Lecture 06 – Inter-procedural Analysis

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav
Previously

- Verifying absence of buffer overruns
- (requires) Heap abstractions
- (requires) Combining heap and numerical information
Today

- LFP computation and join-over-all-paths
- Inter-procedural analysis

Acks

- Some slides adapted from Noam Rinetzky and Mooly Sagiv
- DNA of some slides traces back to Tom Reps
Join over all paths (JOP)

- Propagate analysis information along paths
- A path is a sequence of edges in the CFG \([e_1, e_2, \ldots, e_n]\)
- Can define the transfer function for a path via transformer composition
  - let \(f_i\) denote \(f(e_i)\)
  - \(f[e_1, e_2, \ldots, e_n] = f_n \circ \ldots \circ f_2 \circ f_1\)

- Consider the (potentially infinite) set of paths reaching a label in the program from entry. Denote by Paths-to-in(entry, l) the set of paths from entry to label l, similarly for paths-to-out

- The result at program point l can be defined as
  \[
  \text{JOPin}(l) = \square \{ f[p](\text{initial}) \mid p \in \text{paths-to-in(entry, l)} \}
  \]
  \[
  \text{JOPout}(l) = \square \{ f[p](\text{initial}) \mid p \in \text{paths-to-out(entry, l)} \}
  \]
The lfp computation approximates JOP

- JOPout(l) = □ \{ f[p](initial) | p ∈ paths-to-out(entry,l) \}
- Set of paths potentially infinite (non computable)
- The lfp computation take joins “earlier”
  - Merging sub-paths
- For a monotone function
  - f(x □ y) ⊇ f(x) □ f(y)
lfp computation and JOP

Paths transformers:
- $f[e_1, e_2, e_3, e_4]$ (initial)
- $f[e_1, e_2, e_7, e_8]$ (initial)
- $f[e_5, e_6, e_7, e_8]$ (initial)
- $f[e_5, e_6, e_3, e_4]$ (initial)

JOP:
- $f[e_1, e_2, e_3, e_4] \cup f[e_5, e_6]$ (initial)
- $f[e_1, e_2, e_7, e_8] \cup f[e_5, e_6, e_7, e_8]$ (initial)
- $f[e_5, e_6, e_3, e_4]$ (initial)

“lfp” computation:
- $j_1 = f[e_1, e_2]$ (initial) $\cup f[e_5, e_6]$ (initial)
- $j_2 = f[e_3, e_4](j_1) \cup f[e_7, e_8](j_1)$

(wrote “lfp” just to emphasize that no iteration is required here, no loops)
Interprocedural Analysis

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

- Stack can grow without a bound
- Matching of call/return
Solution Attempt #1

- Inline callees into callers
  - End up with one big procedure
  - CFGs of individual procedures = duplicated many times

- Good: it is precise
  - distinguishes different calls to the same function

- Bad
  - exponential blow-up, not efficient
  - doesn’t work with recursion
Simple Example

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

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

void main() {
    int x;
    x = 7; x = x + 1;
    [x↦8]
    x = 9; x = x + 1;
    [x↦10]
}

Inlining: Exponential Blowup

```c
main() {
    f1();
    f1();
}
f1() {
    f2();
    f2();
}
...
fn() { ... }
```
Solution Attempt #2

- Build a “supergraph” = inter-procedural CFG
- Replace each call from P to Q with
  - An edge from point before the call (call point) to Q’s entry point
  - An edge from Q’s exit point to the point after the call (return pt)
  - Add assignments of actuals to formals, and assignment of return value
- Good: efficient
  - Graph of each function included exactly once in the supergraph
  - Works for recursive functions (although local variables need additional treatment)
- Bad: imprecise, “context-insensitive”
  - The “unrealizable paths problem”: dataflow facts can propagate along infeasible control paths
Simple Example

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

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

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

int p(int a) {
    [a => 7]
    return a + 1;
}
```
Simple Example

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

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

```java
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

```c
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

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

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 -> t]
    return a + 1;
    [a -> t, $$ -> t]
}

Simple Example
Simple Example

