CS 131 Compilers: Discussion 10: Static Analysis

杨易为 吴凌云 樊雨鑫

{yangyw,wuly2,fanyx}@shanghaitech.edu.cn

2022 年 4 月 26 日

1 Data Flow Analysis[njuswanalysis]

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.

1.1 Iterative Algorithm

1. 对于一个含 k 个节点的 CFG, 每个迭代算法对于每个 node n 更新 OUT $[n]$。
2. 假设迭代算法的研究对象 (domain) 是 V, 定义一个 k 元组 $V^k = (\mathrm{OUT}[\mathrm{n_1}], \mathrm{OUT}[\mathrm{n_2}], \ldots, \mathrm{OUT}[\mathrm{n_k}])$ \$, \$$V^k \in (V_1 \times V_2 \times \cdots \times V_k)$ V^k 即一次迭代产生的输出，每次迭代会更新 V^k, 可以将每次迭代经过 transfer functions 和 control-flow handing 的过程抽象为 $F : V^k \to V^{k'}$
3. 当 $V^k \to V^{k'}$ 时, 即 $X = F(X)$, 称 $F(x)$ 在 X 处到达了不动点, X 为 $F(x)$ 的不动点,

1.2 Poset & partial order（偏序集和偏序）

We define poset as a pair $(P, 5)$ where $\subseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

1. $\forall x \in P, x \sqsubseteq x$ (Reflexivity, 自反性)
2. $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$ (Antisymmetry, 反对称性)
3. $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$ (Transitivity, 传递性)

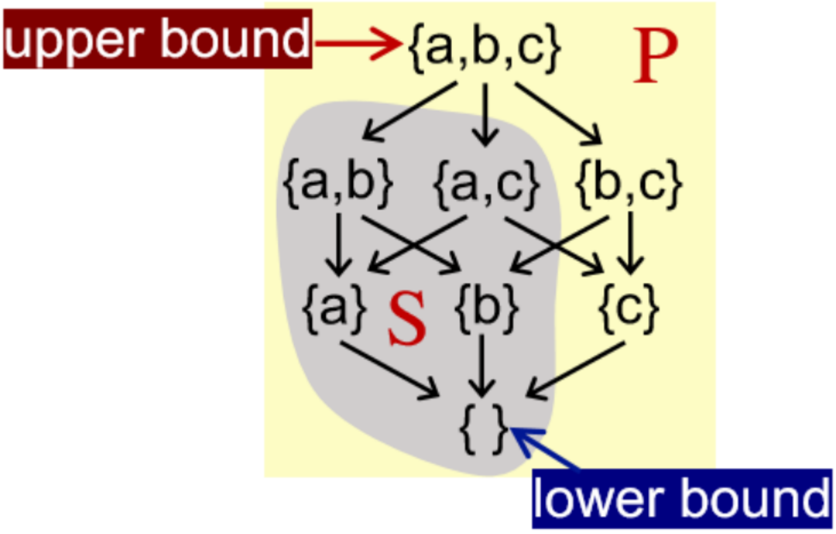
偏序集为一个二元组 $(P, \sqsubseteq)$, P 为一集合, 三为在集合上的一种比较关系，这个二元组为偏序集当且仅当集合元素在关系上满足自反性、反对称性和传递性。

偏序的含义：一个集合中的任意两个元素不一定存在顺序关系 (任意两元素不一定能比较大小)

1.3 Uppper and Lower Bounds（上界和下界）

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an upper bound of S, if $\forall x \in P, x \sqsubseteq u$. Similarly, $l \in P$ is an lower bound of S, if $\forall x \in P, l \sqsubseteq u$.

如图, $\{a, b, c\}$ 是S 的上界 (灰色) , {} 是S 的下界:



1.4 最小上界、最大下界

We define the least upper bound (lub or join) of S, written $\sqcup S$, if for every upper bound of S_1 say $u, \sqcup S \sqsubseteq u$. Similarly, We define the greatest lower bound (glb or meet) of S, written $\sqcap S$, if for every lower bound of S, say $l, l \sqsubseteq \sqcap S$.

