Detecting Mechanical Bugs using Static Analysis: Going Inter-procedural
Acknowledgements

These notes combine slides from:

- Galeotii, Gorla, Rau
  “Automated Testing and Verification - Interprocedural Dataflow Analysis”

- Andrey Goder
- Erkan Keremogly
  Both slide decks discuss IDFS framework paper “Precise Interprocedural Dataflow Analysis via graph Reachability” by Reps, Horwitz, Sagiv

- Rintetzky, Sagiv
  “Interprocedural Dataflow Analysis”

- Reps
  Tutorial “Program Analysis via Graph Reachability” at PLDI 2000

- Eran Yahav
  “Program Analysis and Synthesis - Interprocedural Analysis”
Intra-procedural dataflow analysis revisited: join over all paths (JOP) \(\iff\) least fixpoint (LFP)

we propagate dataflow facts along paths in CFG

\(\Rightarrow\) can define transfer function for an entire path
as composition of transfer functions for nodes:

\[\text{start} \xrightarrow{C} f_1 \xrightarrow{f_2} \cdots \xrightarrow{f_{k-1}} f_k \xrightarrow{n}\]

\[p_{pf} = f_k \circ f_{k-1} \cdots f_2 \circ f_1\]

\[\text{JOP} [n] = \bigcup_{p \in \text{PathsTo}[n]} p_{pf} (C)\]

\[\text{Compute } \text{out}[n] \text{ for each } n \in N:\]
\[\text{out}[n] := \bot \text{ (or TOP if MUST analysis)}\]

Repeat

For each \(n\)

\[\text{in}[n] := \oplus \{ \text{out}[m] \mid m \in \text{pred}(n) \}\]

\[\text{out}[n] := \text{transfer}[n](\text{in}[n])\]

Until no further changes to \(\text{in/out}\)

**Theoretically best possible solution to dataflow problem:**
considers all paths separately and only combines at the last possible moment (vs at every join point)

**Ifp solution has been proven to be at least as precise**
and to be the same for transfer functions that are distributive w.r.t \(\oplus\):
\[f(a \oplus b) = f(a) \oplus f(b)\]

**From previous lecture:**

\begin{tabular}{|c|c|}
\hline
Direction & U (MAY) \\
\hline
Forward & Given \text{in}[n], compute \text{out}[n] \\
& Apply \text{transfer}[n] to \text{predecessors}[n] \\
& Property holds in some path (reaching defs, zero analysis) \\
\hline
\end{tabular}
Intra-procedural dataflow analysis revisited: JOP example - possibly uninitialized vars

```
Start

{x = 3

if ...

{w, y}

{w, x, y}

\lambda V. \{w, x, y\}

\lambda V. V - \{x\}

\{w, y\}

{w, y}

\lambda V. V

\lambda V. V

y = x

\lambda V. \{w\}

\{w\}

\{w\}

w = 8

\lambda V. V - \{w\}

\{w, y\}

{w, y}

\lambda V. \{w, y\}

\{w, y\}

\lambda V. \{w, y\}

\{w, y\}

\lambda V. V - \{w\}

printf(y)

\{w, y\}

\{w, y\}

\{w, y\}

V = \{w, y\}

\{w, y\}
```

Intra-procedural dataflow analysis revisited: JOP example - possibly uninitialized vars
Going inter-procedural: inter-procedural control flow graph (icfg)

crossing procedure boundaries

effect of calling procedure = effect of executing its body

goal: compute effect on dataflow facts
Problem: JOP propagates along unrealizable paths

path along edges a, b, c is not valid
leads to imprecise (but still sound) dataflow results
Problem:
JOP propagates along unrealizable paths

leads to imprecise (but still sound) dataflow results

and possible false positives in warnings

unrealizable path a, b, c will lead us to warn about y being possibly 0 at node 3 = false positive!
Solution: join over all valid paths

$J_{OVP}[n] = \bigcup_{p \in \text{MatchedPathsTo}[n]} pf_p(C)$

paths with matching calls and returns (possibly left unbalanced):
language of valid paths can be defined via context free grammar

matched ::= (i matched) | matched | ε
valid ::= valid (i matched | matched
Valid versus invalid path

is a node reachable along a valid path? = Dyck-reachability
Two Common Approaches to Limiting Propagation Along Valid Paths

