Static Program Analysis
CFL-Reachability and IFDS
Nanjing University
Yue Li
2020
Contents

1. Feasible and Realizable Paths
2. CFL-Reachability
3. Overview of IFDS
4. Supergraph and Flow Functions
5. Exploded Supergraph and Tabulation Algorithm
6. Understanding the Distributivity of IFDS
Control Flow Graph of a method in JDK

Yue Li @ Nanjing University
Control Flow Graph of a method in JDK

Are all the paths executable?

Yue Li @ Nanjing University
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?

Yue Li @ Nanjing University
Control Flow Graph of a method in JDK

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.
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

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.

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

Yue Li @ Nanjing University
Control Flow Graph of a method in JDK

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.

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

Yue Li @ Nanjing University
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;
}

x = 18, 30, -1
y = 18, 30, -1
main() {
    x = foo(18);
    y = foo(30);
}

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

Enter main
Call foo(18)
    Enter foo
        r = age
        return r
    Exit foo
    x = Return foo
        y = Return foo
            x = 18, 30, -1
            y = 18, 30, -1
            Exit foo
        return r
    x = 18, 30, -1
    y = 18, 30, -1

Inevitable
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:
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.

• Our goal is to recognize realizable paths so that we could avoid polluting analysis results along unrealizable paths.
main() {
    x = foo(18);
    
    y = foo(30);
}

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

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

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 \( \alpha \) 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 \( aSb/\varepsilon \) 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 \( \text{i} \) is balanced by a preceding left parenthesis \( \text{i} \), but the converse needs not hold.
- For each call site \text{i}, label its call edge \( \text{i} \) and return edge \( \text{i} \).
- Label all other edges with symbol “e”.

A path is a **realizable path** iff the path’s word is in the language \( L(\text{realizable}) \).
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 \( L(\text{realizable}) \)

\[ \text{realizable } \Rightarrow \text{matched realizable} \]

\[ e.g., \( (1 (2 e )_2 )_1 (3 \) \]
CFL-Reachability

Partially **Balanced-Parenthesis** Problem via CFL

- Every right parenthesis “\( \text{)}_i \)” 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 \( L(\text{realizable}) \).

\[
\text{realizable} \rightarrow \text{matched realizable} \\
\rightarrow (i)
\]

\[e.g., (1 \text{ e}_2 \text{ e}_2)_1 (3)\]
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 \( L(\text{realizable}) \)

\[
\text{realizable} \rightarrow \text{matched realizable} \\
\rightarrow (i)
\]

- e.g., \( (1 \ (2 \ e \ )_2)_1 \ (3 \ (4 \)
- e.g., \( (1 \ (2 \ e \ )_2)_1 \ (3 \ (4_2)

Yue Li @ Nanjing University
CFL-Reachability

**Partially Balanced-Parenthesis Problem via CFL**

- Every right parenthesis \( \text{)}_i \) 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 \( \text{)}_i \).
- Label all other edges with symbol “e”.

A path is a **realizable path** iff the path’ word is in the language \( L(\text{realizable}) \):

\[
\text{realizable} \rightarrow \text{matched realizable} \\
\rightarrow (i \text{ realizable})
\]

- 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 \(L(\text{realizable})\).

\[
\text{realizable} \rightarrow \text{matched realizable} \rightarrow \text{realizable} \rightarrow \varepsilon
\]

\[\text{e.g., } (1 \ 2 \ e \ 2)_1 (3)\]

\[\text{e.g., } (1 \ 2 \ e \ 2)_1 (3 \ 4)\]
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 \( L(\text{realizable}) \)

\[
\begin{align*}
\text{realizable} & \rightarrow \text{matched realizable} \\
& \rightarrow (i \text{ realizable}) \\
& \rightarrow \varepsilon \\
\text{matched} & \rightarrow (i \text{ matched })_i \\
\end{align*}
\]

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 \( L(realizable) \)

\[
\text{realizable} \rightarrow \text{matched realizable} \\
\rightarrow (i \text{ realizable}) \\
\rightarrow \epsilon
\]

\[
\text{matched} \rightarrow (i \text{ matched } )_i \\
\rightarrow e \\
\rightarrow \epsilon
\]

