Interprocedural Analysis

Noam Rinetzky

Mooly Sagiv
Why do runtime errors occur?

• Logical errors
• Incorrect initializations
• Resource limitations
• Aliasing problems
• API mismatch
Procedural program

```c
void main() {
    int x;
    x = p(7);
    x = p(9);
}

int p(int a) {
    return a + 1;
}
```
Effect of procedures

The effect of calling a procedure is the effect of executing its body.
Interprocedural Analysis

goal: compute the abstract effect of calling a procedure
Reduction to intraprocedural analysis

- Procedure inlining
- Naive solution: call-as-goto
Reminder: Constant Propagation

- $\perp$: No information
- $\top$: Variable not a constant

Diagram:

```
-\infty \quad -1 \quad 0 \quad 1 \quad \infty
```

$Z_{\perp}$
Reminder: Constant Propagation

• \( L = (\text{Var} \rightarrow Z^\top, \sqsubseteq) \)

• \( \sigma_1 \sqsubseteq \sigma_2 \) iff \( \forall x: \sigma_1(x) \sqsubseteq' \sigma_2(x) \)
  – \( \sqsubseteq' \) ordering in the \( Z^\top \) lattice

• Examples:
  – \([x \mapsto \bot, y \mapsto 42, z \mapsto \bot] \sqsubseteq [x \mapsto \bot, y \mapsto 42, z \mapsto 73]\)
  – \([x \mapsto \bot, y \mapsto 42, z \mapsto 73] \sqsubseteq [x \mapsto \bot, y \mapsto 42, z \mapsto \top]\)
Reminder: Constant Propagation

• Conservative Solution
  – Every detected constant is indeed constant
    • But may fail to identify some constants
  – Every potential impact is identified
    • Superfluous impacts
void main() {
    int x;
    x = p(7);
    x = p(9);
}

int p(int a) {
    return a + 1;
}
Procedure Inlining

```c
void main() {
    int x;
    x = p(7);
    x = p(9);
}

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

void main() {
    int a, x, ret;
    [a ↦⊥, x ↦⊥, ret ↦⊥]
    a = 7; ret = a+1; x = ret;
    [a ↦7, x ↦8, ret ↦8]
    a = 9; ret = a+1; x = ret;
    [a ↦9, x ↦10, ret ↦10]
}
```
Procedure Inlining

• Pros
  – Simple

• Cons
  – Does not handle recursion
  – Exponential blow up
  – Reanalyzing the body of procedures

\[
p_1 \{ \\
\text{call } p_2 \\
\ldots \\
\text{call } p_2 \\
\} \\
p_2 \{ \\
\text{call } p_3 \\
\ldots \\
\text{call } p_3 \\
\} \\
p_3 \{
\}
\]
A Naive Interprocedural solution

• Treat procedure calls as gotos
void main() {
    int x ;
    x = p(7);
    x = p(9) ;
}

int p(int a) {
    return a + 1;
}
void main() {
    int x;
    x = p(7);
    x = p(9);
}

int p(int a) {
    [a↦7]
    return a + 1;
}
void main() {
    int x;
    x = p(7);
    x = p(9);
}

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

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

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

void main() {
    int x;
    x = p(7);
    x = p(9);
}

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

void main() {
    int x;
    x = p(7);
    x = p(9);
}
void main() {
    int x;
    x = p(7);
    [x ↦ 8]
    x = p(9);
    [x ↦ 8]
}

int p(int a) {
    [a ↦ 7]
    return a + 1;
    [a ↦ 7, $$ ↦ 8]
}

---

**Simple Example**

void main() {
    int x;
    x = p(7);
    [x ↦ 8]
    x = p(9);
    [x ↦ 8]
}
void main() {
    int x;
    x = p(7);
    [x ↦ 8]
    x = p(9);
    [x ↦ 8]
}

int p(int a) {
    [a ↦ 7, $$ ↦ 8]
    return a + 1;
    [a ↦ 7, $$ ↦ 8]
}
void main() {
    int x;
    x = p(7);
    [x → 8]
    x = p(9);
    [x → 8]
}