**call string approach**
- blend inter-procedural with intra-procedural by tagging every dataflow fact with call history

**functional approach**
- determine effect of a procedure once (represent out symbolically in terms of in)
- repeatedly apply effect on dataflow facts

[Sharir, Pnueli 82]
Call String Approach

- Record at every node a pair \((l, c)\) where \(l \in L\) is the dataflow information and \(c\) is a suffix of unmatched calls.
- Use Chaotic iterations.
- To guarantee termination limit the size of \(c\) (typically 1 or 2).
- Emulates inline (but no code growth).
- Exponential in size of \(c\).
Call String Approach: Example 1 for $|cs|=1$

```c
int p(int a) {
    return a + 1;
}

void main() {
    int x;

    c1: x = p(7);
    c2: x = p(9);
}
```

```c
int p(int a) {
    c1: [a 7]
    return a + 1;
}

void main() {
    int x;

    c1: x = p(7);
    c2: x = p(9);
}
```
Call String Approach: Example 1 for $|cs|=1$

```c
void main() {
    int x;
    c1: x = p(7);
    ε: x → 8
    c2: x = p(9);
}

void main() {
    int x;
    c1: x = p(7);
    ε: [x → 8]
    c2: x = p(9);
}

void main() {
    int x;
    c1: x = p(7);
    ε: [x → 8]
    c2: x = p(9);
}
```

```c
int p(int a) {
    c1: [a → 7]
    return a + 1;
    c1:[a → 7, $$ → 8]
}
```

```c
int p(int a) {
    c1:[a → 7]
    return a + 1;
    c1:[a → 7, $$ → 8]
}
```

```c
int p(int a) {
    c1:[a → 7]
    c2:[a → 9]
    return a + 1;
    c1:[a → 7, $$ → 8]
}
```
Call String Approach: Example 1 for $|cs|=1$

```c
void main() {
    int x;
    c1: x = p(7);
    $\varepsilon$: [x → 8]
    c2: x = p(9);
}

void main() {
    int x;
    c1: x = p(7);
    $\varepsilon$: [x → 8]
    c2: x = p(9) ;
    $\varepsilon$: [x → 10]
}
```

```c
int p(int a) {
    c1:[a → 7]
    c2:[a → 9]
    return a + 1;
    c1:[a → 7, $\$$ → 8]
    c2:[a → 9, $\$$ → 10]
}
```
Call String Approach: Example 2 for $|cs|=1$

```c
void main() {
    int x;
    c1: x = p(7);
    ε: [x ⊸ 8]
    c2: x = p(9);
    ε: [x ⊸ 10]
}

int p(int a) {
    c1:[a ⊸ 7]
    c2:[a ⊸ 9]
    return c3: p1(a + 1);
    ε: [x ⊸ τ]
    c1:[a ⊸ τ, $$$ ⊸ τ]
    c2:[a ⊸ τ, $$$ ⊸ τ]
}

int p1(int b) {
    (c1|c2)c3:[b ⊸ τ]
    return 2 * b;
    (c1|c2)c3:[b ⊸ τ, $$$ ⊸ τ]
}
```

Would not have moved to top for $|cs|=2$

Easy to implement
Only feasible for very short call strings (exponential in their length)
Quickly loses precision for recursive programs

Order of calls can be abstracted
Related method: procedure cloning
Functional Approach

Abstract summaries symbolically represent effect of procedure on elements of the lattice

- The call-graph is traversed in a Bottom-Up walk:
  - Starting by leafs (no further method invokations)
    - Perform intraprocedural dataflow analysis
    - Store summary
  - When a method makes a call
    - Look for summary (bottom-up traverse ensures it exists)
    - Instantiate it with actual parameters (top-down)

- For recursive methods (or cycles) requires another fix-point
  - On the recursive subcomponent
    - Cycle in the Call Graph
Functional Approach: Example