**Example**:
- \( (1 (2 e )_2 )_1 (3 \)
- \( (1 (2 e )_2 )_1 (3 (4 \)
CFL-Reachability

Partially **Balanced-Parenthesis** Problem via CFL

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

A path is a **realizable path** iff the path’ word is in the language \( L(\text{realizable}) \)

\[
\text{realizable} \rightarrow \text{matched realizable} \\
\rightarrow (i \text{ realizable}) \\
\rightarrow \varepsilon
\]

\[
\text{matched} \rightarrow (i \text{ matched})_i \\
\rightarrow e \\
\rightarrow \varepsilon
\]
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’s word is in the language \( L(\text{realizable}) \).

\[
\begin{align*}
\text{realizable} & \rightarrow \text{matched realizable} \\
& \rightarrow ( \_i \text{ realizable} ) \\
& \rightarrow \varepsilon
\end{align*}
\]

\[
\begin{align*}
\text{matched} & \rightarrow ( \_i \text{ matched} ) \_i \\
& \rightarrow e \\
& \rightarrow \varepsilon \\
& \rightarrow \text{matched matched}
\end{align*}
\]
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 \( L(\text{realizable}) \)

\[
\text{realizable} \rightarrow \text{matched realizable} \\
\quad \rightarrow (i \text{ realizable}) \\
\quad \rightarrow e \\
\text{matched} \rightarrow (i \text{ matched } )_i \\
\quad \rightarrow e \\
\quad \rightarrow e \\
\quad \rightarrow \text{matched matched}
\]

\[
\text{e.g., } (1(2e)_2)_1(3 \\
\text{e.g., } (1(2e)_2)_1(3(4 \\
\text{e.g., } (1(2eee)_2)_1(3(4 \\
\text{e.g., } eee(1(2eee)_2)_1(3(4e}
\]
$L(realizable)$:

realizable $\rightarrow$ matched realizable $\rightarrow$ (i realizable $\rightarrow$ $\varepsilon$ $\rightarrow$ e $\rightarrow$ matched matched)
$L(\text{realizable})$:

realizable $\Rightarrow$ matched realizable
$t_i$ realizable
$\Rightarrow$ $e$

matched $\Rightarrow$ (matched $i$)
$e$
$\Rightarrow$ $e$
$\Rightarrow$ matched matched

$e(1\text{eee})_1e \in L(\text{realizable})$
\( L(\text{realizable}):\)

realizable \(\Rightarrow\) matched realizable
\(\Rightarrow\) \(\varepsilon\)

matched \(\Rightarrow\) \(\varepsilon\)
\(\Rightarrow\) matched matched

\( e(1\varepsilon e)_1 e \in L(\text{realizable})\)

\( e(1\varepsilon e)_1 e(2\varepsilon e)_1 \notin L(\text{realizable})\)
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 $p_f^p$, is a composition of flow functions for all edges (sometimes nodes) on $p$.

\[ p_f^p = f_n \circ \ldots \circ f_2 \circ f_1 \]
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 $p$.

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

Recall

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

For each node $n$, MOP$_n$ provides a “meet-over-all-paths” solution where Paths($\text{start}, n$) 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

\[
pf_p = f_n \circ \ldots \circ f_2 \circ f_1
\]

For each node $n$, $\text{MOP}_n$ provides a “meet-over-all-paths” solution where $\text{Paths}(\text{start}, n)$ denotes the set of paths in CFG from the start node to $n$.

\[
\text{MOP}_n = \bigsqcup_{p \in \text{Paths}(\text{start}, n)} pf_p(\perp)
\]

For each node $n$, $\text{MRP}_n$ provides a “meet-over-all-realizable-paths” solution where $\text{RPaths}(\text{start}, n)$ denotes the set of realizable paths (the path’s word is in the language $L(\text{realizable})$) from the start node to $n$.

\[
\text{MRP}_n = \bigsqcup_{p \in \text{RPaths}(\text{start}, n)} pf_p(\perp)
\]
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 $p$.

\[ pf_p = f_n \circ \ldots \circ f_2 \circ f_1 \]

Recall

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

\[ MOP_n = \bigsqcup_{p \in \text{Paths}(\text{start}, n)} pf_p(\perp) \]

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

\[ MRP_n = \bigsqcup_{p \in \text{RPaths}(\text{start}, n)} pf_p(\perp) \]

\[ MRP_n \subseteq MOP_n \]

Yue Li @ Nanjing University
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 $d \in \text{MRP}_n$, iff there is a realizable path in $G#$ from $s_{\text{main}}$, 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#

Let n be a program point, data fact d \( \in \) MRP\(_n\), iff there is a realizable path in G# from \( s_{main}, 0 \) to \( <n, d> \).

