Lecture 16 – Dataflow Analysis

THEORY OF COMPILATION

Eran Yahav

www.cs.technion.ac.il/~yahave/tocs2011/compilers-lec16.pptx

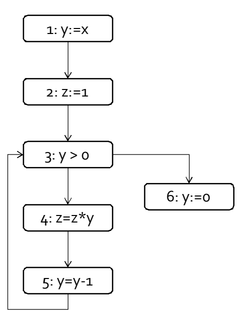
Reference: Dragon 9, 12

Last time… Dataflow Analysis

- Information flows along (potential) execution paths
- Conservative approximation of all possible program executions
- Can be viewed as a sequence of transformations on program state
  - Every statement (block) is associated with two abstract states: input state, output state
  - Input/output state represents all possible states that can occur at the program point
  - Representation is finite
  - Different problems typically use different representations

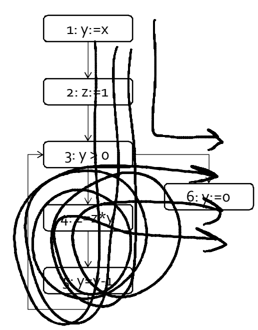
Control-Flow Graph

1: y := x;
2: z := 1;
3: while y > 0 {
4:   z := z * y;
5:   y := y – 1
}
6: y := 0



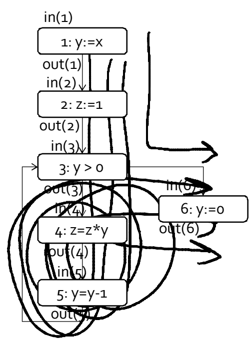
Executions

1: y := x;
2: z := 1;
3: while y > 0 {
4:   z := z * y;
5:   y := y – 1
}
6: y := 0

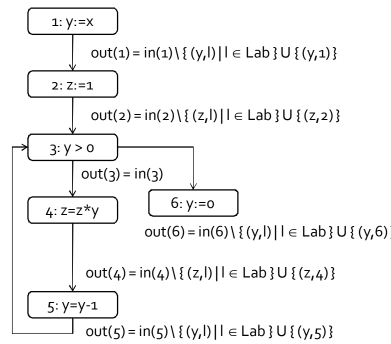


Input/output Sets

1: y := x;
2: z := 1;
3: while y > 0 {
4:   z := z * y;
5:   y := y – 1
}
6: y := 0



Transfer Functions



in(1) = { (x,?), (y,?), (z,?) }
in(2) = out(1)
in(3) = out(2) U out(5)
in(4) = out(3)
in(5) = out(4)
in(6) = out(3)

Kill/Gen formulation for Reaching Definitions

Block	out (lab)
$[x := a]^{lab}$	in(lab) \ { (x,l) | l ∈ Lab } U { (x,lab) }
$[skip]^{lab}$	in(lab)
$[b]^{lab}$	in(lab)

Block	kill	gen
$[x := a]^{lab}$	{ (x,l) | l ∈ Lab }	{ (x,lab) }
$[skip]^{lab}$	Ø	Ø
$[b]^{lab}$	Ø	Ø

For each program point, which assignments <u>may</u> have been made and not overwritten, when program execution reaches this point along <u>some path</u>.

Solving Gen/Kill Equations

```
OUT[ENTRY] = ∅;
for (each basic block B other than ENTRY) OUT[B] = ∅;
while (changes to any OUT occur) {
   for (each basic block B other than ENTRY) {
      OUT[B]= (IN[B] ∖ killB) ∪ genB
      IN[B] = ∪p∈pred(B)  OUT[p]
   }
}
```

- Designated block Entry with OUT[Entry]=∅
- pred(B) = predecessor nodes of B in the control flow graph

Available Expressions Analysis

$[x := a+b]^1$;
$[y := a*b]^2$;
while $[y > a+b]^3$ (
 $[a := a + 1]^4$;
 $[x := a + b]^5$
)

(a+b) always available at label 3

For each program point, which expressions must have already been computed, and not later modified, on all paths to the program point

9

Some required notation

blocks : Stmt → P(Blocks)
blocks($[x := a]^{lab}$) = {$[x := a]^{lab}$}
blocks($[skip]^{lab}$) = {$[skip]^{lab}$}
blocks(S1; S2) = blocks(S1) ∪ blocks(S2)
blocks(if $[b]^{lab}$ then S1 else S2) = {$[b]^{lab}$} ∪ blocks(S1) ∪ blocks(S2)
blocks(while $[b]^{lab}$ do S) = {$[b]^{lab}$} ∪ blocks(S)

