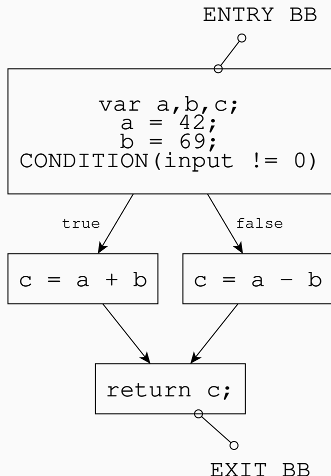
Альманах потокочувствительных анализов

Марат Ахин Михаил Беляев

6 марта 2018 г.

В предыдущей серии

```
var a,b,c;
a = 42;
b = 69;
if (input) {
  c = a + b;
} else {
  c = a - b;
}
return c;
```



В предыдущей серии

Монотонный фреймворк

- CFG для анализируемой программы в виде вершин $Nodes = \{v_1, \ldots, v_n\}$
- Решетка конечной длины L, представляющая интересующий нас абстрактный домен
- Набор переменных $[\![v_i]\!] \in L$ для каждой вершины CFG
- Ограничения для различных видов узлов CFG
  - Определяют значение $[\![v_i]\!]$ через другие переменные
  - Часто учитывают структуру CFG
  - Должны быть монотонными

Анализы на основе решёток в реальности

(где реальность = компиляторы)

- Live variables
- Available expressions
- (Very) busy expressions
- Reaching definitions
- Constant propagation

(aka Анализ живости)

Живые переменные

Переменная называется **живой** в точке программы, если её ещё будут читать в этой или последующих точках программы

Зачем это нужно?

«Мёртвые» переменные можно игнорировать во многих анализах и оптимизациях, что значительно уменьшает размерность задачи

Очевидно, что задача поиска всех живых переменных неразрешима, но её можно аппроксимировать.

Как аппроксимировать?

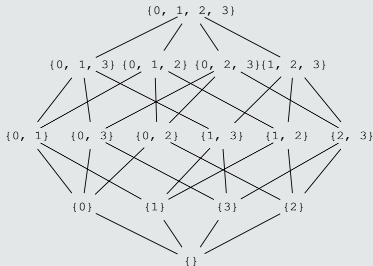
Поскольку нас интересуют в первую очередь мёртвые переменные, то нужно найти те, которые *совершенно точно мертвы*.

Какую решётку будем брать?

В предыдущей серии

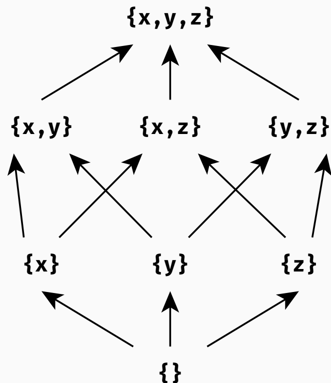
Булеан aka $\mathcal{P}(S)$ lattice

- Решетка над множеством всех подмножеств S
- $\top = S$
- $\bot = \emptyset$
- $x \sqcup y = x \cup y$
- $x \sqcap y = x \cap y$

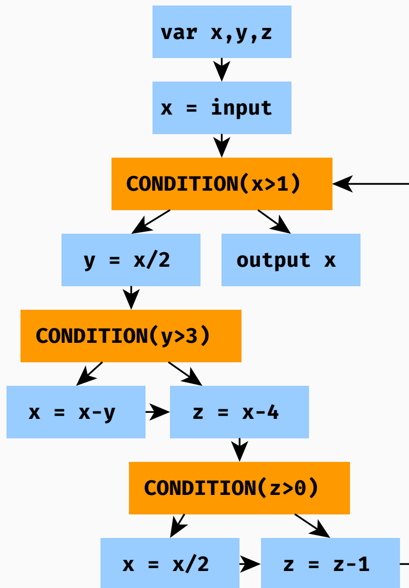


Рассмотрим булеан над множеством всех переменных программы

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>4) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```



Exploded CFG



Правила вывода (дубль 2)