特别的, 对于仅有两个元素的集合$S = \{a, b\}, \sqcup S$ 可以写为$a \sqcup b$, 同理 $\sqcap S$ 可以写为$a \sqcap b$ 。

注意:（最小）上界和（最大）下界是针对集合中的特定子集的，而上下界本身不一定在子集中，并且:

1. 不是所有偏序集均存在 lub 或者 glb（如先前灰色的集合就不含 lub）
2. 如果一个偏序集存在 lub 和 glb，那么它是唯一的

   证明：设 $g1$ 和 $g2$ 同为 P 的 glb，那么根据定义 $g_1 \sqsubseteq (g_2 = \sqcap P)$ 并且 $g_2 \sqsubseteq (g_1 = \sqcap P)$，又因为反对称性，所以 $g_1 = g_2$

2 Lattice, Semilattice, Complete and Product Lattice

2.1 Lattice（格）

Given a poset $(P, \sqsubseteq), \forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice.

如果一个偏序集的任意两个元素都有最小上界和最大下界，那么这一偏序集是一个**格**

2.2 Semilattice

最小上界和最大下界只存在一个的偏序集称**半格**，只存在最小上界称为 “join semilattice”，只存在最大下界称为 “meet semilattice”。

2.3 Complete Lattice

Given a lattice $(P, \sqsubseteq)$, for arbitrary subset S of P, if $\sqcup S$ and $\sqcap S$ exist, then $(P, \sqsubseteq)$ is called a complete lattice.

一个偏序集的任意子集均存在最小上界和最大下界，那么这个偏序集成为全格。每个全格都存在一个最大元素 top $(\top = \sqcup P)$ 和最小元素 bottom $(\bot = \sqcap P)$ 所有元素有限的格 (finite lattice) 均是全格。(反之不成立)

2.4 Product Lattice

Given lattices $L_1 = (P_1, \sqsubseteq_1), L_2 = (P_2, \sqsubseteq_2), \ldots, L_n = (P_n, \sqsubseteq) n)$, if for all $i, (P_i, \sqsubseteq_i)$ has $\sqcup i$ (least upper bound) and Π_i (greatest lower bound), then we can have a product lattice $L^n = (P, \sqsubseteq)$ that is defined by:

1. $P = P_1 \times \ldots \times P_n$
2. $(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \Leftrightarrow (x_1 \sqsubseteq y_1) \wedge \ldots \wedge (x_n \sqsubseteq y_n)$
3. $(x_1, \ldots, x_n) \sqcup (y_1, \ldots, y_n) = (x_1 \sqcup_1 y_1, \ldots, x_n \cup_n y_n)$
4. $(x_1, \ldots, x_n) \sqcap (y_1, \ldots, y_n) = (x_1 \sqcap_1 y_1, \ldots, x_n \sqcap_n y_n)$

Product Lattice 仍是 Lattice, 若每个子格为全格，那么乘积也是全格。

2.5 Data Flow Analysis Framework via Lattice

一个数据流分析框架可以表示为一个三元组 (D, L, F), 其中:

1. D: 指数据流分析的方向, i.e., forward or backward;
2. L：指 lattice, 该格表示所有 domain 值域, 以及 meet ( ) 或 join ($\sqcup$) 操作;
3. F: 一组 transfer function.

3 Monotonicity and Fixed Point Theorem

Monotonicity A function $f : L \to L$(L is a lattice) is monotonic if $\forall x, y \in L, x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$ 普通函数的单调性的推广

3.1 Fixed-Point Theorem

Given a complete lattice $(L, \sqsubseteq)$, if (1) $f : L \to L$ is monotonic and (2) L is finite, then the least fixed point of f can be found by iterating $f(\bot), f(f(\bot)), \ldots, f^k(\bot)$ until a fixed point is reached the greatest fixed point of f can be found by iterating $f(\top), f(f(\top)), \ldots, f^k(T)$ until a fixed point is reached.