Phase 1

```c
void main() {
    p(7);
}
```

```c
int p(int a) {
    [a ↦ a₀, x ↦ x₀]
    if (...) {
        a = a - 1;
        [a ↦ a₀ - 1, x ↦ x₀]
        p(a);
        [a ↦ a₀ - 1, x ↦ 2a₀ + 7]
        a = a + 1;
        [a ↦ a₀, x ↦ 2a₀ + 7]
    }
    [a ↦ a₀, x ↦ τ]
    ⇔ x = 2 * a + 5;
    [a ↦ a₀, x ↦ 2a₀ + 5]
}
```

Phase 2

```c
void main() {
    p(7);
}
```

```c
int p(int a) {
    [a ↦ τ, x ↦ 0]
    if (...) {
        a = a - 1;
        [a ↦ τ, x ↦ 0]
        p(a);
        [a ↦ τ, x ↦ τ]
        a = a + 1;
        [a ↦ τ, x ↦ τ]
    }
    [a ↦ τ, x ↦ τ]
    p(a₀, x₀) = [a ↦ a₀, x ↦ 2a₀ + 5]
    x = 2 * a + 5;
    ⇔ [a ↦ τ, x ↦ τ]
}
```

Involves fixpoints: need to guarantee finite height for functional lattice
possible that L has finite height, but lattice of monotonic functions from L to L not
Requires efficient data structure: functional join, functional composition, equality tests
Special Case: IFDS dataflow problems

- Finite subset distributive
  - Lattice $L = \mathcal{P}(D)$
  - $\subseteq$ is $\subseteq$
  - $\sqcup$ is $\sqcup$
  - Transfer functions are distributive

- Efficient solution through formulation as CFL reachability
Special ICFG for IFDS problems: super graph

**Example**

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

**Example**

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

**Call edge** connects call site to callee (to pass info that concerns callee (e.g., actuals to formals))

**Return edge** connects return site to call site (to pass invocation results back)

**Call-to-return edge**:
pass information from before call site to successor (to pass info that does not concern callee (e.g., about local variables))
IFDS: why distributive transfer functions and finite set of dataflow facts?

=> every distributive function in $2^D \rightarrow 2^D$

is defined by its value on the empty set and every singleton subset of $D$

e.g., suppose $S = \{a, b, c\}$ is a subset of $D$

\[ f(S) = \{\} \cup f(\{\}) \cup f(\{a\}) \cup f(\{b\}) \cup f(\{c\}) \]

enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ R_f = \{(0,0)\} \]

\[ \cup \{(0, y) : y \in f(\emptyset)\} \]

\[ \cup \{ (x, y) : y \in f(\{x\}) \text{ and } y \notin f(\emptyset) \} \]

(node 0 represents empty set: every transfer function goes from 0 to 0 (invariant: 0 always holds)

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

(node 0 represents empty set: every transfer function goes from 0 to 0 (invariant: 0 always holds)

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]

$R_f$ enables representing transfer function as a graph with $2(D+1)$ nodes and edges:

\[ f(X) = \{y : \exists x \in X \text{ such that } (x, y) \in R\} \cup \{y : (0, y) \in R\} - \{0\} \]
IFDS: examples of encoded functions

**Identity Function**

\[ f = \lambda V.V \]

\[ f(\{a, b\}) = \{a, b\} \]

**Constant Function**

\[ f = \lambda V.\{b\} \]

\[ f(\{a, b\}) = \{b\} \]
IFDS: examples of encoded functions

"Gen/Kill" Function

\[ f = \lambda V. (V - \{b\}) \cup \{c\} \]
\[ f(\{a, b\}) = \{a, c\} \]

Non-"Gen/Kill" Function

\[ f = \lambda V. \text{if } a \in V \]
\[ \quad \text{then } V \cup \{b\} \]
\[ \quad \text{else } V - \{b\} \]
\[ f(\{a, b\}) = \{a, b\} \]
IFDS: composing functions = combining graphs

\[ R_f; R_g =_{df} \{ (x, y) \in S \times S \mid \exists z \in S \text{ such that } (x, z) \in R_f \text{ and } (z, y) \in R_g \} \]

For all \( f, g \in 2^D \to 2^D \), \([R_f; R_g] = g \circ f\).