FV: (BExp ∪ AExp) → Var
Variables used in an expression

AExp(a) = all non-unit expressions in the arithmetic expression a
similarly AExp(b) for a boolean expression b

10

Available Expressions Analysis

- Property space
  - in_{AE}, out_{AE}: Lab → $\wp$(AExp)
  - Mapping a label to set of arithmetic expressions available at that label
- Dataflow equations
  - Flow equations – how to join incoming dataflow facts
  - Effect equations - given an input set of expressions S, what is the effect of a statement

11

Available Expressions Analysis

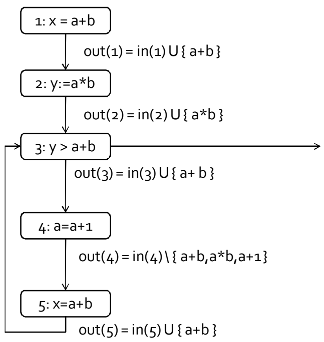
- in_{AE} (lab) =
  - ∅ when lab is the initial label
  - ∩ { out_{AE}(lab') | lab' ∈ pred(lab) } otherwise
- out_{AE} (lab) = …

Block	out (lab)
$[x := a]^{lab}$	in(lab) \ { a' ∈ AExp | x ∈ FV(a') } U { a' ∈ AExp(a) | x ∉ FV(a') }
$[skip]^{lab}$	in(lab)
$[b]^{lab}$	in(lab) U AExp(b)

From now on going to drop the AE subscript when clear from context

12

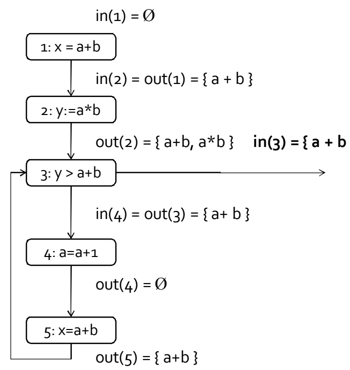
Transfer Functions



in(1) = Ø
in(2) = out(1)
in(3) = out(2) ∩ out(5)
in(4) = out(3)
in(5) = out(4)

```
[x := a+b]1;
[y := a*b]2;
while [y > a+b]3 (
  [a := a + 1]4;
  [x := a + b]5
)
```


Solution



Kill/Gen

Block	out (lab)
$[x := a]^{lab}$	in(lab) \ { a' ∈ AExp | x ∈ FV(a') } U { a' ∈ AExp(a) | x ∉ FV(a') }
$[skip]^{lab}$	in(lab)
$[b]^{lab}$	in(lab) U AExp(b)

Block	kill	gen
$[x := a]^{lab}$	{ a' ∈ AExp | x ∈ FV(a') }	{ a' ∈ AExp(a) | x ∉ FV(a') }
$[skip]^{lab}$	Ø	Ø
$[b]^{lab}$	Ø	AExp(b)

out(lab) = in(lab) \ kill(B^{lab}) U gen(B^{lab})

B^{lab} = block at label lab

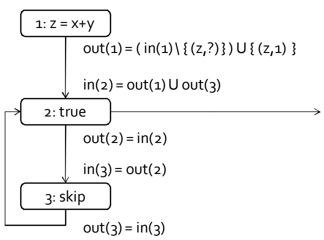
Reaching Definitions Revisited

Block	out (lab)
$[x := a]^{lab}$	in(lab) \ { (x,l) | l ∈ Lab } U { (x,lab) }
$[skip]^{lab}$	in(lab)
$[b]^{lab}$	in(lab)

Block	kill	gen
$[x := a]^{lab}$	{ (x,l) | l ∈ Lab }	{ (x,lab) }
$[skip]^{lab}$	Ø	Ø
$[b]^{lab}$	Ø	Ø

For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.