Нас интересует $\sigma : [\![n]\!]$ *— абстрактное состояние программы после вершины* n *из CFG, которое является* решеткой отображений *из переменных программы в наш абстрактный знаковый домен*

$$[\![\text{var } x;]\!] = JOIN(n) \setminus \{x\}$$

$$[\![x = E;]\!] = JOIN(n) \setminus \{x\} \cup vars(E)$$

$$[\![n]\!] = JOIN(n) \cup vars(n) \quad \text{для любых других } n$$

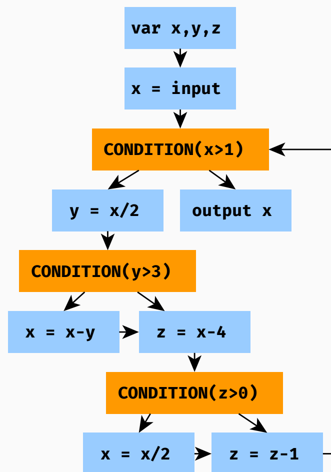
$vars(E)$ — все переменные, используемые в E

$vars(n)$ — все переменные, используемые во всех выражениях в n

$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$

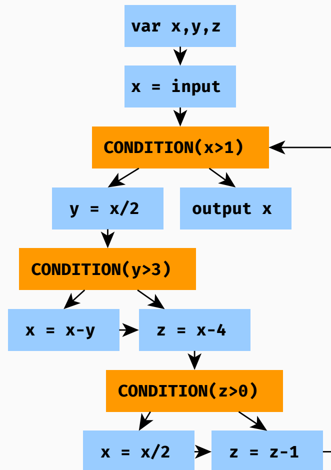
Итоговые ограничения

$$[\![\text{var x,y,z}]\!] = [\![\text{x = input}]\!] \setminus \{x, y, z\}$$
$$[\![\text{x = input}]\!] = [\![\text{x > 1}]\!] \setminus \{x\}$$
$$[\![\text{x > 1}]\!] = [\![\text{y = x/2}]\!] \cup [\![\text{output x}]\!] \cup \{x\}$$
$$[\![\text{y = x/2}]\!] = [\![\text{y > 3}]\!] \setminus \{y\} \cup \{x\}$$
$$[\![\text{y > 3}]\!] = [\![\text{x = x-y}]\!] \cup [\![\text{z = x-4}]\!] \cup \{y\}$$
$$[\![\text{x = x-y}]\!] = [\![\text{z = x-4}]\!] \setminus \{x\} \cup \{x, y\}$$
$$[\![\text{z = x-4}]\!] = [\![\text{z > 0}]\!] \setminus \{z\} \cup \{x\}$$
$$[\![\text{z > 0}]\!] = [\![\text{x = x/2}]\!] \cup [\![\text{z = z-1}]\!] \cup \{z\}$$
$$[\![\text{x = x/2}]\!] = [\![\text{z = z-1}]\!] \setminus \{x\} \cup \{x\}$$
$$[\![\text{z = z-1}]\!] = [\![\text{x > 1}]\!] \setminus \{z\} \cup \{z\}$$
$$[\![\text{output x}]\!] = \{x\}$$



Решение

$$[\![\text{var x,y,z}]\!] = \{\}$$
$$[\![\text{x = input}]\!] = \{\}$$
$$[\![\text{x > 1}]\!] = \{x\}$$
$$[\![\text{y = x/2}]\!] = \{x\}$$
$$[\![\text{y > 3}]\!] = \{x, y\}$$
$$[\![\text{x = x-y}]\!] = \{x, y\}$$
$$[\![\text{z = x-4}]\!] = \{x\}$$
$$[\![\text{z > 0}]\!] = \{x, z\}$$
$$[\![\text{x = x/2}]\!] = \{x, z\}$$
$$[\![\text{z = z-1}]\!] = \{x, z\}$$
$$[\![\text{output x}]\!] = \{x\}$$



Чего мы узнали?