**Definition**

\[ f_1 = \lambda V. \text{if } a \in V \text{ then } V \cup \{b\} \text{ else } V - \{b\} \]

\[ f_2 = \lambda V. \text{if } b \in V \text{ then } \{c\} \text{ else } \emptyset \]

\[ f_2 \circ f_1(\{a, c\}) = \{c\} \]

becomes a reachability problem!
IFDS: exploded super graph

\[ G_{IP}^\# = (N^\#, E^\#), \] where
\[ N^\# = N^* \times (D \cup \{ 0 \}), \]
\[ E^\# = \{ \langle m, d_1 \rangle \rightarrow \langle n, d_2 \rangle \mid (m, n) \in E^* \]
\[ \text{and } (d_1, d_2) \in R_{M(m, n)} \}. \]

constructed from super graph \( G^* = (N^*, E^*) \):

replace each node by graph representation of transfer function of successor edge

main theorem:

\[ d \in JOVP[n] \iff \text{there is a valid path in graph } G^\# \text{ from node } \langle s_{\text{main}}, 0 \rangle \text{ to node } \langle n, d \rangle. \]

equivalent to computing CFL reachability over the exploded super graph using the valid parentheses grammar
IFDS: exploded super graph for uninitialized variables
IFDS: exploded super graph for uninitialized variables

start main
- \( x = 3 \)
- \( p(x,y) \)
- return from \( p \)
  - printf(y)
  - exit main

start \( p(a,b) \)
- if...
  - \( b = a \)
  - \( p(a,b) \)
  - return from \( p \)
    - printf(b)
    - exit \( p \)

Λ x y
Λ a b
IFDS: reachability over valid path of dataflow fact

\( \Lambda \ x \ y \)

start main

\( x = 3 \)

p(x,y)

return from p

printf(y)

YES!

exit main

\( \Lambda \ a \ b \)

start p(a,b)

if . . .

b = a

p(a,b)

return from p

printf(b)

NO!

exit p
IFDS reachability: taint analysis example

```c
void main() {
    int x = secret();
    int y = 0;
    y = foo(x);
    print(y);
}

int foo(int p) {
    return p;
}
```

Can information x flow from secret (source) to print (sink), without passing a sanitizer?

Transfer function for assignment \( p = x \):

```plaintext
0 \rightarrow x \rightarrow p
```

Regardless of whether \( p \) was tainted, \( p \) becomes as tainted as \( x \).

Control-flow edge

Data-flow edge

Call

Call-to-return

Violating information flow

Yes, \( y \) is tainted because reachable over valid path

[Bodden 13]
IFDS: efficient tabulation algorithm

- Worklist algorithm, start from entry of “main”
- Keep track of
  - Path edges: matched paren paths from procedure entry
  - Summary edges: matched paren call-return paths
- At each instruction
  - Propagate facts using transfer functions; extend path edges
- At each call
  - Propagate to procedure entry, start with an empty path
  - If a summary for that entry exits, use it
- At each exit
  - Store paths from corresponding call points as summary paths
  - When a new summary is added, propagate to the return node
\textbf{Algorithm Tabulate($G^I_p$)}

