软件分析与架构设计

指称语义和CFL可达性问题

何冬杰

重庆大学

指称语义（Denotation Semantics）

- **Revisit the flow function $f: L \to L$ on an instruction $inst$**
  - We define it as $\sigma' = f[\![inst]\!](\sigma)$
  - What does $f[\![inst]\!]$ here mean?
    - $f[\![inst]\!]$ is a partial function from states to states!
    - $f[\![inst]\!]$ is the denotation of $inst$
- **Revisit SIMP**

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)

Meta-Variable

$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{lcl}
b & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } b \\
 & | & b_1\ op_b\ b_2 \\
 & | & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{lcl}
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{lcl}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leqslant \mid = \\
 & | & > \mid \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

Abstract Syntax

指称语义（Denotation Semantics）

❑Denotations of Aexp: $\mathcal{A}$: Aexp $\rightarrow (\Sigma \rightarrow N)$

$$\mathcal{A}[\![n]\!] = \{(\sigma, n) \mid \sigma \in \Sigma\}$$
$$\mathcal{A}[\![X]\!] = \{(\sigma, \sigma(X)) \mid \sigma \in \Sigma\}$$
$$\mathcal{A}[\![a_0 + a_1]\!] = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \;\&\; (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$
$$\mathcal{A}[\![a_0 - a_1]\!] = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \;\&\; (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$
$$\mathcal{A}[\![a_0 \times a_1]\!] = \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in \mathcal{A}[\![a_0]\!] \;\&\; (\sigma, n_1) \in \mathcal{A}[\![a_1]\!]\}$$

Equivalent λ −notation

$$\mathcal{A}[\![n]\!] = \lambda\sigma \in \Sigma.n$$
$$\mathcal{A}[\![X]\!] = \lambda\sigma \in \Sigma.\sigma(X)$$
$$\mathcal{A}[\![a_0 + a_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma)$$
$$\mathcal{A}[\![a_0 - a_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!]\sigma - \mathcal{A}[\![a_1]\!]\sigma)$$
$$\mathcal{A}[\![a_0 \times a_1]\!] = \lambda\sigma \in \Sigma.(\mathcal{A}[\![a_0]\!]\sigma \times \mathcal{A}[\![a_1]\!]\sigma)$$

❑Example

- For any state σ:

$$\mathcal{A}[\![3 + 5]\!]\sigma = \mathcal{A}[\![3]\!]\sigma + \mathcal{A}[\![5]\!]\sigma = 3 + 5 = 8$$

指称语义（Denotation Semantics）

❑Denotations of Bexp: $\mathcal{B}$: $\mathbf{Bexp} \rightarrow (\Sigma \rightarrow T)$

$$\mathcal{B}[\![\mathbf{true}]\!] = \{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma\}$$

$$\mathcal{B}[\![\mathbf{false}]\!] = \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma\}$$

$$\mathcal{B}[\![a_0 = a_1]\!] = \{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma\} \cup \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma \neq \mathcal{A}[\![a_1]\!]\sigma\},$$

$$\mathcal{B}[\![a_0 \leq a_1]\!] = \{(\sigma, \mathbf{true}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma\} \cup \{(\sigma, \mathbf{false}) \mid \sigma \in \Sigma \;\&\; \mathcal{A}[\![a_0]\!]\sigma \not\leq \mathcal{A}[\![a_1]\!]\sigma\},$$

$$\mathcal{B}[\![\neg b]\!] = \{(\sigma, \neg_T t) \mid \sigma \in \Sigma \;\&\; (\sigma, t) \in \mathcal{B}[\![b]\!]\},$$

$$\mathcal{B}[\![b_0 \wedge b_1]\!] = \{(\sigma, t_0 \wedge_T t_1) \mid \sigma \in \Sigma \;\&\; (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \;\&\; (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\},$$

$$\mathcal{B}[\![b_0 \vee b_1]\!] = \{(\sigma, t_0 \vee_T t_1) \mid \sigma \in \Sigma \;\&\; (\sigma, t_0) \in \mathcal{B}[\![b_0]\!] \;\&\; (\sigma, t_1) \in \mathcal{B}[\![b_1]\!]\}.$$

- Each denotation is a function
- For example, for all σ ∈ Σ,

$$\mathcal{B}[\![a_0 \leq a_1]\!]\sigma = \begin{cases} \mathbf{true} & \text{if } \mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma, \\ \mathbf{false} & \text{if } \mathcal{A}[\![a_0]\!]\sigma \not\leq \mathcal{A}[\![a_1]\!]\sigma \end{cases}$$

指称语义（Denotation Semantics）

- **Denotations of Com:** $\mathcal{C}$: $\mathrm{Com} \to (\Sigma \to \Sigma)$
  - **Com** stands for commands/statements/Instructions

$$\mathcal{C}[\![\mathbf{skip}]\!] = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$$

$$\mathcal{C}[\![X := a]\!] = \{(\sigma, \sigma[n/X]) \mid \sigma \in \Sigma \;\&\; n = \mathcal{A}[\![a]\!]\sigma\}$$

$$\mathcal{C}[\![c_0; c_1]\!] = \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!]$$

$$\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1]\!] =$$
$$\{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \;\&\; (\sigma, \sigma') \in \mathcal{C}[\![c_0]\!]\} \cup$$
$$\{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false} \;\&\; (\sigma, \sigma') \in \mathcal{C}[\![c_1]\!]\}$$

$$\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = \mathit{fix}(\Gamma)$$

Let us explain why this is.

where

$$\Gamma(\varphi) = \{(\sigma, \sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \;\&\; (\sigma, \sigma') \in \varphi \circ \mathcal{C}[\![c]\!]\} \cup$$
$$\{(\sigma, \sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\}.$$

指称语义（Denotation Semantics)

- **Denotations of Com: $\mathcal{C}$: $\mathrm{Com} \to (\Sigma \to \Sigma)$**
  - $w \equiv$ while b do c (use w to denote while b do c)
  - $w \sim$ **if** b **then** c; w **else skip**

$$\begin{aligned}\mathcal{C}[\![w]\!] =& \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \ \&\ (\sigma,\sigma') \in \mathcal{C}[\![c;w]\!]\} \cup \\ & \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\} \\ =& \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \ \&\ (\sigma,\sigma') \in \mathcal{C}[\![w]\!] \circ \mathcal{C}[\![c]\!]\} \cup \\ & \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\}.\end{aligned}$$

$$\begin{aligned}\Gamma(\varphi) =& \{(\sigma,\sigma') \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{true} \ \&\ (\sigma,\sigma') \in \varphi \circ \mathcal{C}[\![c]\!]\} \cup \\ & \{(\sigma,\sigma) \mid \mathcal{B}[\![b]\!]\sigma = \mathbf{false}\}.\end{aligned}$$

When $\varphi = \Gamma(\varphi)$, φ (the fixed point of Γ) denotes $\mathcal{C}[\![w]\!]$.