```c
int p(int a) {
    [a \mapsto \tau]
    return a + 1;
    [a \mapsto \tau, $$ \mapsto \tau]
}

void main() {
    int x;
    x = p(7); \iff
    [x \mapsto \tau]
    x = p(9); \iff
    [x \mapsto \tau]
}
```
Unrealizable Paths

foo()

Call bar()

bar()

Call bar()

zoo()
IVP: Interprocedural Valid Paths

- IVP: all paths with matching calls and returns
- And prefixes
Valid Paths
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 (i matched )i | \( \varepsilon \)
  - valid ::= valid (i matched | matched
    - *paths* can be defined by a regular expression
The Join-Over-Valid-Paths (JVP)

- \( \text{vpaths}(n) \) all valid paths from program start to \( n \)
- \( \text{JVP}[n] = \bigcap \{[e_1, e_2, \ldots, e] \text{ (initial)} \mid (e_1, e_2, \ldots, e) \in \text{vpaths}(n) \} \)
- \( \text{JVP} \subseteq \text{JFP} \)
  - In some cases the JVP can be computed
  - (Distributive problem)
Sharir and Pnueli ‘82

- Call String approach
  - Blend interprocedural flow with intra procedural flow
  - Tag every dataflow fact with call history

- Functional approach
  - Determine the effect of a procedure
    - E.g., in/out map
  - Treat procedure invocations as “super ops”
The 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 \(c\)
begin
  proc p() is
    [x := a + 1]
  end

  [a=7]
  [call p]
  [print x]
  [a=9]
  [call p]
  [print a]
end
begin
proc p() is
    if [b] then ( [a := a - 1] [call p()] [a := a + 1] )
    [x := -2 * a + 5]
end
[a=7; [call p()] [print(x)]
end
Simple Example

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

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

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

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

```c
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]
}
```
Simple Example

```c
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]
}
```
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
Simple Example

```plaintext
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]
}
```
Simple Example

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]
}
Simple Example

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

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

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

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

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

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

- Simple
- Only feasible for very short call strings
  - exponential in call-string length
- Often loses precision under recursion
  - although can still be precise in some cases
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
Functional Approach: Main Idea

- Iterate on the abstract domain of functions from $L$ to $L$
- 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
- Computes JVP for distributive problems
Phase 1

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

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

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

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

int p(int a) {
    [a → a₀, x → x₀]
    if (...) {
        a = a - 1;
        p(a);
        a = a + 1;
    }
    [a → a₀, x → x₀]
    \[x = -2a + 5;\]
    \[a → a₀, x → -2a₀ + 5]\}
```
**Phase 1**

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

int p(int a) {
    [a → a₀, x → x₀]
    if (...) {
        a = a - 1;
        [a → a₀ - 1, x → x₀]
        p(a);
        a = a + 1;
    }
    [a → a₀, x → x₀]
    x = -2*a + 5;
    [a → a₀, x → -2*a₀ + 5]
}
```
Phase 1

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

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 ↦ x₀]
    x = -2*a + 5;
    [a ↦ a₀, x ↦ -2a₀ + 5]
}
```
Phase 1

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

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 ↦ x₀]
    x = -2*a + 5;
    [a ↦ a₀, x ↦ -2a₀ + 5]
}
```
Phase 1

```c
int p(int a) {
    [a \mapsto a_0, x \mapsto x_0]
    if (...) {
        a = a - 1;
        [a \mapsto a_0 - 1, x \mapsto x_0]
        p(a);
        [a \mapsto a_0 - 1, x \mapsto -2a_0 + 7]
        a = a + 1;
        [a \mapsto a_0, x \mapsto -2a_0 + 7]
    }\rightarrow
    [a \mapsto a_0, x \mapsto \pi]
    x = -2*a + 5;
    [a \mapsto a_0, x \mapsto -2a_0 + 5]
}
```
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);
    [x ↦ -9]
}
```

```c
int p(int a) {
    [a ↦ 7, x ↦ 0]
    if (...) {
        a = a -1;
        p (a);
        a = a + 1;
    }
    x = -2*a + 5;
}
```

\[ p(a_0,x_0) = [a ↦ a_0, x ↦ -2a_0 + 5] \]
Phase 2

```c
void main() {
    p(7);
    [x ↦ -9]
}

int p(int a) {
    [a ↦ 7, x ↦ 0]
    if (...) {
        a = a - 1;
        p (a);
        a = a + 1;
    }
    [a ↦ 7, x ↦ 0]
    x = -2*a + 5;
}

p(a₀,x₀) = [a ↦ a₀, x ↦ -2a₀ + 5]
```
Phase 2