\begin{enumerate}
\item Let $(N^I, E^I) = G^I_p$
\item \textbf{PathEdge} := \{ $\langle s_{main}, 0 \rangle \rightarrow \langle s_{main}, 0 \rangle$ \}
\item \textbf{WorkList} := \{ $\langle s_{main}, 0 \rangle \rightarrow \langle s_{main}, 0 \rangle$ \}
\item \textbf{SummaryEdge} := \emptyset
\item \textbf{ForwardTabulateSLRPs()}
\item for each $n \in N^I$ do
\item \hspace{1em} $X_n := \{ d_2 \in D \mid \exists d_1 \in (D \cup \{0\}) \text{ such that } \langle s_{procOf}(n), d_1 \rangle \rightarrow \langle n, d_2 \rangle \in \textbf{PathEdge} \}$
\item \hspace{1em} \textbf{od}
\item \hspace{1em} \textbf{end for}
\item \hspace{1em} \textbf{procedure Propagate(e)}
\item \hspace{2em} \textbf{begin}
\item \hspace{3em} if $e \in \textbf{PathEdge}$ then Insert $e$ into $\textbf{PathEdge}$; Insert $e$ into $\textbf{WorkList}$ \textbf{fi}
\item \hspace{2em} \textbf{end procedure}
\item \hspace{1em} \textbf{procedure ForwardTabulateSLRPs()}
\item \hspace{2em} \textbf{begin}
\item \hspace{3em} \textbf{while} $\textbf{WorkList} \neq \emptyset$ \textbf{do}
\item \hspace{4em} \textbf{Select and remove an edge} $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ \textbf{from} $\textbf{WorkList}$
\item \hspace{4em} \textbf{switch} $n$
\item \hspace{5em} \textbf{case} $n \in \text{Call}_p$:
\item \hspace{6em} \textbf{for} each $d_3$ such that $\langle n, d_2 \rangle \rightarrow \langle s_{calledProc}(n), d_3 \rangle \in E^I$ do
\item \hspace{7em} \textbf{Propagate($\langle s_{calledProc}(n), d_3 \rangle \rightarrow \langle s_{calledProc}(n), d_3 \rangle$)}
\item \hspace{6em} \textbf{end for}
\item \hspace{5em} \textbf{end case}
\item \hspace{5em} \textbf{case} $n = e_p$:
\item \hspace{6em} \textbf{for} each $c \in \text{callers}(p)$ do
\item \hspace{7em} \textbf{for} each $d_4, d_5$ such that $\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle \in E^I$ and $\langle e_p, d_2 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle \in E^I$ do
\item \hspace{8em} \textbf{if} $\langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle \in \textbf{SummaryEdge}$ then
\item \hspace{9em} \textbf{Insert} $\langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle$ \textbf{into} \textbf{SummaryEdge}
\item \hspace{9em} \textbf{for} each $d_3$ such that $\langle s_{procOf}(c), d_3 \rangle \rightarrow \langle c, d_4 \rangle \in \textbf{PathEdge}$ do
\item \hspace{10em} \textbf{Propagate($\langle s_{procOf}(c), d_3 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle$)}
\item \hspace{9em} \textbf{end for}
\item \hspace{9em} \textbf{end if}
\item \hspace{8em} \textbf{end if}
\item \hspace{7em} \textbf{end for}
\item \hspace{5em} \textbf{end case}
\item \hspace{5em} \textbf{case} $n \in (N_p - \text{Call}_p - \{ e_p \})$:
\item \hspace{6em} \textbf{for} each $\langle m, d_3 \rangle$ such that $\langle n, d_2 \rangle \rightarrow \langle m, d_3 \rangle \in E^I$ do
\item \hspace{7em} \textbf{Propagate($\langle s_p, d_1 \rangle \rightarrow \langle m, d_3 \rangle$)}
\item \hspace{6em} \textbf{end for}
\item \hspace{5em} \textbf{end case}
\item \hspace{4em} \textbf{end switch}
\item \hspace{3em} \textbf{od}
