Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 12: Interprocedural Analysis + Numerical Analysis

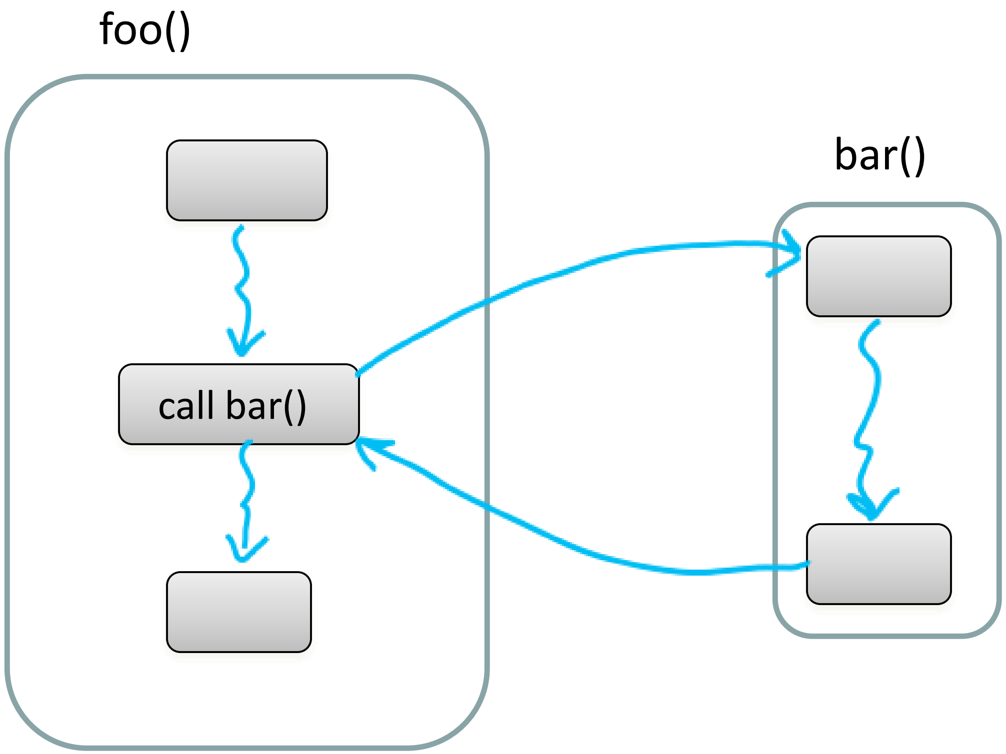
Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

Procedural program

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

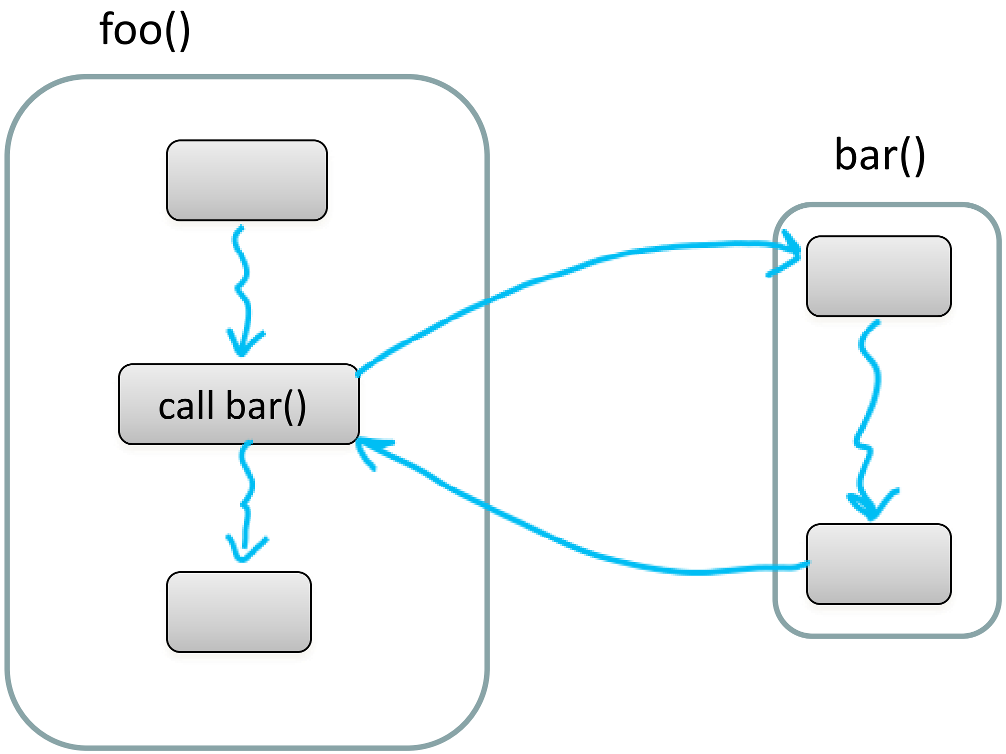
```
int p(int a) {
  return a + 1;
}
```