void main() {
    p(7);
    [x → -9]
}

int p(int a) {
    [a → 7, x → 0]
    if (...) {
        a = a - 1;
        p(a);
        a = a + 1;
    }
    [a → 7, x → 0]
    x = -2*a + 5;
    [a → 7, x → -9]
}

\[
p(a_0, x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]
\]
Phase 2

```c
void main() {
    p(7);
    [x ⇝ -9]
}
```

```c
int p(int a) {
    [a ⇝7, x ⇝0]
    if (…) {
        a = a - 1;
        [a ⇝6, x ⇝0]
        p(a);
        a = a + 1;
    }
    [a ⇝7, x ⇝0]
    x = -2*a + 5;
    [a ⇝7, x ⇝-9]
}
```

\[ p(a_0, x_0) = [a ⇝ a_0, x ⇝ -2a_0 + 5] \]
Phase 2

void main() {
    p(7);
    [x ← -9]
}

int p(int a) {
    [a ← 7, x ← 0]
    if (…) {
        a = a - 1;
        [a ← 6, x ← 0]
        p (a);
        [a ← 6, x ← -9]
        a = a + 1;
    }
    [a ← 7, x ← 0]
    x = -2*a + 5;
    [a ← 7, x ← -9]
}

p(a_0, x_0) = [a ← a_0, x ← -2a_0 + 5]
Phase 2

```c
void main() {
    p(7);
    [x ← -9]
}

p(a, x) = [a ← a, x ← 2a + 5]
```

```c
int p(int a) {
    [a ← 7, x ← 0]
    if (…) {
        a = a - 1;
        [a ← 6, x ← 0]
        p(a);
        [a ← 6, x ← -9]
        a = a + 1;
        [a ← 7, x ← -9]
    }
    [a ← 7, x ← 0]
    x = -2*a + 5;
    [a ← 7, x ← -9]
}
```
Phase 2

int p(int a) {
    [a → 7, x → 0]
    if (...) {
        a = a - 1;
        [a → 6, x → 0]
        p (a);
        [a → 6, x → -9]
        a = a + 1;
        [a → 7, x → -9]
    }
    [a → 7, x → ∑]
    x = -2*a + 5;
    [a → 7, x → -9]
}

void main() {
    p(7);
    [x → -9]
}

p(a_0, x_0) = [a → a_0, x → -2a_0 + 5]
Phase 2

```
void main() {
  p(7);
  [x ↦ -9]
}

p(a₀, x₀) = [a ↞ a₀, x ↞ -2a₀ + 5]
```

```
int p(int a) {
  [a ↞ 7, x ↞ 0]
  if (...) {
    a = a - 1;
    [a ↞ 6, x ↞ 0]
  } else {
    p(a);
    [a ↞ 6, x ↞ -9]
    a = a + 1;
    [a ↞ 7, x ↞ -9]
  }
  [a ↞ 7, x ↞ τ]
  x = -2*a + 5;
  [a ↞ 7, x ↞ -9]
}
```
Phase 2

```c
void main() {
    p(7);
    [x ⌷ -9]
}
```

```c
int p(int a) {
    [a ⌷ 7, x ⌷ 0] [a ⌷ 6, x ⌷ 0]
    if (…) {
        a = a - 1;
        [a ⌷ 6, x ⌷ 0]
        p (a);
        [a ⌷ 6, x ⌷ -9]
        a = a + 1;
        [a ⌷ 7, x ⌷ -9]
    }
    [a ⌷ 7, x ⌷ τ]
    x = -2*a + 5;
    [a ⌷ 7, x ⌷ -9]
}
```

\[ p(a_0, x_0) = [a \mapsto a_0, \ x \mapsto -2a_0 + 5] \]
void main() {
    p(7);
    [x ↦ -9]
}

p(a_0, x_0) = [a ↦ a_0, x ↦ -2a_0 + 5]
Phase 2

```c
int p(int a) {
    [a ↦ τ, x ↦ 0]
    if (...) {
        a = a - 1;
        [a ↦ τ, x ↦ 0]
        p (a);
        [a ↦ 6, x ↦ 9]
        a = a + 1;
        [a ↦ 7, x ↦ 9]
    }
    [a ↦ 7, x ↦ τ]
    x = -2*a + 5;
    [a ↦ 7, x ↦ -9]
}
```