\item \hspace{2em} \textbf{end procedure}
\end{enumerate}
```plaintext
declare PathEdge, WorkList, SummaryEdge: global edge set

algorithm Tabulate($G^\#_{IP}$)
begin
[1] Let ($N^\#, E^\#$) = $G^\#_{IP}$
[2] PathEdge := \{ $s_{\text{main}}, 0 \} \rightarrow \{ s_{\text{main}}, 0 \} \}
[3] WorkList := \{ $s_{\text{main}}, 0 \} \rightarrow \{ s_{\text{main}}, 0 \} \}
[4] SummaryEdge := \emptyset
[5] ForwardTabulateSLRPs()
[6] for each $n \in N^\#$ do
[7] \hspace{1em} $X_n := \{ d_2 \in D \mid \exists d_1 \in (D \cup \{ 0 \}) \text{ such that } (s_{\text{procOf}(n)}, d_1) \rightarrow (n, d_2) \in \text{PathEdge} \}$
[8] od
end
procedure Propagate($e$)
begin
[9] if $e \in \text{PathEdge}$ then Insert $e$ into PathEdge; Insert $e$ into WorkList fi
end
procedure ForwardTabulateSLRPs()
begin
[10] while WorkList $\neq \emptyset$ do
[11] Select and remove an edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ from WorkList
[12] switch $n$
[13] case $n \in \text{Call}_p$:
[14] for each $d_3$ such that $\langle n, d_2 \rangle \rightarrow \langle s_{\text{calledProc}(n)}, d_3 \rangle \in E^\#$ do
[15] Propagate($\langle s_{\text{calledProc}(n)}, d_3 \rangle \rightarrow \langle \text{calledProc}(n), d_3 \rangle$)
[16] od
[17] for each $d_3$ such that $\langle n, d_2 \rangle \rightarrow \langle \text{returnSite}(n), d_3 \rangle \in (E^\# \cup \text{SummaryEdge})$ do
[18] Propagate($\langle s_p, d_1 \rangle \rightarrow \langle \text{returnSite}(n), d_3 \rangle$)
[19] od
[20] end case
[21] case $n = e_p$ :
[22] for each $c \in \text{callers}(p)$ do
[23] for each $d_4, d_5$ such that $\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle \in E^\#$ and $\langle e_p, d_2 \rangle \rightarrow \langle \text{returnSite}(c), d_3 \rangle \in E^\#$ do
[24] if $\langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle \in \text{SummaryEdge}$ then
[25] Insert $\langle c, d_4 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle$ into SummaryEdge
[26] for each $d_3$ such that $\langle s_{\text{procOf}(c)}, d_3 \rangle \rightarrow \langle c, d_4 \rangle \in \text{PathEdge}$ do
[27] Propagate($\langle s_{\text{procOf}(c)}, d_3 \rangle \rightarrow \langle \text{returnSite}(c), d_5 \rangle$)
[28] od
[29] fi
[30] od
[31] od
[32] end case
[33] case $n \in (N_p - \text{Call}_p - \{ e_p \})$:
[34] for each $\langle m, d_3 \rangle$ such that $\langle n, d_2 \rangle \rightarrow \langle m, d_5 \rangle \in E^\#$ do
[35] Propagate($\langle s_p, d_1 \rangle \rightarrow \langle m, d_3 \rangle$)
[36] od
[37] end case
[38] end switch
[39] od
end

---

**Descend to Callee**

PathEdge ($\langle s_q, d_3 \rangle, \langle s_q, d_3 \rangle$):

EE\textsubscript{call} ($\langle m, d_2 \rangle, \langle s_q, d_3 \rangle$).

---

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

algorithm Tabulate($G^p_{IP}$)
begin
[1] Let $(N^p, E^p) = G^p_{IP}$
[2] PathEdge := $\{ (s_{main}, 0) \rightarrow (s_{main}, 0) \}$
[3] WorkList := $\{ (s_{main}, 0) \rightarrow (s_{main}, 0) \}$
[4] SummaryEdge := $\emptyset$
[5] ForwardTabulateSLRPs()
[6] for each $n \in N^p$ do
[7] $X_n := \{ d_2 \in D \mid \exists d_1 \in (D \cup \{ 0 \}) \text{ such that } (s_{procOf(n)}, d_1) \rightarrow (n, d_2) \in \text{PathEdge} \}$