How to understand?
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.
Supergraph

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

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

```c
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 $G_1, G_2, \ldots$ (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 $\text{Call}_p$, and a return-site node $\text{Ret}_p$

```c
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:

```c
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:
- An intraprocedural call-to-return-site edge from $Call^*_p$ to $Ret^*_p$
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

```c
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:

- An intraprocedural \textit{call-to-return-site edge} from $\text{Call}_p$ to $\text{Ret}_p$
- An interprocedural \textit{call-to-start edge} from $\text{Call}_p$ to $s_p$ of the called procedure
- An interprocedural \textit{exit-to-return-site edge} from $e_p$ of the called procedure to $\text{Ret}_p$

```c
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 $n \in N^*$, determine the set of variables that may be uninitialized before execution reaches $n$.

```c
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 $n \in N^*$, determine the set of variables that may be uninitialized before execution reaches $n$. 

$$\lambda e_{param} \cdot e_{body}$$
Design Flow Functions

“Possibly-uninitialized variables”: for each node $n \in N^*$, determine the set of variables that may be uninitialized before execution reaches $n$.

$$\lambda e_{\text{param}} \cdot e_{\text{body}}$$

*e.g., $\lambda x . x+1$*
Design Flow Functions

“Possibly-uninitialized variables”: for each node $n \in N^*$, determine the set of variables that may be uninitialized before execution reaches $n$.

$$\lambda \ e_{param} \cdot e_{body}$$

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

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

Yue Li @ Nanjing University
“Possibly-uninitialized variables”: for each node $n \in N^*$, determine the set of variables that may be uninitialized before execution reaches $n$.

$$\lambda e_{param} \cdot e_{body}$$

*E.g.*, $\lambda x. x+1$

$$(\lambda x. x+1)^3 \Rightarrow 3+1 \Rightarrow 4$$
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

```c
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

```c
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);
  }
}
```

Yue Li @ Nanjing University
Design Flow Functions

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

Yue Li @ Nanjing University
```c
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

```c
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

```c
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

```c
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);
    }
}
```

Yue Li @ Nanjing University
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:

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

- Edge: $0 \rightarrow 0$
- Edge: $0 \rightarrow d_1$
- Edge: $d_1 \rightarrow d_2$
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:

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

- Edge: $0 \rightarrow 0$
- Edge: $0 \rightarrow d_1$
- Edge: $d_1 \rightarrow d_2$

<table>
<thead>
<tr>
<th>0 a b</th>
<th>0 a b</th>
<th>0 a b c</th>
<th>0 a b c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 o o o</td>
<td>0 o o o</td>
<td>0 o o o</td>
<td>0 o o o</td>
</tr>
<tr>
<td>0 0 o 0</td>
<td>0 0 o 0</td>
<td>0 0 o 0</td>
<td>0 0 o 0</td>
</tr>
<tr>
<td>0 o o o</td>
<td>0 o o o</td>
<td>0 o o o</td>
<td>0 o o o</td>
</tr>
</tbody>
</table>

Yue Li - Nanjing University
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:

$$R_f = \{ (0,0) \} \cup \{ (0,y) \mid y \in f(\emptyset) \} \cup \{ (x,y) \mid y \not\in f(\emptyset) \text{ and } y \in f(\{x\}) \}$$

Edges:
- $0 \rightarrow 0$
- $0 \rightarrow d_1$
- $d_1 \rightarrow d_2$

\[
\begin{array}{c|c|c|c}
\lambda S. S & \lambda S. \{a\} & \lambda S. (S-\{a\}) \cup \{b\} & \lambda S. \begin{cases} 
\text{if } a \in S & \text{then } S \cup \{b\} \\
\text{else } S - \{b\} & 
\end{cases} \\
0 \ a \ b & 0 \ a \ b & 0 \ a \ b \ c & 0 \ a \ b \ c \\
0 \ 0 \ 0 & 0 \ 0 \ 0 & 0 \ 0 \ 0 \ 0 & 0 \ 0 \ 0 \ 0 \\
\end{array}
\]

Yue Li @ Nanjing University
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:

$$R_f = \{ (0,0) \} \cup \{ (0,y) | y \in f(\emptyset) \} \cup \{ (x,y) | y \notin f(\emptyset) \text{ and } y \in f(\{x\}) \}$$

