Static Program Analysis

Yue Li and Tian Tan

2020 Spring

Static Program Analysis

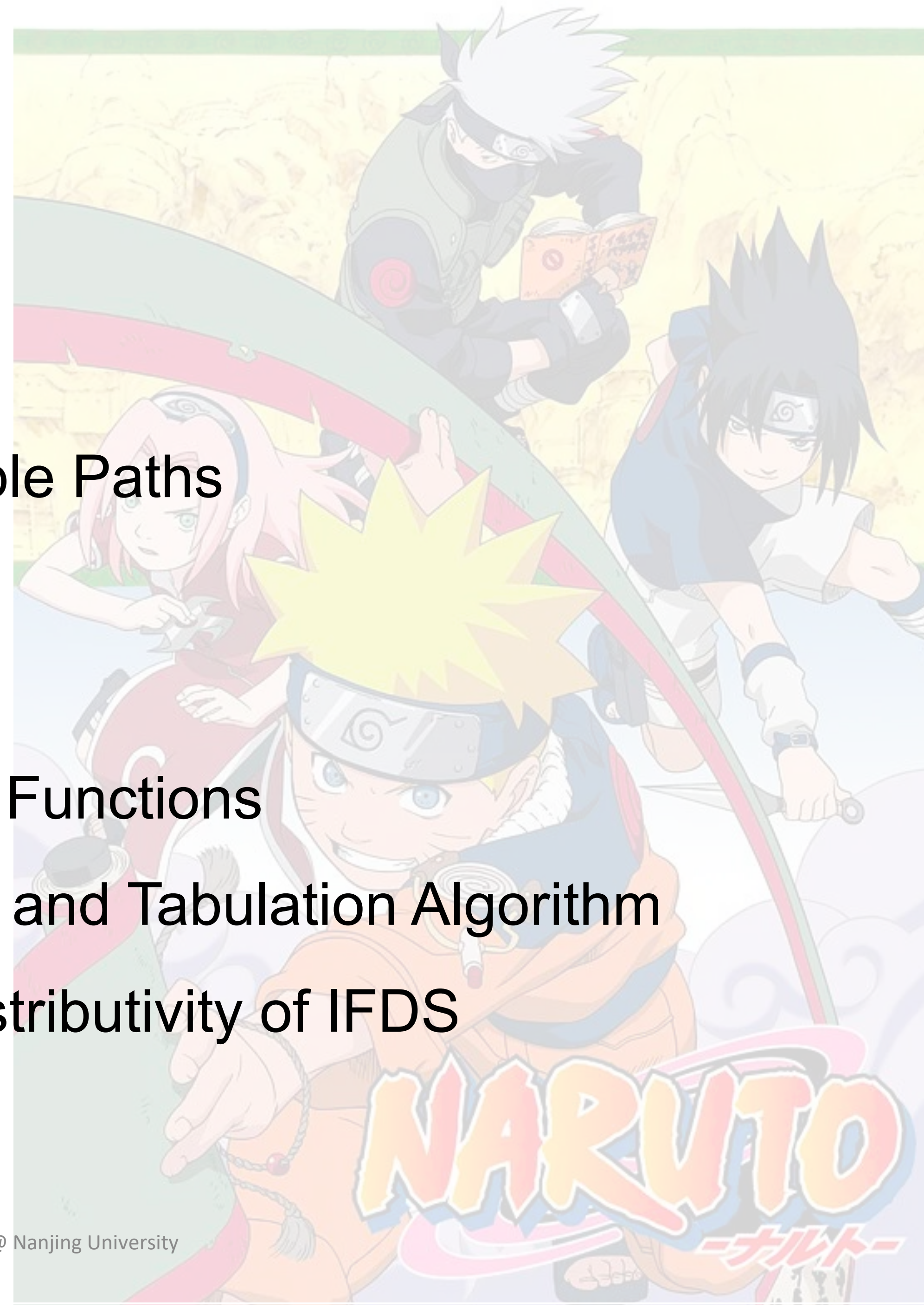
CFL-Reachability and IFDS

Nanjing University

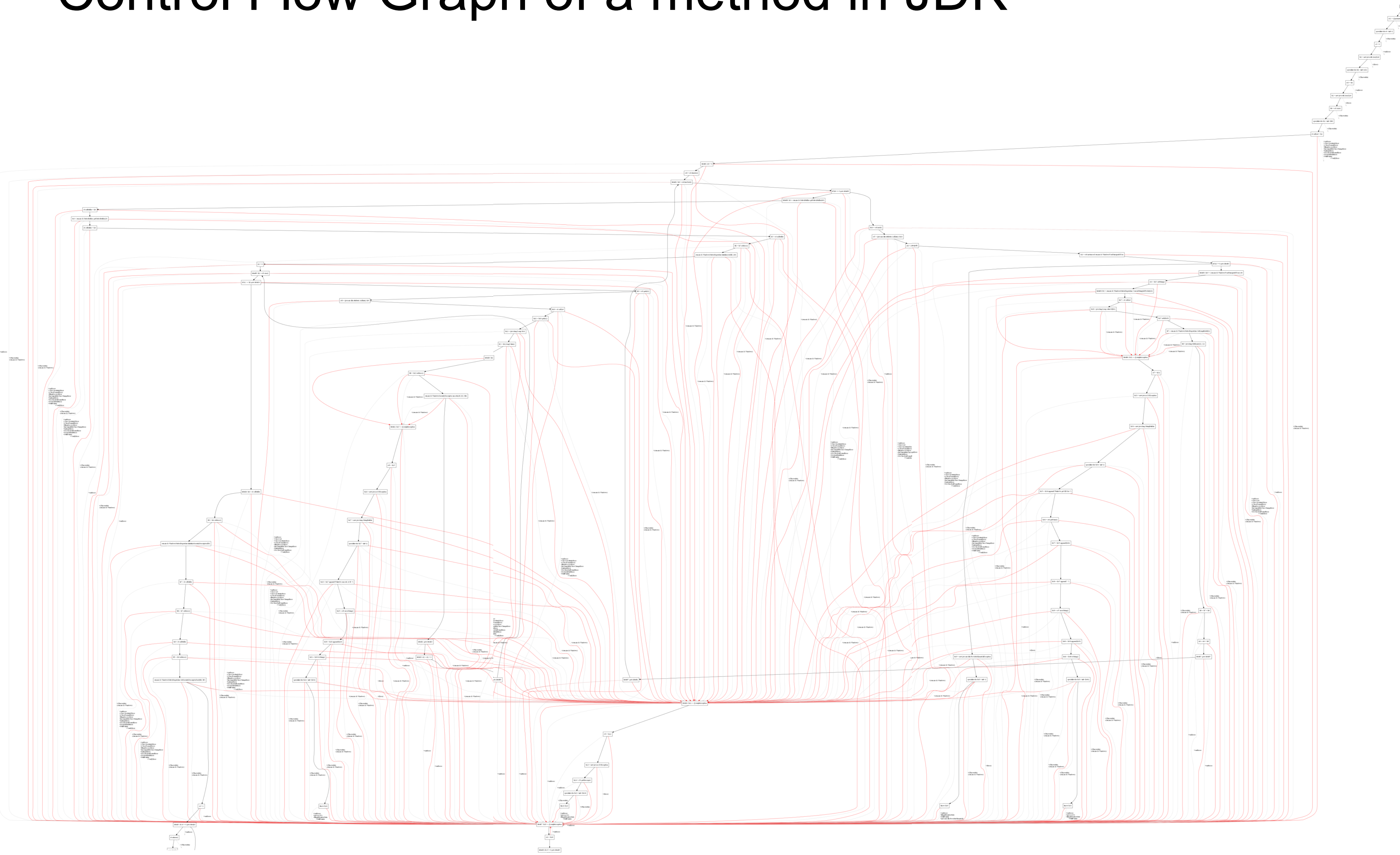
Yue Li

2020

Contents

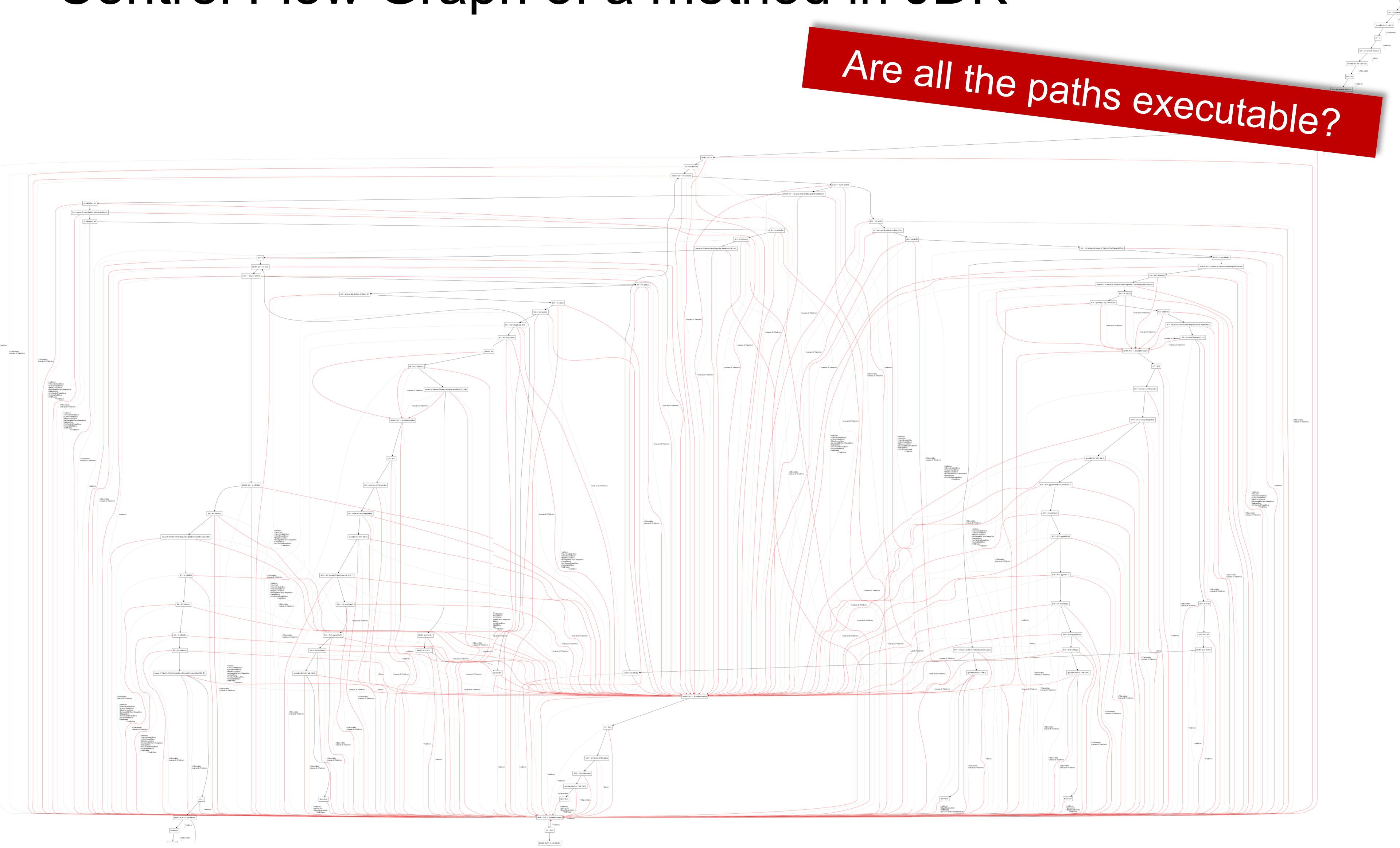


Control Flow Graph of a method in JDK



Control Flow Graph of a method in JDK

Are all the paths executable?

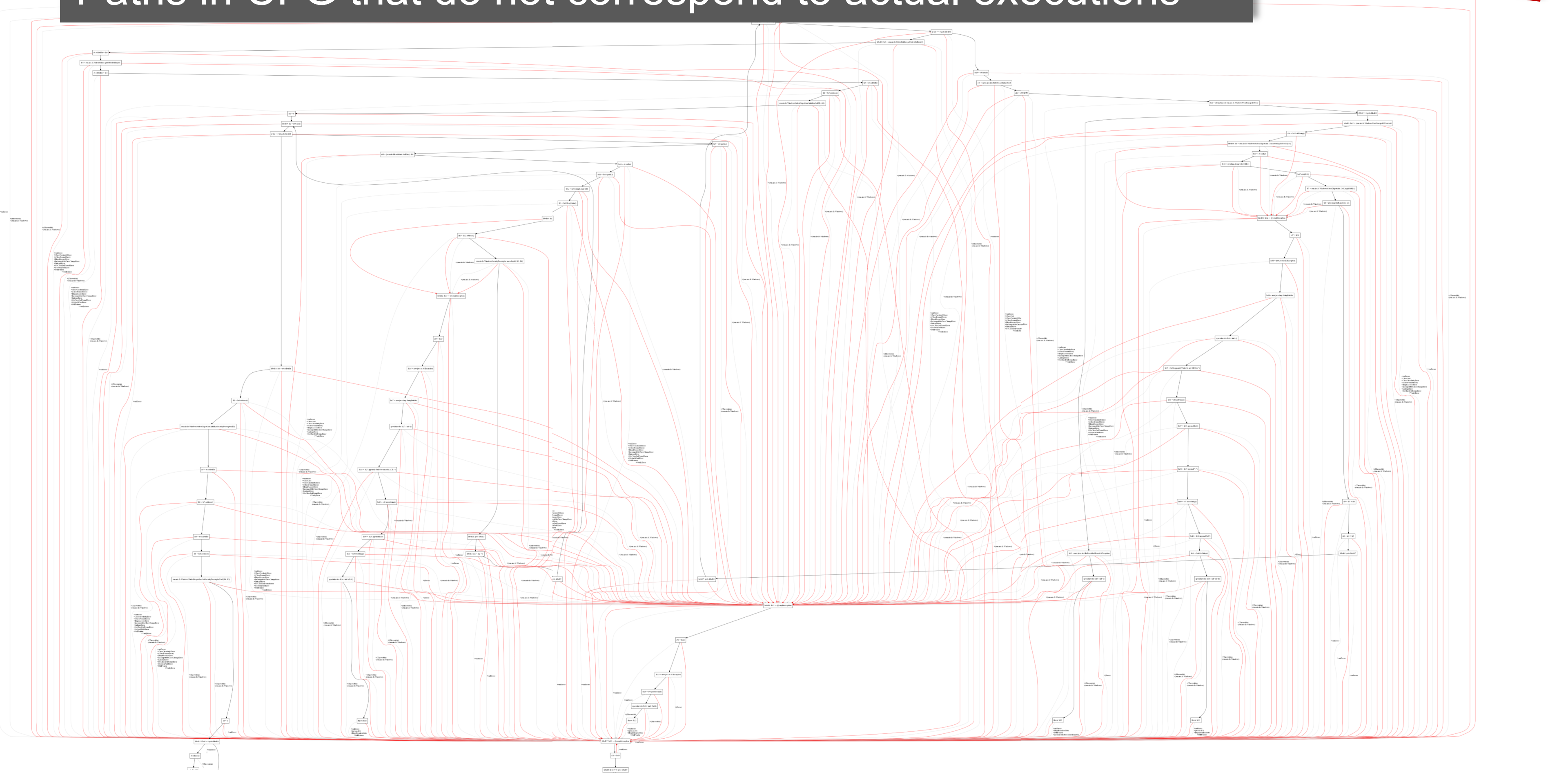


Control Flow Graph of a method in JDK

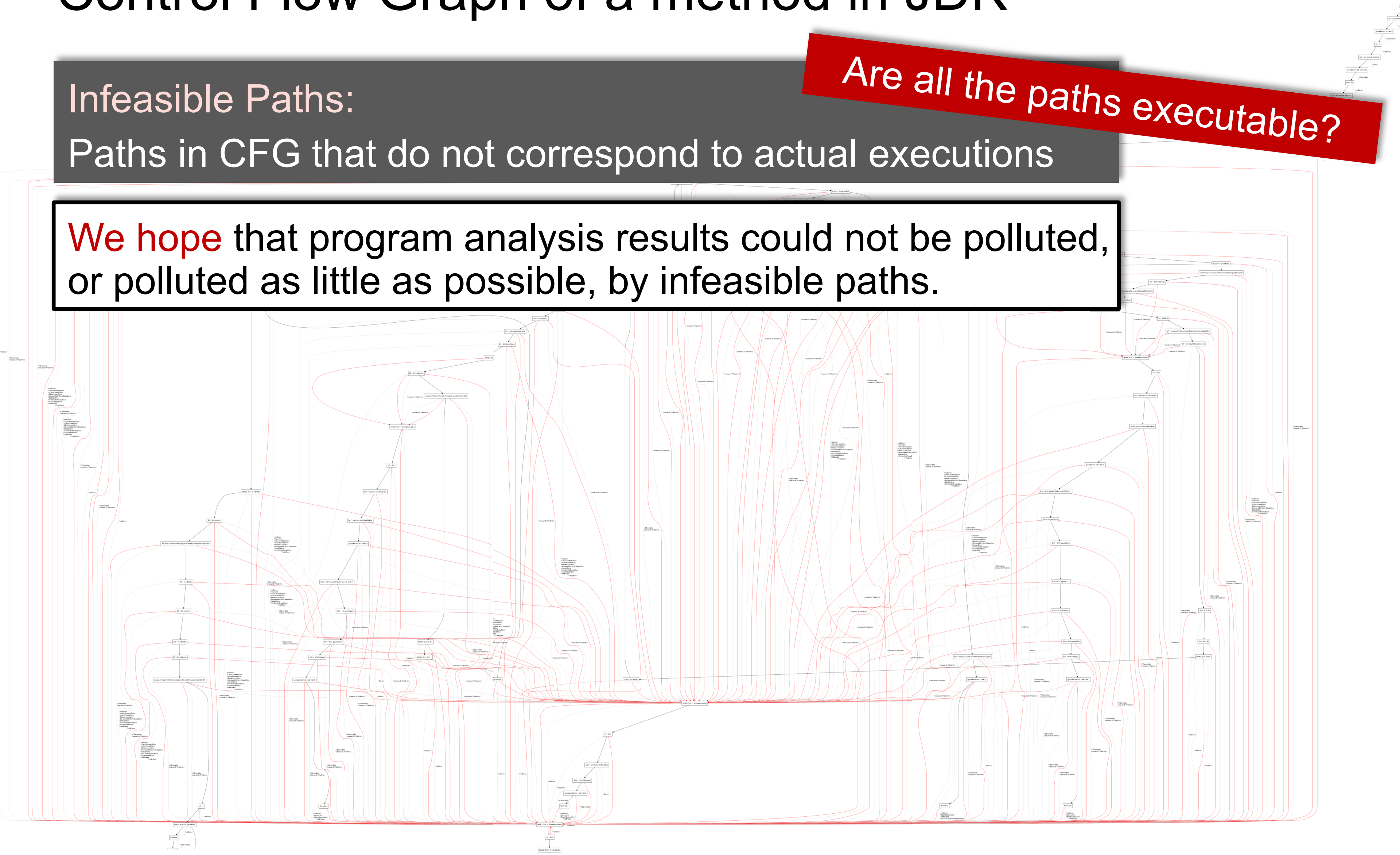
Infeasible Paths:

Paths in CFG that do not correspond to actual executions

Are all the paths executable?



Control Flow Graph of a method in JDK



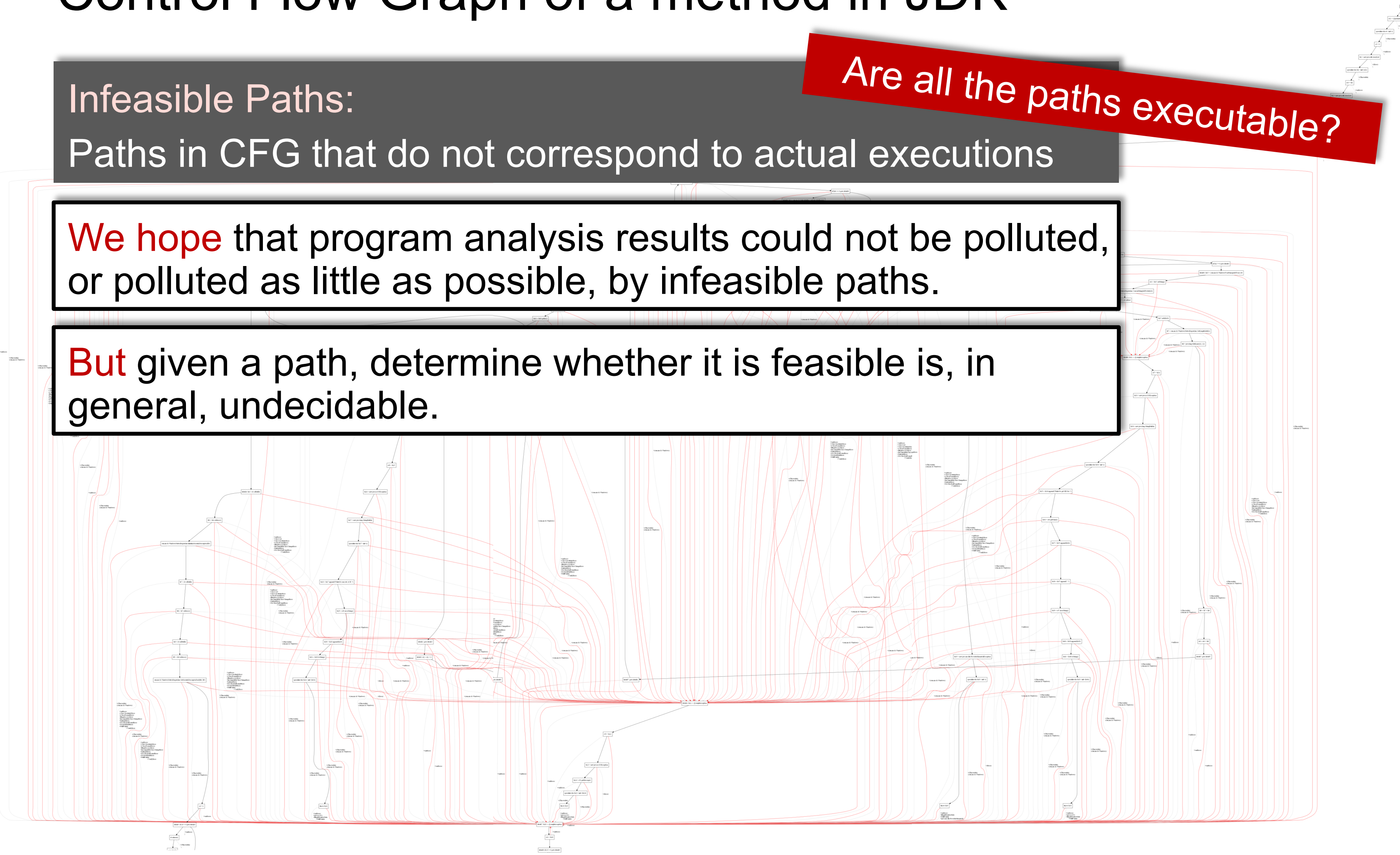
Are all the paths executable?

Infeasible Paths:

Paths in CFG that do not correspond to actual executions

We hope that program analysis results could not be polluted, or polluted as little as possible, by infeasible paths.

Control Flow Graph of a method in JDK



Are all the paths executable?

Infeasible Paths:

Paths in CFG that do not correspond to actual executions

We hope that program analysis results could not be polluted, or polluted as little as possible, by infeasible paths.

But given a path, determine whether it is feasible is, in general, undecidable.

Control Flow Graph of a method in JDK

Are all the paths executable?

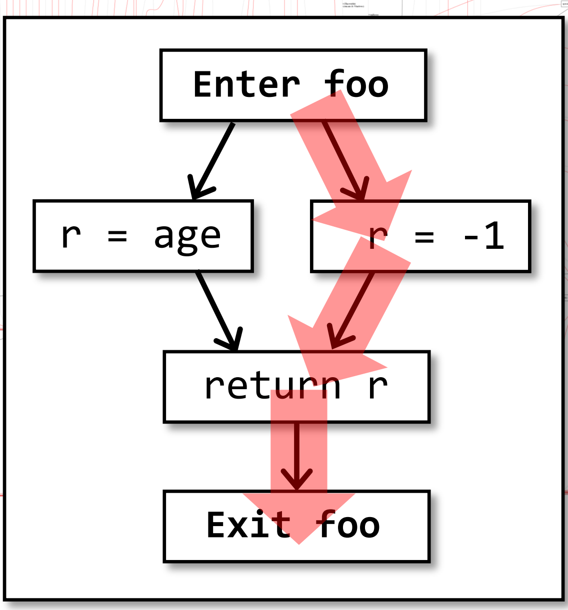
Infeasible Paths:

Paths in CFG that do not correspond to actual executions

We hope that program analysis results could not be polluted, or polluted as little as possible, by infeasible paths.

But given a path, determine whether it is feasible is, in general, undecidable.

```
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



Control Flow Graph of a method in JDK

Infeasible Paths:

Paths in CFG that do not correspond to actual executions

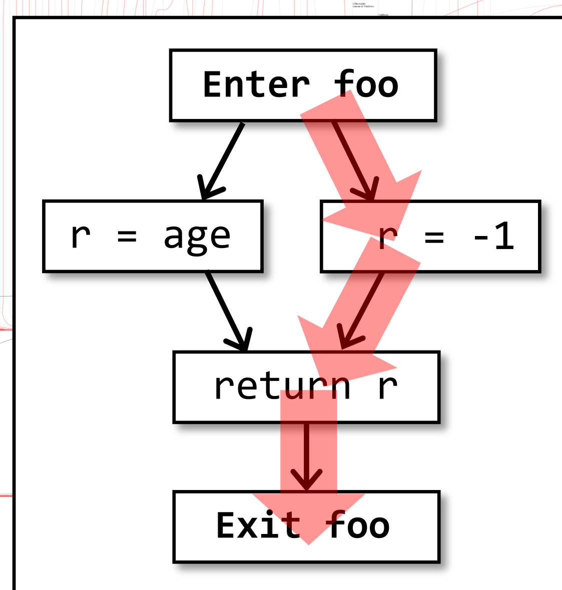
Are all the paths executable?

We hope that program analysis results could not be polluted, or polluted as little as possible, by infeasible paths.

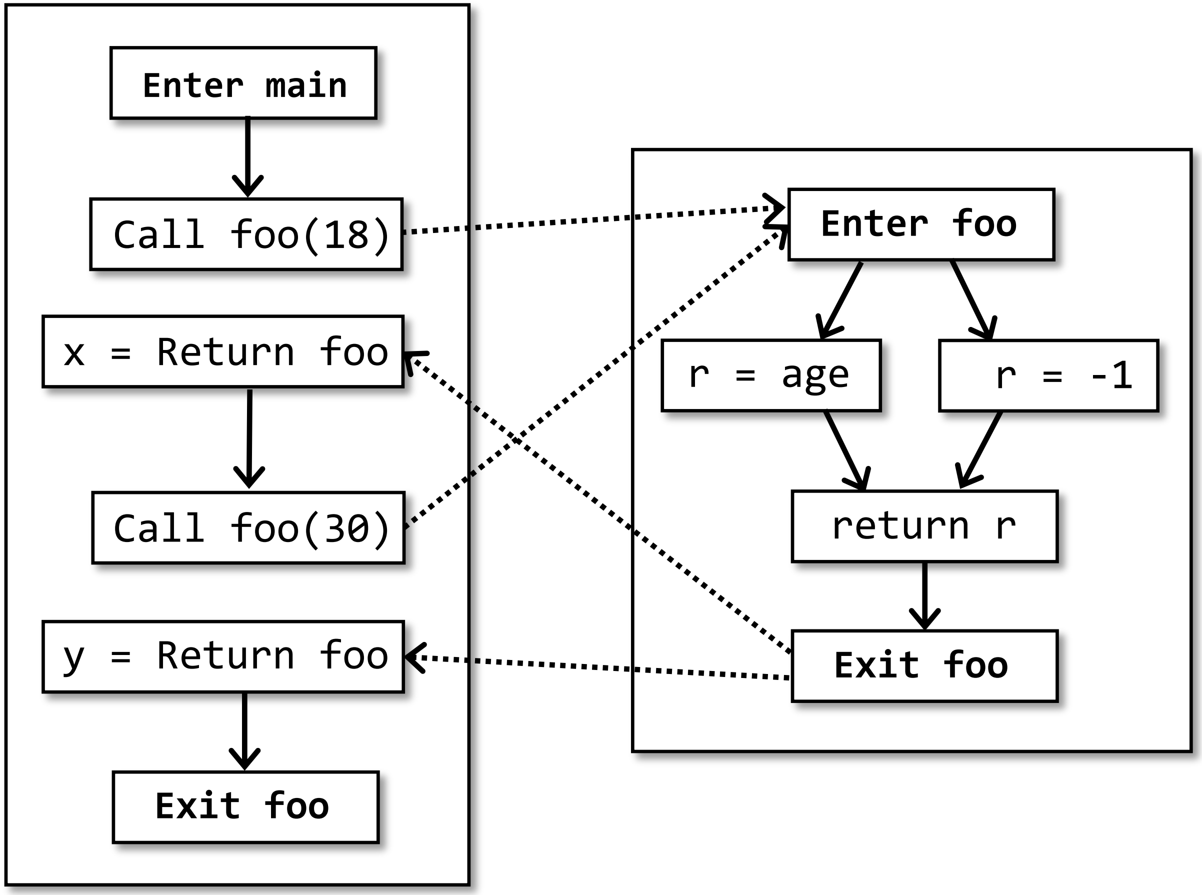
No Hope?

But given a path, determine whether it is feasible is, in general, undecidable.

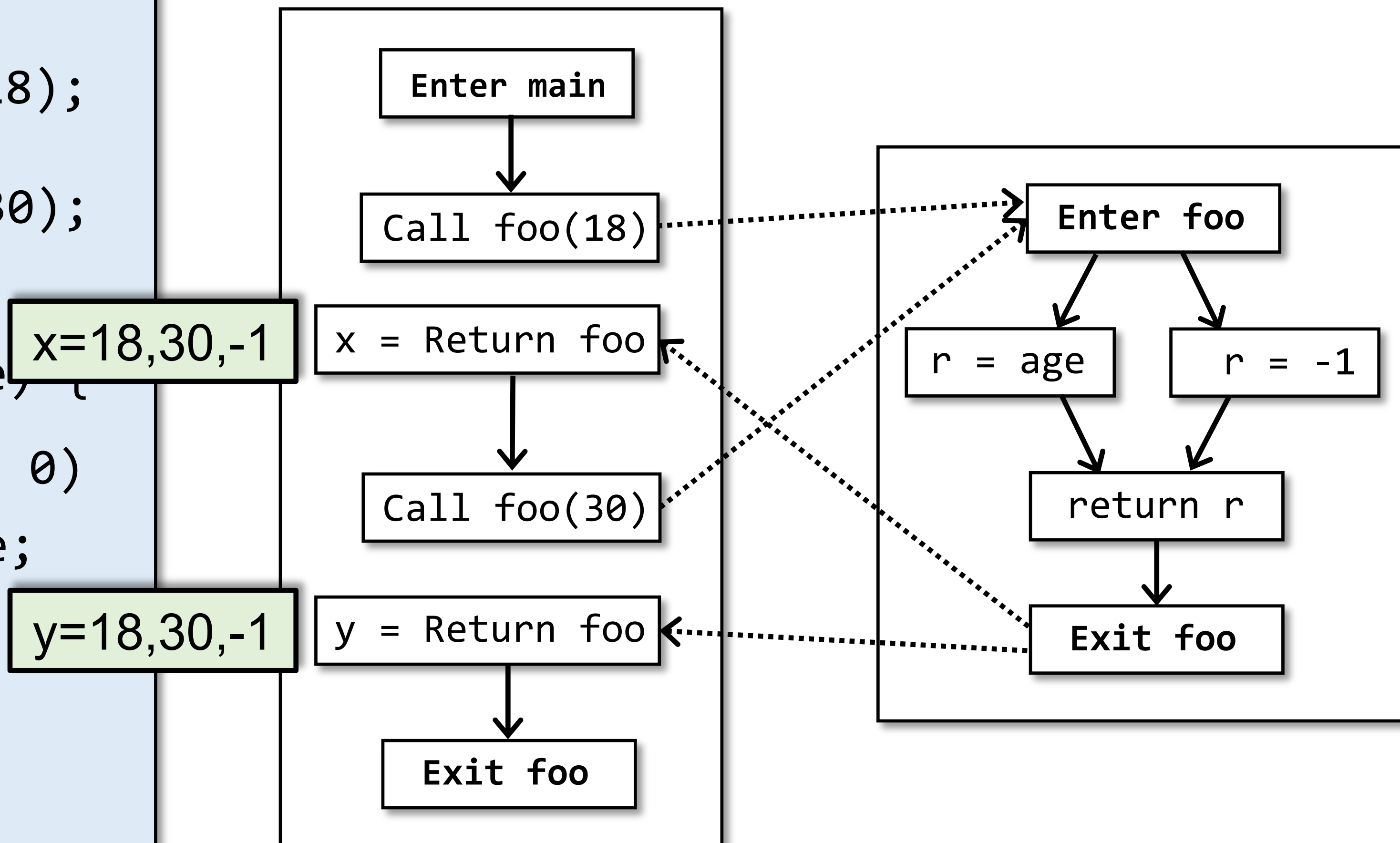
```
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



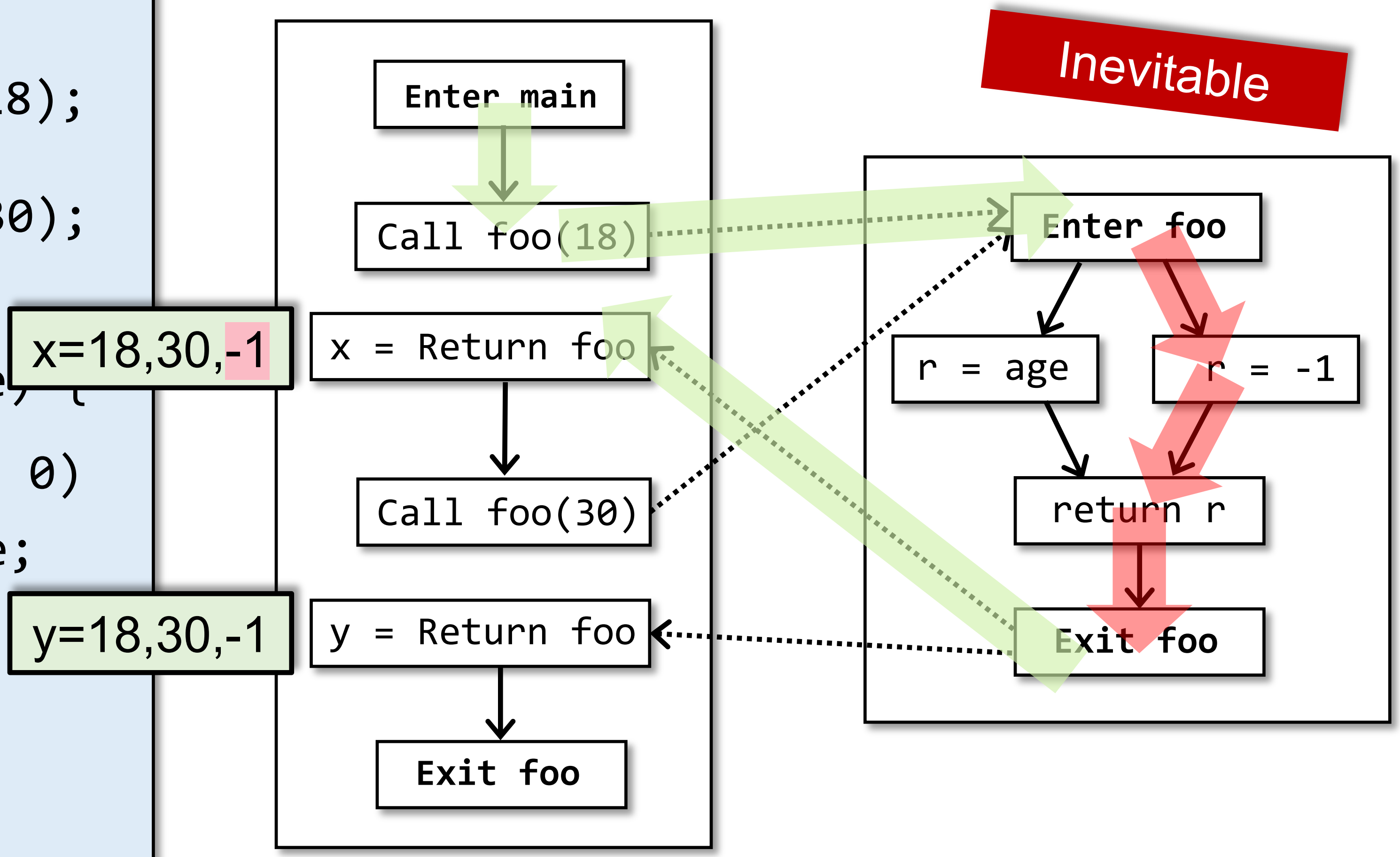
```
main() {
  x = foo(18);
    ⋮
  y = foo(30);
}
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



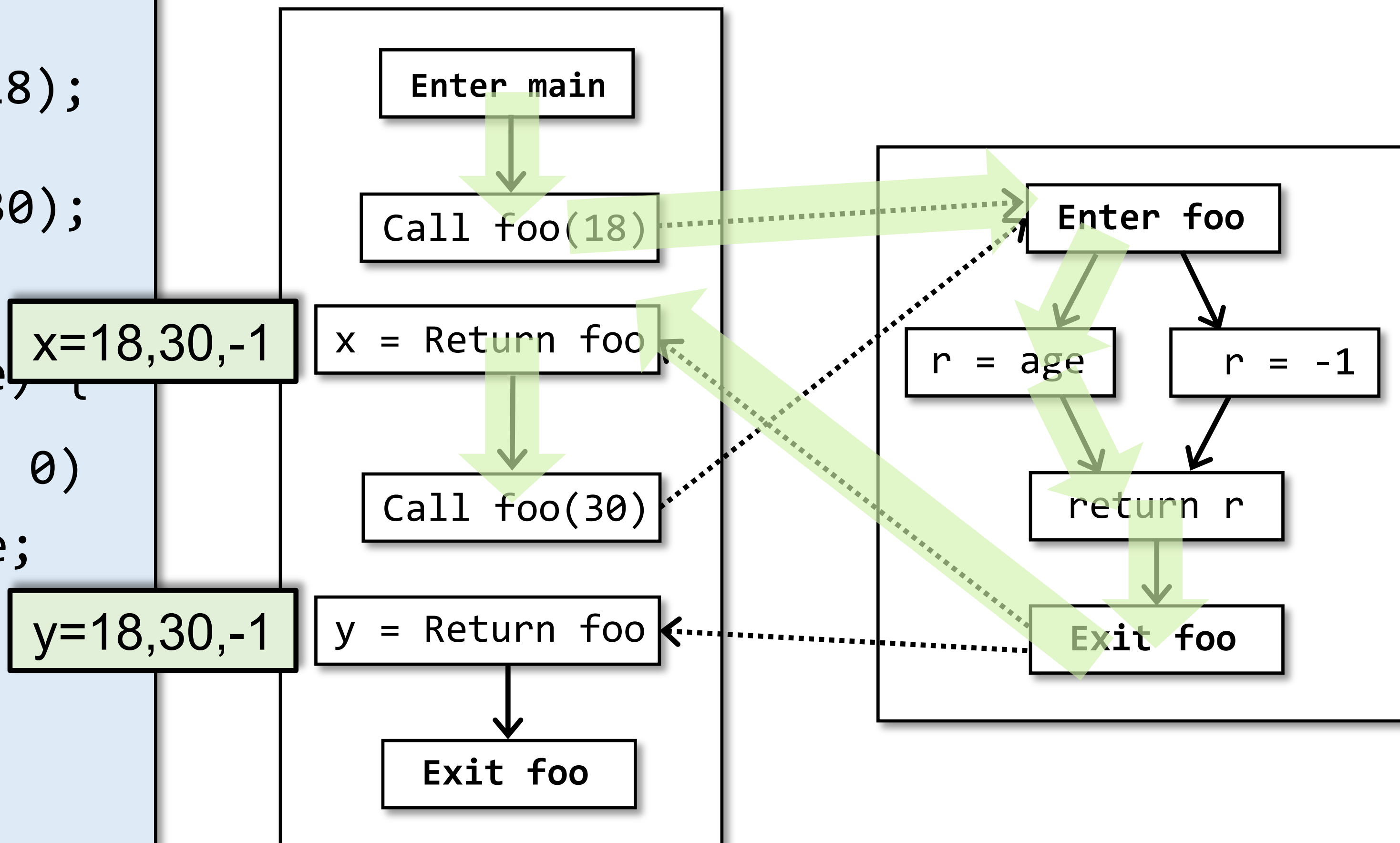
```
main() {
  x = foo(18);
    ⋮
  y = foo(30);
}
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



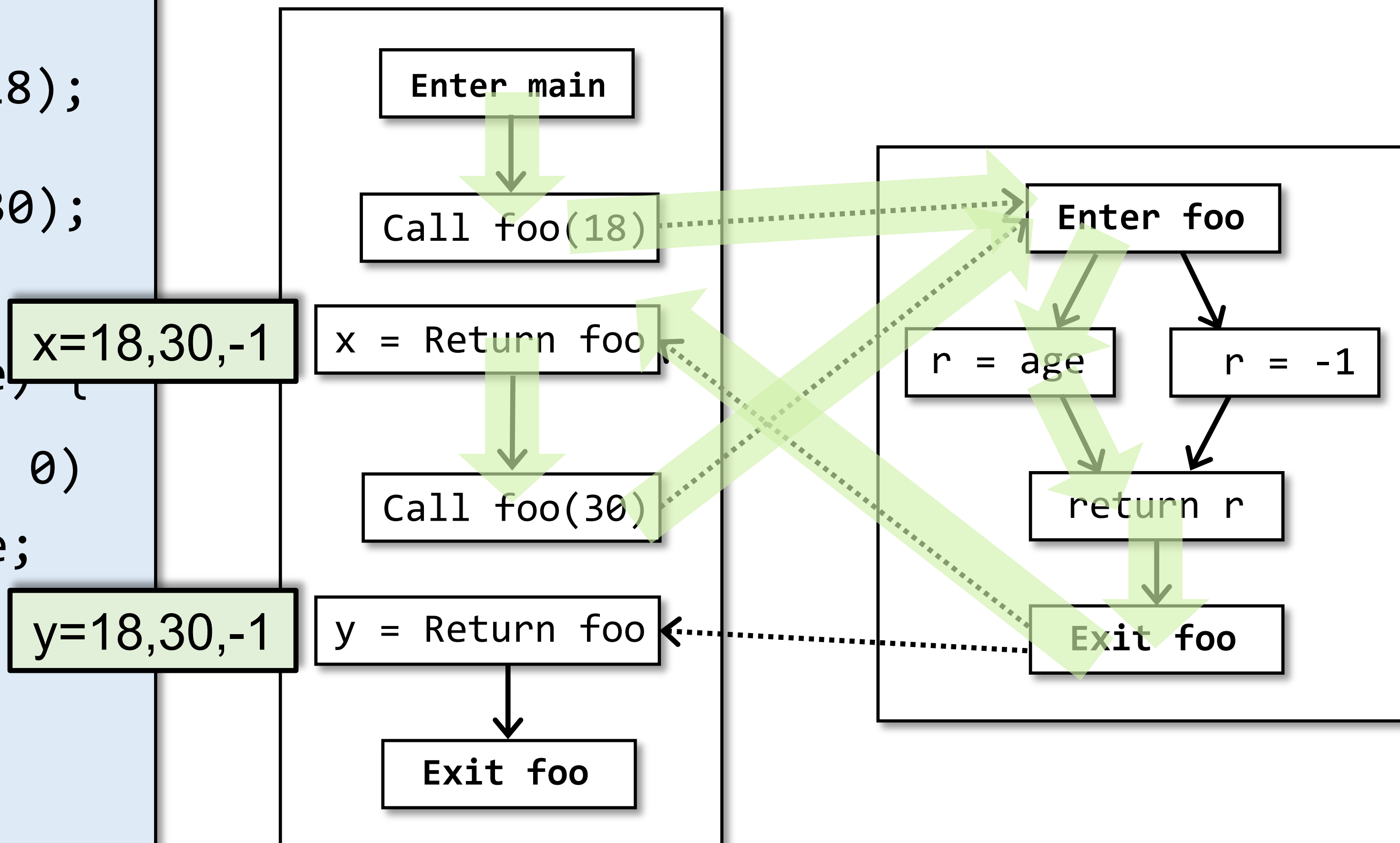
```
main() {
  x = foo(18);
     ⋮
  y = foo(30);
}

foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



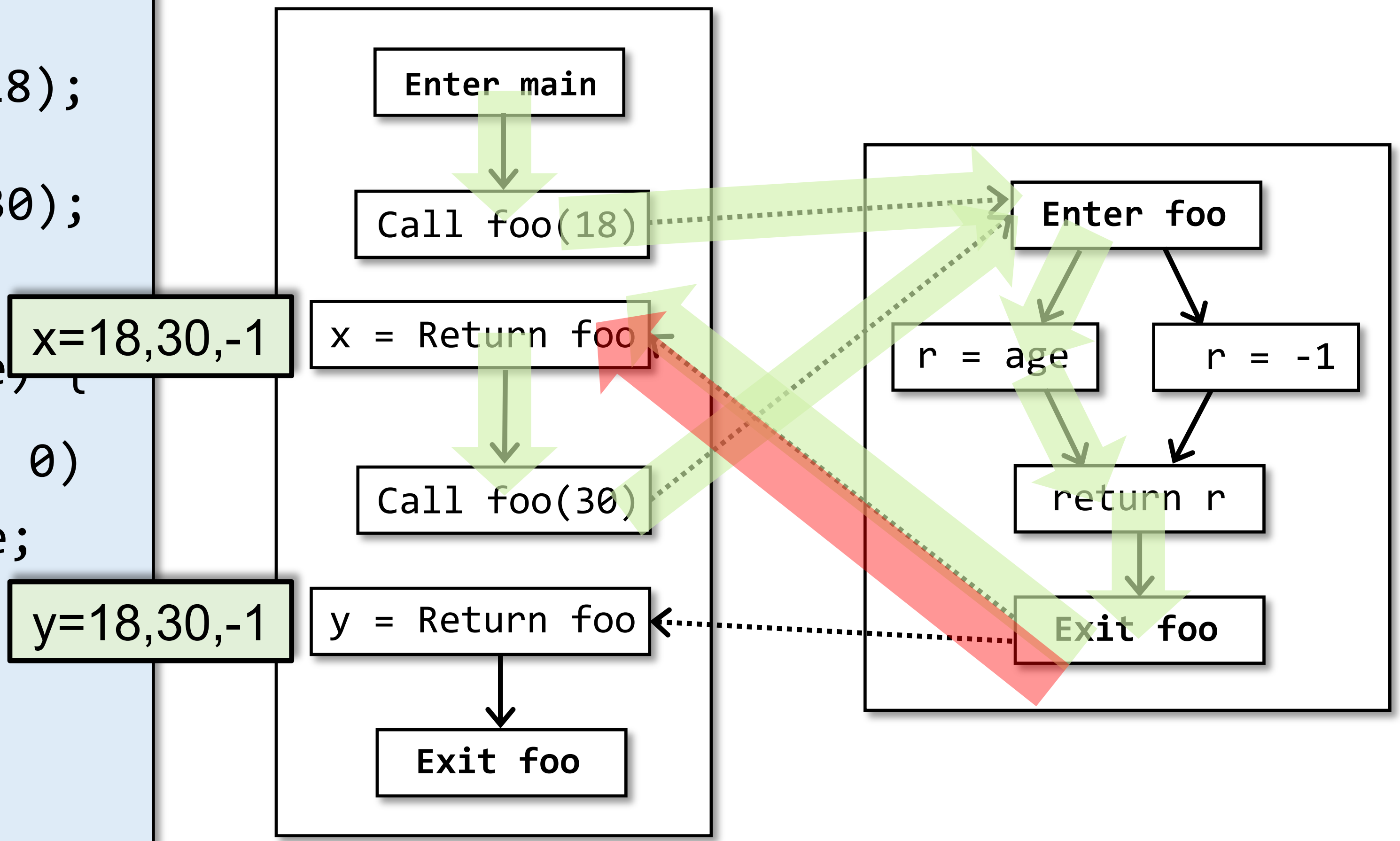
```
main() {
  x = foo(18);
    ⋮
  y = foo(30);
}

foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



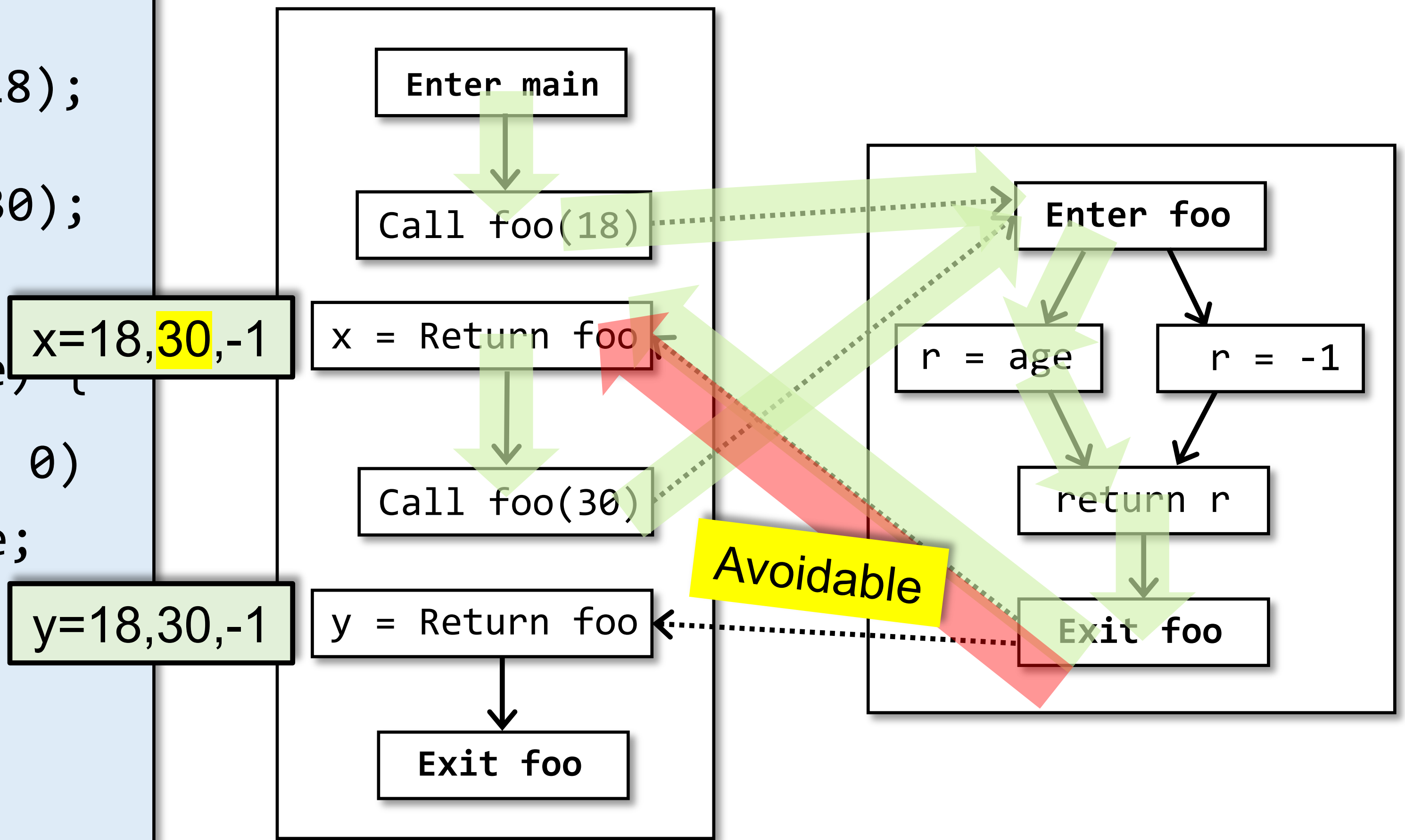
```
main() {
  x = foo(18);
     ⋮
  y = foo(30);
}
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



```
main() {
  x = foo(18);
    ⋮
  y = foo(30);
}

foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



```
main() {
  x = foo(18);
   ⋮
  y = foo(30);
}
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



Realizable Paths

Realizable Paths:
The paths in which "returns" are matched with corresponding "calls"

Realizable Paths

Realizable Paths:
The paths in which “returns” are matched with corresponding “calls”

- Realizable paths may not be executable, but unrealizable paths must not be executable.
- Our goal is to recognize realizable paths so that we could avoid polluting analysis results along unrealizable paths.

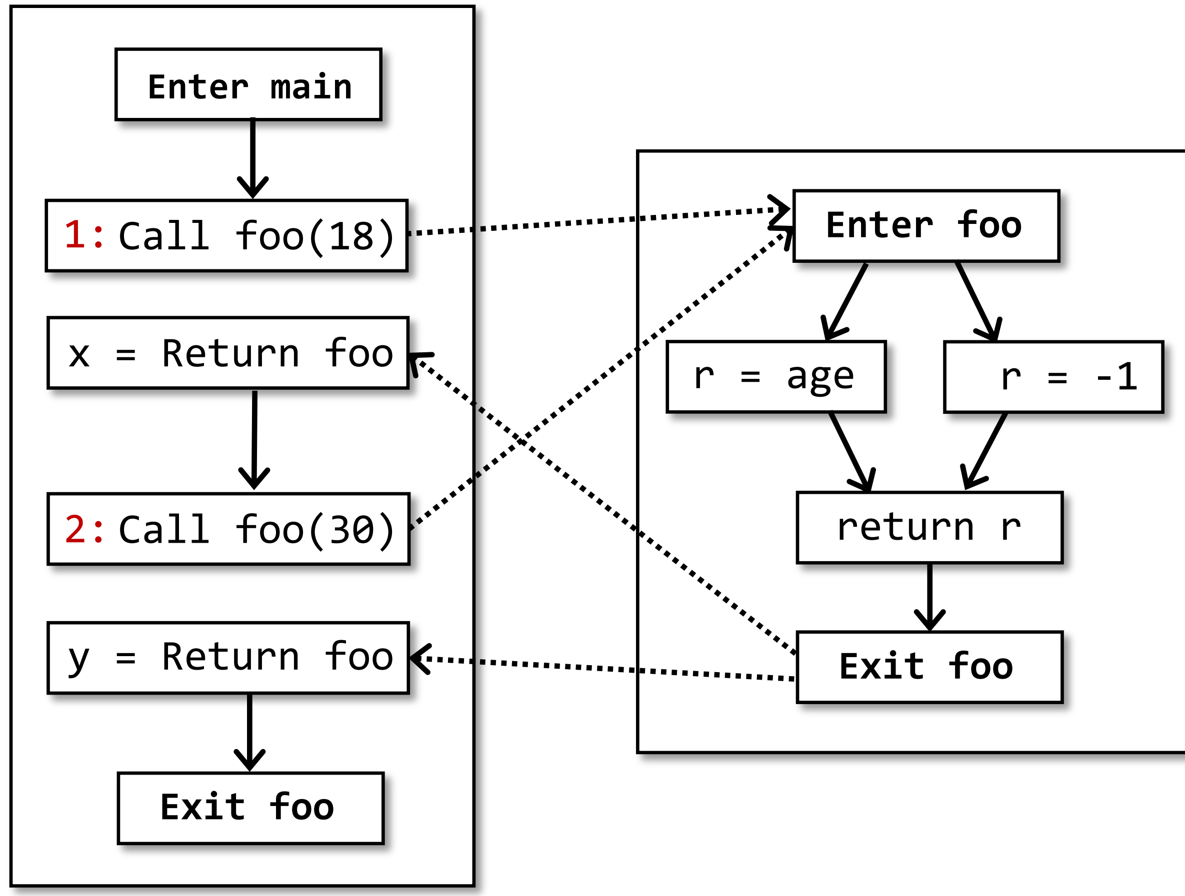
Realizable Paths

> **Realizable Paths:**
> The paths in which "returns" are matched with corresponding "calls"

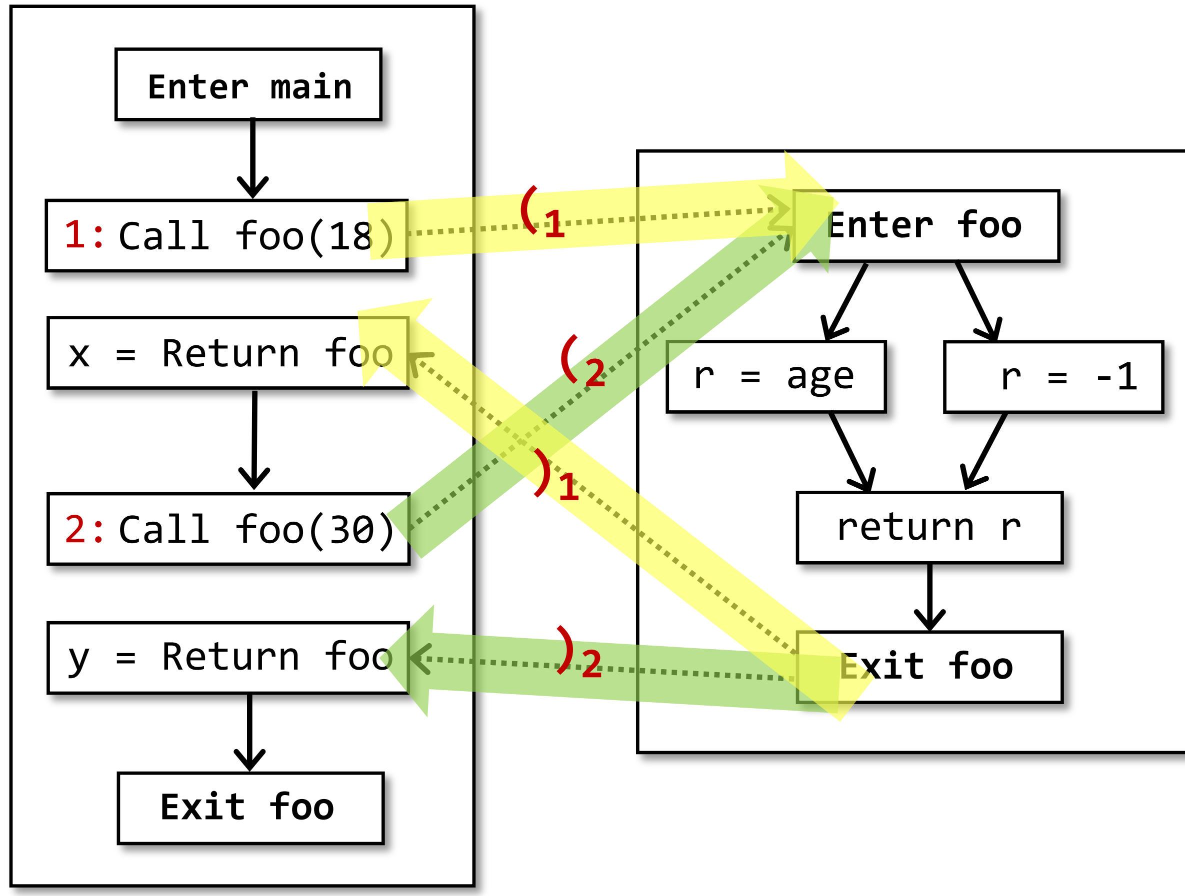
- Realizable paths may not be executable, but unrealizable paths must not be executable.
- O[illegible] is to recognize realizable paths so that we could avoid polluting analysis results along unrealizable paths.

How?

