



软件分析与架构设计

CFL可达性：理论与算法

何冬杰
重庆大学

指称语义 (Denotation Semantics)

□ Revisit the flow function $f: L \rightarrow L$ 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

$S ::= x := a$	$b ::= \text{true}$	$a ::= x$	$op_b ::= \text{and} \mid \text{or}$
skip	false	n	$op_r ::= < \mid \leq \mid =$
$S_1; S_2$	not b	$a_1 op_a a_2$	$> \mid \geq$
if b then S_1 else S_2	$b_1 op_b b_2$		$op_a ::= + \mid - \mid * \mid /$
while b do S	$a_1 op_r a_2$		

Abstract Syntax

指称语义 (Denotation Semantics)

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

$$\mathcal{A}[n] = \{(\sigma, n) \mid \sigma \in \Sigma\}$$

$$\mathcal{A}[X] = \{(\sigma, \sigma(X)) \mid \sigma \in \Sigma\}$$

$$\mathcal{A}[a_0 + a_1] = \{(\sigma, n_0 + n_1) \mid (\sigma, n_0) \in \mathcal{A}[a_0] \ \& \ (\sigma, n_1) \in \mathcal{A}[a_1]\}$$

$$\mathcal{A}[a_0 - a_1] = \{(\sigma, n_0 - n_1) \mid (\sigma, n_0) \in \mathcal{A}[a_0] \ \& \ (\sigma, n_1) \in \mathcal{A}[a_1]\}$$

$$\mathcal{A}[a_0 \times a_1] = \{(\sigma, n_0 \times n_1) \mid (\sigma, n_0) \in \mathcal{A}[a_0] \ \& \ (\sigma, n_1) \in \mathcal{A}[a_1]\}$$

$$\mathcal{A}[n] = \lambda\sigma \in \Sigma. n$$

$$\mathcal{A}[X] = \lambda\sigma \in \Sigma. \sigma(X)$$

$$\mathcal{A}[a_0 + a_1] = \lambda\sigma \in \Sigma. (\mathcal{A}[a_0]\sigma + \mathcal{A}[a_1]\sigma)$$

$$\mathcal{A}[a_0 - a_1] = \lambda\sigma \in \Sigma. (\mathcal{A}[a_0]\sigma - \mathcal{A}[a_1]\sigma)$$

$$\mathcal{A}[a_0 \times a_1] = \lambda\sigma \in \Sigma. (\mathcal{A}[a_0]\sigma \times \mathcal{A}[a_1]\sigma)$$

Equivalent
 λ -notation



□ 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: $\mathcal{B}: \text{Bexp} \rightarrow (\Sigma \rightarrow T)$

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

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

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

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

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

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

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

- Each denotation is a function
- For example, for all $\sigma \in \Sigma$,

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

指称语义 (Denotation Semantics)

□ Denotations of Com: $\mathcal{C}: \text{Com} \rightarrow (\Sigma \rightarrow \Sigma)$

➤ Com stands for commands/statements/Instructions

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

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

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

$$\begin{aligned} \mathcal{C}[\text{if } b \text{ then } c_0 \text{ else } c_1] = \\ \{(\sigma, \sigma') \mid \mathcal{B}[b]\sigma = \text{true} \ \& \ (\sigma, \sigma') \in \mathcal{C}[c_0]\} \cup \\ \{(\sigma, \sigma') \mid \mathcal{B}[b]\sigma = \text{false} \ \& \ (\sigma, \sigma') \in \mathcal{C}[c_1]\} \end{aligned}$$

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

—————> Let us explain why this is.

where

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

指称语义 (Denotation Semantics)

□ **Denotations of Com:** $\mathcal{C}: \text{Com} \rightarrow (\Sigma \rightarrow \Sigma)$

➤ $w \equiv \text{while } b \text{ do } c$ (use w to denote while b do c)

➤ $w \sim \text{if } b \text{ then } c; w \text{ else skip}$

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



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

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

语义等价性

Lemma 5.3 *For all $a \in \mathbf{Aexp}$,*

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

Lemma 5.4 *For $b \in \mathbf{Bexp}$,*

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

Theorem 5.7 *For all commands c*

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

- 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:** $c_0 \sim c_1$ iff $\llbracket c_0 \rrbracket \Leftrightarrow \llbracket c_1 \rrbracket$

- $\llbracket c_0 \rrbracket \Leftrightarrow \llbracket c_1 \rrbracket$ means $\forall \sigma. \llbracket c_0 \rrbracket(\sigma) \Leftrightarrow \llbracket c_1 \rrbracket(\sigma)$
- $\llbracket c_0 \rrbracket \Leftrightarrow \llbracket c_1 \rrbracket$ also means $\forall \sigma, \sigma'. \langle c_0, \sigma \rangle \rightarrow \sigma' \Leftrightarrow \langle c_1, \sigma \rangle \rightarrow \sigma'$
- If c_0 and c_1 are semantically equivalent and c_1 is more efficient in terms of running time or memory, the compiler would be happy using c_1 to replace c_0 during its optimization passes

□ **Function/Method Summary (函数摘要):**

- Let c_0 and c_1 be two call instructions to the same method m
- Then $\llbracket c_0 \rrbracket \Leftrightarrow \llbracket c_1 \rrbracket$,
- given a state σ , if we have $\sigma' = \llbracket c_0 \rrbracket(\sigma)$, we conclude $\sigma' = \llbracket c_1 \rrbracket(\sigma)$
- 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
```

此处g可能未被初始化