语义等价性

Lemma 5.3 *For all* $a \in$ **Aexp**,

$$\mathcal{A}[\![a]\!] = \{(\sigma, n) \mid \langle a, \sigma\rangle \to n\}.$$

Lemma 5.4 *For* $b \in$ **Bexp**,

$$\mathcal{B}[\![b]\!] = \{(\sigma, t) \mid \langle b, \sigma\rangle \to t\}.$$

Theorem 5.7 *For all commands* c

$$\mathcal{C}[\![c]\!] = \{(\sigma, \sigma') \mid \langle c, \sigma\rangle \to \sigma'\}.$$

- Formal proof requires structural induction
- (Optional) Refer to Chapter 5.3 of "The Formal Semantics of Programming Languages"

指称语义的作用

- **Equivalence:** $\boldsymbol{c_0 \sim c_1}$ iff $[\![\boldsymbol{c_0}]\!] \Leftrightarrow [\![\boldsymbol{c_1}]\!]$
  - $[\![\boldsymbol{c_0}]\!] \Leftrightarrow [\![\boldsymbol{c_1}]\!]$ means $\forall \sigma. [\![\boldsymbol{c_0}]\!](\sigma) \Leftrightarrow [\![\boldsymbol{c_1}]\!](\sigma)$
  - $[\![\boldsymbol{c_0}]\!] \Leftrightarrow [\![\boldsymbol{c_1}]\!]$ also means $\forall\, \sigma, \sigma'. \langle c_0, \sigma \rangle \rightarrow \sigma' \;\Leftrightarrow \langle c_1, \sigma \rangle \rightarrow \sigma'$
  - If $\boldsymbol{c_0}$ and $\boldsymbol{c_1}$ are semantically equivalent and $\boldsymbol{c_1}$ is more efficient in terms of running time or memory, the compiler would be happy using $\boldsymbol{c_1}$ to replace $\boldsymbol{c_0}$ during its optimization passes
- **Function/Method Summary (函数摘要):**
  - Let $\boldsymbol{c_0}$ and $\boldsymbol{c_1}$ be two call instructions to the same method m
  - Then $[\![\boldsymbol{c_0}]\!] \Leftrightarrow [\![\boldsymbol{c_1}]\!]$,
  - given a state σ, if we have $\sigma' = [\![c_0]\!](\sigma)$, we conclude $\sigma' = [\![c_1]\!](\sigma)$
  - implying that we do not need to analyze method m a second time with the given state σ

指称语义举例

❑Possibly uninitialized variables

➢ Reps，POPL 1995

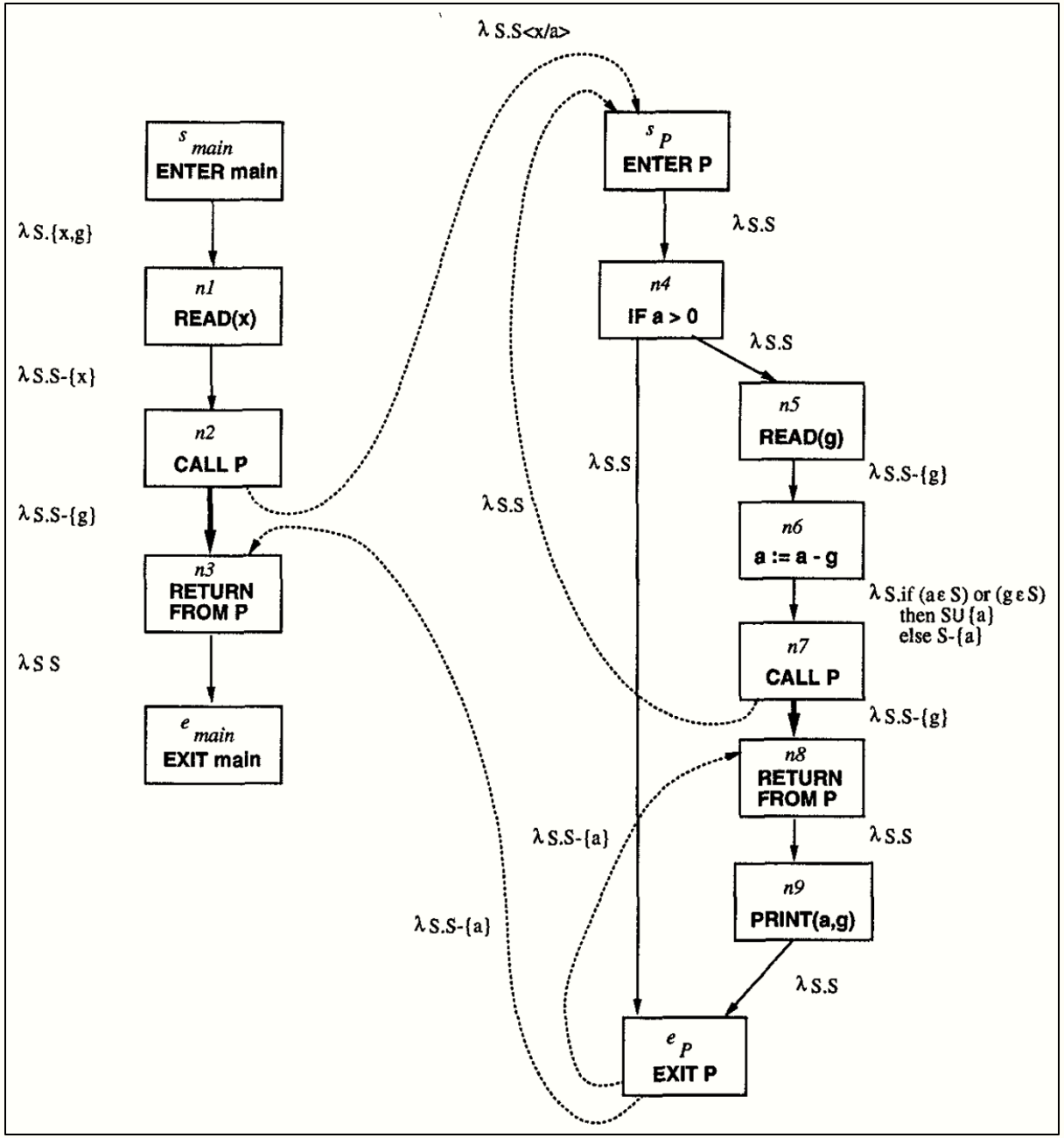
```
declare g: integer

program main
begin
  declare x: integer
  read(x)
  call P(x)
end

procedure P(value a: integer)
begin
  if (a > 0) then
    read(g)
    a := a − g
    call P(a)
    print(a, g)
  fi
end
```

此处g可能未被初始化

- 边上的lambdas是流函数
- 对应source节点的指称语义
- 比如$[\![READ(x)]\!] = \lambda S.S - x$
- 过程P在n2和n7被分别调用，其指称语义可作摘要被复用
- 如何区分P的两次调用？