```
main() {
  x = foo(18);
   ⋮
  y = foo(30);
}
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```

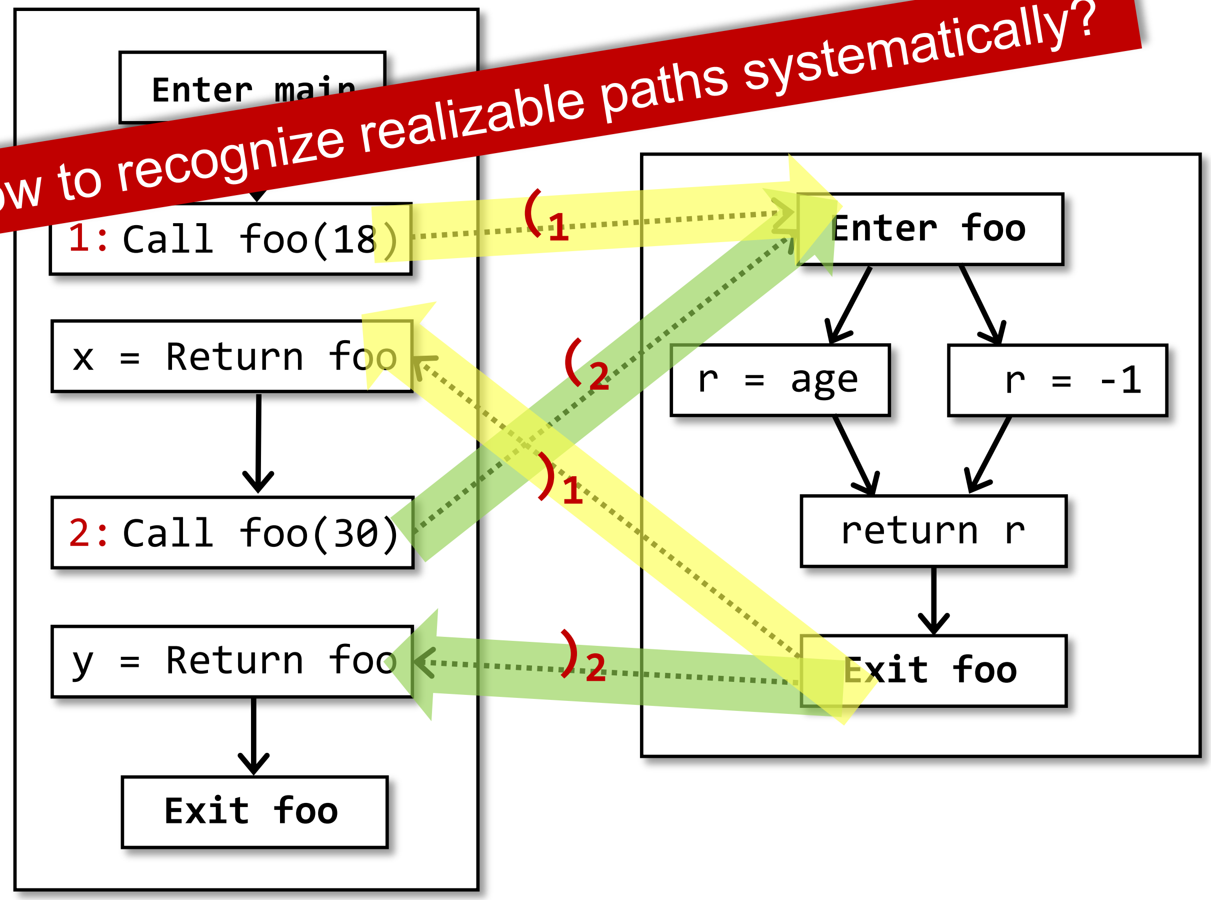


```
main() {
  x = foo(18);
    ⋮
  y = foo(30);
}
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```



```
main() {
  x = foo(18);
    ⋮
  y = foo(30);
}
foo(int age) {
  if(age >= 0)
    r = age;
  else
    r = -1;
  return r;
}
```

How to recognize realizable paths systematically?



CFL-Reachability

CFL-Reachability

A path is considered to connect two nodes A and B, or B is reachable from A, only if the concatenation of the labels on the edges of the path is a word in a specified context-free language.

CFL-Reachability

CFL-Reachability

A path is considered to connect two nodes A and B, or B is reachable from A, only if the concatenation of the labels on the edges of the path is a word in a specified context-free language.

- A valid sentence in language L must follow L's grammar.
- A context-free language is a language generated by a context-free grammar (CFG).

CFL-Reachability

CFL-Reachability

A path is considered to connect two nodes A and B, or B is reachable from A, only if the concatenation of the labels on the edges of the path is a word in a specified context-free language.

- A valid sentence in language L must follow L's grammar.
- A context-free language is a language generated by a context-free grammar (CFG).

CFG is a formal grammar in which every production is of the form:

$$S \rightarrow \alpha$$

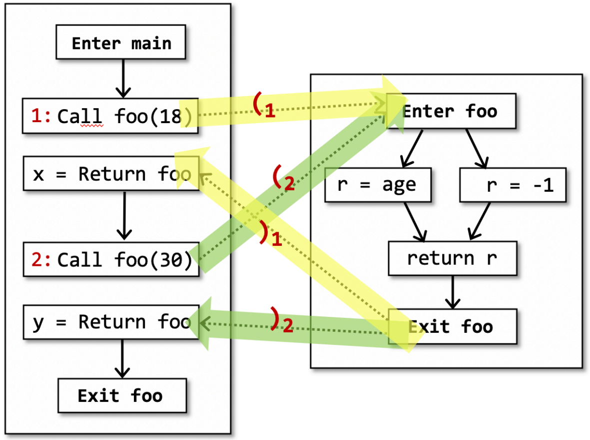
where S is a single nonterminal and α could be a string of terminals and/or nonterminals, or empty.

- $S \rightarrow aSb$
- $S \rightarrow \varepsilon$

Context-free means S could be replaced by aSb/ε anywhere, regardless of where S occurs.

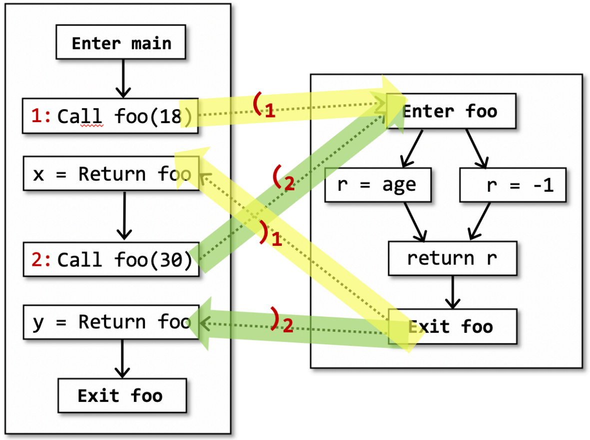
CFL-Reachability

Partially Balanced-Parenthesis Problem via CFL



- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

CFL-Reachability

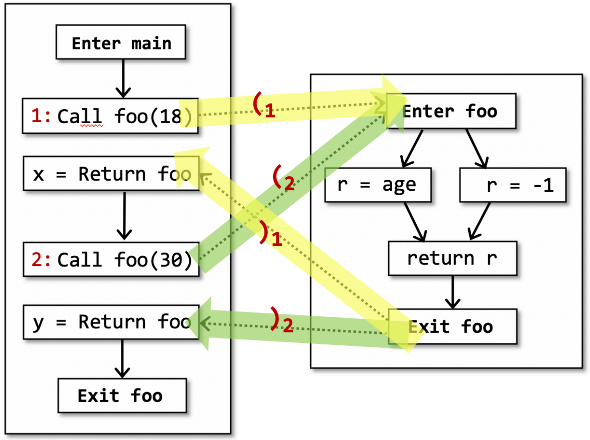


Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

CFL-Reachability



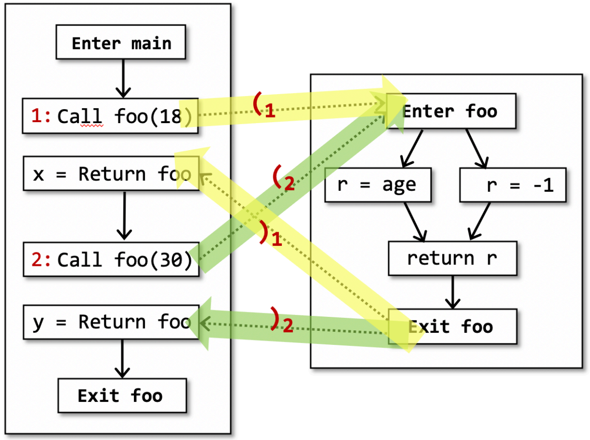
Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site `i`, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

e.g., $(_1\ (_2\ e\ )_2\ )_1\ (_3$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

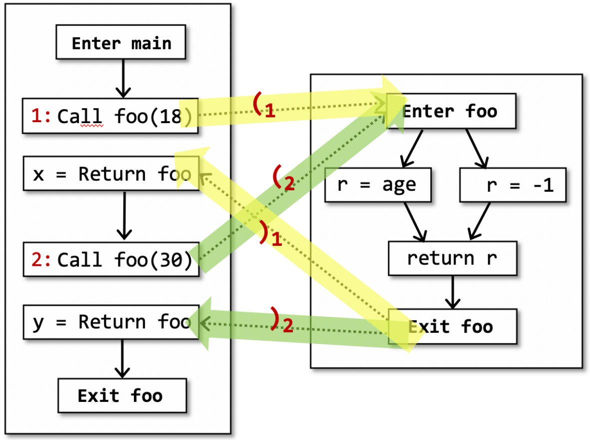
- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

$$realizable \rightarrow matched\ \ realizable$$

e.g., $(_1\, (_2\ e\ )_2\,)_1\, (_3$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

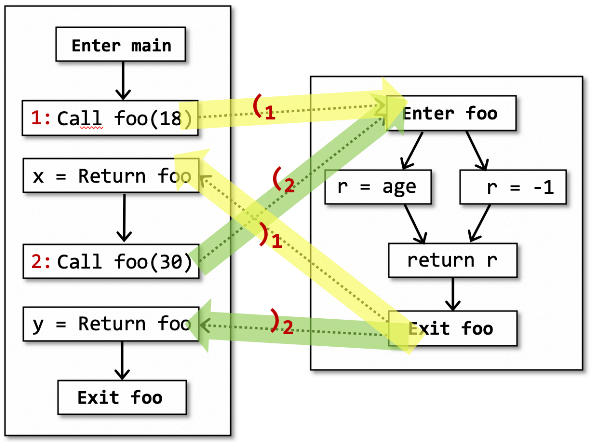
- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

$$realizable \rightarrow matched\ \ realizable$$
$$\rightarrow (_i$$

e.g., $(_1 (_2\ e\ )_2\ )_1\ (_3$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

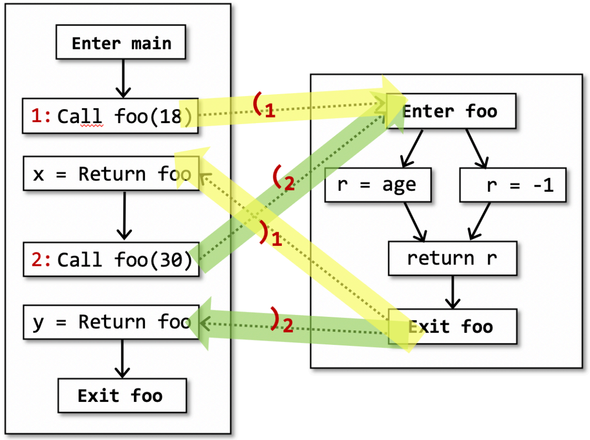
realizable → *matched realizable*

→ $(_i$

e.g., $(_1\, (_2\; e\;)_2\,)_1\, (_3$

e.g., $(_1\, (_2\; e\;)_2\,)_1\, (_3\, (_4$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

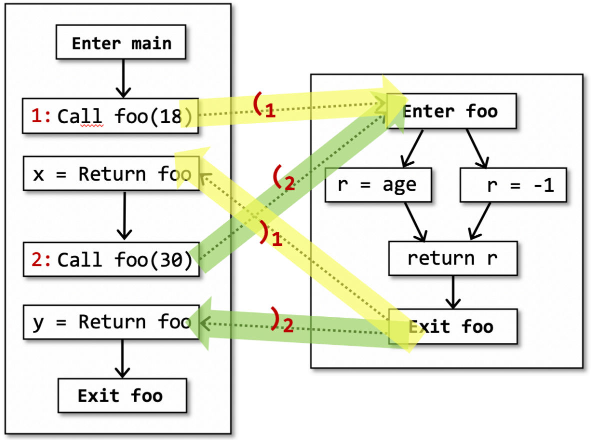
realizable → *matched realizable*

→ $(_i$ *realizable*

e.g., $(_1\ (_2\ e\ )_2\ )_1\ (_3$

e.g., $(_1\ (_2\ e\ )_2\ )_1\ (_3\ (_4$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

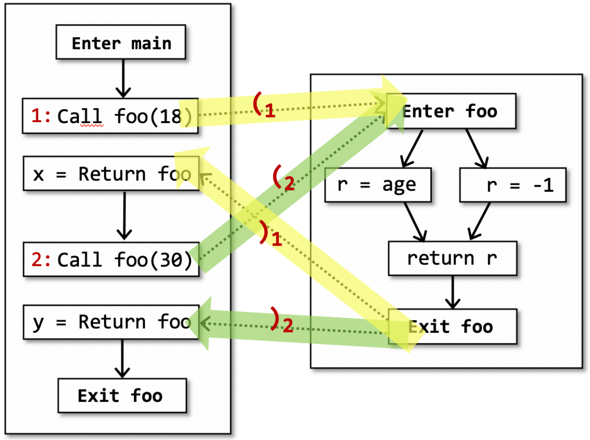
A path is a realizable path iff the path' word is in the language *L(realizable)*

$$
\begin{aligned}
\textit{realizable} &\rightarrow \textit{matched}\ \ \textit{realizable} \\
&\rightarrow (_i\ \ \textit{realizable} \\
&\rightarrow \varepsilon
\end{aligned}
$$

e.g., $(_1 (_2\ e\ )_2)_1 (_3$

e.g., $(_1 (_2\ e\ )_2)_1 (_3 (_4$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language $L(realizable)$

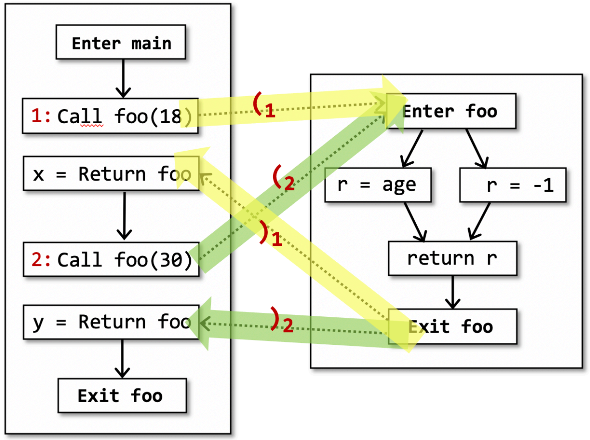
$$
\begin{aligned}
realizable &\rightarrow matched\ \ realizable \\
&\rightarrow (_i\ \ realizable \\
&\rightarrow \varepsilon \\
matched &\rightarrow (_i\ matched\ )_i
\end{aligned}
$$

e.g., $(_1 (_2\ e\ )_2)_1 (_3$

e.g., $(_1 (_2\ e\ )_2)_1 (_3 (_4$

CFL-Reachability

Partially Balanced-Parenthesis Problem via CFL



- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

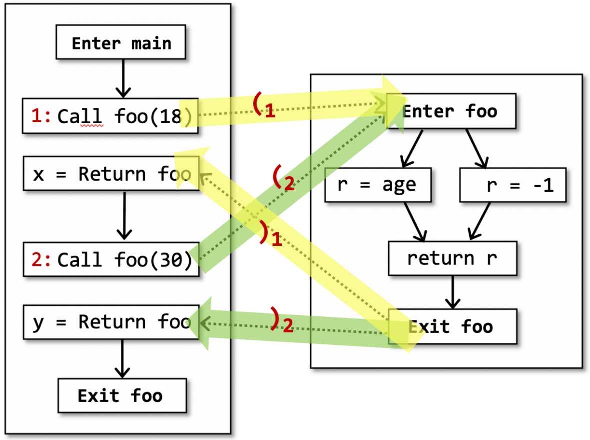
A path is a realizable path iff the path' word is in the language *L(realizable)*

$$
\begin{aligned}
\textit{realizable} &\rightarrow \textit{matched}\ \textit{realizable} \\
&\rightarrow (_i\ \textit{realizable} \\
&\rightarrow \varepsilon \\
\textit{matched} &\rightarrow (_i\ \textit{matched}\ )_i \\
&\rightarrow \mathrm{e} \\
&\rightarrow \varepsilon
\end{aligned}
$$

e.g., $(_1\ (_2\ e\ )_2\ )_1\ (_3$

e.g., $(_1\ (_2\ e\ )_2\ )_1\ (_3\ (_4$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

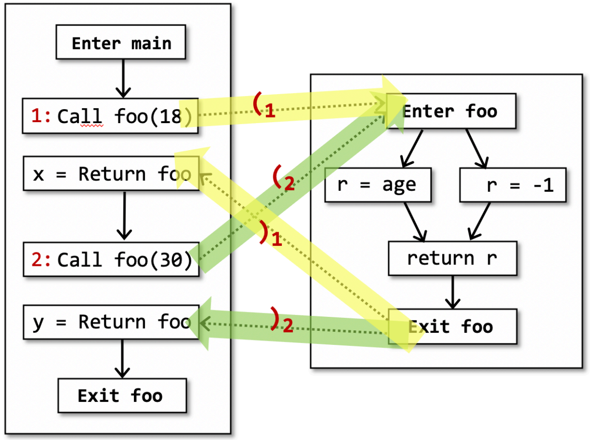
$$
\begin{aligned}
\mathit{realizable} &\rightarrow \mathit{matched}\ \ \mathit{realizable} \\
&\rightarrow (_i\ \ \mathit{realizable} \\
&\rightarrow \varepsilon \\
\mathit{matched} &\rightarrow (_i\ \mathit{matched}\ )_i \\
&\rightarrow \mathrm{e} \\
&\rightarrow \varepsilon
\end{aligned}
$$

e.g., $(_1\ (_2\ e\ )_2\ )_1\ (_3$

e.g., $(_1\ (_2\ e\ )_2\ )_1\ (_3\ (_4$

e.g., $(_1\ (_2\ e\ e\ e\ )_2\ )_1\ (_3\ (_4$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

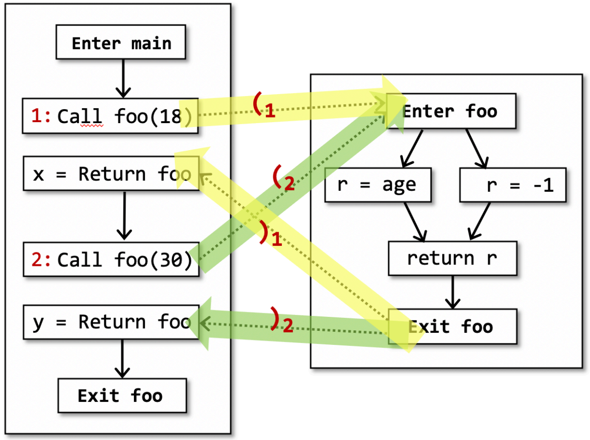
$$
\begin{aligned}
\textit{realizable} &\to \textit{matched}\ \ \textit{realizable}\\
&\to (_i\ \ \textit{realizable}\\
&\to \varepsilon\\
\textit{matched} &\to (_i\ \ \textit{matched}\ \ )_i\\
&\to \mathrm{e}\\
&\to \varepsilon\\
&\to \textit{matched}\ \textit{matched}
\end{aligned}
$$

e.g., $(_1\, (_2\, e\,)_2\,)_1\, (_3$

e.g., $(_1\, (_2\, e\,)_2\,)_1\, (_3\, (_4$

e.g., $(_1\, (_2\, e\, e\, e\,)_2\,)_1\, (_3\, (_4$

CFL-Reachability



Partially Balanced-Parenthesis Problem via CFL

- Every right parenthesis "$)_i$" is balanced by a preceding left parenthesis "$(_i$", but the converse needs not hold
- For each call site i, label its call edge "$(_i$" and return edge "$)_i$"
- Label all other edges with symbol "e"

A path is a realizable path iff the path' word is in the language *L(realizable)*

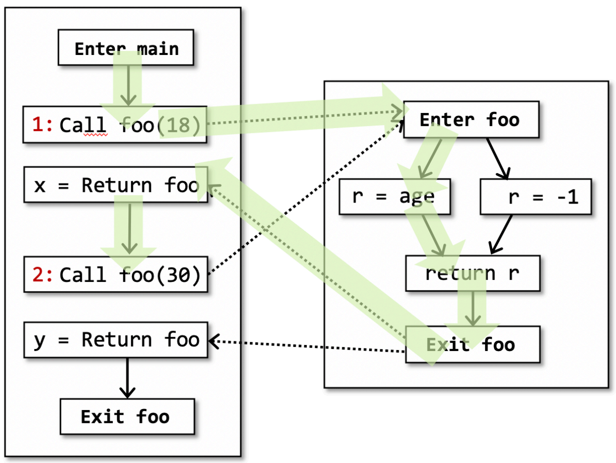
$$
\begin{aligned}
\textit{realizable} &\rightarrow \textit{matched}\ \ \textit{realizable} \\
&\rightarrow (_i\ \ \textit{realizable} \\
&\rightarrow \varepsilon \\
\textit{matched} &\rightarrow (_i\ \textit{matched}\ )_i \\
&\rightarrow e \\
&\rightarrow \varepsilon \\
&\rightarrow \textit{matched}\ \textit{matched}
\end{aligned}
$$

e.g., $(_1\,(_2\ e\ )_2\,)_1\,(_3$

e.g., $(_1\,(_2\ e\ )_2\,)_1\,(_3\,(_4$

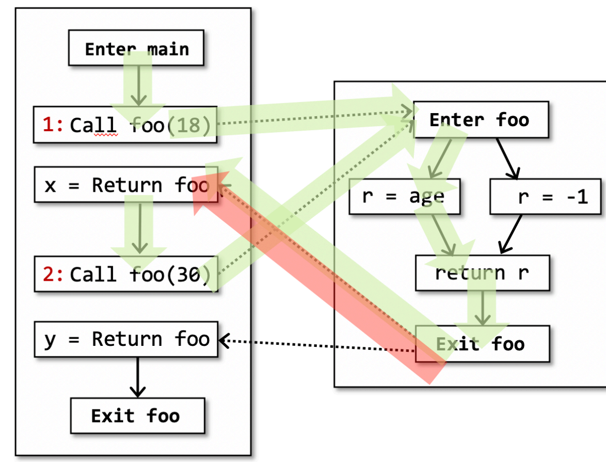
e.g., $(_1\,(_2\ e\ e\ e\ )_2\,)_1\,(_3\,(_4$

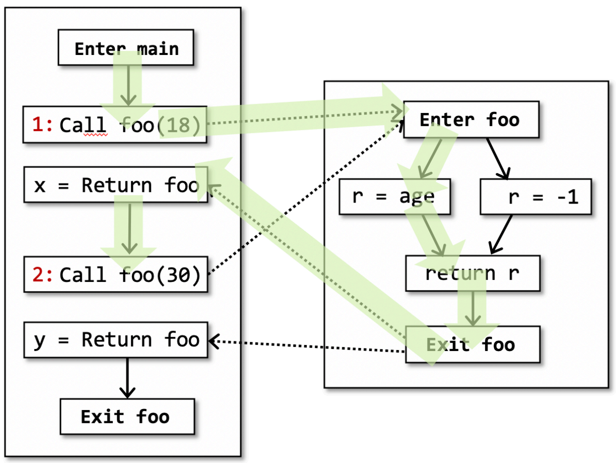
e.g., $e\ e\ (_1\,(_2\ e\ e\ e\ )_2\,)_1\,(_3\,(_4\ e$



L(realizable):

$$\begin{aligned}
realizable &\rightarrow matched\ \ realizable \\
&\rightarrow (_i\ \ realizable \\
&\rightarrow \varepsilon \\
matched &\rightarrow (_i\ \ matched\ \ )_i \\
&\rightarrow e \\
&\rightarrow \varepsilon \\
&\rightarrow matched\ matched
\end{aligned}$$





L(realizable):

realizable → *matched realizable*

→ $(_i$ *realizable*

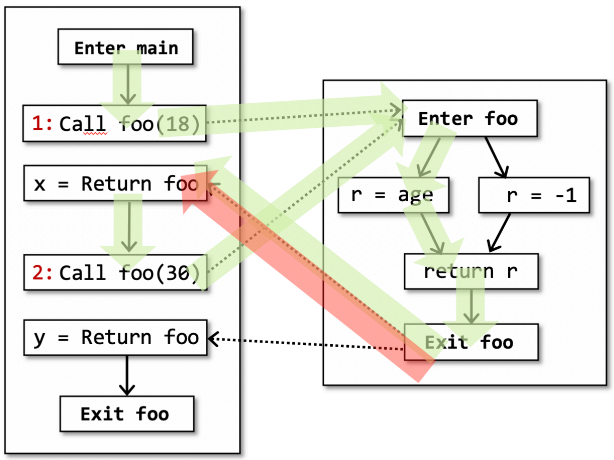
→ ε

matched → $(_i$ *matched* $)_i$

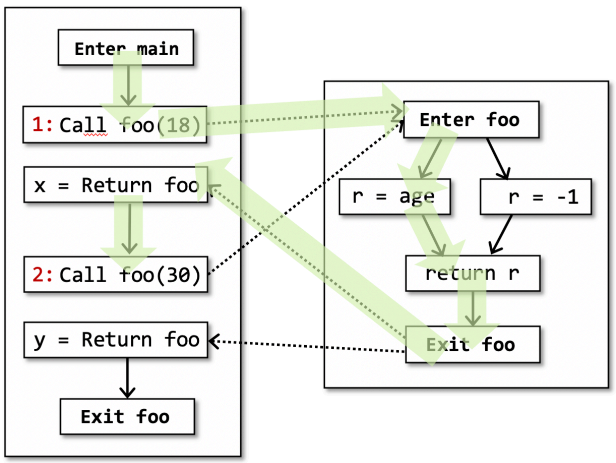
→ e

→ ε

→ *matched matched*



$e(_1eee)_1e \in L(realizable)$



$L(\mathit{realizable})$:

$\mathit{realizable} \rightarrow \mathit{matched}\ \ \mathit{realizable}$

$\rightarrow (_i\ \ \mathit{realizable}$

$\rightarrow \varepsilon$

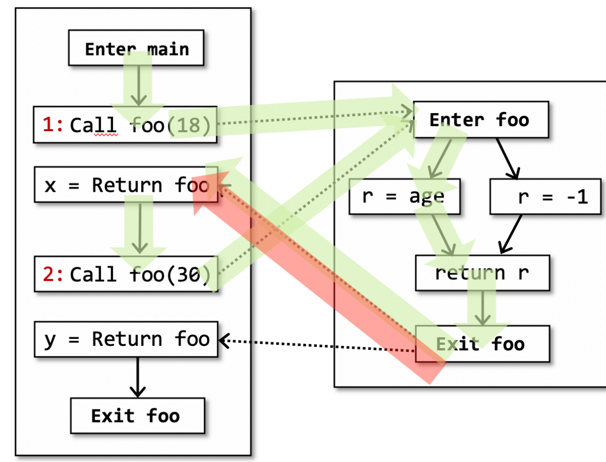
$\mathit{matched} \rightarrow (_i\ \mathit{matched}\ )_i$

$\rightarrow e$

$\rightarrow \varepsilon$

$\rightarrow \mathit{matched}\ \mathit{matched}$

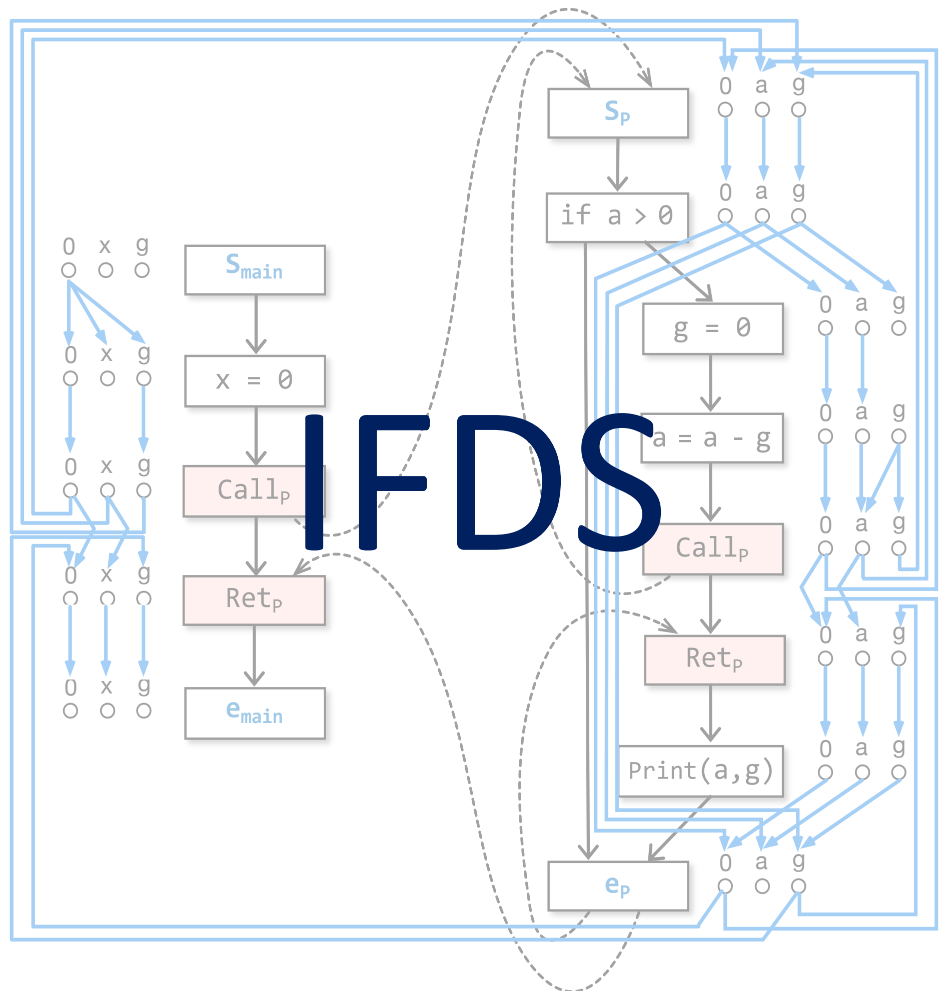
$e(_1eee)_1e \in L(\mathit{realizable})$



$e(_1eee)_1e(_2eee)_1$

$\notin L(\mathit{realizable})$

IFDS



A Program Analysis Framework via Graph Reachability

IFDS

"Precise Interprocedural Dataflow Analysis via Graph Reachability"

Thomas Reps, Susan Horwitz, and Mooly Sagiv, POPL'95

IFDS (Interprocedural, Finite, Distributive, Subset Problem)

IFDS

"Precise Interprocedural Dataflow Analysis via Graph Reachability"

Thomas Reps, Susan Horwitz, and Mooly Sagiv, POPL'95

IFDS (Interprocedural, Finite, Distributive, Subset Problem)

IFDS is for interprocedural data flow analysis
with distributive flow functions over finite domains.

IFDS

"Precise Interprocedural Dataflow Analysis via Graph Reachability"

Thomas Reps, Susan Horwitz, and Mooly Sagiv, POPL'95

IFDS (Interprocedural, Finite, Distributive, Subset Problem)

IFDS is for interprocedural data flow analysis
with distributive flow functions over finite domains.

Provide meet-over-all-realizable-paths (MRP) solution.

Meet-Over-All-Realizable-Paths (MRP)

Path function for path *p*, denoted as pf_p, is a composition of flow functions for all edges (sometimes nodes) on *p*.

Recall

$$\boxed{pf_p = f_n \circ \ldots \circ f_2 \circ f_1}$$

Meet-Over-All-Realizable-Paths (MRP)

Path function for path p, denoted as pf_p, is a composition of flow functions for all edges (sometimes nodes) on p.

Recall

$$\boxed{pf_p = f_n \circ \ldots \circ f_2 \circ f_1}$$

$$\boxed{\mathrm{MOP}_n = \bigsqcup_{p \in \mathrm{Paths}(start,\, n)} pf_p(\perp)}$$

For each node n, MOP_n provides a "meet-over-all-paths" solution where $\mathrm{Paths}(start, n)$ denotes the set of paths in CFG from the start node to n.

Meet-Over-All-Realizable-Paths (MRP)

Path function for path *p*, denoted as pf_p, is a composition of flow functions for all edges (sometimes nodes) on *p*.

Recall

$$pf_p = f_n \circ \ldots \circ f_2 \circ f_1$$

$$\mathrm{MOP}_n = \bigsqcup_{p \in \mathrm{Paths}(start, n)} pf_p(\bot)$$

For each node n, MOP_n provides a "meet-over-all-paths" solution where $\mathrm{Paths}(start, n)$ denotes the set of paths in CFG from the start node to n.

$$\mathrm{MRP}_n = \bigsqcup_{p \in \mathrm{RPaths}(start, n)} pf_p(\bot)$$

For each node n, MRP_n provides a "meet-over-all-realizable-paths" solution where $\mathrm{RPaths}(start, n)$ denotes the set of realizable paths (the path's word is in the language $L(realizable)$) from the start node to n.

Meet-Over-All-Realizable-Paths (MRP)

Path function for path *p*, denoted as pf_p, is a composition of flow functions for all edges (sometimes nodes) on *p*.

Recall

$$\boxed{pf_p = f_n \circ \ldots \circ f_2 \circ f_1}$$

$$\boxed{\mathrm{MOP}_n = \bigsqcup_{p \in \mathrm{Paths}(start, n)} pf_p(\bot)}$$

For each node n, MOP_n provides a "meet-over-all-paths" solution where $\mathrm{Paths}(start, n)$ denotes the set of paths in CFG from the start node to n.

$$\boxed{\mathrm{MRP}_n = \bigsqcup_{p \in \mathrm{RPaths}(start, n)} pf_p(\bot)}$$

For each node n, MRP_n provides a "meet-over-all-realizable-paths" solution where $\mathrm{RPaths}(start, n)$ denotes the set of realizable paths (the path's word is in the language *L(realizable)*) from the start node to n.

$$\boxed{\mathrm{MRP}_n \sqsubseteq \mathrm{MOP}_n}$$

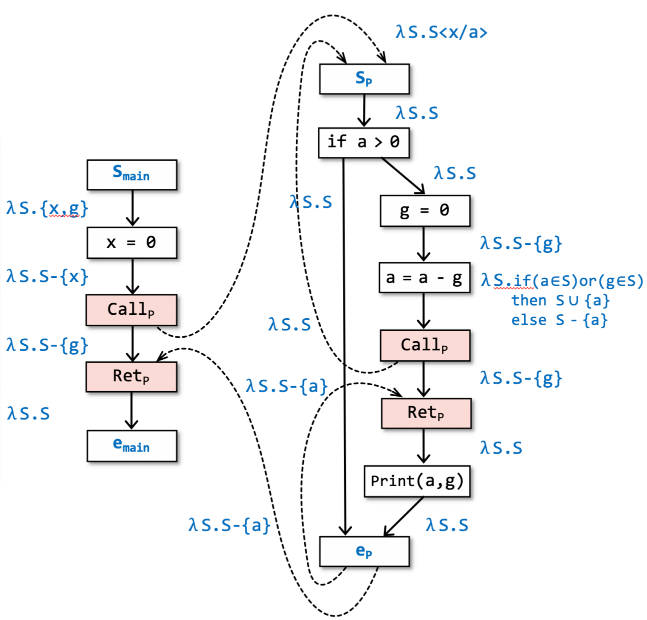
Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

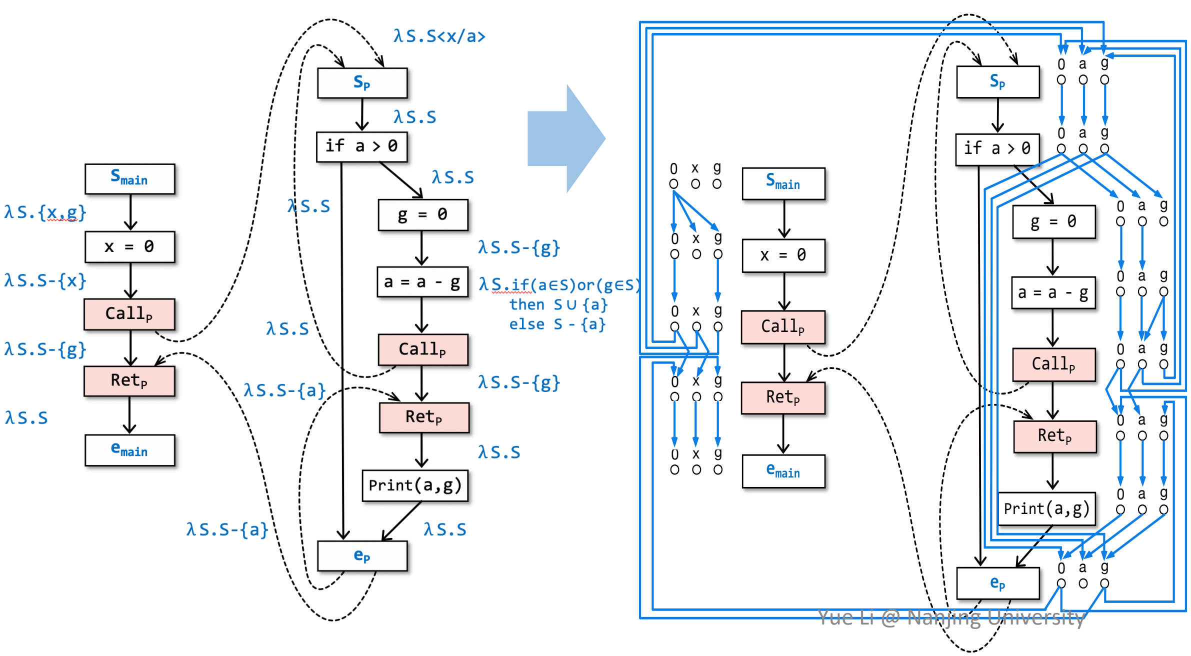
- Build a supergraph G* for P and
  define flow functions for edges in G* based on Q



Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

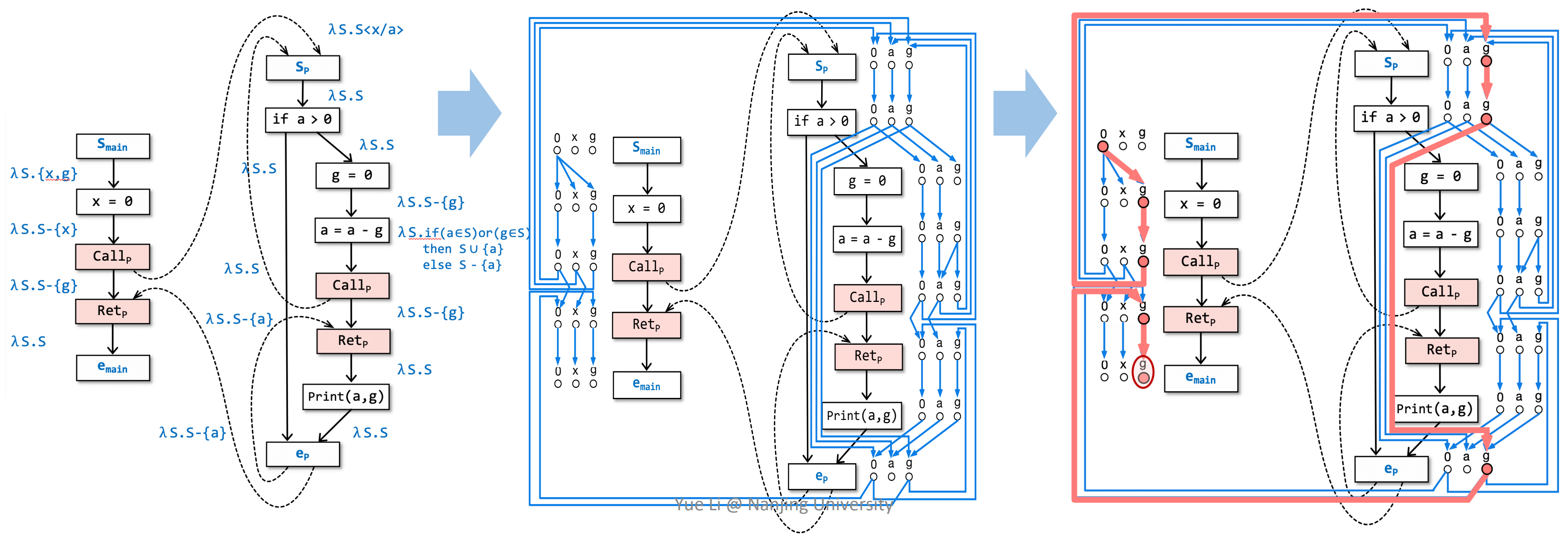
- Build a supergraph G* for P and
  define flow functions for edges in G* based on Q
- Build exploded supergraph G# for P by transforming
  flow functions to representation relations (graphs)



Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

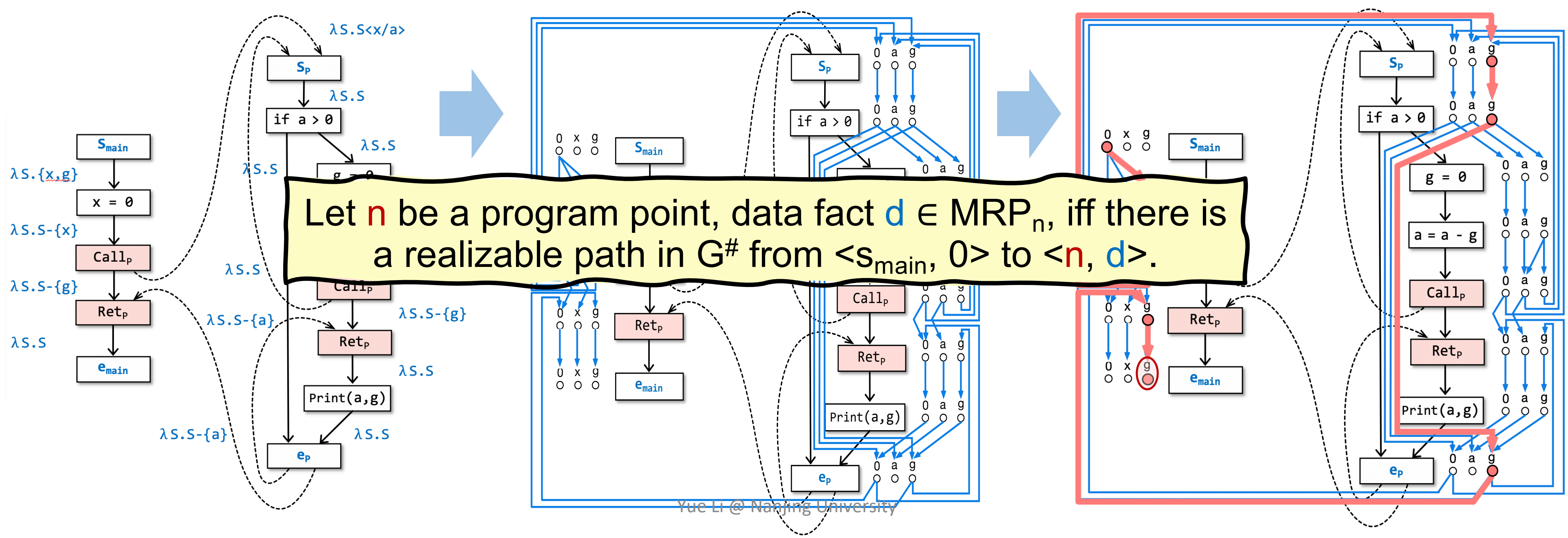
- Build a supergraph G* for P and
  define flow functions for edges in G* based on Q
- Build exploded supergraph $G^{\#}$ for P by transforming
  flow functions to representation relations (graphs)
- Q can be solved as graph reachability problems (find out MRP solutions)
  via applying Tabulation algorithm on $G^{\#}$



Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

- Build a supergraph G* for P and
  define flow functions for edges in G* based on Q
- Build exploded supergraph $G^\#$ for P by transforming
  flow functions to representation relations (graphs)
- Q can be solved as graph reachability problems (find out MRP solutions)
  via applying Tabulation algorithm on $G^\#$

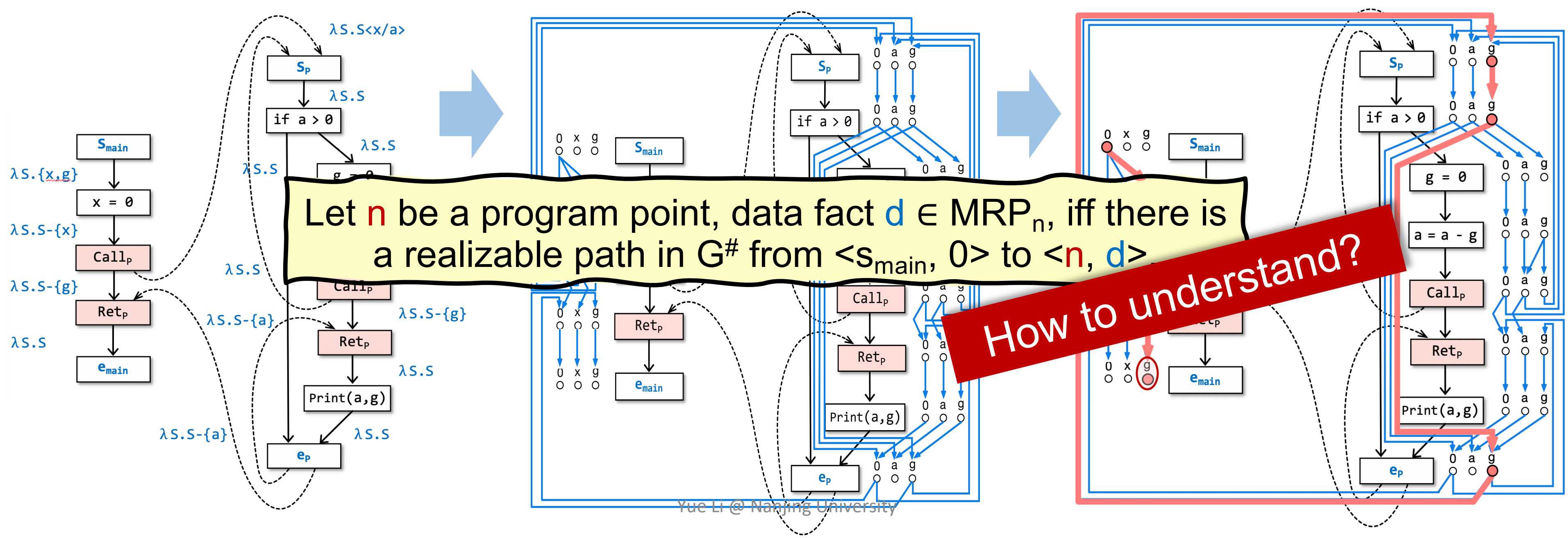


Let n be a program point, data fact d ∈ MRP_n, iff there is a realizable path in $G^\#$ from <s_{main}, 0> to <n, d>.

Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

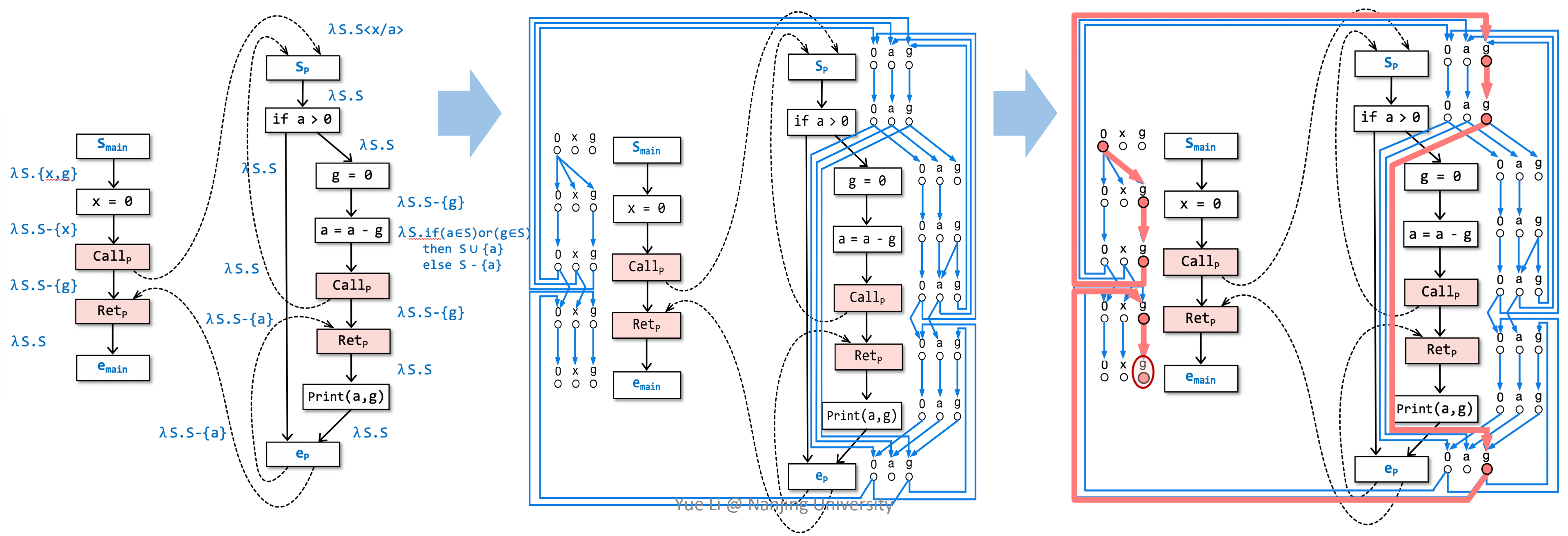
- Build a supergraph G* for P and define flow functions for edges in G* based on Q
- Build exploded supergraph $G^{\#}$ for P by transforming flow functions to representation relations (graphs)
- Q can be solved as graph reachability problems (find out MRP solutions) via applying Tabulation algorithm on $G^{\#}$



Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

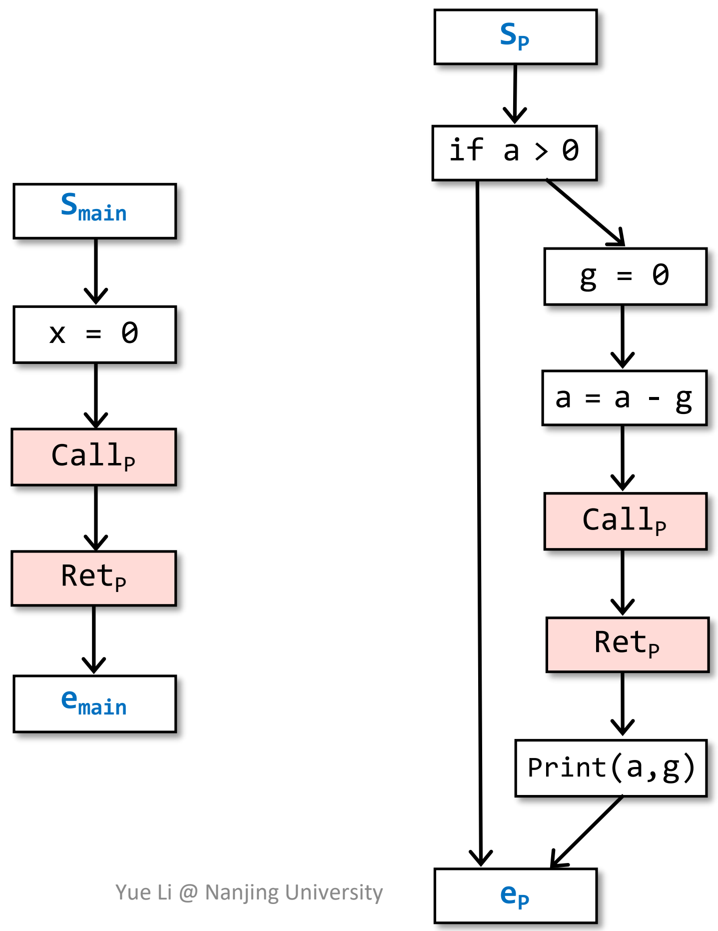
- Build a supergraph G* for P and
  define flow functions for edges in G* based on Q
- Build exploded supergraph $G^{\#}$ for P by transforming flow functions to representation relations (graphs)
- Q can be solved as graph reachability problems (find out MRP solutions) via applying Tabulation algorithm on $G^{\#}$



Supergraph

In IFDS, a program is represented by G* = (N*, E*) called a supergraph.

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```

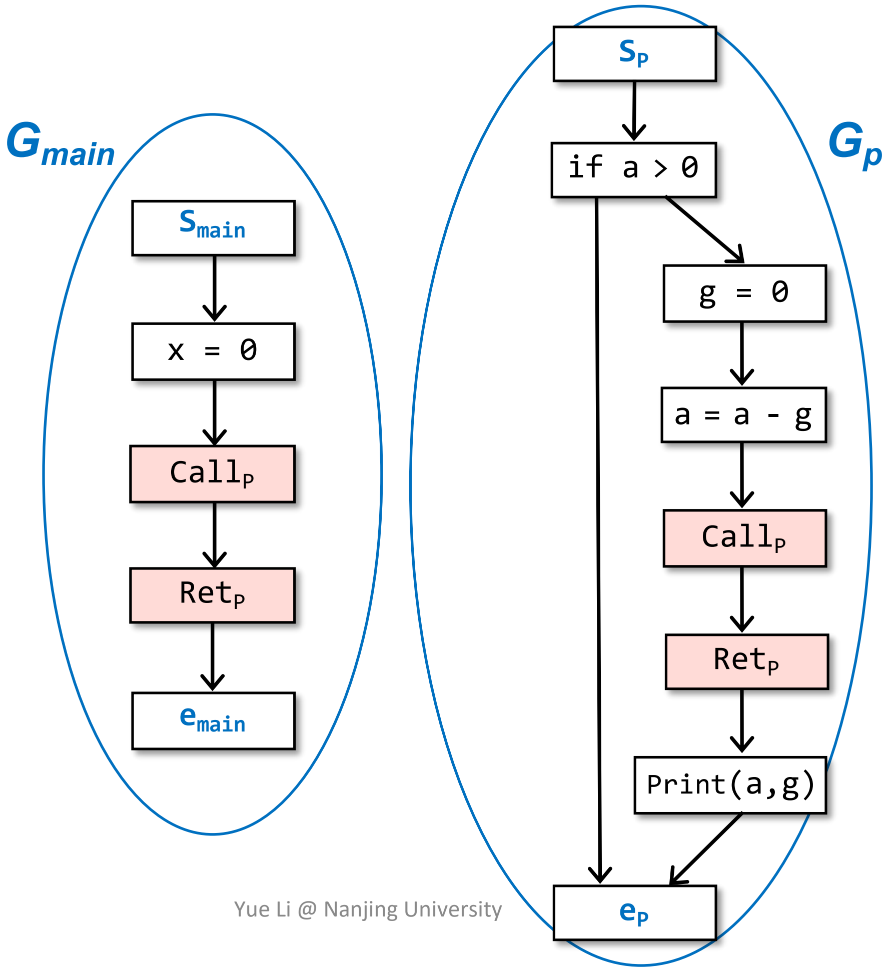


Supergraph

In IFDS, a program is represented by G* = (N*, E*) called a supergraph.

- *G* consists of a collection of flow graphs G_1, G_2,… (one for each procedure)*

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```

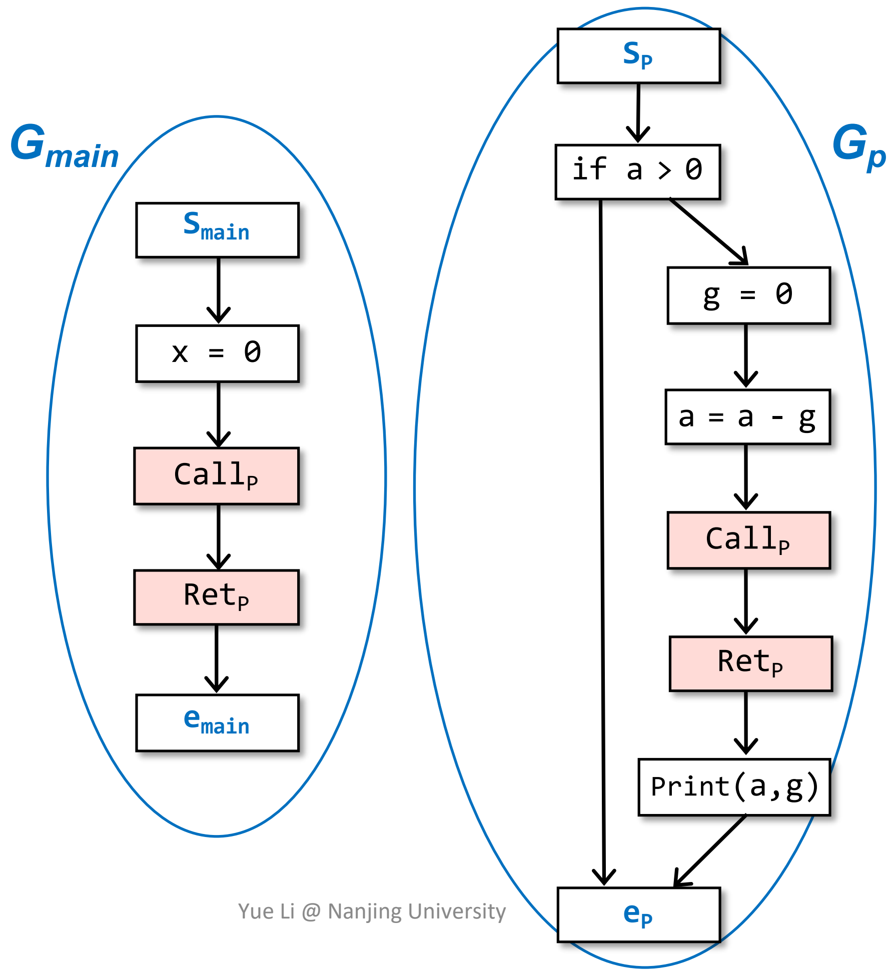


Supergraph

In IFDS, a program is represented by $G^* = (N^*, E^*)$ called a supergraph.

- *G^* consists of a collection of flow graphs G_1, G_2,… (one for each procedure)*
- *Each flowgraph G_p has a unique start node s_p, and a unique exit node e_p*

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```

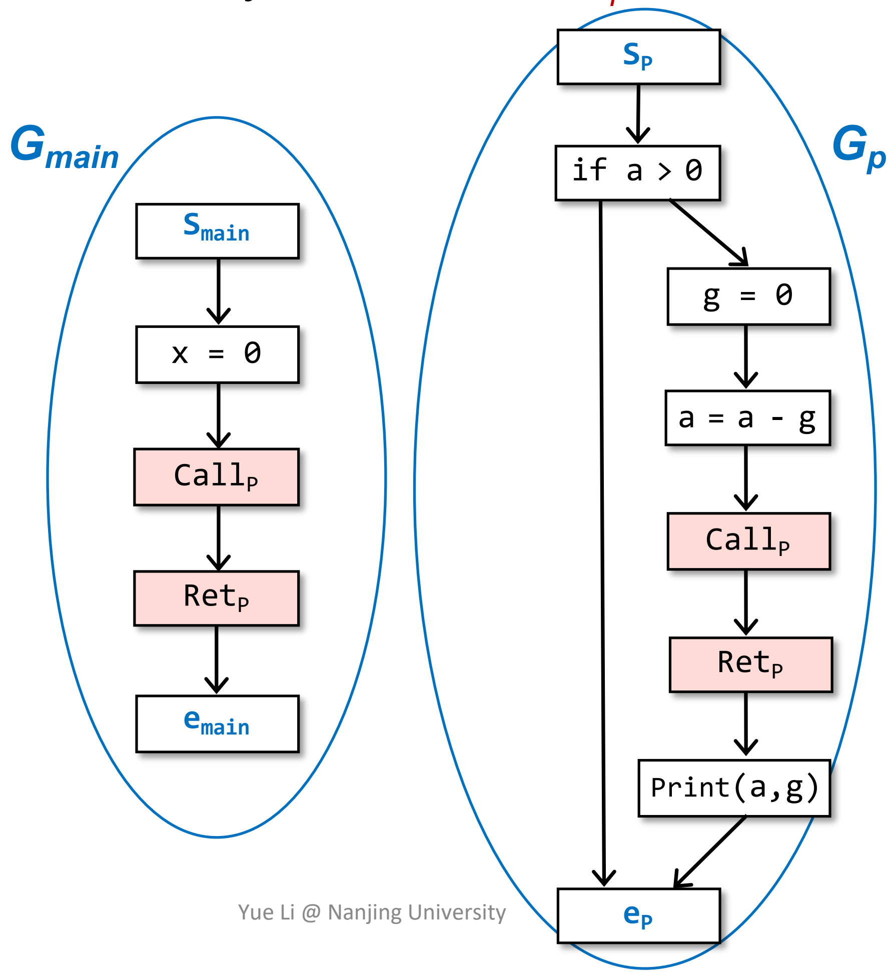


Supergraph

In IFDS, a program is represented by G* = (N*, E*) called a supergraph.

- *G* consists of a collection of flow graphs G_1, G_2,… (one for each procedure)*
- *Each flowgraph G_p has a unique start node s_p, and a unique exit node e_p*
- *A procedure call is represented by a call node $Call_p$, and a return-site node Ret_p*

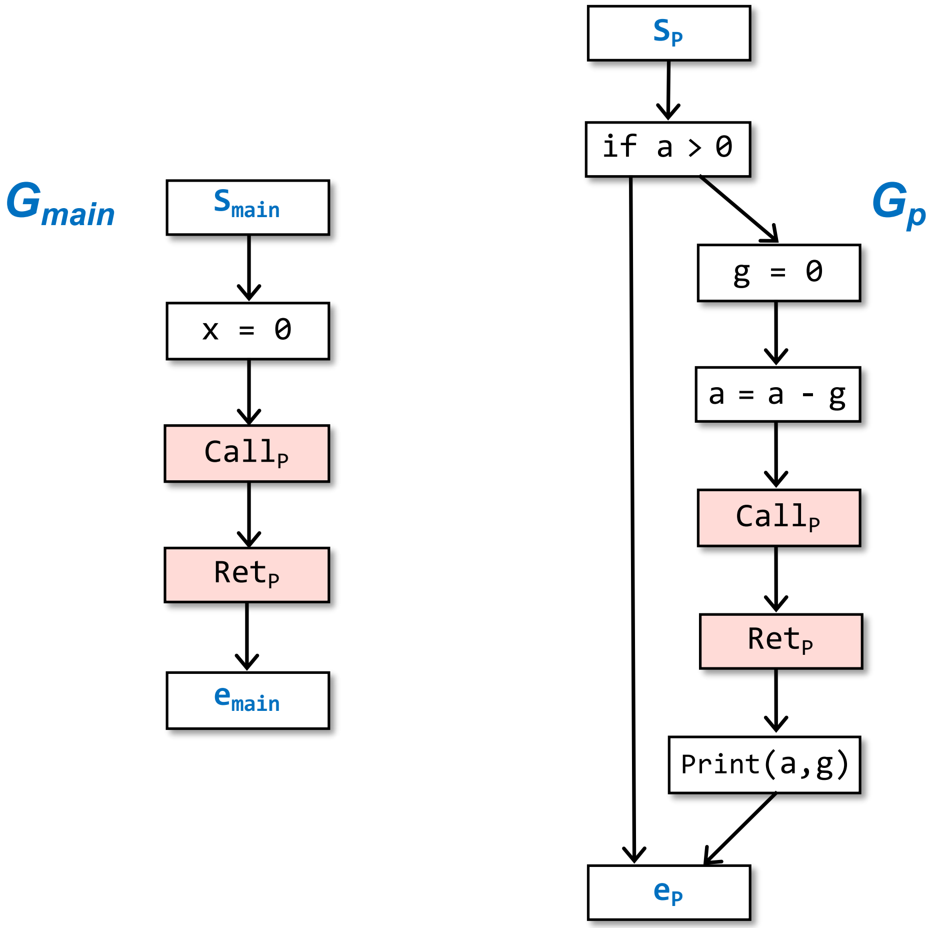
```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