Why solution with smallest sets?



in(1) = { (x,?),(y,?),(z,?)}
in(2) = out(1) **U** out(3)
in(3) = out(2)

[z := x+y][1];
while [true][2] (
  [skip][3];
)

After simplification: in(2) = in(2) U { (x,?),(y,?),(z,1) }

Many solutions: any superset of { (x,?),(y,?),(z,1) }

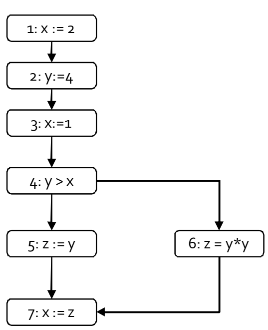
Live Variables

$[x := 2]^1$;
$[y := 4]^2$;
$[x := 1]^3$;
(if $[y>x]^4$ then $[z := y]^5$
else $[z := y*y]^6$);
$[x := z]^7$

For each program point, which variables may be live at the exit from the point.

Live Variables

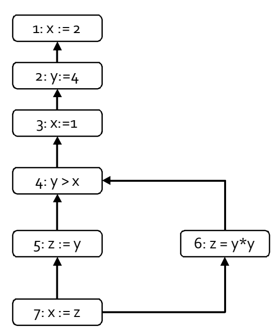
$[x := 2]^1$;
$[y := 4]^2$;
$[x := 1]^3$;
(if $[y>x]^4$ then $[z := y]^5$
else $[z := y*y]^6$);
$[x := z]^7$



Live Variables

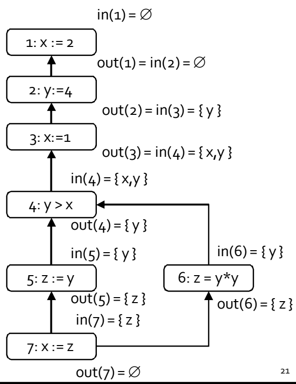
$[x := 2]^1$;
$[y := 4]^2$;
$[x := 1]^3$;
(if $[y>x]^4$ then $[z := y]^5$
else $[z := y*y]^6$);
$[x := z]^7$

Block	kill	gen
$[x := a]^{lab}$	{ x }	{ FV(a) }
$[skip]^{lab}$	Ø	Ø
$[b]^{lab}$	Ø	FV(b)



Live Variables: solution

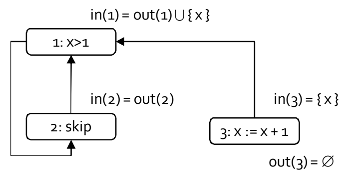
[x :=2]¹;
[y:=4]²;
[x:=1]³;
(if [y>x]⁴ then [z:=y]⁵
else [z:=y*y]⁶);
[x:=z]⁷



Block	kill	gen
$[x := a]^{lab}$	{ x }	{ FV(a) }
$[skip]^{lab}$	Ø	Ø
$[b]^{lab}$	Ø	FV(b)

21

Why solution with smallest set?



out(1) = in(2) ∪ in(3)
out(2) = in(1)
out(3) = ∅

while [x>1]¹ (
  [skip]²;
)
[x := x+1]³;

After simplification: in(1) = in(1) U { x }

Many solutions: any superset of { x }

22

Monotone Frameworks

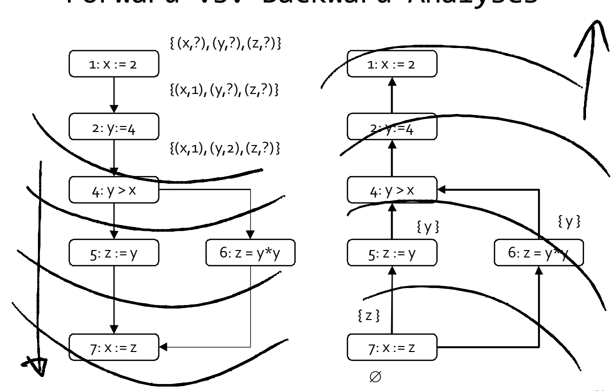
$$\text{In(lab)} = \begin{cases} \text{Initial} & \text{when lab} \in \text{Entry labels} \\ \sqcup \{ \text{out(lab')} \mid (\text{lab'},\text{lab}) \in \text{CFG edges} \} & \text{otherwise} \end{cases}$$

$$\text{out(lab)} = f_{lab}(\text{in(lab)})$$

- ⊔ is ∪ or ∩
- CFG edges go either forward or backwards
- Entry labels are either initial program labels or final program labels (when going backwards)
- Initial is an initial state (or final when going backwards)
- f_{lab} is the transfer function associated with the blocks B^{lab}