```
procedure P(value a: integer)
```

```
begin
```

```
  if (a > 0) then
```

```
    read(g)
```

```
    a := a - g
```

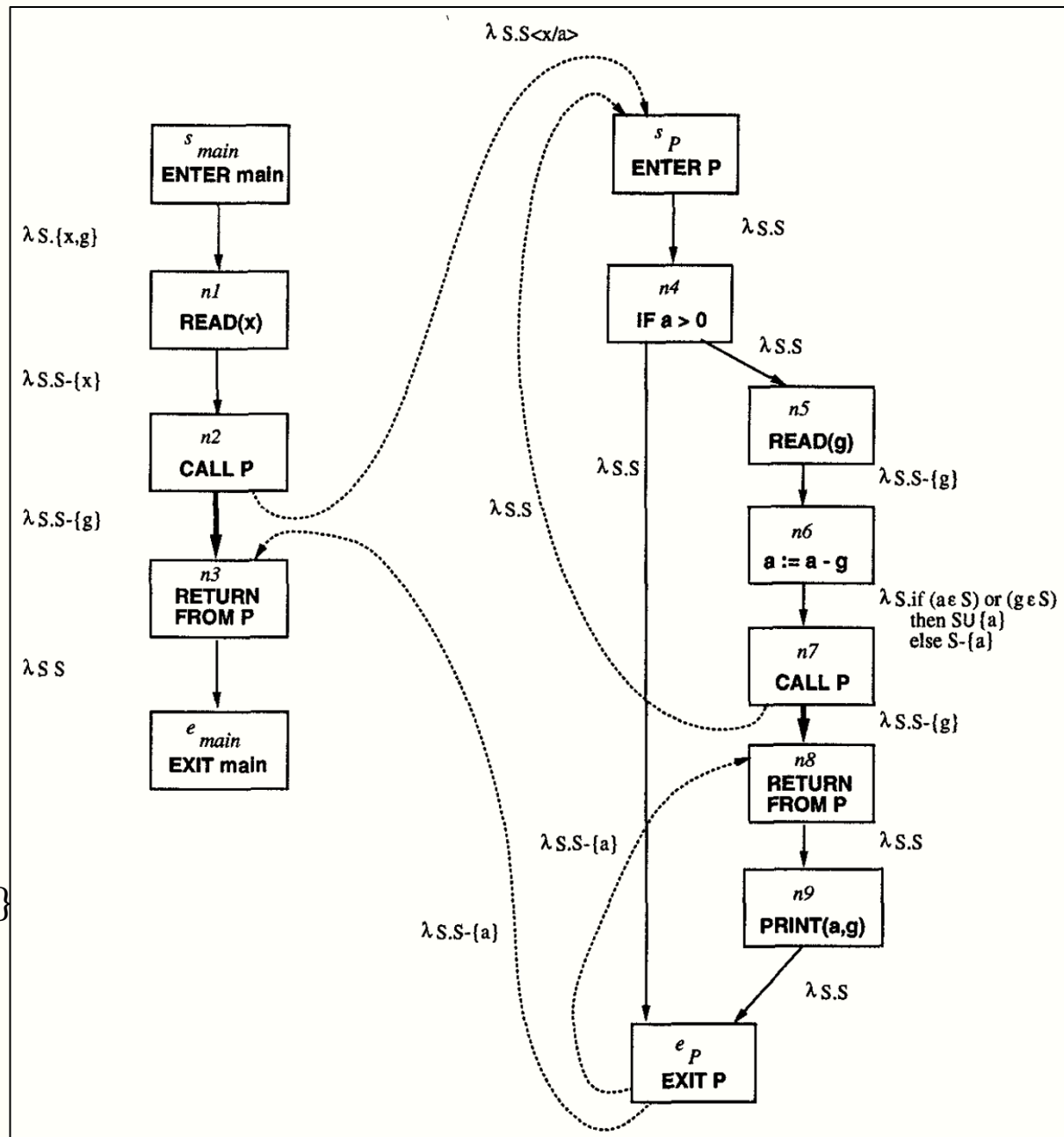
```
    call P(a)
```