G has three edges for each procedure call:*

Supergraph

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```

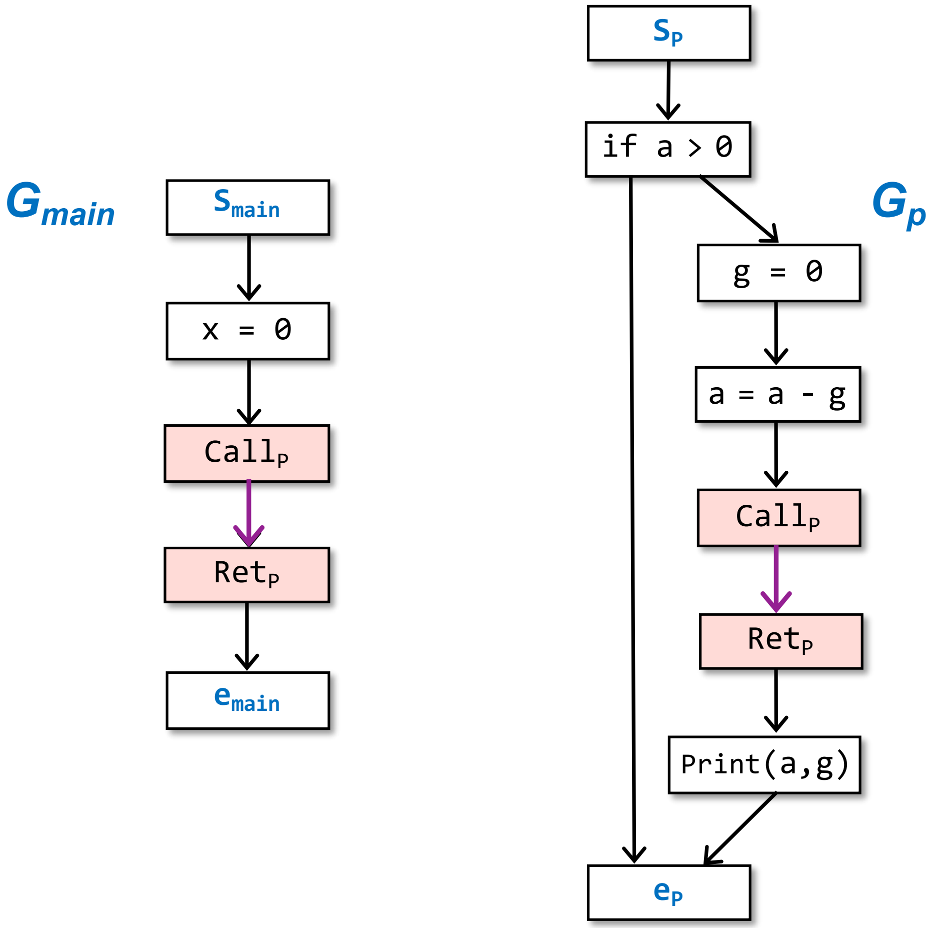


Supergraph

G has three edges for each procedure call:*

- *An intraprocedural call-to-return-site edge from $Call_p$ to Ret_p*

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```

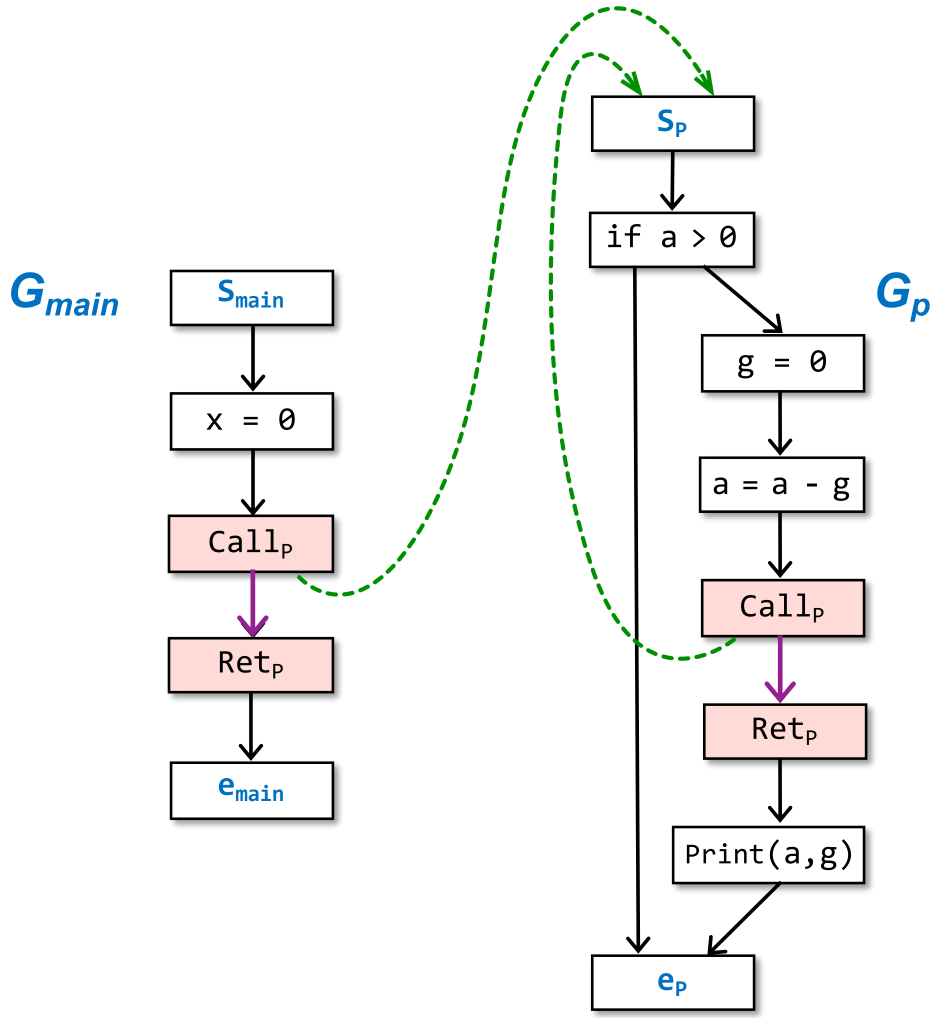


Supergraph

G has three edges for each procedure call:*

- *An intraprocedural call-to-return-site edge from $Call_p$ to Ret_p*
- *An interprocedural call-to-start edge from $Call_p$ to s_p of the called procedure*

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```

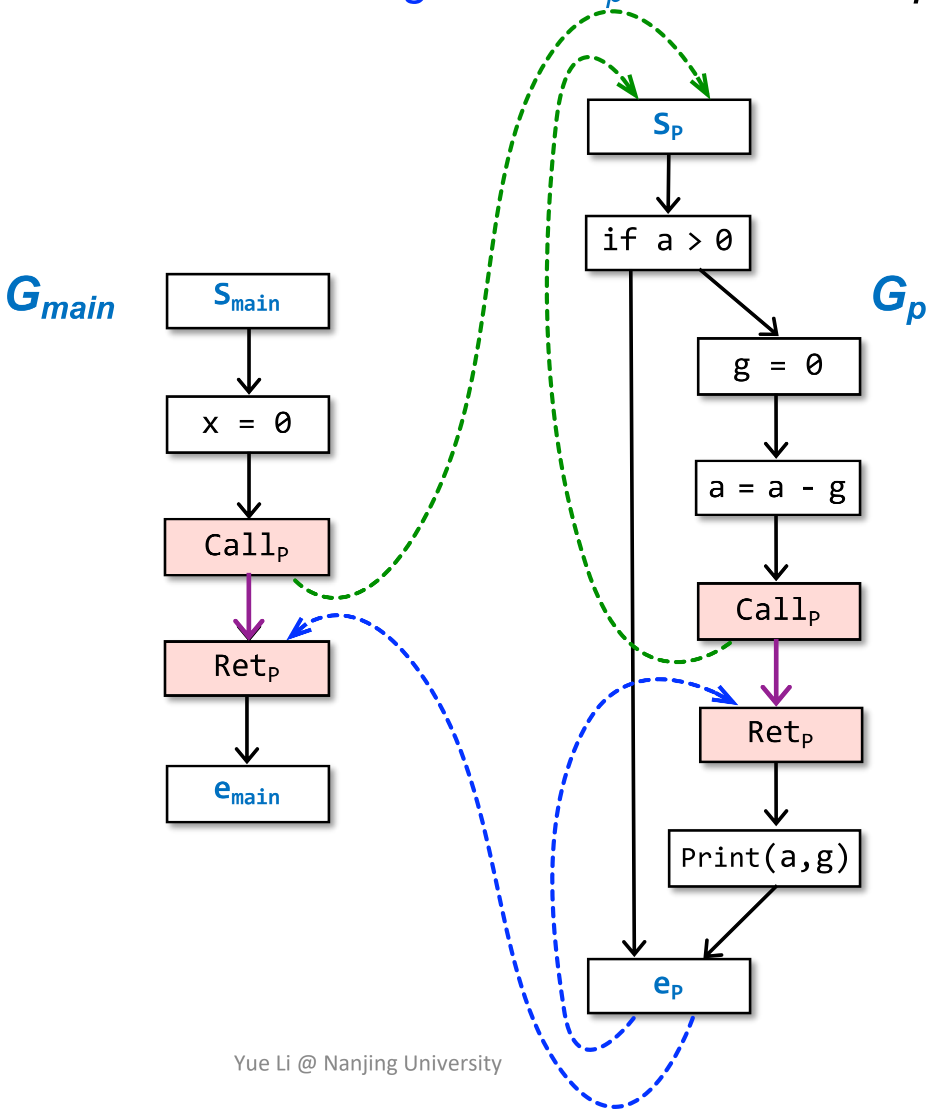


Supergraph

G has three edges for each procedure call:*

- *An intraprocedural call-to-return-site edge from $Call_p$ to Ret_p*
- *An interprocedural call-to-start edge from $Call_p$ to s_p of the called procedure*
- *An interprocedural exit-to-return-site edge from e_p of the called procedure to Ret_p*

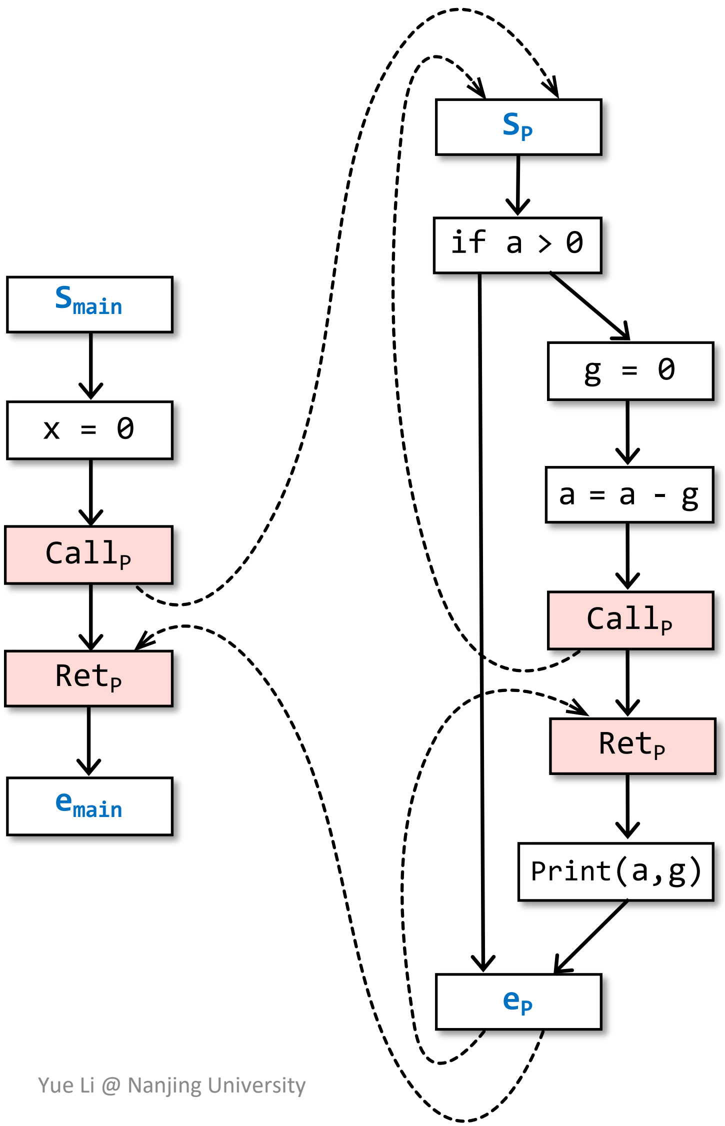
```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a – g;
    P(a);
    Print(a,g);
  }
}
```



Design Flow Functions

"Possibly-uninitialized variables": for each node n ∈ N*, determine the set of variables that may be uninitialized before execution reaches n.

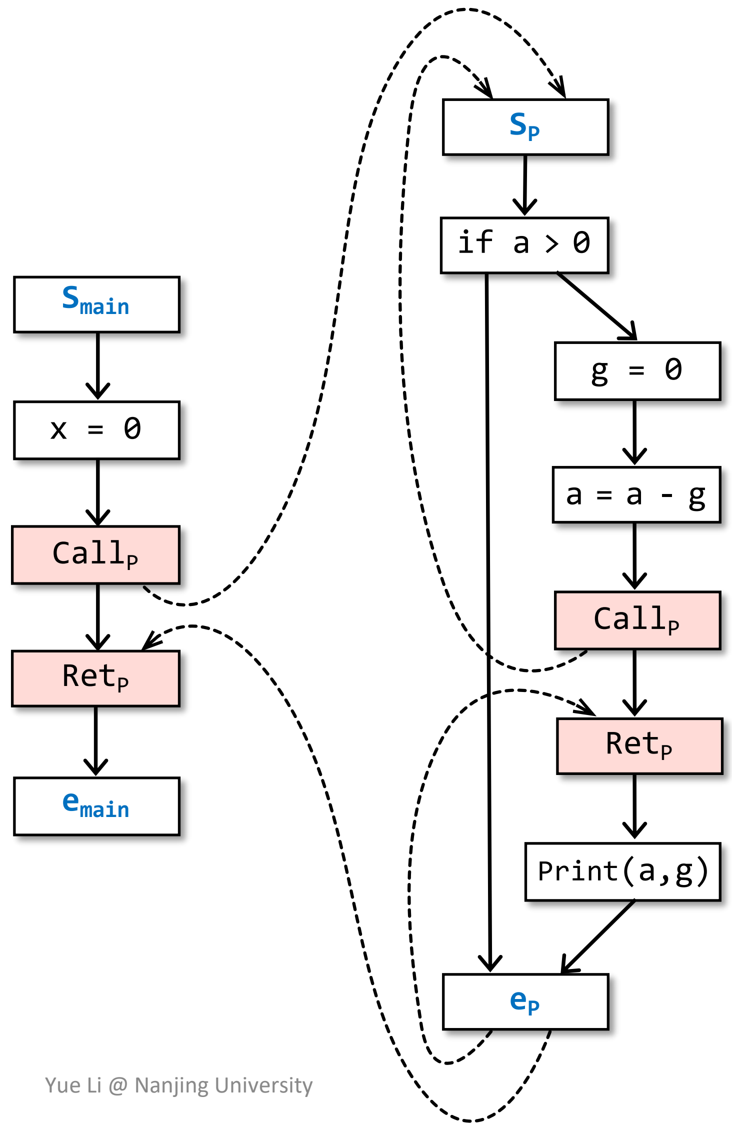
```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



Design Flow Functions

"Possibly-uninitialized variables": for each node n ∈ N*, determine the set of variables that may be uninitialized before execution reaches n.

$$\lambda\, e_{param} .\, e_{body}$$

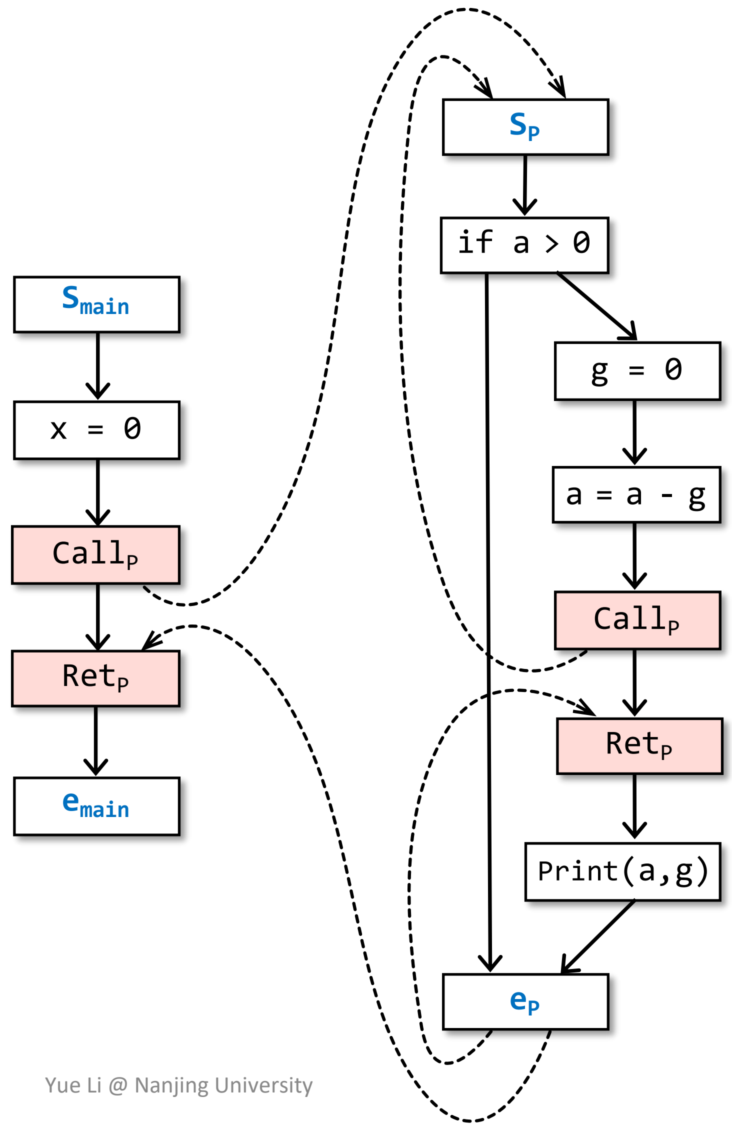


Design Flow Functions

"Possibly-uninitialized variables": for each node n ∈ N*, determine the set of variables that may be uninitialized before execution reaches n.

$$\lambda\, e_{param} \,.\, e_{body}$$

e.g., $\lambda\, x \,.\, x+1$



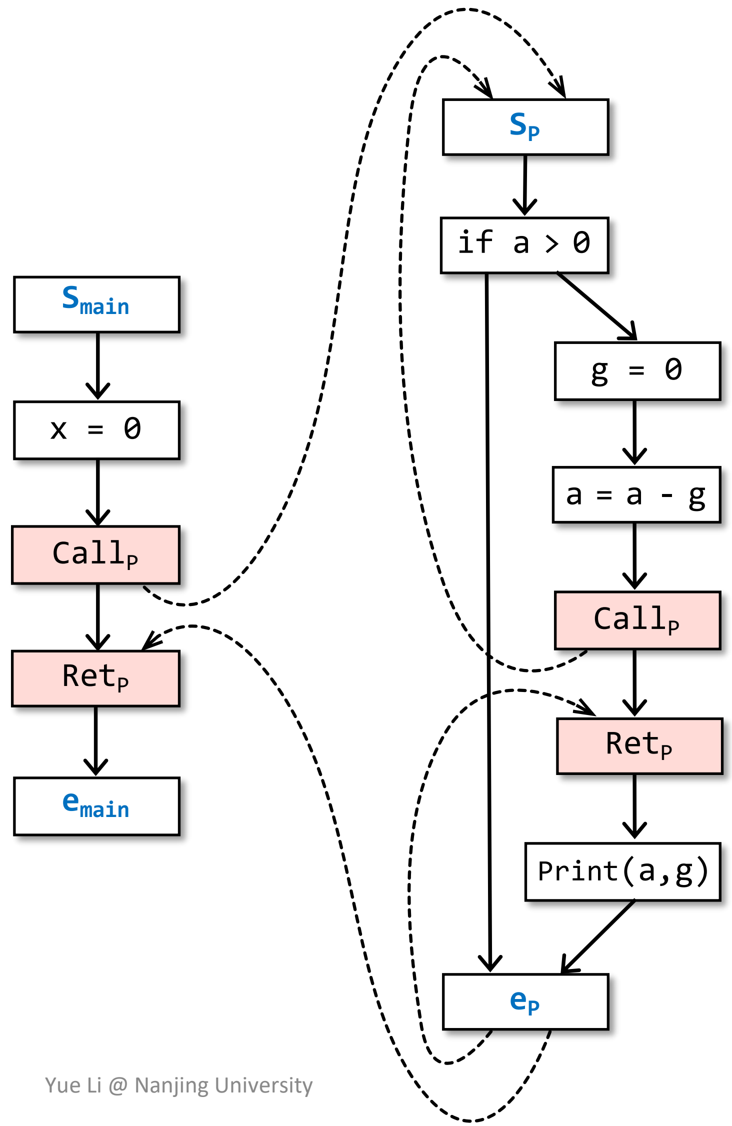
Design Flow Functions

"Possibly-uninitialized variables": for each node n ∈ N*, determine the set of variables that may be uninitialized before execution reaches n.

$\lambda\, e_{param} .\, e_{body}$

e.g., λ x.x+1

(λ x.x+1)3



Design Flow Functions

"Possibly-uninitialized variables": for each node n ∈ N*, determine the set of variables that may be uninitialized before execution reaches n.

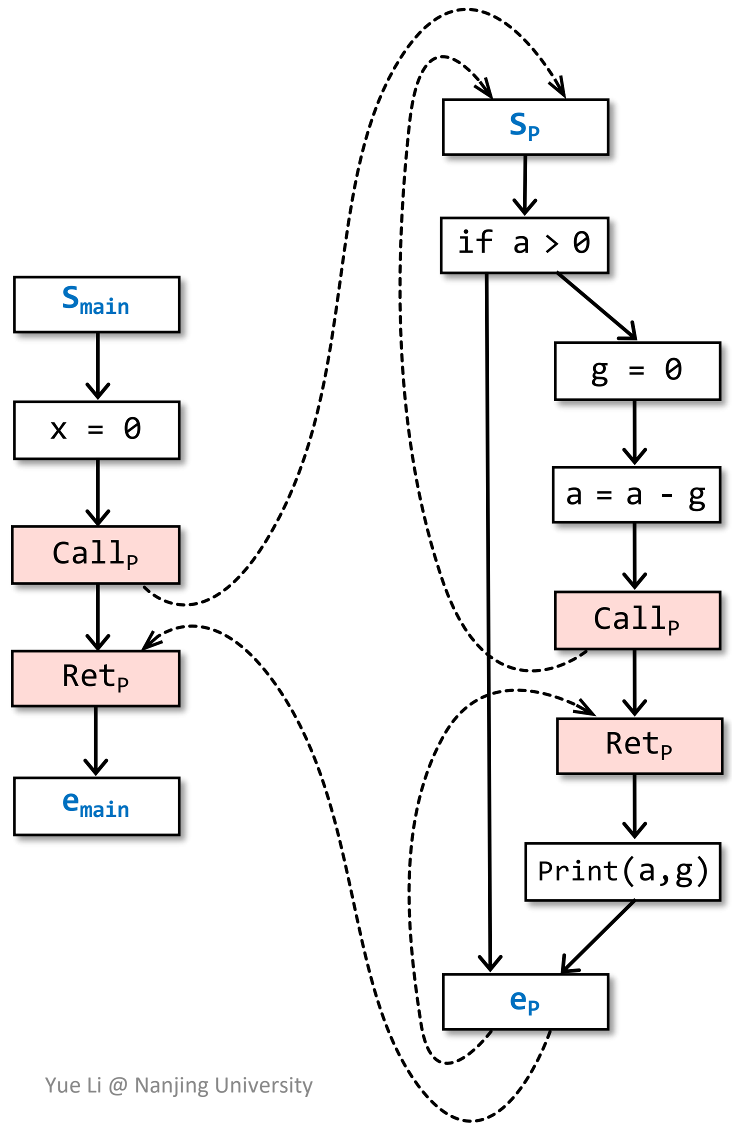
$\lambda\, e_{param} .\, e_{body}$

e.g., $\lambda\, x.x+1$

$(\lambda\, x.x+1)3$

$\Rightarrow 3+1$

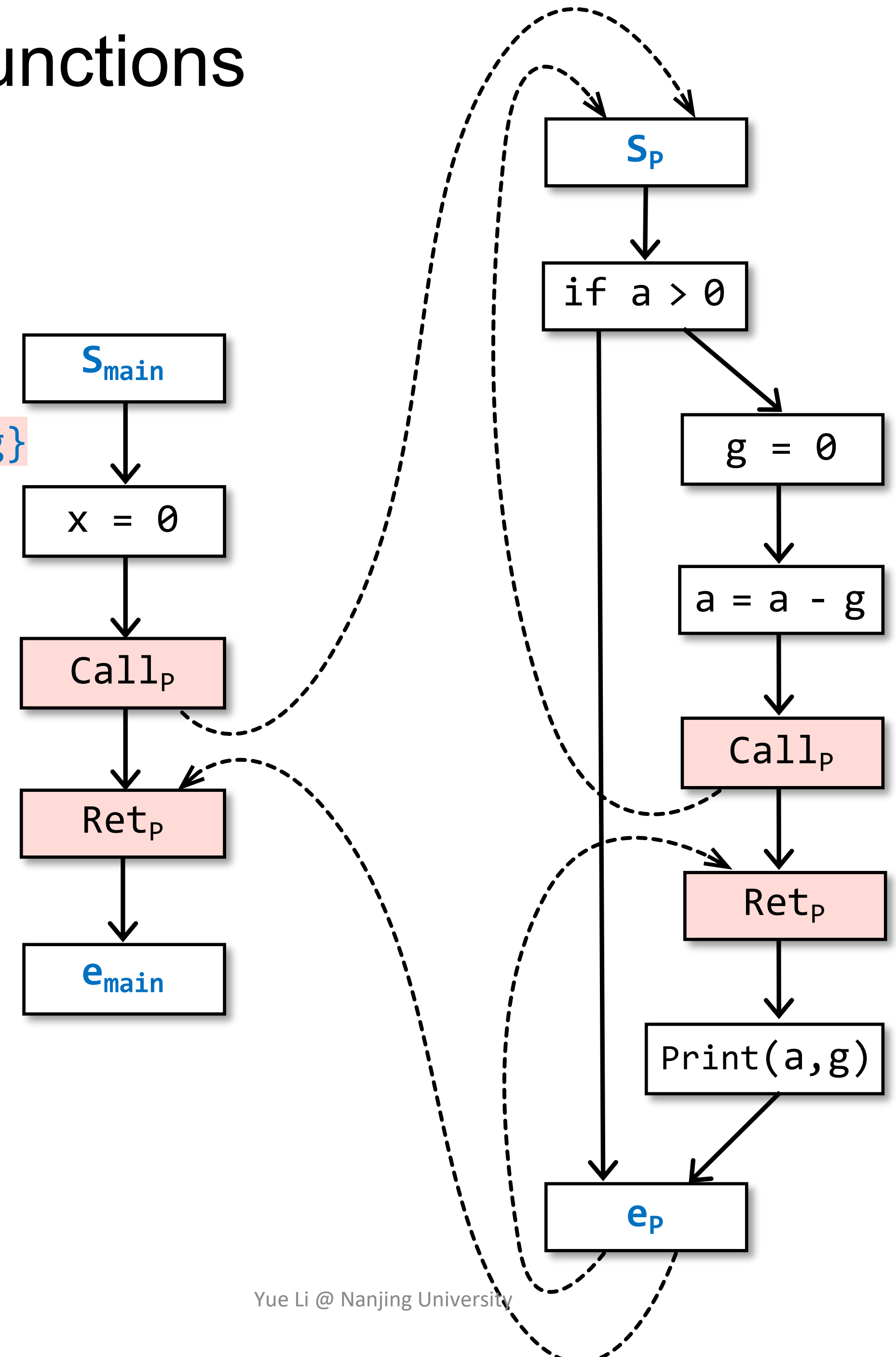
$\Rightarrow 4$



Design Flow Functions

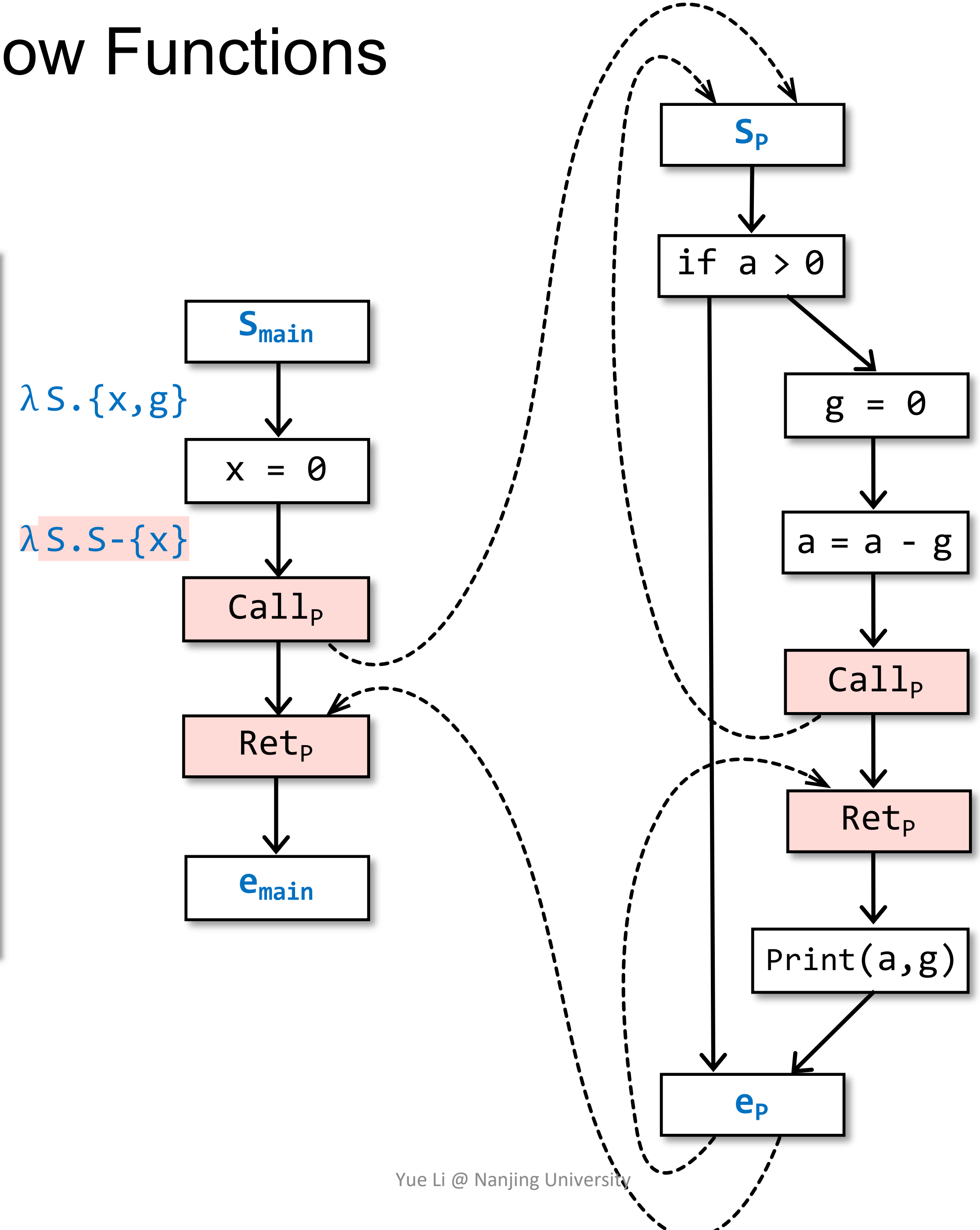
```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```