如果番调且 L 有界, 那么荐在不动点, 从上开始迭代执行 f 可得最小不动点, 从丁开始迭代可得最大不动点。

证明:

(1) Existence 由上定义以及 $f : L \to L$ 可榻

$$\bot \sqsubseteq f(\bot)$$

又因 f 是单调的，因此

$$f(\bot) \sqsubseteq f(f(\bot)) = f^2(\bot)$$

由于 L 是有限 (finite) 的, 因此总会存在一个 k, 有

$$f^{Fix} = f^k(\bot) = f^{k+1}(\bot)$$

(2) Least Fixed Point (数归法, 证明最小) 假设我们有另一个不动点 x, i.e., $x = f(x)$(单调性) 由上的定义, 我们有 $\bot \sqsubseteq x_i$ 下面用数归法证明：

由于 f 是单调的，因此

$$f(\bot) \sqsubseteq f(x)$$

对于 $f^i(\bot) \sqsubseteq f^i(x)$, 由于 f 是单调的, 因此有

$$f^{i+1}(\bot) \sqsubseteq f^{i+1}(x)$$

因此对于任意 i, 有

$$f^i(\bot) \sqsubseteq f^i(x)$$

又因为 $x = f(x)$, 所以存在一个 i, 有 $f^i(\bot) \sqsubseteq f^i(x) = x$, 因此有

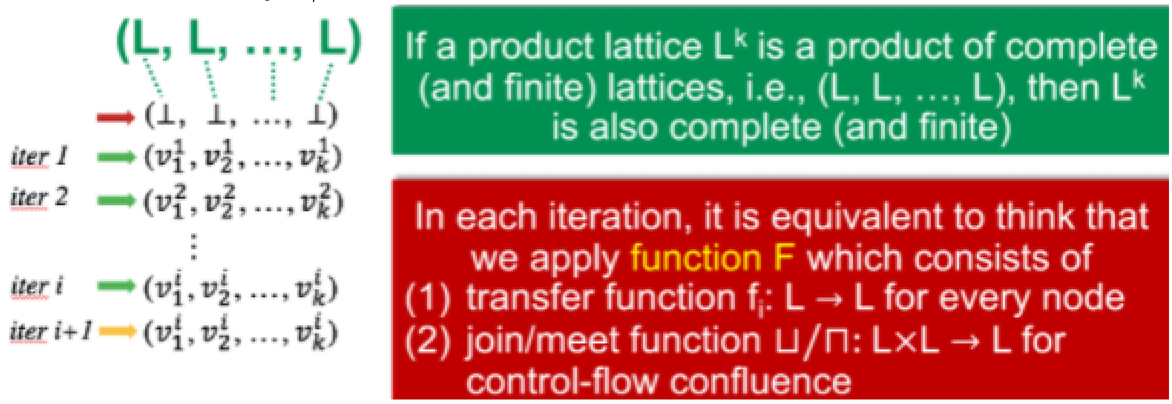
$$f^{Fix} = f^k(\bot) \sqsubseteq x$$

因此 $f^i(\bot)$ 是最小不动点。

4 Relate lterative Algorithm to Fixed Point Theorem

如何将迭代算法和不动点定理联系起来?

1. 程序中每一个状态为一个 product lattice
2. Transfer function 和 join/meet fucntion 可以视为 F



下面只需要证明 Transfer function 和 join/meet function 均为单调的即可

1. Transfer function 是单调的, 因为通过之前分析，所有 Gen/Kill 的函数都是单调的;
2. Join/meet function 是单调的, 证明如下

要证 Join/meet function，就是要证

$$\forall x, y, z \in L, x \sqsubseteq y \rightarrow x \sqcup z \sqsubseteq -y \sqcup z$$

由 $\sqcup$ 定义可得, $y \sqsubseteq y \sqcup z$, 由于巨传递性, $x \sqsubseteq y$, 因此 $x \sqsubseteq y \sqcup z$, 因此 $y \sqcup z$ 是 x 的上界, 注意到 $y \sqcup z$ 也是 z 的上界, 而 $x \sqcup z$ 是 x 和 z 的最小上界, 因此 $x \sqsubseteq y \Rightarrow x \sqcup z \sqsubseteq y \sqcup z$