[8] od
procedure Propagate($e$)
begin
[9] if $e \in \text{PathEdge}$ then Insert $e$ into PathEdge; Insert $e$ into WorkList fi
end
procedure ForwardTabulateSLRPs()
begin
[10] while WorkList $\neq \emptyset$ do
[11] Select and remove an edge $(s_p, d_1) \rightarrow (n, d_2)$ from WorkList
[12] switch $n$
[13] case $n \in \text{Call}_p$:
[14] for each $d_3$ such that $(n, d_2) \rightarrow (s_{calledProc(n)}, d_3) \in E^p$ do
[15] Propagate($(s_{calledProc(n)}, d_3) \rightarrow (s_{calledProc(n)}, d_3))$
[16] od
[17] for each $d_3$ such that $(n, d_2) \rightarrow (\text{returnSite}(n), d_3) \in (E^p \cup \text{SummaryEdge})$ do
[18] Propagate($(s_p, d_1) \rightarrow (\text{returnSite}(n), d_3))$
[19] od
[20] end case
[21] case $n = e_p$:
[22] for each $c \in \text{callers}(p)$ do
[23] for each $d_4, d_5$ such that $(c, d_4) \rightarrow (s_p, d_1) \in E^p$ and $(e_p, d_2) \rightarrow (\text{returnSite}(c), d_3) \in E^p$ do
[24] if $(c, d_4) \rightarrow (\text{returnSite}(c), d_5) \in \text{SummaryEdge}$ then
[25] Insert $(c, d_4) \rightarrow (\text{returnSite}(c), d_5)$ into SummaryEdge
[26] for each $d_3$ such that $(s_{procOf(c), d_3} \rightarrow (c, d_4) \in \text{PathEdge}$ do
[27] Propagate($(s_{procOf(c)}, d_3) \rightarrow (\text{returnSite}(c), d_5))$
[28] od
[29] fi
[30] od
[31] end case
[32] case $n \in (N_p - \text{Call}_p - \{ e_p \})$:
[33] for each $(m, d_3)$ such that $(n, d_2) \rightarrow (m, d_3) \in E^p$ do
[34] Propagate($(s_p, d_1) \rightarrow (m, d_3))$
[35] od
[36] end case
[37] end switch
[38] od
end

\(O(\#e^p)\)

Intra-procedural Closure via Summary

PathEdge ($(s_p, d_1), (n, d_2)$) :-
PathEdge ($(s_p, d_1), (m, d_2)$),
Summary ($(m, d_2), (n, d_3)$).

ordinary $E^p$ edge
ordinary $E^p$ edge or
summary edge
ordinary $E^p$, call-to-start or exit-to-return-site $E^p$ edge
ordinary $E^p$, path edge
(possibly new) path edge
(possibly new) summary edge
declare PathEdge, WorkList, SummaryEdge: global edge set

algorithm Tabulate($G'_IP$)
begin
[1] Let ($N^*, E^*$) = $G'_IP$
[2] PathEdge := { $s_{main}, 0 \rightarrow s_{main}, 0$ }
[3] WorkList := { $s_{main}, 0 \rightarrow s_{main}, 0$ }
[4] SummaryEdge := $\emptyset$
[5] ForwardTabulateSLRPs()
[6] for each $n \in N$ do
[7] 
[8] od
end
procedure Propagate($e$)
begin
[9] if $e \in$ PathEdge then Insert $e$ into PathEdge; Insert $e$ into WorkList fi
end
procedure ForwardTabulateSLRPs()
begin
[10] while WorkList $\neq \emptyset$ do
[11] Select and remove an edge $s_{p}, d_{1} \rightarrow (n, d_{2})$ from WorkList
[12] switch $n$
[13] case $n \in Call_{p}$ :
[14] for each $d_{3}$ such that $(n, d_{2}) \rightarrow s_{calledProc(n), d_{3}} \in E^{#}$ do
[15] Propagate($s_{calledProc(n), d_{3}} \rightarrow s_{calledProc(n), d_{3}}$)
[16] od
[17] for each $d_{3}$ such that $(n, d_{2}) \rightarrow \text{returnSite}(n, d_{3}) \in (E^{#} \cup \text{SummaryEdge})$ do
[18] Propagate($s_{p}, d_{1} \rightarrow \text{returnSite}(n, d_{3})$)
[19] od
[20] end case
[21] case $n = e_{p}$ :
[22] for each $c \in \text{callers}(p)$ do