λS.{x,g}



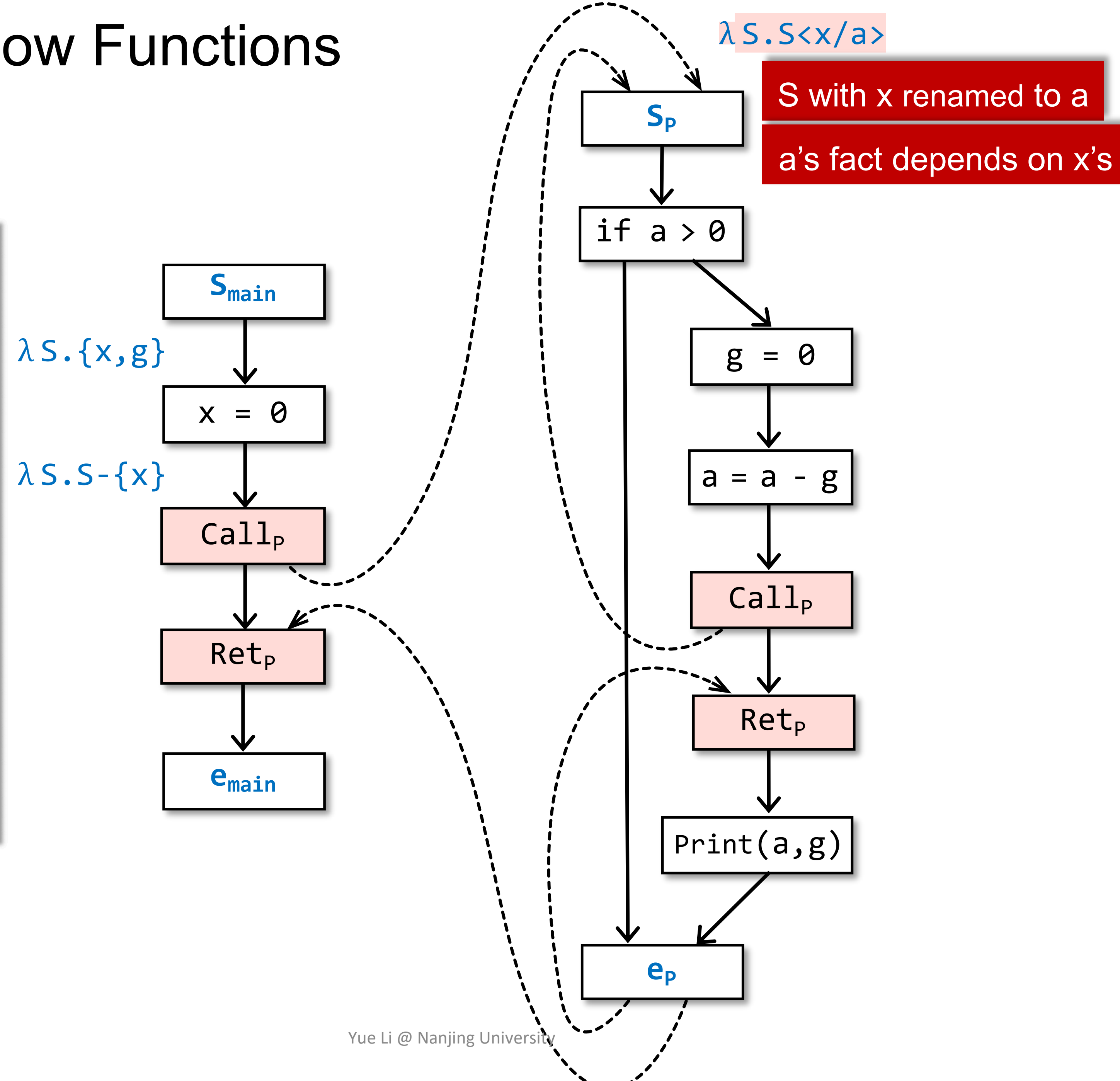
Design Flow Functions

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



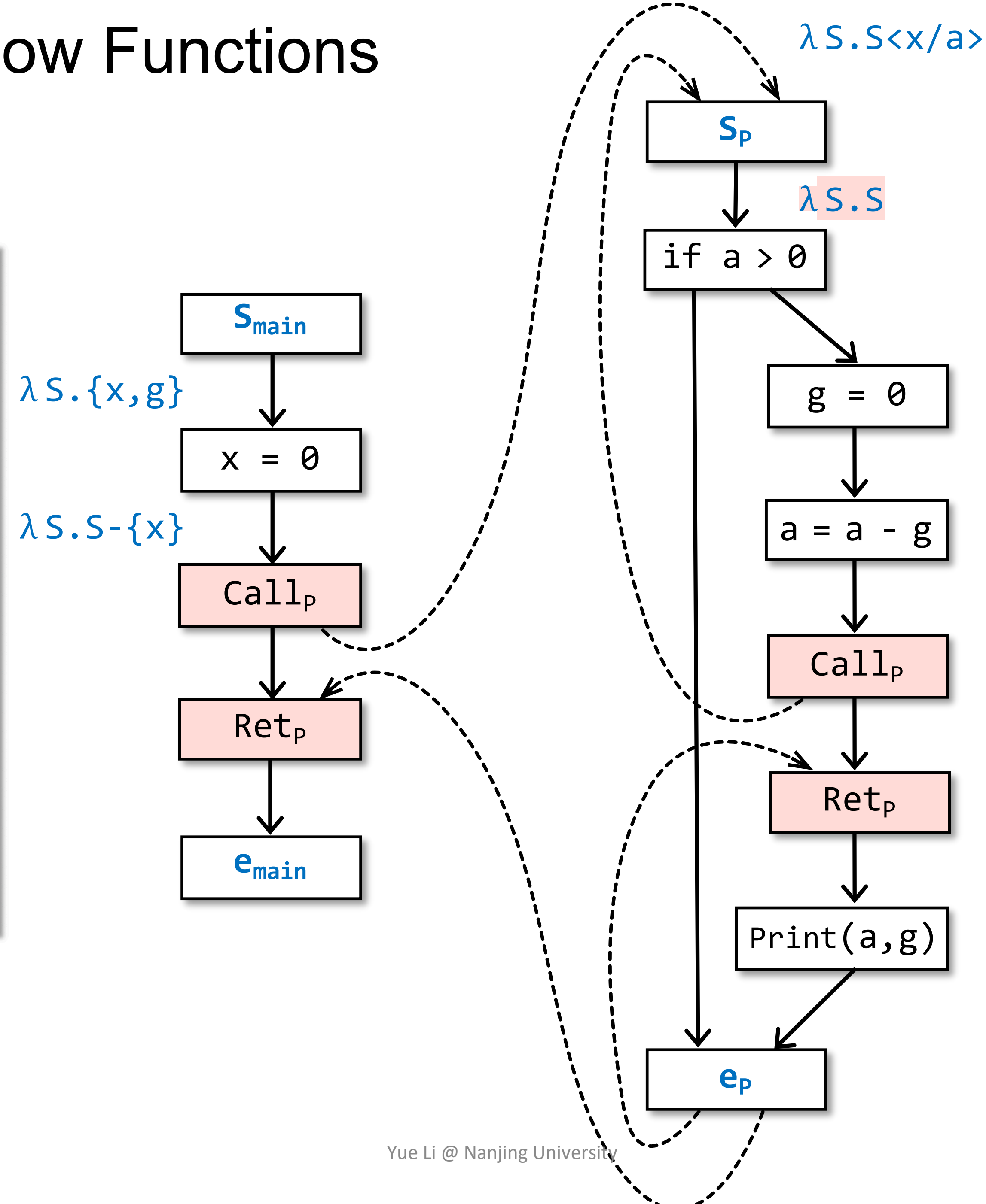
Design Flow Functions

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



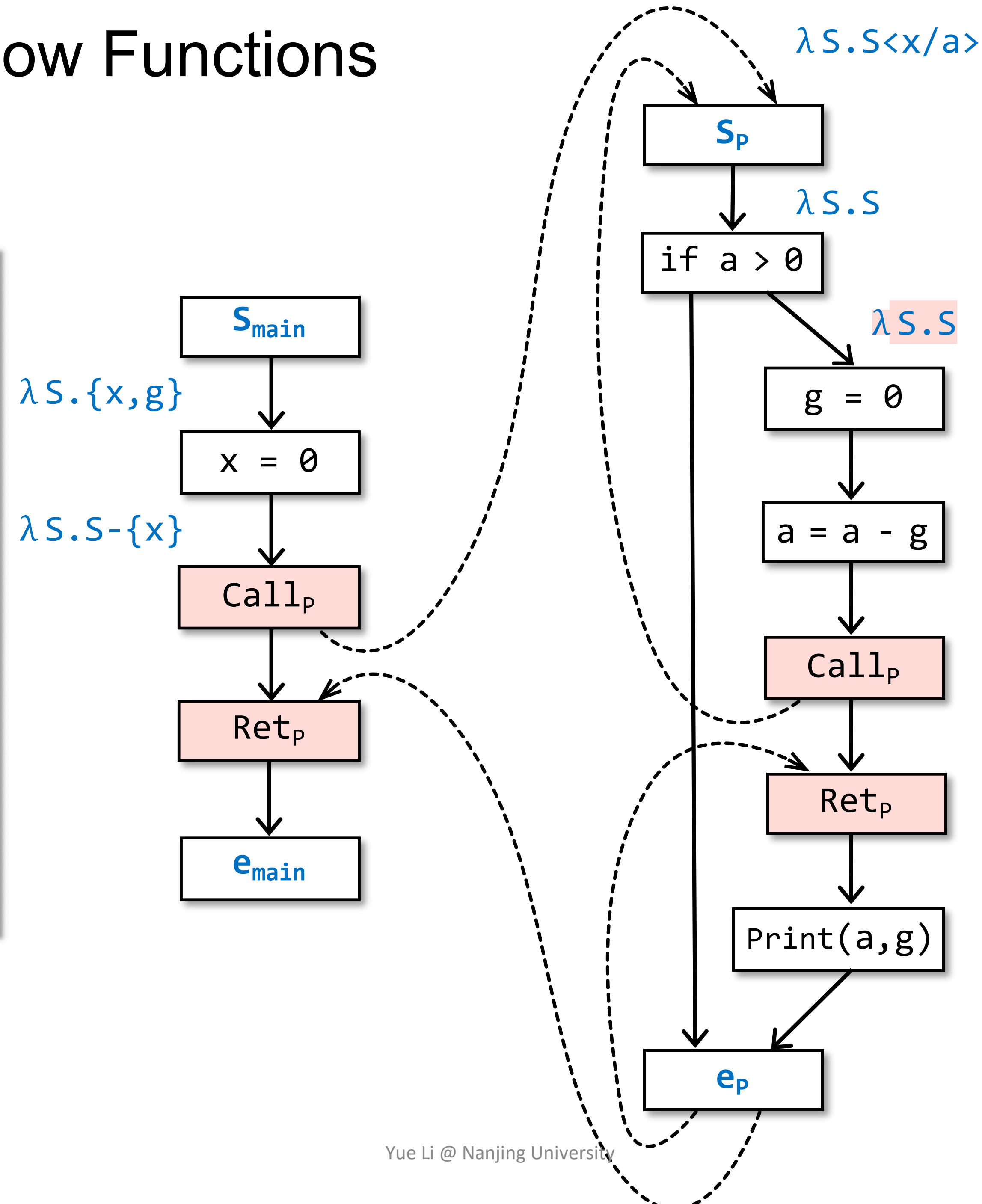
Design Flow Functions

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



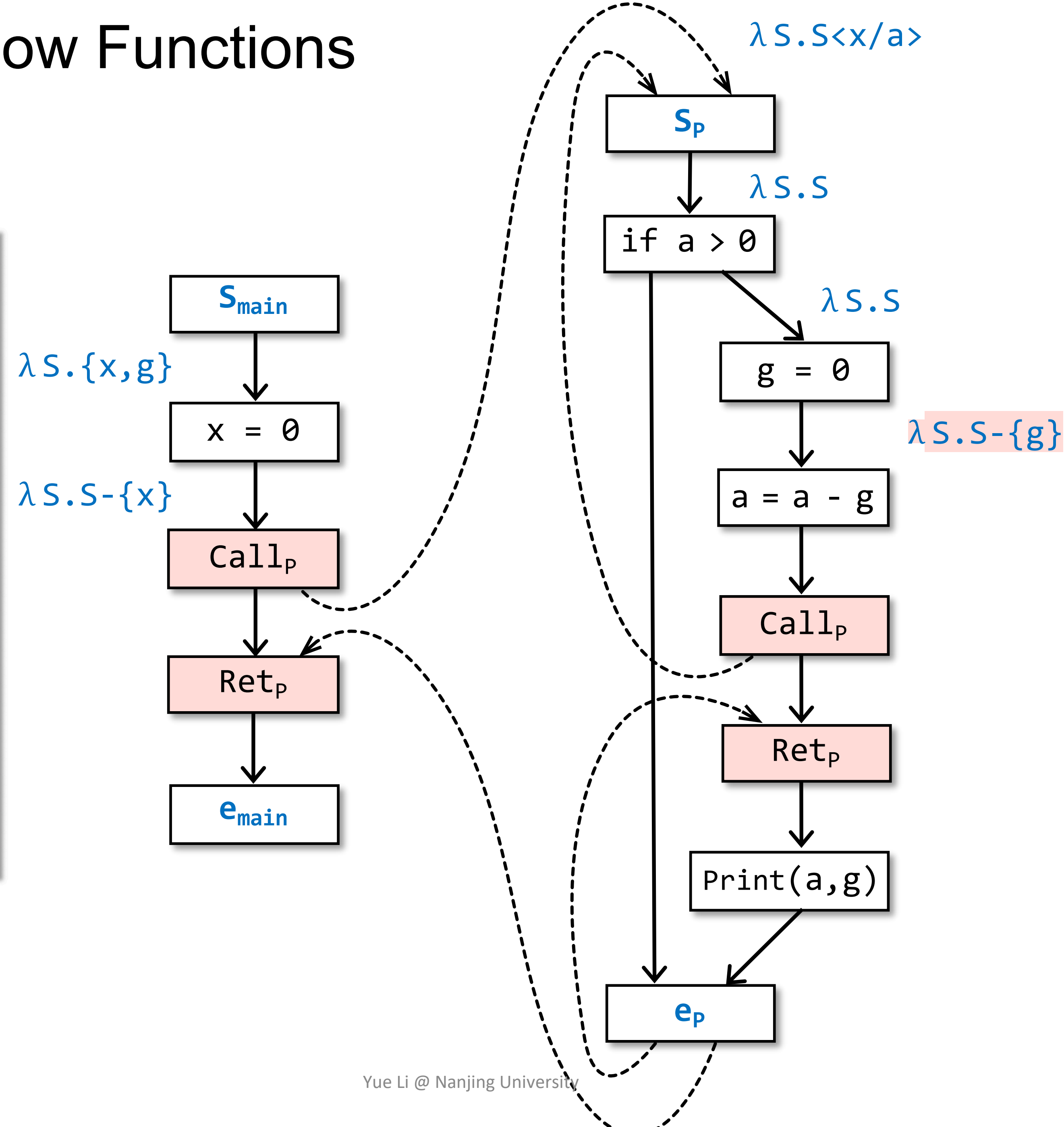
Design Flow Functions

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



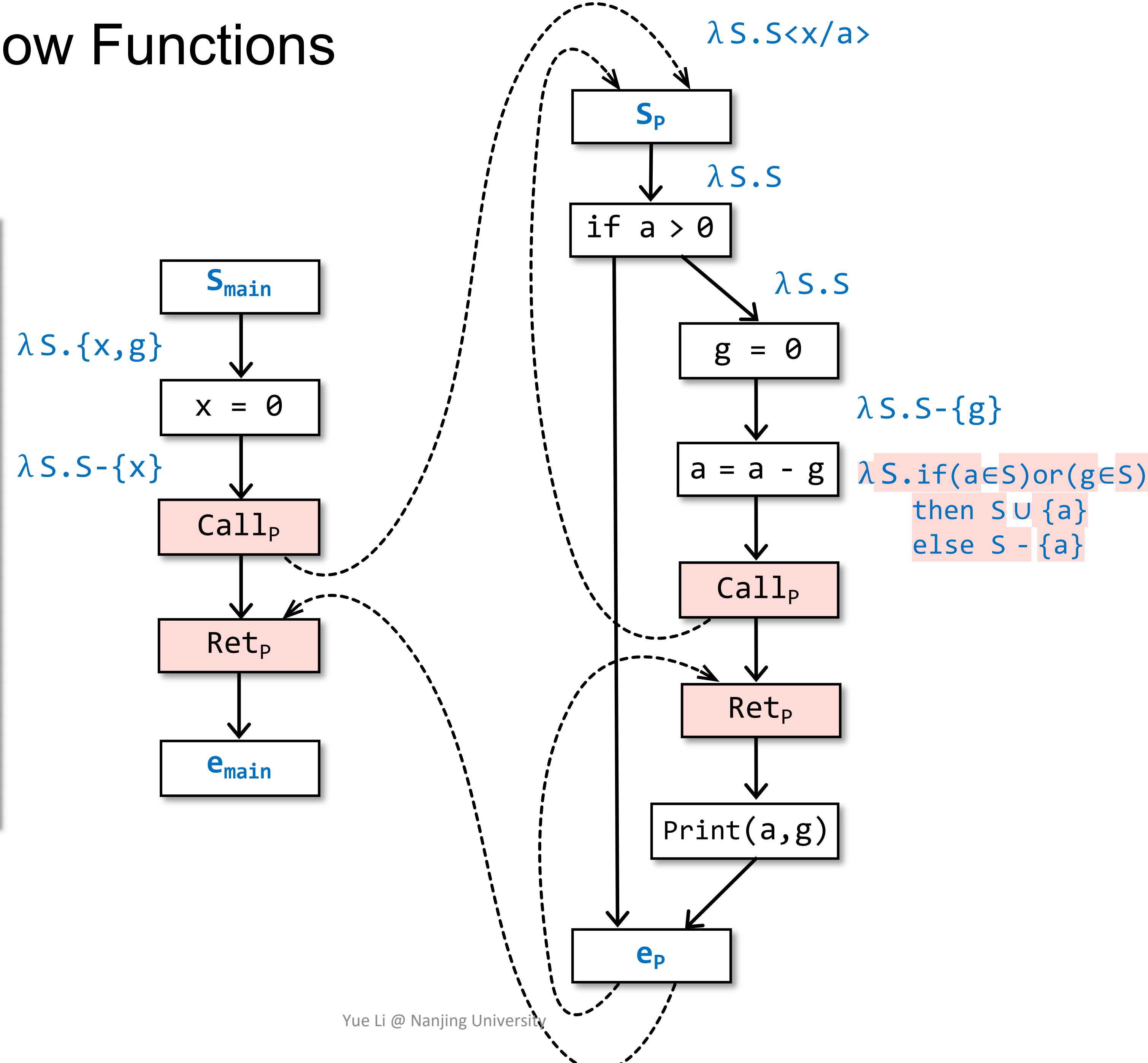
Design Flow Functions

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



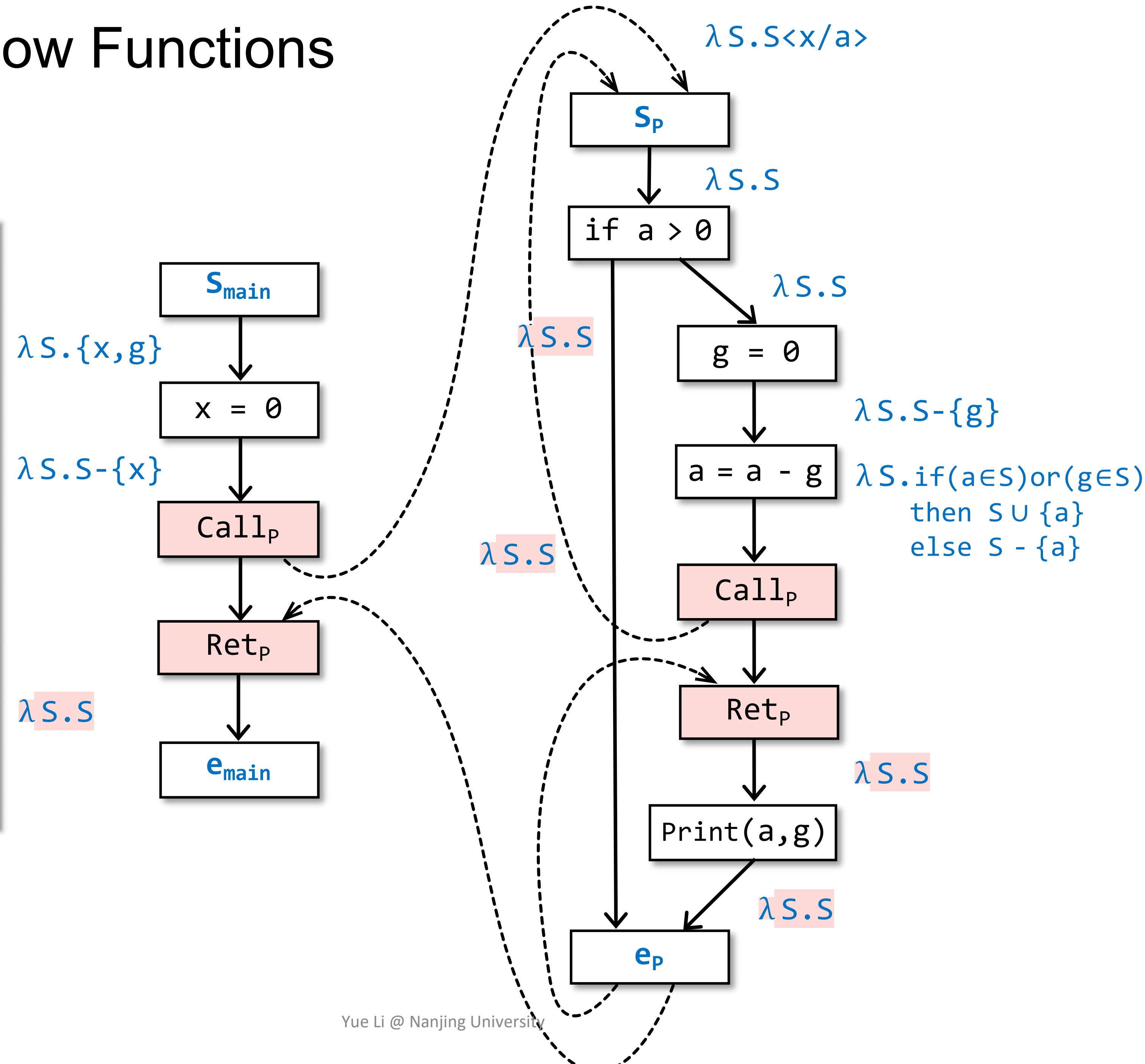
Design Flow Functions

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a – g;
    P(a);
    Print(a,g);
  }
}
```



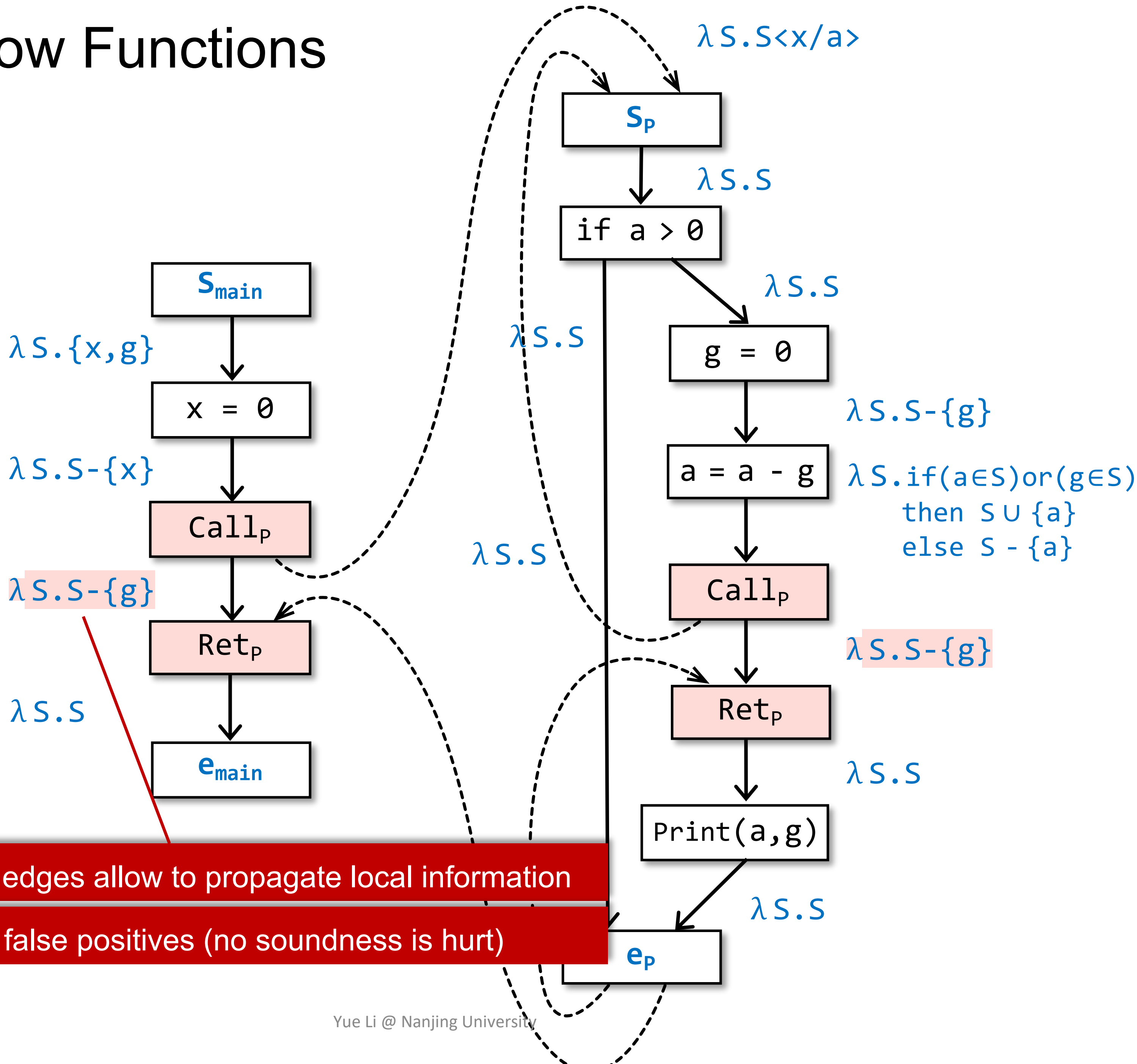
Design Flow Functions

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a – g;
    P(a);
    Print(a,g);
  }
}
```



Design Flow Functions

```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a – g;
    P(a);
    Print(a,g);
  }
}
```

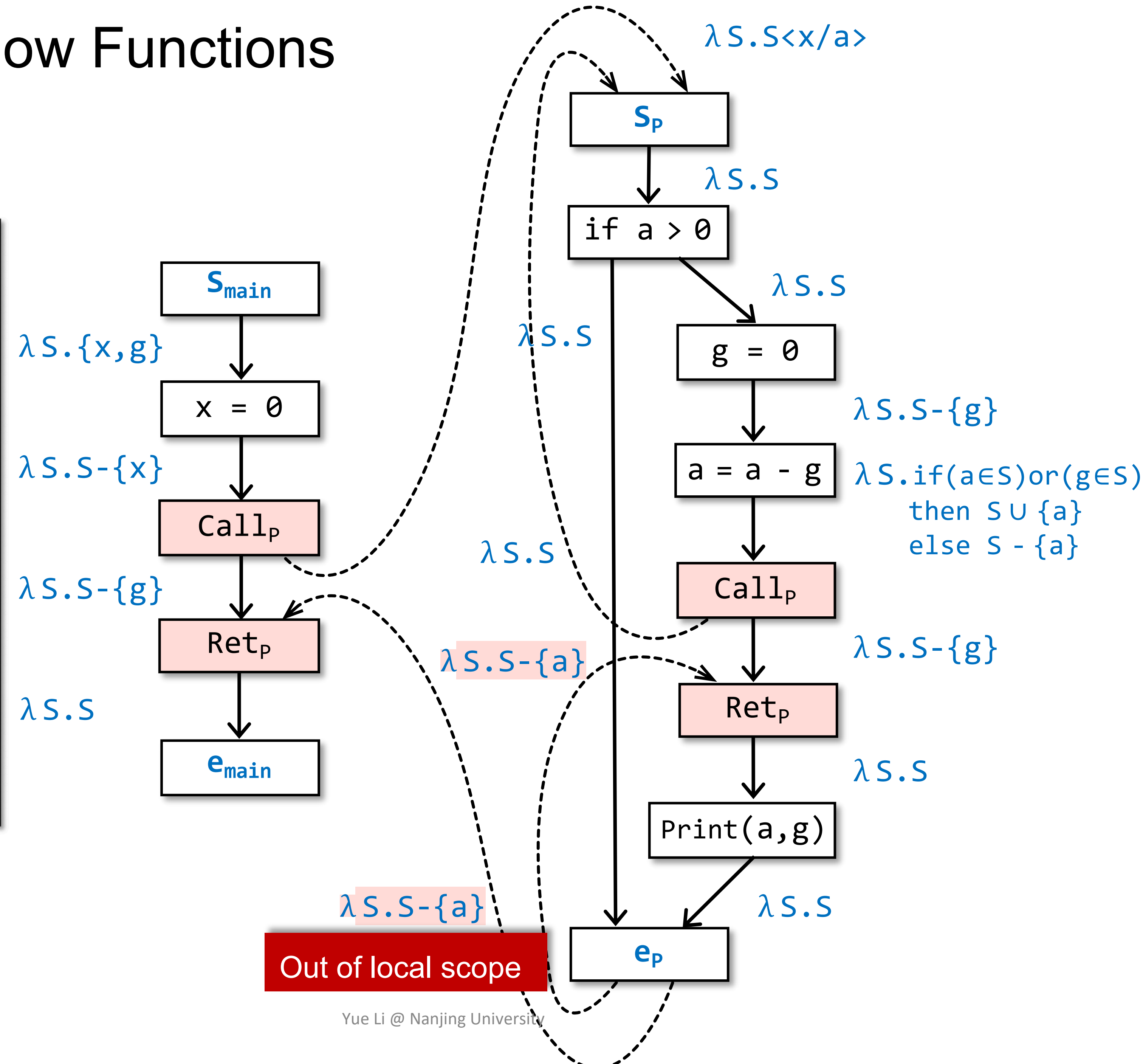


“call-to-return-site” edges allow to propagate local information

S-{g} helps reduce false positives (no soundness is hurt)

Design Flow Functions

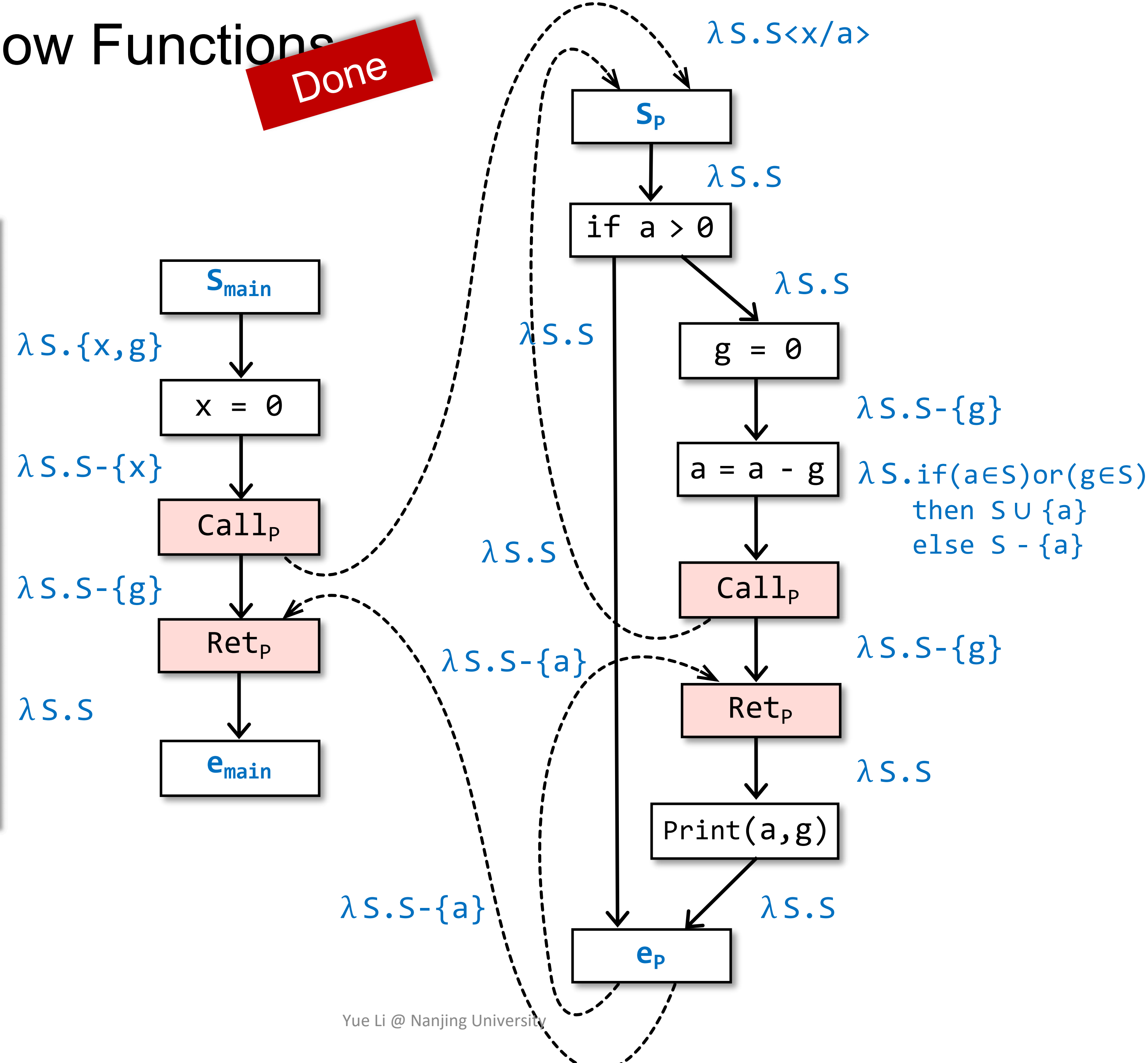
```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



Design Flow Functions

Done

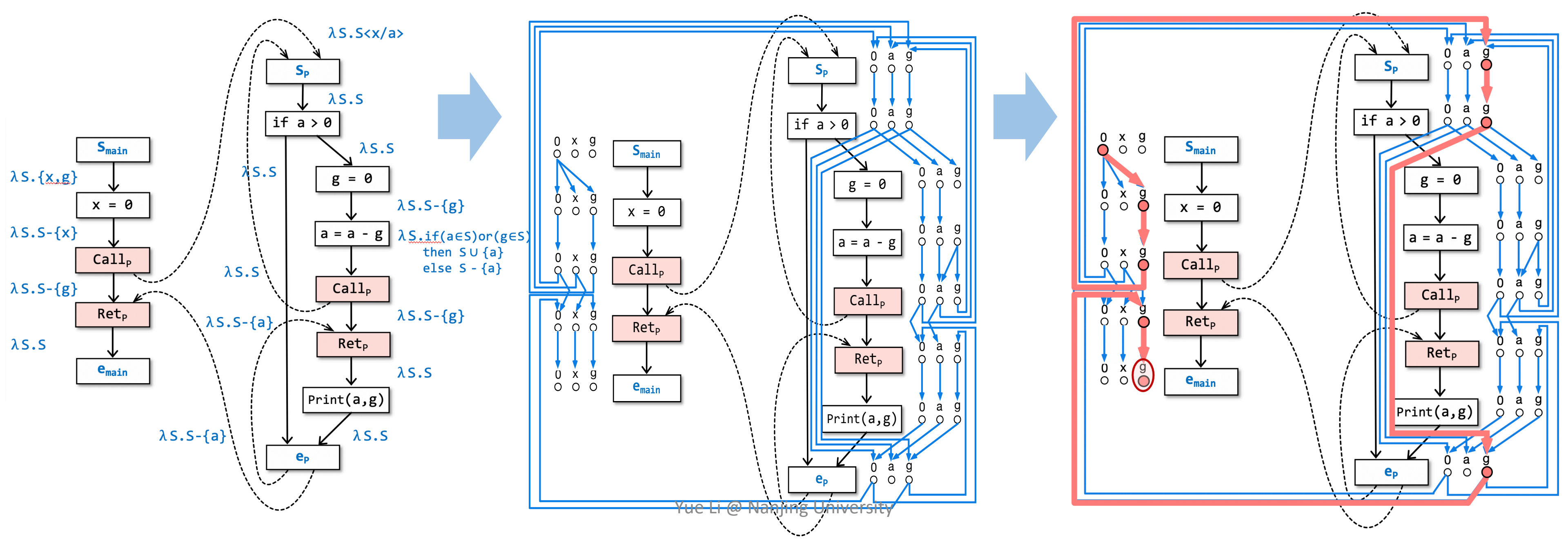
```
int g;
main(){
  int x;
  x = 0;
  P(x);
}
P(int a){
  if(a > 0){
    g = 0;
    a = a - g;
    P(a);
    Print(a,g);
  }
}
```



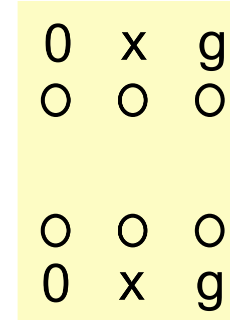
Overview of IFDS

Given a program P, and a dataflow-analysis problem Q

- Build a supergraph G* for P and
  define flow functions for edges in G* based on Q
- Build exploded supergraph $G^{\#}$ for P by transforming flow functions to representation relations (graphs)
- Q can be solved as graph reachability problems (find out MRP solutions) via applying Tabulation algorithm on $G^{\#}$

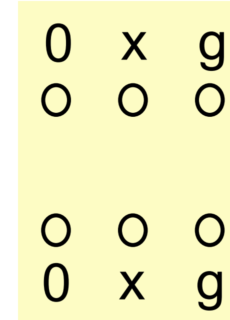


Build Exploded Supergraph



- Build exploded supergraph $G^{\#}$ for a program by transforming flow functions to representation relations (graphs)
- Each flow function can be represented as a graph with 2(D+1) nodes (at most $(D+1)^2$ edges), where D is a finite set of dataflow facts

Build Exploded Supergraph



- Build exploded supergraph $G^{\#}$ for a program by transforming flow functions to representation relations (graphs)
- Each flow function can be represented as a graph with $2(D+1)$ nodes (at most $(D+1)^2$ edges), where D is a finite set of dataflow facts

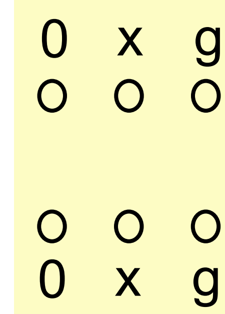
The representation relation of flow function f, $R_f \subseteq (D \cup 0) \times (D \cup 0)$ is a binary relation (or graph) defined as follows:

$$R_f = \{\ (0,0)\ \} \qquad \text{Edge: } 0 \longrightarrow 0$$

$$\cup\ \{\ (0,y) \mid y \in f(\emptyset)\ \} \qquad \text{Edge: } 0 \longrightarrow d_1$$

$$\cup\ \{\ (x,y) \mid y \notin f(\emptyset) \text{ and } y \in f(\{x\})\ \} \qquad \text{Edge: } d_1 \longrightarrow d_2$$

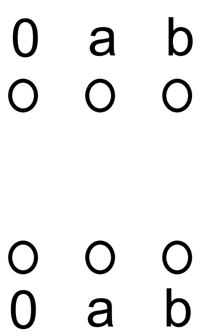
Build Exploded Supergraph



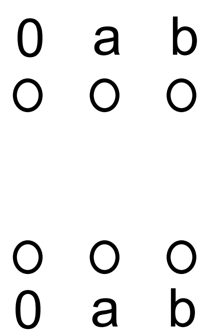
- Build exploded supergraph $G^{\#}$ for a program by transforming flow functions to representation relations (graphs)
- Each flow function can be represented as a graph with 2(D+1) nodes (at most $(D+1)^2$ edges), where D is a finite set of dataflow facts

> The representation relation of flow function f, $R_f \subseteq (D \cup 0) \times (D \cup 0)$ is a binary relation (or graph) defined as follows:
>
> $R_f = \{ (0,0) \}$ Edge: $0 \longrightarrow 0$
>
> $\cup \{ (0,y) \mid y \in f(\emptyset) \}$ Edge: $0 \longrightarrow d_1$
>
> $\cup \{ (x,y) \mid y \notin f(\emptyset) \text{ and } y \in f(\{x\}) \}$ Edge: $d_1 \longrightarrow d_2$

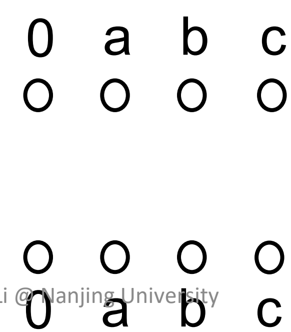
λS.S



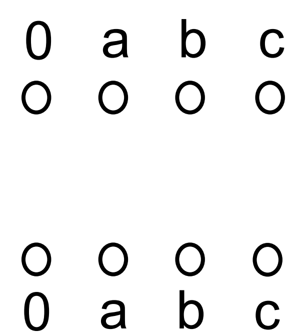
λS.{a}



λS.(S-{a})∪{b}



λS.if a∈S
then S∪{b}
else S - {b}



Build Exploded Supergraph

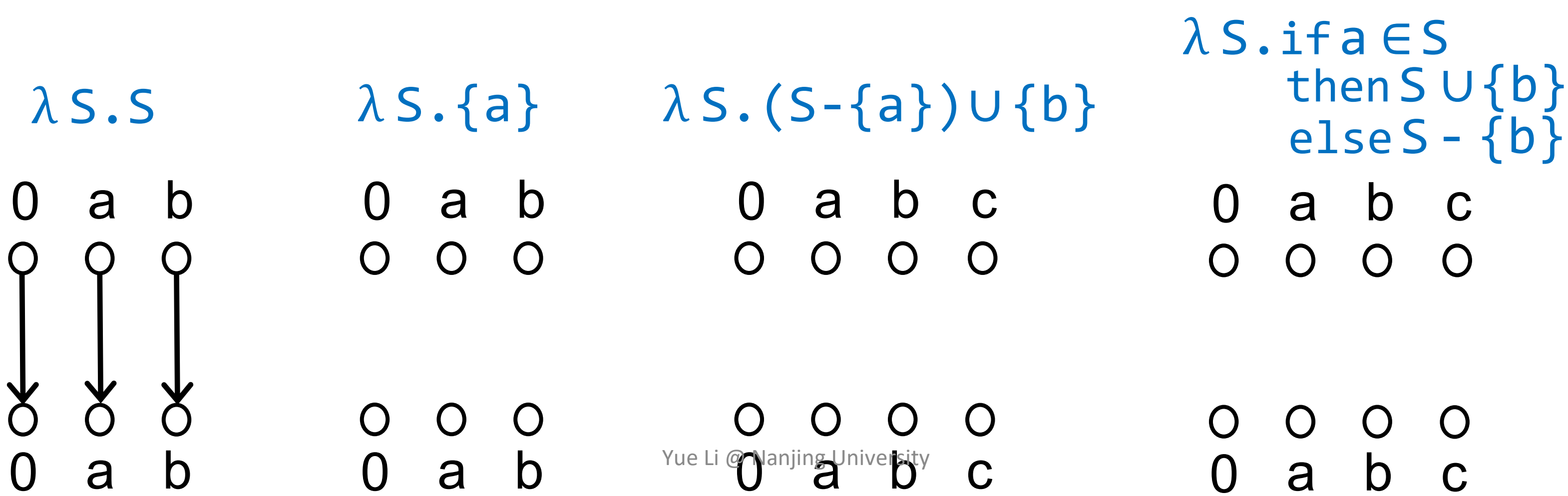
- Build exploded supergraph $G^\#$ for a program by transforming flow functions to representation relations (graphs)
- Each flow function can be represented as a graph with 2(D+1) nodes (at most $(D+1)^2$ edges), where D is a finite set of dataflow facts

The representation relation of flow function f, $R_f \subseteq (D \cup 0) \times (D \cup 0)$ is a binary relation (or graph) defined as follows:

$R_f = \{ (0,0) \}$ Edge: $0 \longrightarrow 0$

$\cup \{ (0,y) \mid y \in f(\emptyset) \}$ Edge: $0 \longrightarrow d_1$

$\cup \{ (x,y) \mid y \notin f(\emptyset) \text{ and } y \in f(\{x\}) \}$ Edge: $d_1 \longrightarrow d_2$



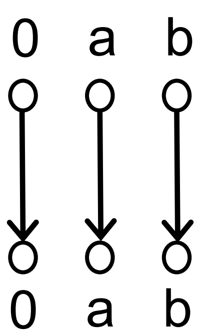
Build Exploded Supergraph

- Build exploded supergraph $G^\#$ for a program by transforming flow functions to representation relations (graphs)
- Each flow function can be represented as a graph with 2(D+1) nodes (at most $(D+1)^2$ edges), where D is a finite set of dataflow facts

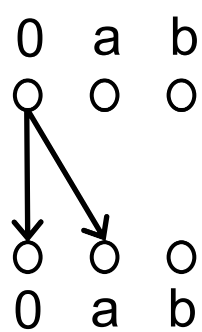
The representation relation of flow function f, $R_f \subseteq (D \cup 0) \times (D \cup 0)$ is a binary relation (or graph) defined as follows:

$R_f = \{ (0,0) \}$	Edge: $0 \longrightarrow 0$
$\cup \; \{ (0,y) \mid y \in f(\emptyset) \}$	Edge: $0 \longrightarrow d_1$
$\cup \; \{ (x,y) \mid y \notin f(\emptyset) \text{ and } y \in f(\{x\}) \}$	Edge: $d_1 \longrightarrow d_2$

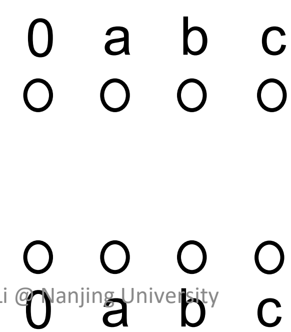
`λS.S`



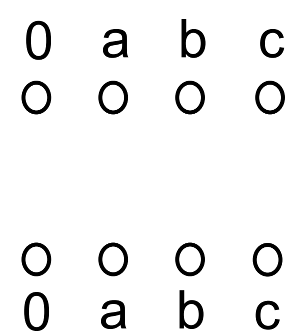
`λS.{a}`



`λS.(S-{a})∪{b}`




```
λS.if a∈S
    then S∪{b}
    else S-{b}
```



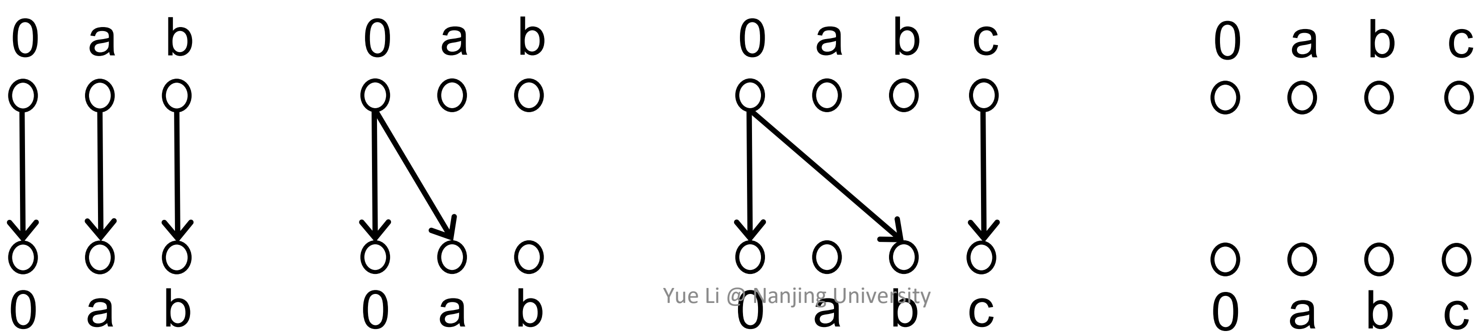
Build Exploded Supergraph

- Build exploded supergraph $G^{\#}$ for a program by transforming flow functions to representation relations (graphs)
- Each flow function can be represented as a graph with $2(D+1)$ nodes (at most $(D+1)^2$ edges), where D is a finite set of dataflow facts

The representation relation of flow function f, $R_f \subseteq (D \cup 0) \times (D \cup 0)$ is a binary relation (or graph) defined as follows:

$R_f = \{\ (0,0)\ \}$	Edge: $0 \longrightarrow 0$
$\cup\ \{\ (0,y) \mid y \in f(\emptyset)\ \}$	Edge: $0 \longrightarrow d_1$
$\cup\ \{\ (x,y) \mid y \notin f(\emptyset) \text{ and } y \in f(\{x\})\ \}$	Edge: $d_1 \longrightarrow d_2$

`λS.S` `λS.{a}` `λS.(S-{a})∪{b}` `λS.if a∈S then S∪{b} else S-{b}`



Build Exploded Supergraph

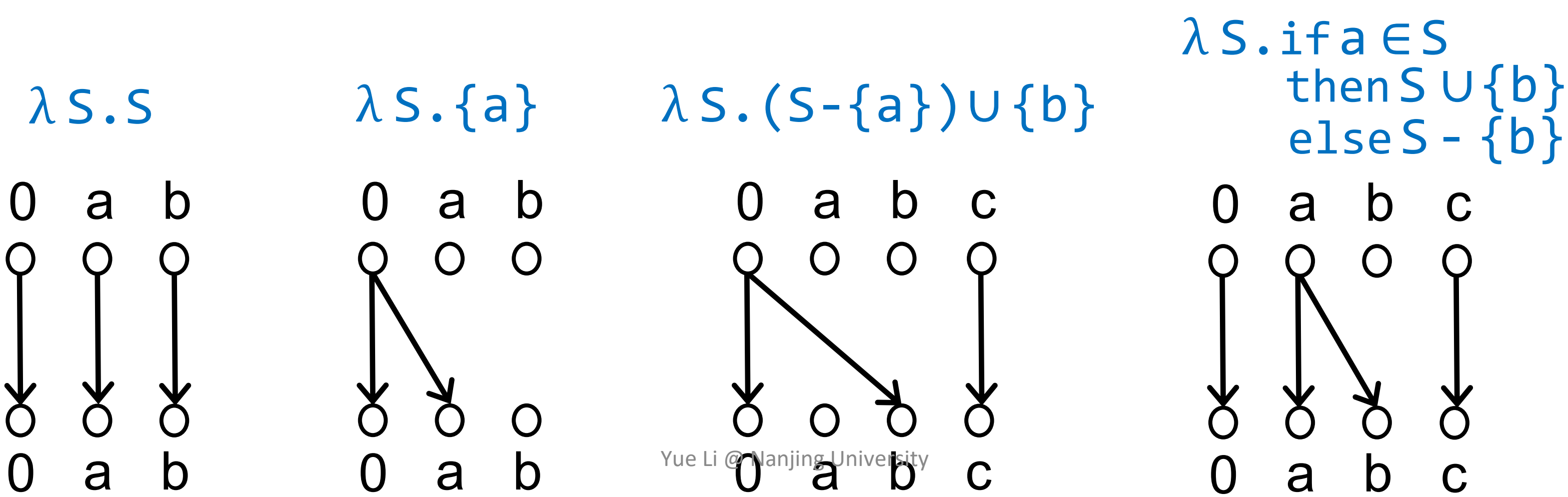
- Build exploded supergraph $G^{\#}$ for a program by transforming flow functions to representation relations (graphs)
- Each flow function can be represented as a graph with 2(D+1) nodes (at most $(D+1)^2$ edges), where D is a finite set of dataflow facts

The representation relation of flow function f, $R_f \subseteq (D \cup 0) \times (D \cup 0)$ is a binary relation (or graph) defined as follows:

$R_f = \{ (0,0) \}$ Edge: $0 \longrightarrow 0$

$\cup \{ (0,y) \mid y \in f(\emptyset) \}$ Edge: $0 \longrightarrow d_1$

$\cup \{ (x,y) \mid y \notin f(\emptyset) \text{ and } y \in f(\{x\}) \}$ Edge: $d_1 \longrightarrow d_2$



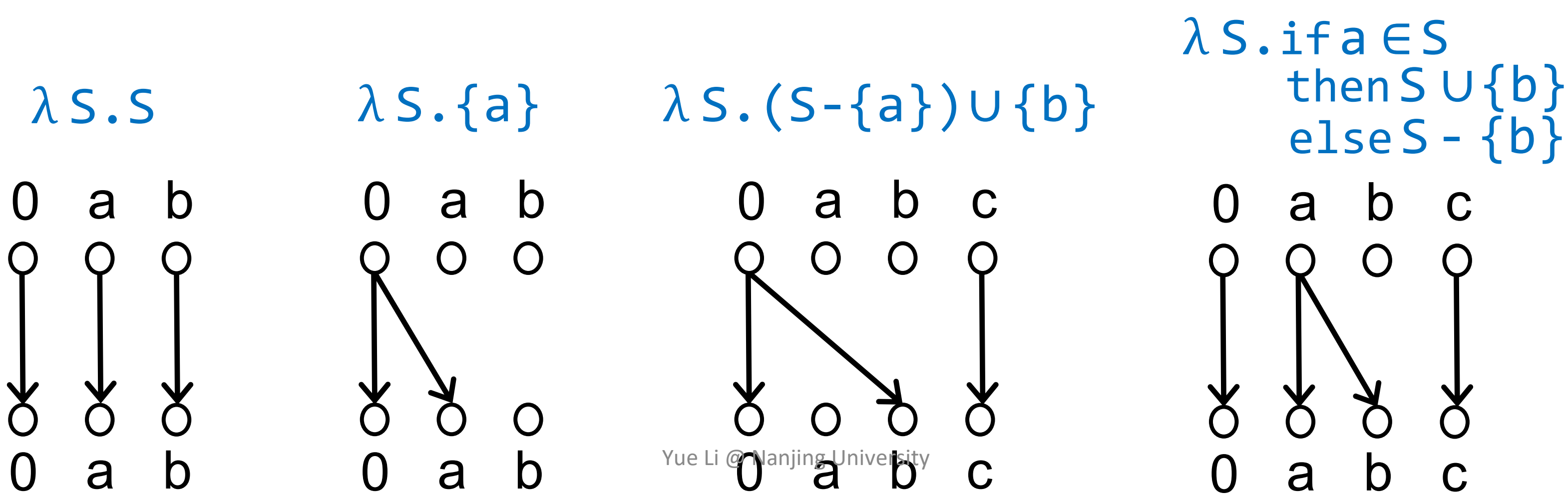
Build Exploded Supergraph

Exploded Supergraph $G^\#$:

Each node n in supergraph G^* is "exploded" into D+1 nodes in $G^\#$, and each edge $n_1 \longrightarrow n_2$ in G^* is "exploded" into the representation relation of the flow function associated with $n_1 \longrightarrow n_2$ in $G^\#$

The representation relation of flow function f, $R_f \subseteq (D \cup 0) \times (D \cup 0)$ is a binary relation (or graph) defined as follows:

$R_f = \{\ (0,0)\ \}$	Edge: $0 \longrightarrow 0$
$\cup\ \{\ (0,y) \mid y \in f(\emptyset)\ \}$	Edge: $0 \longrightarrow d_1$
$\cup\ \{\ (x,y) \mid y \notin f(\emptyset) \text{ and } y \in f(\{x\})\ \}$	Edge: $d_1 \longrightarrow d_2$



Build Exploded Supergraph

Exploded Supergraph $G^\#$:

Each node n in supergraph G* is "exploded" into D+1 nodes in $G^\#$, and each edge $n_1 \longrightarrow n_2$ in G* is "exploded" into the representation relation of the flow function associated with $n_1 \longrightarrow n_2$ in $G^\#$

The representation relation of flow function f, $R_f \subseteq (D \cup 0) \times (D \cup 0)$ is a binary relation (or graph) defined as follows:

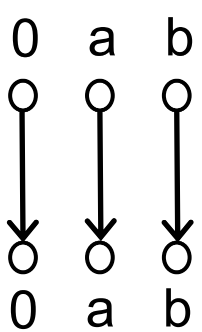
$R_f = \{ (0,0) \}$ Edge: $0 \longrightarrow 0$

$\cup \{ (0,y) \mid y \in f(\emptyset) \}$ Edge: $0 \longrightarrow d_1$

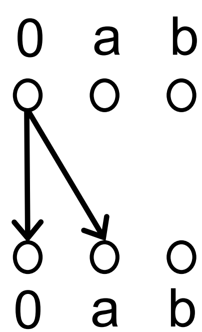
$\cup \{ (x,y) \mid y \notin f(\emptyset$ … $\})\}$ Edge: $d_1 \longrightarrow d_2$

Why we need $0 \longrightarrow 0$ edges?

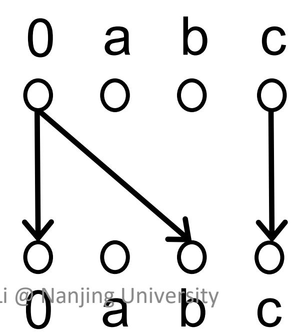
λS.S



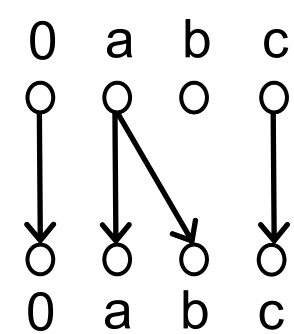
λS.{a}



λS.(S-{a})∪{b}



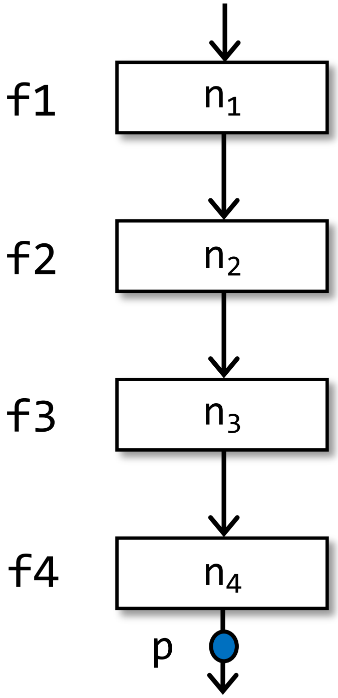
λS.if a∈S
then S∪{b}
else S-{b}



Why We Need Edge $0 \longrightarrow 0$?

In traditional data flow analysis, to see whether data fact a holds at program point p, we check if a is in OUT[n_4] after the analysis finishes

$$\texttt{OUT}[n_4] = \texttt{f4} \circ \texttt{f3} \circ \texttt{f2} \circ \texttt{f1}(\texttt{IN}[\texttt{n1}])$$

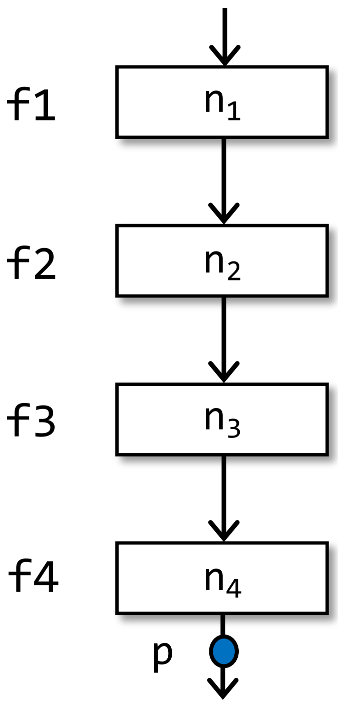


Why We Need Edge $0 \longrightarrow 0$?

In traditional data flow analysis, to see whether data fact a holds at program point p, we check if a is in OUT[n_4] after the analysis finishes

$$\texttt{OUT}[n_4] = f4 \circ f3 \circ f2 \circ f1(\texttt{IN}[n1])$$

Data facts are propagated via the composition of flow functions. In this case, the "reachability" is directly retrieved from the final result in OUT[n_4] .

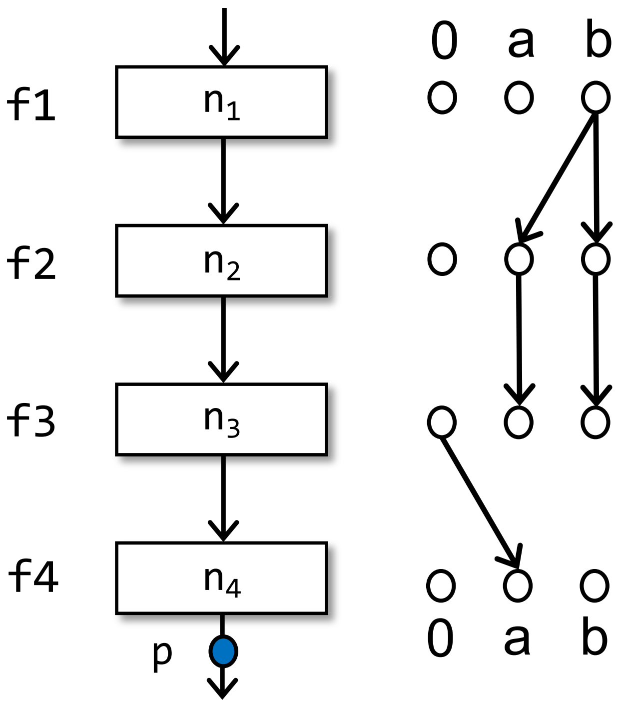


Why We Need Edge 0 ⟶ 0?

In traditional data flow analysis, to see whether data fact a holds at program point p, we check if a is in OUT[n_4] after the analysis finishes

$$OUT[n_4] = f4 \circ f3 \circ f2 \circ f1(IN[n1])$$

Data facts are propagated via the composition of flow functions. In this case, the "reachability" is directly retrieved from the final result in OUT[n_4] .



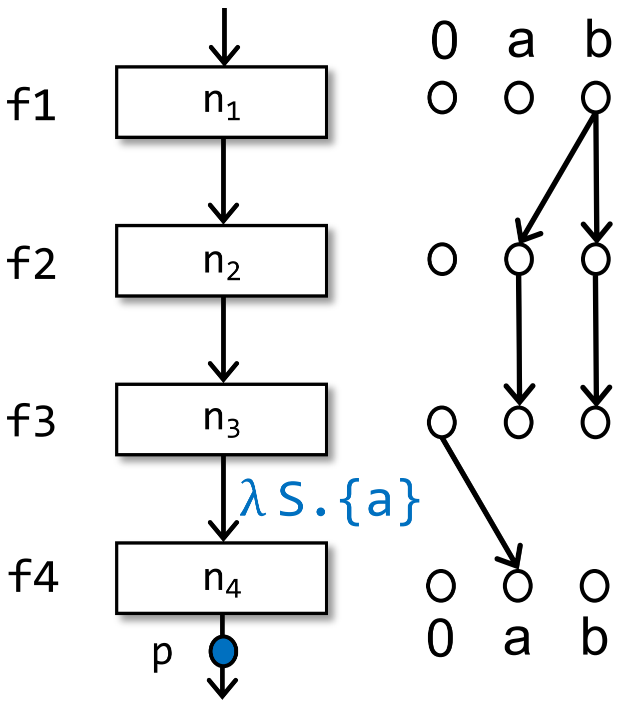
For the same case, in IFDS, whether data fact a holds at p depends on if there is a path from <s_{main}, 0> to <n_4,a>, and the "reachability" is retrieved by connecting the edges (finding out a path) on the "pasted" representation relations

Why We Need Edge 0 ⟶ 0?

In traditional data flow analysis, to see whether data fact a holds at program point p, we check if a is in OUT[n_4] after the analysis finishes

$$OUT[n_4] = f4 \circ f3 \circ f2 \circ f1(IN[n1])$$

Data facts are propagated via the composition of flow functions. In this case, the "reachability" is directly retrieved from the final result in OUT[n_4] .



For the same case, in IFDS, whether data fact a holds at p depends on if there is a path from <s_{main}, 0> to <n_4,a>, and the "reachability" is retrieved by connecting the edges (finding out a path) on the "pasted" representation relations

λS.{a} says a holds at p regardless of input S; however, *without edge 0⟶0*,

the representation relation for each edge cannot be connected or "pasted" together, like flow functions cannot be composed together in traditional data flow analysis.

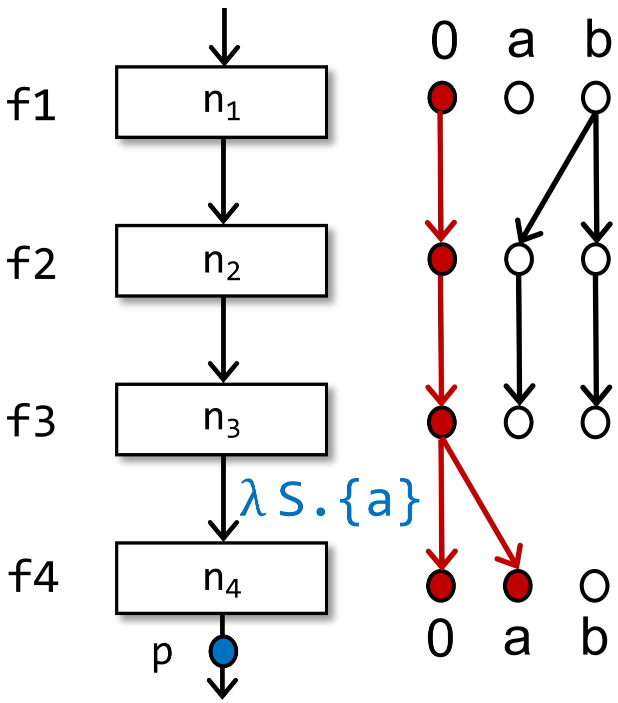
Thus IFDS cannot produce correct solutions via such disconnected representation relations.

So We Need the “Glue Edge” $0 \longrightarrow 0$!

In traditional data flow analysis, to see whether data fact a holds at program point p, we check if a is in OUT[n_4] after the analysis finishes