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

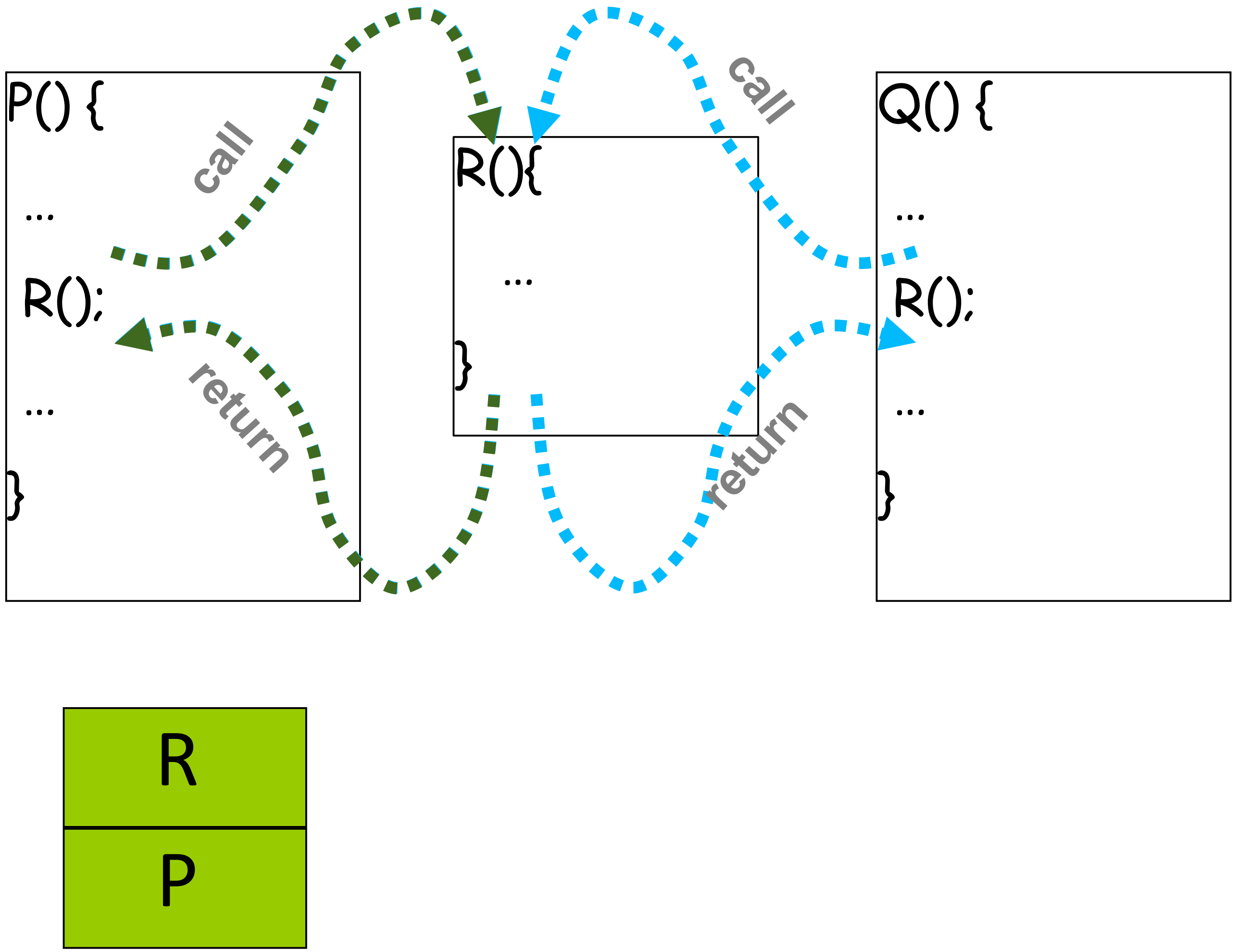
Naïve solutions

- Inilining
- Call/Return as goto

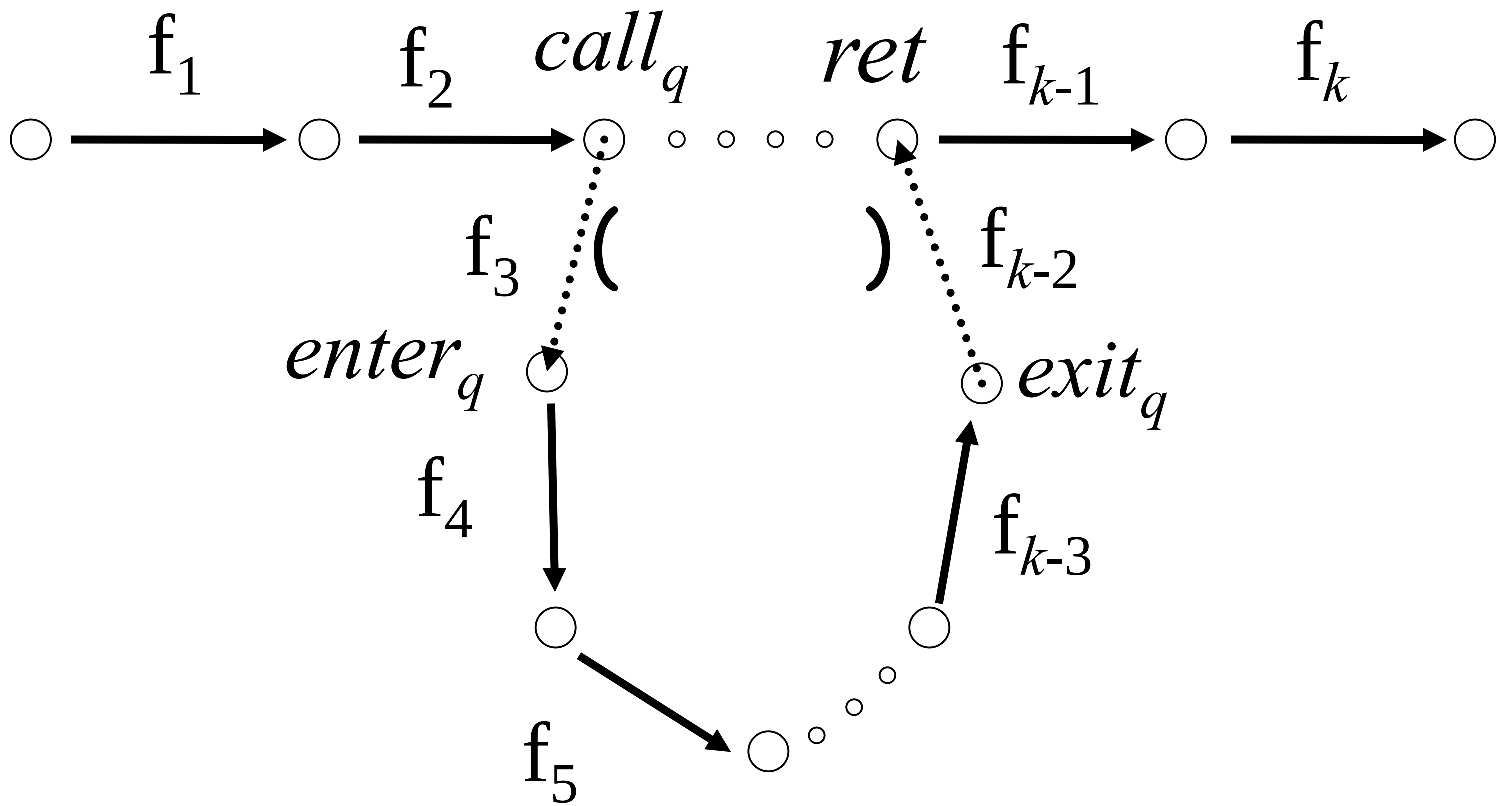
Guiding light

- Exploit stack regime
  - ➔Precision
  - ➔Efficiency

Stack regime

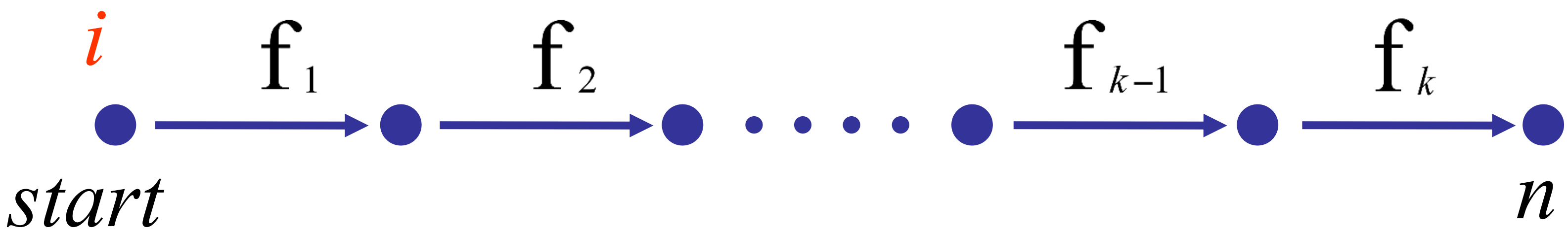


Interprocedural Valid Paths



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

Join Over All Paths (JOP)



$$[\![f_k \circ \ldots \circ f_1]\!] \in L \rightarrow L$$

- $JOP[v] = \sqcup\{[[e_1, e_2, \ldots, e_n]](\iota) \mid (e_1, \ldots, e_n) \in paths(v)\}$
- $JOP \sqsubseteq LFP$
  - Sometimes JOP = LFP
    - precise up to "**symbolic execution**"
    - Distributive problem