**Edge**:
- $0 \rightarrow 0$
- $0 \rightarrow d_1$
- $d_1 \rightarrow d_2$

### Example

<table>
<thead>
<tr>
<th>$\lambda S. S$</th>
<th>$\lambda S. {a}$</th>
<th>$\lambda S. (S-{a}) \cup {b}$</th>
<th>$\lambda S. \text{if } a \in S \text{ then } S \cup {b}$</th>
<th>$\lambda S. \text{else } S - {b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  a  b</td>
<td>0  a  b</td>
<td>0  a  b  c</td>
<td>0  a  b  c</td>
<td>0  a  b  c</td>
</tr>
<tr>
<td>0  0  0</td>
<td>0  0  0</td>
<td>0  0  0  0</td>
<td>0  0  0  0</td>
<td>0  0  0  0</td>
</tr>
</tbody>
</table>

Yue Li @ Nanjing University
**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:

$$R_f = \{(0,0)\} \cup \{(0,y) | y \in f(\emptyset)\} \cup \{(x,y) | y \notin f(\emptyset) \text{ and } y \in f(\{x\})\}$$

- Edge: $0 \rightarrow 0$
- Edge: $0 \rightarrow d_1$
- Edge: $d_1 \rightarrow d_2$

<table>
<thead>
<tr>
<th>$\lambda S.S$</th>
<th>$\lambda S{a}$</th>
<th>$\lambda S.(S-{a})\cup{b}$</th>
<th>$\lambda S.\mathrm{if}\ a \in S$ then $S \cup {b}$ else $S - {b}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a b</td>
<td>0 a b</td>
<td>0 a b c</td>
<td>0 a b c</td>
</tr>
<tr>
<td>0 o o o</td>
<td>0 o o o</td>
<td>0 o o o o</td>
<td>0 o o o o</td>
</tr>
<tr>
<td>0 o o o</td>
<td>0 o o o</td>
<td>0 o o o o</td>
<td>0 o o o o</td>
</tr>
<tr>
<td>0 a b</td>
<td>0 a b</td>
<td>0 a b c</td>
<td>0 a b c</td>
</tr>
</tbody>
</table>

Yue Li @ Nanjing University
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:

- $R_f = \{ (0,0) \}$ \hspace{1cm} Edge: $0 \rightarrow 0$
- $\cup \{ (0,y) | y \in f(\emptyset) \}$ \hspace{1cm} Edge: $0 \rightarrow d_1$
- $\cup \{ (x,y) | y \notin f(\emptyset) \mbox{ and } y \in f(\{x\}) \}$ \hspace{1cm} Edge: $d_1 \rightarrow d_2$

![Diagram](image)
Build Exploded Supergraph

Exploded Supergraph G#:
Each node n in supergraph G* is “exploded” into D+1 nodes in G#, and each edge \( n_1 \rightarrow n_2 \) in G* is “exploded” into the representation relation of the flow function associated with \( n_1 \rightarrow n_2 \) 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:
\[
R_f = \{ (0,0) \} \\
\cup \{ (0,y) | y \in f(\emptyset) \} \\
\cup \{ (x,y) | y \notin f(\emptyset) \text{ and } y \in f(\{x\}) \}
\]

Edge: 0 \( \rightarrow \) 0

Edge: 0 \( \rightarrow \) \( d_1 \)

Edge: \( d_1 \rightarrow d_2 \)

\[
\begin{align*}
\lambda S.S & \quad \lambda S.\{a\} & \quad \lambda S.(S-\{a\})\cup\{b\} & \quad \lambda S.\text{if } a \in S \text{ then } S \cup \{b\} \text{ else } S - \{b\}
\end{align*}
\]

\[
\begin{array}{cccc}
0 & a & b & \quad 0 & a & b & \quad 0 & a & b & c & \quad 0 & a & b & c \\
\hline
0 & 0 & 0 & \quad 0 & 0 & 0 & \quad 0 & 0 & 0 & \quad 0 & 0 & 0 & \\
0 & 0 & 0 & \quad 0 & 0 & 0 & \quad 0 & 0 & 0 & \quad 0 & 0 & 0 & \\
0 & a & b & \quad 0 & a & b & \quad 0 & a & b & c & \quad 0 & a & b & c \\
\hline
\end{array}
\]

Yue Li @ Nanjing University
Build Exploded Supergraph

Exploded Supergraph $G^\#$:

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

$$R_f = \{ (0,0) \} \cup \{ (0,y) \mid y \in f(\emptyset) \} \cup \{ (x,y) \mid y \not\in f(\emptyset) \text{ and } y \in f(\{x\}) \}$$