```
OUT[n4] = f4 ∘ f3 ∘ f2 ∘ f1(IN[n1])
```

Data facts are propagated via the composition of flow functions. In this case, the “reachability” is directly retrieved from the final result in OUT[n_4] .



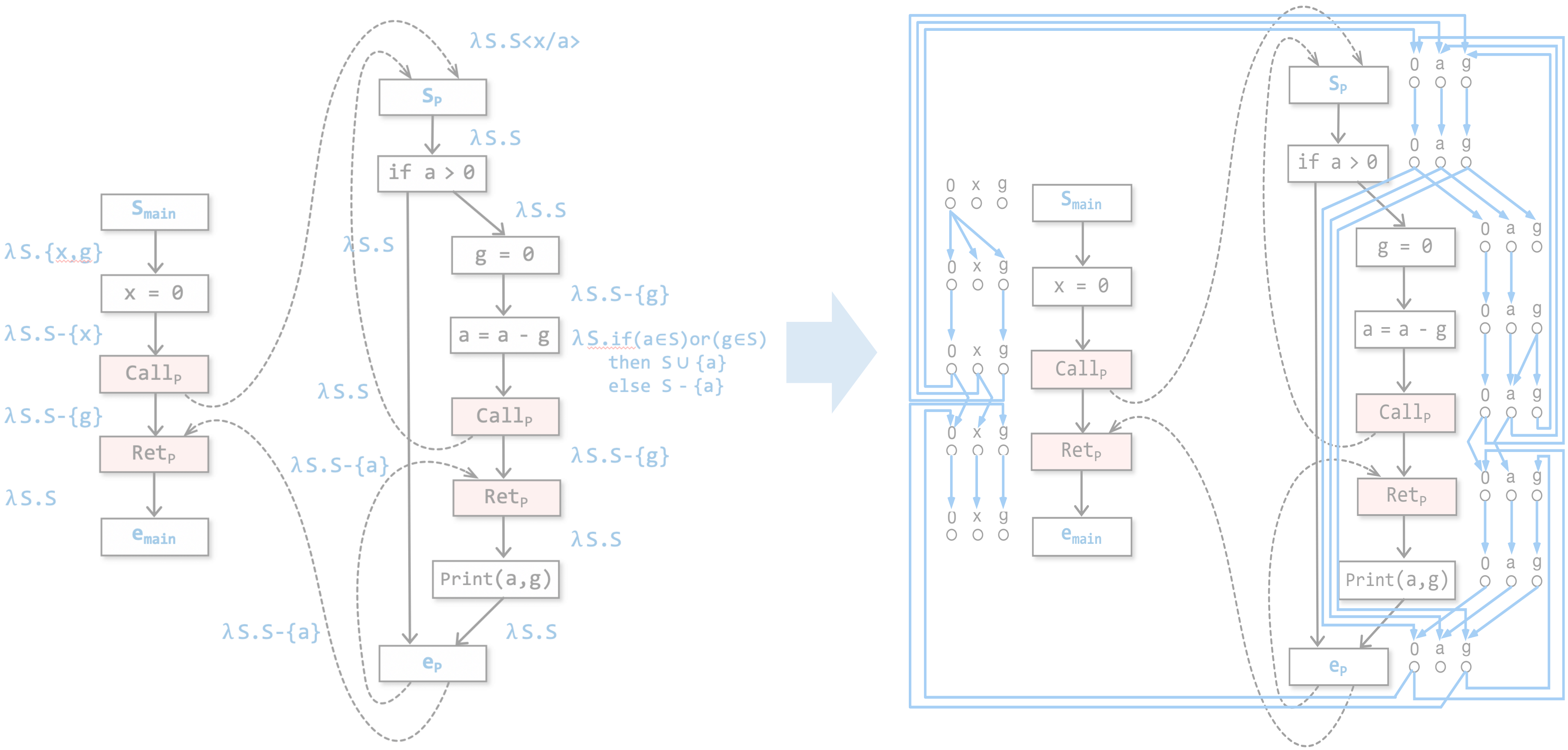
For the same case, in IFDS, whether data fact a holds at p depends on if there is a path from <s_{main}, 0> to <n_4,a>, and the “reachability” is retrieved by connecting the edges (finding out a path) on the “pasted” representation relations

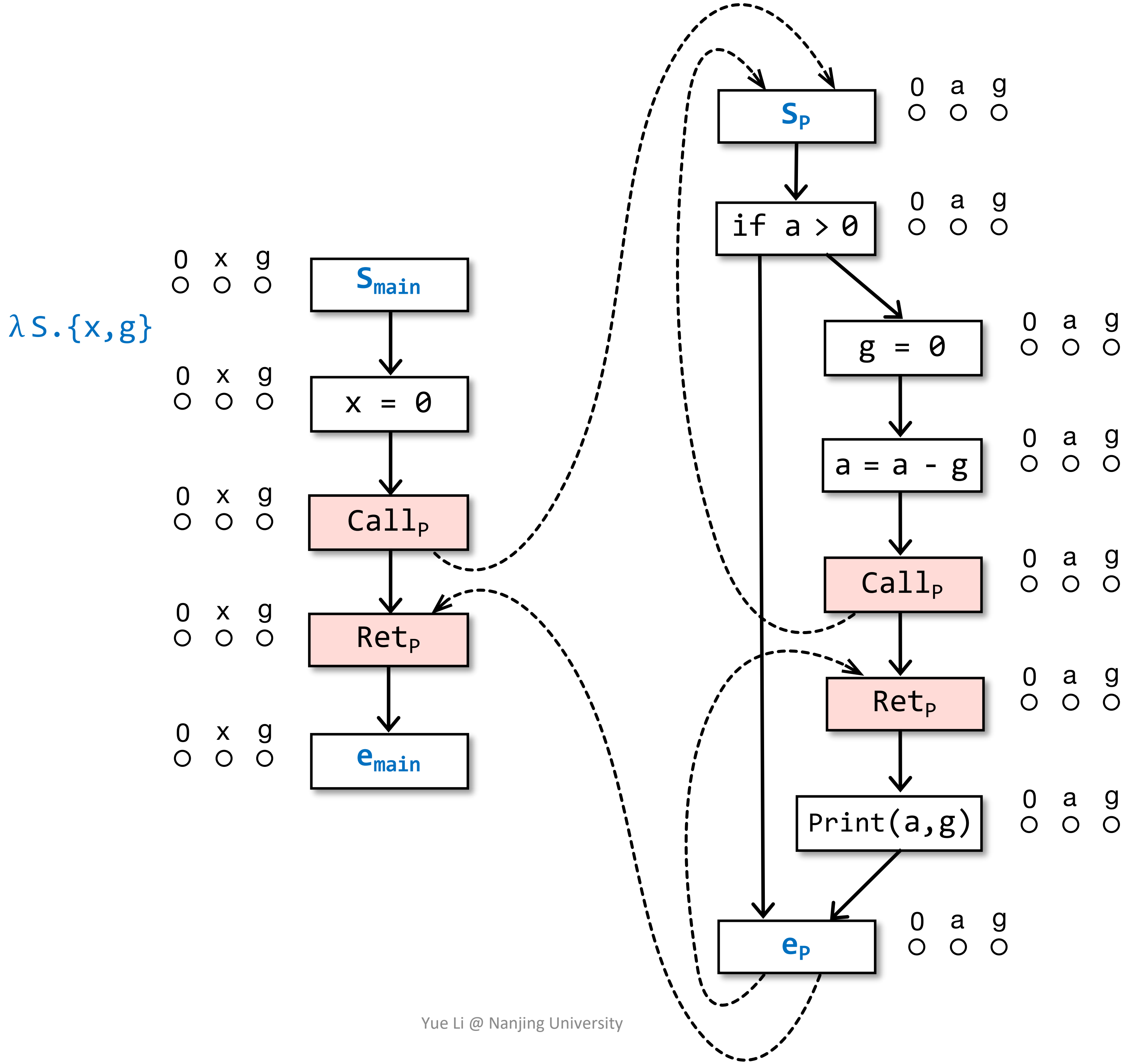
`λS.{a}` says a holds at p regardless of input S; however, *without edge 0⟶0,*

the representation relation for each edge cannot be connected or “pasted” together, like flow functions cannot be composed together in traditional data flow analysis.

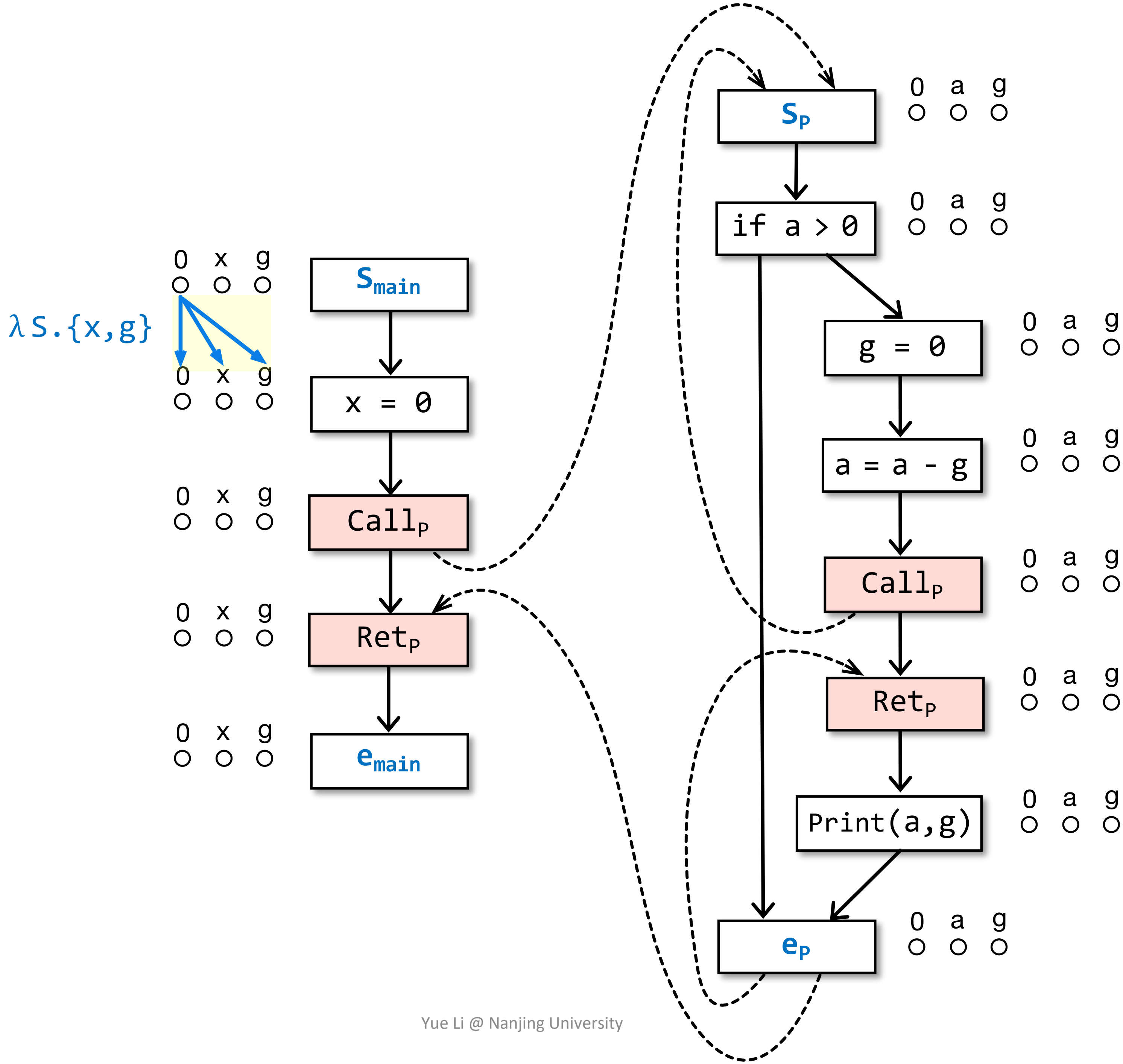
Thus IFDS cannot produce correct solutions via such disconnected representation relations.

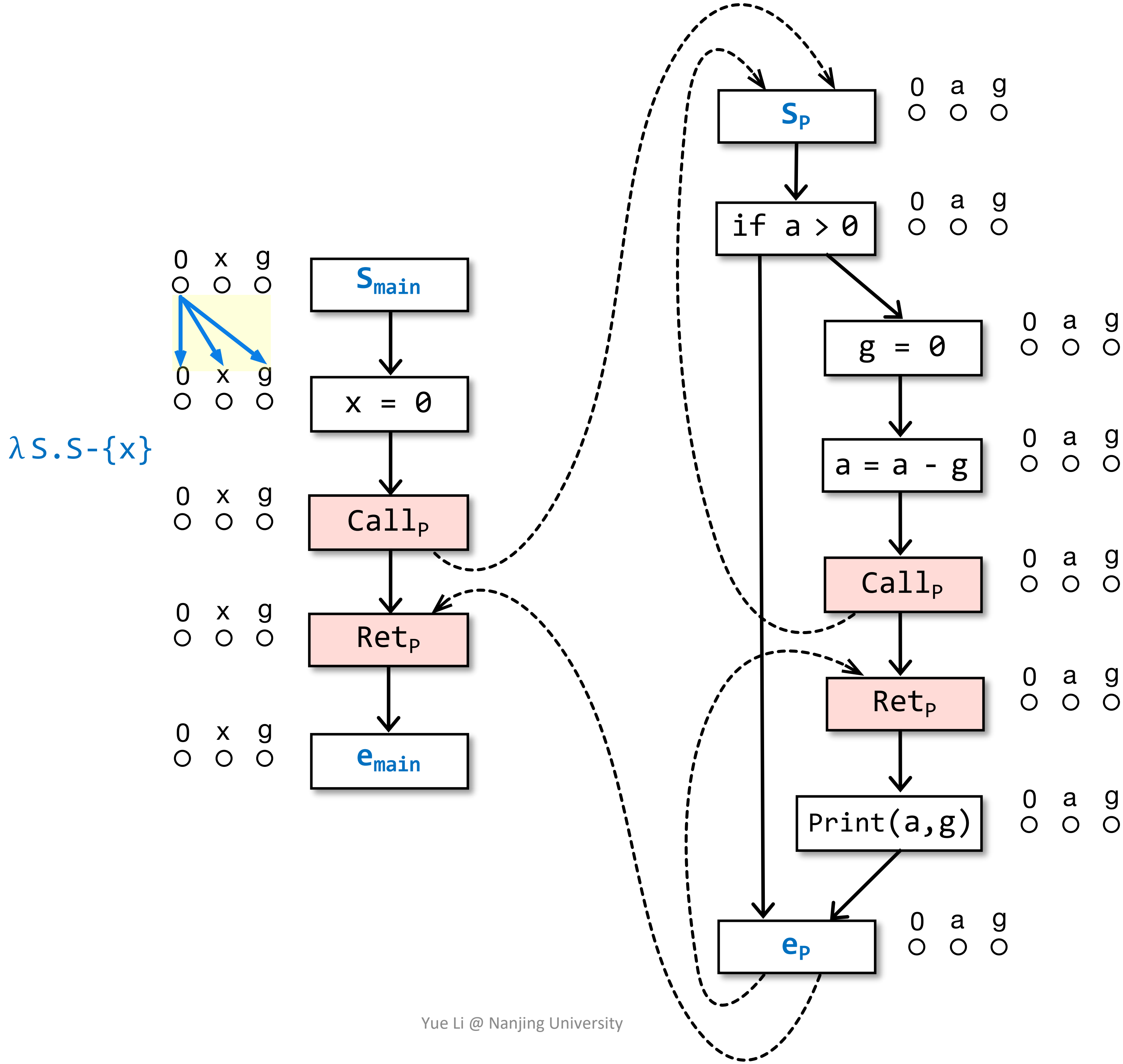
Now, let's build an exploded supergraph

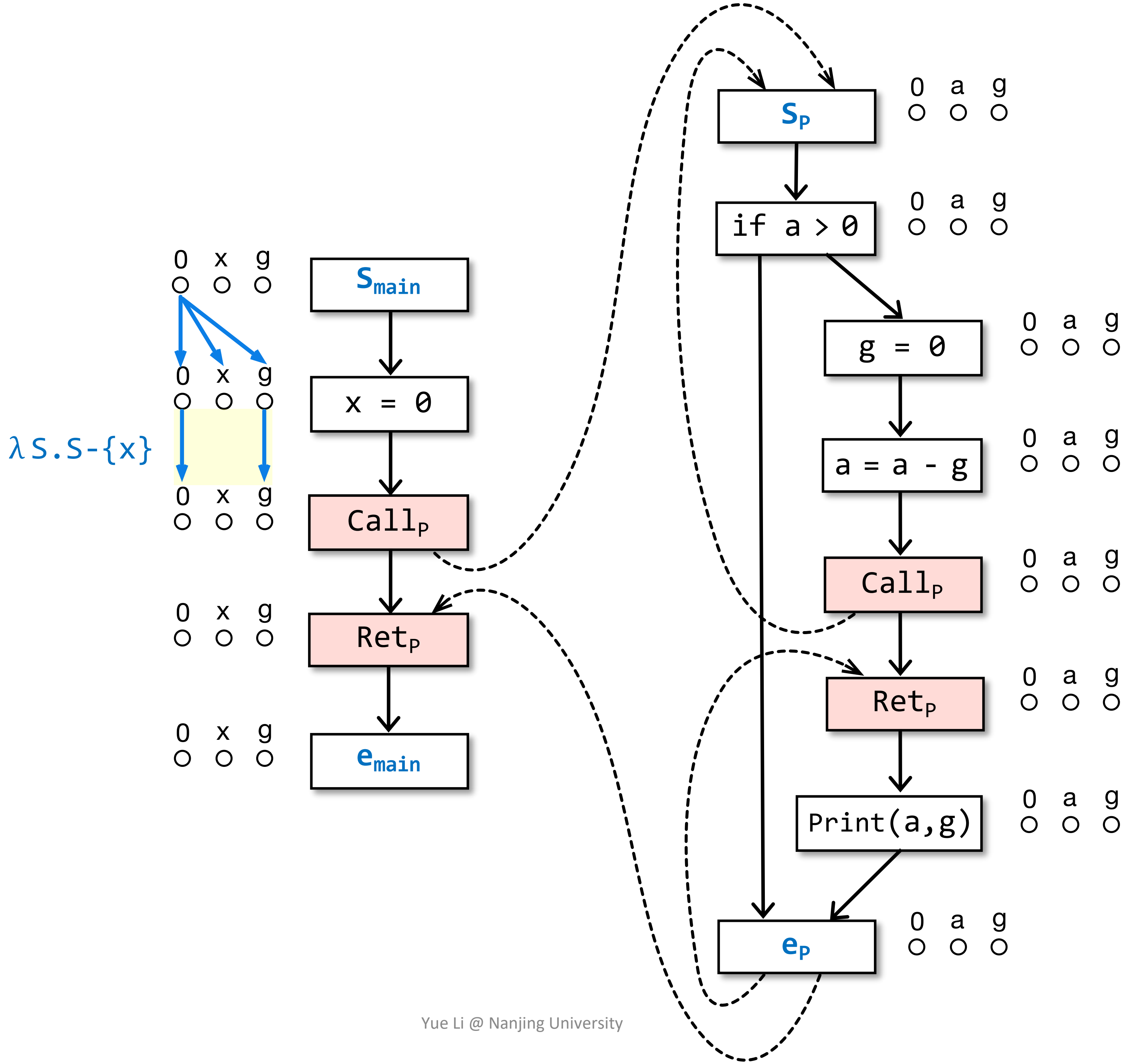




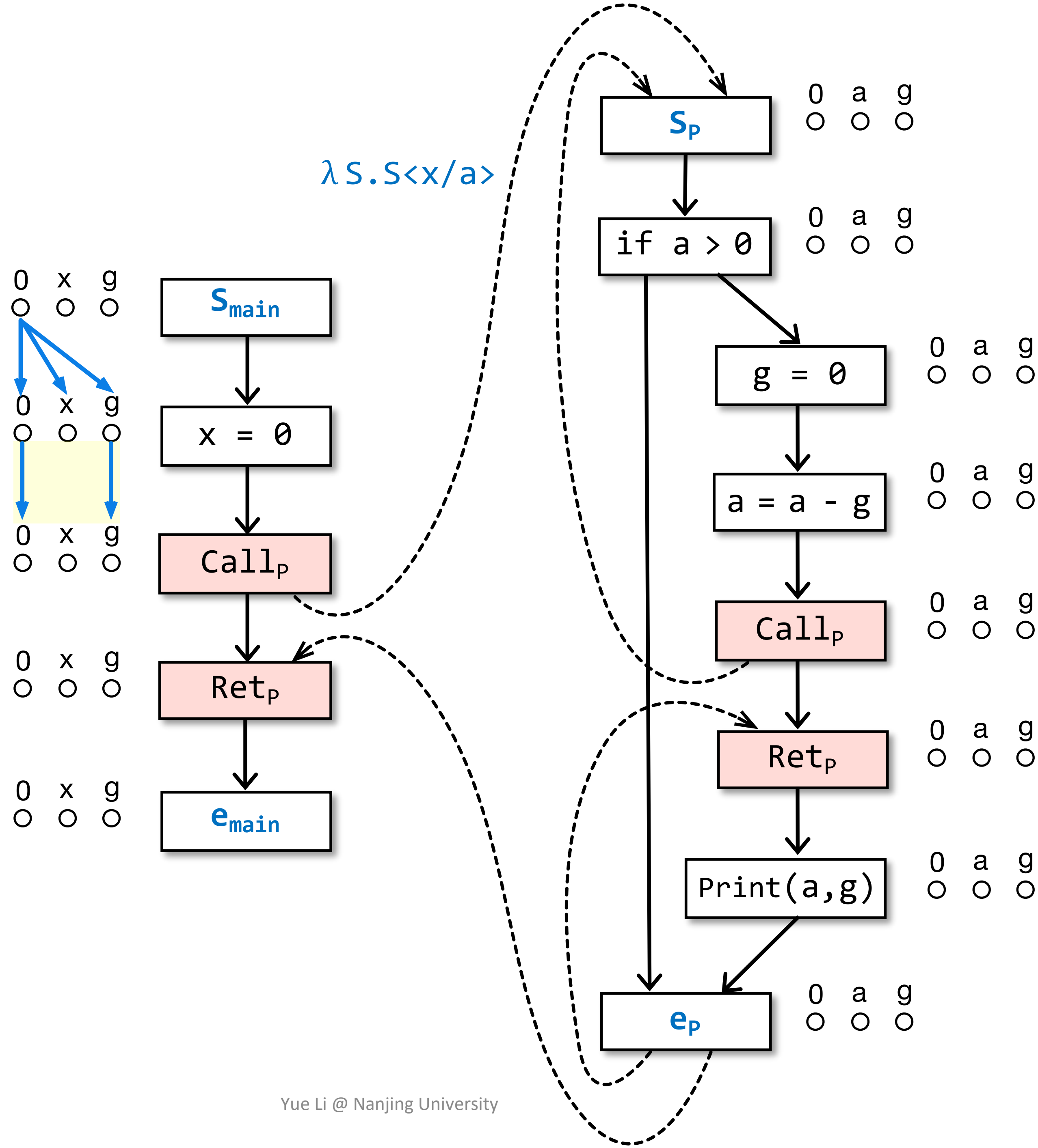
λS.{x,g}
0 x g
Smain
x = 0
CallP
RetP
emain
0 a g
SP
if a > 0
g = 0
a = a - g
CallP
RetP
Print(a,g)
eP

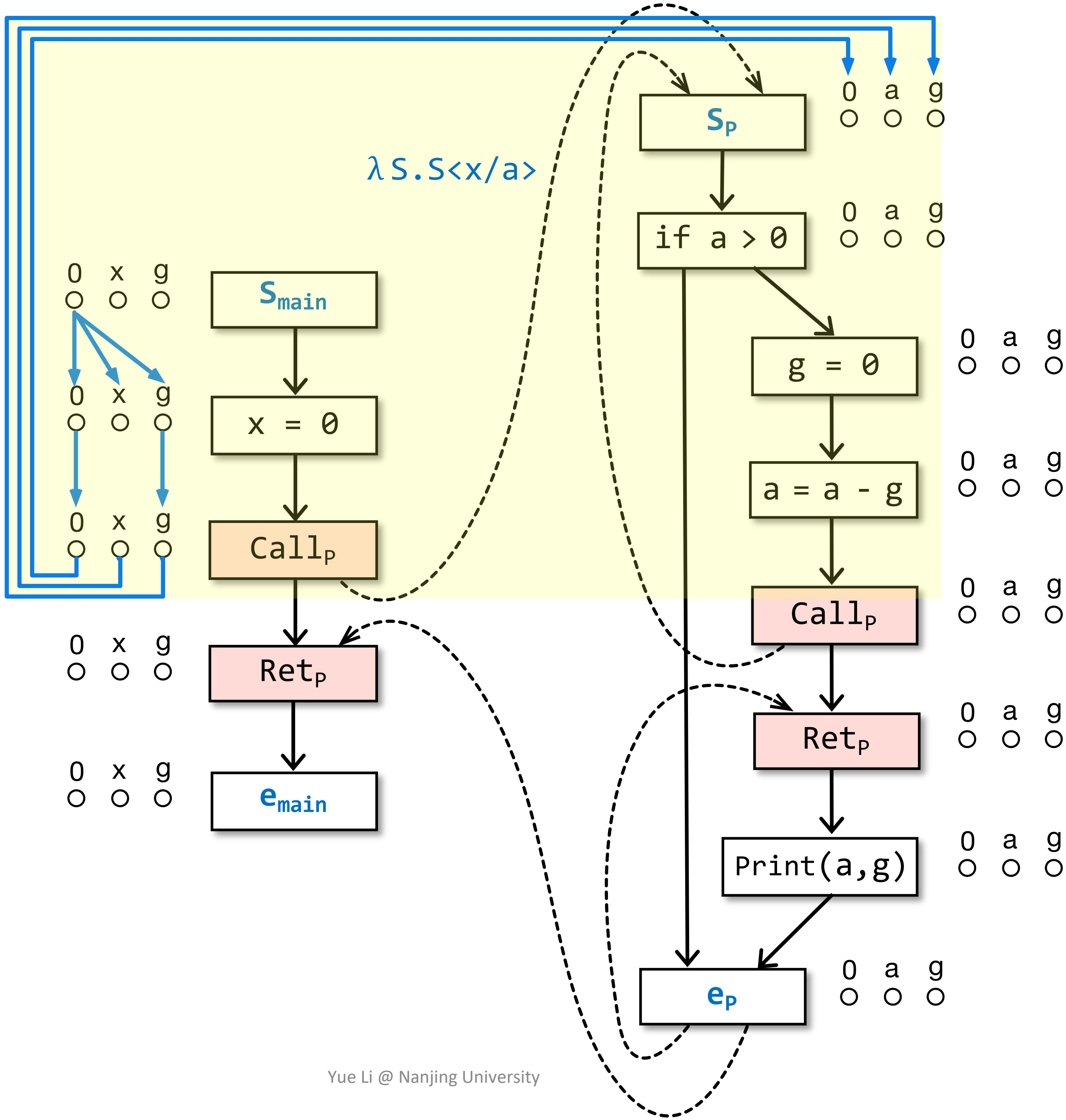

λS.{x,g}
0 x g
S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
0 a g
S_p
if a > 0
g = 0
a = a - g
Print(a,g)
e_p


λS.S-{x}
0 x g
S_main
x = 0
Call_p
Ret_p
e_main
0 a g
S_p
if a > 0
g = 0
a = a - g
Call_p
Ret_p
Print(a,g)
e_p

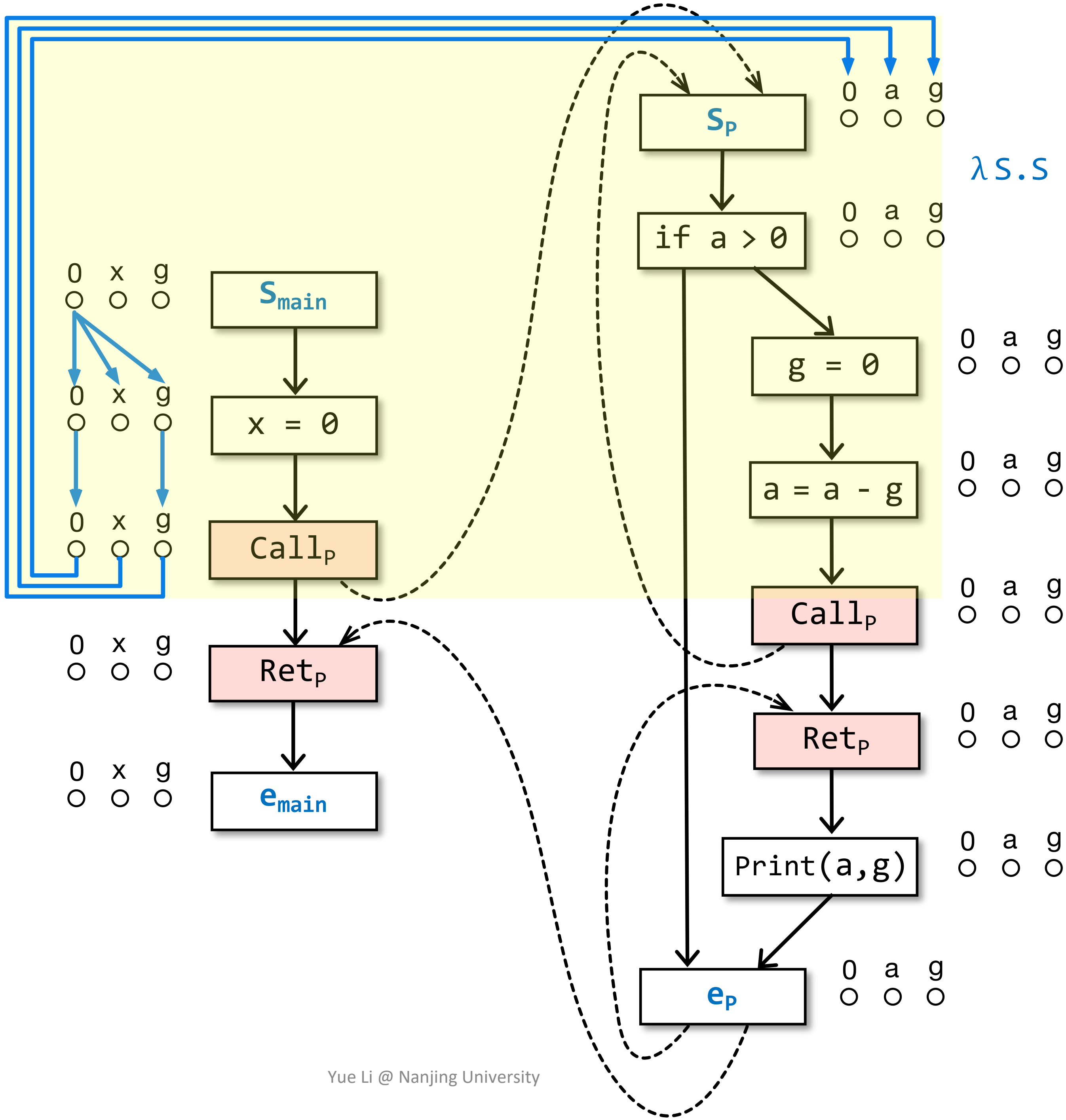


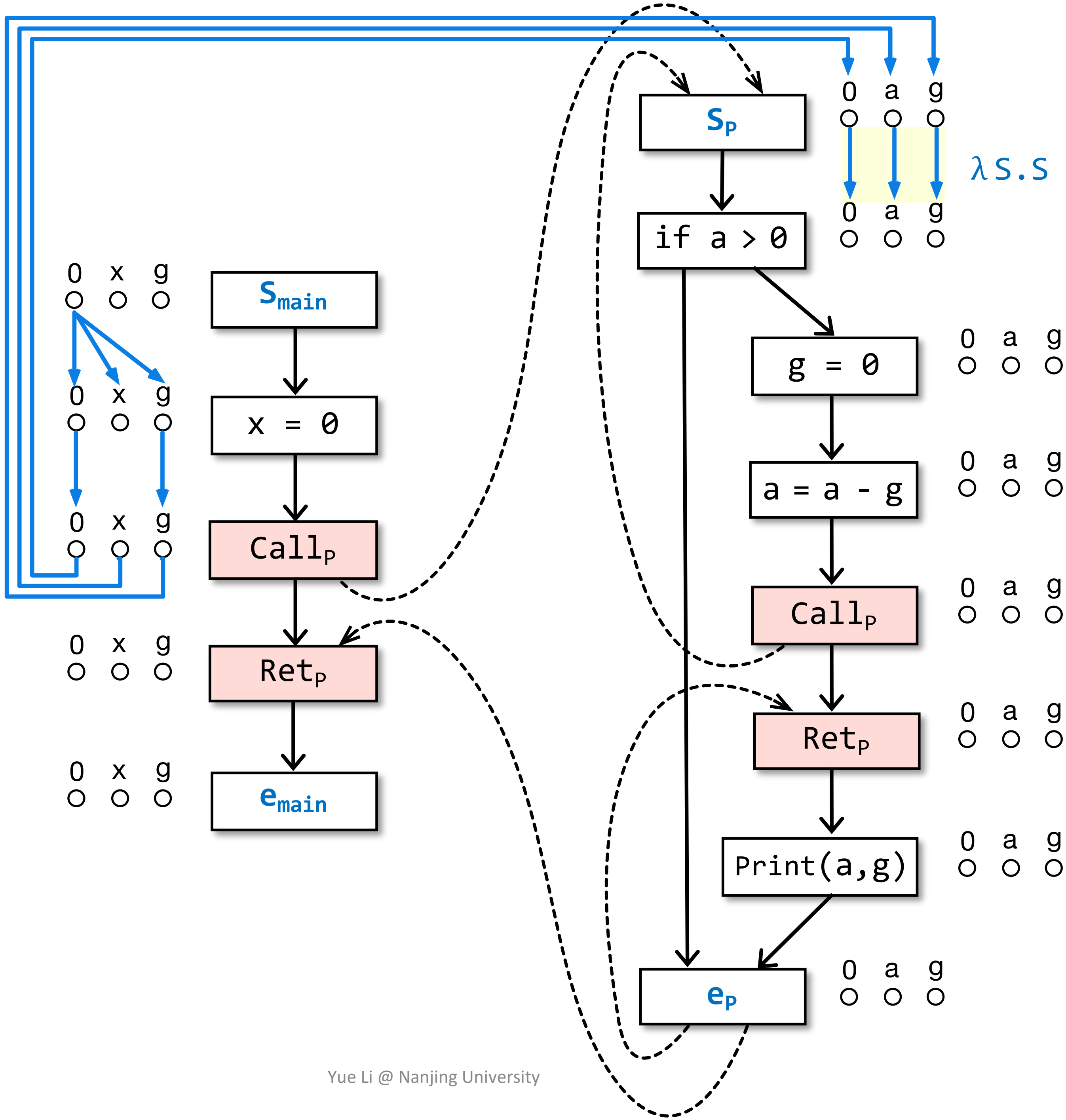
S_{main}
x = 0
$Call_P$
Ret_P
e_{main}
$\lambda S.S-\{x\}$
0 x g
S_P
if a > 0
g = 0
a = a - g
$Call_P$
Ret_P
Print(a,g)
e_P
0 a g


λS.S<x/a>
Smain
x = 0
CallP
RetP
emain
SP
if a > 0
g = 0
a = a - g
CallP
RetP
Print(a,g)
eP
0 x g
0 a g

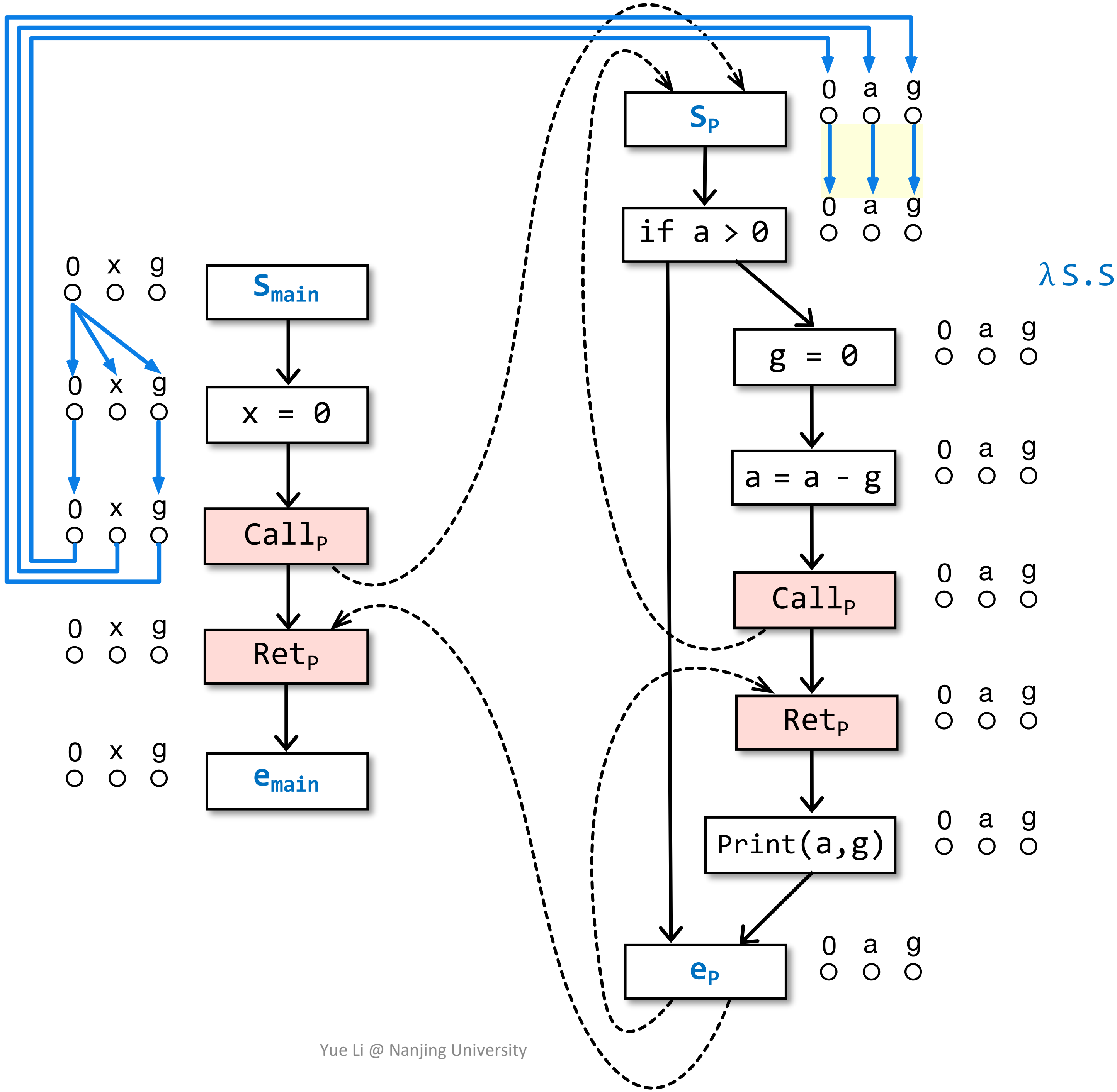


λS.S<x/a>
s_{main}
x = 0
$Call_P$
Ret_P
e_{main}
s_P
if a > 0
g = 0
a = a - g
$Call_P$
Ret_P
Print(a,g)
e_P
0 x g
0 a g

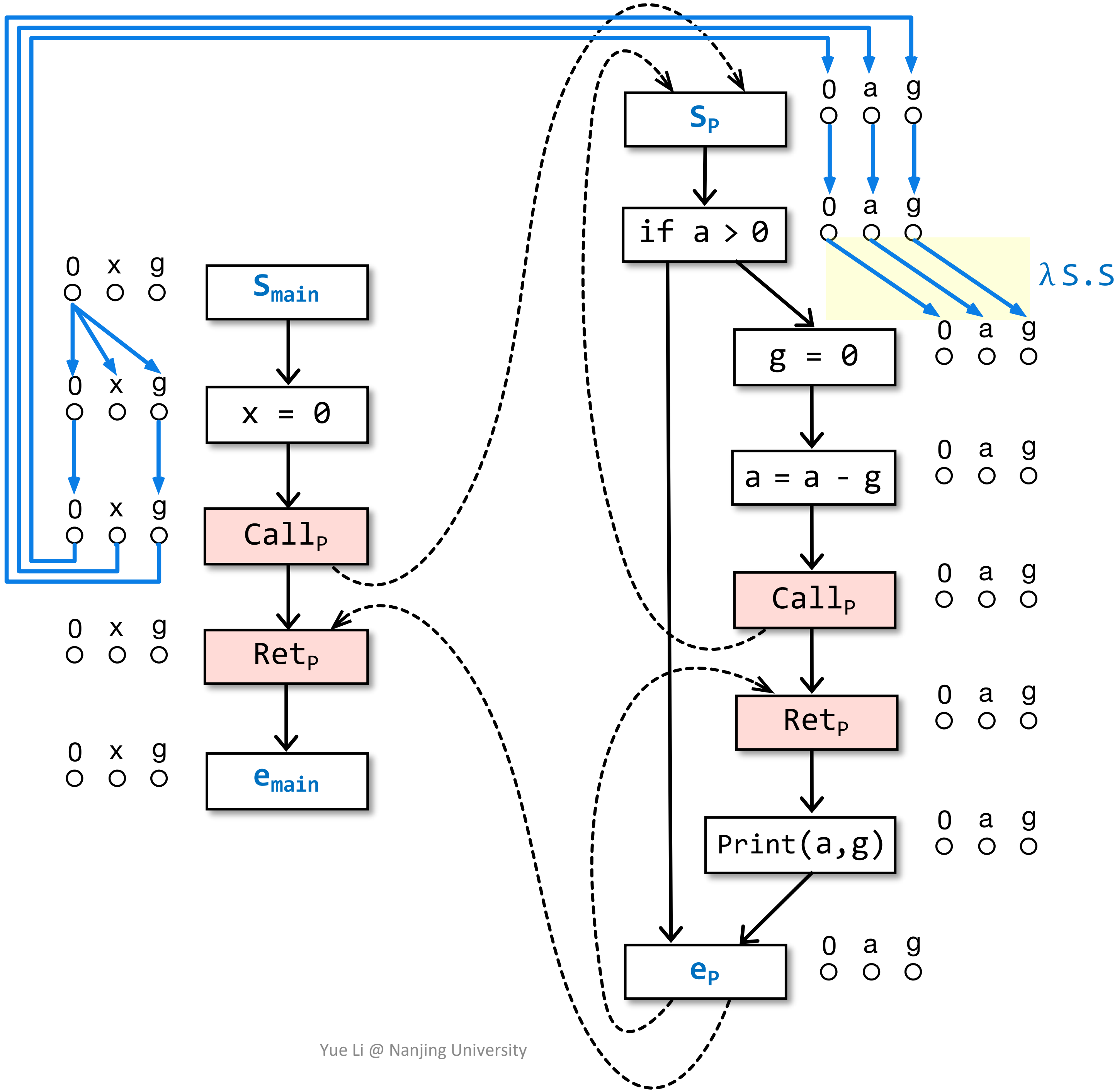

λS.S
S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
Print(a,g)
e_p
0 x g
0 a g



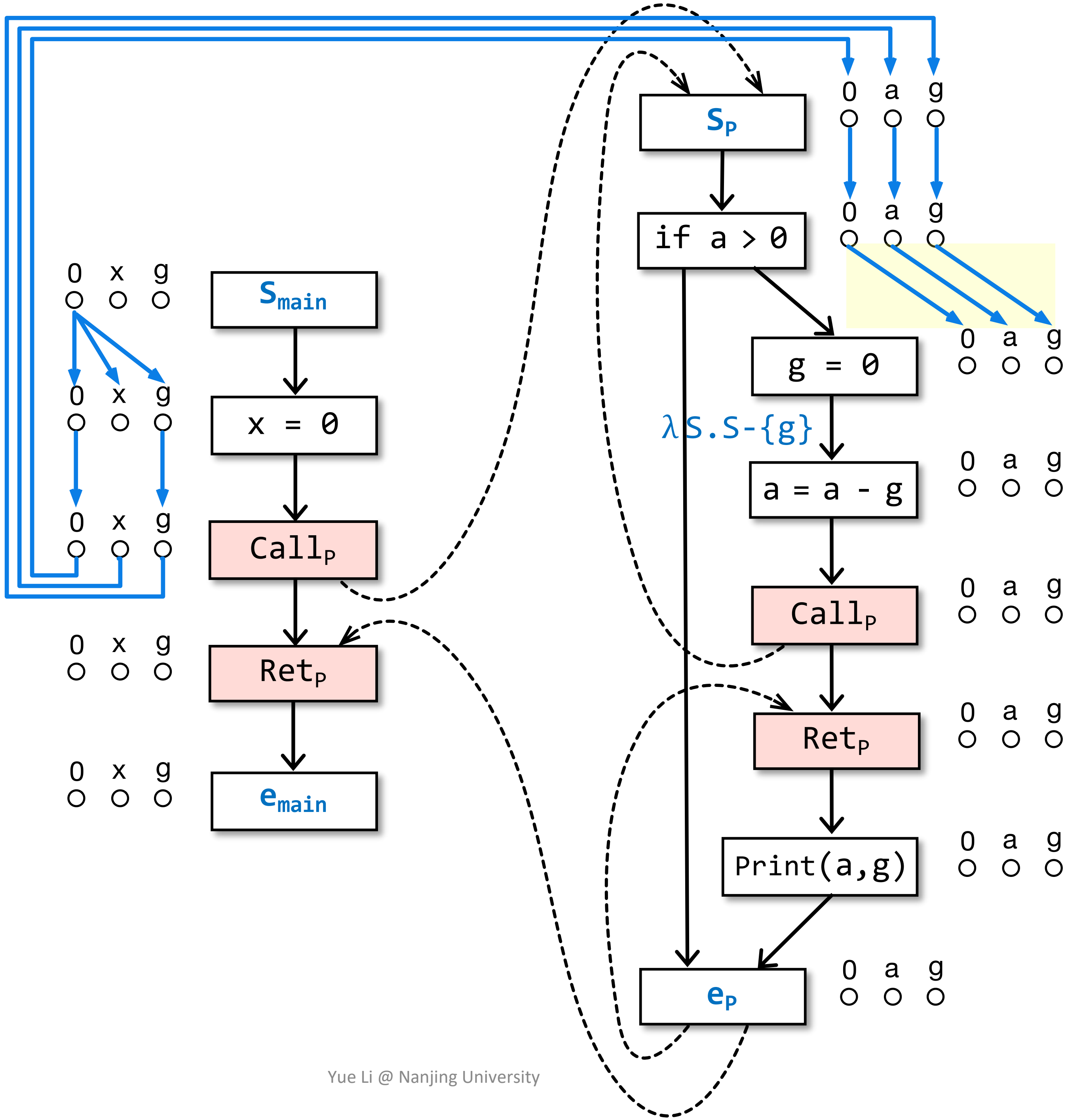
s_{main}
x = 0
$Call_p$
Ret_p
e_{main}
0 x g
s_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
0 a g
λs.s

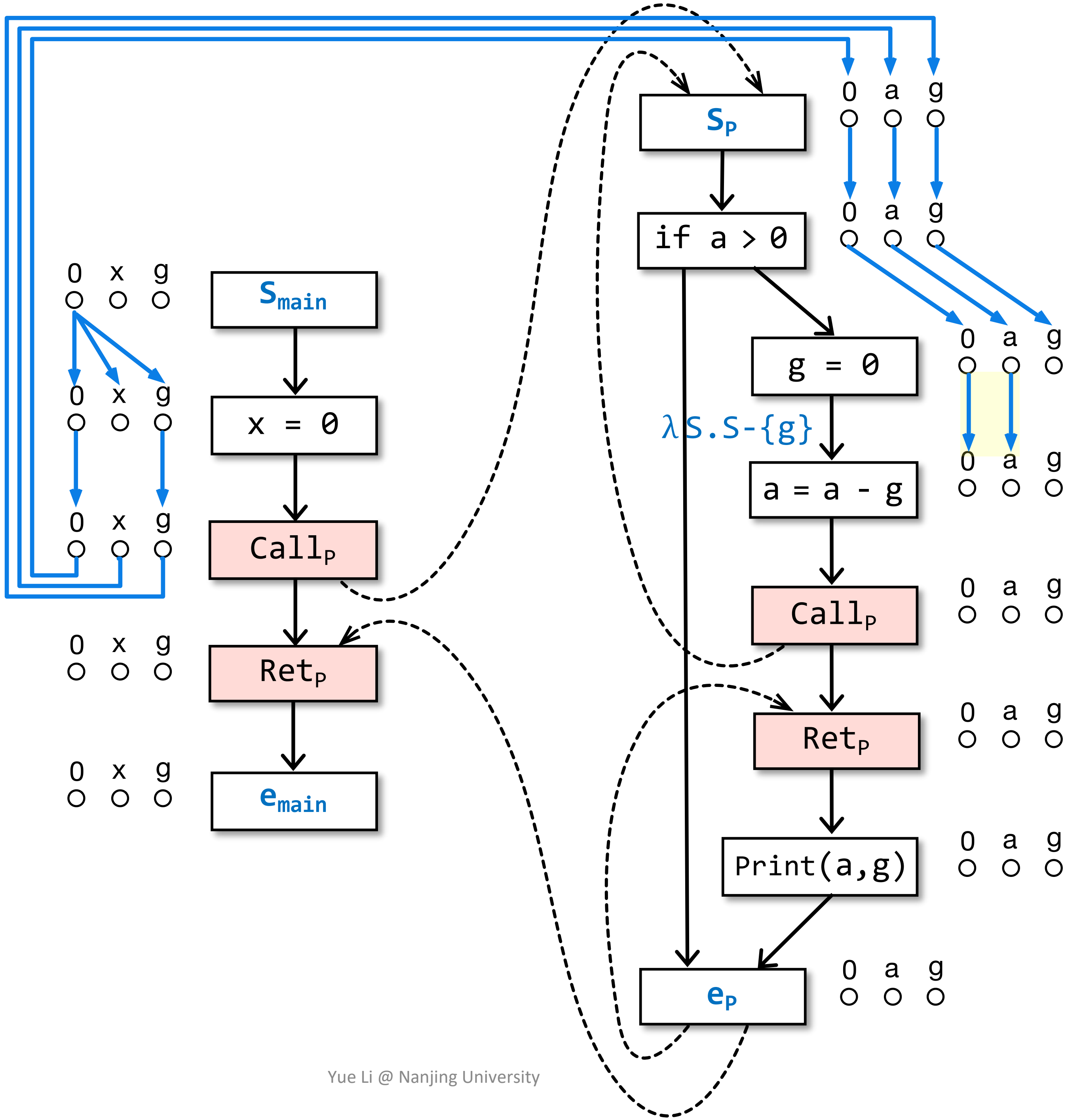


0 a g
0 a g
λS.S
0 x g
S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p

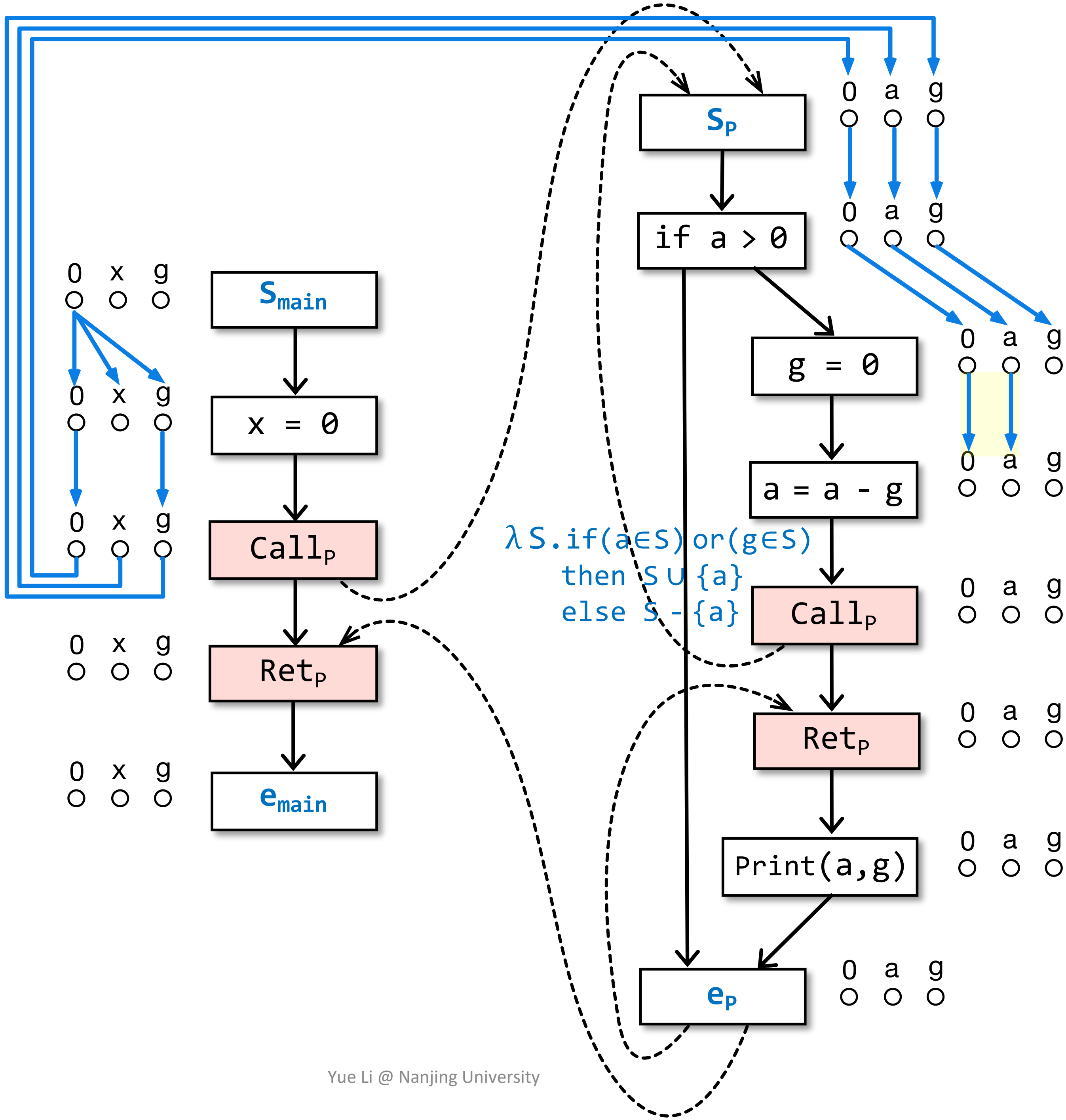


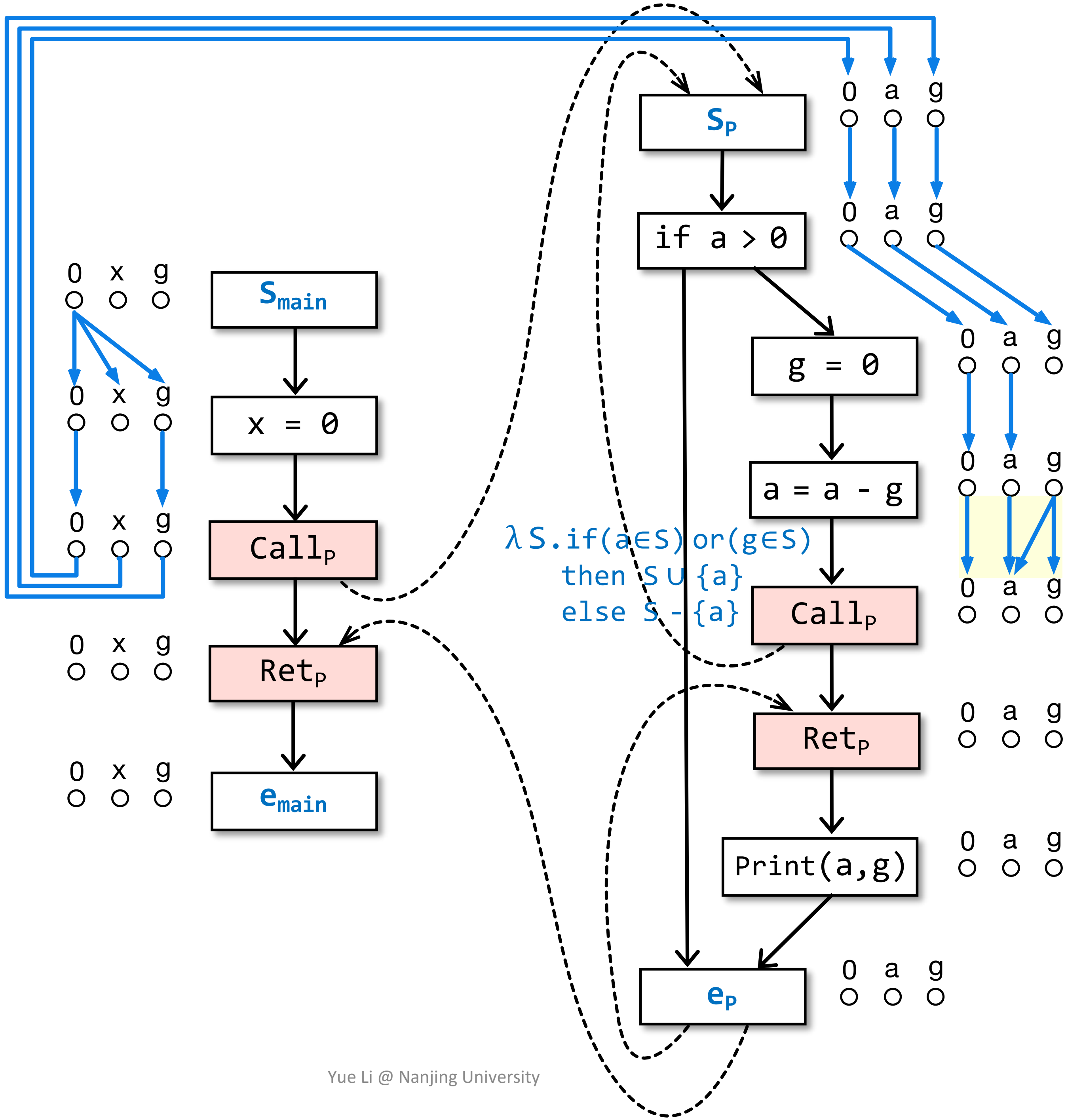
0 x g
s_{main}
x = 0
$Call_p$
Ret_p
e_{main}
0 a g
s_p
if a > 0
g = 0
a = a - g
Print(a,g)
e_p
λs.s


0 x g
Smain
x = 0
Callp
Retp
emain
Sp
if a > 0
g = 0
λS.S-{g}
a = a - g
Callp
Retp
Print(a,g)
ep
0 a g

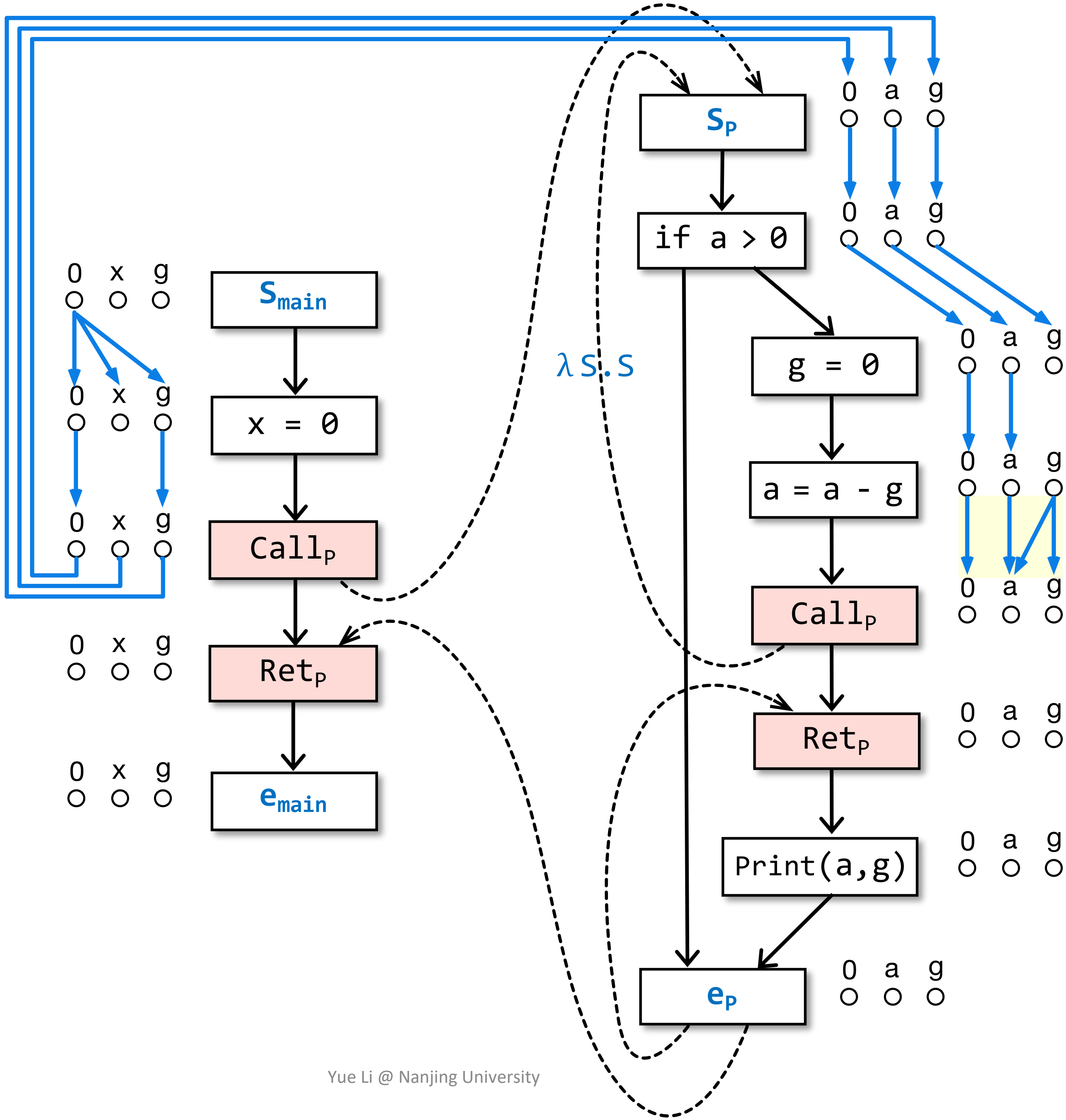


S_main
x = 0
Call_p
Ret_p
e_main
S_p
if a > 0
g = 0
λS.S-{g}
a = a - g
Call_p
Ret_p
Print(a,g)
e_p
0 x g
0 a g

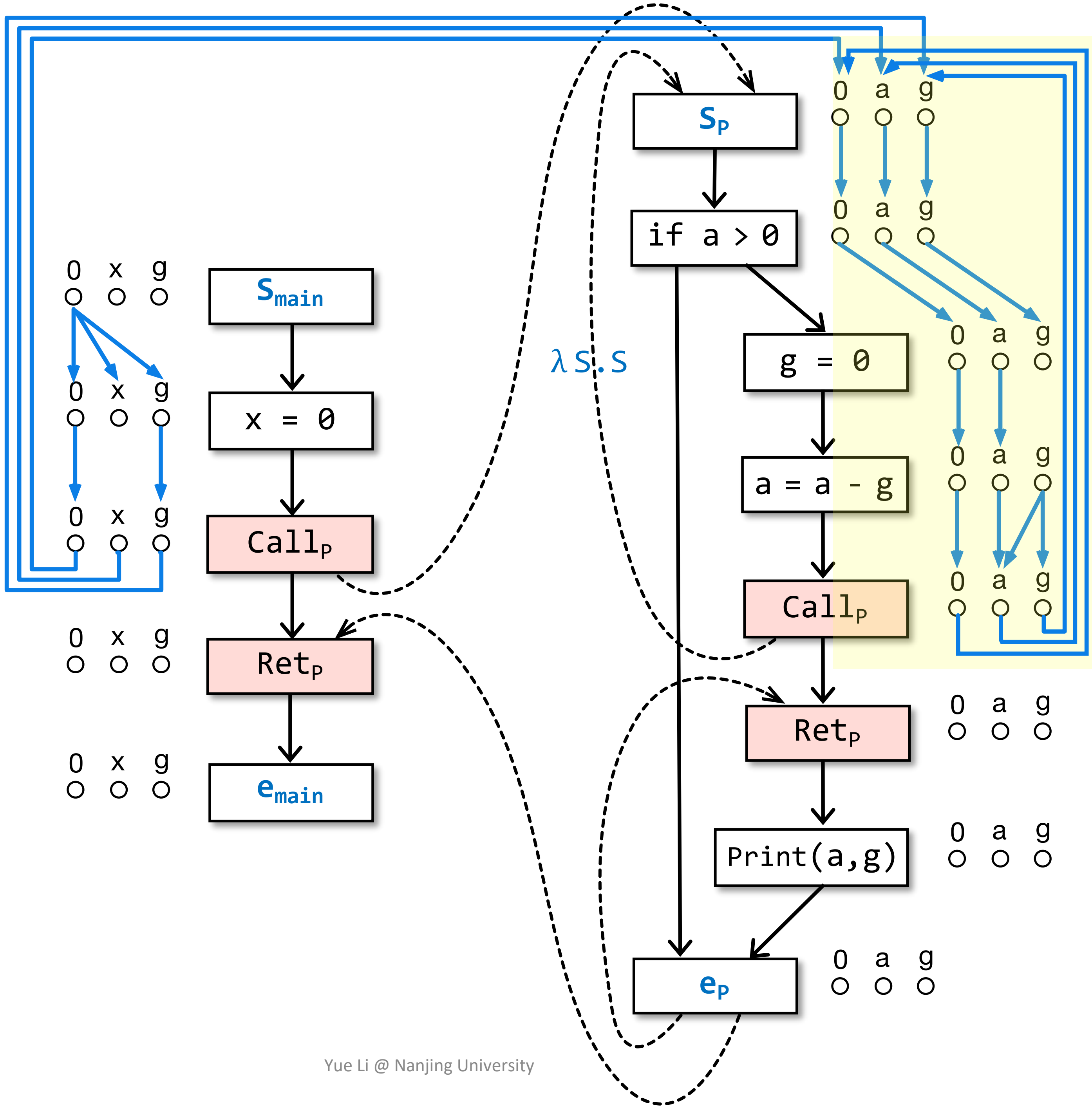

0 x g
s_{main}
x = 0
$Call_P$
Ret_P
e_{main}
s_P
if a > 0
g = 0
a = a - g
Print(a,g)
e_P
0 a g
λS.if(a∈S) or(g∈S)
then S ∪ {a}
else S - {a}



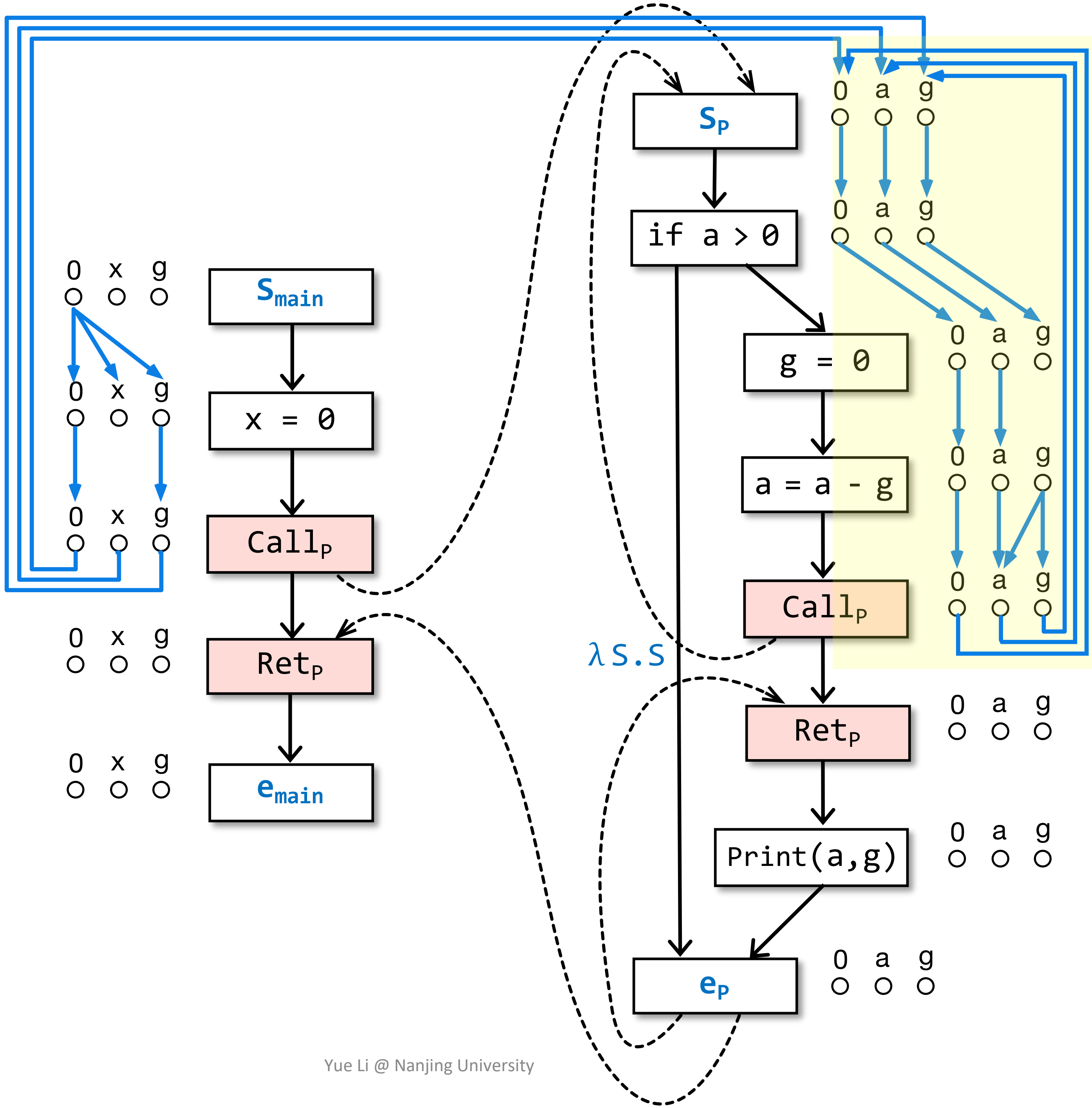
0 x g
s_main
x = 0
Call_P
Ret_P
e_main
s_P
if a > 0
g = 0
a = a - g
Print(a,g)
e_P
0 a g
λS.if(a∈S) or(g∈S)
then S ∪ {a}
else S - {a}



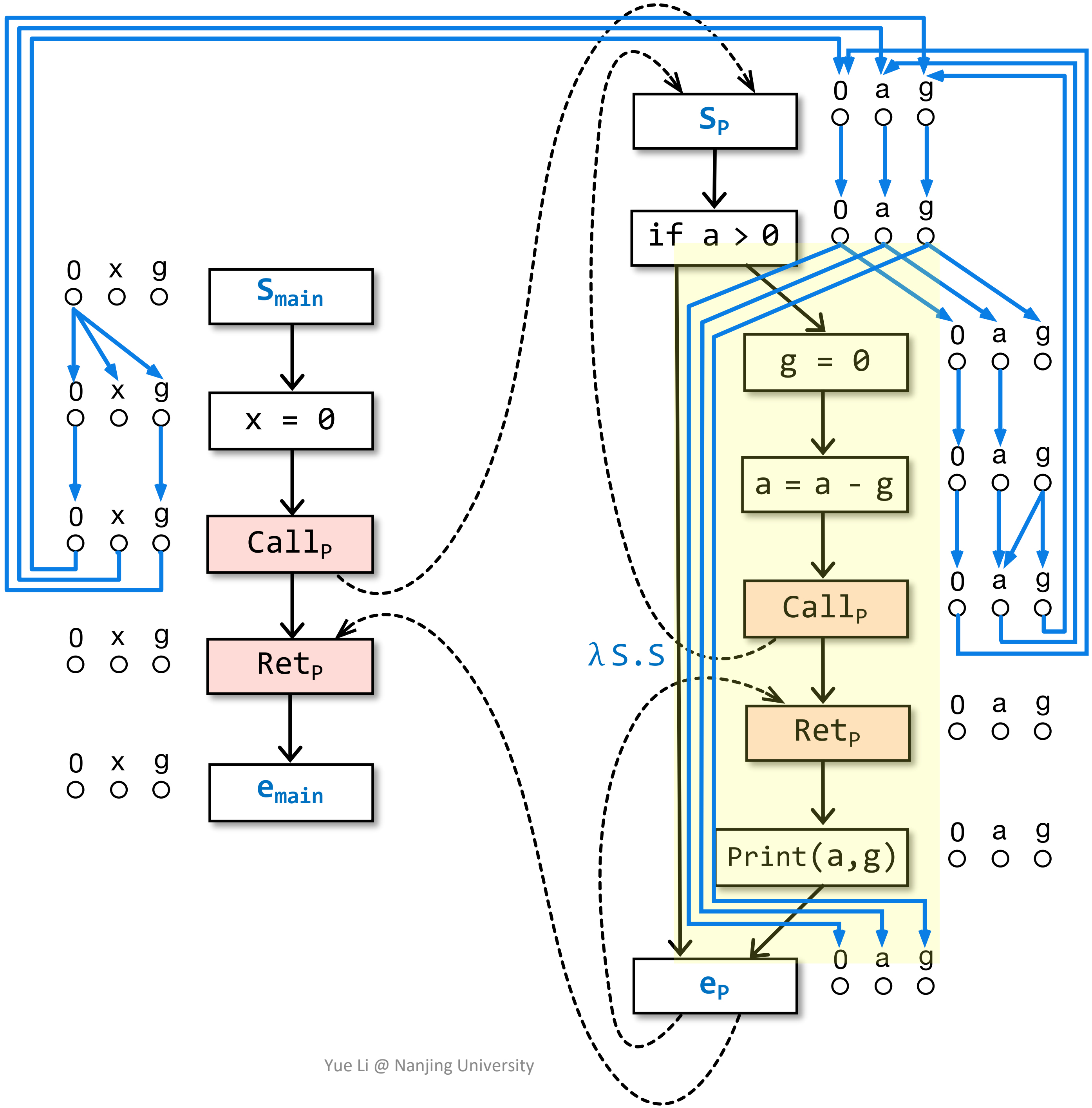
Yue Li @ Nanjing University



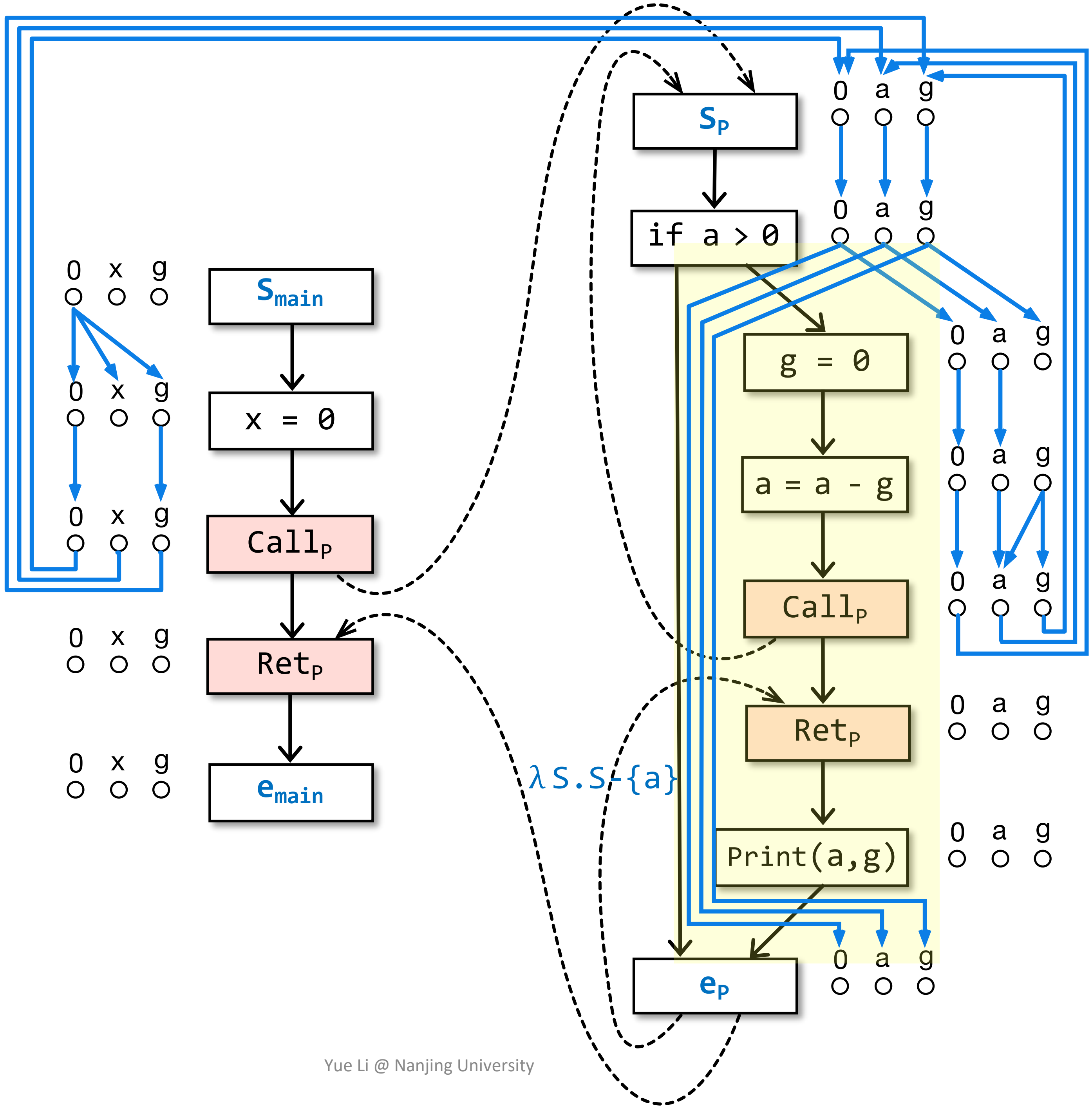
0 x g
S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
λ S.S
S_p
0 a g
if a > 0
g = 0
a = a - g
Print(a,g)
e_p



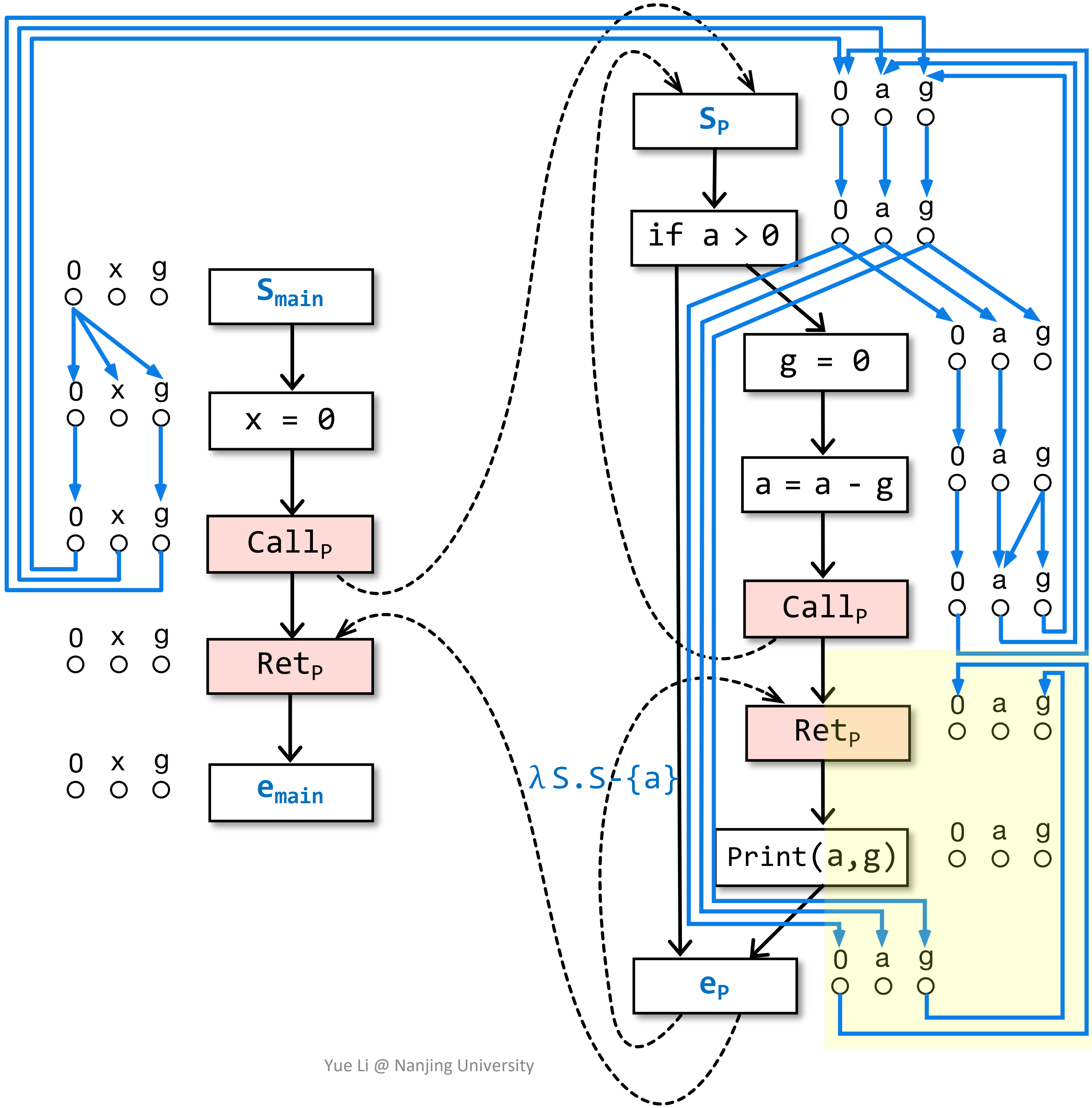
S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
λS.S
0 x g
0 a g



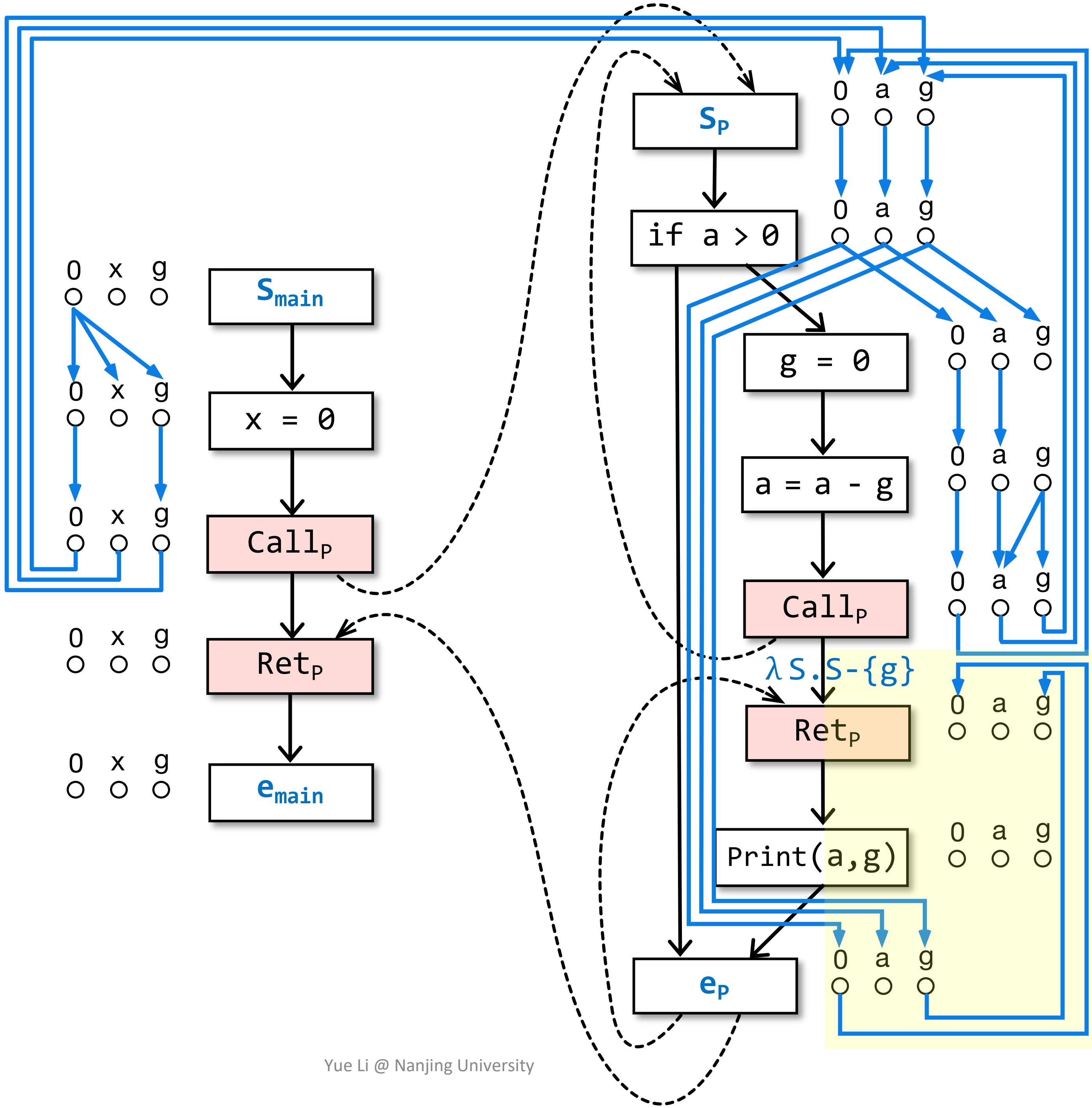
S_{main}
x = 0
$Call_P$
Ret_P
e_{main}
S_P
if a > 0
g = 0
a = a - g
$Call_P$
Ret_P
Print(a,g)
e_P
λS.S
0 x g
0 a g

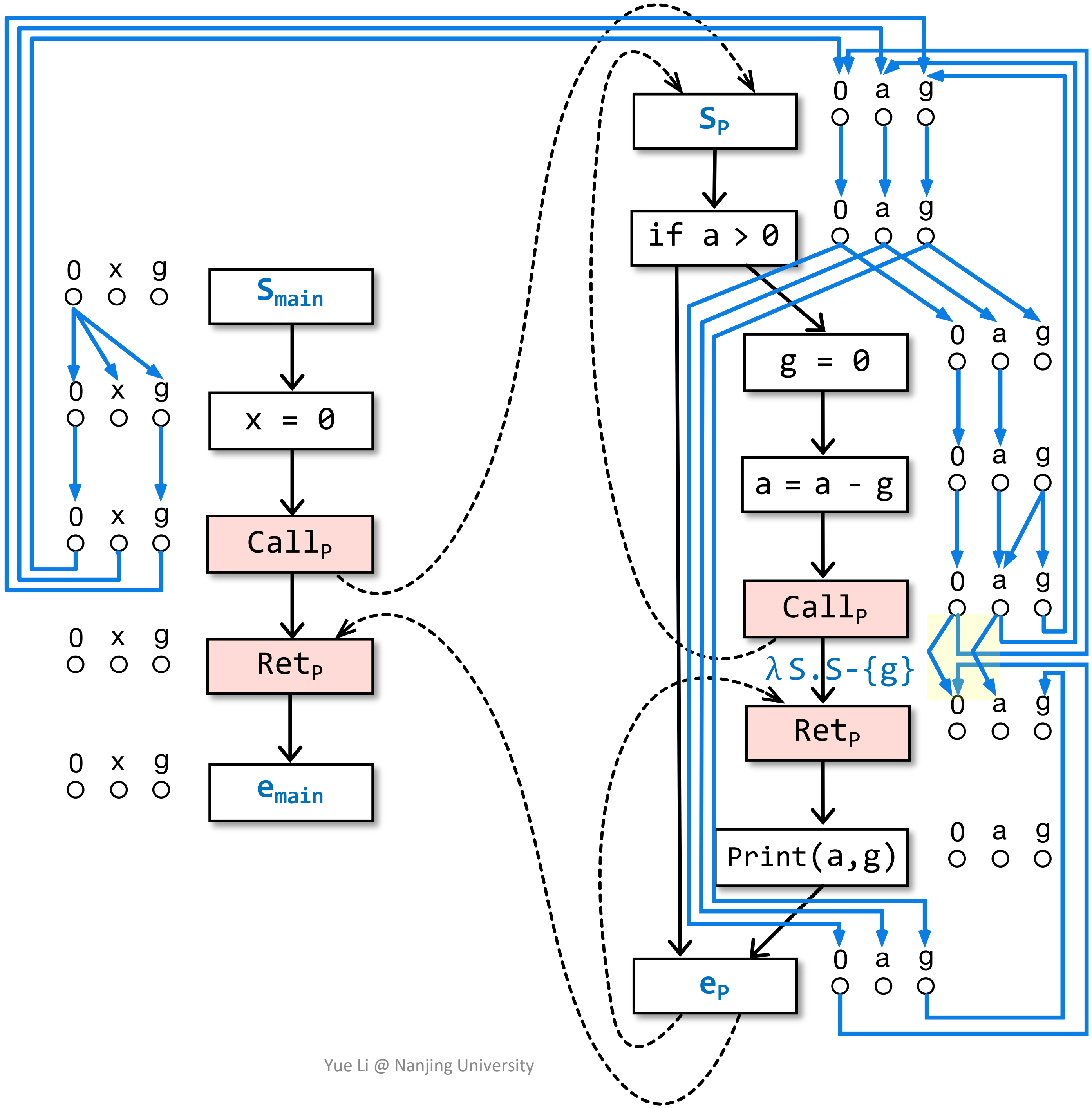


S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
λS.S-{a}
0 x g
0 a g

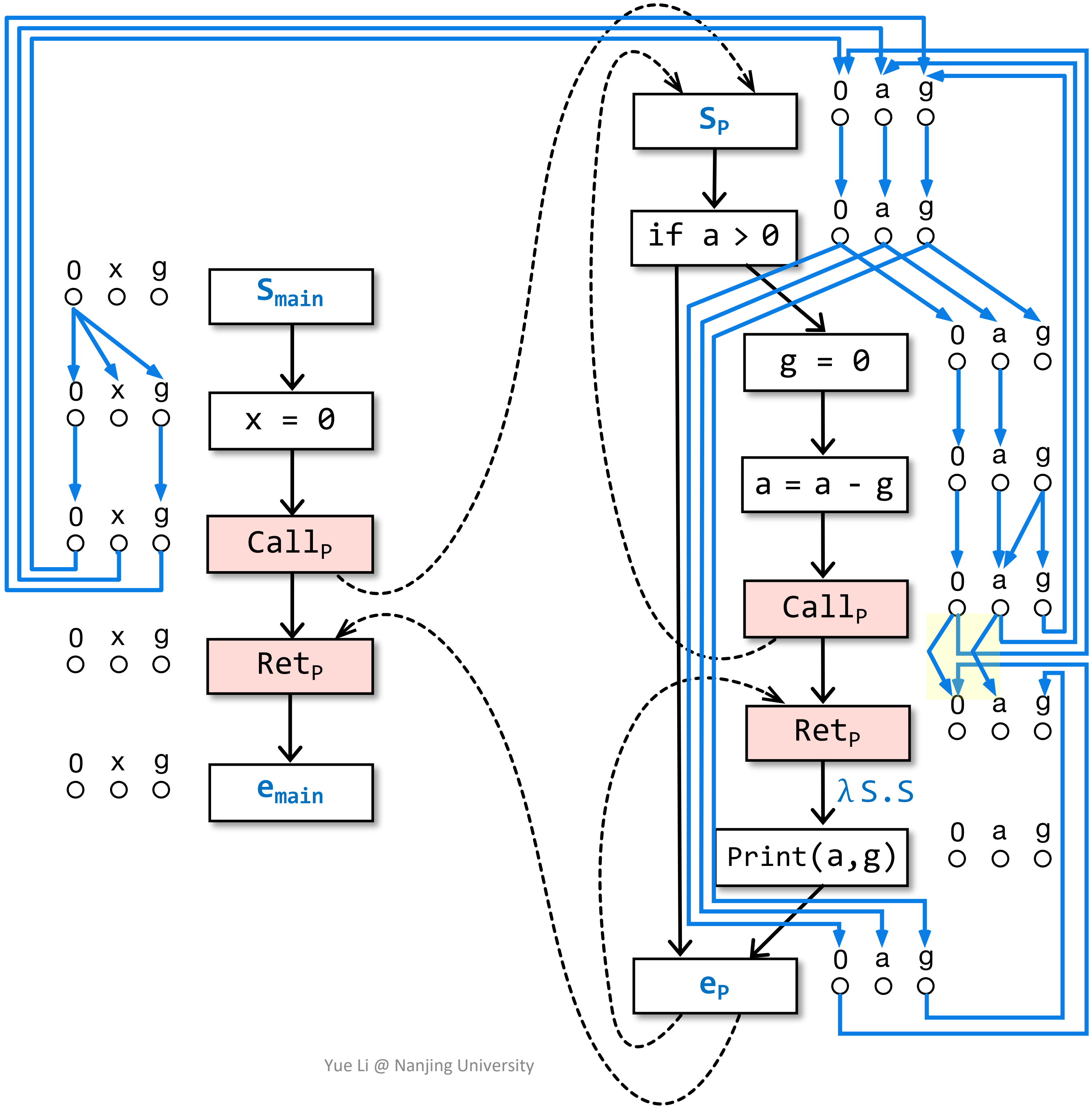


0 x g
s_{main}
x = 0
$Call_p$
Ret_p
e_{main}
s_p
if a > 0
0 a g
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
λS.S-{a}

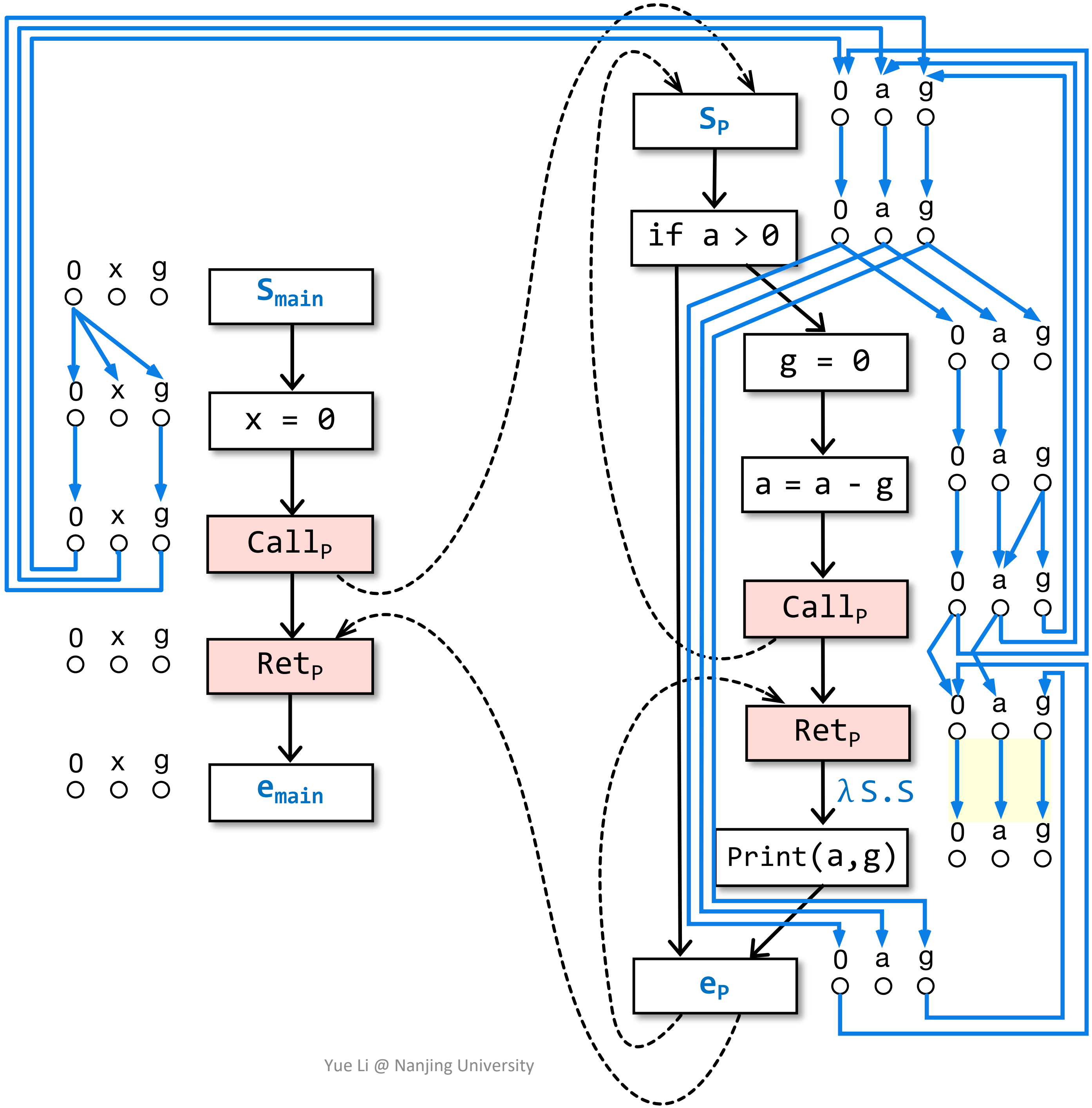

S_main
x = 0
Call_p
Ret_p
e_main
S_p
if a > 0
g = 0
a = a - g
Call_p
λS.S-{g}
Ret_p
Print(a,g)
e_p
0 x g
0 a g



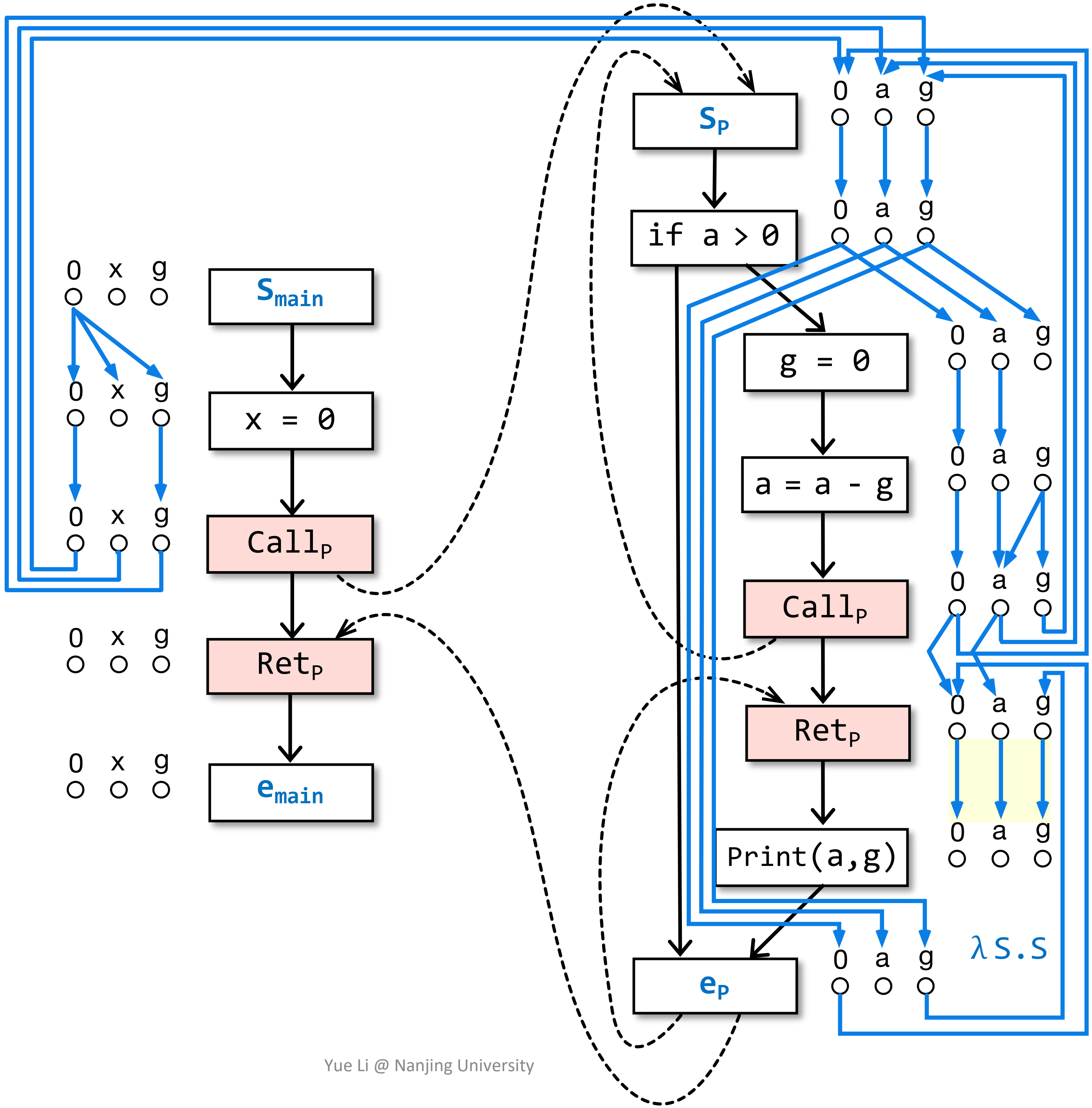
s_{main}
x = 0
$Call_p$
Ret_p
e_{main}
s_p
if a > 0
g = 0
a = a - g
$Call_p$
λS.S-{g}
Ret_p
Print(a,g)
e_p
0 x g
0 a g



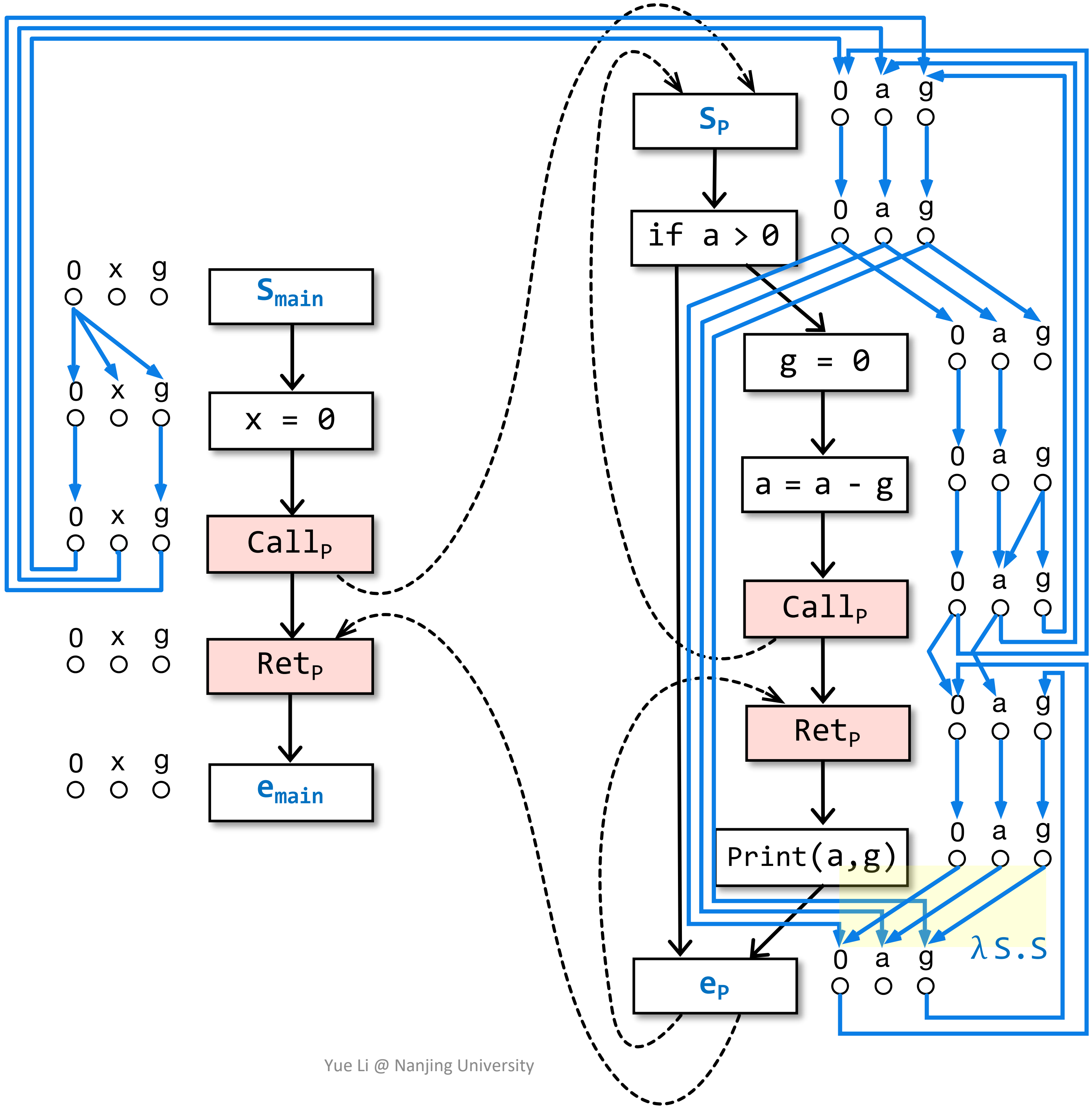
S_main
x = 0
Call_p
Ret_p
e_main
S_p
if a > 0
g = 0
a = a - g
Call_p
Ret_p
λS.S
Print(a,g)
e_p
0 x g
0 a g



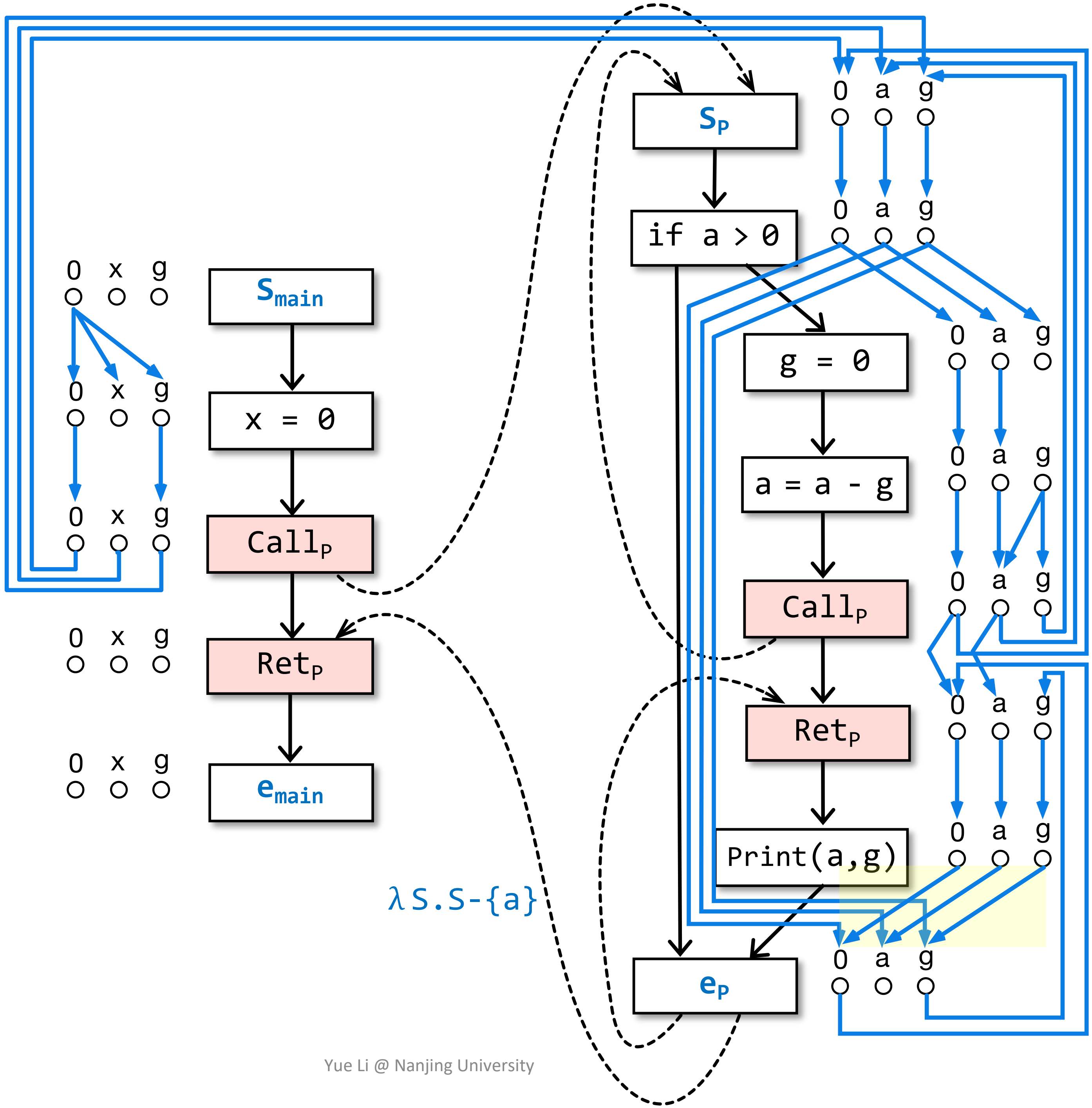
0 x g
S_{main}
x = 0
$Call_P$
Ret_P
e_{main}
S_P
if a > 0
0 a g
g = 0
a = a - g
Print(a,g)
$\lambda S.S$
e_P



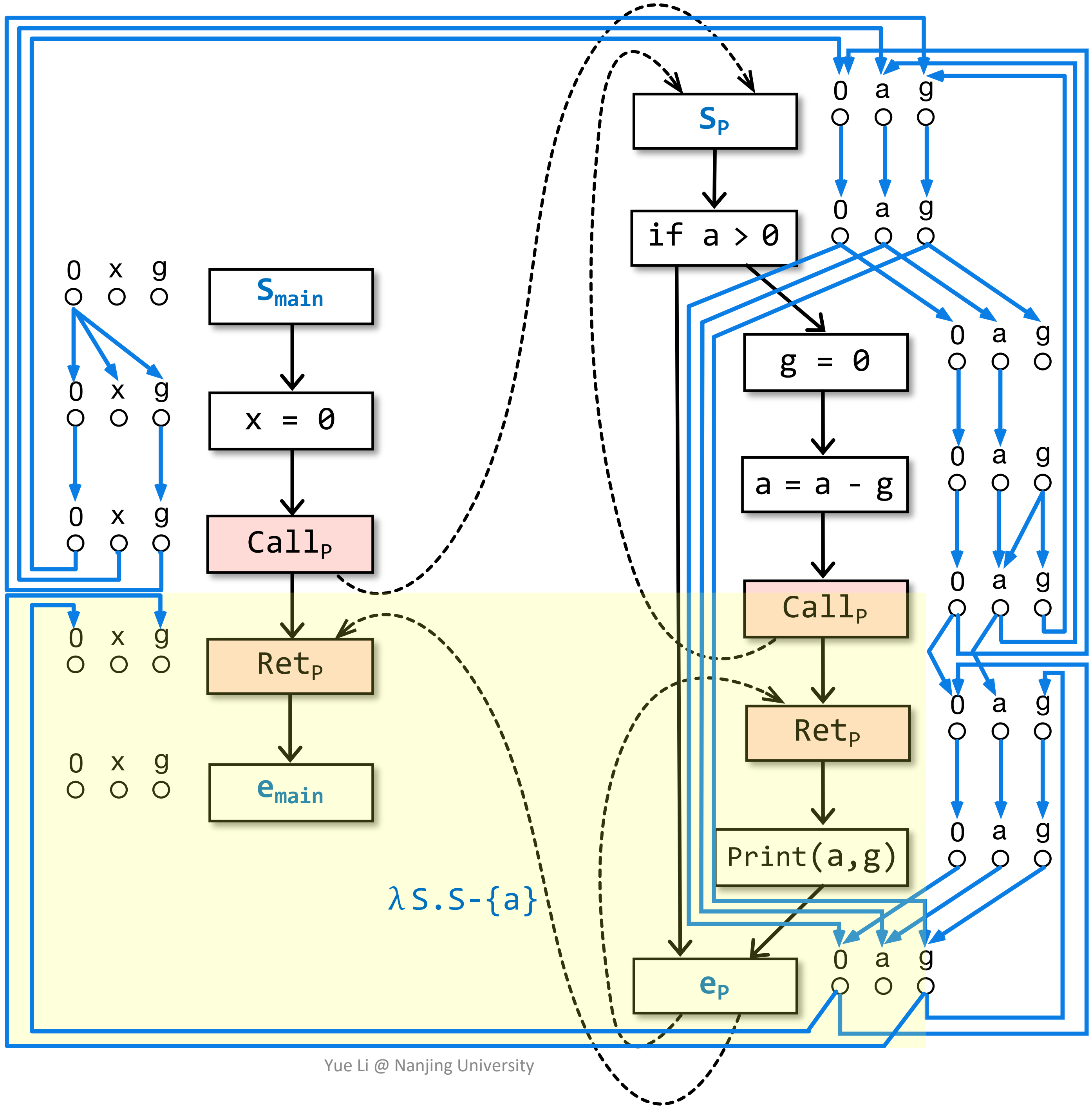
Smain
x = 0
Callp
Retp
emain
Sp
if a > 0
g = 0
a = a - g
Callp
Retp
Print(a,g)
ep
0 x g
0 a g
λS.S

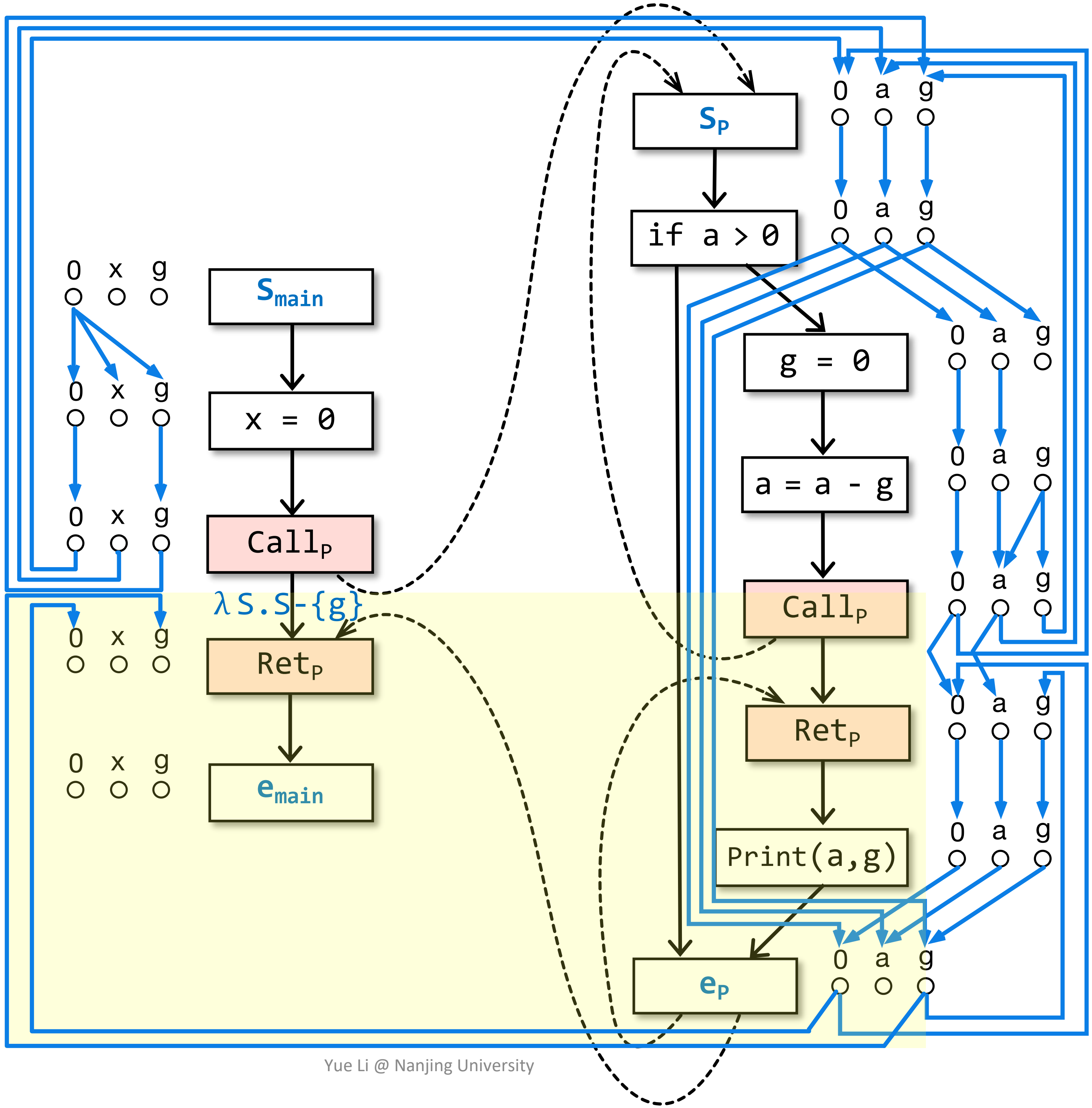


Smain
x = 0
CallP
RetP
emain
SP
if a > 0
g = 0
a = a - g
CallP
RetP
Print(a,g)
eP
0 x g
0 a g
λS.S

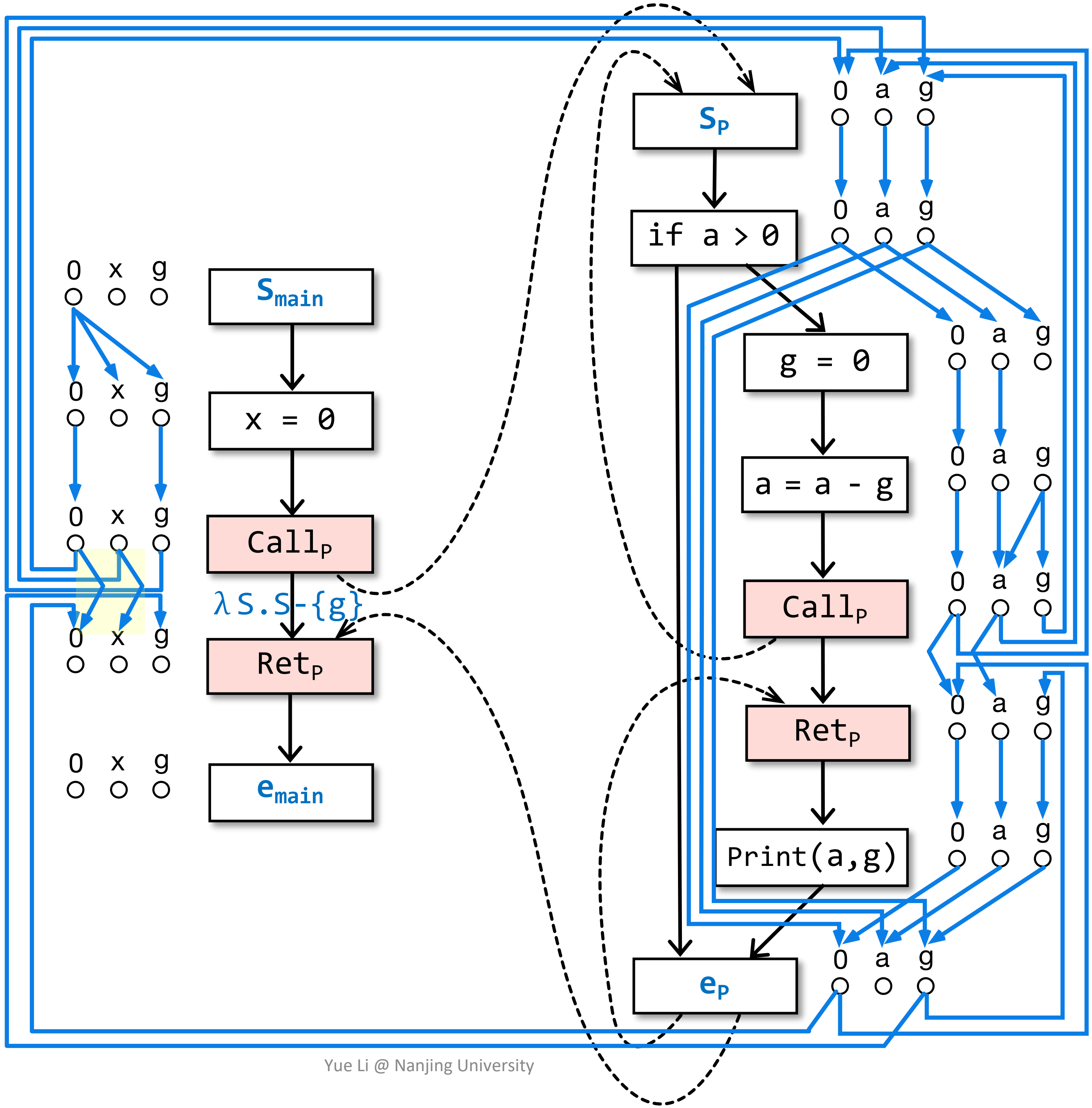


S_main
x = 0
Call_p
Ret_p
e_main
S_p
if a > 0
g = 0
a = a - g
Call_p
Ret_p
Print(a,g)
e_p
0 x g
0 a g
λS.S-{a}

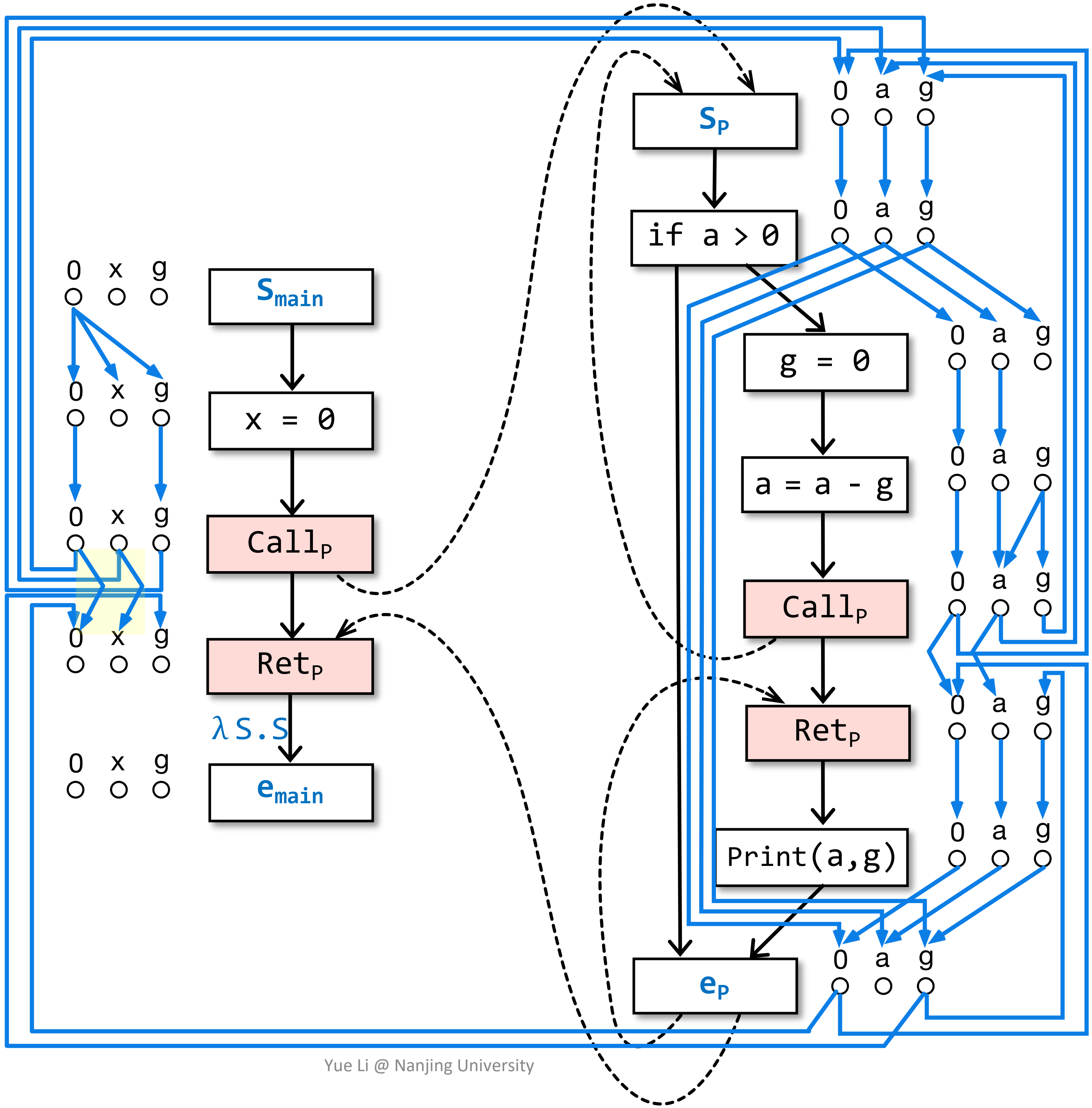

S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
0 x g
0 a g
λS.S-{a}

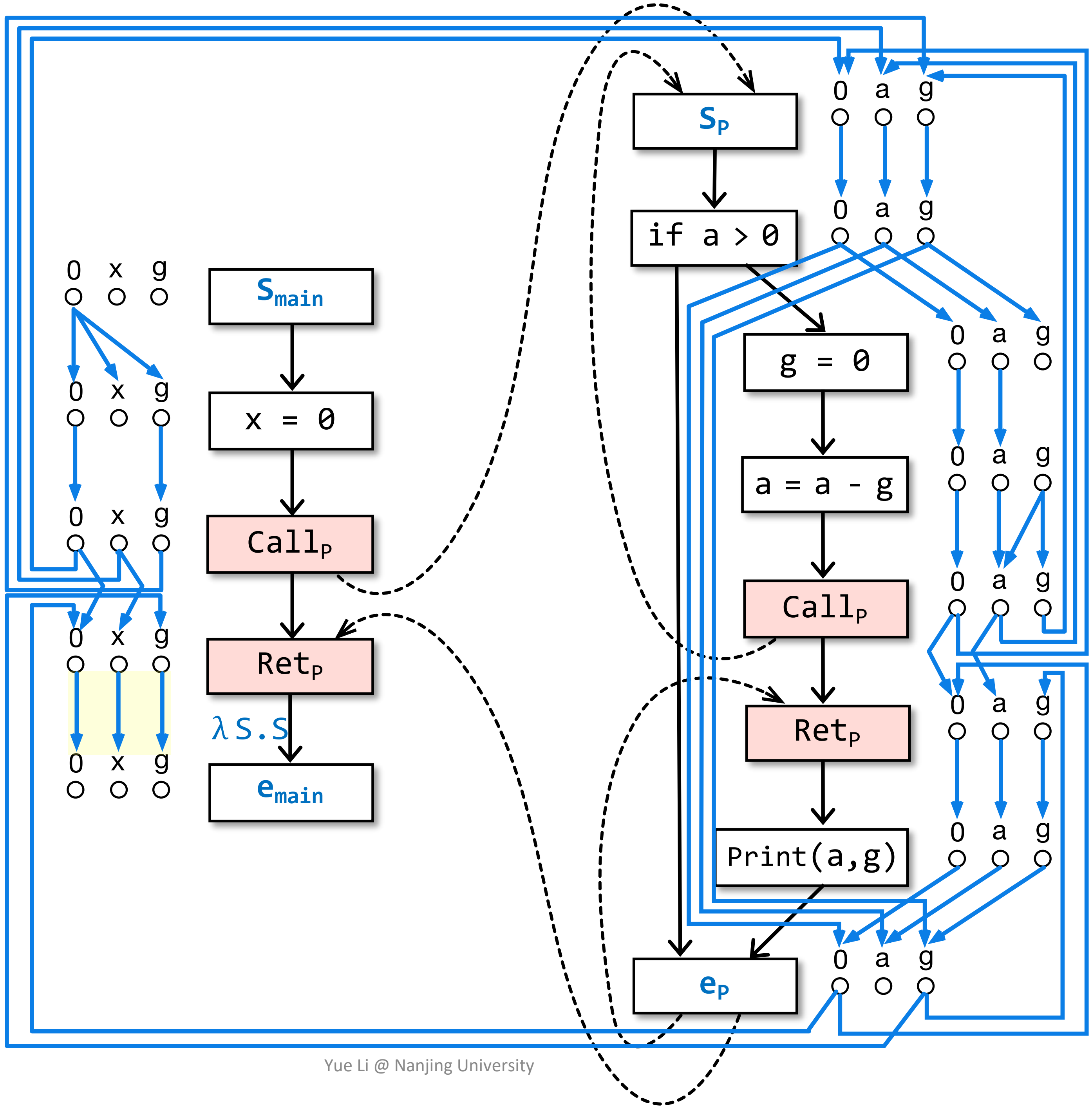


S_{main}
x = 0
$Call_P$
Ret_P
e_{main}
λS.S-{g}
S_P
if a > 0
g = 0
a = a - g
$Call_P$
Ret_P
Print(a,g)
e_P

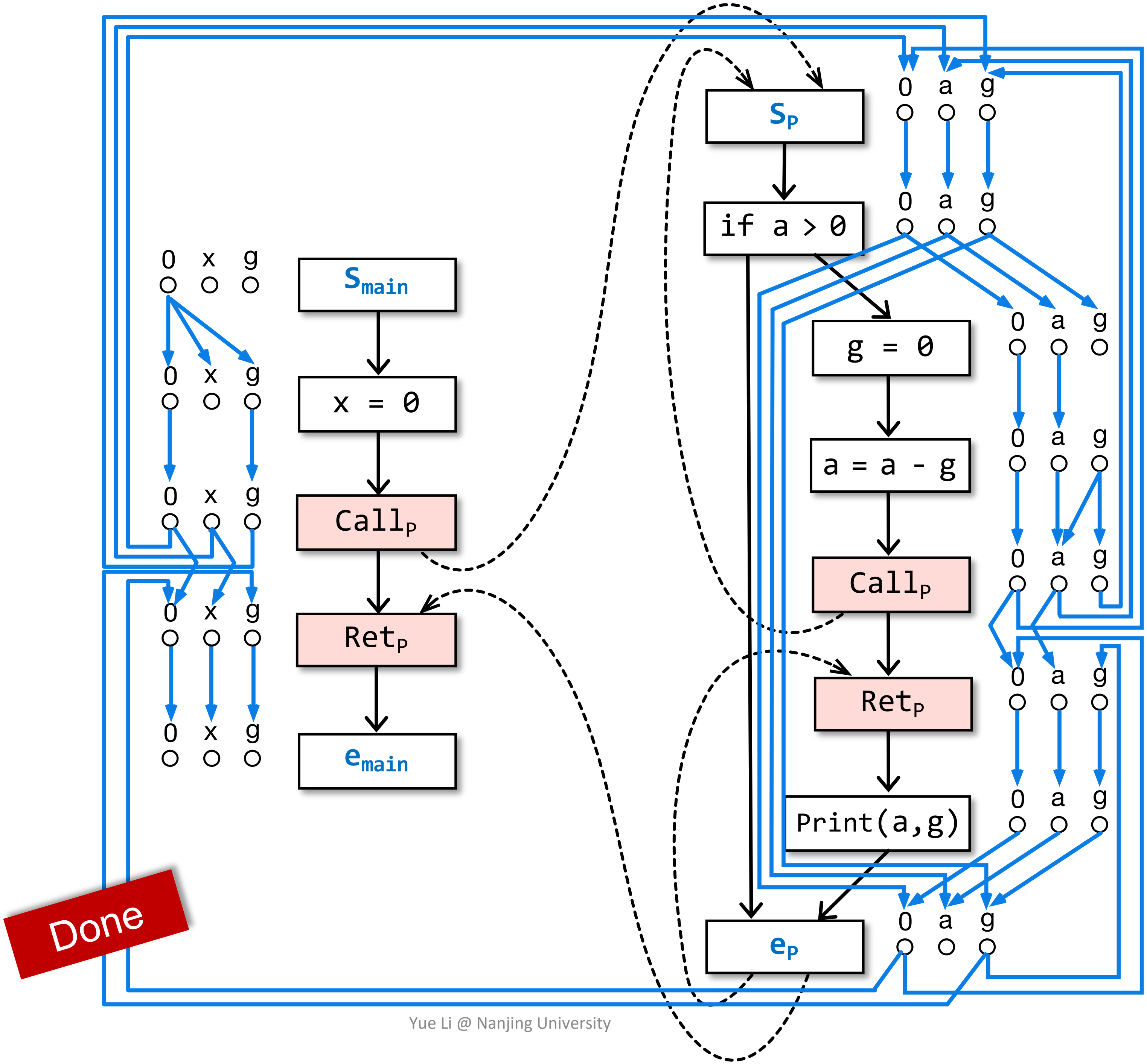


0 x g
S_{main}
x = 0
$Call_p$
λS.S-{g}
Ret_p
e_{main}
0 a g
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p

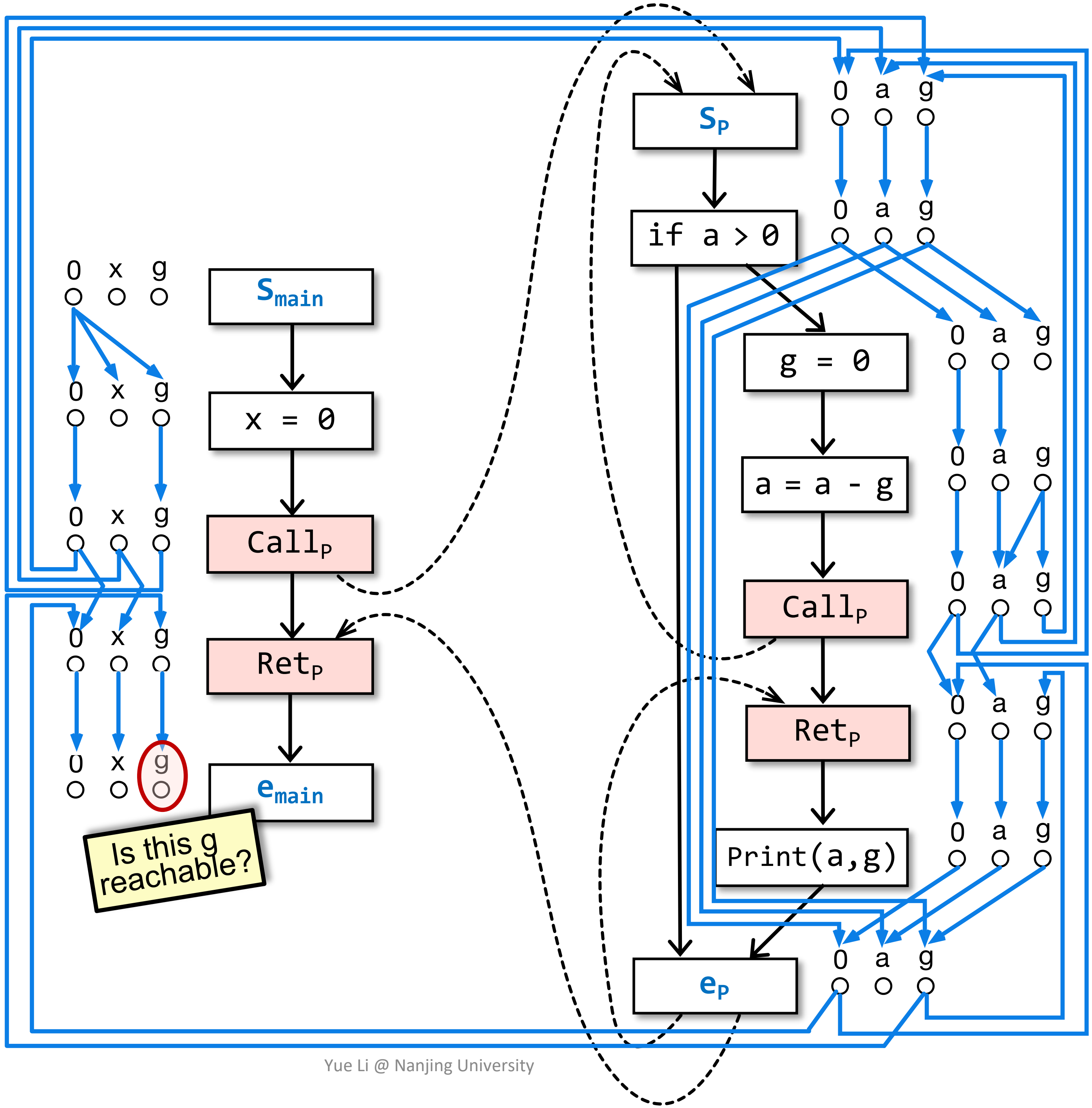

S_main
x = 0
Call_p
Ret_p
λS.S
e_main
S_p
if a > 0
g = 0
a = a - g
Call_p
Ret_p
Print(a,g)
e_p
0 x g
0 a g

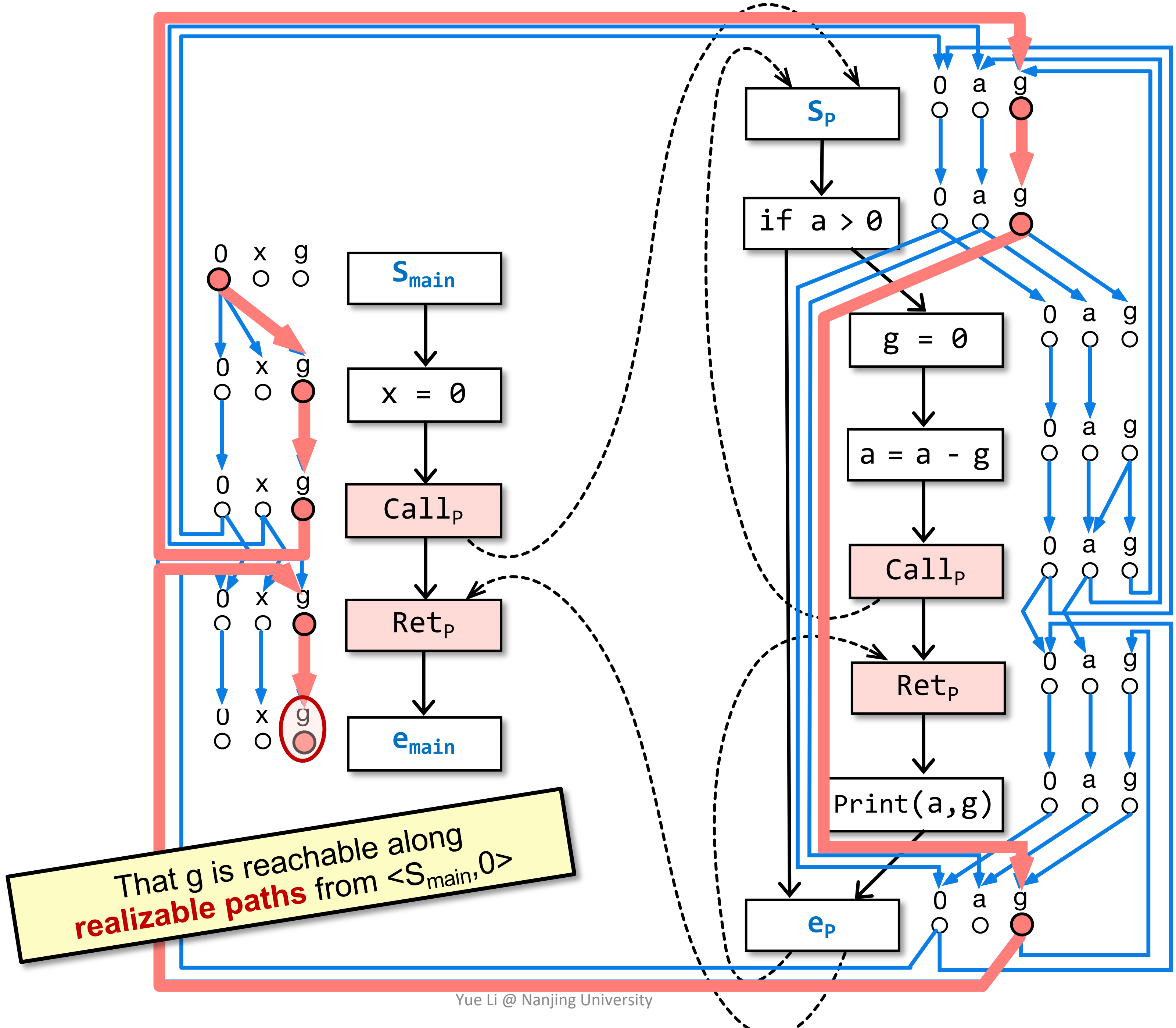


S_{main}
x = 0
$Call_p$
Ret_p
λS.S
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
0 x g
0 a g

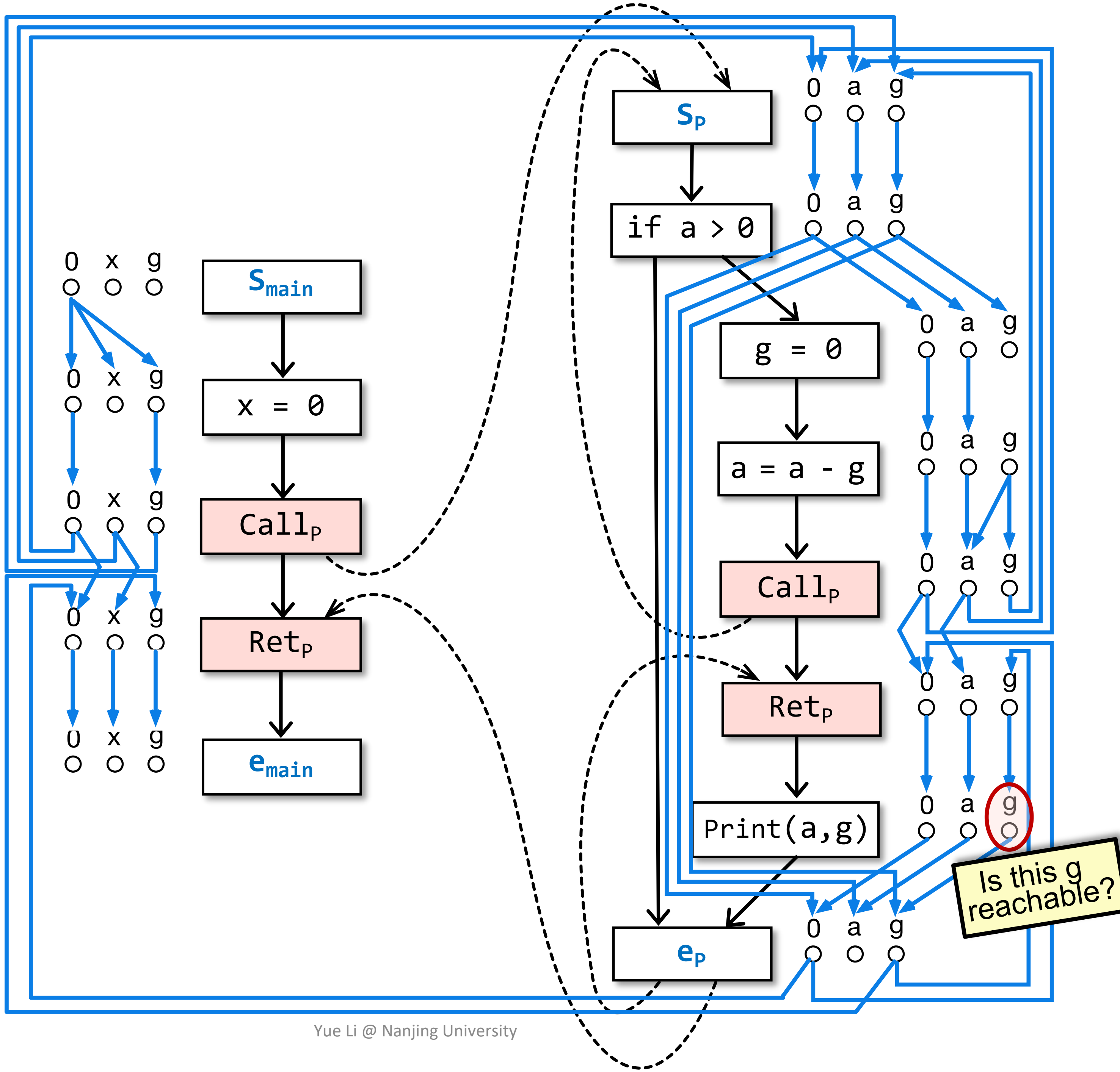


0 x g
S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
0 a g
Done

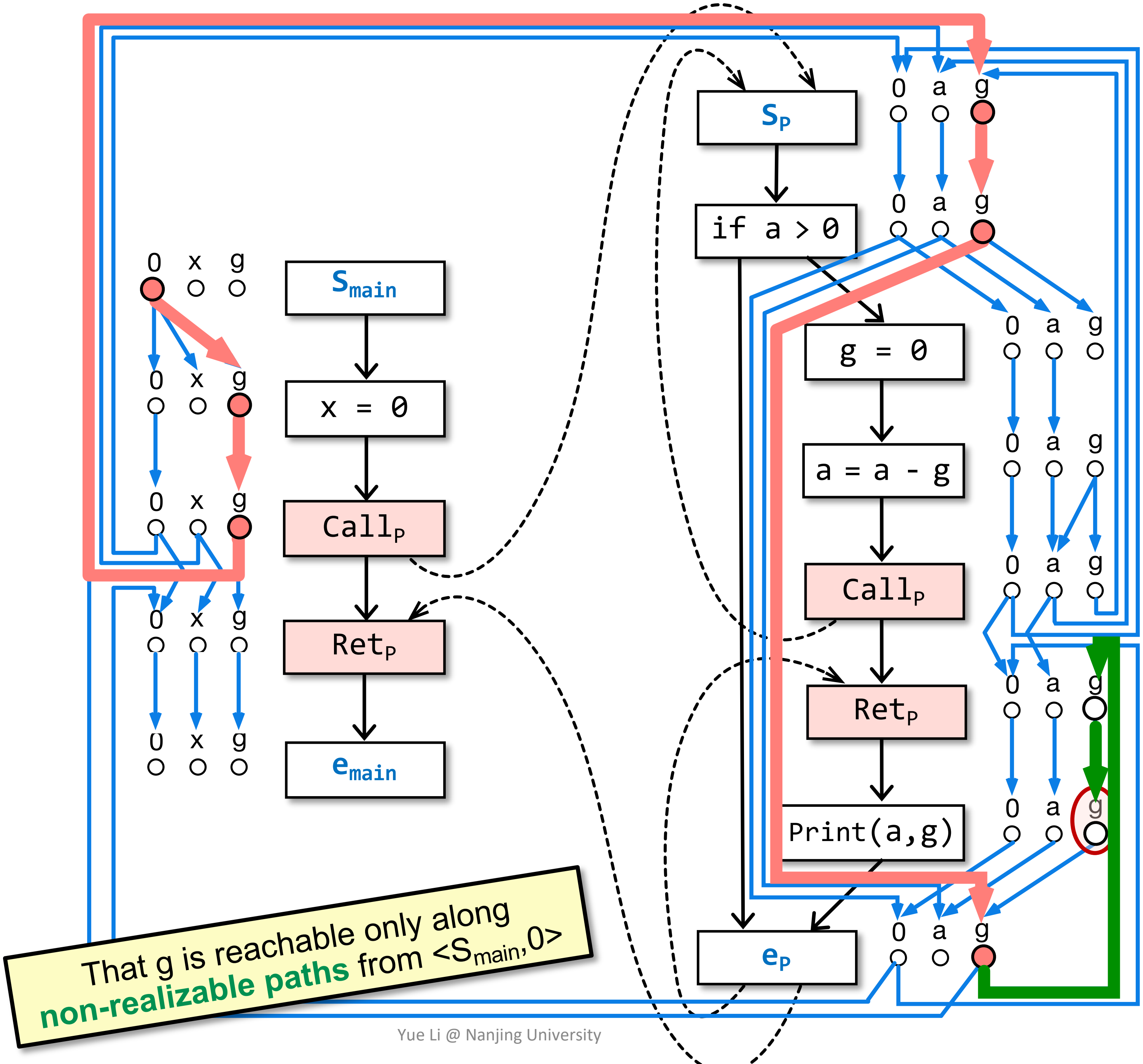

0 x g
S_main
x = 0
Call_p
Ret_p
e_main
Is this g reachable?
0 a g
S_p
if a > 0
g = 0
a = a - g
Call_p
Ret_p
Print(a,g)
e_p



S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
That g is reachable along **realizable paths** from <S_{main},0>

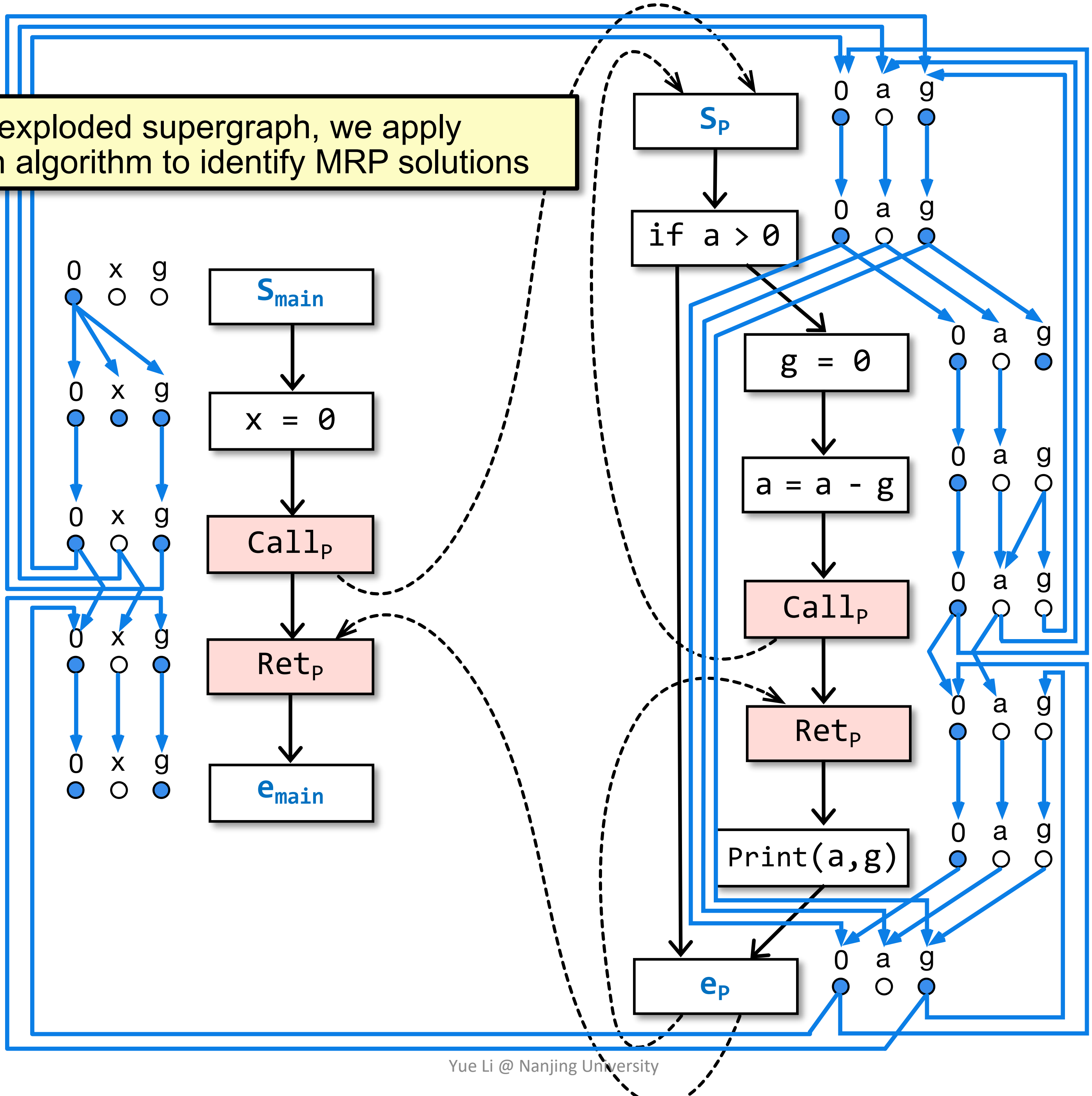


0 x g
S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
0 a g
Is this g reachable?

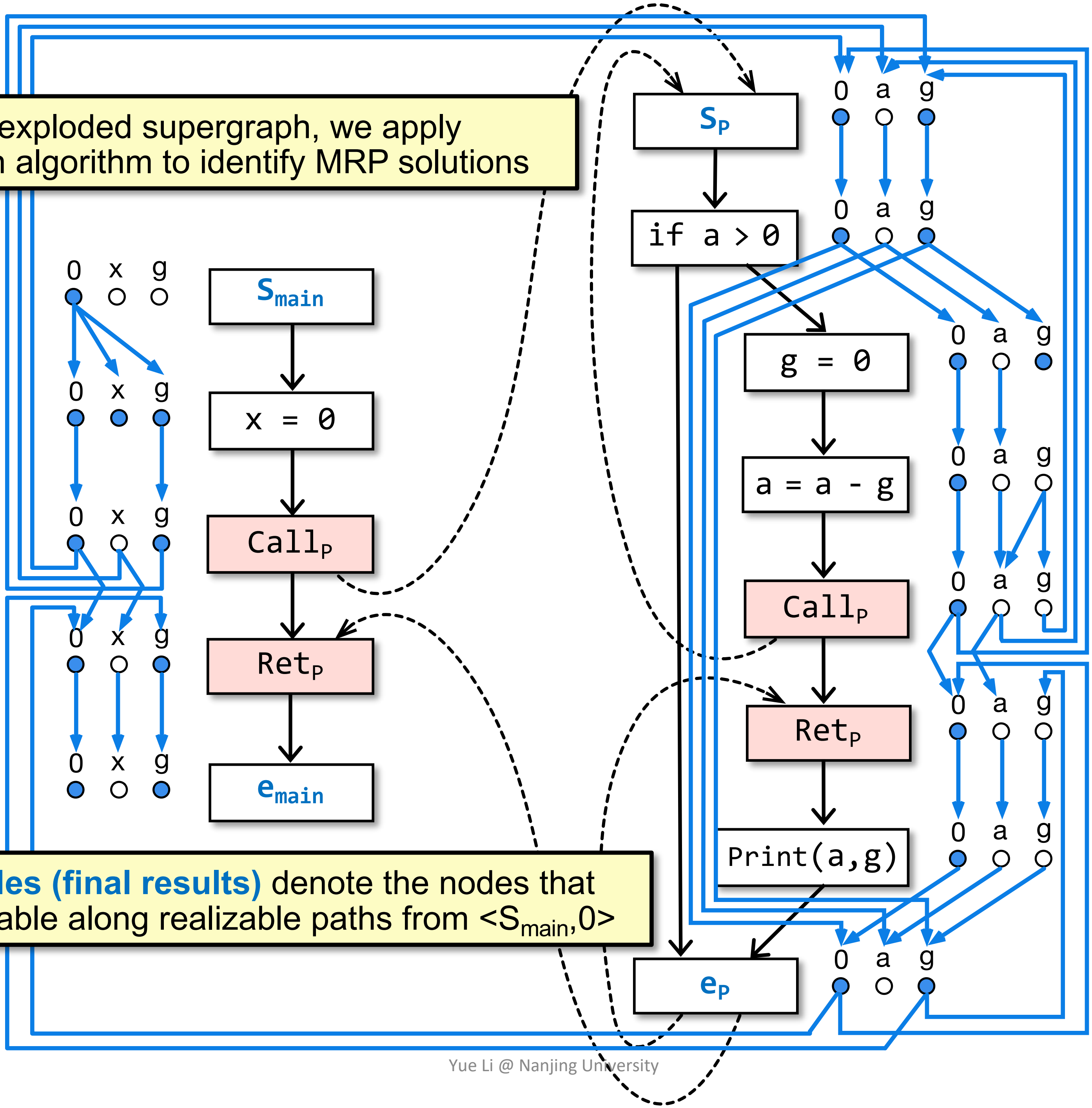


S_{main}
x = 0
$Call_p$
Ret_p
e_{main}
S_p
if a > 0
g = 0
a = a - g
$Call_p$
Ret_p
Print(a,g)
e_p
0 x g
0 a g
That g is reachable only along non-realizable paths from <S_{main},0>

Given an exploded supergraph, we apply Tabulation algorithm to identify MRP solutions



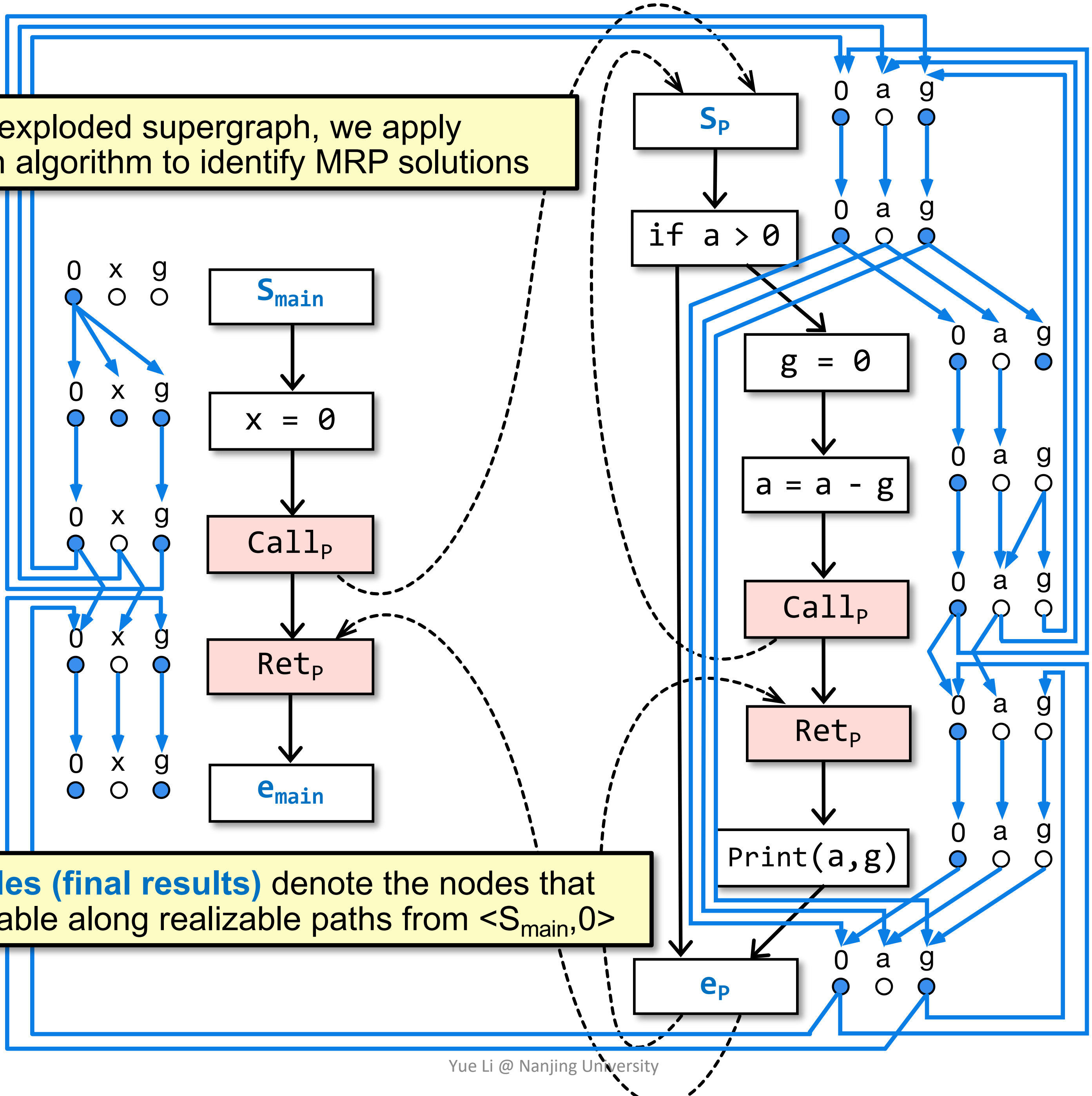
Given an exploded supergraph, we apply Tabulation algorithm to identify MRP solutions



Blue circles (final results) denote the nodes that are reachable along realizable paths from $<S_{main},0>$

How?

Given an exploded supergraph, we apply Tabulation algorithm to identify MRP solutions

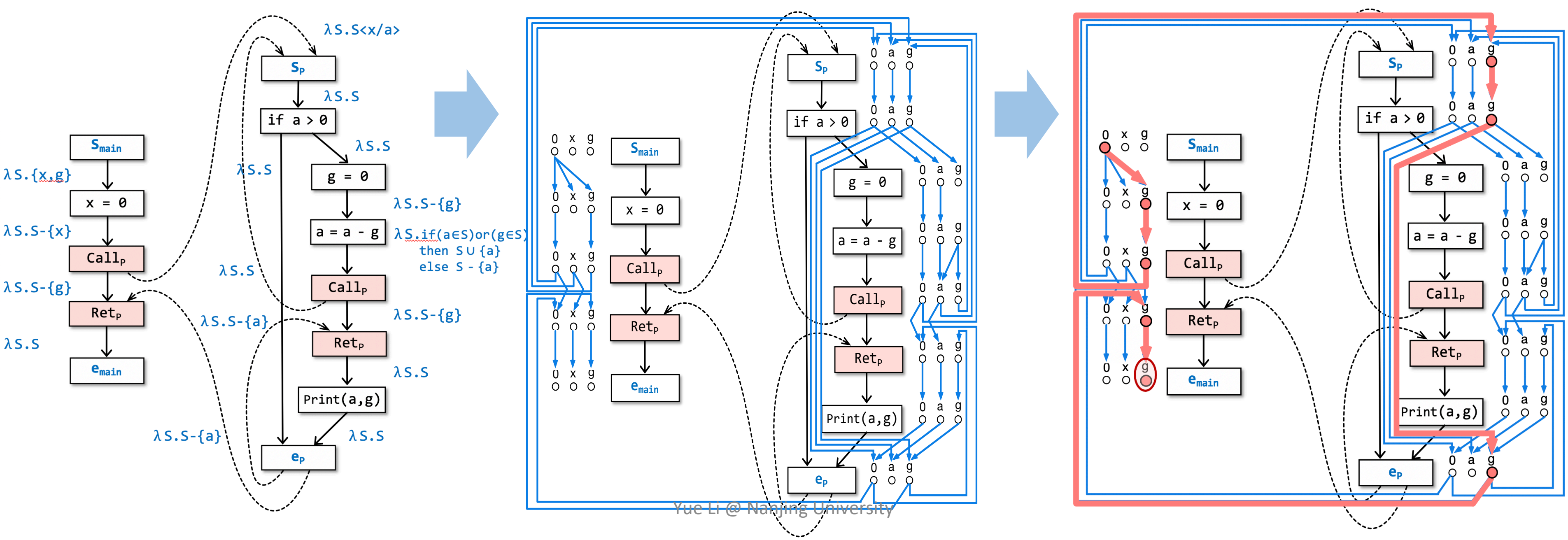


Blue circles (final results) denote the nodes that are reachable along realizable paths from $\langle S_{main}, 0 \rangle$

Overview of IFDS

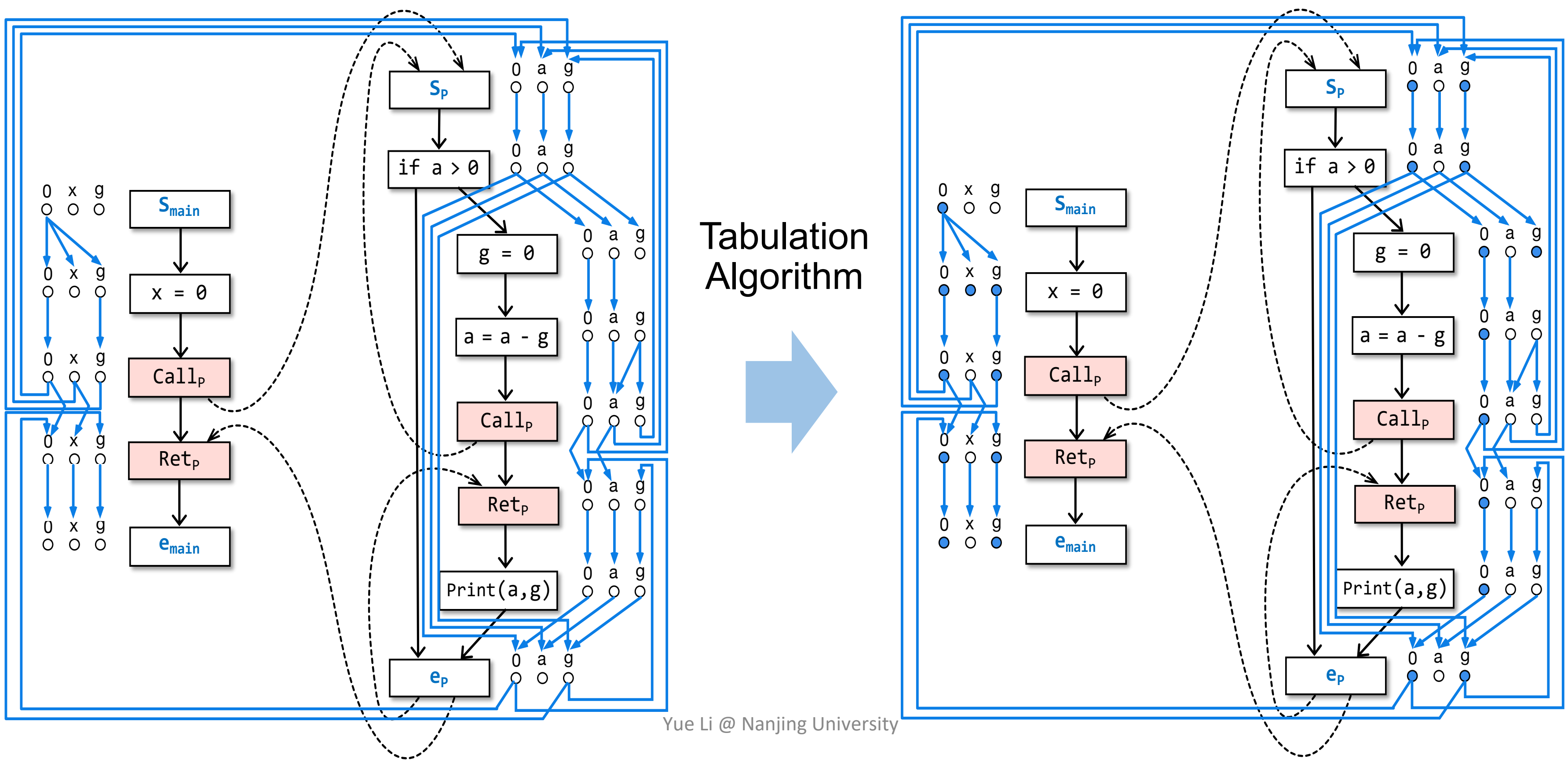
Given a program P, and a dataflow-analysis problem Q

- Build a supergraph G* for P and define flow functions for edges in G* based on Q
- Build exploded supergraph $G^{\#}$ for P by transforming flow functions to representation relations (graphs)
- Q can be solved as graph reachability problems (find out MRP solutions) via applying Tabulation algorithm on $G^{\#}$



Tabulation Algorithm

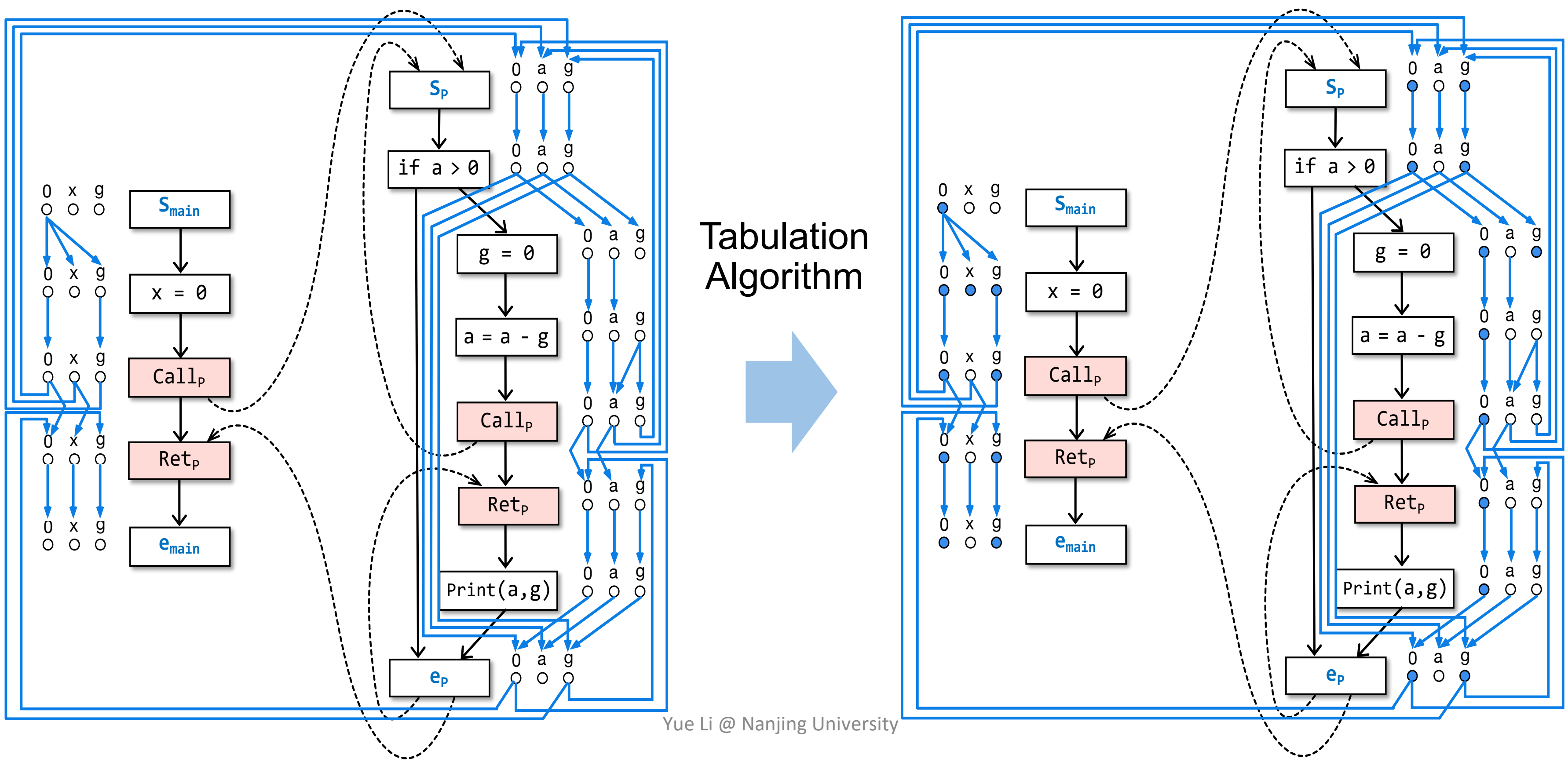
Given an exploded supergraph $G^{\#}$, Tabulation algorithm determines the MRP solution by finding out all realizable paths starting from $\langle s_{main}, 0\rangle$



Tabulation Algorithm

Given an exploded supergraph $G^\#$, Tabulation algorithm determines the MRP solution by finding out all realizable paths starting from <s_{main}, 0>

Let n be a program point, data fact d ∈ MRP_n, iff there is a realizable path in $G^\#$ from <s_{main}, 0> to <n, d>. (then d's white circle turns to blue)



Tabulation Algorithm