```
    print(a, g)
```

```
  fi
```

```
end
```

- 边上的lambdas是流函数
- 对应source节点的指称语义
- 比如 $\llbracket READ(x) \rrbracket = \lambda S.S - \{x\}$
- 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

$$S \rightarrow \{_1 S \}_1 \mid \{_2 S \}_2 \mid \dots \mid SS \mid \epsilon$$

□ 通过括号匹配去除不可行路径

➤ Feasible path:

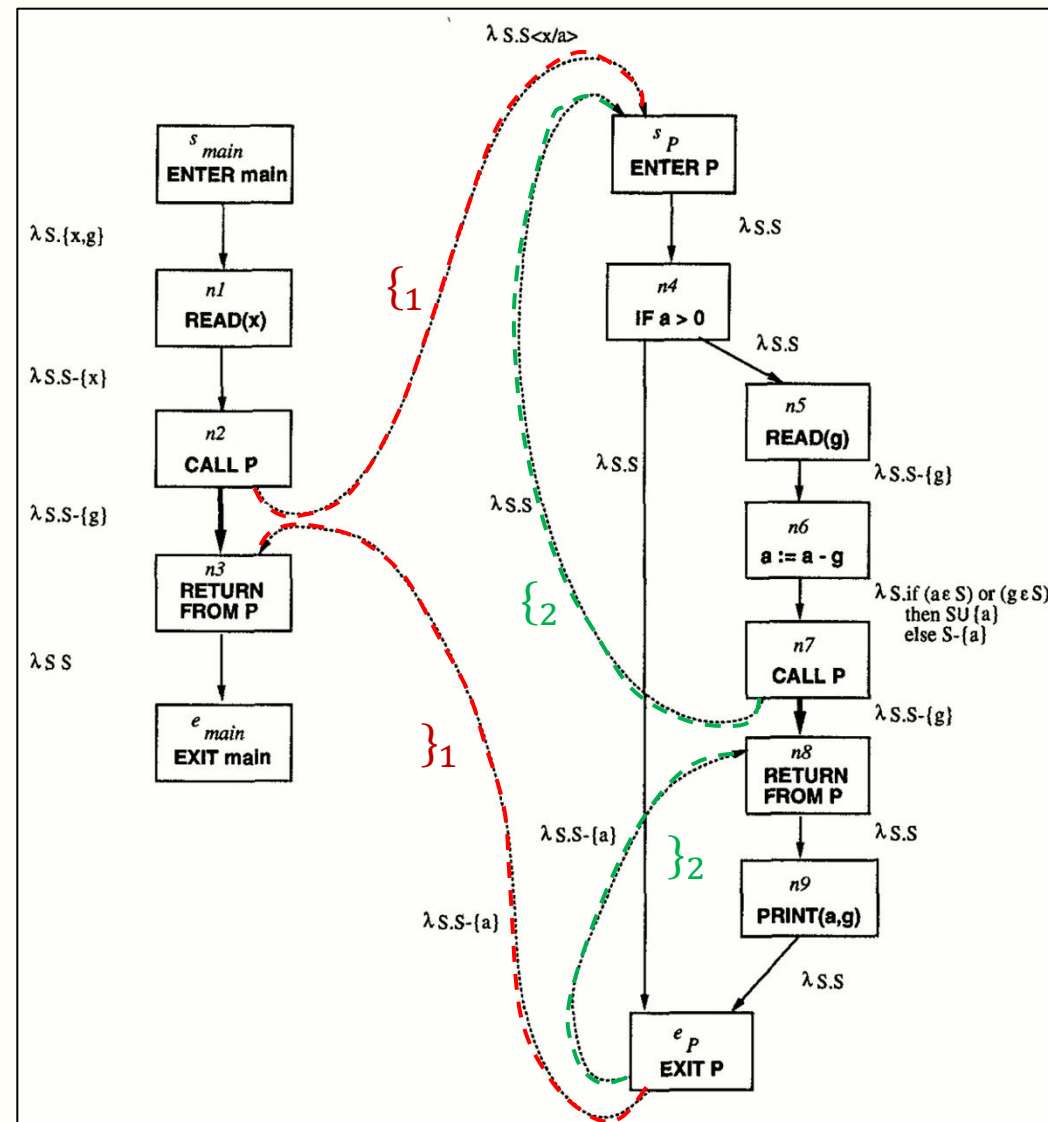
○ $S_{main}, n_2, \{_1, S_P, \dots, e_P, \}_1, n_3, e_{main}$

➤ Infeasible path:

○ $S_{main}, n_2, \{_1, S_P, \dots, e_P, \}_2, n_8, e_P, \dots$

○ $n_7, \{_2, S_P, \dots, e_P, \}_1, n_3, e_{main}$

➤ 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)

$$\forall v \in Insts, x, y \in \Sigma:$$

$$f[[v]](x) \sqcup f[[v]](y) = f[[v]](x \sqcup y)$$

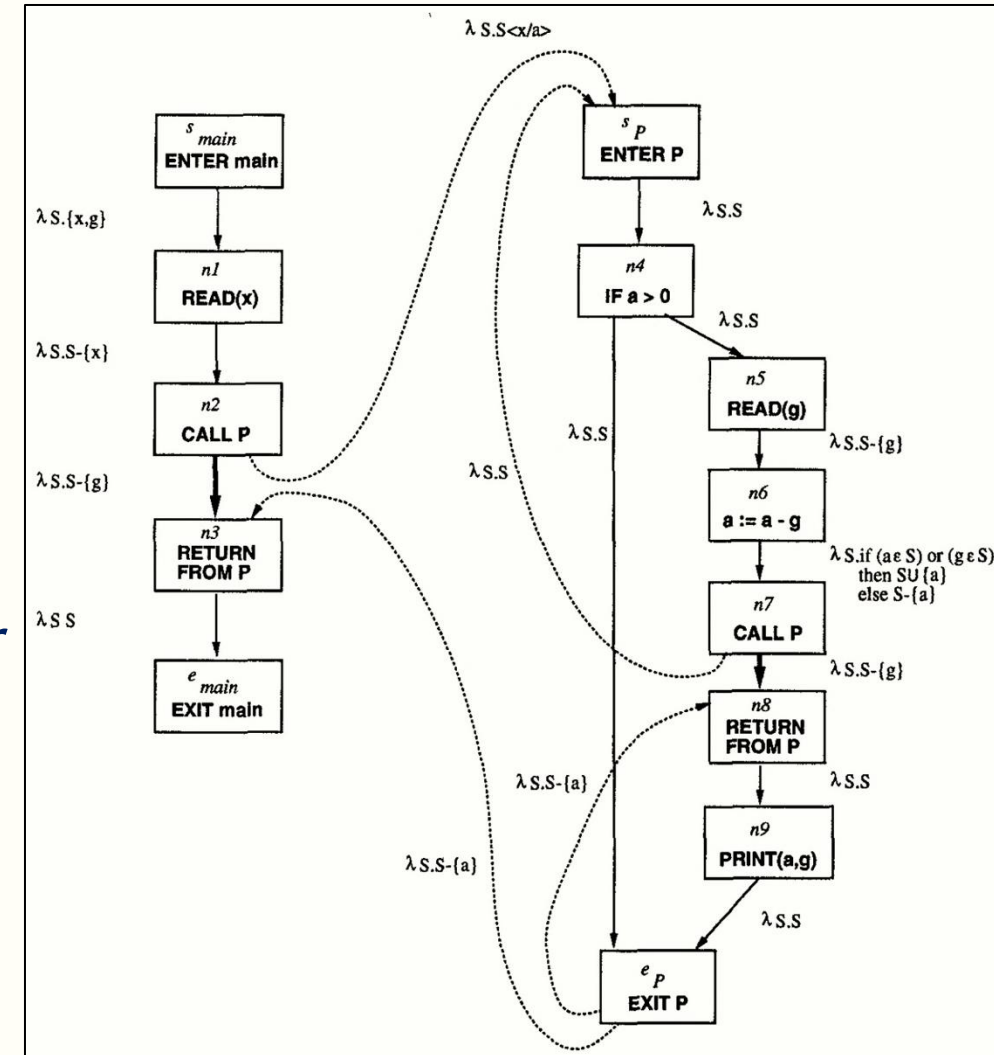
➤ Hold for composition:

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: $\lambda S. S - \{g\}$

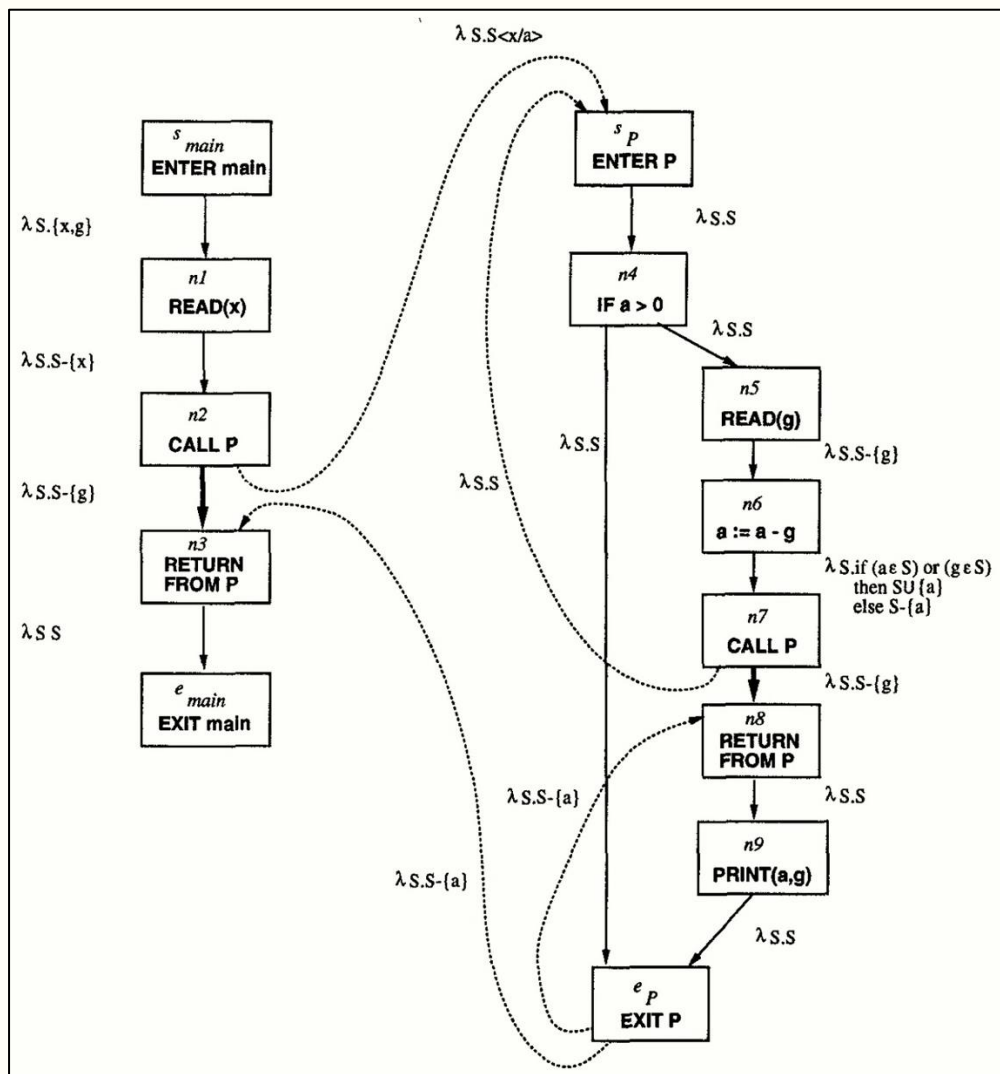


如何精确计算函数指称语义[[m]]?

□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**