\( p(a_0, x_0) = [a ↦ a_0, x ↦ -2a_0 + 5] \)
Phase 2

\[
\text{void main()} \{ \\
\text{p(7);} \\
[x \mapsto -9] \\
\}
\]

\[
\text{int p(int a) \{} \\
[a \mapsto \text{t}, x \mapsto 0] \\
\text{if (…) \{} \\
\quad a = a - 1; \\
[a \mapsto \text{t}, x \mapsto 0] \\
\quad \text{p(a);} \\
[a \mapsto \text{t}, x \mapsto \text{t}] \\
\quad a = a + 1; \\
[a \mapsto 7, x \mapsto -9] \\
\quad \} \\
[a \mapsto 7, x \mapsto \text{t}] \\
\text{x = -2*a + 5;} \\
[a \mapsto 7, x \mapsto -9] \\
\} \\
\]

\[
p(a_0, x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]
\]
Phase 2

```c
void main() {
    p(7);
    [x ↦ -9]
}

p(a₀, x₀) = [a ↦ a₀, x ↞ -2a₀ + 5]
```

```c
int p(int a) {
    [a ↦ t, x ↞ 0]
    if (...) {
        a = a - 1;
        [a ↦ t, x ↞ 0]
        p(a);
        [a ↦ t, x ↦ t]
        a = a + 1;
        [a ↦ t, x ↦ t]
    }
    [a ↦ 7, x ↦ t]
    x = -2*a + 5;
    [a ↦ 7, x ↞ -9]
}
```
Phase 2

```c
void main() {
    p(7);
    [x ← -9]
}
```

```c
int p(int a) {
    [a ← t, x ← 0]
    if (...) {
        a = a - 1;
        [a ← t, x ← 0]
        p (a);
        [a ← t, x ← t]
        a = a + 1;
        [a ← t, x ← t]
    }
    [a ← t, x ← t]
    x = -2*a + 5;
    [a ← 7, x ← -9]
}
```

\[ p(a_0, x_0) = [a ← a_0, x ← -2a_0 + 5] \]
Phase 2

```c
void main() {
    p(7);
    [x ↔ -9]
}

p(a₀, x₀) = [a ↔ a₀, x ↔ -2a₀ + 5]
```

```c
int p(int a) {
    [a ↔ t, x ↔ 0]
    if (...) {
        a = a - 1;
        [a ↔ t, x ↔ 0]
        p(a);
        [a ↔ t, x ↔ t]
        a = a + 1;
        [a ↔ t, x ↔ t]
    }
    [a ↔ t, x ↔ t]
    x = -2*a + 5;
    [a ↔ t, x ↔ t]
}
```
Issues in Functional Approach

- How to guarantee finite height for functional lattice?
  - Possible 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
Tabulation

- Special case: L is finite
- Data facts: \( d \in L \times L \)
- Initialization:
  - \( f_{\text{start}, \text{start}} = (T, T) \); otherwise \((\bot, \bot)\)
  - \( S[\text{start}, T] = T \)

- Propagation of \((x, y)\) over edge \( e = (n, n') \)
  - Maintain summary: \( S[n', x] = S[n', x] \sqcup [n] (y) \)
  - n intra-node: \( \Rightarrow n': (x, [n] (y)) \)
  - n call-node: \( \Rightarrow n': (y, y) \) if \( S[n', y] = \bot \) and \( n' = \) entry node
    \( \Rightarrow n': (y, z) \) if \( S[n', y] = z \) and \( n' = \) rest-site-of \( n \)
  - n return-node: \( \Rightarrow n': (u, y) ; n_c = \) call-site-of \( n' \), \( S[n_c, u] = x \)
IFDS

- **IFDS** Interprocedural Distributive Finite Subset Precise interprocedural dataflow analysis via graph reachability. *Reps, Horowitz, and Sagiv, POPL’95*
IFDS Problems

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

- Efficient solution through formulation as CFL reachability
Possibly Uninitialized Variables

```
x = 3
if ... y = x
w = 8
```

```
printf(y)
```
Encoding Transfer Functions

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

![Graph Example]

\[ \begin{align*}
  o & \quad a & \quad b & \quad c \\
  o & \quad o & \quad o & \quad o \\
  o & \quad a & \quad b & \quad c \\
\end{align*} \]
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

\[ f = \lambda V. \text{if } a \in V \]
\[ \text{then } V \cup \{b\} \]
\[ \text{else } V - \{b\} \]
\[ f(\{a, b\}) = \{a, b\} \]
Exploded Supergraph