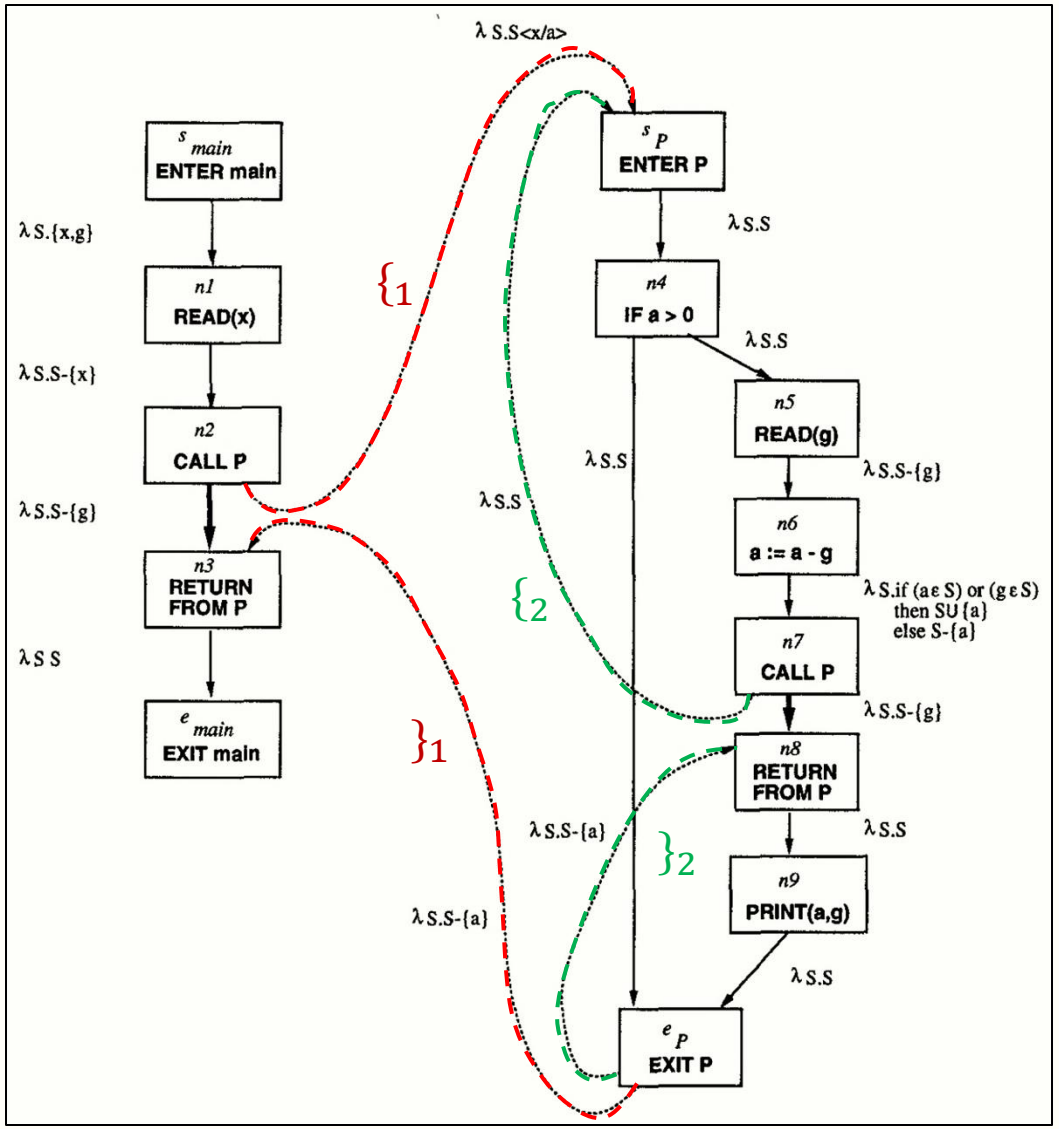
Supergraph（ICFG的变种，调用语句被拆成两个节点）

CFL可达性分析

- **Dyck-CFL:**
  - 括号匹配的上下文无关语言

$$S \rightarrow \{_1\ S\ \}_1 \mid \{_2\ S\ \}_2 \mid \cdots \mid S\ S \mid \epsilon$$

- **通过括号匹配去除不可行路径**
  - Feasible path:
    - $s_{main}, n_2, \{_1, s_P, \cdots, e_P, \}_1, n_3, e_{main}$
  - Infeasible path:
    - $s_{main}, n_2, \{_1, s_P, \cdots, e_P, \}_2, n_8, e_P$, …
    - $n_7, \{_2, s_P, \cdots, e_P, \}_1, n_3, e_{main}$
  - 给定一条路径，如果该路径上的符号组成Dyck-CFL的句子，则该路径是**可行路径**，否则是**不可行路径**



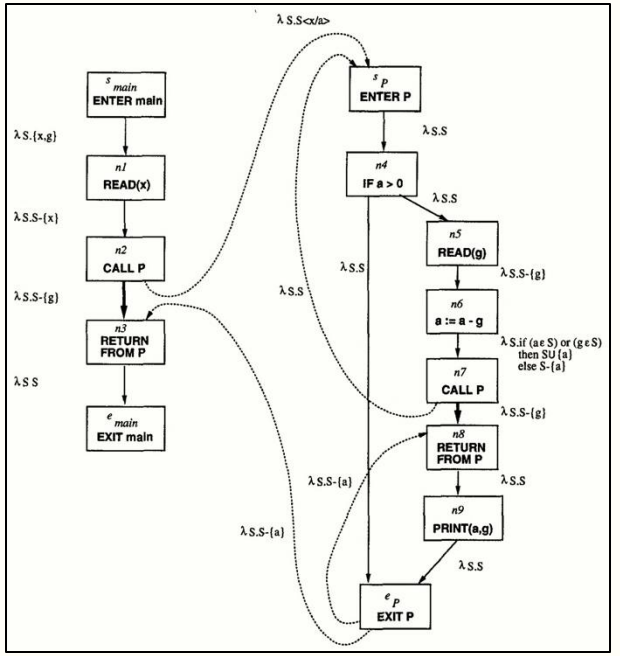
Distributive Denotation

- **Denotation is a function**
- **Distributive (分配性)**

$$\forall\, v \in \boldsymbol{Insts}, \sigma_1, \sigma_2 \in \Sigma\colon \mathrm{f}[\![v]\!](x) \sqcup \mathrm{f}[\![v]\!](y) = \mathrm{f}[\![v]\!](x \sqcup y)$$

  - Hold for composition: *If both f_1 and f_2 are distributive, then their composition $f_1 \circ f_2$ is also distributive.*