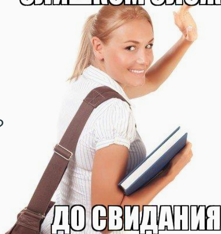
- `y` и `z` никогда не бывают живы одновременно
  - Можно разместить их в одном регистре
- Результат `z = z - 1` никогда не используется
  - Можно удалить этот код

Для наивного алгоритма:

- Пусть n — число нод в CFG, k — число переменных
- Высота решётки равна k, общее число итераций $k \times n$
- Каждая операция над подмножеством размера k — $O(k)$
- На каждой итерации таких операций $O(n)$
- Итого по времени: $O(k^2 \times n^2)$

На подумать:

- Какова сложность структурного алгоритма?
- А если в CFG нет циклов?
- Какова сложность по памяти?



Алгоритмическая сложность: а что делать?

- В реальности сложность редко доходит до таких высоких значений
- Но всё упирается в эффективные реализации $\cup$ и $\cap$ для множеств
    - Spoiler: их не существует
    - Можно использовать битовые множества
    - Можно использовать персистентные структуры данных

Available expressions

(aka Анализ готовых выражений)

Готовые выражения

Выражение называется **готовым** в какой-то точке программы, если до этой точки его гарантированно уже считали

Зачем это нужно?

Готовые выражения можно не пересчитывать

Очевидно, что задача поиска всех готовых выражений неразрешима, но её можно аппроксимировать.

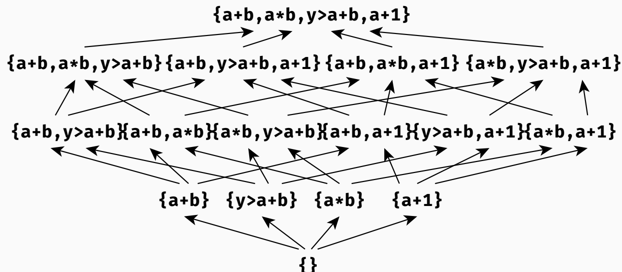
Как аппроксимировать?

Нам интересны только те выражения, которые уже точно считались, если мы что-то пропустим, не беда.

Какую решётку будем брать?

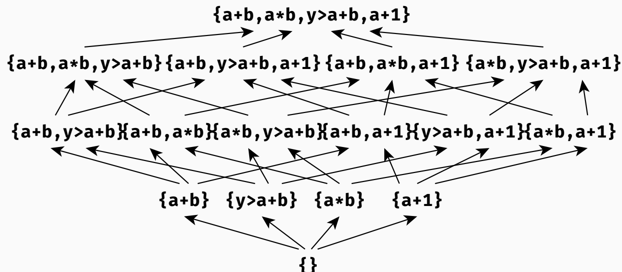
Рассмотрим булеан над множеством всех сложных выражений

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```



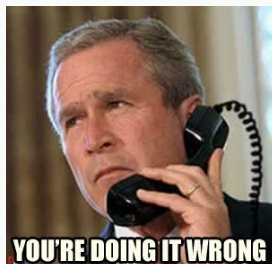
Рассмотрим булеан над множеством всех сложных выражений

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```



Всё правильно?

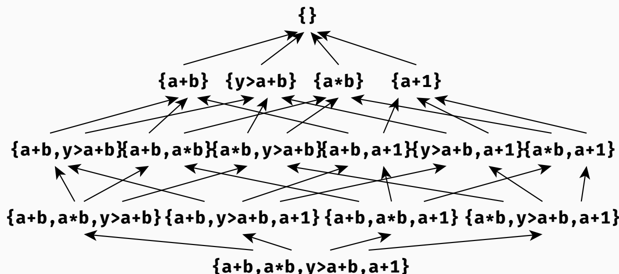
Нет, неправильно!



Тривиальное решение для нас — это когда никакие выражения не готовы. Следовательно, пустое множество должно быть сверху!