- Exploded supergraph
  - Start with supergraph
  - Replace each node by its graph representation
  - Add edges between corresponding elements in D at consecutive program points

- CFL reachability
  - Finding MOVP solution is equivalent to computing CFL reachability over the exploded supergraph using the valid parentheses grammar.
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
Composing Dataflow Functions

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

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

\[ f_2 \circ f_1(\{a, c\}) = \{c\} \]
\[
x = 3
\]
This results in:
\[
p(x, y)
\]
Now, considering the context:

**Might y be uninitialized here?**

- **Yes!**
  - **YES!**
    - **printf(y)**
    - **exit main**

**Might b be uninitialized here?**

- **No!**
  - **b = a**
  - **p(a, b)**
  - **return from p**
  - **printf(b)**
  - **NO!**

The flow of execution is as follows:

1. Start main
2. \( x = 3 \)
3. \( p(x, y) \)
4. Might y be uninitialized here? (Yes)
5. printf(y)
6. Exit main
7. Might b be uninitialized here? (No)
8. b = a
9. p(a, b)
10. return from p
11. printf(b)
12. NO!
Interprocedural Dataflow Analysis via CFL-Reachability

- **Graph:** Exploded control-flow graph
- **$L$:** $L(unbalLeft)$
  - $unbalLeft = valid$
- **Fact:** $d$ holds at $n$ iff there is an $L(unbalLeft)$-path from $<\text{Start main, } \Lambda>$ to $<n,d>$
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: $O(ED)$
Recap

- inter-procedural analysis
- inlining
- call-string approach
- functional approach
begin
proc p() is
    if \([b]\) then (\[a := a - 1\]; \[call p()\]; \[a := a + 1\]) \[x := -2 \times a + 5\]
end
\[a=7\]; \[call p()\]; \[print(x)\]
end

\(\lambda e.\ [x \mapsto -2e(a) + 5, a \mapsto e(a)]\)
begin
proc p() is
    if [b] then (  
        [a := a - 1]  
        [call p()]  
        [a := a + 1]  
    )  
    [x := -2*a + 5]  
end

[read(a)] ; [call p()] ; [print(x)]
end
Example: Constant propagation

- $L = \text{Var} \rightarrow \mathbb{N} \cup \{\top, \bot\}$
- Domain: $F : L \rightarrow L$
  - $(f_1 \sqcup f_2)(x) = f_1(x) \sqcup f_2(x)$

\[
\begin{align*}
\lambda_{\text{env}}.\text{env}[x\mapsto 7] & \quad \lambda_{\text{env}}.\text{env}[x\mapsto 7] \circ \lambda_{\text{env}}.\text{env} \\
\lambda_{\text{env}}.\text{env}[y\mapsto \text{env}(x)+1] & \quad \lambda_{\text{env}}.\text{env}[y\mapsto \text{env}(x)+1] \circ \lambda_{\text{env}}.\text{env}[x\mapsto 7] \circ \lambda_{\text{env}}.\text{env}
\end{align*}
\]
Example: Constant propagation

- $L = \text{Var} \rightarrow \mathbb{N} \cup \{\top, \bot\}$
- Domain: $F: L \rightarrow L$
  - $(f_1 \sqcup f_2)(x) = f_1(x) \sqcup f_2(x)$

```
Id = \lambda \text{env}.\text{env}

x = 7
\lambda \text{env}.\text{env}[x \mapsto 7]

y = x + 1
\lambda \text{env}.\text{env}[y \mapsto \text{env}(x) + 1]

\lambda \text{env}.\text{env}[y \mapsto \text{env}(x) + 1] \sqcup \lambda \text{env}.\text{env}[x \mapsto 7]
```

$x = y$
init

begin^0
  a=7^9
  Call p^{10}

Call p^{11}

print(x)^{12}

end^{13}

If( ... )^2

a=a-1^3

Call p^4

Call p^5

a=a+1^6

x=-2a+5^7

end^8

<table>
<thead>
<tr>
<th>N</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\lambda e. [x \mapsto e(x), a \mapsto e(a)] = id)</td>
</tr>
<tr>
<td>1</td>
<td>(\lambda e. [x \mapsto e(x), a \mapsto e(a)] = id)</td>
</tr>
<tr>
<td>3-13</td>
<td>(\lambda e. \bot)</td>
</tr>
</tbody>
</table>
begin^0
a=7^9
Call p^{10}
Call p^{11}
print(x)^{12}
end^{13}