4.1 讨论算法复杂度

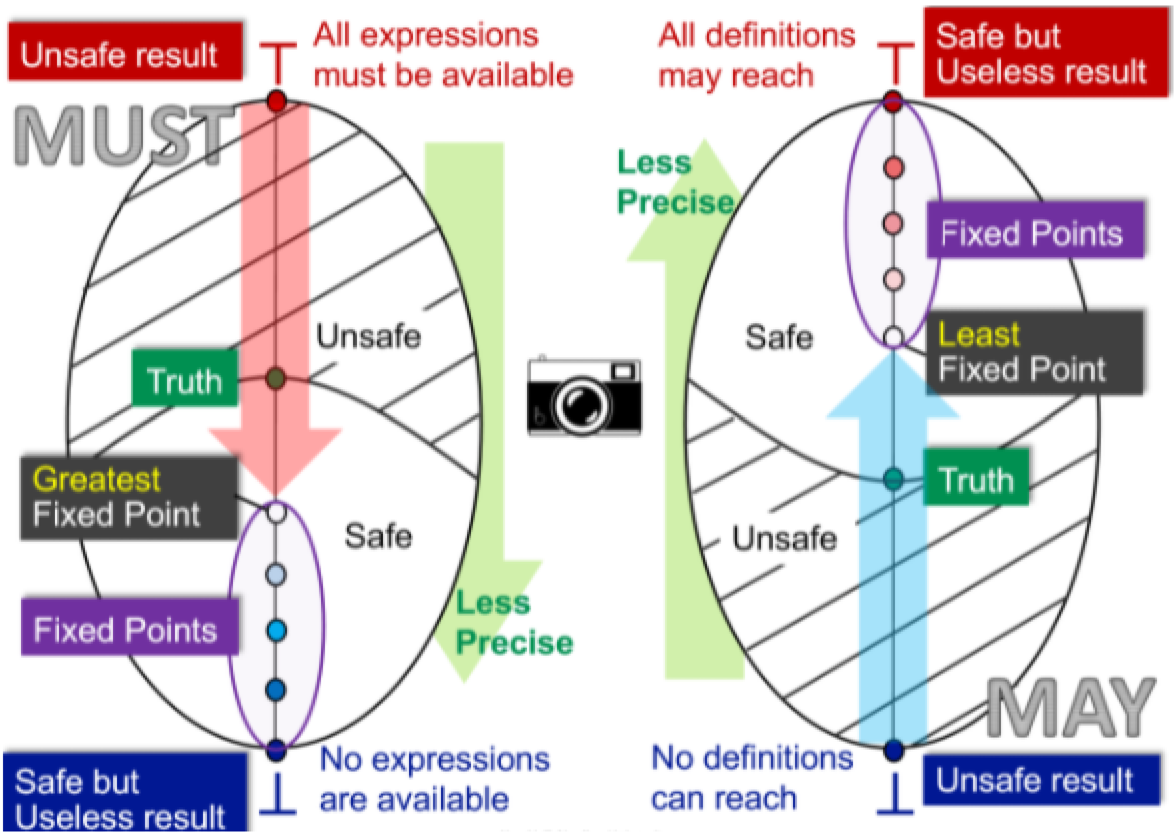
定义格的高度即从 top 至 bottom 的最长路径长, The height of a lattice h is the length of the longest path from Top to Bottom in the lattice. 最坏情况即一次迭代, 只变化一个单位高度, 因此复杂度为 $O(h \times k)$.

5 May/Must Analysis, A Lattice View*

任何分析初始状态都从 unsafe 至 safe。针对于分析结果而言，处理所有分析结果后，程序行为正常为 safe，反之为 unsafe，极端的 safe 是无用的（如安全扫描中的模式匹配）

Must 和 May 分析的示意图如下图所示，对于 Must 分析，每个代码块都是从 $\top$ 开始的，因为在程序一开始，算法认为所有的待分析对象都是“合格”的（例如存活表达式分析中，算法认为每个表达式都是成活的）——这是一个不安全的状态，经过不断迭代，算法逐渐下降到最大不动点，虽然已经过了 truth 点（漏报），但是这已经是最好情况了（越往下走越 safe 但是结果也没意义了），对这些结果做优化能确保程序不出错（safe）。

