, которая возвращает, независимо от полученного аргумента, отображение m_0, где $m_0(v) = Undef$ для всех переменных v.
  Она используется в качестве граничного условия:
  $c(m_{Entry}) = m_0$ означает, что перед началом выполнения программы не определена ни одна переменная.

7.1 Глобальное распространение констант

7.1.4 Передаточные функции структуры распространения констант

◊ Пусть $f_s \in \mathcal{F}_{CP}$ – передаточная функция для инструкции s, и пусть m и m' – значения потока данных в точках до и после s: $m' = f_s(m)$.

◊ Опишем f_s через m и m':

(1) Если s – не содержит присваивания какому-либо SSA-имени, то f_s – тождественная функция i, т.е. для всех имен переменных v: $m'(v) = m(v)$

(2) Если s – содержит присваивание SSA-имени x, то
для $v \neq x$ $m'(v) = m(v)$,
для $v = x$ $m'(v) = m(x)$

7.1 Глобальное распространение констант

7.1.4 Передаточные функции структуры распространения констант

◊ Описание f_s через m и m'

(2) Пусть s – присваивание переменной x и пусть m и m' – значения потока данных, тогда $m' = m(x)$ определяется следующим образом:

(a) Если s – присваивание константы c: $x = c$, то $m'(x) = c$

(b) Если s – присваивание выражения: $x = y + z$, то

$$m'(x) = \begin{cases} m(y) + m(z), & \text{если } m(y) \text{ и } m(z) \text{ константы} \\ NAC, & \text{если } m(y) = NAC \text{ или } m(z) = NAC \\ Undef & \text{в остальных случаях} \end{cases}$$

(c) Если s – присваивание любого другого выражения (например, вызова функции или указателя), то $m'(x) = NAC$.

7.1 Глобальное распространение констант

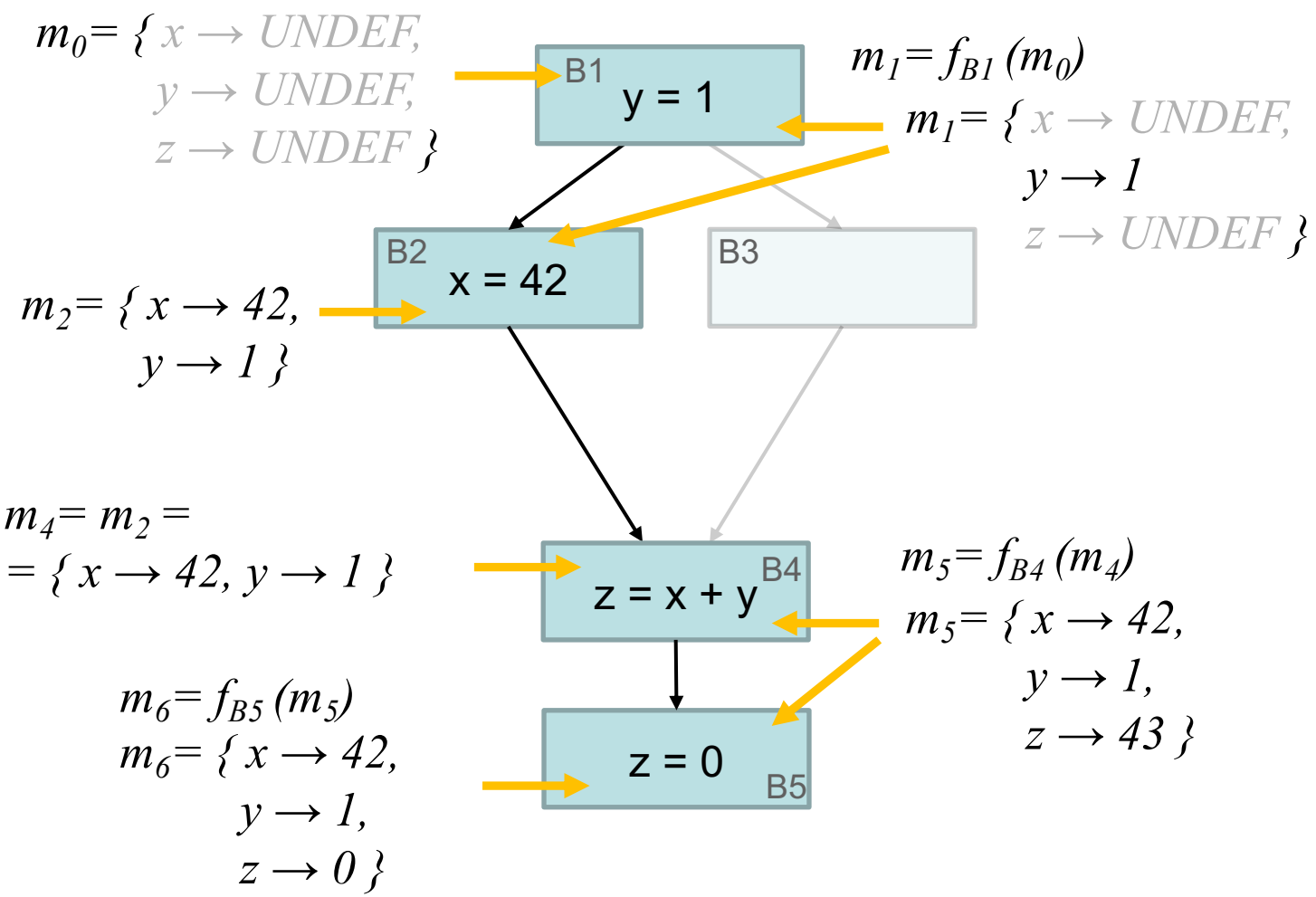
7.1.5 Монотонность структуры распространения констант

◊ **Утверждение**. Структура распространения констант монотонна:
если для m_1, m_2, m_1', m_2' выполняются соотношения:
$m_1 \le m_2, \quad m_1' = f_s(m_1), m_2' = f_s(m_2)$,
то $m_1' \le m_2'$
(т.е. меньшим значениям аргумента соответствует меньшее значение функции).