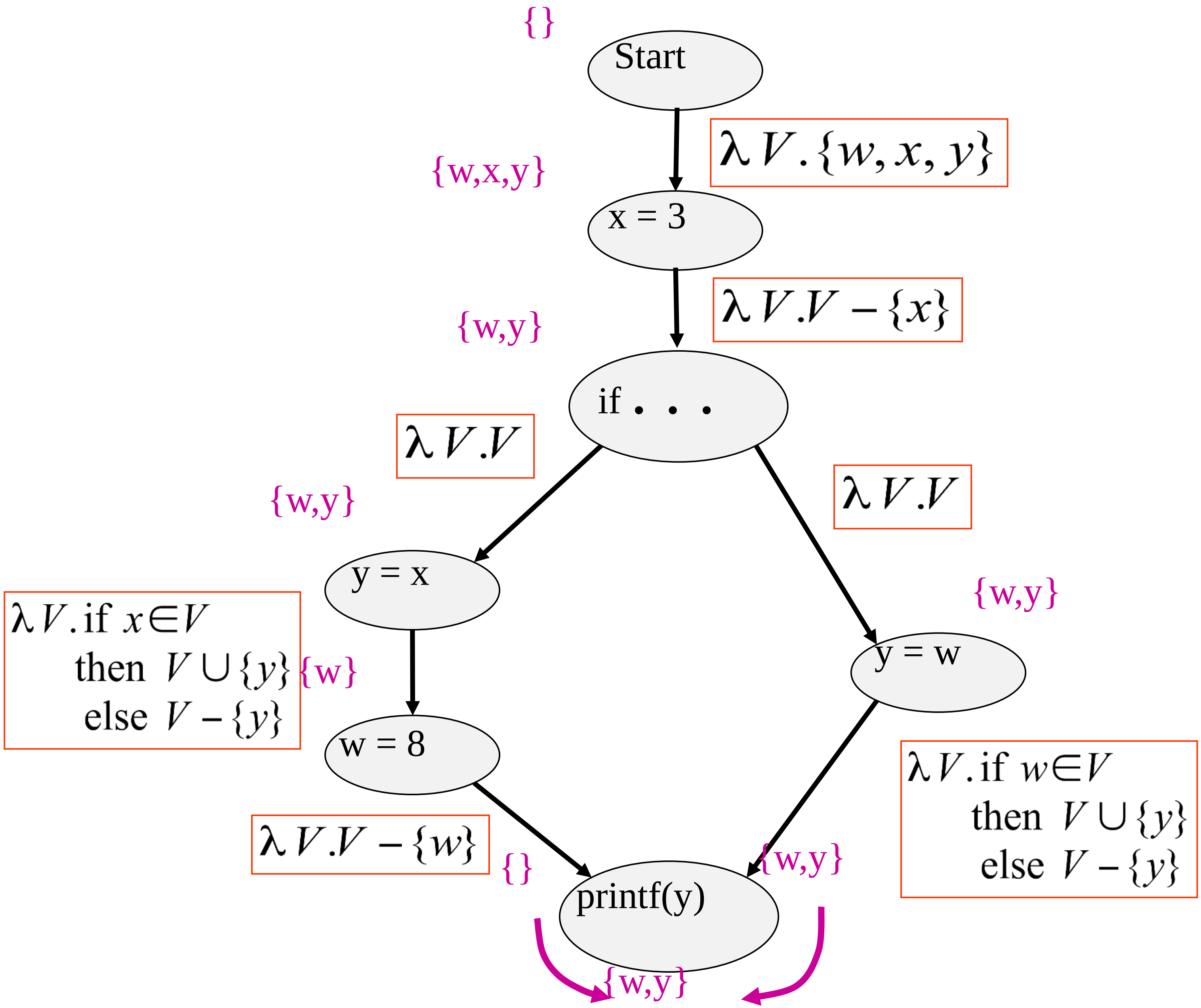
CFL-Graph reachability

- Special cases of functional analysis
- Finite distributive lattices
- Provides more efficient analysis algorithms
- Reduce the interprocedural analysis problem to finding context free reachability

IDFS / IDE

- **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. *Reps, Horowitz, and Sagiv, FASE' 95, TCS' 96*
  - ***More general solutions exist***

Possibly Uninitialized Variables



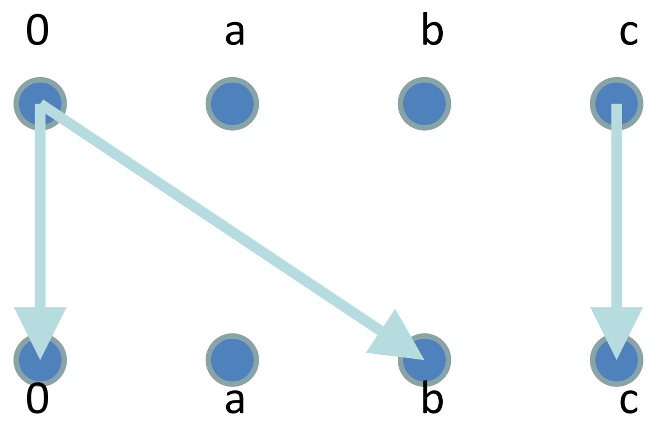
IFDS Problems

- Finite subset distributive
  - Lattice L = $\wp$(D)