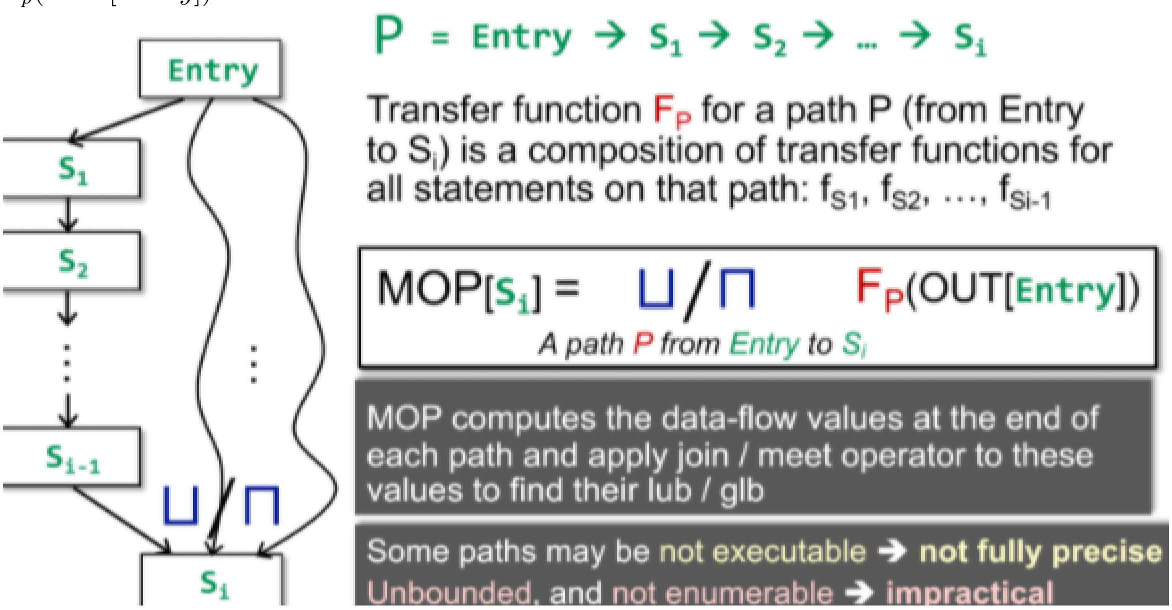
对于 May 分析，每个代码块从 $\bot$ 开始，即在一开始，算法认为所有分析对象都是不合格的（例如定义可达性分析中，算法认为每一条定义都没有新的定义）——这是 May 类型的不安全状态，经过不断迭代，算法逐渐上升到最小不动点，同样也过了 truth（误报），这也是分析的最好情况，算法依旧停留在 safe 区域。



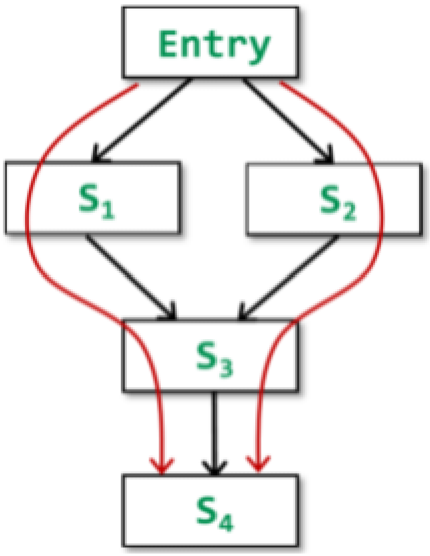
6 Distributivity（分配性）and MOP

6.1 MOP（Meet-Over-All-Paths Solution）

设 s_x 为矢设 F_p 是一个路径 P 上的 transfer function, 那么 MOP $[S_i] = \sqcup/\sqcap_{\text{A path } P \text{ from Entry to } S_i} F_p(OUT[Entry])$



就是说，之前数据流分析的结果是流敏感的，而 MOP 的结果是路径敏感的，例如下图所示的数据流，，数据流分析的结果是 IN $[s_4] = f_{s_3}(f_{s_1}(\text{OUT}[\text{ entry }]) \sqcup f_{S_2}(\text{ OUT }[\text{ Entry }]))$，而 MOP 是 MOP $[s_4] = f_{S_3}(f_{s_1}(\text{OUT}[\text{ entry }])) \sqcup f_{S_3}(f_{S_2}(\text{OUT}[\text{ Entry }]))$ (注意 f_{s_3} 的位置)