```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate(G#_IP)
begin
  Let (N#, E#) = G#_IP
  PathEdge := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  WorkList := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  SummaryEdge := ∅
  ForwardTabulateSLRPs()
  for each n ∈ N* do
    X_n := { d_2 ∈ D | ∃ d_1 ∈ (D ∪ { 0 }) such that ⟨s_procOf(n), d_1⟩ → ⟨n, d_2⟩ ∈ PathEdge }
  od
end
procedure Propagate(e)
begin
  if e ∉ PathEdge then  Insert e into PathEdge;  Insert e into WorkList  fi
end
procedure ForwardTabulateSLRPs()
begin
  while WorkList ≠ ∅ do
    Select and remove an edge ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from WorkList
    switch n
      case n ∈ Call_p :
        for each d_3 such that ⟨n, d_2⟩ → ⟨s_calledProc(n), d_3⟩ ∈ E# do
          Propagate(⟨s_calledProc(n), d_3⟩ → ⟨s_calledProc(n), d_3⟩)
        od
        for each d_3 such that ⟨n, d_2⟩ → ⟨returnSite(n), d_3⟩ ∈ (E# ∪ SummaryEdge) do
          Propagate(⟨s_p, d_1⟩ → ⟨returnSite(n), d_3⟩)
        od
      end case
      case n = e_p :
        for each c ∈ callers(p) do
          for each d_4, d_5 such that ⟨c, d_4⟩ → ⟨s_p, d_1⟩ ∈ E# and ⟨e_p, d_2⟩ → ⟨returnSite(c), d_5⟩ ∈ E# do
            if ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ ∉ SummaryEdge then
              Insert ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ into SummaryEdge
              for each d_3 such that ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩ ∈ PathEdge do
                Propagate(⟨s_procOf(c), d_3⟩ → ⟨returnSite(c), d_5⟩)
              od
            fi
          od
        od
      end case
      case n ∈ (N_p − Call_p − { e_p }) :
        for each ⟨m, d_3⟩ such that ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E# do
          Propagate(⟨s_p, d_1⟩ → ⟨m, d_3⟩)
        od
      end case
    end switch
  od
end
```

$O(\mathrm{ED}^3)$

Tabulation Algorithm