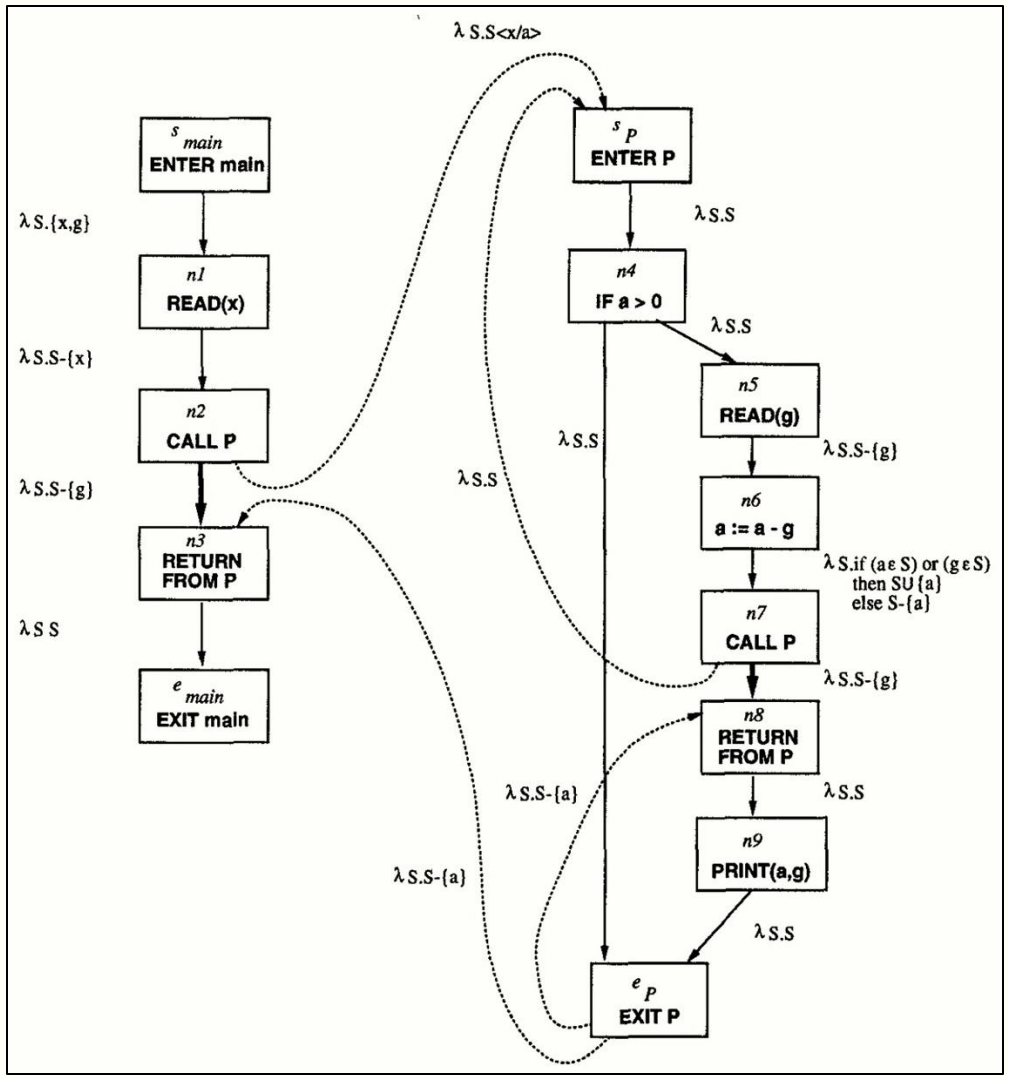
- **Possibly uninitialized variables:**
  - 可验证其流函数恰好满足分配性
  - 例：$\lambda \mathrm{S}.\,\mathrm{S} - \{\mathrm{g}\}$



如何精确计算函数指称语义$[\![m]\!]$?

- 假定流函数满足分配性：
  - 给定一条路径，其边上流函数顺次组合构成该**路径的流函数**
  - 超图上任意两点间所有路径的流函数meet到一起构成这两点间程序语句的denotation语义
    - 给定函数m，s_m到e_m的所有路径流函数meet到一起构成函数m的denotation
  - 两点间路径可能有无数个（loops or recursions）？
  - 可证Merge Over Flow(MFP)与Merge Over Paths(MOP)等价
    - 归纳法（Optimal）：参考《程序分析原理》2.4节
  - 通过不动点迭代计算，结果非常精确

举例：如何精确计算函数指称语义



$$[\![\mathrm{P}]\!] = [\![e_p]\!] \circ [\![\mathrm{IF\ a\ >\ 0}]\!] \circ [\![s_p]\!] \sqcup \cdots = \lambda S.\,\mathrm{S} \sqcup \cdots = \lambda S.\,\mathrm{S}$$

接下来计算[main]:

$$\begin{aligned}
&[\![\mathrm{main}]\!] = [\![e_{\mathrm{main}}]\!] \circ [\![\mathrm{call\ P}]\!] \circ [\![\mathrm{READ(x)}]\!] \circ [\![s_{main}]\!] \\
&= \lambda S.\,\mathrm{S} \circ (\lambda S.\,\mathrm{S} - \{\mathrm{g}\} \sqcup \lambda S.\,\mathrm{S}) \circ \lambda S.\,\mathrm{S} - \{\mathrm{x}\} \circ \lambda S.\,\{\mathrm{x,g}\} \\
&= \lambda S.\,\mathrm{S} \circ (\lambda S.\,\mathrm{S} - \{\mathrm{g}\} \sqcup \lambda S.\,\mathrm{S}) \circ \lambda S.\,\{\mathrm{g}\} \\
&= \lambda S.\,\mathrm{S} \circ \lambda S.\,\mathrm{S} \circ \lambda S.\,\{\mathrm{g}\} = \lambda S.\,\{\mathrm{g}\}
\end{aligned}$$

因此，可以推断全局变量g有可能未初始化

需要传统数据流分析的不动点计算

当configuration为有限集合时有更高效算法：**IFDS**!

基于IFDS的数据流分析


Thomas Reps

❑1995年由Reps等人提出

Thomas Reps, Susan Horwitz, and Mooly Sagiv. "***Precise interprocedural dataflow analysis via graph reachability***", **POPL 1995**

- **IFDS** (**I**nterprocedural, **F**inite, **D**istributive, **S**ubset) problems
  - 要求configuration或dataflow facts元素数量有限
  - 要求flow functions满足分配性：$\forall\, v \in \boldsymbol{Insts}, x, y \in S\colon\ f_v(x) \sqcup f_v(y) = f_v(x \sqcup y)$
    - ❖意味着每一条路径的数据流计算可以**并行处理**

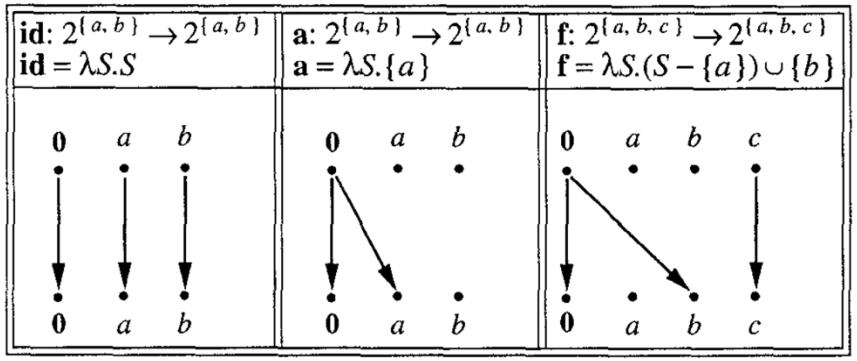
❑将数据流分析问题转化CFL-可达性问题

- 将流函数转为二元关系（即图）

Definition 3.1. The *representation relation of f*, $R_f \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$, is a binary relation (*i.e.*, graph) defined as follows:

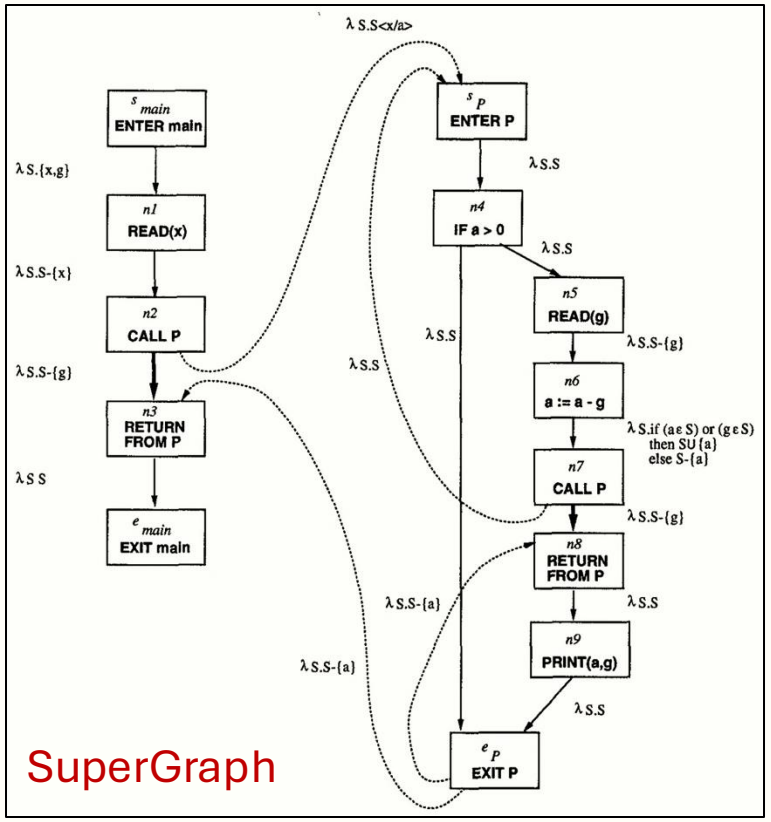
$$R_f =_{df} \{(\mathbf{0}, \mathbf{0})\} \cup \{(\mathbf{0}, y) \mid y \in f(\varnothing)\} \cup \{(x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\varnothing)\}.$$ □

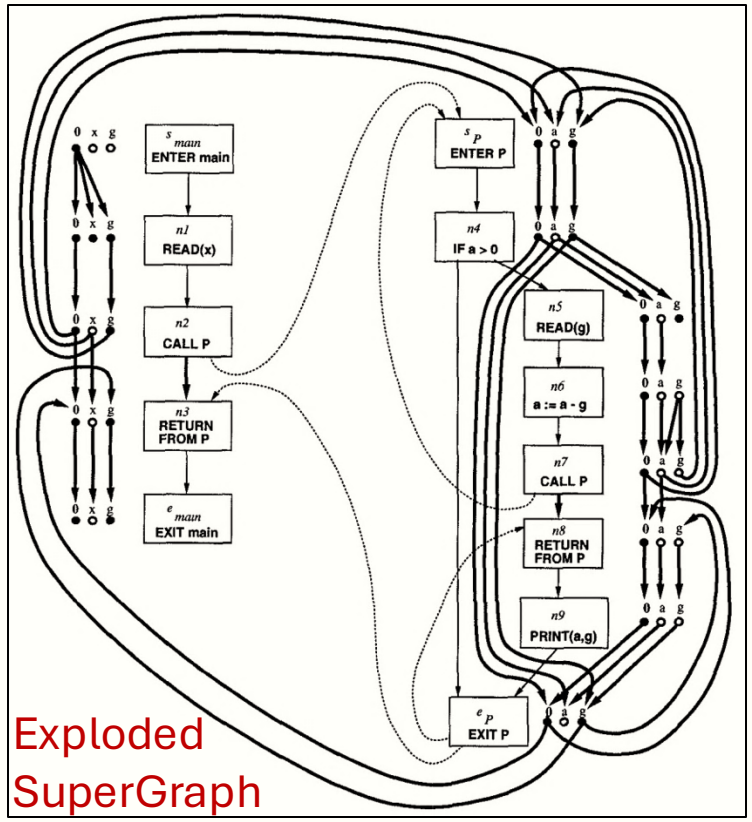
示例



将IFDS问题转化CFL-可达性问题

- 将超图上所有边对应流函数进行转化成二元关系得爆炸超图

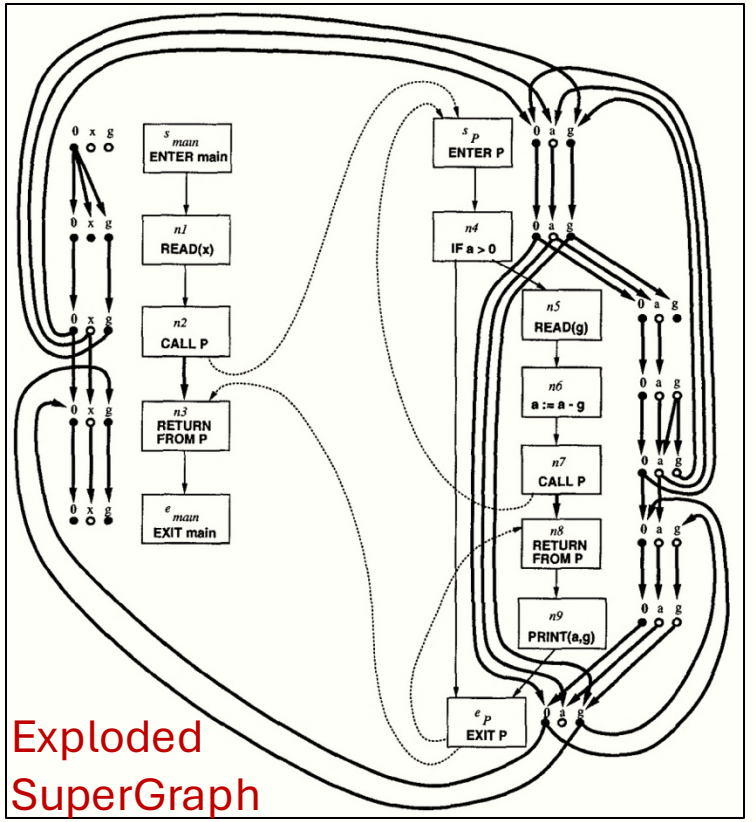




- 跨过程的边标上$\{_i$或$\}_i$，使用CFL语言过滤infeasible path

将IFDS问题转化CFL-可达性问题

- **对于可能未初始化变量问题：**
  - 变量a在程序点P处可能未初始化当且仅有爆炸超图中存在一条从main函数入口到当前程序点P的feasible可达路径
- **计算函数指称语义：**
  - 对于函数m，在s_m处的data facts和在e_m处的data facts中之间的CFL可达性关系构成m的denotation $[\![m]\!]$
  - 右图容易得到：
    - $[\![P]\!] = \lambda S.S$
    - $[\![main]\!] = \lambda S.\{0, g\}$



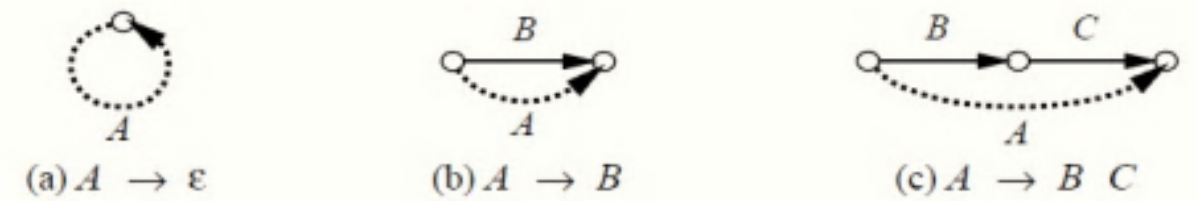
如何计算CFL可达性呢?