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

call p(7)
retc p(7)
call p(9)
retc p(9)
Simple Example

```c
void main() {
    int x;
    x = p(7);
    [x → 8]
    x = p(9);
    [x → 8]
}

int p(int a) {
    [a → 𝛽]
    return a + 1;
    [a → 𝛽, $$ → 𝛽]
}
```

---

**Diagram:**
- `main()` function:
  - `call p(7)`, `retn p(7)`, `call p(9)`, `retn p(9)`
- `p()` function:
  - `return a + 1`
void main() {
    int x;
    x = p(7);
    x = p(9);
}

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

Simple Example
A Naive Interprocedural solution

- Treat procedure calls as gotos
- Pros:
  - Simple
  - Usually fast
- Cons:
  - Abstract call/return correlations
  - Obtain a conservative solution
Analysis by reduction

### Call-as-goto

```c
void main() {
    int x;
    x = p(7);
    x = p(9);
}
```

### Procedure inlining

```c
void main() {
    int a, x, ret;
    a = 7; ret = a + 1; x = ret;
    a = 9; ret = a + 1; x = ret;
}
```

why was the naive solution less precise?
Stack regime

P() {
  ...
  R();
  ...
}

R() {
  ...
}

Q() {
  ...
  R();
  ...
}

Diagram showing the stack regime with calls and returns between functions P, R, and Q.
Guiding light

• Exploit stack regime
  ➔ Precision
  ➔ Efficiency
Simplifying Assumptions

- Parameter passed by value
- No procedure nesting
- No concurrency

✓ Recursion is supported
Topics Covered

✓ Procedure Inlining
✓ The naive approach
  • Valid paths
  • The callstring approach
  • The Functional Approach
  • IFDS: Interprocedural Analysis via Graph Reachability
  • IDE: Beyond graph reachability

• The trivial modular approach
Join-Over-All-Paths (JOP)

- Let $\text{paths}(v)$ denote the potentially infinite set paths from start to $v$ (written as sequences of edges).

- For a sequence of edges $[e_1, e_2, \ldots, e_n]$ define $f [e_1, e_2, \ldots, e_n]: \mathbb{L} \rightarrow \mathbb{L}$ by composing the effects of basic blocks:
  
  $$f [e_1, e_2, \ldots, e_n](l) = f(e_n) (\ldots (f(e_2) (f(e_1) (l)) \ldots)$$

- $\text{JOP}[v] = \bigcup \{ f [e_1, e_2, \ldots, e_n](l) | [e_1, e_2, \ldots, e_n] \in \text{paths}(v) \}$
Join-Over-All-Paths (JOP)

Paths transformers:
- $f[e_1,e_2,e_3,e_4]$
- $f[e_1,e_2,e_7,e_8]$
- $f[e_5,e_6,e_7,e_8]$
- $f[e_5,e_6,e_3,e_4]$
- $f[e_1,e_2,e_7,e_8,e_9, e_1,e_2,e_3,e_4,e_9,...]$  
  ...

JOP:
- $f[e_1,e_2,e_3,e_4](\text{initial})$ □
- $f[e_1,e_2,e_7,e_8](\text{initial})$ □
- $f[e_5,e_6,e_7,e_8](\text{initial})$ □
- $f[e_5,e_6,e_3,e_4](\text{initial})$ □ ...