7.1 Глобальное распространение констант

7.1.5 Монотонность структуры распространения констант

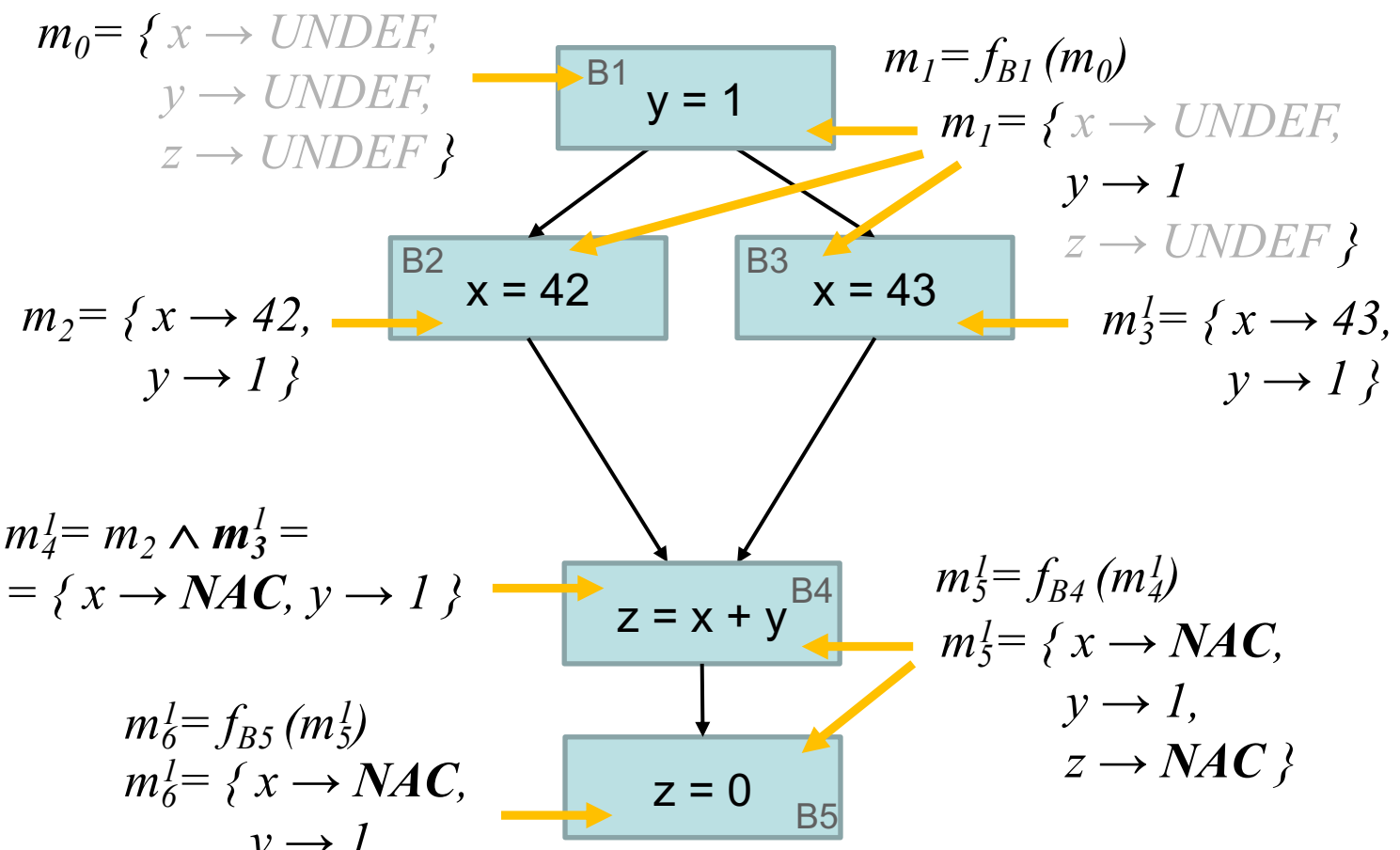
Пусть m_i – состояния программы, f_{Bn} – передаточные функции. Сначала рассмотрим изменения состояний программы только вдоль одного пути B1→B2→B4→B5:



7.1 Глобальное распространение констант

7.1.5 Монотонность структуры распространения констант

Добавим путь B1→B3→B4 и убедимся, что все состояния m_i^1 изменились монотонно, т.е. раз уж $m_4^1 \le m_4$ вследствие операции сбора, то и последующие $m_i^1 \le f_{Bn}(m_j^1)$.

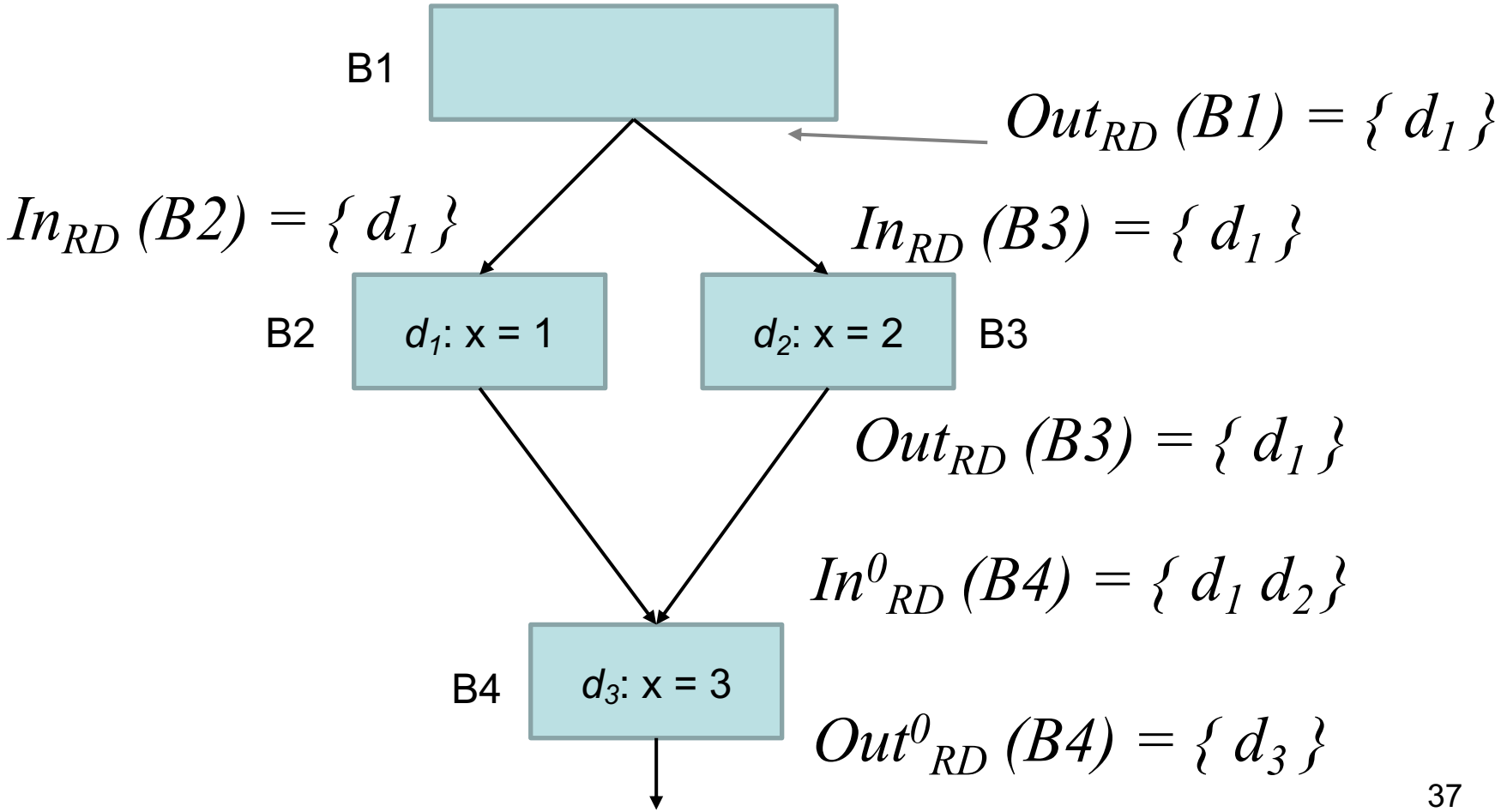


(На следующих слайдах монотонность будет доказана в общем виде)