23

Forward vs. Backward Analyses



24

Must vs. May Analyses

- When ⊔ is ∩ - must analysis
  - Want largest sets that solves the equation system
  - Properties hold on all paths reaching a label (exiting a label, for backwards)
- When ⊔ is ∪ - may analysis
  - Want smallest sets that solve the equation system
  - Properties hold at least on one path reaching a label (existing a label, for backwards)

25

Example: Reaching Definition

- L = ℘(Var×Lab) is partially ordered by ⊆
- ⊔ is ∪
- L satisfies the Ascending Chain Condition because Var × Lab is finite (for a given program)

26

Example: Available Expressions

- L = ℘(AExp) is partially ordered by ⊇
- ⊔ is ∩
- L satisfies the Ascending Chain Condition because AExp is finite (for a given program)

27

Analyses Summary

	Reaching Definitions	**Available Expressions**	**Live Variables**
L	℘(Var x Lab)	℘(AExp)	℘(Var)
⊑	⊆	⊇	⊆
⊔	∪	∩	∪
⊥	∅	AExp	∅
Initial	{ (x,?) | x ∈ Var}	∅	∅
Entry labels	{ init }	{ init }	final
Direction	Forward	Forward	Backward
F	{ f: L → L | ∃k,g : f(val) = (val \ k) U g }		
f_{lab}	f_{lab}(val) = (val \ kill) ∪ gen		

28

Analyses as Monotone Frameworks

- Property space
  - Powerset
  - Clearly a complete lattice
- Transformers
  - Kill/gen form
  - Monotone functions (let's show it)

29

Monotonicity of Kill/Gen transformers

- Have to show that $x \sqsubseteq x'$ implies $f(x) \sqsubseteq f(x')$
- Assume $x \sqsubseteq x'$, then for kill set k and gen set g
  $(x \setminus k) \cup g \sqsubseteq (x' \setminus k) \cup g$
- Technically, since we want to show it for all functions in F, we also have to show that the set is closed under function composition

30

Distributivity of Kill/Gen transformers

- Have to show that $f(x \sqcup y) \sqsubseteq f(x) \sqcup f(y)$
- $f(x \sqcup y) = ((x \sqcup y) \setminus k) \cup g$
  $= ((x \setminus k) \sqcup (y \setminus k)) \cup g$
  $= (((x \setminus k) \cup g) \sqcup ((y \setminus k) \cup g))$
  $= f(x) \sqcup f(y)$
- Used distributivity of $\sqcup$ and U
  - Works regardless of whether $\sqcup$ is U or $\cap$

31

Points-to Analysis

- Many flavors
- PWHILE language

p ∈ PExp pointer expressions

a ::= x | n | a1 op_a a2 | &x | *x | nil

S ::= $[x := a]^{lab}$
| $[skip]^{lab}$
| S1;S2
| if $[b]^{lab}$ then S1 else S2
| while $[b]^{lab}$ do S
| x = malloc

32

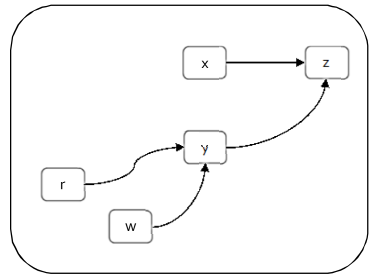
Points-to Analysis

- Aliases
  - Two pointers p and q are aliases if they point to the same memory location
- Points-to pair
  - (p,q) means p holds the address of q
- Points-to pairs and aliases
  - (p,q) and (r,q) means that p and r are aliases
- Challenge: no a priori bound on the set of heap locations

33

Terminology Example

$[x := \&z]^1$
$[y := \&z]^2$
$[w := \&y]^3$
$[r := w]^4$



Points-to pairs: (x,z), (y,z), (w,y), (r,y)
Aliases: (x,y), (r,w)

34

(May) Points-to Analysis

- Property Space
  - $L = (\wp(Var \times Var), \subseteq, \cup, \cap, \emptyset, Var \times Var)$
- Transfer functions

Statement	out(lab)
$[p = \&x]^{lab}$	in(lab) ∪ { (p,x) }
$[p = q]^{lab}$	in(lab) U { (p,x) | (q,x) ∈ in(lab) }
$[*p = q]^{lab}$	in(lab) U { (r,x) | (q,x) ∈ in(lab) and (p,r) ∈ in(lab) }
$[p = *q]^{lab}$	in(lab) U { (p,r) | (q,x) ∈ in(lab) and (x,r) ∈ in(lab) }