6.2 Iterative Algorithm v.s. MOP

MOP 比 Iterative 分析更精确，也就是说路径敏感比敏感更精确，下面为证明（这里只需证明两条路径的情况，其它数归即可）:

然而，当 F 是 distributive（F 满足分配率）时，Interative 和 MOP 一样准确。

BitVector 或是 Gen/Kill 问题（set union/intersection for join/meet）都是满足分配率的

6.3 Constant Propagation

Given a variable x at program point p, determine whether x is guaranteed to hold a constant value at p 对于在程序点 p 的一个变量 x，判断 x 是否为在 p 点为一个常量

分析结果：对于每一个 CFG 节点，对应一个 (x,v) 集合，x 是变量，v 是 x 的值

6.4 Lattice

Domain: UNDEF $\rightarrow \{\ldots, -2, -1, 0, 1, 2, \ldots\} \rightarrow$ NAC, $\rightarrow$ 表示 $\sqsubseteq$ 关系 Meet Operator $\sqcup$:

1. $\text{NAC} \sqcap v = \text{NAC}$
2. $UNDEF \sqcap v = v$
3. $c \sqcap v = \text{NAC}$ (c 为一常量)
4. $c \sqcap c = c$
5. $c_1 \sqcap c_2 = \text{NAC}$

6.5 Transfer Function

讨论 transfer function, 对于一个赋值语句 $\mathrm{s : x = \ldots}$ 来说, 定义其 F 为

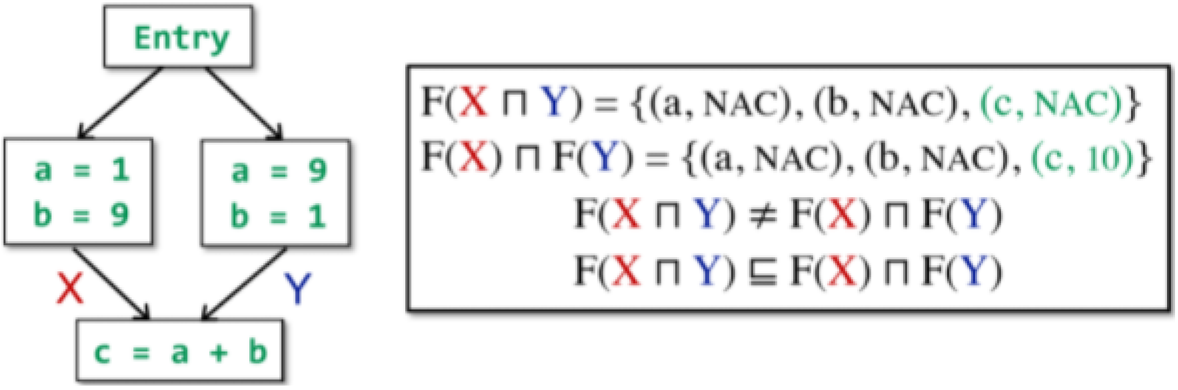
$$F : \text{OUT}[s] = \text{gen} \cup (\text{IN}[s] - \{(x, \_)\})$$

- $\mathrm{s : x = c}$; gen $= \{(x, c)\}$ -s: x = y; gen $= \{(x, \text{val}(y))\}$ - s: $x = y$ op z; gen $= \{(x, f(x, z))\}$ 而 $f(x, z)$ 有三种情况:

$$f(y, z) = \begin{cases} \text{val}(y) \text{ op } \text{val}(z) & \text{// if val(y) and val(z) are constants} \\ \text{NAC} & \text{// if val(y) or val(z) is NAC} \\ \text{UNDEF} & \text{// otherwise} \end{cases}$$