7.1 Глобальное распространение констант

7.1.5 Монотонность структуры распространения констант

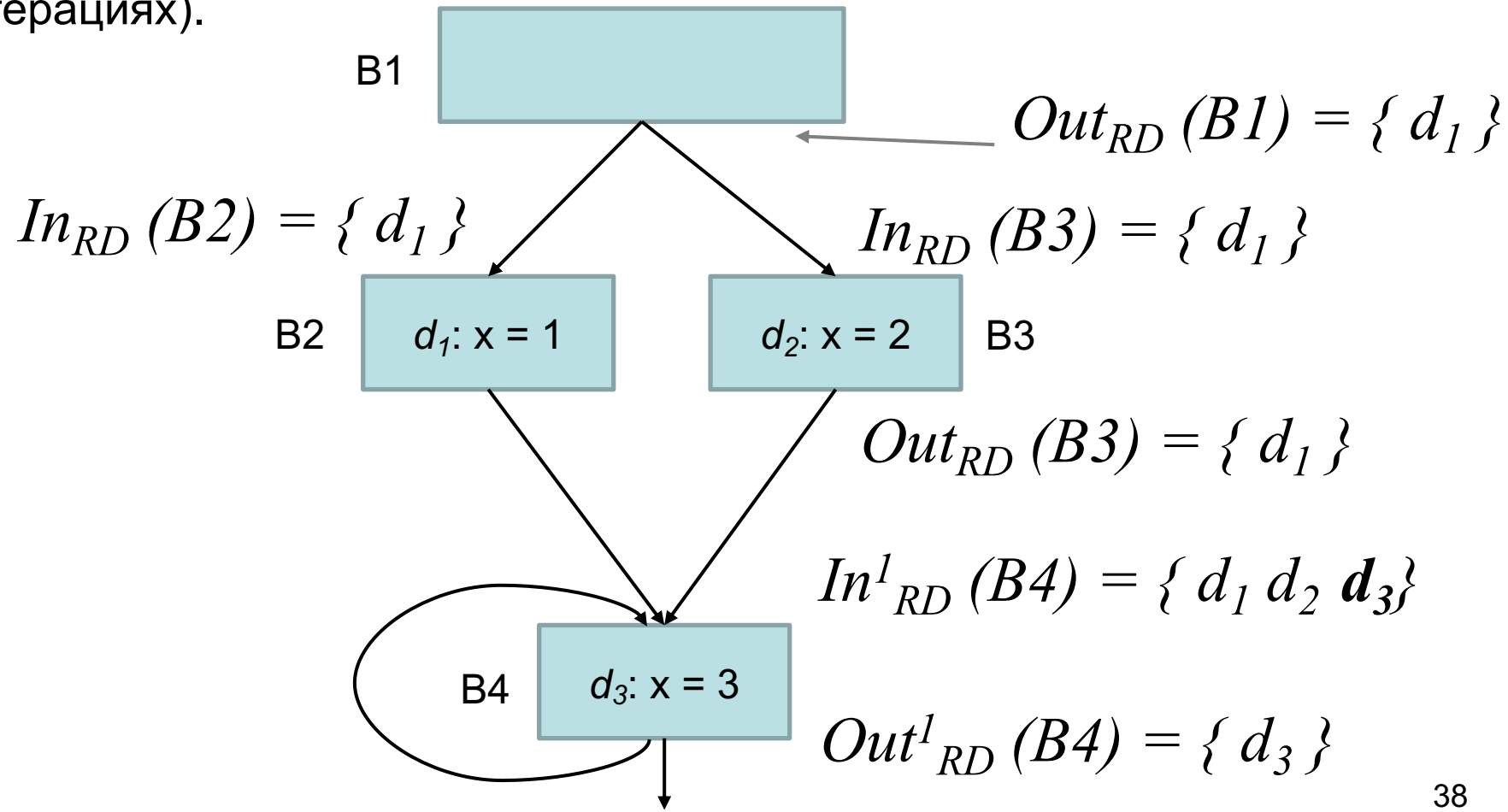
Монотонность для структуры распространения констант имеет тот же смысл, что и для других задач, например, достигающих определений:



7.1 Глобальное распространение констант

7.1.5 Монотонность структуры распространения констант

Монотонность: $x \le y => f(x) \le f(y)$. Здесь x и y – это состояния в одной и той же точке до/после рассмотрения еще одного пути (например, на разных итерациях).



Монотонность: $In^1_{RD}(B4) \le In^0_{RD}(B4) =>$
$Out^1_{RD}(B4) = f_{B4}(In^1_{RD}(B4)) \le f_{B4}(In^0_{RD}(B4)) = Out^0_{RD}(B4)$