Number of program paths is unbounded due to loops
LFP approximates JOP

- JOP[v] = \bigcup \{ f[e_1, e_2, \ldots, e_n](l) \mid [e_1, e_2, \ldots, e_n] \in \text{paths}(v) \}

- LFP[v] = \bigcup \{ f[e](LFP[v']) \mid e = (v', v) \}
  \quad \text{LFP}[v_0] = \top

- JOP \subseteq LFP - for a monotone function
  - f(x \sqcup y) \supseteq f(x) \sqcup f(y)

- JOP = LFP - for a distributive function
  - f(x \sqcup y) = f(x) \sqcup f(y)

JOP may not be precise enough for interprocedural analysis!
Interprocedural analysis

Supergraph
Paths

- **paths(n)** the set of paths from s to n
  - ( (s,n₁), (n₁,n₃), (n₃,n₁) )
Interprocedural Valid Paths

- IVP: all paths with matching calls and returns
  - And prefixes
Interprocedural Valid Paths

• **IVP** set of paths
  – Start at program entry

• Only considers matching calls and returns
  – aka, *valid*

• Can be defined via context free grammar
  – `matched ::= matched ( ; matched ) | ε`
  – `valid ::= valid ( ; matched | matched`
    • *paths* can be defined by a regular expression
Join Over All Paths (JOP)

\[ \text{JOP}[v] = \bigcup \{[[e_1, e_2, \ldots, e_n]](\bar{f}) \mid (e_1, \ldots, e_n) \in \text{paths}(v) \} \]

\[ \mathbb{L} \ni f_k \circ \ldots \circ f_1 \ni L \rightarrow L \]

- JOP[v] = \bigcup \{[[e_1, e_2, \ldots, e_n]](\bar{f}) \mid (e_1, \ldots, e_n) \in \text{paths}(v) \}
- JOP \subseteq \text{LFP}
  - Sometimes JOP = LFP
    - precise up to "symbolic execution"
    - Distributive problem
The Join-Over-Valid-Paths (JVP)

• \text{vpaths}(n) \text{ all valid paths from program start to } n
• \text{JVP}[n] = \bigcup\{[[e_1, e_2, \ldots, e]](l) \\
\quad (e_1, e_2, \ldots, e) \in \text{vpaths}(n)\}
• \text{JVP} \subseteq \text{JFP}
  – In some cases the JVP can be computed
  – (Distributive problem)
The Call-String Approach

• The data flow value is associated with sequences of calls (call string)
• Use Chaotic iterations over the supergraph
Supergraph

P

Call node

s_P

f_1

n_4

| Call Q |

n_5

f_5

f_{x2r}

e_P

R

f_{c2e}

s_R

f_1

n_1

f_2

n_3

f_3

f_{x2r}

e_R

Entry node

Q

s_Q

Call node

f_1

n_6

| Call Q |

n_7

f_6

f_{x2r}

e_Q

Exit node

Return node
void main() {
    int x;
    c1: x = p(7);
    c2: x = p(9);
}

int p(int a) {
    return a + 1;
}
void main() {
    int x;
    c1: x = p(7);
    c2: x = p(9);
}

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

Simple Example
void main() {
    int x;
    c1: x = p(7);
    c2: x = p(9);
}

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

void main() {
    int x ;
    c1: x = p(7); ←
    ε: x ↦ 8
    c2: x = p(9) ;
}

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

Simple Example

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

int p(int a) {
    c1:[a \rightarrow 7]
    return a + 1;
    c1:[a \rightarrow 7, $$ \rightarrow 8$]
}
```
void main() {
    int x;
    c1: x = p(7);
    ε: [x → 8]
    c2: x = p(9);
}

int p(int a) {
    c1:[a → 7]
    c2:[a → 9]
    return a + 1;
    c1:[a → 7, $$ → 8]$
}

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

int p(int a) {
    c1:[a ↦ 7]
    c2:[a ↦ 9]
    return a + 1;
    c1:[a ↦ 7, $$ ↦ 8]$
    c2:[a ↦ 9, $$ ↦ 10]
}
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 a + 1;
    c1:[a → 7, $$ → 8]
    c2:[a → 9, $$ → 10]
}

Simple Example
The Call-String Approach

• The data flow value is associated with sequences of calls (call string)
• Use Chaotic iterations over the supergraph
• To guarantee termination limit the size of call string (typically 1 or 2)
  – Represents tails of calls

• Abstract inline
Another Example ($|cs|=2$)