- Supergraph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



$$\llbracket P \rrbracket = \llbracket e_p \rrbracket \circ \llbracket \text{IF } a > 0 \rrbracket \circ \llbracket s_p \rrbracket \sqcup \dots = \lambda S.S \sqcup \dots = \lambda S.S$$

接下来计算 $\llbracket \text{main} \rrbracket$:

$$\begin{aligned} \llbracket \text{main} \rrbracket &= \llbracket e_{main} \rrbracket \circ \llbracket \text{call P} \rrbracket \circ \llbracket \text{READ}(x) \rrbracket \circ \llbracket s_{main} \rrbracket \\ &= \lambda S.S \circ (\lambda S.S - \{g\} \sqcup \lambda S.S) \circ \lambda S.S - \{x\} \circ \lambda S.\{x, g\} \\ &= \lambda S.S \circ (\lambda S.S - \{g\} \sqcup \lambda S.S) \circ \lambda S.\{g\} \\ &= \lambda S.S \circ \lambda S.S \circ \lambda S.\{g\} = \lambda S.\{g\} \end{aligned}$$

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 (Interprocedural, Finite, Distributive, Subset) problems

- 要求configuration或dataflow facts元素数量有限
- 要求flow functions满足分配性: $\forall v \in Insts, x, y \in S: f_v(x) \sqcup f_v(y) = f_v(x \sqcup y)$

❖ 意味着每一条路径的数据流计算可以并行处理

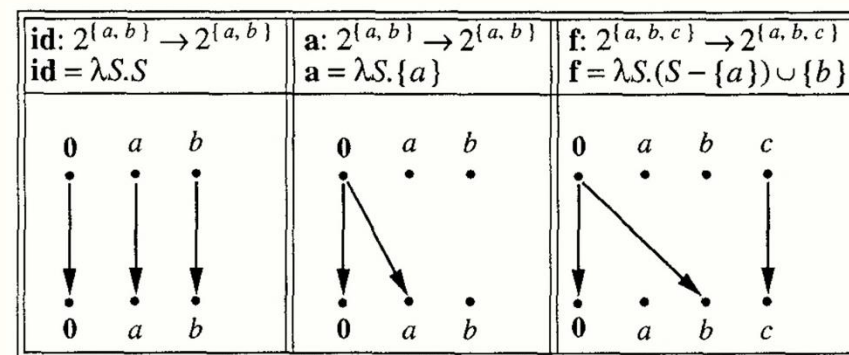
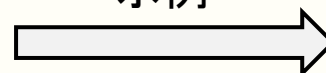
□将数据流分析问题转化CFL-可达性问题

➤将流函数转为二元关系 (即图)

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

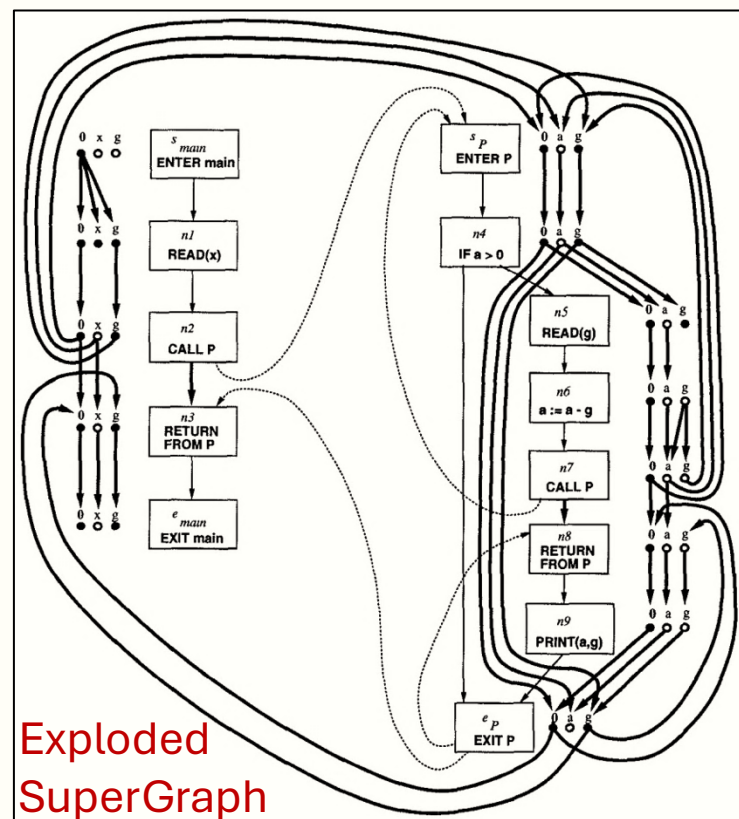
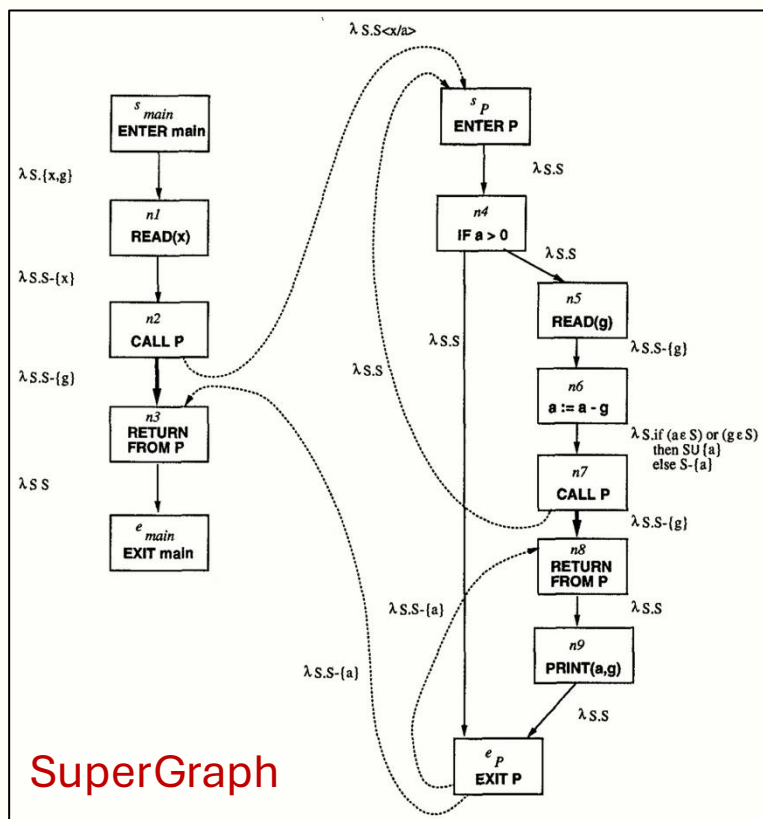
$$R_f =_{df} \begin{aligned} & \{ (0, 0) \} \\ & \cup \{ (0, y) \mid y \in f(\emptyset) \} \\ & \cup \{ (x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset) \}. \end{aligned} \quad \square$$

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转化成二元关系，获得爆炸超图



□跨过程的边标上 $\{i \text{ 或 } i\}$ ，使用CFL语言过滤infeasible path

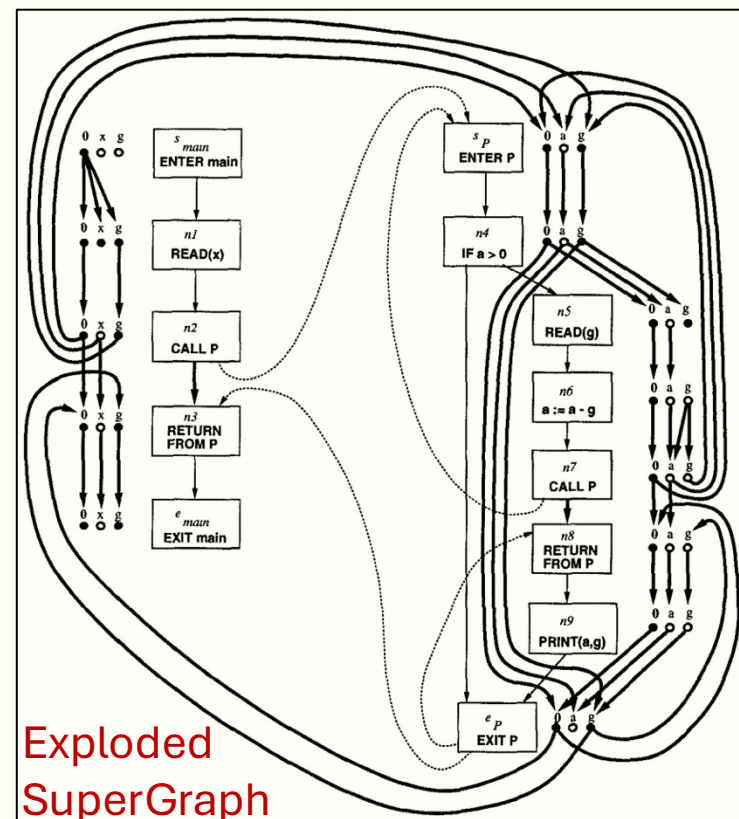
将IFDS问题转化CFL-可达性问题

□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当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 $\llbracket m \rrbracket$
- 右图容易得到:
 - $\llbracket P \rrbracket = \lambda S. S$
 - $\llbracket \text{main} \rrbracket = \lambda S. \{0, g\}$