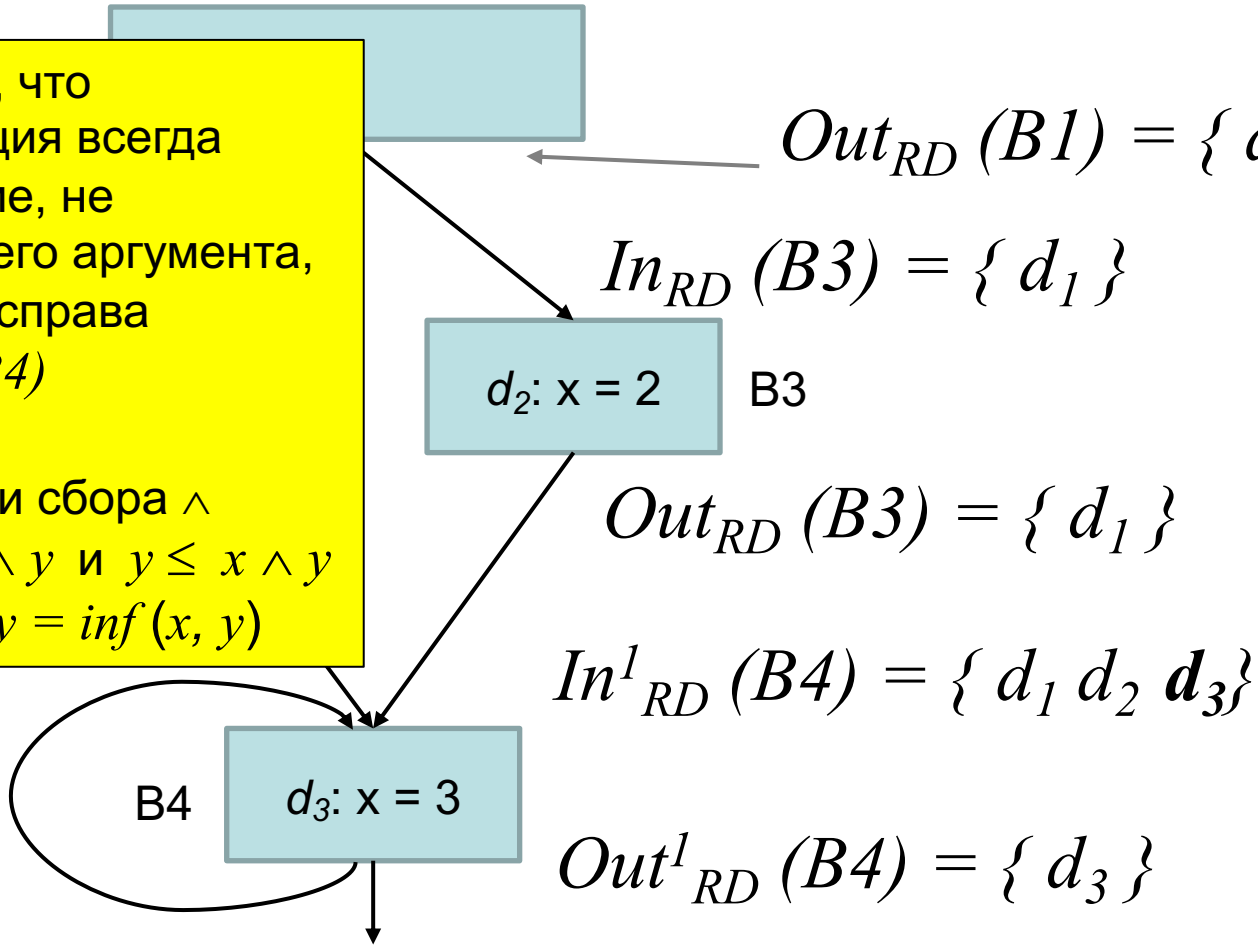
7.1 Глобальное распространение констант

7.1.5 Монотонность структуры распространения констант

Монотонность: $x \le y => f(x) \le f(y)$. Здесь x и y – это состояния в одной и той же точке до/после рассмотрения еще одного пути (например, на разных итерациях).

Но неверно считать, что передаточная функция всегда возвращает значение, не превосходящее своего аргумента, например, для ГПУ справа $Out_{RD}(B4) > In_{RD}(B4)$

Только для операции сбора $\wedge$ выполняется $x \le x \wedge y$ и $y \le x \wedge y$ в силу того, что $x \wedge y = inf(x, y)$



Монотонность: $In^1_{RD}(B4) \le In^0_{RD}(B4) =>$
$Out^1_{RD}(B4) = f_{B4}(In^1_{RD}(B4)) \le f_{B4}(In^0_{RD}(B4)) = Out^0_{RD}(B4)$

7.1 Глобальное распространение констант

7.1.5 Монотонность структуры распространения констант

◊ **Утверждение**. Структура распространения констант монотонна:
если для $m_1,\ m_2,\ m_1', m_2'$ выполняются соотношения:
$m_1 \le m_2,\quad m_1' = f_s(m_1),\ m_2' = f_s(m_2)$,
то $m_1' \le m_2'$

◊ **Доказательство**. Рассмотрим определение f_s через m и m'.

Случай 1): $\boldsymbol{f_s \equiv i} \Rightarrow\ m_1' = m_1, m_2' = m_2 \Rightarrow m_1' \le m_2'$

Случай 2a): $\forall m: f_s(m) = c \Rightarrow m_1' = m_2' = c \Rightarrow m_1' \le m_2'$

Случай 2b): на следующем слайде

Случай 2c): $\forall m : f_s(m) = NAC \Rightarrow m_1' = m_2' = NAC \Rightarrow m_1' \le m_2'$

7.1 Глобальное распространение констант

7.1.5 Монотонность структуры распространения констант

◊ **Доказательство утверждения (случай 2b)**

Для присваивания $x = y + z$ передаточная функция f_s описывается формулой:

$$m'(x) = \begin{cases} m(y) + m(z), & \text{если } m(y) \text{ и } m(z) \text{ константы} \\ NAC, & \text{если } m(y) = NAC \text{ или } m(z) = NAC \\ Undef & \text{в остальных случаях} \end{cases}$$

которой соответствует таблица справа.

Из таблицы видно, что для каждого фиксированного $m(y)$ при уменьшении $m(z)$, значения в колонке $m'(x)$ также не возрастает. В силу симметрии, аналогичные рассуждения справедливы и для $m(y)$ относительно фиксированного $m(z)$.

Таким образом, при невозрастании аргумента значение передаточной функции также не возрастает, откуда следует монотонность.

$m(y)$	$m(z)$	$m'(x)$
$Undef$	$Undef$	$Undef$
	c_2	$Undef$
	NAC	NAC
c_1	$Undef$	$Undef$
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	$Undef$	NAC
	c_2	NAC
	NAC	NAC

7.1 Глобальное распространение констант

7.1.5 Монотонность структуры распространения констант

Операции над элементами полурешетки:

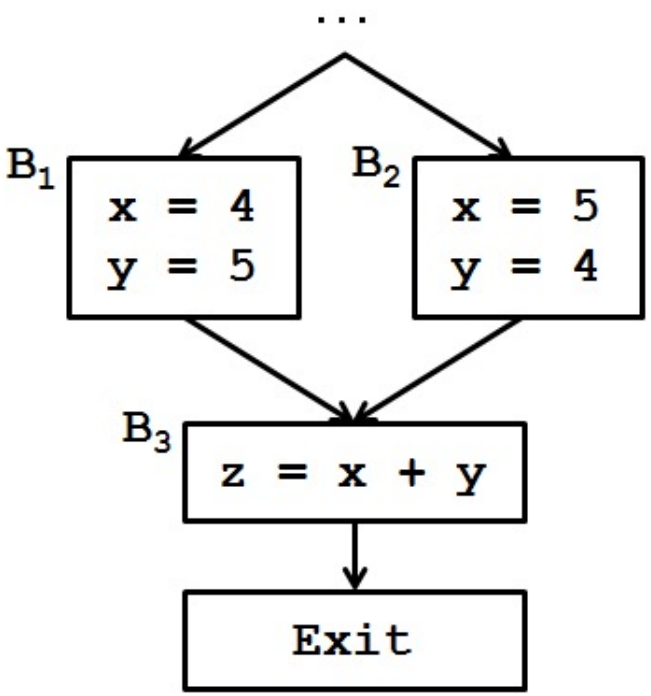
m(x)	m(y)	m(x) ∧ m(y)	m(x) + m(y)
Undef	Undef	Undef	Undef
	C_2	C_2	Undef
	NAC	NAC	NAC
C_1	Undef	C_1	Undef
	C_2	$(C_1 == C_2)\ ?\ C_1 : NAC$	$C_1 + C_2$
	NAC	NAC	NAC
NAC	Undef	NAC	NAC
	C_2	NAC	NAC
	NAC	NAC	NAC

Операция сбора «∧» соответствует объединению состояний по разным путям в точке сбора ГПУ.
Под операцией «+» подразумеваются любые арифметические операции в обобщенном смысле.

7.1 Глобальное распространение констант

7.1.6 Недистрибутивность структуры распространения констант

◊ Для установления отсутствия дистрибутивности у передаточной функции структуры распространения констант рассмотрим пример



Пример.

Итеративный алгоритм не может обнаружить, что в программе на рисунке слева значение **x + y** на входе в блок B_3 всегда равно 9

И **x**, и **y** на входе в B_3 имеют значение NAC

Этот результат безопасен (консервативен), но неточен: не удается отследить зависимость:

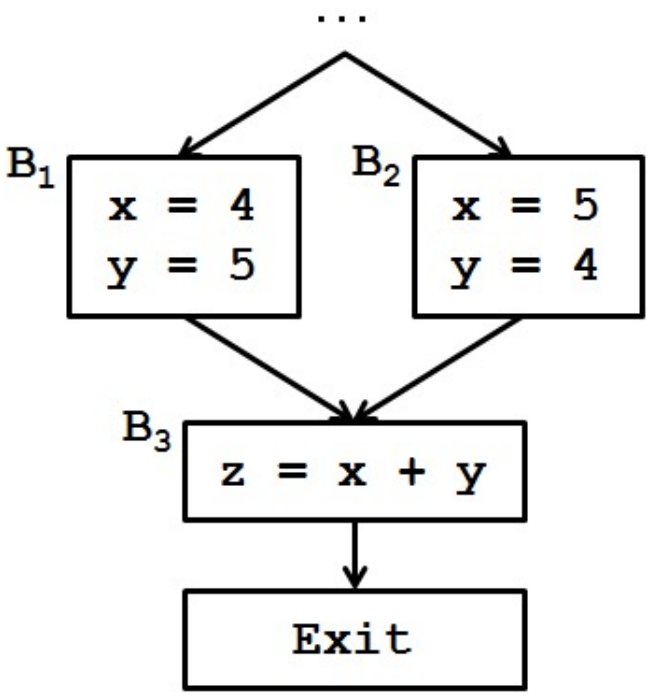
когда **x** равно 4, **y** равно 5,

а когда **x** равно 5, **y** равно 4.