Нам нужен *перевёрнутый булеан* над сложными выражениями

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
return x
```



Вообще, любую решётку можно перевернуть, и она останется решёткой.

Правда, монотонную операцию $\sqcup$ обычно нужно определять заново.

Для булеана всё тривиально: $x \sqcup y = x \cap y$

$$[\![x = E;]\!] = \mathit{removerefs}(\mathit{JOIN}(n) \cup \mathit{exprs}(n), x)$$

$$[\![n]\!] = \mathit{JOIN}(n) \cup \mathit{exprs}(n) \quad \text{для любых других } n$$

$\mathit{exprs}(E)$ — все нетривиальные выражения в E, не содержащие `input`
$\mathit{exprs}(N) - \bigcup \mathit{exprs}(E)$ для всех E в N
$\mathit{removerefs}(S, x) = \{s \in S \mid x \notin \mathit{vars}(s)\}$
$\mathit{JOIN}(v) = \bigcap_{w \in \mathit{pred}(v)} [\![w]\!]$

Итоговые ограничения

$$[\![\text{var x,y,z,a,b}]\!] = \{\}$$

$$[\![\text{z = a+b}]\!] = \mathit{removerefs}(\mathit{exprs}(a + b), z)$$

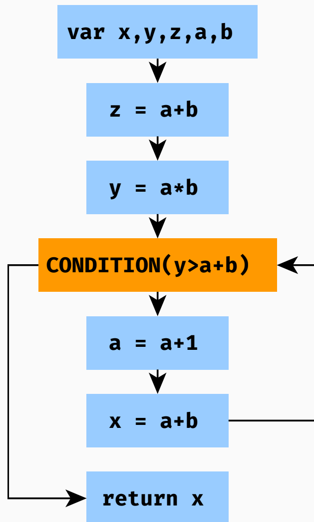
$$[\![\text{y = a*b}]\!] = \mathit{removerefs}([\![\text{z = a+b}]\!] \cup \mathit{exprs}(\text{a*b}), y)$$

$$[\![\text{y>a+b}]\!] = ([\![\text{y = a*b}]\!] \cap [\![\text{x = a+b}]\!]) \cup \mathit{exprs}(\text{y>a+b})$$

$$[\![\text{a = a+1}]\!] = \mathit{removerefs}([\![\text{y>a+b}]\!] \cup \mathit{exprs}(\text{a+1}), a)$$

$$[\![\text{x = a+b}]\!] = \mathit{removerefs}([\![\text{a = a+1}]\!] \cup \mathit{exprs}(\text{a+b}), x)$$

$$[\![\text{return x}]\!] = [\![\text{y>a+b}]\!] \cup \mathit{exprs}(x)$$



Итоговые ограничения

$$[\![\text{var x,y,z,a,b}]\!] = \{\}$$

$$[\![\text{z = a+b}]\!] = \mathit{removerefs}(\{\text{a+b}\}, z)$$

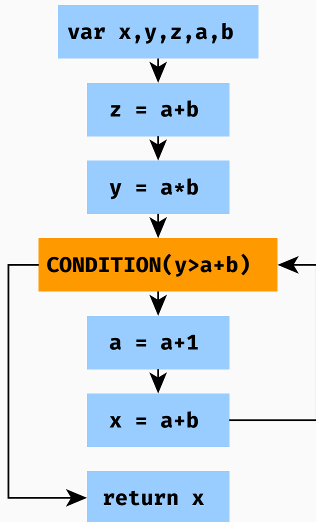
$$[\![\text{y = a*b}]\!] = \mathit{removerefs}([\![\text{z = a+b}]\!] \cup \{\text{a*b}\}, y)$$