计算CFL可达性：CYK算法

- **CYK算法是一种动态规划算法**
  - Invented by John **C**ocke, Daniel **Y**ounger, and Tadao **K**asami
- **假定CFL产生式为Chomsky Normal Form形式**
  - $A \rightarrow BC(A, B, C\ are\ non-terminals)\ or$
  - $\mathrm{A} \rightarrow \mathrm{d}$ (d is a terminal) or
  - $\mathrm{S} \rightarrow \epsilon$ (only the start symbol can derive ϵ)

All Context-free grammars can be rewritten to this form.

- 具体算法：在图中按照如下方式不断加边直至饱和



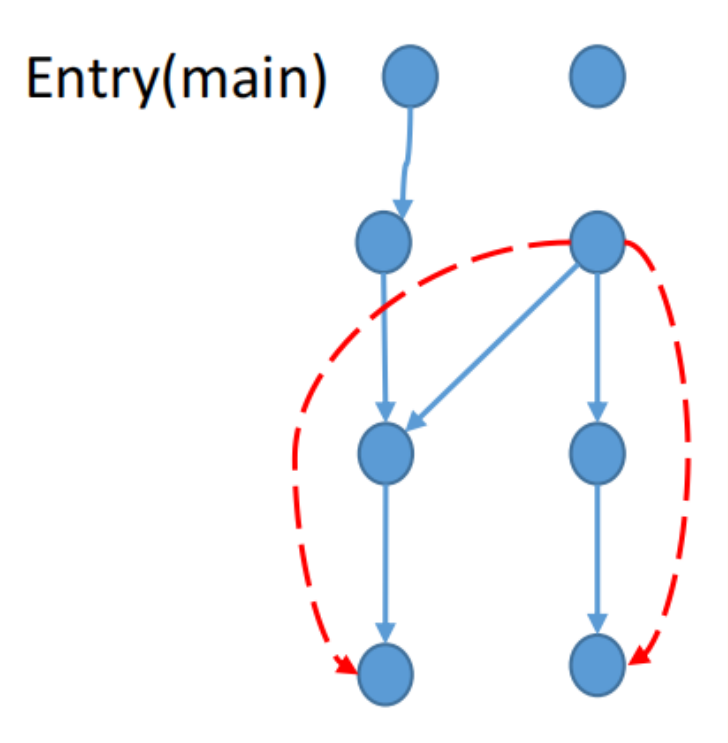
(a) $A \rightarrow \varepsilon$ (b) $A \rightarrow B$ (c) $A \rightarrow B\ C$

  - 以开始符号S标记的边即为答案
  - $\mathrm{O}(N^3)$ time, $\mathrm{O}(N^2)$ space，N是爆炸超图中的节点=超图节点$n * (D + 1)$

CYK算法缺陷

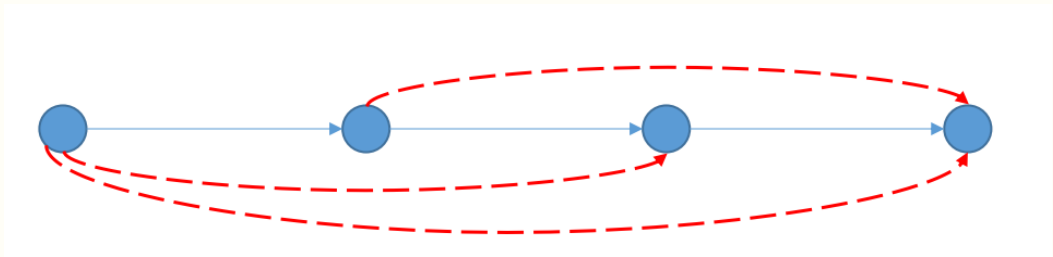
❑无效计算

- ➢只关心从起始点开始的可达性
- ➢CYK算法会计算过程内部可达性



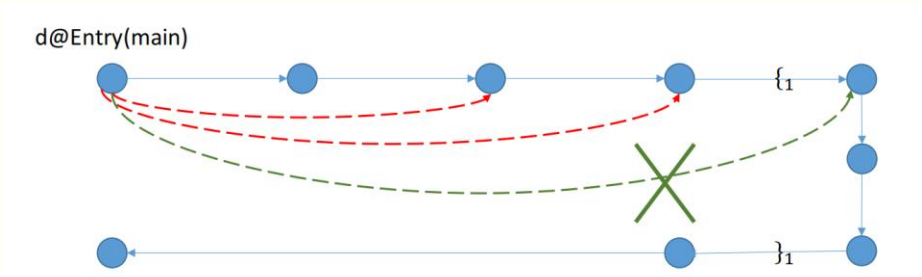
❑重复计算

- ➢一条边可能从几个不同途径添加，导致重复计算

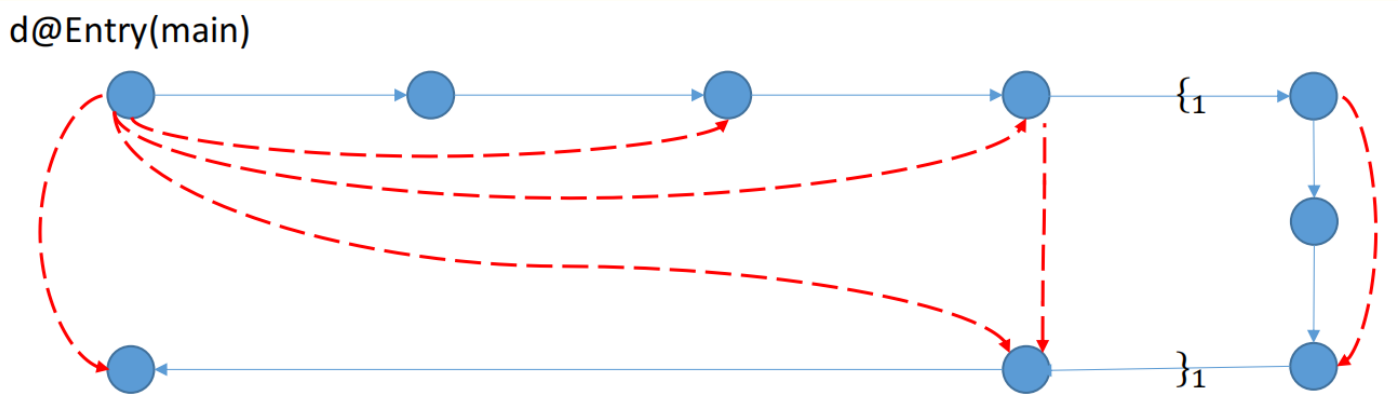


计算CFL可达性：Tabulation算法

- **只添加从函数entry出发的边**
  - 不会对entry不可达路径进行无效计算
  - 因为固定顺序，不会产生重复计算
  - 不添加从entry到被调用函数的entry的边



- **标记所有从entry可达的过程**
  - 利用和上面一样的方法，继续添加从entry可达的过程开始位置的边
- **对调用语句出发的边应用函数摘要**



Tabulation算法

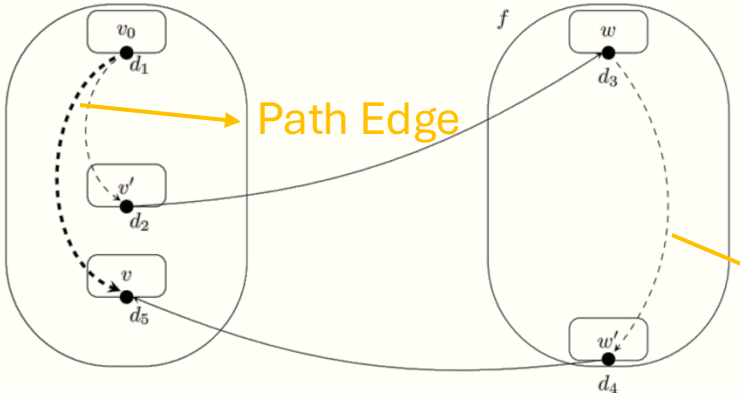
- $O(ED^3)$ time，更高效
  - E是超图边数，D是数据流值数
  - 路径边和摘要边
    - $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$