7.1 Глобальное распространение констант

7.1.6 Недистрибутивность структуры распространения констант

◊ Таким образом, структура распространения констант не дистрибутивна, т.е. MFP-решение безопасно, но необязательно равно MOP-решению и может оказаться «меньше» (грубее) него.



Пример.

Итеративный алгоритм не сможет обнаружить, что в программе на рисунке слева значение **x + y** на входе в блок B_3 всегда равно **9**,

так как и **x**, и **y** на входе в B_3 имеют значение **NAC**

Этот результат безопасен (консервативен), но неточен: из-за недистрибутивности структуры распространения констант итеративный алгоритм не может отследить зависимость:

когда **x** равно 4, **y** равно 5,

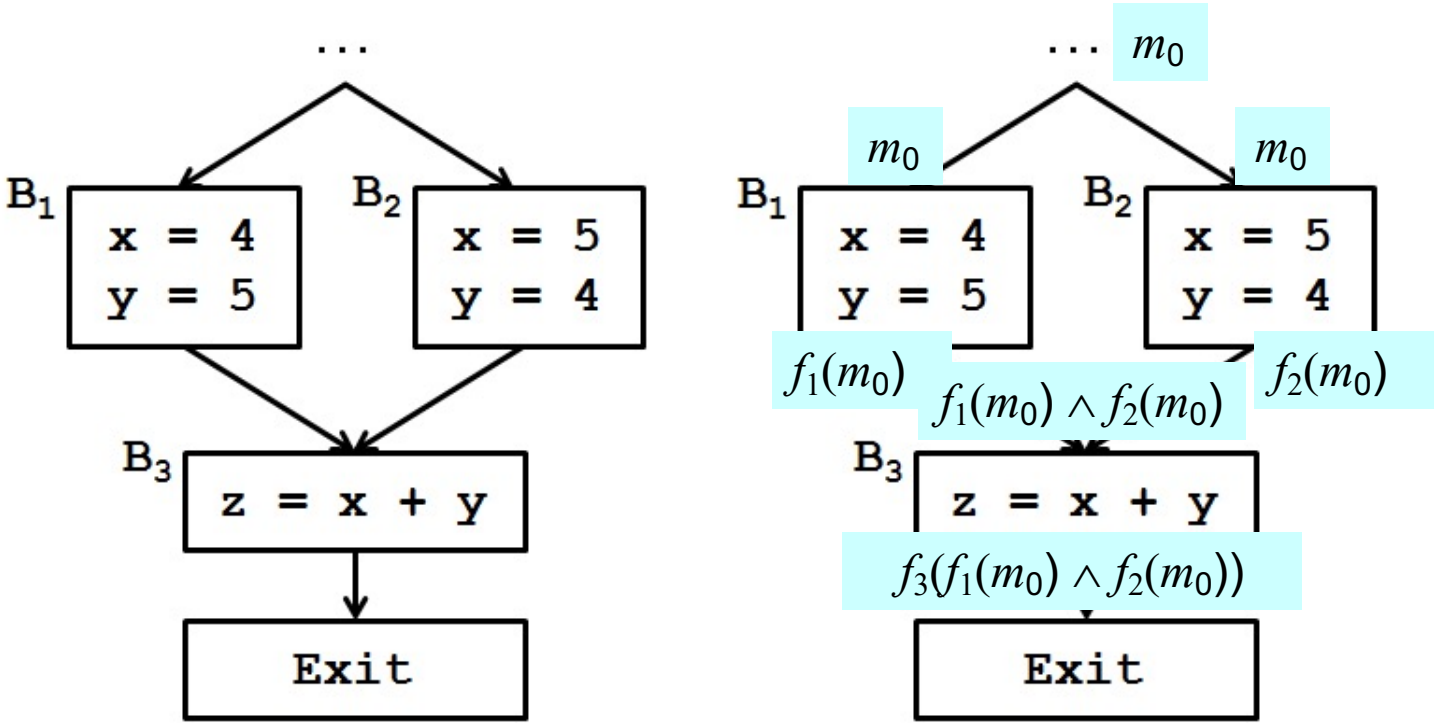
а когда **x** равно 5, **y** равно 4.

7.1 Глобальное распространение констант

7.1.6 Недистрибутивность структуры распространения констант

◊ **Окончание примера.**

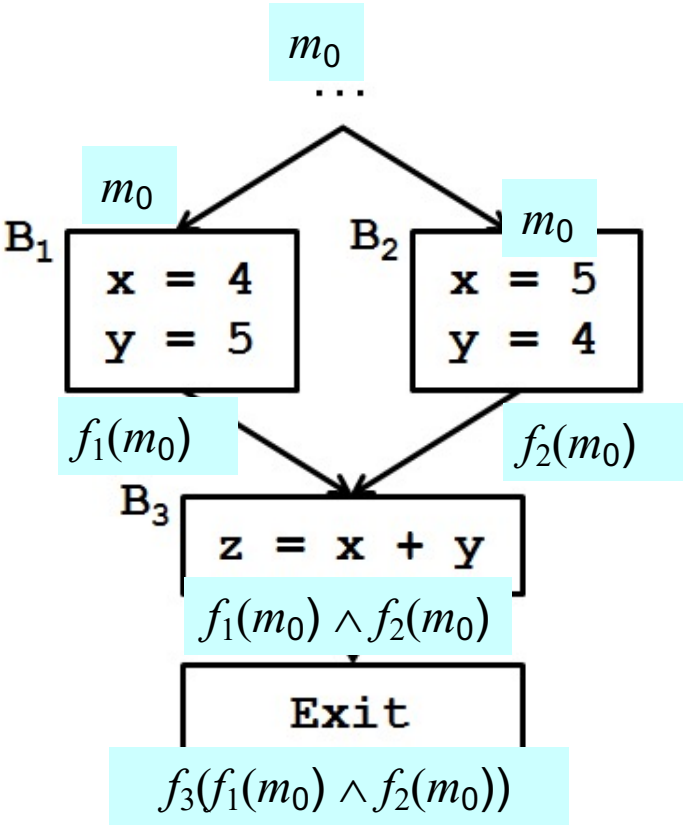
Пусть f_1, f_2 и f_3 – передаточные функции блоков B_1, B_2 и B_3 соответственно, а m_0 – состояние на входе в блоки B_1 и B_2. Тогда



7.1 Глобальное распространение констант

7.1.6 Недистрибутивность структуры распространения констант

◊ **Пример.** Таким образом, как видно из таблицы



m	$m(x)$	$m(y)$	$m(z)$
m_0	*Undef*	*Undef*	*Undef*
$f_1(m_0)$	4	5	*Undef*
$f_2(m_0)$	5	4	*Undef*
$f_1(m_0) \wedge f_2(m_0)$	*NAC*	*NAC*	*Undef*
$f_3(f_1(m_0) \wedge f_2(m_0))$	*NAC*	*NAC*	*NAC*
$f_3(f_1(m_0))$	4	5	9
$f_3(f_2(m_0))$	5	4	9
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	*NAC*	*NAC*	9

$f_3(f_1(m_0) \wedge f_2(m_0)) = NAC$, а $f_3(f_1(m_0)) \wedge f_3(f_2(m_0)) = 9$,

откуда следует **недистрибутивность** структуры:

$$f_3(f_1(m_0) \wedge f_2(m_0)) \neq f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$$

7.1 Глобальное распространение констант

7.1.7 Интерпретация значения *Undef*

◊ Значение *Undef* используется в итеративном алгоритме:

(1) для инициализации входного узла (граничное условие).