$$[\![\text{y>a+b}]\!] = ([\![\text{y = a*b}]\!] \cap [\![\text{x = a+b}]\!]) \cup \{\text{a+b, y>a+b}\}$$

$$[\![\text{a = a+1}]\!] = \mathit{removerefs}([\![\text{y>a+b}]\!] \cup \{\text{a+1}\}, a)$$

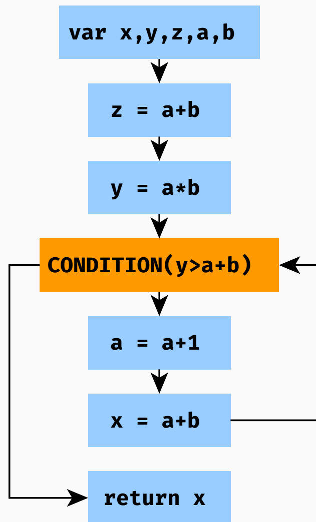
$$[\![\text{x = a+b}]\!] = \mathit{removerefs}([\![\text{a = a+1}]\!] \cup \{\text{a+b}\}, x)$$

$$[\![\text{return x}]\!] = [\![\text{y>a+b}]\!] \cup \{\}$$



Решение

$$[\![\text{var x,y,z,a,b}]\!] = \{\}$$
$$[\![\text{z = a+b}]\!] = \{\text{a+b}\}$$
$$[\![\text{y = a*b}]\!] = \{\text{a+b, a*b}\}$$
$$[\![\text{y>a+b}]\!] = \{\text{y>a+b, a+b}\}$$
$$[\![\text{a = a+1}]\!] = \{\}$$
$$[\![\text{x = a+b}]\!] = \{\text{a+b}\}$$
$$[\![\text{return x}]\!] = \{\text{y>a+b, a+b}\}$$



Чего мы узнали?

- `a+b` всегда считается до проверки условия цикла
    - Можно в проверке его заново не считать

(aka Анализ занятости)

Очень занятые выражения

Выражение называется **очень занятым** в некоей точке программы, если до этой точки оно гарантированно было использовано

Зачем это нужно?

- Очень занятые выражения можно предрассчитывать, вынося их из условных конструкций и циклов
- В языках с ленивой семантикой выполнения их можно предрассчитывать энергично, снижая таким образом побочные эффекты ленивой семантики

Очевидно, что задача поиска всех очень занятых выражений неразрешима, но её можно аппроксимировать.

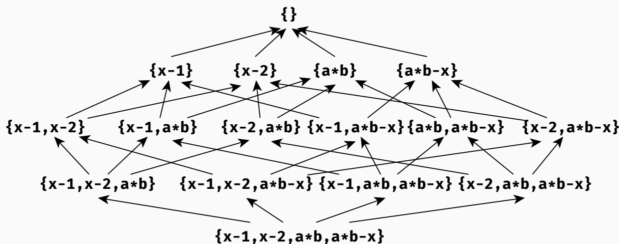
Как аппроксимировать?

Нам интересны только те выражения, которые **точно** будут использованы, если мы что-то пропустим, не беда.

Какую решётку будем брать?

Да ту же самую

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x > 0) {
  output a*b-x;
  x = x-1;
}
output a*b;
```