[23] for each $d_{4}, d_{5}$ such that $(c, d_{4}) \rightarrow s_{p}, d_{1} \in E^{#}$ and $(e_{p}, d_{2}) \rightarrow \text{returnSite}(c, d_{5}) \in E^{#}$ do
[24] if $(c, d_{4}) \rightarrow \text{returnSite}(c, d_{5}) \in \text{SummaryEdge}$ then
[25] Insert $(c, d_{4})$ into SummaryEdge
[26] for each $d_{3}$ such that $s_{\text{procOf(c), d_{3}} \rightarrow (c, d_{4}) \in \text{PathEdge do}$
[27] Propagate($s_{\text{procOf(c), d_{3}} \rightarrow \text{returnSite}(c, d_{5})$)
[28] od
[29] fi
[30] od
[31] od
[32] end case
[33] case $n \in (N_{p} - \text{Call}_{p} - \{ e_{p} \})$ :
[34] for each $(m, d_{3})$ such that $(n, d_{2}) \rightarrow (m, d_{3}) \in E^{#}$ do
[35] Propagate($s_{p}, d_{1} \rightarrow (m, d_{3})$)
[36] od
[37] end case
[38] end switch
[39] od
end
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate($G_{hp}$)
begin
[1] Let ($N^g$, $E^g$) = $G_{hp}$
[2] PathEdge := \{ (s_{main}, 0) \rightarrow (s_{main}, 0) \}
[3] WorkList := \{ (s_{main}, 0) \rightarrow (s_{main}, 0) \}
[4] SummaryEdge := \emptyset
[5] ForwardTabulateSLRPs()
begin
for each $n \in N$ do
[6] $X_n := \{ d_2 \in D \mid \exists d_1 \in (D \cup \{ 0 \})$ such that $(s_{procOf(n)}, d_1) \rightarrow (n, d_2) \in PathEdge \} \od$
end procedure Propagate($e$)
begin
[7] if $e \in$ PathEdge then Insert $e$ into PathEdge; Insert $e$ into WorkList fi
end procedure ForwardTabulateSLRPs()
begin
while WorkList \neq \emptyset do
[8] Select and remove an edge $(s_p, d_1) \rightarrow (n, d_2)$ from WorkList
[9] switch $n$
[10] case $n \in Call_p$:
[11] for each $d_3$ such that $(n, d_2) \rightarrow (s_{calledProc(n)}, d_3) \in E^g$ do
[12] Propagate$(s_{calledProc(n)}, d_3) \rightarrow (s_{calledProc(n)}, d_3))$ od
[13] for each $d_3$ such that $(n, d_2) \rightarrow (returnSite(n), d_3) \in (E^g \cup SummaryEdge)$ do
[14] Propagate$((s_p, d_1) \rightarrow (returnSite(n), d_3))$ od
[15] end case
[16] case $n = e_p$:
[17] for each $c \in callers(p)$
[18] for each $d_4, d_5$ such that $(c, d_4) \rightarrow (s_p, d_1) \in E^g$ and $(e_p, d_2) \rightarrow (returnSite(c), d_3) \in E^g$ do
[19] if $(c, d_4) \rightarrow (returnSite(c), d_5)$ \in SummaryEdge then
[20] Insert $(c, d_4) \rightarrow (returnSite(c), d_5)$ into SummaryEdge
[21] for each $d_3$ such that $(s_{procOf(c)}, d_3) \rightarrow (c, d_4) \in PathEdge$ do
[22] Propagate$(s_{procOf(c)}, d_3) \rightarrow (returnSite(c), d_5))$ od
[23] fi
[24] od
[25] end case
[26] case $n \in (N_p - Call_p - \{ e_p \})$:
[27] for each $(m, d_3)$ such that $(n, d_2) \rightarrow (m, d_3) \in E^g$ do
[28] Propagate$((s_p, d_1) \rightarrow (m, d_3))$ od
[29] end case
[30] end switch
[31] od
end
## IFDS: Complexity

### Runtime Complexity

<table>
<thead>
<tr>
<th>Class</th>
<th>Properties</th>
<th>Intra.</th>
<th>Inter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributive</td>
<td>relation $O(D^2)$</td>
<td>$O(ED^2)$</td>
<td>$O(ED^3)$</td>
</tr>
<tr>
<td>h-sparse</td>
<td>at most $h$ edges outgoing from src. of relation</td>
<td>$O(hED)$</td>
<td>$O(Call D^3 + hED^2)$</td>
</tr>
<tr>
<td>Locally Separable</td>
<td>edges in rel. of form $o \rightarrow x$ or $x \rightarrow x$</td>
<td>$O(ED)$</td>
<td>$O(ED)$</td>
</tr>
</tbody>
</table>