(2) для инициализации внутренних точек программы на нулевой итерации

◊ Смысл случаев (1) и (2):

⟐ **Случай (1)**: в начале выполнения программы значения ее переменных не определены.

В конце итеративного процесса переменные на выходе из *Entry* сохраняют значение *Undef*, поскольку *Out*[*Entry*] не изменяется.

7.1 Глобальное распространение констант

7.1.7 Интерпретация значения *Undef*

◊ Смысл случаев (1) и (2):

- ◈ **Случай (2):** из-за недостатка информации в начале итерационного процесса решение апроксимируется верхним элементом *Undef*.

  В конце итерационного процесса эти *Undef* сохранятся только в тех точках, для которых не найдется определений соответствующих переменных ни на одном пути, ведущем к этим точкам (либо в правой части этих определений в свою очередь используется неинициализированная переменная).

  Если же найдется хотя бы один путь, содержащий определение переменной (в котором все используемые переменные инициализированы), достигающее рассматриваемой точки программы, то эта переменная не будет иметь значение *Undef*.

7.1 Глобальное распространение констант

7.1.7 Интерпретация значения *Undef*

◊ Если все определения какой-либо переменной, достигающие некоторой точки программы, имеют одно и то же константное значение (но не $Undef$ и не NAC), то эта переменная рассматривается как константа, **даже если может оказаться, что она не определена на одном из путей**.

◊ Если предположить, что оптимизируемая программа корректна, то итеративный алгоритм может найти большее количество констант, т.к. он может считать константами и те переменные, которые на части путей имеют значения $Undef$.

Такой подход правомерен для языков программирования, которые исходят из того, что неопределенная ($Undef$) переменная может иметь любое значение.

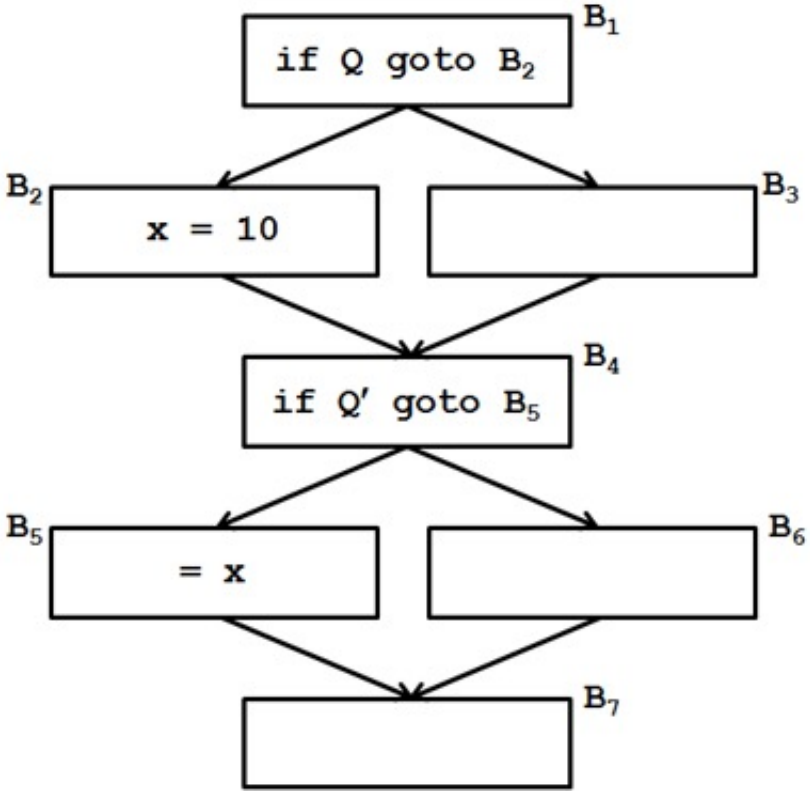
Если семантика языка программирования требует, чтобы все неопределенные переменные принимали некоторое конкретное значение, необходимо это ограничение включить в формулировку задачи анализа потока данных.

7.1 Глобальное распространение констант

7.1.7 Интерпретация значения *Undef*

◊ На рисунке справа значения x на выходе из блоков B_2 и B_3 – соответственно 10 и $Undef$. Из $Undef \wedge 10 = 10$ следует, что значение x на входе в B_4 равно 10, и в B_5 можно заменить x на 10.

◊ Если выполнение пойдет по пути $B_1 \to B_3 \to B_4 \to B_5$, то блока B_5 будет достигать значение $x = Undef$, и замена x на 10 будет корректной не для всех исходных языков.

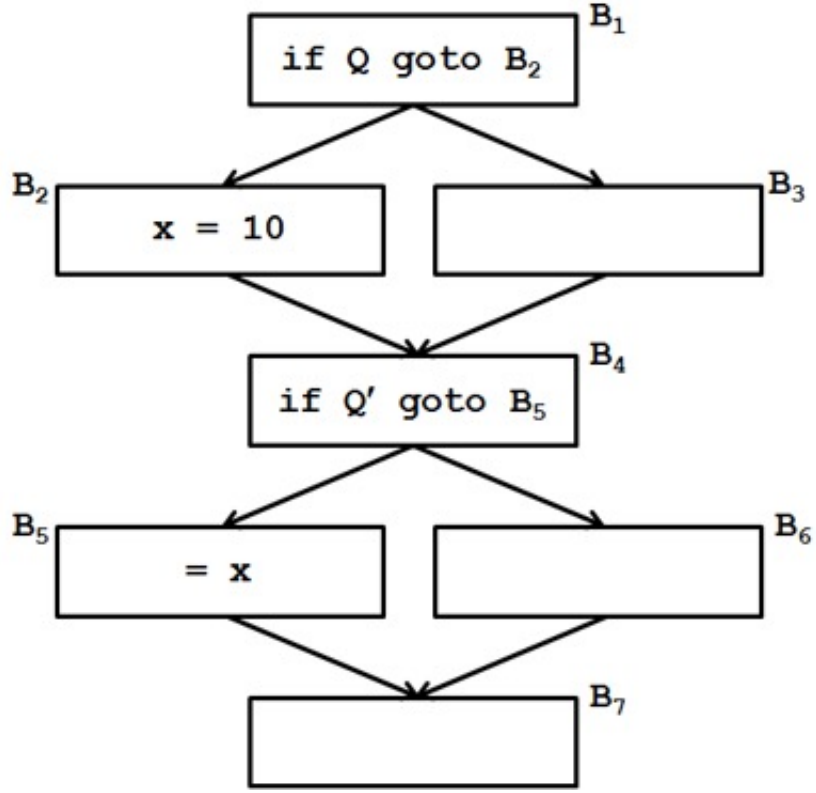


7.1 Глобальное распространение констант

7.1.7 Интерпретация значения *Undef*

Пример.