$$[\![x = E;]\!] = \mathit{removerefs}(\mathit{JOIN}(n), x) \cup \mathit{exprs}(n)$$

$$[\![n]\!] = \mathit{JOIN}(n) \cup \mathit{exprs}(n) \quad \text{для любых других } n$$

$\mathit{exprs}(E)$ — все нетривиальные выражения в E, не содержащие `input`

$\mathit{exprs}(N) - \bigcup \mathit{exprs}(E)$ для всех E в N

$\mathit{removerefs}(S, x) = \{s \in S \mid x \notin \mathit{vars}(s)\}$

$\mathit{JOIN}(v) = \bigcap_{w \in \mathit{succ}(v)} [\![\mathrm{w}]\!]$

На подумать: попробуйте расписать систему ограничений для примера выше и решить её

Reaching definitions

Текущие определения

Набор присваиваний/объявлений, отвечающих за текущие значения переменных в данной точке

Зачем это нужно?

Чтобы определять, что значит конкретное имя в конкретном выражении

Ну, про неразрешимость вы уже поняли.

Как аппроксимировать?

Нам интересен наиболее полный набор определений, возможно, среди них будут лишние.

Какую решётку будем брать?

```
var x,y,z;
x = input;
while (x > 1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

{x=input,y=x/2,x=x-y,
z=x-4,x=x/2,z=z-1}

{...} ... {...}

(тут могла бы быть ваша реклама)

{...} ... {...}

{}

$$[\![\text{var } x;]\!] = \{\text{var } x\}$$

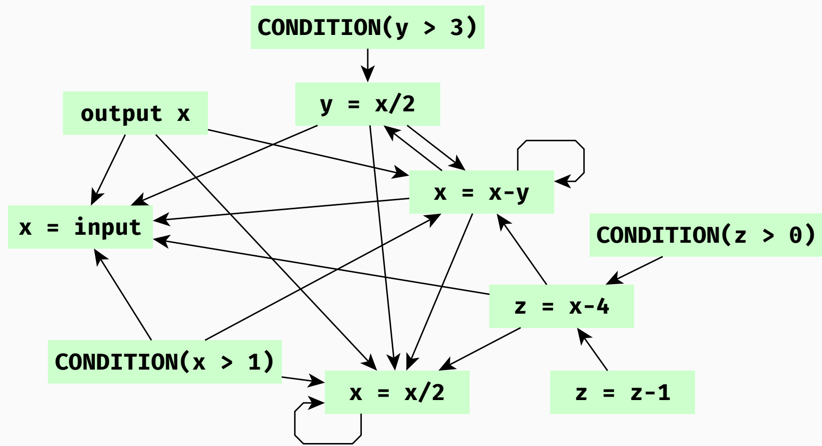
$$[\![x = E;]\!] = removedefs(JOIN(n), x) \cup \{x = E\}$$

$$[\![n]\!] = JOIN(n) \quad \text{для любых других } n$$

$removerefs(S, x)$ — все элементы S, кроме определений x

$JOIN(v) = \bigcup\limits_{w \in pred(v)} [\![\mathrm{w}]\!]$

- Можно построить по результатам RD
- Часто используется как дополнение к CFG



Forward vs backward analysis

Forward analysis

- Строится из прошлого к будущему
- Уравнения для каждой ноды строятся на основе **предыдущих** нод
- Примеры: available expressions, reaching definitions

Backward analysis

- Строится из будущего в прошлое
- Уравнения для каждой ноды строятся на основе **последующих** нод
- Примеры: very busy expressions, live variables

May vs Must analysis

May analysis

- Результат **возможно** является истинным, но точно полным
- Для множеств имеет место переаппроксимация
- Примеры: reaching definitions, live variables

Must analysis

- Результат **точно** является истинным, но возможно неполным
- Для множеств имеет место недоаппроксимация