```c
void main() {
    int x;
    c1: x = p(7);
    ε : [x ↦ 16]
    c2: x = p(9);
    ε : [x ↦ 20]
}

int p(int a) {
    c1:[a ↦ 7]
    c2:[a ↦ 9]
    return c3: p1(a + 1);
    c1:[a ↦ 7, $$ ↦ 16]
    c2:[a ↦ 9, $$ ↦ 20]
}

int p1(int b) {
    c1.c3:[b ↦ 8]
    c2.c3:[b ↦ 10]
    return 2 * b;
    c1.c3:[b ↦ 8, $$ ↦ 16]
    c2.c3:[b ↦ 10, $$ ↦ 20]
}
```
Another Example ($|cs|=1$)

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

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

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

```c
void main() {
  c1: p(7);
  ε: [x ↦ Τ]
}

int p(int a) {
  c1: [a ↦ 7]  c1.c2+: [a ↦ τ]
  if (...) {
    c1: [a ↦ 7]  c1.c2+: [a ↦ τ]
    a = a - 1;
    c1: [a ↦ 6]  c1.c2+: [a ↦ τ]
    c2: p(a);
    c1.c2*: [a ↦ τ]
    a = a + 1;
    c1.c2*: [a ↦ τ]
  }
  c1.c2*: [a ↦ τ]
  x = -2*a + 5;
  c1.c2*: [a ↦ τ, x ↦ τ]
}
```
Summary Call-String

• Easy to implement
• Efficient for very small call strings
• Limited precision
  – Often loses precision for recursive programs
  – For finite domains can be precise even with recursion (with a bounded callstring)

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

• The meaning of a procedure is mapping from states into states
• The abstract meaning of a procedure is function from an abstract state to abstract states
• Relation between input and output
• In certain cases can compute JVP
The Functional Approach

• Two phase algorithm
  – Compute the dataflow solution at the exit of a procedure as a function of the initial values at the procedure entry (functional values)
  – Compute the dataflow values at every point using the functional values
void main() {
    p(7);
}

int p(int a) {
    [a ↦ a₀, x ↦ x₀]
    if (…) {
        [a ↦ a₀, x ↦ x₀]
        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 ↦ x₀][a ↦ a₀, x ↦ τ]
    x = -2*a + 5;
    [a ↦ a₀, x ↦ -2*a₀+5]
Phase 2

```c
int p(int a) {
    if (...) {
        a = a - 1;
        p(a);
        a = a + 1;
    }
    x = -2*a + 5;
}
void main() {
    p(7);
    [x ← -9]
}