从main可达的过程，接着从entry开始可达性分析

尝试使用已有摘要

使用摘要的过程确保了括号匹配

新添函数摘要，并尝试使用新摘要



普通的图可达性分析

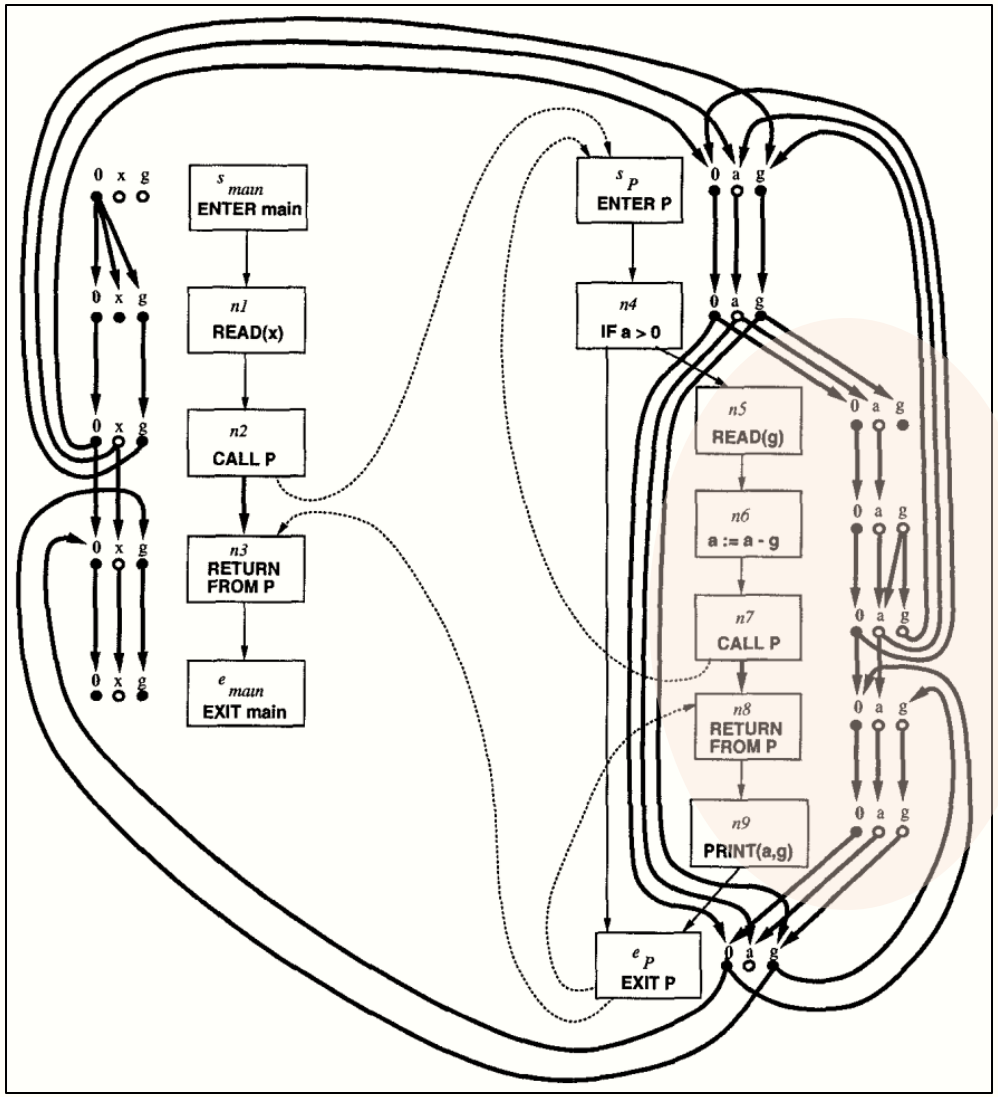
```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate(G#_IP)
begin
[1]   Let (N#, E#) = G#_IP
[2]   PathEdge := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
[3]   WorkList := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
[4]   SummaryEdge := ∅
[5]   ForwardTabulateSLRPs()
[6]   for each n ∈ N* do
[7]     X_n := { d2 ∈ D | ∃ d1 ∈ (D ∪ { 0 }) such that ⟨s_procOf(n), d1⟩ → ⟨n, d2⟩ ∈ PathEdge }
[8]   od
end

procedure Propagate(e)
begin
[9]   if e ∉ PathEdge then Insert e into PathEdge; Insert e into WorkList fi
end

procedure ForwardTabulateSLRPs()
begin
[10]  while WorkList ≠ ∅ do
[11]    Select and remove an edge ⟨s_p, d1⟩ → ⟨n, d2⟩ from WorkList
[12]    switch n
[13]      case n ∈ Call_p :
[14]        for each d3 such that ⟨n, d2⟩ → ⟨s_calledProc(n), d3⟩ ∈ E# do
[15]          Propagate(⟨s_calledProc(n), d3⟩ → ⟨s_calledProc(n), d3⟩)
[16]        od
[17]        for each d3 such that ⟨n, d2⟩ → ⟨returnSite(n), d3⟩ ∈ (E# ∪ SummaryEdge) do
[18]          Propagate(⟨s_p, d1⟩ → ⟨returnSite(n), d3⟩)
[19]        od
[20]      end case
[21]      case n = e_p :
[22]        for each c ∈ callers(p) do
[23]          for each d4, d5 such that ⟨c, d4⟩ → ⟨s_p, d1⟩ ∈ E# and ⟨e_p, d2⟩ → ⟨returnSite(c), d5⟩ ∈ E# do
[24]            if ⟨c, d4⟩ → ⟨returnSite(c), d5⟩ ∉ SummaryEdge then
[25]              Insert ⟨c, d4⟩ → ⟨returnSite(c), d5⟩ into SummaryEdge
[26]              for each d3 such that ⟨s_procOf(c), d3⟩ → ⟨c, d4⟩ ∈ PathEdge do
[27]                Propagate(⟨s_procOf(c), d3⟩ → ⟨returnSite(c), d5⟩)
[28]              od
[29]            fi
[30]          od
[31]        od
[32]      end case
[33]      case n ∈ (N_p − Call_p − { e_p }) :
[34]        for each ⟨m, d3⟩ such that ⟨n, d2⟩ → ⟨m, d3⟩ ∈ E# do
[35]          Propagate(⟨s_p, d1⟩ → ⟨m, d3⟩)
[36]        od
[37]      end case
[38]    end switch
[39]  od
end
```

例子：可能未初始化变量问题

- **右图实心点表示从$\langle s_{main}, 0\rangle$可达**
  - 变量g在n5,n6,n7,n8,n9处不可达
  - 在其他点处可达
  - g是可能未初始化变量