  - $\sqsubseteq$ is $\subseteq$
  - $\sqcup$ is $\cup$
  - Transfer functions are distributive

- Efficient solution through formulation as CFL reachability

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 Λ)
- Example: D = { a, b, c }
  f(S) = (S – {a}) U {b}



207

Efficiently Representing Functions

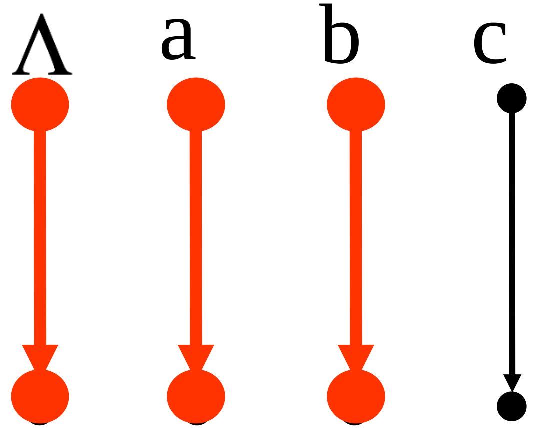
- Let $f: 2^D \rightarrow 2^D$ be a distributive function
- Then:
  - $f(X) = f(\emptyset) \cup (\bigcup \{ f(\{z\}) \mid z \in X \})$
  - $f(X) = f(\emptyset) \cup (\bigcup \{ f(\{z\}) \setminus f(\emptyset) \mid z \in X \})$