6.6 function 不满足分配性

如下图所示，$F(X \sqcap Y)$ 中 C 的值为 NAC，而 $F(X) \sqcap F(Y)$ 中 C 的值为 10，因此 F 不满足分配性。



7 Worklist Algorithm

Iterative Algorithm 的优化，Iterative 存在冗余的计算，而 Worklist 只计算有变化的 node：

```
OUT[entry] = ;
for(each basic block B\entry)
        OUT[B] = ;
    Worklist←all basic blocks
        while (Worklist is notempty)
        Pick a basic block B from Worklist
                old_OUT= OUT[B]
        IN[B] = OUT[P]; # join/meet P为B的前置代码块
        OUT[B] = genB U (IN[B] - killB); # transfer function
        if(old_OUT OUT[B])
                Add all successors of B to Worklis
```

7.1 分析特性

1. 流敏感：程序语句随意调换位置，分析结果不变为流非敏感，否则为流敏感；
2. 路径敏感：考虑程序中路径的可行性；

8 Space of dataflow ananlys [cs153lec18]

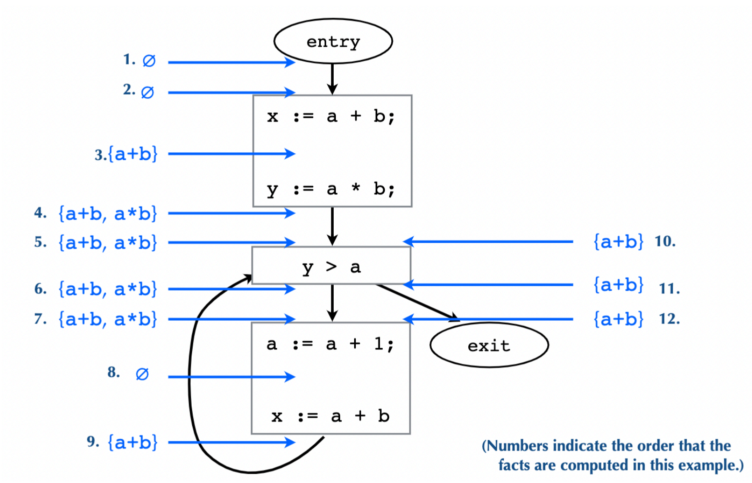
	May	Must
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

8.1 Available Expressions

1. An expression e is available at program point p if on all paths from the entry to p, expression e is computed at least once, and there are no intervening assignment to x or to the free variables of e
2. If e is available at p, we do not need to re-compute e
   (a) (i.e., for common sub-expression elimination)
3. How do we compute the available expressions at each program point?

Formally:

1. Suppose D is a set of expressions that are available at program point p
2. Suppose p is immediately before " x := e_1;　B″
3. Then the statement " x := e_1 "
   (a) generates the available expression $\mathbf{e}_1$



(b) kills any available expression $\mathbf{e}_2$ in D such that $\mathbf{x}$ is in variables $(\mathbf{e}_2)$

4. So the available expressions for B are: $(D \cup \{\mathbf{e}_1\}) - \{\mathbf{e}_2 \mid \mathbf{x} \in \text{variables}\,(\mathbf{e}_2)\}$

Stmt	Gen	Kill
$\mathbf{x} := \mathbf{v}$	{v}	{e | x in e}
$\mathbf{x} := \mathbf{v}_1$ op $\mathbf{v}_2$	{$\mathbf{v}_1$ op v_2}	{e | x in e}
$\mathbf{x} := *(\text{v} + i)$	{}	{e | x in e}
$*(\text{v} + i) := \text{x}$	{}	{}
jump L	{}	{}
return v	{}	{}
if v1 op v2 goto I1 else goto L2	{}	{}
$\mathbf{x} := \mathbf{v}\,(\mathbf{v}_1, \ldots \mathbf{v}_n)$	{}	{e | in e}

Transfer function for stmt $S : \lambda D.\, (D \cup Gen_S) -$ Kill $_S$

8.1.1 Aliasing

You can use Aliasing to save register!