IDE问题：IFDS向无限数据域的扩展

- **IFDS要求data facts元素数量是有限的**
- **IDE(Interprocedural Distributive Environment)问题：**
  - Environments $Env(D, L)$: a set of functions from D to L
    - D: a finite set of program symbols (same as in IFDS)
    - L: a finite-height meet semi-lattice with a top element $\top$
      - L可看作是一个函数，可以表示无限数量的value
    - Meet operator: $env_1 \sqcap env_2$ is $\lambda \mathrm{d}.\,(env_1(\mathrm{d}) \sqcap env_2(\mathrm{d}))$
  - Flow function (environment transformer): $t: Env(D, L) \rightarrow Env(D, L)$
    - **distributive**: $\forall env_i \in Env(D, L), \text{and } d \in D,\ (t(\sqcap_i env_i))(d) = \sqcap_i (t(env_i))(d)$
    - 意味着可以独立为每个$d \in D$计算其在不同程序点对应的$l \in L$的值，D和L都是有限的，所以迭代可以终止
- **适合用于计算Linear Constant Propagation**
  - $F_{lc} = \{\lambda \mathrm{l}.\,(\mathrm{a} * \mathrm{l} + \mathrm{b}) \sqcap \mathrm{c} \mid \mathrm{a} \in \mathrm{Z} - \{0\}, \mathrm{b} \in \mathrm{Z}, \text{and } \mathrm{c} \in Z_{\perp}^{\top}\}$
  - $\mathrm{f} \in F_{lc}$: $(\mathrm{a}, \mathrm{b}, \mathrm{c})$ $\quad f = \lambda l. \begin{cases} \top & l = \top \\ (a * l + b) \sqcap c & \text{otherwise} \end{cases}$

(Optional) IDE问题的Tabulation算法

```
procedure ComputePathFunctions()
begin
  for all ⟨s_p, d'⟩, ⟨m, d⟩ such that m occurs in procedure p and d', d ∈ D ∪ {Λ} do
    PathFn(⟨s_p, d'⟩, ⟨m, d⟩) = λl.⊤ od
  for all corresponding call-return pairs c, r and d', d ∈ D ∪ {Λ} do
    SummaryFn(⟨c, d'⟩, ⟨r, d⟩) = λl.⊤ od
  WorkList:= {⟨s_main, Λ⟩ → ⟨s_main, Λ⟩}
  PathFn(⟨s_main, Λ⟩ → ⟨s_main, Λ⟩) := id
  while WorkList ≠ ∅ do
    Select and remove an edge ⟨s_p, d1⟩ → ⟨n, d2⟩ from WorkList
    let f = PathFn(⟨s_p, d1⟩ → ⟨n, d2⟩)
    switch(n)
      case n is a call node in p, calling a procedure q:
        for each d3 s.t. ⟨n, d2⟩ → ⟨s_q, d3⟩ ∈ E# do
          Propagate (⟨s_q, d3⟩ → ⟨s_q, d3⟩, id) od
        let r be the return-site node that corresponds to n
        for each d3 s.t. e = ⟨n, d2⟩ → ⟨r, d3⟩ ∈ E# do
          Propagate(⟨s_p, d1⟩ → ⟨r, d3⟩, EdgeFn(e) o f) od
        for each d3 s.t. f3 = SummaryFn(⟨n, d2⟩ → ⟨r, d3⟩) ≠ λl.⊤ do
          Propagate(⟨s_p, d1⟩ → ⟨r, d3⟩, f3 o f) od endcase
      case n is the exit node of p:
        for each call node c that calls p with corresponding return-site node r do
          for each d4, d5 s.t. ⟨c, d4⟩ → ⟨s_p, d1⟩ ∈ E# and ⟨e_p, d2⟩ → ⟨r, d5⟩ ∈ E# do
            let f4 = EdgeFn(⟨c, d4⟩ → ⟨s_p, d1⟩) and
                f5 = EdgeFn(⟨e_p, d2⟩ → ⟨r, d5⟩) and
                f' = (f5 o f o f4)⊓SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩)
            if f' ≠ SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩) then
              SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩) := f'
              let s_q be the start node of c's procedure
              for each d3 s.t. f3 = PathFn(⟨s_q, d3⟩ → ⟨c, d4⟩) ≠ λl.⊤ do
                Propagate(⟨s_q, d3⟩ → ⟨r, d5⟩, f' o f3) od fi od od endcase
      default:
        for each ⟨m, d3⟩ s.t. ⟨n, d2⟩ → ⟨m, d3⟩ ∈ E# do
          Propagate(⟨s_p, d1⟩ → ⟨m, d3⟩, EdgeFn(⟨n, d2⟩ → ⟨m, d3⟩) o f) od endcase
    end switch od
end
procedure Propagate(e, f)
begin
  let f' = f⊓PathFn(e)
  if f' ≠ PathFn(e) then
    PathFn(e) := f'
    Insert e into WorkList fi
end
```

FIGURE 3. The algorithm for Phase I.

算法复杂度$O(ED^3)$

```
procedure ComputeValues()
begin
  /* Phase II(i) */
  for each en ∈ N# do val(en) := ⊤ od
  val(⟨s_main, Λ⟩) := ⊥
  WorkList:= {⟨s_main, Λ⟩}
  while WorkList ≠ ∅ do
    Select and remove an exploded-graph node ⟨n, d⟩ from WorkList
    switch(n)
      case n is the start node of p:
        for each c that is a call node inside p do
          for each d' s.t. f' = PathFn(⟨n, d⟩ → ⟨c, d'⟩) ≠ λl.⊤ do
            PropagateValue(⟨c, d'⟩,f'(val(⟨s_p, d⟩))) od od endcase
      case n is a call node in p, calling a procedure q:
        for each d' s.t. ⟨n, d⟩ → ⟨s_q, d'⟩ ∈ E# do
          PropagateValue(⟨s_q, d'⟩, EdgeFn(⟨n, d⟩ → ⟨s_q, d'⟩)(val(⟨n, d⟩))) od endcase
    end switch od
  /* Phase II(ii) */
  for each node n, in a procedure p, that is not a call or a start node do
    for each d', d s.t. f' = PathFn(⟨s_p, d'⟩ → ⟨n, d⟩) ≠ λl.⊤ do
      val(⟨n, d⟩) := val(⟨n, d⟩) ⊓ f'(val(⟨s_p, d'⟩)) od od
end
procedure PropagateValue(en, v)
begin
    let v' = v ⊓ val(en)
    if v' ≠ val(en) then
      val(en) := v'
      Insert en into WorkList fi
end
```

FIGURE 4. The algorithm for Phase II.

计算每个程序点的values：$\langle \mathrm{n}, d_2 \rangle \to \mathrm{L}$
此时的L是从s_{main}计算到n值

计算PathEdges以及其上的PathFunctions
$(\langle s_p, d_1 \rangle \to \langle \mathrm{n}, d_2 \rangle) \to (\mathrm{L} \to \mathrm{L})$，这里L是过程p中的局部值

(optional)作业：学习IFDS/IDE框架实现

- IFDS/IDE框架的Java实现：
  - https://github.com/soot-oss/heros
- IFDS/IDE框架的C/C++实现：
  - https://github.com/secure-software-engineering/phasar/tree/development/include/phasar/DataFlow/IfdsIde
- SparseIDE框架实现：
  - https://github.com/secure-software-engineering/SparseIDE
- 选择一个实现学习，写一写自己的理解和心得体会