p(a₀,x₀) = [a ← a₀, x ← -2a₀ + 5]
```
Tabulation for finite lattices $L$

- **Data facts**: $d \in L \times L$
- **Initialization**:
  - $f_{\text{start}, \text{start}} = (\tau, \tau)$; otherwise $(\bot, \bot)$
  - $S[\text{start}, \tau] = \tau$

- **Propagation of** $(x, y)$ over edge $e = (n, n')$
  - **Maintain summary**: $S[n', x] = S[n', x] \sqcup [n, n'] (y))$
  - **n intra-node**: $\Rightarrow n' : (x, [n, n'] (y))$
  - **n call-node**:
    - $\Rightarrow n' : (y, y)$ *if* $S[n', y] = \bot$ and $n' = \text{entry node}$
    - $\Rightarrow n' : (x, z)$ *if* $S[\text{exit} (\text{call} (n), y] = z$ and $n' = \text{ret-site-of } n$
  - **n return-node**: $\Rightarrow n' : (u, y) ; n_c = \text{call-site-of } n' , S[n_c, u] = x$
Issues in Functional Approach

• How to guarantee that finite height for functional lattice?
  – It may happen that L has finite height and yet the lattice of monotonic function from L to L do not

• Efficiently represent functions
  – Functional join
  – Functional composition
  – Testing equality
Summary Functional approach

• Computes procedure abstraction
• Sharing between different contexts
• Rather precise
• Recursive procedures may be more precise/efficient than loops
• But requires more from the implementation
  – Representing (input/output) relations
  – Composing relations
CFL-Graph reachability [RHS’95]

- Static analysis of programs with procedures
- Special cases of functional analysis
- Reduce the interprocedural analysis problem to finding context free reachability

[RHS’95] Thomas W. Reps, Susan Horwitz, Shmuel Sagiv: Precise Interprocedural Dataflow Analysis via Graph Reachability. POPL 1995
The Context-Free Reachability Problem

- A finite directed graph $G(s, V, E)$
- A finite alphabet $\Sigma$
- A labeling function $l: E \rightarrow \Sigma$
- A context-free grammar $C$ over $\Sigma$
- A property **holds** at $n \in N$ if there exists a path from $s$ to $n$ whose labels are in $C$
Might \( y \) be uninitialized here?

YES!

Might \( b \) be uninitialized here?

NO!
Questions

• When can we reduce a static analysis problem to CFL reachability
• What is the complexity of solving CFL reachability problems
• Interesting generalizations of CFL reachability
IFDS 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

• Can be generalized to certain infinite lattices
Possibly Uninitialized Variables

Start

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

x = 3

\{w,y\} → \lambda \mathcal{V}. \mathcal{V} - \{x\}

if . . .

\lambda \mathcal{V}. \mathcal{V}

\{w,y\} → \lambda \mathcal{V}. \mathcal{V}

\lambda \mathcal{V} \cdot \mathcal{V} - \{w\}

y = x

\{w\} \rightarrow \{w\}

\lambda \mathcal{V}. \text{if } x \in \mathcal{V} \text{ then } \mathcal{V} \cup \{y\} \text{ else } \mathcal{V} - \{y\}

w = 8

\lambda \mathcal{V}. \mathcal{V} - \{w\}

printf(y)

\{w,y\} \rightarrow \{w,y\}

\lambda \mathcal{V}. \text{if } w \in \mathcal{V} \text{ then } \mathcal{V} \cup \{y\} \text{ else } \mathcal{V} - \{y\}

\{w\} \rightarrow \{w\}

y = w

\{w,y\} \rightarrow \{w,y\}

\lambda \mathcal{V} \cdot \mathcal{V} - \{w\}

\{w,y\} \rightarrow \{w,y\}

\{w\} \rightarrow \{w\}

\{w,y\} \rightarrow \{w,y\}
Efficiently Representing Functions

- Let $f : 2^D \rightarrow 2^D$ be a distributive function
- Then:
  - $f(X) = \{ f(\{z\}) \mid z \in X \}$
  - $f(X) = f(\emptyset) \cup \{ f(\{z\}) \mid z \in X \}$
Encoding Transfer Functions

- Enumerate all input space and output space
- Represent functions as graphs with $2(D+1)$ nodes
- Special symbol “0” denotes empty sets (sometimes denoted $\Lambda$)
- Example: $D = \{ a, b, c \}$
  \[ f(S) = (S - \{a\}) \cup \{b\} \]

![Graph representation of the example function](image)
Representing Dataflow Functions

Identity Function

\[ f = \lambda V. V \]

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

Constant Function

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

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

“Gen/Kill” Function

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

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

Non-“Gen/Kill” Function

\[ f = \lambda V. \begin{cases} 