1. We can tell whether variables names are equal.
2. We cannot (in general) tell whether two variables will have the same value.
3. If we track * x as an available expression, and then see $*y := e'$, don't know whether to kill * x
4. Don't know whether x's value will be the same as y' s value

```
void accumulate_restrict(int* restrict source, int len, int* restrict target) {
    for (int i = 0; i < len; i++) {
        *target += source[i];
    }
}
```

8.2 Reaching Definitions

1. A definition x := e reaches a program point p if there is some path from the assignment to p that contains no other assignment to x
2. Reaching definitions useful in several optimizations, including constant propagation
3. Can also define reaching definitions analysis using gen and kill sets; combine dataflow facts at merge points by union

Define defs($\mathbf{x}$) to be the set of all definitions of variable x

Stmt	Gen	Kill
d : $\mathbf{x} := \mathbf{v}$	{d}	defs($\mathbf{x}$) − {d}
d : $\mathbf{x} := \mathbf{v}_1$ op $\mathbf{v}_2$	{ d}	defs($\mathbf{x}$) − {d}
everything else		

1.
$$D_{\text{in}}\ [\text{L}] = \cup \{D_{\text{out}}\ [\text{L}'] \mid \text{L}' \text{ in pred } [\text{L}]\}$$

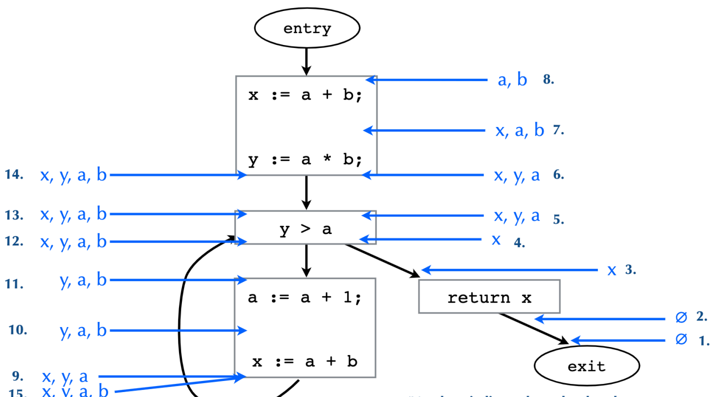
2. Transfer function for stmt $S : \lambda D.$ (D $\cup$Gen$_S$) − Kill$_S$

8.3 Liveness

1. Variable $\mathbf{x}$ is live at program point p is there is a path from p to a use of variable x
2. Liveness useful in dead code elimination and register allocation
3. Can also define using gen-kill sets
4. However, we use a backward dataflow analysis i.e., instead of flowing facts forwards over statement (computing D_{out} from D_{in}) we flow facts backwards over statements (compute D_{in} from D_{out})

Stmt	Gen	Kill
$\mathbf{x} := \mathbf{v}$	$\{\mathbf{v} \mid$ if v is variable $\}$	$\{\mathbf{x}\}$
$\mathbf{x} := \mathbf{v}_1$ op $\mathbf{v}_2$	$\{\text{v}_i \mid i \in 1, 2, \text{v}_i$ is var $\}$	$\{\mathbf{x}\}$
$\mathbf{x} := *(\text{v} + i)$	$\{\mathbf{v} \mid$ if v is variable $\}$	$\{\mathbf{x}\}$
$*(\text{v} + i) := \mathbf{x}$	$\{\text{x}\} \cup \{\text{v} \mid$ if v is variable $\}$	{}
jump L	{}	{}
return v	$\{\text{v} \mid$ if v is variable $\}$	{}
if v_1 op v2 goto L1 else goto L2	$\{\mathbf{v}_i \mid i \in 1, 2, \mathbf{v}_i$ is var $\}$	{}
$\mathbf{x} := \mathbf{v}_0\,(\text{v}_1, \ldots \text{v}_\text{n})$	$\{\mathbf{v}_i \mid i \in 0..n, \text{v}_i$ is var $\}$	{x}

1. i.e., any use of a variable generates liveness, any definition kills liveness
2. $D_{\text{out}}\ [\text{L}] = \cup \{D_{\text{in}}\ [\text{L}'] \mid \text{L}' \text{ in succ}[\text{L}]\}$
3.
$$\text{Transfer function for stmt } S : \lambda D.(D \cup \text{ Gen }_S) - \text{ Kill }_S$$



(Numbers indicate the order that the facts are computed in this example.)