<table>
<thead>
<tr>
<th>$\lambda S. S$</th>
<th>$\lambda S. {a}$</th>
<th>$\lambda S.(S-{a}) \cup {b}$</th>
<th>$\lambda S. \begin{cases} \text{if } a \in S \ \text{then } S \cup {b} \ \text{else } S - {b} \end{cases}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a b</td>
<td>0 a b</td>
<td>0 a b c</td>
<td>0 a b c</td>
</tr>
<tr>
<td>O O O</td>
<td>O O O</td>
<td>O O O</td>
<td>O O O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O O O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O O O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 a b</td>
<td>0 a b</td>
<td>0 a b c</td>
<td>0 a b c</td>
</tr>
</tbody>
</table>

Edge: $0 \rightarrow 0$

Edge: $0 \rightarrow d_1$

Edge: $d_1 \rightarrow d_2$

Why we need $0 \rightarrow 0$ edges?
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 \( \text{OUT}[n_4] \) after the analysis finishes.

\[
\text{OUT}[n_4] = f_4 \circ f_3 \circ f_2 \circ f_1(\text{IN}[n_1])
\]
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 \( \text{OUT}[n_4] \) after the analysis finishes.

\[
\text{OUT}[n_4] = f_4 \circ f_3 \circ f_2 \circ f_1(\text{IN}[n_1])
\]

Data facts are propagated via the composition of flow functions. In this case, the “reachability” is directly retrieved from the final result in \( \text{OUT}[n_4] \).
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 \( \text{OUT}[n_4] \) after the analysis finishes.

\[
\text{OUT}[n_4] = f_4 \circ f_3 \circ f_2 \circ f_1(\text{IN}[n_1])
\]

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

For the same case, in IFDS, whether data fact \( a \) holds at \( p \) depends on if there is a path from \( <s_{\text{main}}, 0> \) 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 \( \text{OUT}[n_4] \) after the analysis finishes.

\[
\text{OUT}[n_4] = f_4 \circ f_3 \circ f_2 \circ f_1(\text{IN}[n_1])
\]

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

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

\( \lambda S.\{a\} \) says \( a \) holds at \( p \) regardless of input \( S \); however, without edge \( 0 \rightarrow 0 \), 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.

Yue Li @ Nanjing University
So We Need the “Glue Edge” $0 \rightarrow 0$!

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

$$\text{OUT}[n_4] = f_4 \circ f_3 \circ f_2 \circ f_1(\text{IN}[n_1])$$

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

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.

$\lambda S.\{a\}$ says $a$ holds at $p$ regardless of input $S$; however, without edge $0 \rightarrow 0$,

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.

Yue Li @ Nanjing University
Now, let’s build an exploded supergraph
$$\lambda S.\{x, g\}$$

1. **$S_{main}$**
   - $x = 0$
   - **Call$_p$**
   - **Ret$_p$**
   - **$e_{main}$**

2. **if $a > 0$**
   - $g = 0$
   - $a = a - g$
   - **Call$_p$**
   - **Ret$_p$**
   - **Print($a, g$)**

3. **$S_p$**
\[ \lambda S.S-\{x\} \]
\( \lambda S. S<x/a> \)
\[ \lambda S.S-\{g\} \]

**Diagram**

- **S**\(_{\text{main}}\)
  - \(x = 0\)
  - Call\(_{p}\)
  - Ret\(_{p}\)
  - \(e_{\text{main}}\)

- **S**\(_{p}\)
  - if \(a > 0\)
    - \(g = 0\)
    - \(a = a - g\)
    - Call\(_{p}\)
    - Ret\(_{p}\)
    - \(\text{Print}(a,g)\)
    - \(e_{p}\)
\[ \lambda S. \text{if}(a \in S) \text{ or } (g \in S) \text { then } S \cup \{a\} \text { else } S - \{a\} \]
\( \lambda S. \text{if}(a \in S) \) or \( (g \in S) \) then \( S \cup \{a\} \) else \( S - \{a\} \)
\[ \lambda S.S.S \]
\[ \lambda S.S-\{a\} \]

- \(S_{main}\)
- \(x = \emptyset\)
- Call
- Ret
- e
- \(S_p\)
- if \(a > \emptyset\)
- g = \emptyset
- a = a - g
- Call
- Ret
- Print(a, g)
- e_p
\[ \lambda S.S-(a) \]
\( \lambda S.S-\{g\} \)