35

(May) Points-to Analysis

- What to do with malloc?
- Need some static naming scheme for dynamically allocated objects
- Single name for the entire heap
  - $[\![[p = malloc]^{lab}]\!](S) = S \cup \{ (p,H) \}$
- Name based on static allocation site
  - $[\![[p = malloc]^{lab}]\!](S) = S \cup \{ (p,lab) \}$

36

(May) Points-to Analysis

∅

[x :=malloc]^1;

{ (x,H) }

[y:=malloc]^2;

{ (x,H), (y,H) }

(if [x==y]^3 then

{ (x,H), (y,H) }

[z:=x]^4

{ (x,H), (y,H), (z,H) }

else

[z:=y]^5

{ (x,H), (y,H), (z,H) }

);

{ (x,H), (y,H), (z,H) }

Single name H for the entire heap

37

Allocation Sites

- Divide the heap into a fixed partition based on allocation site
- All objects allocated at the same program point represented by a single "abstract object"

38

(May) Points-to Analysis

∅

[x :=malloc]^1; // A1

{ (x,A1) }

[y:=malloc]^2; // A2

{ (x,A1), (y,A2) }

(if [x==y]^3 then

{ (x,A1), (y,A2) }

[z:=x]^4

{ (x,A1), (y,A2), (z,A1) }

else

[z:=y]^5

{ (x,A1), (y,A2), (z,A2) }

);

{ (x,A1), (y,A2), (z,A1), (z,A2) }

Allocation-site based naming (using A_{lab} instead of just "lab" for clarity)

39

Weak Updates

Statement	out(lab)
[p = &x]^{lab}	in(lab) ∪ { (p,x) }
[p = q]^{lab}	in(lab) **U** { (p,x) | (q,x) ∈ in(lab) }
[*p = q]^{lab}	in(lab) **U** { (r,x) | (q,x) ∈ in(lab) and (p,r) ∈ in(lab) }
[p = *q]^{lab}	in(lab) **U** { (p,r) | (q,x) ∈ in(lab) and (x,r) ∈ in(lab) }

∅

[x :=malloc]^1; // A1

{ (x,A1) }

[y:=malloc]^2; // A2

{ (x,A1), (y,A2) }

[z:=x]^3;

{ (x,A1), (y,A2), (z,A1) }

[z:=y]^4;

{ (x,A1), (y,A2), (z,A1), (z,A2) }

40

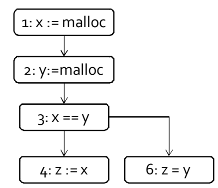
(May) Points-to Analysis

- Fixed partition of the (unbounded) heap to static names
  - Allocation sites
  - Types
  - Calling contexts
  - …
- What we saw so far – flow-insensitive
  - Ignoring the structure of the flow in the program

41

Flow-sensitive vs. Flow-insensitive Analyses

[x :=malloc][1];
[y:=malloc][2];
(if [x==y][3] then
  [z:=x][4]
else
  [z:=y][5]
);



- Flow sensitive: respect program flow
  - a separate set of points-to pairs for every program point
  - the set at a point represents possible may-aliases on some path from entry to the program point
- Flow insensitive: assume all execution orders are possible, abstract away order between statements

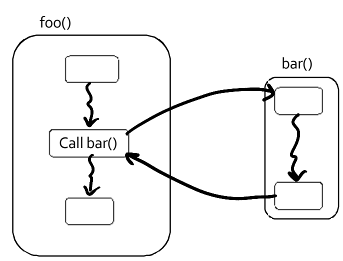
42

So far…

- Intra-procedural analysis
- How are we going to deal with procedures?
- Inter-procedural analysis

43

Interprocedural Analysis



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

44

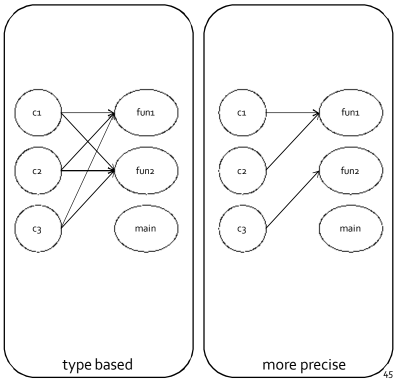
Call Graph

```
int (*pf)(int);

int fun1 (int x) {
  if (x < 10)
    return (*pf) (x+l); // C1
  else
    return x;
}

int fun2(int y) {
  pf = &fun1;
  return (*pf) (y) ; // C2
}

void main() {
  pf = &fun2;
  (*pf )(5) ; // C3
}
```