◊ Даже если можно доказать, что предикат Q не может иметь значение $false$, когда предикат Q' имеет значение $true$, и поэтому путь $B_1 \to B_3 \to B_4 \to B_5$ никогда не будет пройден, следует понимать, что **такой вывод находится за пределами возможностей анализа потоков данных.**



◊ Таким образом, если считать, что исходная программа корректна и что все ее переменные определяются до их использования, то корректно считать, что x при входе в B_5 определена, и ее значение равно 10.

7.1 Глобальное распространение констант

7.1.7 Интерпретация значения *Undef*

В операции сбора:

$\forall c$: $Undef \wedge c = c$,

При этом при вычислении констант в выражениях:

$c + Undef = Undef$

В чем смысл таких определений?

```
void foo(bool p) {
  int z, x, y = 20; // x = UNDEF

  if (p) {
    x = 10;
    z = x + y; // z = 30
  } else {
    z = x + y;
    // UNDEF + 20 = UNDEF
  }
}

// 30 ∧ UNDEF = 30
printf("%d\n", z); // <- z = 30
```

В операции сбора компилятор исходит из того, что конкретное значение «30» не хуже любого другого неинициализированного значения, которое представляет «UNDEF», а в сложении – что при выполнении арифметической операции над неопределенным значением оно остается неопределенным, но сохраняет шанс стать константой при сборе с равным значением.

В данном случае удается выполнить распространение констант вдоль одной ветви.

7.2 Алгоритм глобального распространения констант

7.2.1 Псевдокод алгоритма

- На вход алгоритма поступает SSA-граф G программы
- Алгоритм завершает работу, когда $worklist$ становится пустым
- $Value(v)$ – возвращает значение ячейки, связанной с SSA-именем v
- $Evaluate(v)$ интерпретирует выражение SSA-имени v в соответствии с передаточной функцией, если выражение является φ-функцией, то применяется оператор сбора для операндов.
- $Evaluate(v)$ возвращает $\top$ ($Undef$), константу или $\bot$ (NAC)
- Алгоритм состоит из двух этапов: этап инициализации и этап распространения констант

7.2 Алгоритм глобального распространения констант

7.2.1 Фаза инициализации

```
// Initialization Phase
WorkList = ∅;
for each SSA-name v {
     initialize Value(v) by rules specified in
the text;
     if Value(v) ≠ Undef then
          WorkList = WorkList ∪ {v};
}
```

На фазе инициализации устанавливаются начальные значения SSA-имен и формируется начальное состояние **`WorkList`**.

Для каждого SSA-имени **`v`** анализируется операция, определяющая **`v`** и устанавливающая значение **`Value(v)`** в соответствии со следующими простыми правилами (следующий слайд).

7.2 Алгоритм глобального распространения констант

7.2.1 Фаза инициализации

```
// Initialization Phase
WorkList = ∅;
for each SSA-name v
initialize Value(v) by rules specified in the
text;
if Value(v) ≠ Undef then
WorkList   WorkList ∪ {v};
```

"The rules specified in the text":

1. **v** определяется φ-функцией, **Value(v) = Undef**.
2. **v** равно одной из констант c_i, **Value(v) = c_i**.
3. **v** может быть присвоено входное значение, **Value(v) = NAC**.
4. Значение **v** неизвестно, **Value(v)** = **Undef**.

Если **Value(v) ≠ Undef**, алгоритм добавляет **v** в **WorkList**

7.2 Алгоритм глобального распространения констант

7.2.2 Фаза распространения

```
// Propagation Phase
// Выполнять итерации (до неподвижной точки)
while (WorkList ≠ ∅) {
  remove some v from WorkList
  // Выбрать произвольное имя
  for each operation op that uses v {
    // op: w = use v
    let w be the SSA-name that op defines
    if Value(w) ≠ NAC then {
      // Recompute and test for change
      t ← Value(w)
      // Evaluate by interpreting op over lattice values
      Value(w) ← Evaluate(w)
      if Value(w) ≠ t then
        WorkList ← WorkList ∪ {w};
    }
  }
}
```

Evaluating a Φ-node:

$\Phi(x_1, x_2, x_3, \ldots x_n)$ is

$\text{Value}(x_1) \wedge \text{Value}(x_2)\ \text{Value}(x_3) \wedge \ldots \wedge \text{Value}(x_n)$

Where

$\text{TOP} \wedge x = x$	$\forall\ x$
$c_i \wedge c_j = c_i$	if $c_i = c_j$
$c_i \wedge c_j = \text{BOT}$	if $c_i \neq c_j$
$\text{BOT} \wedge x = \text{BOT}$	$\forall\ x$

7.2 Алгоритм глобального распространения констант

7.2.2 Фаза распространения

Фаза распространения проста:

Из **`WorkList`** извлекается (с удалением) произвольное SSA-имя **`v`**.

Алгоритм анализирует каждую операцию **`op`**,

которая использует **`v`**

и определяет SSA-имя **`w`**.

Если **`Value(w)`** == **`NAC`**, дальнейший анализ **`w`** не имеет смысла.

В противном случае, моделируется выполнение **`op`** с помощью интерпретации операции над полурешетками значений ее операндов.

Если результат отличен от **`Value(w)`** , он рассматривается как новое значение **`Value(w)`** , причем **`w`** добавляется в **`WorkList`**.

Алгоритм завершается, когда **`WorkList`** пуст.

7.2 Алгоритм глобального распространения констант

7.2.2 Фаза распространения

Интерпретация операции над значениями полурешетки требует некоторой осторожности.
Для φ-функции результаты операции получаются применением операции «сбор» ко всем аргументам φ-функции; правила вычисления результатов операции «сбор» на рисунке справа (в порядке предпочтения)

Правила для операции $\wedge$

$\top \wedge x = x \quad \forall x$

$\bot \wedge x = \bot \quad \forall x$

$c_i \wedge c_j = c_i \quad c_i = c_j$

$c_i \wedge c_j = \bot \quad c_i = c_j$

Что касается других операций компилятор должен «знать» особенности их выполнения. Если любой операнд имеет значение T, необходимо вернуть результат T. Если ни один из операндов не имеет значения T, необходимо вернуть соответствующее значение.
Для каждой операции, вычисляющей значение, алгоритму необходимо иметь набор правил, моделирующих поведение операндов.
Рассмотрим, например операцию $a \times b$. Если a = 4, а b = 17, необходимо выдать в качестве результата $a \times b$ значение 68. Однако, если $a = \bot$, необходимо вернуть результат $\bot$ для всех значений b, кроме b = 0, потому что $a \times 0 = 0$ независимо от значения a.

7.2 Алгоритм глобального распространения констант

7.2.3 Заключительные замечания

◊ **Сложность.**

Фаза распространения – это классическая схема нахождения фиксированной точки. Доказательство сходимости и завершимости, а также оценка сложности алгоритма опирается на факт конечности нисходящих цепочек.