Representing Dataflow Functions

Identity Function

$$f = \lambda V.V$$

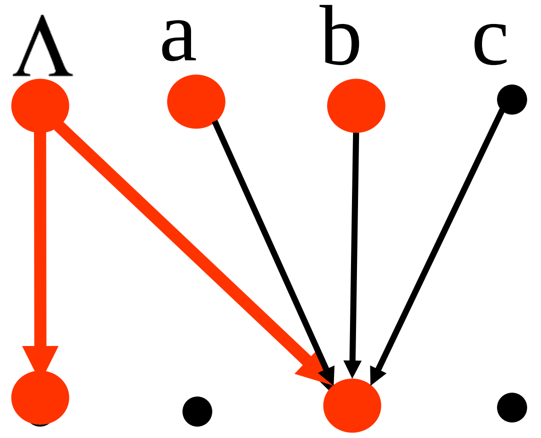
$$f(\{a,b\}) = \{a,b\}$$



Constant Function

$$f = \lambda V.\{b\}$$

$$f(\{a,b\}) = \{b\}$$

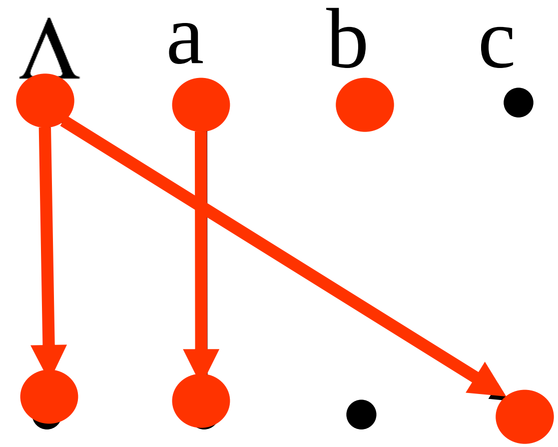


Representing Dataflow Functions

"Gen/Kill" Function

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

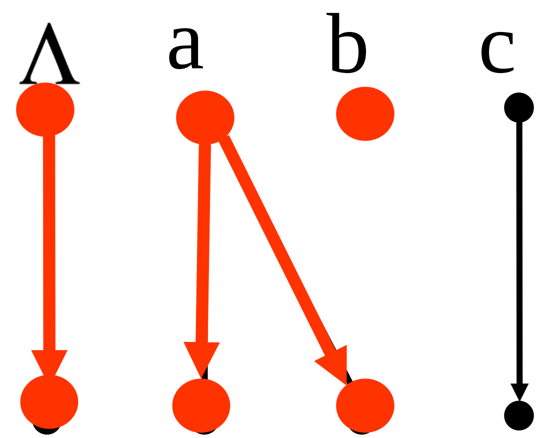