```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate(G#_IP)
begin
  Let (N#, E#) = G#_IP
  PathEdge := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  WorkList := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  SummaryEdge := ∅
  ForwardTabulateSLRPs()
  for each n ∈ N* do
    X_n := { d_2 ∈ D | ∃ d_1 ∈ (D ∪ { 0 }) such that ⟨s_procOf(n), d_1⟩ → ⟨n, d_2⟩ ∈ PathEdge }
  od
end

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

procedure ForwardTabulateSLRPs()
begin
  while WorkList ≠ ∅ do
    Select and remove an edge ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from WorkList
    switch n

      case n ∈ Call_p :
        for each d_3 such that ⟨n, d_2⟩ → ⟨s_calledProc(n), d_3⟩ ∈ E# do
          Propagate(⟨s_calledProc(n), d_3⟩ → ⟨s_calledProc(n), d_3⟩)
        od
        for each d_3 such that ⟨n, d_2⟩ → ⟨returnSite(n), d_3⟩ ∈ (E# ∪ SummaryEdge) do
          Propagate(⟨s_p, d_1⟩ → ⟨returnSite(n), d_3⟩)
        od
      end case

      case n = e_p :
        for each c ∈ callers(p) do
          for each d_4, d_5 such that ⟨c, d_4⟩ → ⟨s_p, d_1⟩ ∈ E# and ⟨e_p, d_2⟩ → ⟨returnSite(c), d_5⟩ ∈ E# do
            if ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ ∉ SummaryEdge then
              Insert ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ into SummaryEdge
              for each d_3 such that ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩ ∈ PathEdge do
                Propagate(⟨s_procOf(c), d_3⟩ → ⟨returnSite(c), d_5⟩)
              od
            fi
          od
        od
      end case

      case n ∈ (N_p − Call_p − { e_p }) :
        for each ⟨m, d_3⟩ such that ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E# do
          Propagate(⟨s_p, d_1⟩ → ⟨m, d_3⟩)
        od
      end case

    end switch
  od
end
```

$O(\mathrm{ED}^3)$

No time to cover the whole algorithm

Tabulation Algorithm

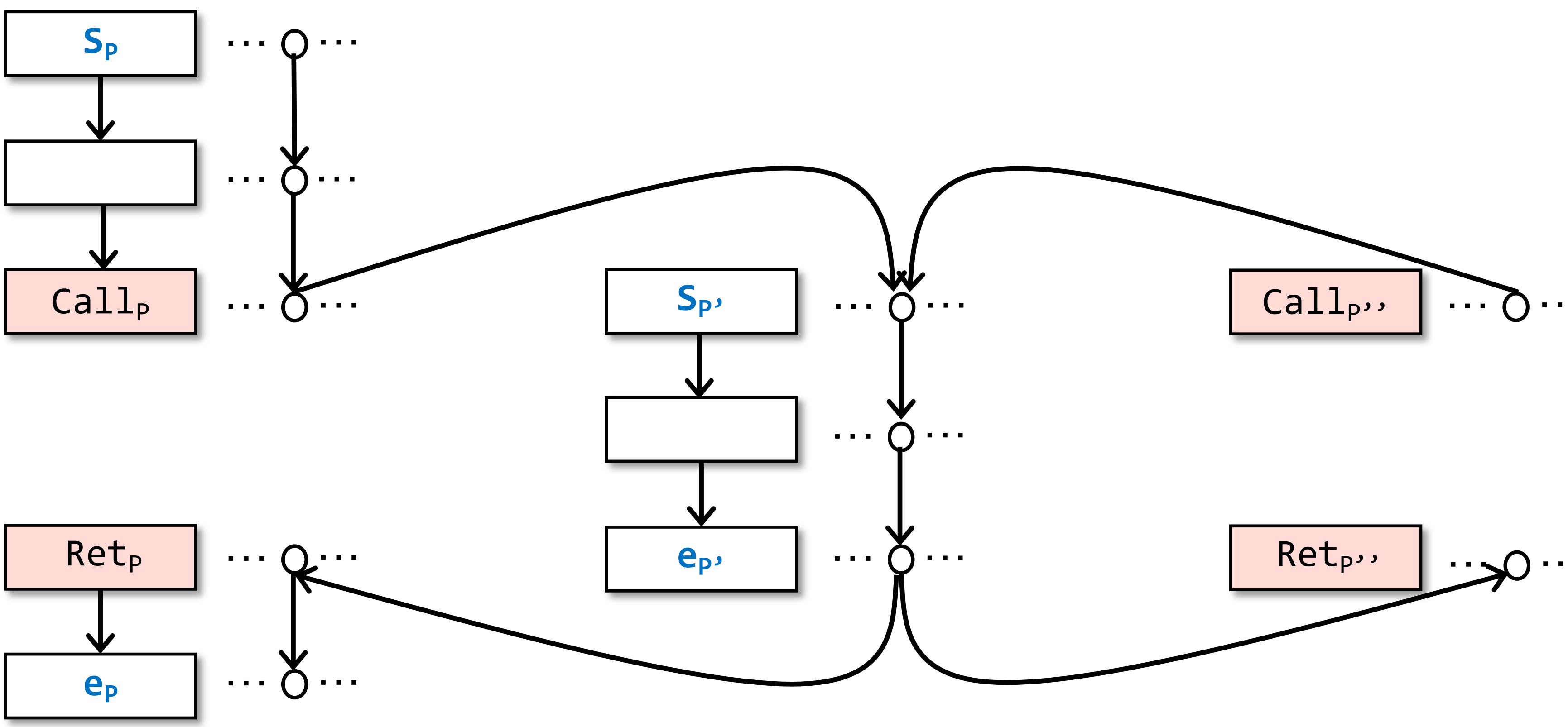
```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate(G#_IP)
begin
  Let (N#, E#) = G#_IP
  PathEdge := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  WorkList := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  SummaryEdge := ∅
  ForwardTabulateSLRPs()
  for each n ∈ N* do
    X_n := { d_2 ∈ D | ∃ d_1 ∈ (D ∪ { 0 }) such that ⟨s_procOf(n), d_1⟩ → ⟨n, d_2⟩ ∈ PathEdge }
  od
end