\text{if } a \in V 
& \text{then } V \cup \{b\} \\
\text{else } V - \{b\} 
\end{cases} \]

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

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

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

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

\text{start main}
\[
x = 3
\]

\text{p}(x,y)
\text{return from p}
\text{printf}(y)
\text{exit main}

\text{start p}(a,b)

\text{if } \ldots
\text{b = a}
\text{p}(a,b)
\text{return from p}
\text{printf}(b)
\text{exit p}
\( x = 3 \)  
\( p(x,y) \)  
return from \( p \)  
exit \( p \)  

Might \( y \) be uninitialized here?  

\( b = a \)  
\( p(a,b) \)  
return from \( p \)  
exit \( p \)  

different diagram
The 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
Asymptotic Running Time

• CFL-reachability
  – Exploded control-flow graph: \( ND \) nodes
  – Running time: \( O(N^3D^3) \)

• Exploded control-flow graph \( \rightarrow \) Special structure

Running time: \( O(ED^3) \)

Typically: \( E \approx N \), hence \( O(ED^3) \approx O(ND^3) \)

“Gen/kill” problems: \( O(ED) \)
Some Applications

Mayur Naik: Jchord a static analysis for Java
IBM Watson: Wala static analysis tool

Thomas Ball, Vladimir Levin, Sriram K. Rajaman
A decade of software model checking with SLAM. CACM’11

Manu Sridharan, Rastislav Bodík: Refinement-based context-sensitive points-to analysis for Java. PLDI 2006

Nomair A. Naeem, Ondrej Lhoták, Jonathan Rodriguez: Practical Extensions to the IFDS Algorithm. CC 2010: 124-144

Osbert Bastani, Saswat Anand, Alex Aiken: Specification Inference Using Context-Free Language Reachability, POPL’15

K Chatterjee, A Pavlogiannis, Y Velner: Quantitative interprocedural analysis, POPL’15

S Yang, D Yan, H Wu, Y Wang: Static control-flow analysis of user-driven callbacks in Android applications
Asymptotic Running Time

- CFL-reachability
  - Exploded control-flow graph: $ND$ nodes
  - Running time: $O(N^3D^3)$
- Exploded control-flow graph → Special structure

Running time: $O(ED^3)$

Typically: $E \approx N$, hence $O(ED^3) \approx O(ND^3)$

“Gen/kill” problems: $O(ED)$
IDE

• Goes beyond IFDS problems
  – Can handle unbounded domains
• Requires special form of the domain
• Can be much more efficient than IFDS
Example Linear Constant Propagation

• Consider the constant propagation lattice
• The value of every variable y at the program exit can be represented by:
  \[ y = \bigcup \{(a_x x + b_x) \mid x \in \text{Var}^*\} \sqcup c \]
  \[ a_x, c \in \mathbb{Z} \star \{\perp, \tau\} \quad b_x \in \mathbb{Z} \]
• Supports efficient composition and “functional” join
  – \([z := a \ast y + b]\)
  – What about \([z := x + y]\)?
IDE Analysis

• Point-wise representation closed under composition
• CFL-Reachability on the exploded graph
• Two phase algorithm
  – Compose functions
  – Compute dataflow values
Linear constant propagation

Point-wise representation of environment transformers
declare x: integer
program main
begin
    call P(7)
    print (x) /* x is a constant here */
end

procedure P (value a: integer)
begin /* a is not a constant here */
    if a > 0 then
        a := a - 2
        call P (a)
        a := a + 2
    fi
    x := -2 * a + 5
    /* x is not a constant here */
end
Costs

- $O(ED^3)$
- Class of value transformers $F \subseteq L \rightarrow L$
  - $id \in F$
  - Finite height
- Representation scheme with (efficient)
  - Application
  - Composition
  - Join
  - Equality
  - Storage
Conclusion

• Handling functions is crucial for abstract interpretation
• Virtual functions and exceptions complicate things
• But scalability is an issue
  – Small call strings
  – Small functional domains
  – Demand analysis
Bibliography

- Textbook 2.5
- Two Approaches to interprocedural analysis by Micha Sharir and Amir Pnueli: 1
- IDFS Interprocedural Distributive Finite Subset Precise interprocedural dataflow analysis via graph reachability. *Reps, Horowitz, and Sagiv, POPL’95*
- IDE Interprocedural Distributive Environment Precise interprocedural dataflow analysis with applications to constant propagation. *Sagiv, Reps, Horowitz, and TCS'96*