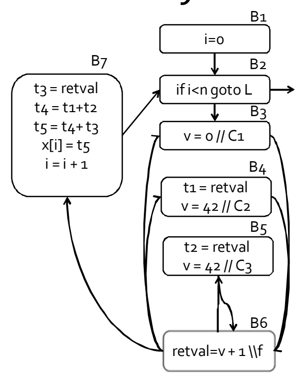


45

Context Sensitivity

```
main() {
 for (i = 0; i < n; i++) {
  t1 = f(0);   // C1
  t2 = f(42); // C2
  t3 = f(42); // C3
  X[i] = t1 + t2 + t3;
 }
}

int f(int v)
  return (v+1);
}
```



46

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

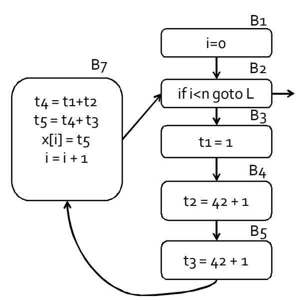
```
main() { f(); f(); }
f() { g(); g(); }
g() { h(); h(); }
h() { ... }
```

47

Inlining

```
main() {
 for (i = 0; i < n; i++) {
  t1 = f(0);   // C1
  t2 = f(42); // C2
  t3 = f(42); // C3
  X[i] = t1 + t2 + t3;
 }
}

int f(int v)
  return (v+1);
}
```



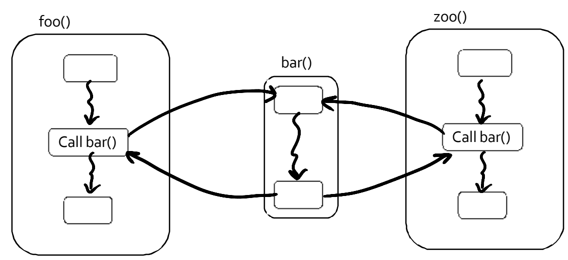
48

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

49

Unrealizable Paths



50

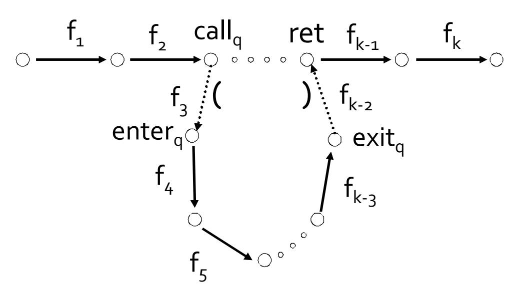
Interprocedural Analysis

begin
proc p() is^{1}
  [x := a + 1]^{2}
end^{3}
[a=7]^{4}
[call p()]$^{5}_{6}$
[print x]^{7}
[a=9]^{8}
[call p()]$^{9}_{10}$
[print a]^{11}
end

- Extend language with begin/end and with [call p()]$^{clab}_{rlab}$
- Call label clab, and return label rlab

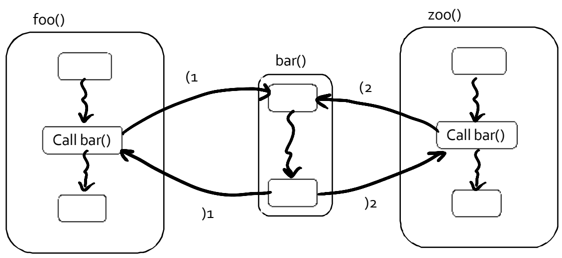
51

IVP: Interprocedural Valid Paths



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

Valid Paths



53

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 $)_i$ | ε
  - valid ::= valid $(_i$ matched | matched
    - *paths* can be defined by a regular expression

The Join-Over-Valid-Paths (JVP)

- vpaths(n) all valid paths from program start to n
- $JVP[n] = \sqcup\{[\![e_1, e_2, \ldots, e]\!] \text{ (initial)}$
  $(e_1, e_2, \ldots, e) \in vpaths(n)\}$
- JVP $\sqsubseteq$ JFP
  - In some cases the JVP can be computed
  - (Distributive problem)