\( \text{S}_{\text{main}} \)

\( x = 0 \)

\( \text{Call}_p \)

\( \text{Ret}_p \)

\( \text{e}_{\text{main}} \)

\( \text{S}_p \)

\( \text{if } a > 0 \)

\( g = 0 \)

\( a = a - g \)

\( \text{Call}_p \)

\( \text{Ret}_p \)

\( \text{Print}(a,g) \)

\( \text{e}_p \)
That $g$ is reachable along realizable paths from $<S_{\text{main}}, 0>$
That g is reachable only along non-realizable paths from $S_{\text{main}, 0}$.
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 \(<S_{\text{main}},0>\).
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 $<S_{main}, 0>$.
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 \(<s_{main}, 0>\)
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 $d \in MRP_n$, 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\textsuperscript{p}_p)
begin
  Let (N\textsuperscript{p}, E\textsuperscript{p}) = G\textsuperscript{p}_p
  PathEdge := \{ \langle s\text{\_main}, 0 \rangle \rightarrow \langle s\text{\_main}, 0 \rangle \}
  WorkList := \{ \langle s\text{\_main}, 0 \rangle \rightarrow \langle s\text{\_main}, 0 \rangle \}
  SummaryEdge := \emptyset
  ForwardTabulateSLRPs()
  for each n \in N \text{\_p} do
    X_n := \{ d_2 \in D \mid \exists d_1 \in (D \cup \{ 0 \}) \text{ such that } \langle \text{procOf}(n), d_1 \rangle \rightarrow \langle n, d_2 \rangle \in \text{PathEdge} \}
    \text{od}
    \begin{procedure}
      Propagate(e)
      begin
        if e \not\in \text{PathEdge} then Insert e into PathEdge; Insert e into WorkList fi
      end
    \end{procedure}
  \end{procedure}
  ForwardTabulateSLRPs()
  while WorkList \not= \emptyset do
    Select and remove an edge \langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle from WorkList
    switch n
    case n \in \text{Call}_p :
      for each d_3 such that \langle n, d_2 \rangle \rightarrow \langle \text{callProc}(n), d_3 \rangle \in E\textsuperscript{p} do
        Propagate(\langle \text{callProc}(n), d_3 \rangle \rightarrow \langle \text{callProc}(n), d_3 \rangle)
      od
      for each d_3 such that \langle n, d_2 \rangle \rightarrow \langle \text{returnSite}(n), d_3 \rangle \in (E\textsuperscript{p} \cup \text{SummaryEdge}) do
        Propagate(\langle s_p, d_1 \rangle \rightarrow \langle \text{returnSite}(n), d_3 \rangle)
      od
    end case
    case n = e_p :
      for each c \in \text{callers}(p) do
        for each d_4, d_5 such that \langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle \in E\textsuperscript{p} and \langle e_p, d_2 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle \in E\textsuperscript{p} do
          if \langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle \not\in \text{SummaryEdge} then
            Insert \langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle into SummaryEdge
          end if
          for each d_3 such that \langle \text{procOf}(c), d_3 \rangle \rightarrow \langle c, d_4 \rangle \in \text{PathEdge} do
            Propagate(\langle \text{procOf}(c), d_3 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle)
          od
        fi
      od
    end case
    case n \in (N_p - \text{Call}_p - \{ e_p \}) :
      for each \langle m, d_3 \rangle such that \langle n, d_2 \rangle \rightarrow \langle m, d_3 \rangle \in E\textsuperscript{p} do
        Propagate(\langle s_p, d_1 \rangle \rightarrow \langle m, d_3 \rangle)
      od
    end case
  end switch
  \text{od}
end
```

$O(ED^3)$
Tabulation Algorithm