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

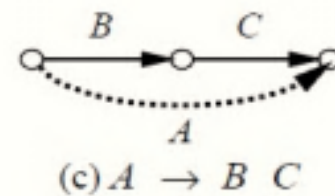
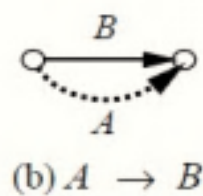
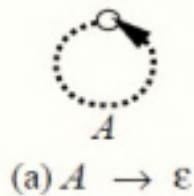
- Invented by John Cocke, Daniel Younger, and Tadao Kasami

□假定CFL产生式为Chomsky Normal Form形式

- $A \rightarrow BC$ (A, B, C are non-terminals) or
- $A \rightarrow d$ (d is a terminal) or
- $S \rightarrow \epsilon$ (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

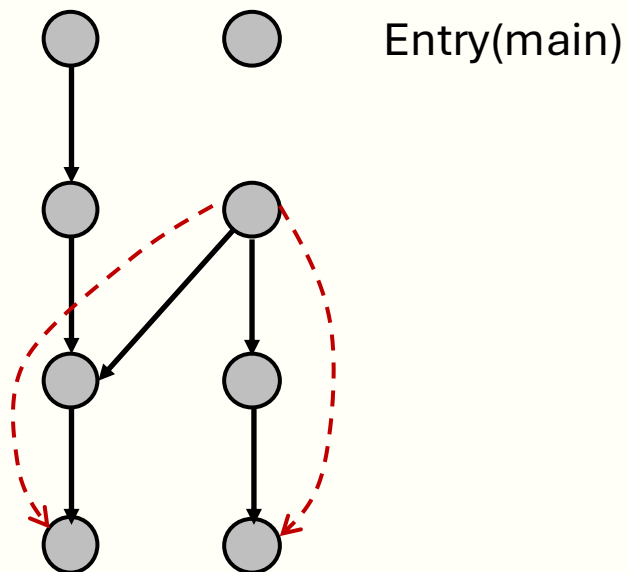


- 以开始符号S标记的边即为答案
- $O(N^3)$ time, $O(N^2)$ space, N 是爆炸超图中的节点=超图节点 $n * (D + 1)$

CYK算法缺陷

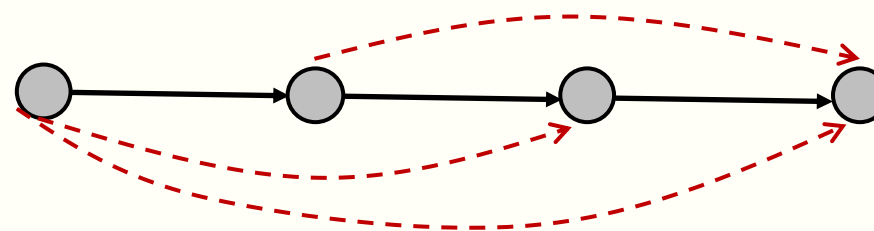
□无效计算

- 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

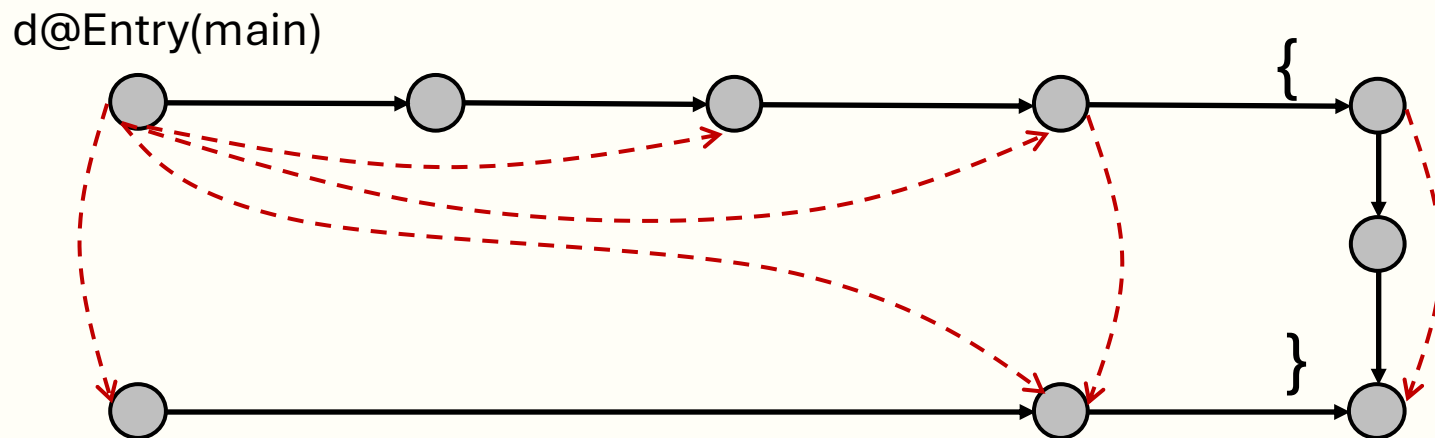
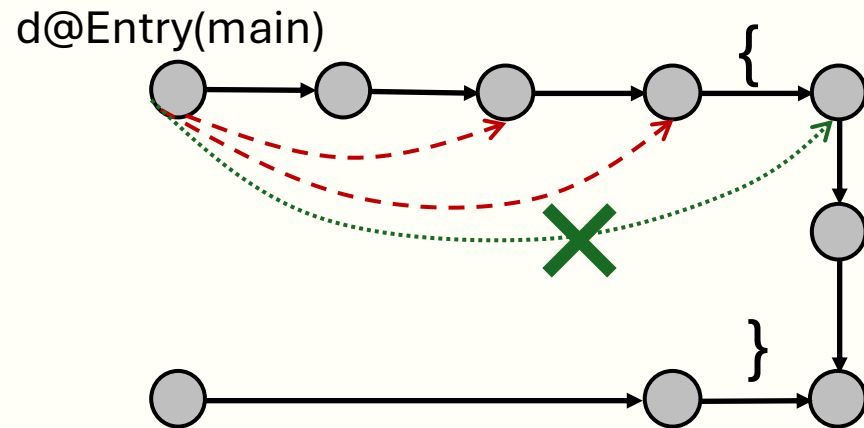
□只添加从函数entry出发的边

- 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

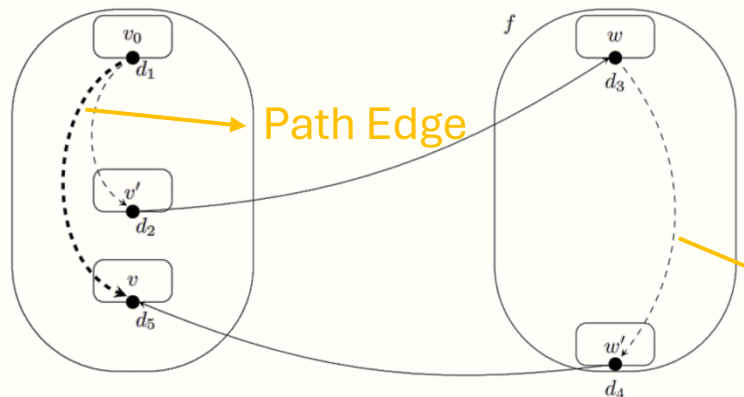
尝试使用已有摘要

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

Summary Edge