\textbf{Solution (F)}

<table>
<thead>
<tr>
<th>N</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(\lambda e. [x\mapsto e(x), a\mapsto e(a)]=\text{id})</td>
</tr>
<tr>
<td>1</td>
<td>(\lambda e. [x\mapsto e(x), a\mapsto e(a)]=\text{id})</td>
</tr>
<tr>
<td>2</td>
<td>id</td>
</tr>
<tr>
<td>3</td>
<td>id</td>
</tr>
<tr>
<td>4</td>
<td>(\lambda e. [x\mapsto e(x), a\mapsto e(a)-1])</td>
</tr>
<tr>
<td>5</td>
<td>(f_8 \circ \lambda e. [x\mapsto e(x), a\mapsto e(a)-1] = \lambda e. [x\mapsto -2(e(a)-1)+5, a\mapsto e(a)-1])</td>
</tr>
<tr>
<td>6</td>
<td>(\lambda e. [x\mapsto -2(e(a)-1)+5, a\mapsto e(a)-1])</td>
</tr>
<tr>
<td>7</td>
<td>(\lambda e. [x\mapsto -2(e(a)-1)+5, a\mapsto e(a)] \square \lambda e. [x\mapsto e(x), a\mapsto e(a)])</td>
</tr>
<tr>
<td>8</td>
<td>(\lambda a, x. [x\mapsto -2e(a)+5, a\mapsto e(a)])</td>
</tr>
<tr>
<td>9</td>
<td>a=7</td>
</tr>
<tr>
<td>10</td>
<td>(\lambda e. [x\mapsto e(x), a\mapsto 7])</td>
</tr>
<tr>
<td>11</td>
<td>(\lambda a, x. [x\mapsto -2e(a)+5, a\mapsto e(a)] \circ f_{10})</td>
</tr>
</tbody>
</table>

\(\lambda \\text{e.} \ [x\mapsto \text{e}(x), \ a\mapsto \text{e}(a)]=\text{id}\)
Solution (D)

```
begin^0
  a=7^9
  Call p^{10}
  Call p^{11}
  print(x)^{12}
  end^{13}

p^1
  If( ... )^2
    a=a-1^3
    Call p^4
    Call p^5
    a=a+1^6
    x=-2a+5^7
    end^8
```

```
<table>
<thead>
<tr>
<th>N</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[x→0, a→0]</td>
</tr>
<tr>
<td>10</td>
<td>[x→0, a→7]</td>
</tr>
<tr>
<td>11</td>
<td>[x→-9, a→7]</td>
</tr>
</tbody>
</table>
```
Interprocedural Analysis

begin
proc p() is
    [x := a + 1]
end
[a=7]
[call p()]
[print x]
[a=9]
[call p()]
[print a]
end

- Extend language with begin/end and with [call p()]\textsuperscript{lab}_{rlab}
- Call label clab, and return label rlab
Work-list Algorithm

Chaotic(G(V, E): CFG, s: Node, L: Lattice, ℓ: L, f: E \rightarrow (L \rightarrow L)) 

for each v in V to n do \(df_{\text{entry}}[v] := \perp\)

\(df[v] = \ell\)

\(WL = \{s\}\)

while (WL ≠ \(\emptyset\)) do

select and remove an element u ∈ WL

for each v, such that (u, v) ∈ E do

\(\text{temp} = f(e)(df_{\text{entry}}[u])\)

\(\text{new} := df_{\text{entry}}(v) \sqcup \text{temp}\)

if (new ≠ df_{\text{entry}}[v]) then

\(df_{\text{entry}}[v] := \text{new};\)

\(WL := WL \cup \{v\}\)
Complexity of Chaotic Iterations

- **Parameters**
  - $n$ the number of CFG nodes
  - $k$ is the maximum outdegree of edges
  - A lattice of height $h$
  - $c$ is the maximum cost of
    - applying $f_{(e)}$
    - $\sqcup$
    - $L$ comparisons

- **Complexity:** $O(n \times h \times c \times k)$

(slide from Mooly Sagiv)