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

procedure ForwardTabulateSLRPs()
begin
  while WorkList ≠ ∅ do
    Select and remove an edge ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from WorkList
    switch n

      case n ∈ Call_p :
        for each d_3 such that ⟨n, d_2⟩ → ⟨s_calledProc(n), d_3⟩ ∈ E# do
          Propagate(⟨s_calledProc(n), d_3⟩ → ⟨s_calledProc(n), d_3⟩)
        od
        for each d_3 such that ⟨n, d_2⟩ → ⟨returnSite(n), d_3⟩ ∈ (E# ∪ SummaryEdge) do
          Propagate(⟨s_p, d_1⟩ → ⟨returnSite(n), d_3⟩)
        od
      end case

      case n = e_p :
        for each c ∈ callers(p) do
          for each d_4, d_5 such that ⟨c, d_4⟩ → ⟨s_p, d_1⟩ ∈ E# and ⟨e_p, d_2⟩ → ⟨returnSite(c), d_5⟩ ∈ E# do
            if ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ ∉ SummaryEdge then
              Insert ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ into SummaryEdge
              for each d_3 such that ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩ ∈ PathEdge do
                Propagate(⟨s_procOf(c), d_3⟩ → ⟨returnSite(c), d_5⟩)
              od
            fi
          od
        od
      end case

      case n ∈ (N_p − Call_p − { e_p }) :
        for each ⟨m, d_3⟩ such that ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E# do
          Propagate(⟨s_p, d_1⟩ → ⟨m, d_3⟩)
        od
      end case

    end switch
  od
end
```

$$O(\mathrm{ED}^3)$$

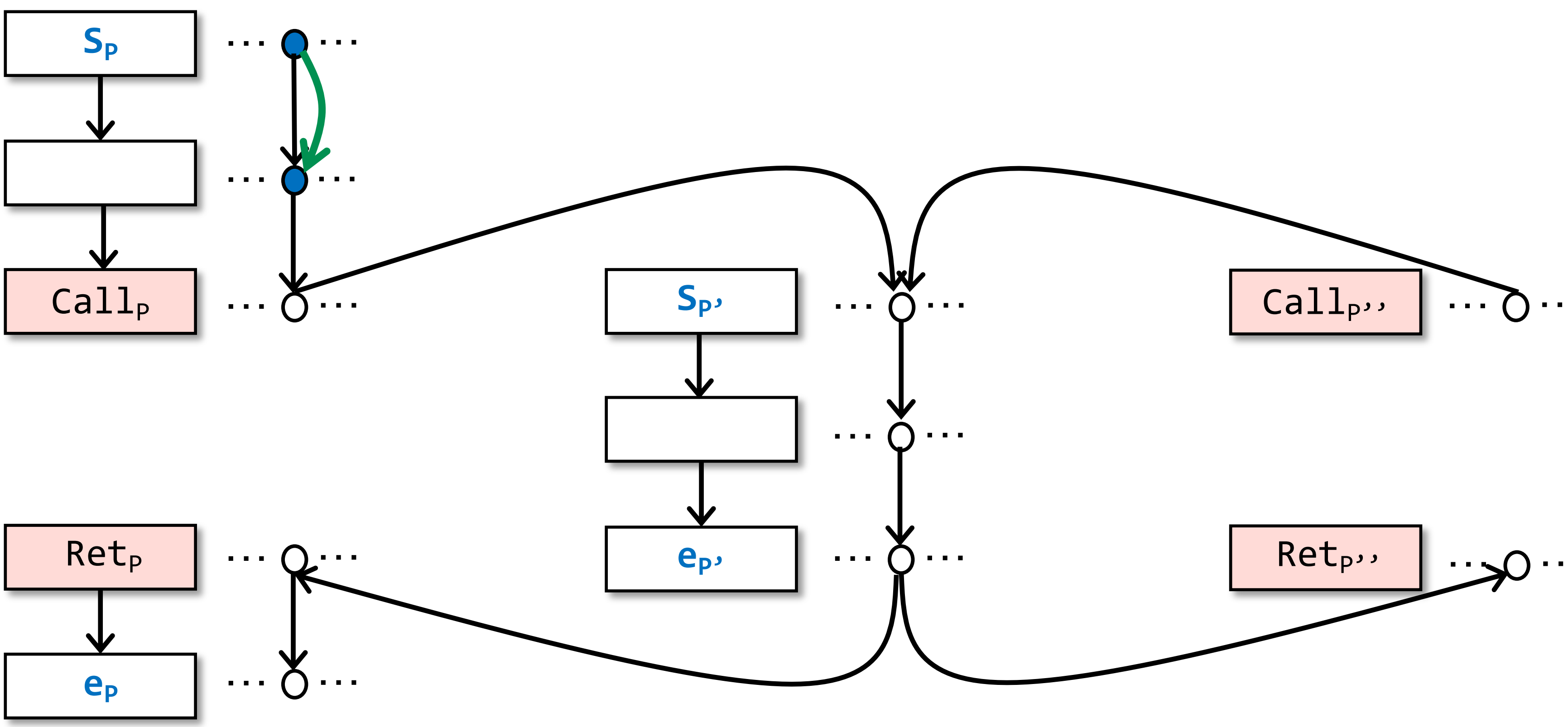
No time to cover the whole algorithm

But we will introduce its core working mechanism by a simple example

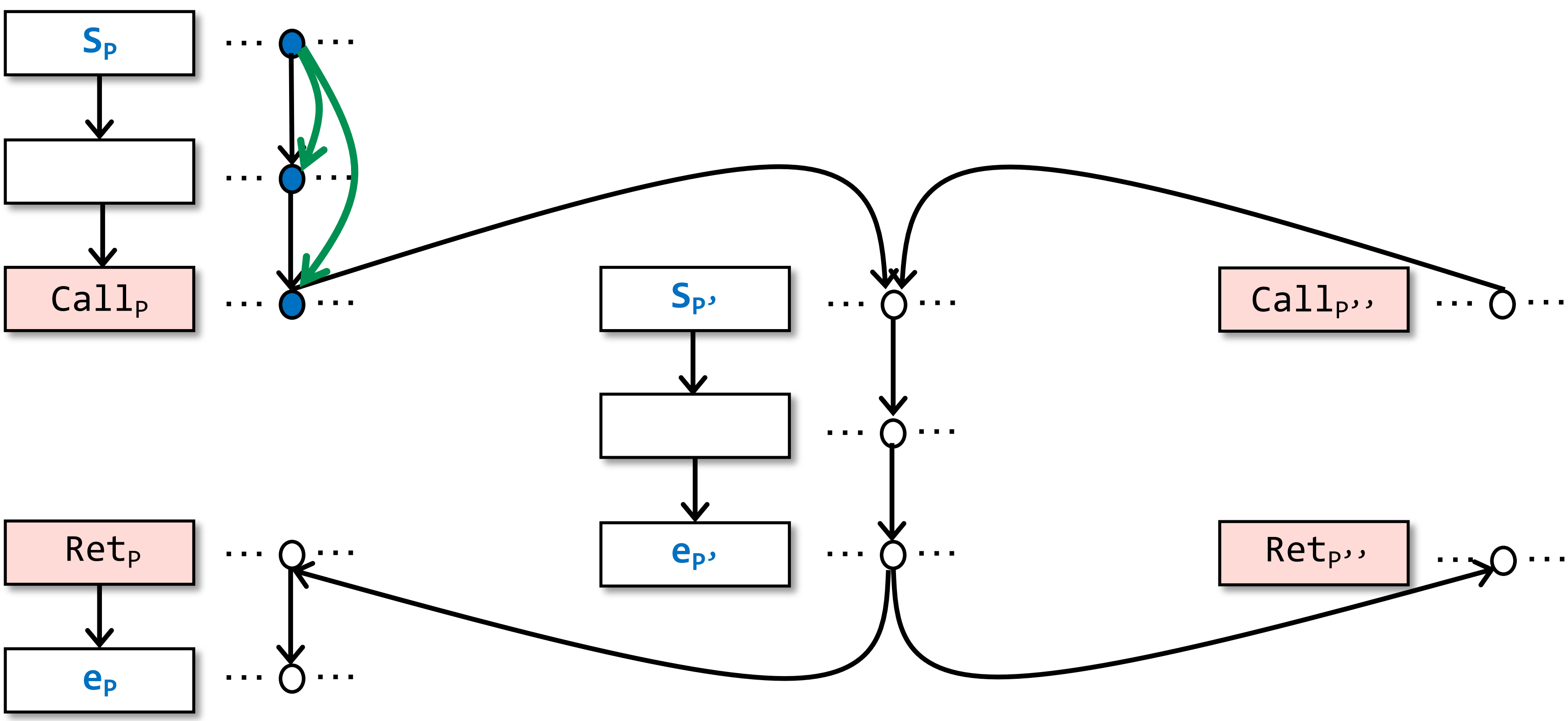
Core Working Mechanism of Tabulation Algorithm



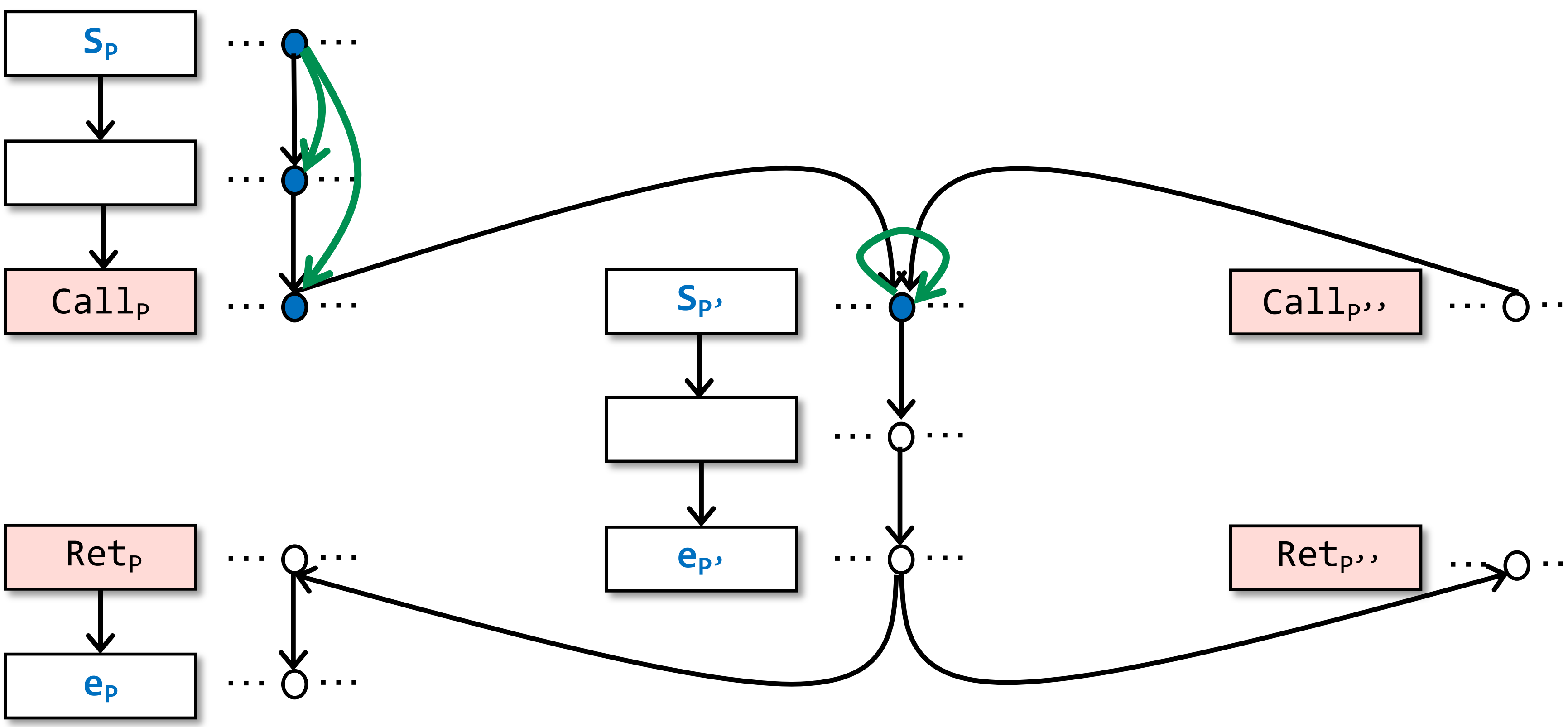
Core Working Mechanism of Tabulation Algorithm



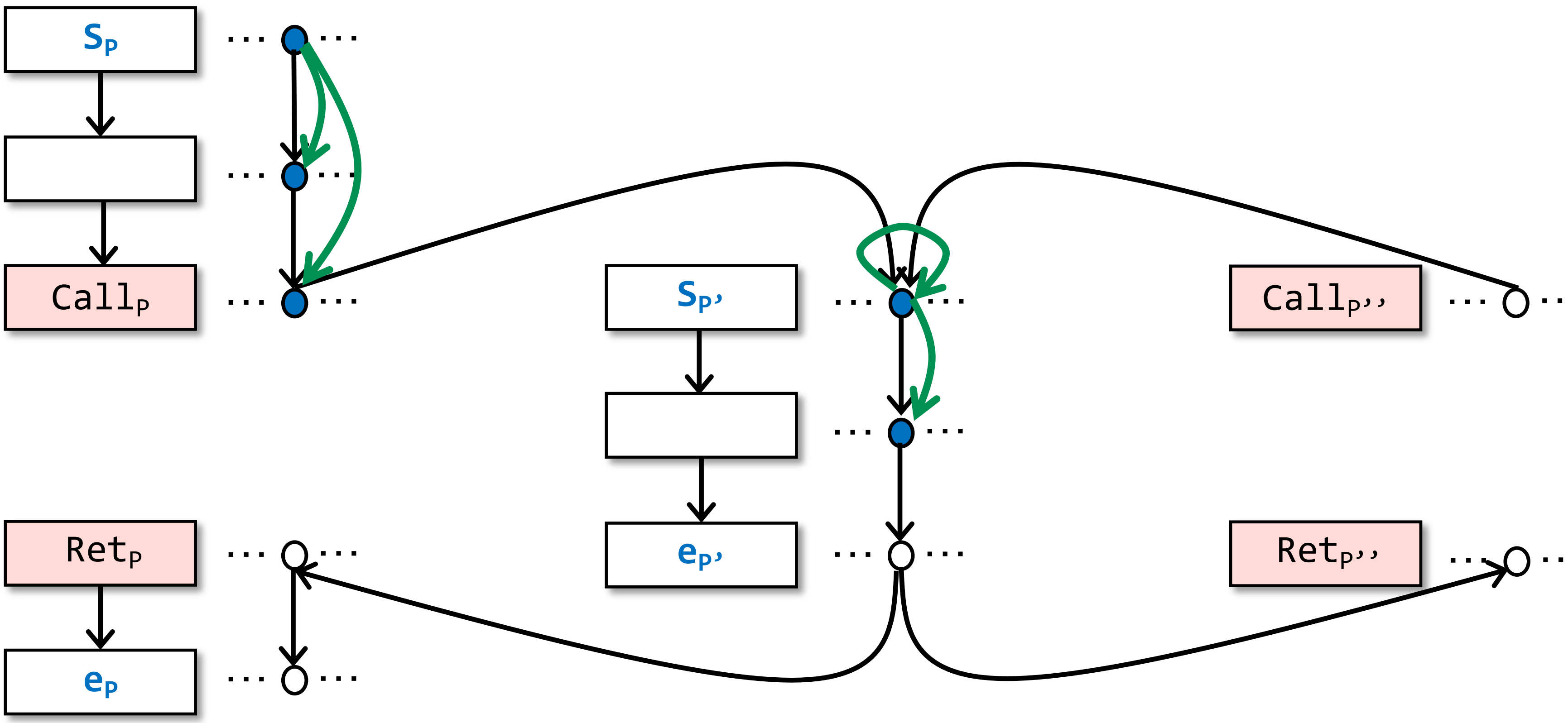
Core Working Mechanism of Tabulation Algorithm



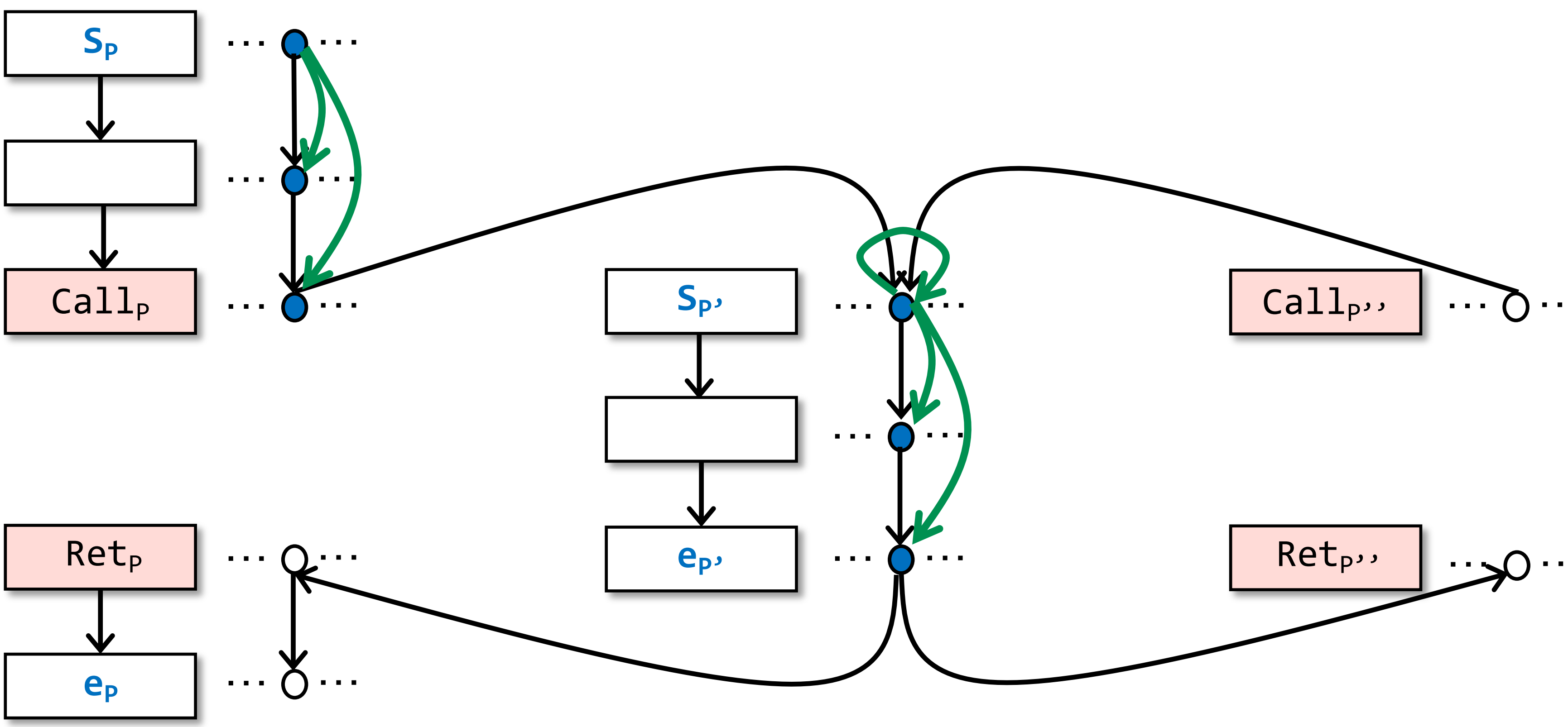
Core Working Mechanism of Tabulation Algorithm



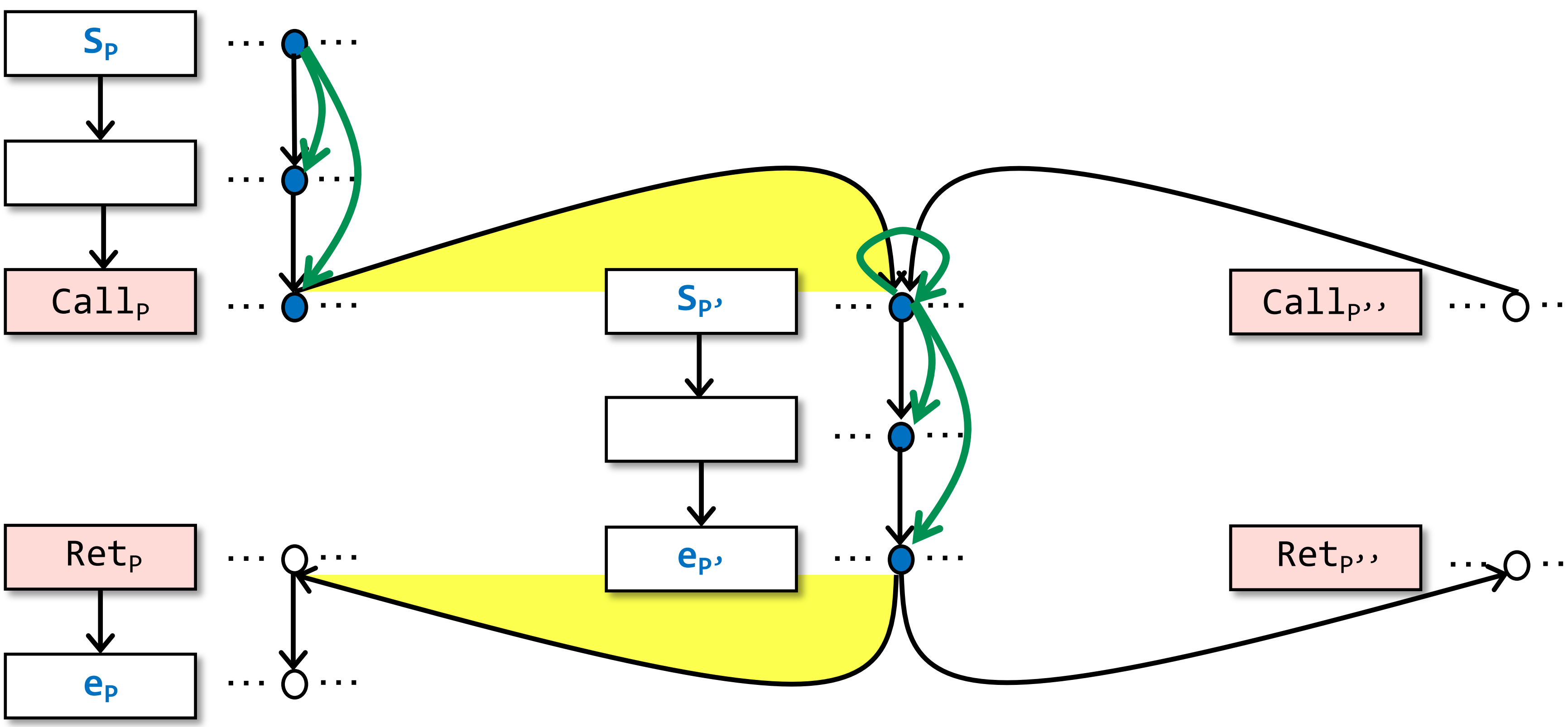
Core Working Mechanism of Tabulation Algorithm



Core Working Mechanism of Tabulation Algorithm



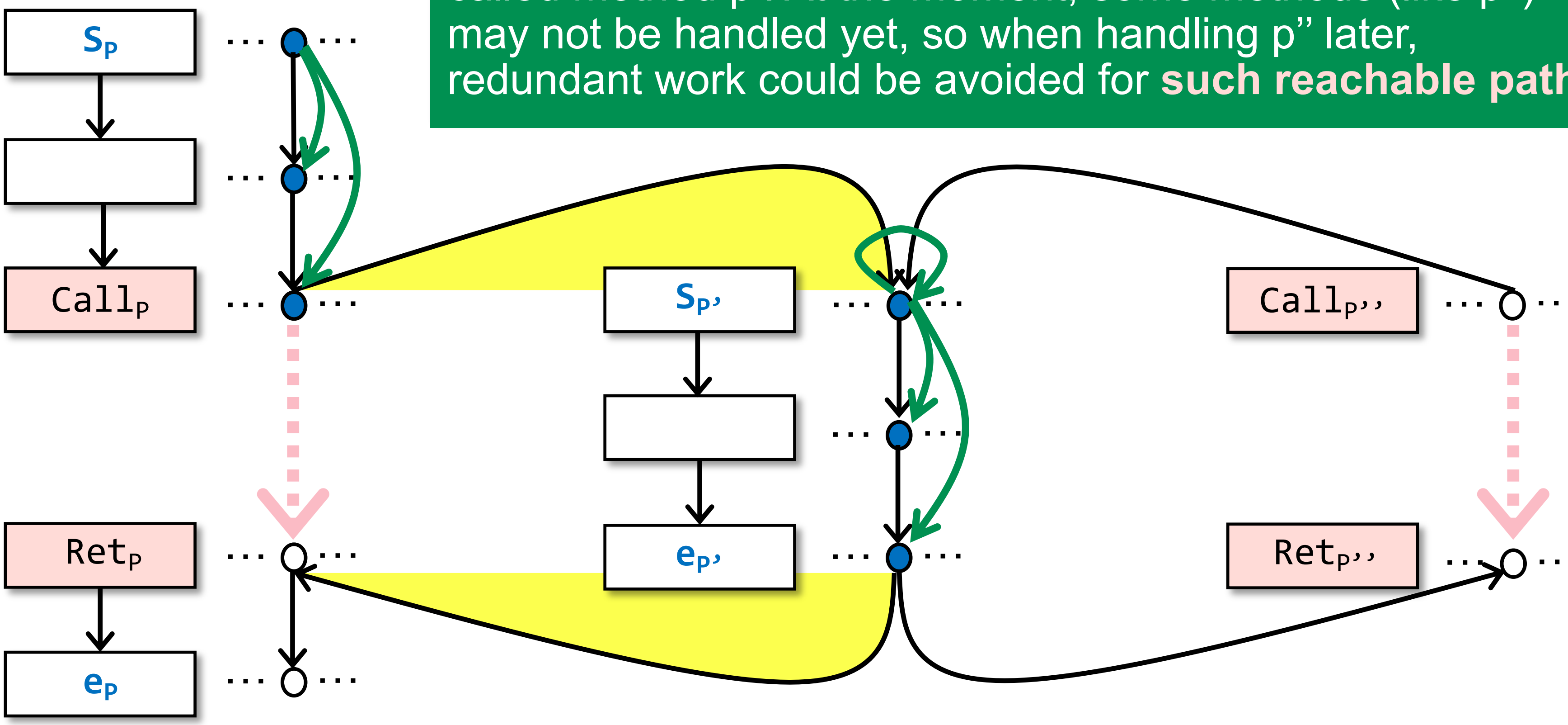
Core Working Mechanism of Tabulation Algorithm



When handling each exit node ($e_{p'}$), call-to-return matching begins: find out the call-sites calling p' ($Call_p$, $Call_{p''}$) and then find out their corresponding return-sites (Ret_p, $Ret_{p''}$).

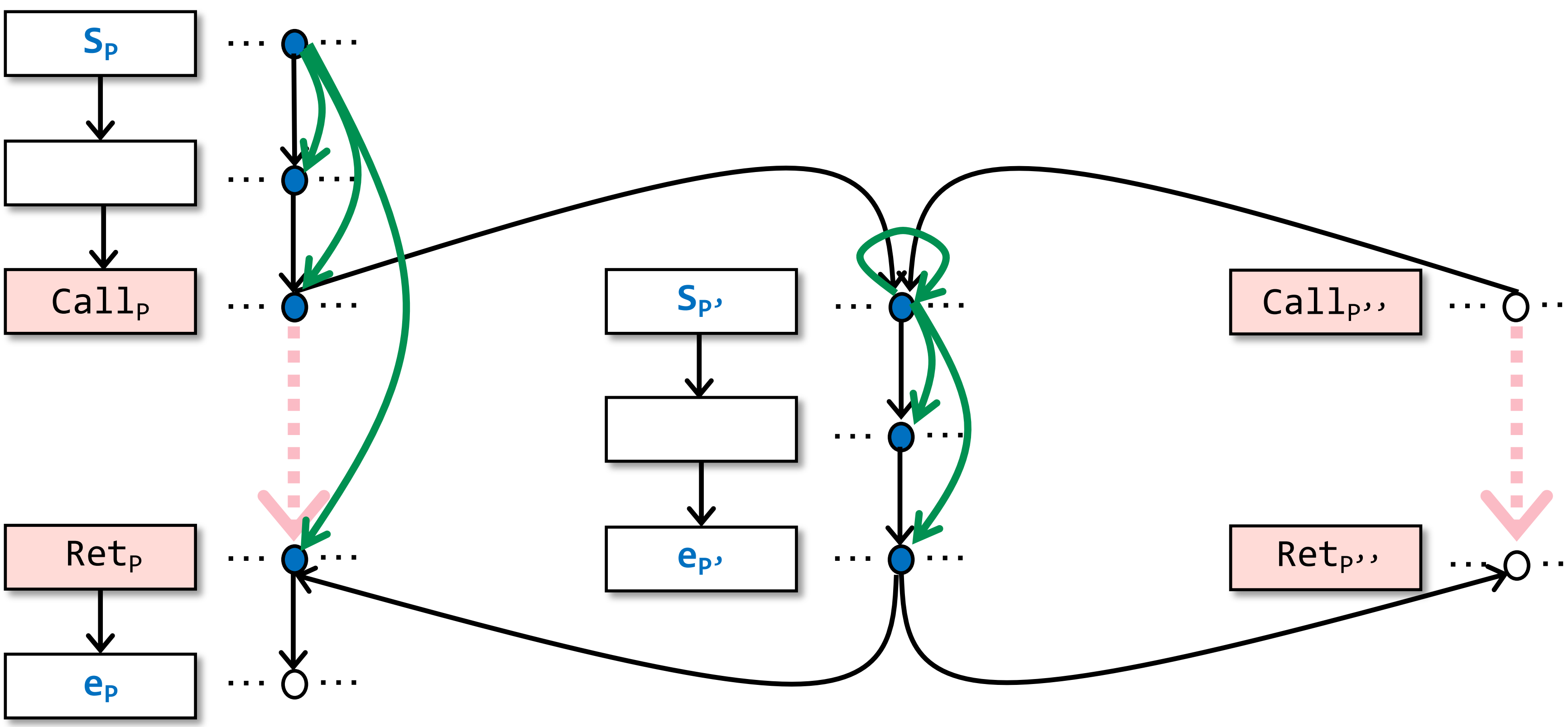
Core Working Mechanism of Tabulation Algorithm

Actually, here **a summary edge** from <Call,d_m> to <Ret,d_n> is added to indicate that d_n is **reachable** from d_m through the called method p'. At the moment, some methods (like p'') may not be handled yet, so when handling p'' later, redundant work could be avoided for **such reachable path**.

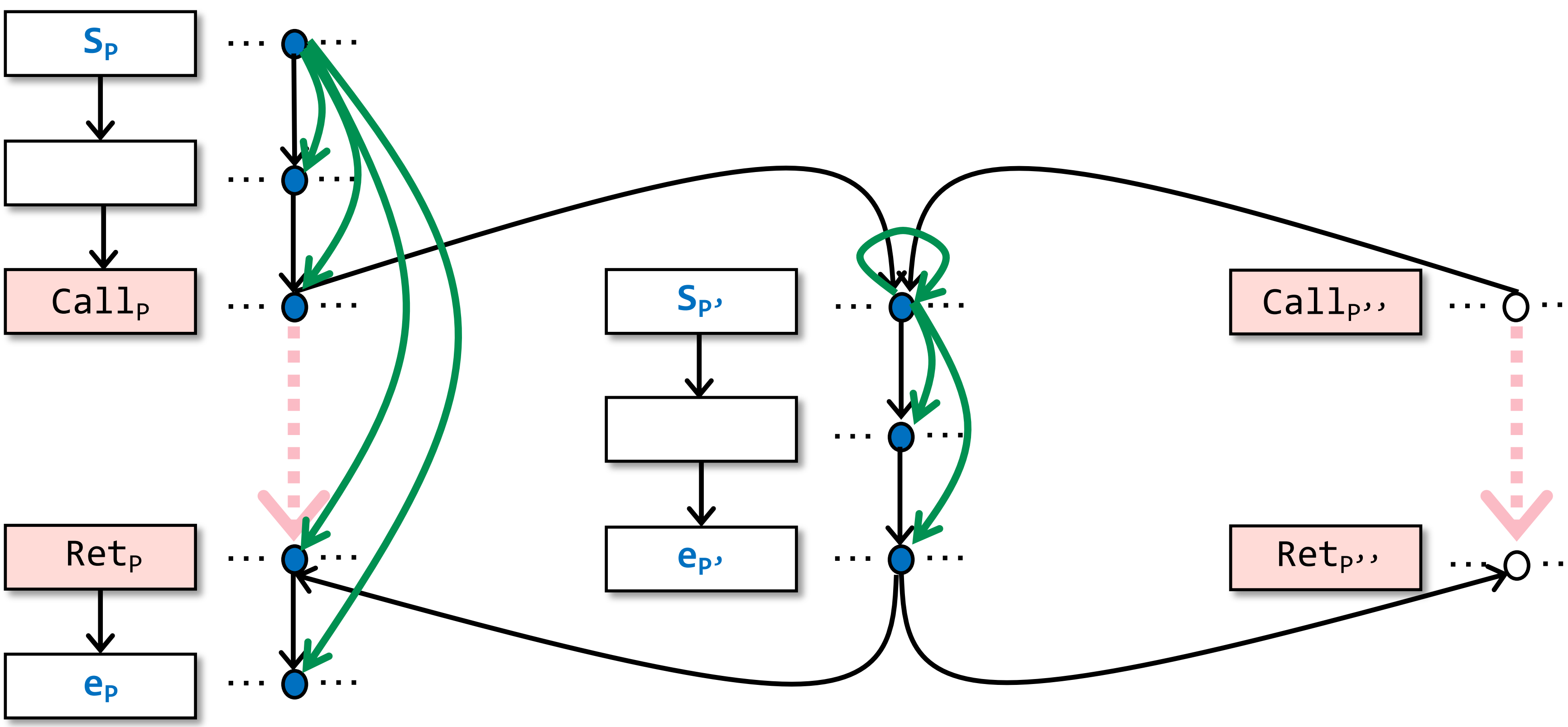


When handling each exit node ($e_{p'}$), call-to-return matching begins: find out the call-sites calling p' ($Call_p$, $Call_{p''}$) and then find out their corresponding return-sites (Ret_p, $Ret_{p''}$).

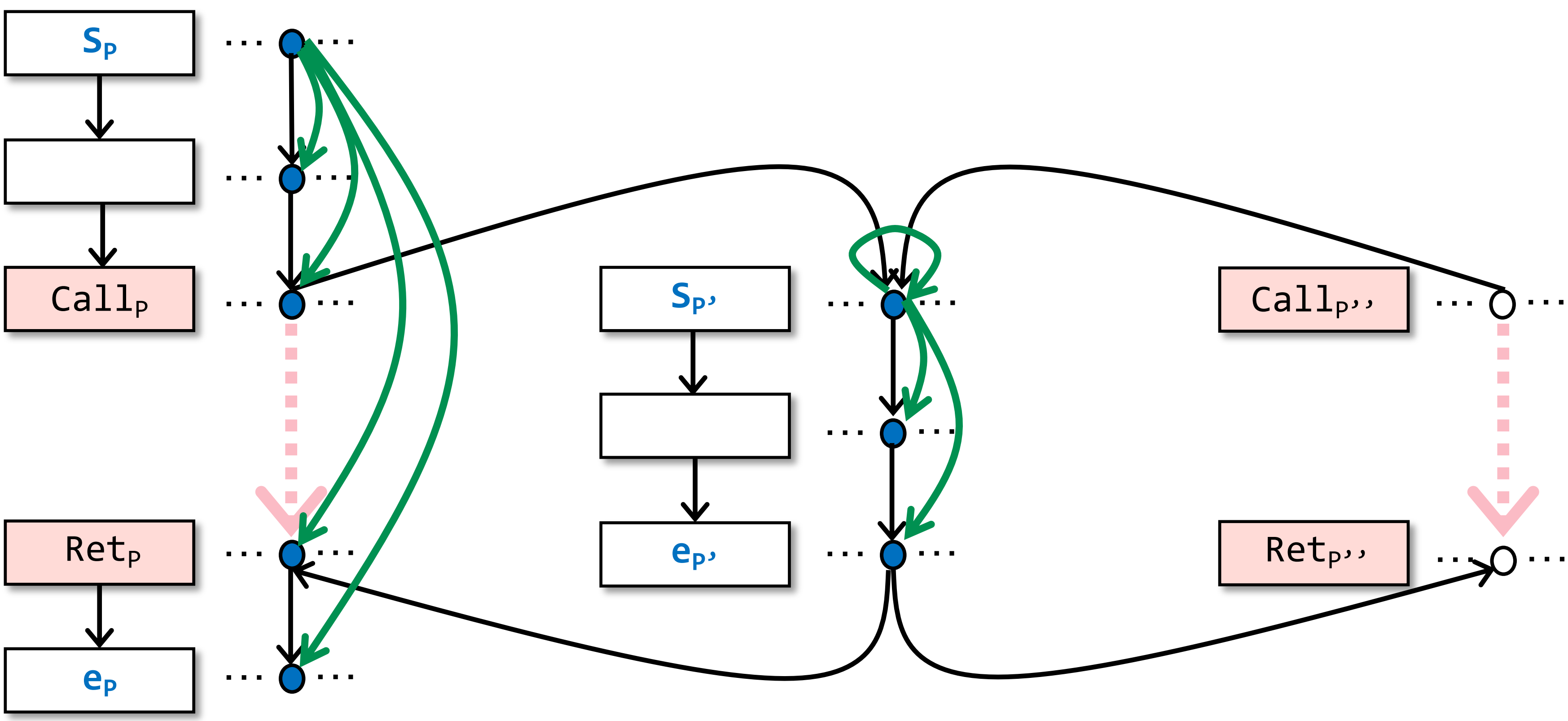
Core Working Mechanism of Tabulation Algorithm



Core Working Mechanism of Tabulation Algorithm



Core Working Mechanism of Tabulation Algorithm



When a data fact (at node n) d's circle is turned to blue, it means that <n, d> is reachable from <S_{main}, 0>

Understanding the Distributivity of IFDS

Understanding the Distributivity of IFDS

- Can we do constant propagation using IFDS?

  Constant propagation has infinite domain, but what if we only deal with finite constant values? Can we still do it using IFDS?

- Can we do pointer analysis using IFDS?

Understanding the Distributivity of IFDS

- Can we do constant propagation using IFDS? No

  Constant propagation has infinite domain, but what if we only deal with finite constant values? Can we still do it using IFDS?

- Can we do pointer analysis using IFDS? No

Understanding the Distributivity of IFDS

- Distributivity

  $F(x \wedge y) = F(x) \wedge F(y)$

- Constant Propagation

  $\boxed{z = x + y}$

  x y z

  z's value depends on both y's and x's

Understanding the Distributivity of IFDS

- Distributivity

$F(x \wedge y) = F(x) \wedge F(y)$

Each flow function in IFDS handles one input data fact per time

$\boxed{z = x + y}$

x y z

O O O

z's value depends on both y's and x's

Understanding the Distributivity of IFDS

- Distributivity

$$F(x \wedge y) = F(x) \wedge F(y)$$

Each flow function in IFDS handles one input data fact per time

z = x + y

x y z

Each representation relation indicates "if x exists, then …", "if y exists then …" But when we need "if both x and y exist", how to draw the representation relation?

Understanding the Distributivity of IFDS

Each flow function in IFDS handles one input data fact per time

- Distributivity

For constant propagation, we cannot define F if we only know x's (or y's) value

z = x + y

x y z

O O O

Each representation relation indicates "if x exists, then …", "if y exists then …" But when we need "if both x and y exist", how to draw the representation relation?

Understanding the Distributivity of IFDS

- Distributivity

Each flow function in IFDS handles one input data fact per time

For constant propagation, we cannot define F if we only know x's (or y's) value

z = x + y

x y z

O O O

Each representation relation indicates "if x exists, then …", "if y exists then …" But when we need "if both x and y exist", how to draw the representation relation?

Given a statement S, besides S itself, if we need to consider **multiple** input data facts to create correct outputs, then the analysis is not distributive and should not be expressed in IFDS.

In IFDS, each data fact (circle) and its propagation (edges) could be handled **independently**, and doing so will not affect the correctness of the final results.

Understanding the Distributivity of IFDS

- Distributivity

Each flow function in IFDS handles one input data fact per time

For constant propagation, we cannot define F if we only know x's (or y's) value

z = x + y

x y z

O O O

Each representation relation indicates "if x exists, then …", "if y exists then …" But when we need "if both x and y exist", how to draw the representation relation?

A simple rule to determine whether your analysis could be expressed in IFDS

Given a statement S, besides S itself, if we need to consider **multiple** input data facts to create correct outputs, then the analysis is not distributive and should not be expressed in IFDS.

In IFDS, each data fact (circle) and its propagation (edges) could be handled **independently**, and doing so will not affect the correctness of the final results.

Understanding the Distributivity of IFDS

- Distributivity

Each flow function in IFDS handles one input data fact per time

For constant propagation, we cannot define F if we only know x's (or y's) value

z = x + y

x y z

O O O

Each representation relation indicates "if x exists, then …", "if y exists then …" But when we need "if both x and y exist", how to draw the representation relation?

A simple rule to determine whether your analysis could be expressed in IFDS

Given a statement S, besides S itself, if we need to consider **multiple** input data facts to create correct outputs, then the analysis is not distributive and should not be expressed in IFDS.

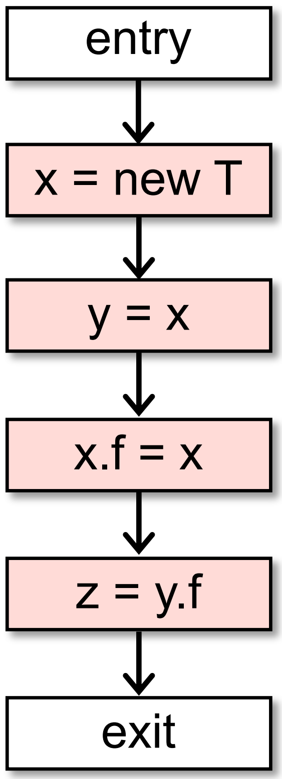
In IFDS, each data fact (circle) and its propagation (edges) could be handled **independently**, and doing so will not affect the correctness of the final results.

Regardless of the infinite domain issue, think about whether we could do *linear constant propagation*, e,g., y = 2x + 3, or
copy constant propagation, e.g., x = 2, y = x, using IFDS-style analysis?

Understanding the Distributivity of IFDS

For simplicity, assume we know the program only contains these four statements when designing flow functions

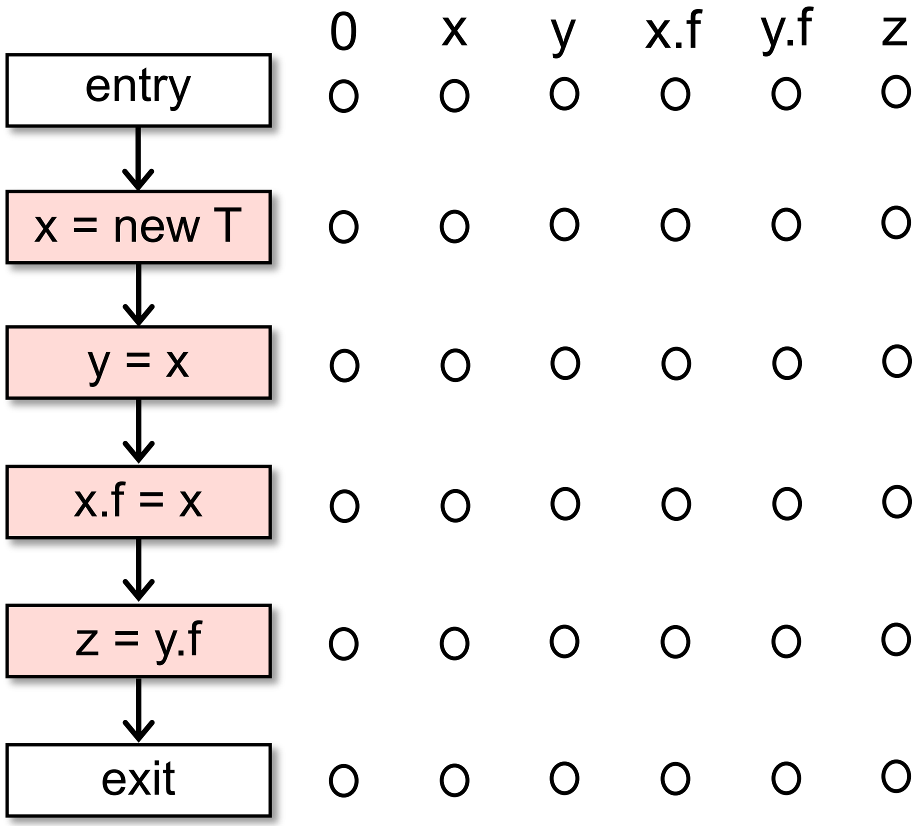
- Pointer Analysis



Understanding the Distributivity of IFDS

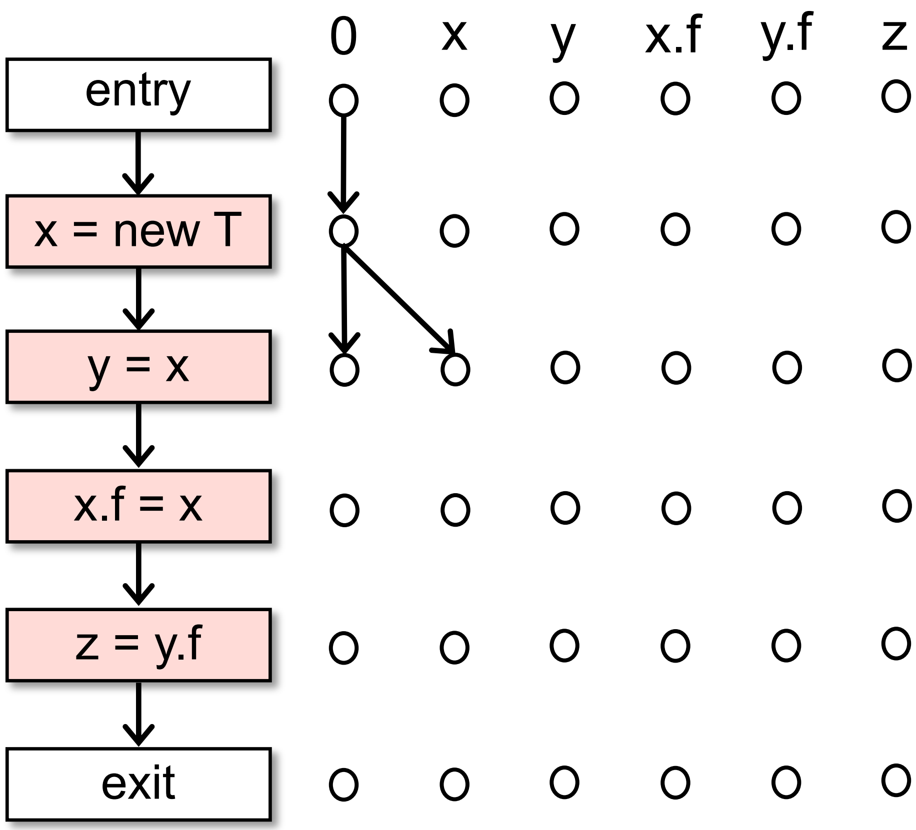
For simplicity, assume we know the program only contains these four statements when designing flow functions

- Pointer Analysis



Understanding the Distributivity of IFDS

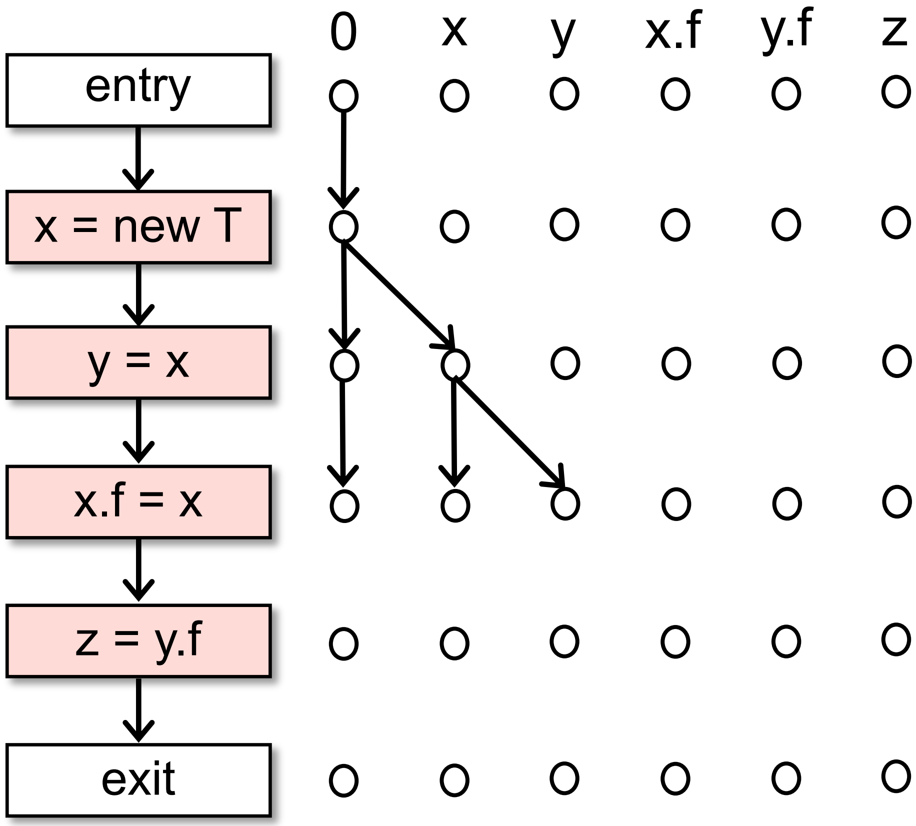
- Pointer Analysis



For simplicity, assume we know the program only contains these four statements when designing flow functions

Understanding the Distributivity of IFDS

- Pointer Analysis

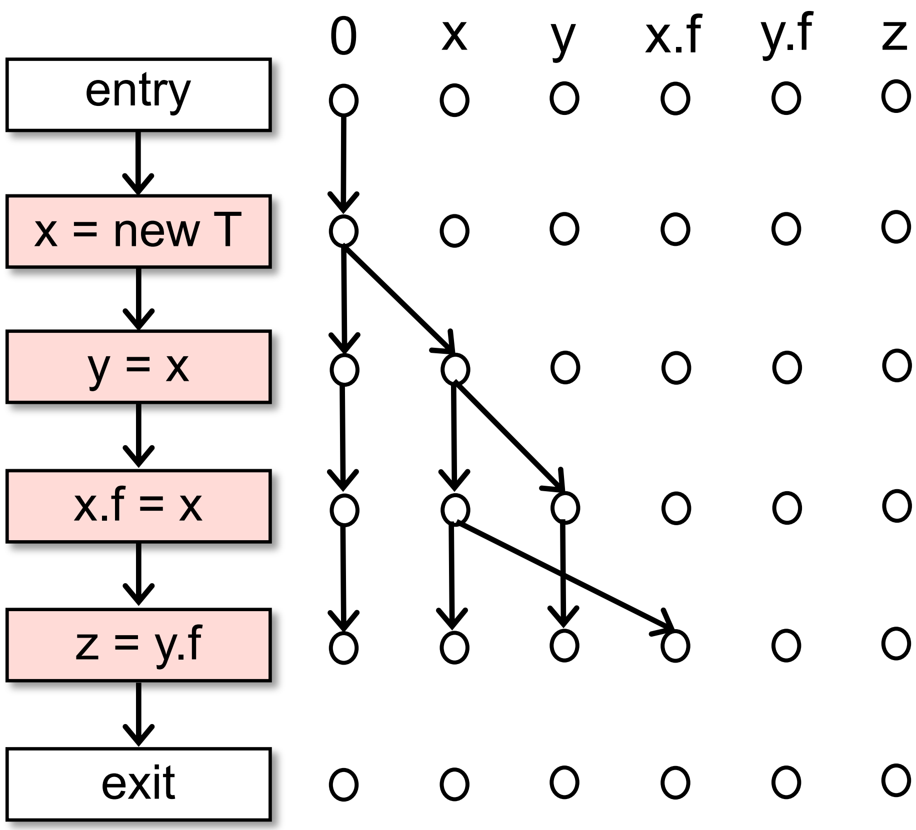


For simplicity, assume we know the program only contains these four statements when designing flow functions

Understanding the Distributivity of IFDS

For simplicity, assume we know the program only contains these four statements when designing flow functions

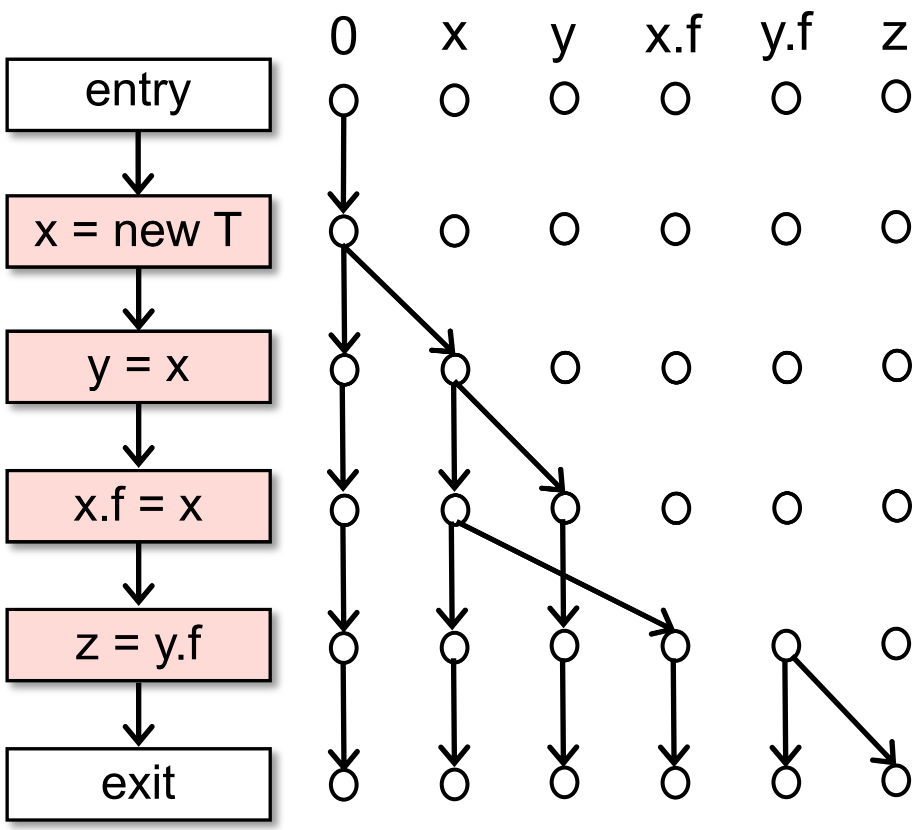
- Pointer Analysis



Understanding the Distributivity of IFDS

For simplicity, assume we know the program only contains these four statements when designing flow functions

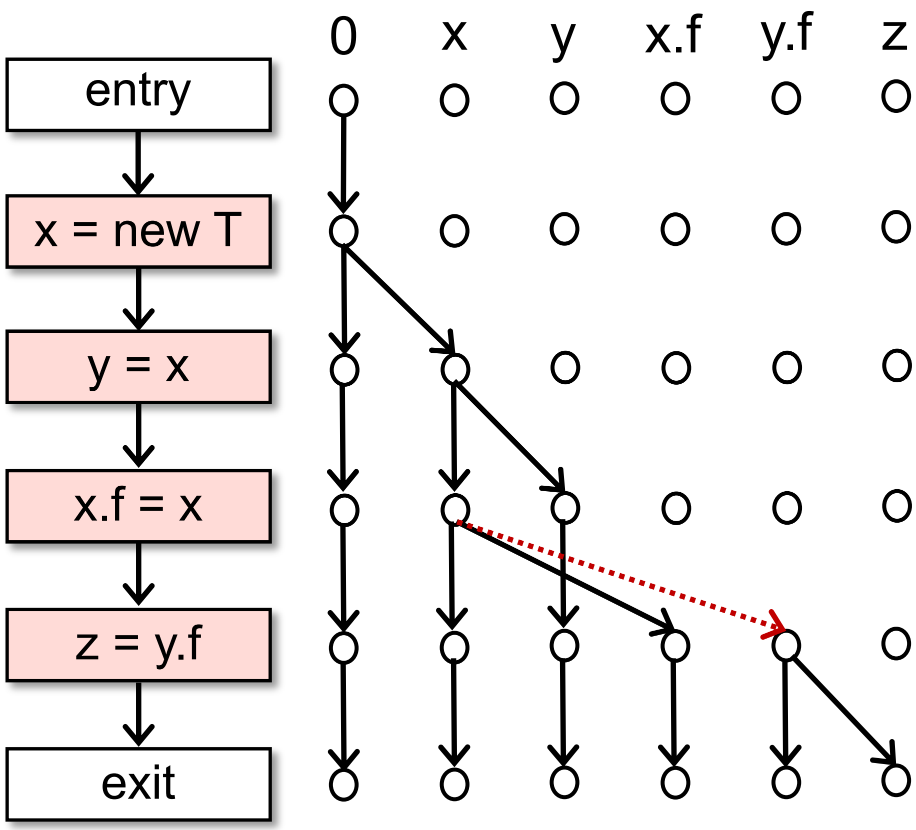
- Pointer Analysis



Understanding the Distributivity of IFDS

For simplicity, assume we know the program only contains these four statements when designing flow functions

- Pointer Analysis

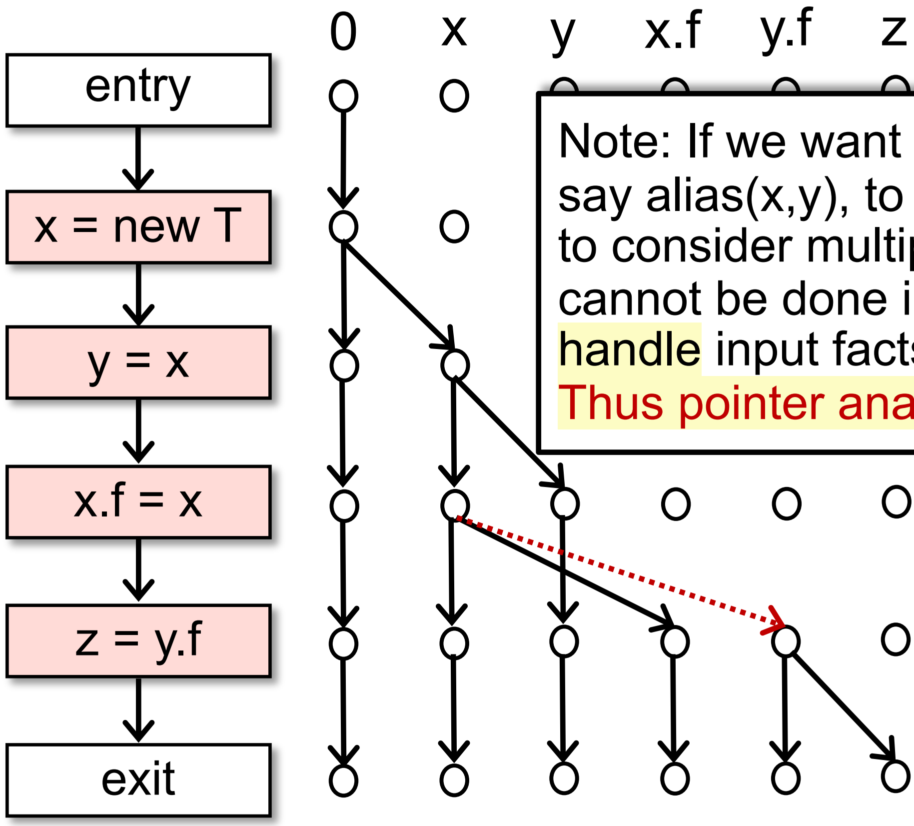


z and y.f should have pointed to object [new T]. However, flow function's input data facts lack of the alias information, alias(x,y), alias(x.f,y.f), and we need alias information to produce correct outputs.

Understanding the Distributivity of IFDS

For simplicity, assume we know the program only contains these four statements when designing flow functions

- Pointer Analysis



Note: If we want to obtain alias information in IFDS, say alias(x,y), to produce correct outputs, we need to consider multiple input data facts, x and y, which cannot be done in *standard* IFDS as flow functions handle input facts independently (one fact per time). Thus pointer analysis is non-distributive.

z and y.f should have pointed to object [new T]. However, flow function's input data facts lack of the alias information, alias(x,y), alias(x.f,y.f), and we need alias information to produce correct outputs.

Contents

The X You Need To Understand in This Lecture

- Understand CFL-Reachability
- Understand the basic idea of IFDS
- Understand what problems can be solved by IFDS

软件分析

南京大学
计算机科学与技术系
程序设计语言与
静态分析研究组
李樾
谭添