```

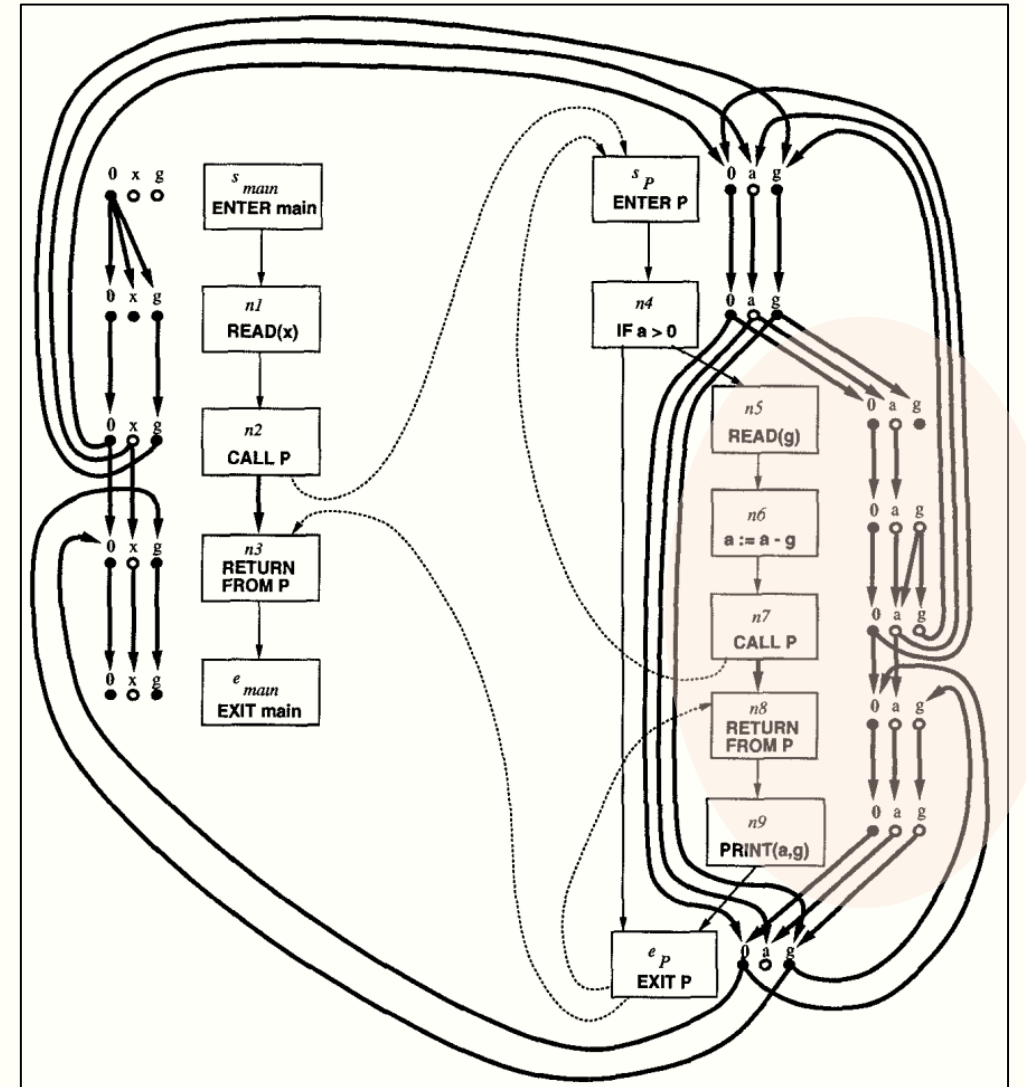
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate( $G_{TP}^{\#}$ )
begin
[1] Let  $(N^{\#}, E^{\#}) = G_{TP}^{\#}$ 
[2] PathEdge :=  $\{ \langle s_{main}, \mathbf{0} \rangle \rightarrow \langle s_{main}, \mathbf{0} \rangle \}$ 
[3] WorkList :=  $\{ \langle s_{main}, \mathbf{0} \rangle \rightarrow \langle s_{main}, \mathbf{0} \rangle \}$ 
[4] SummaryEdge :=  $\emptyset$ 
[5] ForwardTabulateSLRPs()
[6] for each  $n \in N^*$  do
[7]    $X_n := \{ d_2 \in D \mid \exists d_1 \in (D \cup \{ \mathbf{0} \}) \text{ such that } \langle s_{procOf(n)}, d_1 \rangle \rightarrow \langle n, d_2 \rangle \in \text{PathEdge} \}$ 
[8]   od
end
procedure Propagate( $e$ )
begin
[9]   if  $e \in \text{PathEdge}$  then Insert  $e$  into PathEdge; Insert  $e$  into WorkList fi
end
procedure ForwardTabulateSLRPs()
begin
[10] while WorkList  $\neq \emptyset$  do
[11]   Select and remove an edge  $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$  from WorkList
[12]   switch  $n$ 
[13]     case  $n \in \text{Call}_p$  :
[14]       for each  $d_3$  such that  $\langle n, d_2 \rangle \rightarrow \langle s_{calledProc(n)}, d_3 \rangle \in E^{\#}$  do
[15]         Propagate( $\langle s_{calledProc(n)}, d_3 \rangle \rightarrow \langle s_{calledProc(n)}, d_3 \rangle$ )
[16]       od
[17]       for each  $d_3$  such that  $\langle n, d_2 \rangle \rightarrow \langle \text{returnSite}(n), d_3 \rangle \in (E^{\#} \cup \text{SummaryEdge})$  do
[18]         Propagate( $\langle s_p, d_1 \rangle \rightarrow \langle \text{returnSite}(n), d_3 \rangle$ )
[19]       od
[20]     end case
[21]     case  $n = e_p$  :
[22]       for each  $c \in \text{callers}(p)$  do
[23]         for each  $d_4, d_5$  such that  $\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle \in E^{\#}$  and  $\langle e_p, d_2 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle \in E^{\#}$  do
[24]           if  $\langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle \notin \text{SummaryEdge}$  then
[25]             Insert  $\langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle$  into SummaryEdge
[26]             for each  $d_3$  such that  $\langle s_{procOf(c)}, d_3 \rangle \rightarrow \langle c, d_4 \rangle \in \text{PathEdge}$  do
[27]               Propagate( $\langle s_{procOf(c)}, d_3 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle$ )
[28]             od
[29]           fi
[30]         od
[31]       od
[32]     end case
[33]     case  $n \in (N_p - \text{Call}_p - \{ e_p \})$  :
[34]       for each  $\langle m, d_3 \rangle$  such that  $\langle n, d_2 \rangle \rightarrow \langle m, d_3 \rangle \in E^{\#}$  do
[35]         Propagate( $\langle s_p, d_1 \rangle \rightarrow \langle m, d_3 \rangle$ )
[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 可看作是一个抽象值, 可以表示无限数量的value