- Примеры: available expressions, very busy expressions

	May	Must
Forward	Reaching defs	Available expressions
	$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$	$JOIN(v) = \bigcap_{w \in pred(v)} [\![w]\!]$
Backward	Live variables	Busy expressions
	$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$	$JOIN(v) = \bigcap_{w \in succ(v)} [\![w]\!]$

Disclaimer

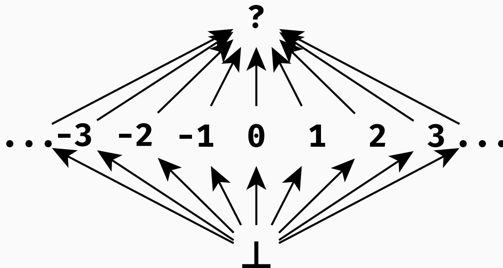
- Бывают ли анализы не forward/backward?
  - Да ещё как!
  - Никто не мешает построить анализ, работающий в обе стороны
  - Но попробуйте написать для него эффективный решатель =)
- Бывают ли анализы не may/must?
  - Да ещё как!
  - Никто не мешает построить анализ, строящий что-то среднее
  - Как правило, это говорит о недоопределённости ограничений
  - Типично для анализов на основе ML или чего-то похожего

- Один из самых полезных анализов в приложении к компиляторам
- Определяет, что отдельные значения являются константами и могут быть вычислены на стадии компиляции

Какую решётку будем использовать?

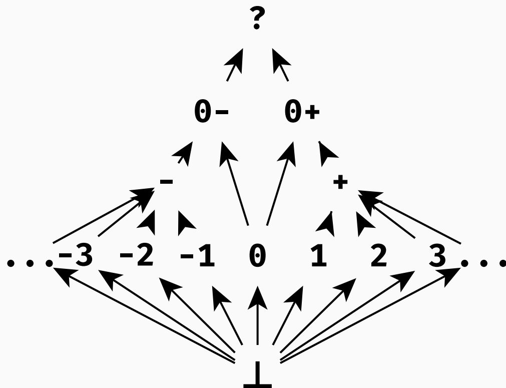
Решётка отображений $var \longrightarrow int$ где

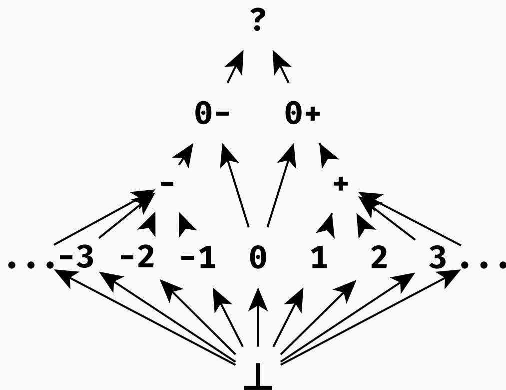
- *var* — это множество переменных в программе
- *int* — это плоская решётка над множеством целых чисел



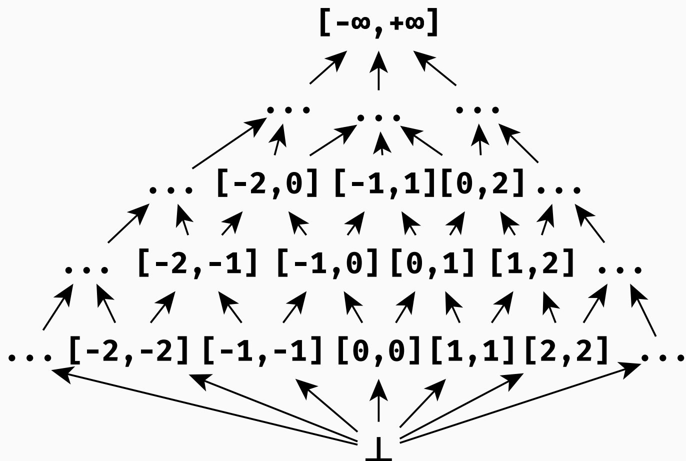
Какие проблемы у такого вида constant propagation?

- Выражения вида `x > 0` не будут константами даже если известно, что `x` положительный
- Давайте попробуем смешать эту решётку и решётку для знаков!



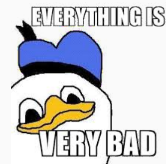


- Не слишком ли сложно?
- А если хотим поддерживать ограничения вида `1+`?



Есть ли тут какие-нибудь проблемы?

The interval lattice



Проблема, собственно, одна — она бесконечная

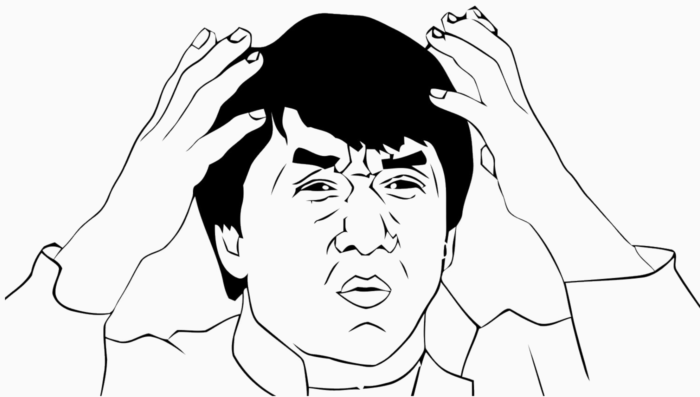
- Причем, во все стороны
- Даже если мы вспомним про ограниченность разрядности `int`
- Влияет ли это на наши алгоритмы?

Задания

- Допишите реализацию live variables analysis
  - `src/tip/analysis/LiveVarsAnalysis.scala`
- Реализуйте reaching definitions analysis
  - `src/tip/analysis/ReachingDefinitionsAnalysis.scala`
  - (такого файла там ещё нет =))

В следующей серии

Как использовать очень большие решётки и не умереть



belyaev@kspt.icc.spbstu.ru

akhin@kspt.icc.spbstu.ru