Переменная, связанная с каждым SSA-именем может иметь одно из трех начальных значений: какая-нибудь константа c_i, отличная от $Undef$, или $Undef$, или NAC.

Фаза распространения может только «уменьшить» ее значение. Для данного SSA-имени это может произойти не более, чем дважды: от $Undef$ к c_i и от c_i к NAC. Следовательно, каждое SSA-имя может встретиться в $Worklist$ не более, чем два раза: алгоритм интерпретирует операцию, когда один из операндов удаляется из $Worklist$. Так что общее число интерпретаций не больше, чем удвоенное число использований переменных (uses).

7.2 Алгоритм глобального распространения констант

7.2.3 Заключительные замечания

◊ **Ценность SSA-формы**

SSA-форма позволяет разработать простой, эффективный и понятный алгоритм глобального распространения констант.

Рассмотрим для сравнения классический подход к проблеме распространения констант методом анализа потока данных. Для его разработки потребуется рассмотреть множество Constants в каждом базовом блоке B_i, составить систему уравнений относительно неизвестных $In[B_i]$, определить передаточную функцию для вычисления $Out[B_i]$. Несомненно представленный выше алгоритм гораздо проще. Наибольшую сложность вызывает разработка традиционного механизма интерпретации операций, но в остальном это простой итерационный алгоритм нахождения неподвижной точки над чрезвычайно разреженной решеткой.

Алгоритм работает гораздо быстрее традиционного анализа потока данных.

7.2 Алгоритм глобального распространения констант

7.2.4 Алгоритмы распространения констант и отношения между нами

1. Simple Constant (SC). Предложен Гэри Килдаллом (Gary Kildall) в 1973 году. Первая работа, в которой описана задача потока данных для распространения констант. В работе предложен итеративный алгоритм распространения констант[1]

2. Sparse Simple Constant (SSC). Предложен Джоном Райфом и Харри Льюисом (John Henry Reif and Harry Roy Lewis) в 1977 году. Алгоритм использует недавно предложенную SSA форму. Позволяет найти те же константы, что и предыдущий алгоритм, но за меньшее время[2]

[1] Gary A. Kildall. **A Unified Approach to Global Program Optimization**.1973

[2] John H. Reif, Harry R. Lewis. **Symbolic evaluation and the global value graph.** January 1977.

7.2 Алгоритм глобального распространения констант

7.2.4 Алгоритмы распространения констант и отношения между нами

3. Conditional Constant (CC). Разновидность алгоритма, предложенного Бэном Уэгбрейтом (Ben Wegbreit) в 1975 году. Алгоритм способен найти все константы, обнаруживаемые предыдущими алгоритмами (простые константы), а также условные константы [1]

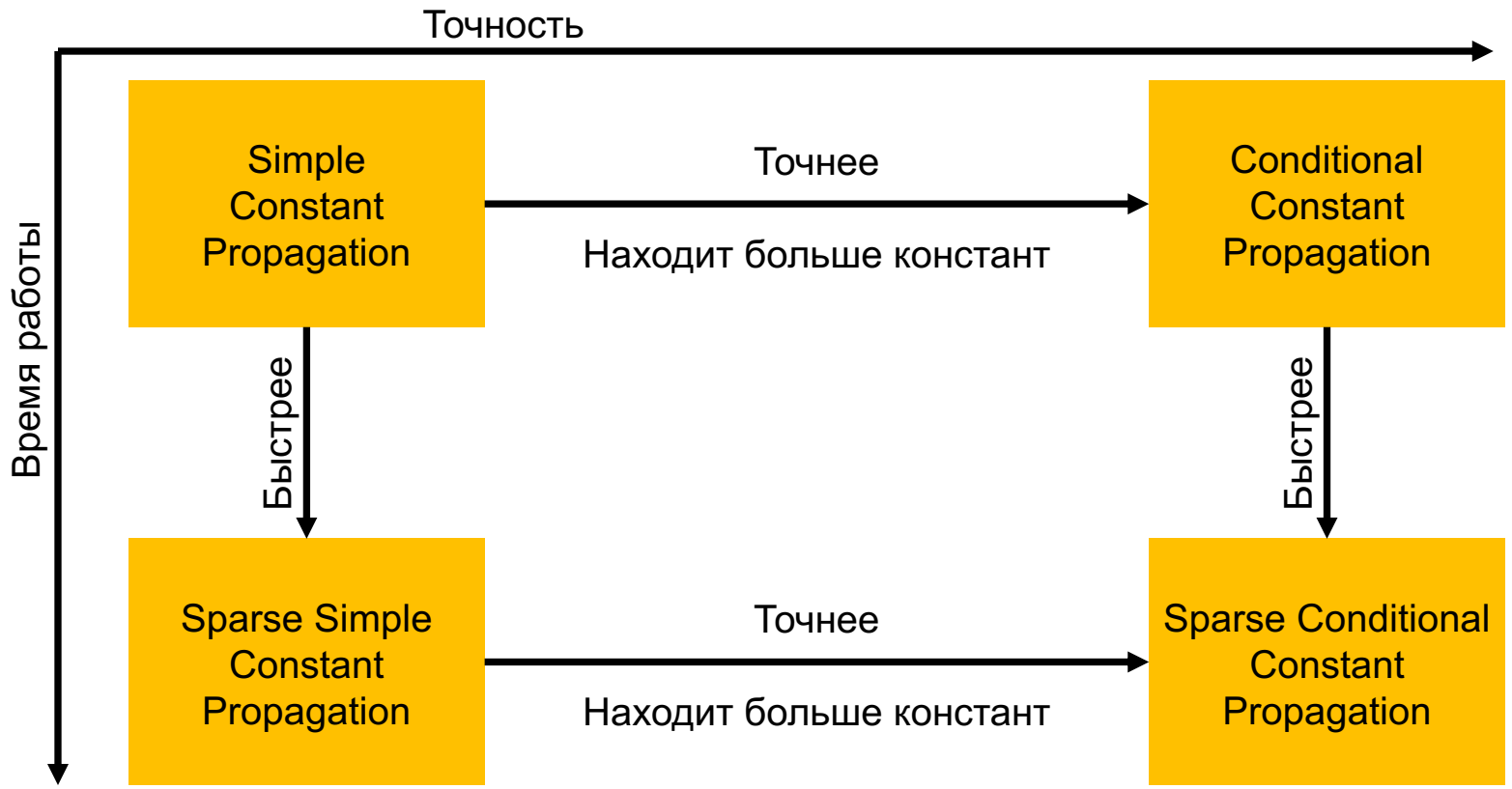
4. Sparse Conditional Constant (SCC). Предложен Марком Уэгменом и Кеном Задеком (Mark Wegman and Ken Zadeck) в 1991 году. Алгоритм способен найти простые константы и некоторые условные константы. Алгоритм выполняется быстрее, чем предыдущий за счет использования SSA-представления [2]

[1] Ben Wegbreit. **Property extraction in well-founded property sets**. Sept. 1975

[2] Mark N. Wegman, Kenneth Zadeck. **Constant Propagation with Conditional Branches**. April 1991

7.2 Алгоритм глобального распространения констант

7.2.4 Алгоритмы распространения констант и отношения между нами



7.2 Алгоритм глобального распространения констант

7.2.4 Алгоритмы распространения констант и отношения между нами