○ Meet operator: $env_1 \sqcap env_2$ is $\lambda d. (env_1(d) \sqcap env_2(d))$

➤ 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

➤ $F_{lc} = \{\lambda l. (a * l + b) \sqcap c \mid a \in \mathbb{Z} - \{0\}, b \in \mathbb{Z}, \text{ and } c \in \mathbb{Z}_{\perp}^{\top}\}$

➤ $f \in F_{lc}: (a, b, c) \quad f = \lambda l. \begin{cases} \top & l = \top \\ (a * l + b) \sqcap c & \text{otherwise} \end{cases}$

IDE问题的Tabulation算法

```

procedure ComputePathFunctions()
begin
  for all  $\langle s_p, d' \rangle, \langle m, d \rangle$  such that  $m$  occurs in procedure  $p$  and  $d', d \in D \cup \{\Lambda\}$  do
     $PathFn(\langle s_p, d' \rangle, \langle m, d \rangle) = \lambda l. \top$  od
  for all corresponding call-return pairs  $c, r$  and  $d', d \in D \cup \{\Lambda\}$  do
     $SummaryFn(\langle c, d' \rangle, \langle r, d \rangle) = \lambda l. \top$  od
   $WorkList := \{\langle s_{main}, \Lambda \rangle \rightarrow \langle s_{main}, \Lambda \rangle\}$ 
   $PathFn(\langle s_{main}, \Lambda \rangle \rightarrow \langle s_{main}, \Lambda \rangle) := id$ 
  while  $WorkList \neq \emptyset$  do
    Select and remove an edge  $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$  from  $WorkList$ 
    let  $f = PathFn(\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle)$ 
    switch( $n$ )
      case  $n$  is a call node in  $p$ , calling a procedure  $q$ :
        for each  $d_3$  s.t.  $\langle n, d_2 \rangle \rightarrow \langle s_q, d_3 \rangle \in E^\#$  do
          Propagate  $(\langle s_q, d_3 \rangle \rightarrow \langle s_q, d_3 \rangle, id)$  od
          let  $r$  be the return-site node that corresponds to  $n$ 
          for each  $d_3$  s.t.  $e = \langle n, d_2 \rangle \rightarrow \langle r, d_3 \rangle \in E^\#$  do
            Propagate  $(\langle s_p, d_1 \rangle \rightarrow \langle r, d_3 \rangle, EdgeFn(e) \circ f)$  od
          for each  $d_3$  s.t.  $f_3 = SummaryFn(\langle n, d_2 \rangle \rightarrow \langle r, d_3 \rangle) \neq \lambda l. \top$  do
            Propagate  $(\langle s_p, d_1 \rangle \rightarrow \langle r, d_3 \rangle, f_3 \circ f)$  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  $d_4, d_5$  s.t.  $\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle \in E^\#$  and  $\langle e_p, d_2 \rangle \rightarrow \langle r, d_5 \rangle \in E^\#$  do
              let  $f_4 = EdgeFn(\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle)$  and
               $f_5 = EdgeFn(\langle e_p, d_2 \rangle \rightarrow \langle r, d_5 \rangle)$  and
               $f' = (f_5 \circ f \circ f_4) \cap SummaryFn(\langle c, d_4 \rangle \rightarrow \langle r, d_5 \rangle)$ 
              if  $f' \neq SummaryFn(\langle c, d_4 \rangle \rightarrow \langle r, d_5 \rangle)$  then
                 $SummaryFn(\langle c, d_4 \rangle \rightarrow \langle r, d_5 \rangle) := f'$ 
                let  $s_q$  be the start node of  $c$ 's procedure
                for each  $d_3$  s.t.  $f_3 = PathFn(\langle s_q, d_3 \rangle \rightarrow \langle c, d_4 \rangle) \neq \lambda l. \top$  do
                  Propagate  $(\langle s_q, d_3 \rangle \rightarrow \langle r, d_5 \rangle, f' \circ f_3)$  od fi od endcase
            default:
              for each  $\langle m, d_3 \rangle$  s.t.  $\langle n, d_2 \rangle \rightarrow \langle m, d_3 \rangle \in E^\#$  do
                Propagate  $(\langle s_p, d_1 \rangle \rightarrow \langle m, d_3 \rangle, EdgeFn(\langle n, d_2 \rangle \rightarrow \langle m, d_3 \rangle) \circ f)$  od endcase
          end switch od
        end
      procedure Propagate( $e, f$ )
      begin
        let  $f' = f \cap PathFn(e)$ 
        if  $f' \neq PathFn(e)$  then
           $PathFn(e) := f'$ 
          Insert  $e$  into  $WorkList$  fi
      end

```

FIGURE 3. The algorithm for Phase I.

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

算法复杂度 $O(ED^3)$

$O(ED^3 \cdot L)$, 但L通常认为是常数

```

procedure ComputeValues()
begin
  /* Phase II(i) */
  for each  $en \in N^\#$  do  $val(en) := \top$  od
   $val(\langle s_{main}, \Lambda \rangle) := \perp$ 
   $WorkList := \{\langle s_{main}, \Lambda \rangle\}$ 
  while  $WorkList \neq \emptyset$  do
    Select and remove an exploded-graph node  $\langle n, d \rangle$  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.  $f' = PathFn(\langle n, d \rangle \rightarrow \langle c, d' \rangle) \neq \lambda l. \top$  do
            PropagateValue  $(\langle c, d' \rangle, f'(val(\langle s_p, d \rangle)))$  od od endcase
        case  $n$  is a call node in  $p$ , calling a procedure  $q$ :
          for each  $d'$  s.t.  $\langle n, d \rangle \rightarrow \langle s_q, d' \rangle \in E^\#$  do
            PropagateValue  $(\langle s_q, d' \rangle, EdgeFn(\langle n, d \rangle \rightarrow \langle s_q, d' \rangle)(val(\langle n, d \rangle)))$  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.  $f' = PathFn(\langle s_p, d' \rangle \rightarrow \langle n, d \rangle) \neq \lambda l. \top$  do
             $val(\langle n, d \rangle) := val(\langle n, d \rangle) \cap f'(val(\langle s_p, d' \rangle))$  od od
        end
      procedure PropagateValue( $en, v$ )
      begin
        let  $v' = v \cap val(en)$ 
        if  $v' \neq val(en)$  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: $\langle n, d_2 \rangle \rightarrow L$
 此时的L是从 s_{main} 计算到 n 值

(optional)项目：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

- 不少于1000字，语言言简意赅，不无病呻吟
- 可用示意图等辅助说明，图不计入字数