Sharir and Pnueli '82

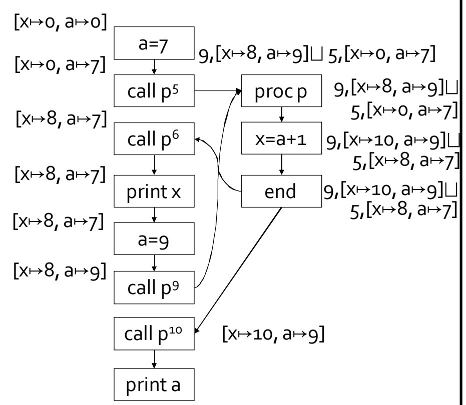
- Call String approach
  - Blend interprocedural flow with intra procedural flow
  - Tag every dataflow fact with call history
- Functional approach
  - Determine the effect of a procedure
    - E.g., in/out map
  - Treat procedure invocations as "super ops"

The Call String Approach

- Record at every node a pair (l, c) where l ∈ 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

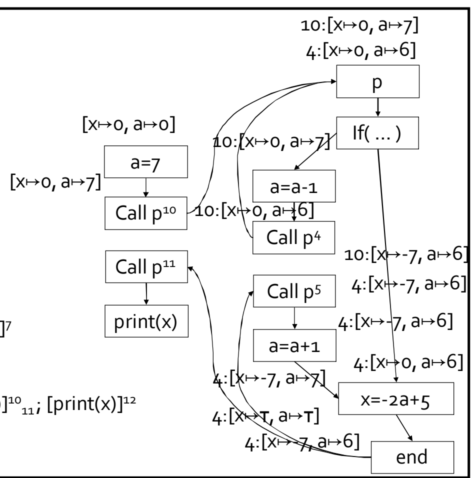
---

begin
proc p() is^1
  [x := a + 1]^2
end^3
[a=7]^4
[call p()]$^5_6$
[print x]^7
[a=9]^8
[call p()]$^9_{10}$
[print a]^{11}
end



---

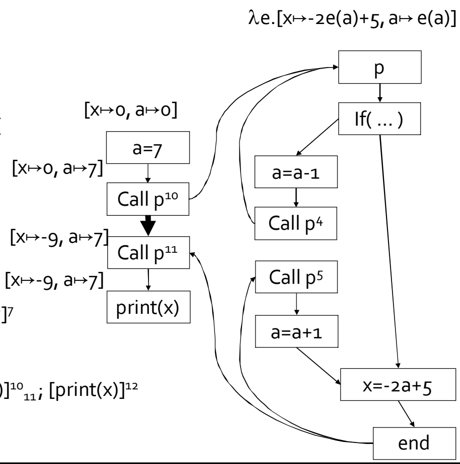
begin^0
proc p() is^1
  if [b]^2 then (
   [a := a -1]^3
   [call p()]$^4_5$
   [a := a + 1]^6
)
  [x := -2* a + 5]^7
end^8
[a=7]^9 ; [call p()]$^{10}_{11}$; [print(x)]^{12}
end^{13}



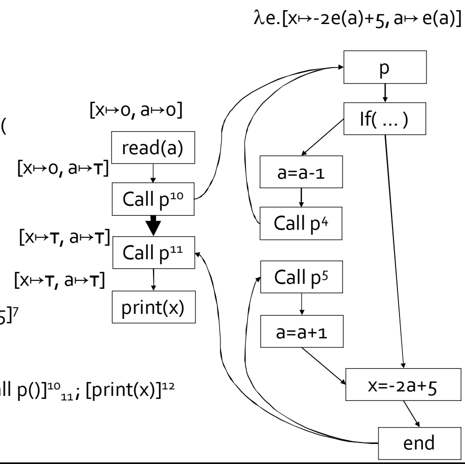
The Functional Approach

- 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

begin
proc p() is^1
if [b]^2 then (
[a := a -1]^3
[call p()]$^4_5$
[a := a + 1]^6
)
[x := -2* a + 5]^7
end^8
[a=7]^9 ; [call p()]$^{10}_{11}$; [print(x)]^{12}
end



begin
proc p() is^1
if [b]^2 then (
[a := a -1]^3
[call p()]$^4_5$
[a := a + 1]^6
)
[x := -2* a + 5]^7
end^8
[read(a)]^9 ; [call p()]$^{10}_{11}$; [print(x)]^{12}
end



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