```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate(Gp)  
begin  
    Let \((N^p, E^p) = Gp\)
    PathEdge := \{ \langle s_{main}, 0 \rangle \rightarrow \langle s_{main}, 0 \rangle \}
    WorkList := \{ \langle s_{main}, 0 \rangle \rightarrow \langle s_{main}, 0 \rangle \}
    SummaryEdge := \emptyset
    ForwardTabulateSLRPs()
    for each \(n \in N^p\) do  
        \(X_n := \{ d_2 \in D \mid \exists d_1 \in (D \cup \{0\}) \text{ such that } \langle s_{procOf(n)}, d_1 \rangle \rightarrow \langle n, d_2 \rangle \in \text{PathEdge} \} \)
    od  
    procedure Propagate(e)  
    begin  
        if \(e \notin \text{PathEdge}\) then  
            Insert e into PathEdge; Insert e into WorkList  
        else  
            ForwardTabulateSLRPs()  
        end  
    end  
    while WorkList \neq \emptyset do  
        Select and remove an edge \(\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle\) from WorkList  
        switch \(n\)  
            case \(n \in \text{Call}_p:\)  
                for each \(d_2\) such that \(\langle n, d_2 \rangle \rightarrow \langle s_{\text{calledProc(n)}}, d_3 \rangle \in E^p\) do  
                    Propagate(\langle s_{\text{calledProc(n)}}, d_3 \rangle \rightarrow \langle s_{\text{calledProc(n)}}, d_3 \rangle)  
                od  
                for each \(d_3\) such that \(\langle n, d_2 \rangle \rightarrow \langle \text{returnSite(n)}, d_3 \rangle \in (E^p \cup \text{SummaryEdge})\) do  
                    Propagate(\langle s_p, d_1 \rangle \rightarrow \langle \text{returnSite(n)}, d_3 \rangle)  
                od  
            end case  
            case \(n = e_p:\)  
                for each \(c \in \text{callers(p)}\) do  
                    for each \(d_4, d_5\) such that \(\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle \in E^p\) and \(\langle e_p, d_2 \rangle \rightarrow \langle \text{returnSite(c)}, d_5 \rangle \in E^p\) do  
                        if \(\langle c, d_4 \rangle \rightarrow \langle \text{returnSite(c)}, d_5 \rangle \notin \text{SummaryEdge}\) then  
                            Insert \(\langle c, d_4 \rangle \rightarrow \langle \text{returnSite(c)}, d_5 \rangle\) into \text{SummaryEdge}  
                        end if  
                        for each \(d_3\) such that \(\langle s_{\text{procOf(c)}}, d_3 \rangle \rightarrow \langle c, d_4 \rangle \in \text{PathEdge}\) do  
                            Propagate(\langle s_{\text{procOf(c)}}, d_3 \rangle \rightarrow \langle \text{returnSite(c)}, d_5 \rangle)  
                        od  
                    fi  
                od  
            end case  
            case \(n \in (N_p - \text{Call}_p - \{e_p\}):\)  
                for each \(m, d_3\) such that \(\langle n, d_2 \rangle \rightarrow \langle m, d_3 \rangle \in E^p\) do  
                    Propagate(\langle s_p, d_1 \rangle \rightarrow \langle m, d_3 \rangle)  
                od  
            end case  
        end switch  
    od  
end```

\(O(ED^3)\)

No time to cover the whole algorithm
Tabulation Algorithm

```
decre PathEdge, WorkList, SummaryEdge: global edge set

algorithm Tabulate(G_
p^p)
begin
    Let (N^p, E^p) = G^p
    PathEdge := { (s_{main}, 0) → (s_{main}, 0) }
    WorkList := { (s_{main}, 0) → (s_{main}, 0) }
    SummaryEdge := Ø
    ForwardTabulateSLRPs()
    for each n ∈ N^p do
        X_n := { d ∈ D | ∃ d_1 ∈ (D ∪ {0}) such that (s_{procOf(n)}, d_1) → (n, d_2) ∈ PathEdge }
        od
    procedure Propagate(e)
        begin
            if e ∉ PathEdge then Insert e into PathEdge; Insert e into WorkList fi
        end
    ForwardTabulateSLRPs()
    while WorkList ≠ Ø do
        Select and remove an edge (s_p, d_1) → (n, d_2) from WorkList
        switch n
        case n ∈ Call_i :
            for each d_3 such that (n, d_2) → (s_{calledProc(n)}, d_3) ∈ E^9 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^9 ∪ 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^9 and (e_p, d_2) → (returnSite (c), d_5) ∈ E^9 do
                    if (c, d_4) → (returnSite (c), d_5) ∈ SummaryEdge then
                        Insert (c, d_4) → (returnSite (c), d_5) into SummaryEdge
                    end if
                    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
        end case
    od
end
```

\( O(ED^3) \)

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

\[ S_p \]
\[ \text{Call}_p \]
\[ \text{Ret}_p \]
\[ e_p \]

\[ S_p', \]
\[ \text{Call}_{p'} \]
\[ \text{Ret}_{p'} \]

Yue Li @ Nanjing University
Core Working Mechanism of Tabulation Algorithm

$s_p$

$\text{Call}_p$

$e_p$

$\text{Ret}_p$

$S_{p',}$

$e_{p',}$

$\text{Call}_{p',}$

$\text{Ret}_{p',}$

Yue Li @ Nanjing University
Core Working Mechanism of Tabulation Algorithm

$S_p$

Call$_p$

Ret$_p$

e$_p$

$S_p'$

Call$_{p'}$

Ret$_{p'}$

e$_{p'}$
Core Working Mechanism of Tabulation Algorithm

$S_p$

Call$_p$

Ret$_p$

$e_p$

$S_{p'}$

Call$_{p'}$

Ret$_{p'}$

Yue Li @ Nanjing University
Core Working Mechanism of Tabulation Algorithm

\[ S_p \]
\[ \text{Call}_p \]
\[ \text{Ret}_p \]
\[ e_p \]

\[ S_p' \]
\[ \text{Call}_p',\]
\[ \text{Ret}_p',\]

Yue Li @ Nanjing University
Core Working Mechanism of Tabulation Algorithm

$s_p$

Call$_p$

$e_p$

Ret$_p$

...
Core Working Mechanism of Tabulation Algorithm

When handling each exit node (e\textsubscript{p'}), call-to-return matching begins: find out the call-sites calling p' (Call\textsubscript{p}, Call\textsubscript{p''}) and then find out their corresponding return-sites (Ret\textsubscript{p}, Ret\textsubscript{p''}).
Actually, here a summary edge from $<\text{Call}, d_m>$ to $<\text{Ret}, d_n>$ 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 $p'$ ($\text{Call}_p, \text{Call}_{p''}$) and then find out their corresponding return-sites ($\text{Ret}_{p}, \text{Ret}_{p''}$).
Core Working Mechanism of Tabulation Algorithm
Core Working Mechanism of Tabulation Algorithm

$s_p$

Call$_p$

Ret$_p$

e$_p$

$s_{p'}$

Call$_{p'}$

Ret$_{p'}$
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
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

\[ F(x \land y) = F(x) \land F(y) \]

• Constant Propagation

\[
\begin{array}{ccc}
  x & y & z \\
  0 & 0 & 0 \\
\end{array}
\]

z’s value depends on both y’s and x’s
Understanding the Distributivity of IFDS

- Distributivity

\[ F(x \land y) = F(x) \land F(y) \]

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ z = x + y \]

z’s value depends on both y’s and x’s
Understanding the Distributivity of IFDS

• Distributivity

\[ F(x \land y) = F(x) \land F(y) \]

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>

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

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

\[
F(x \land y) = F(x) \land F(y)
\]

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\( z = x + y \)

Each representation relation indicates \”if \( x \) exists, then \ldots\”, \”if \( y \) exists then \ldots\”

But when we need \”if both \( x \) and \( y \) exist\”, how to draw the representation relation?
Understanding the Distributivity of IFDS

- Distributivity

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

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

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**

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

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.

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?

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

$z = x + y$

\[ \begin{array}{cccc}
  x & y & z \\
  o & o & o \\
\end{array} \]
Understanding the Distributivity of IFDS

- Distributivity

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

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

- Pointer Analysis

```
x = new T
y = x
x.f = x
z = y.f
```

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

Understanding the Distributivity of IFDS

• Pointer Analysis

```
entry
x = new T
y = x
x.f = x
z = y.f
exit
```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>x.f</th>
<th>y.f</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>
Understanding the Distributivity of IFDS

- Pointer Analysis

```
x = new T
y = x
x.f = x
z = y.f
exit
```

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

Yue Li @ Nanjing University
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

• 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

```
entry
x = new T
y = x
x.f = x
z = y.f
exit
```

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

```
0 x y x.f y.f z
```

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.

Yue Li @ Nanjing University
Understanding the Distributivity of IFDS

- 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

1. Feasible and Realizable Paths
2. CFL-Reachability
3. Overview of IFDS
4. Supergraph and Flow Functions
5. Exploded Supergraph and Tabulation Algorithm
6. Understanding the Distributivity of IFDS
The You Need To Understand in This Lecture

• Understand CFL-Reachability

• Understand the basic idea of IFDS

• Understand what problems can be solved by IFDS
感谢陪伴，期待你们的未来

祝同学们在新的一年有新的收获

再见
南京大学
李樾
谭添
计算机科学与技术系
程序设计语言
静态分析与编译技术
程序分析与数据流算法