$$f(\{a,b\}) = \{a,c\}$$

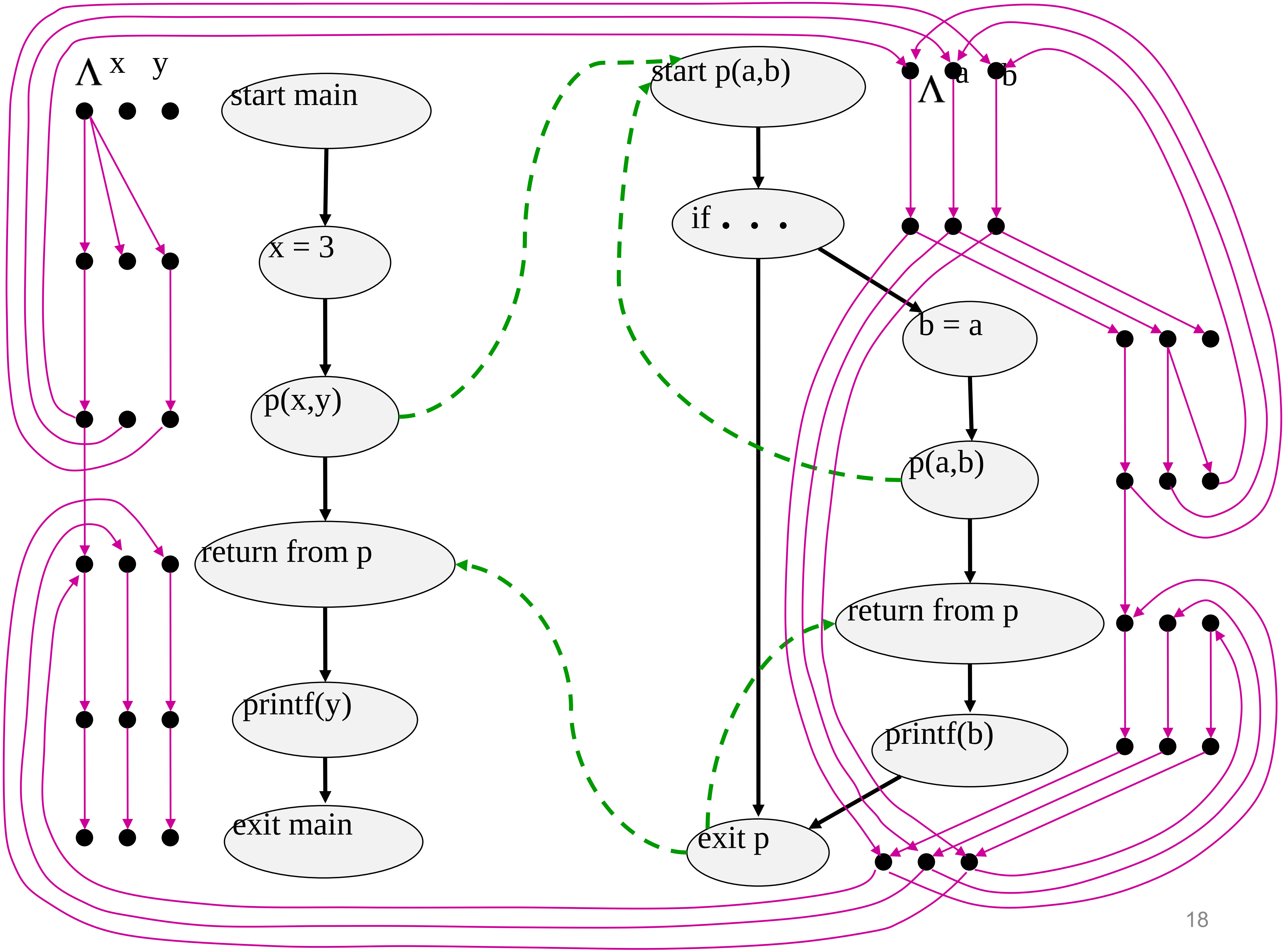


Non-"Gen/Kill" Function

$$f = \lambda V.\text{if } a \in V \text{ then } V \cup \{b\} \text{ else } V - \{b\}$$

$$f(\{a,b\}) = \{a,b\}$$

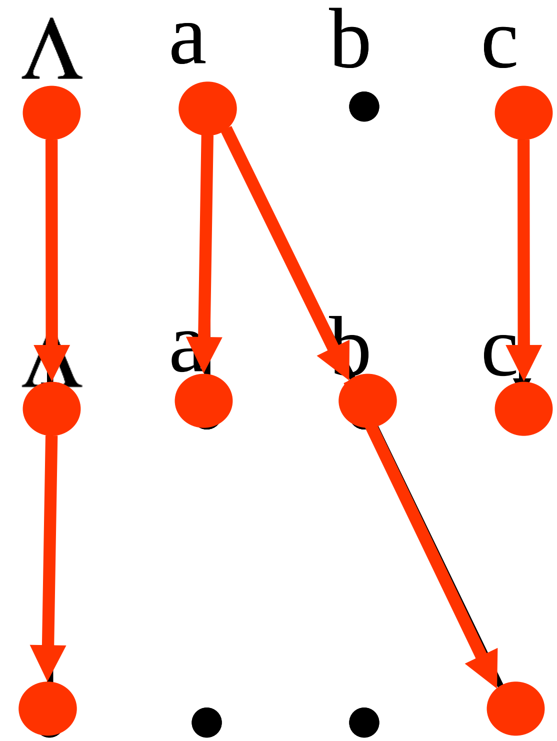



Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
start p(a,b)
Λ a b
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p

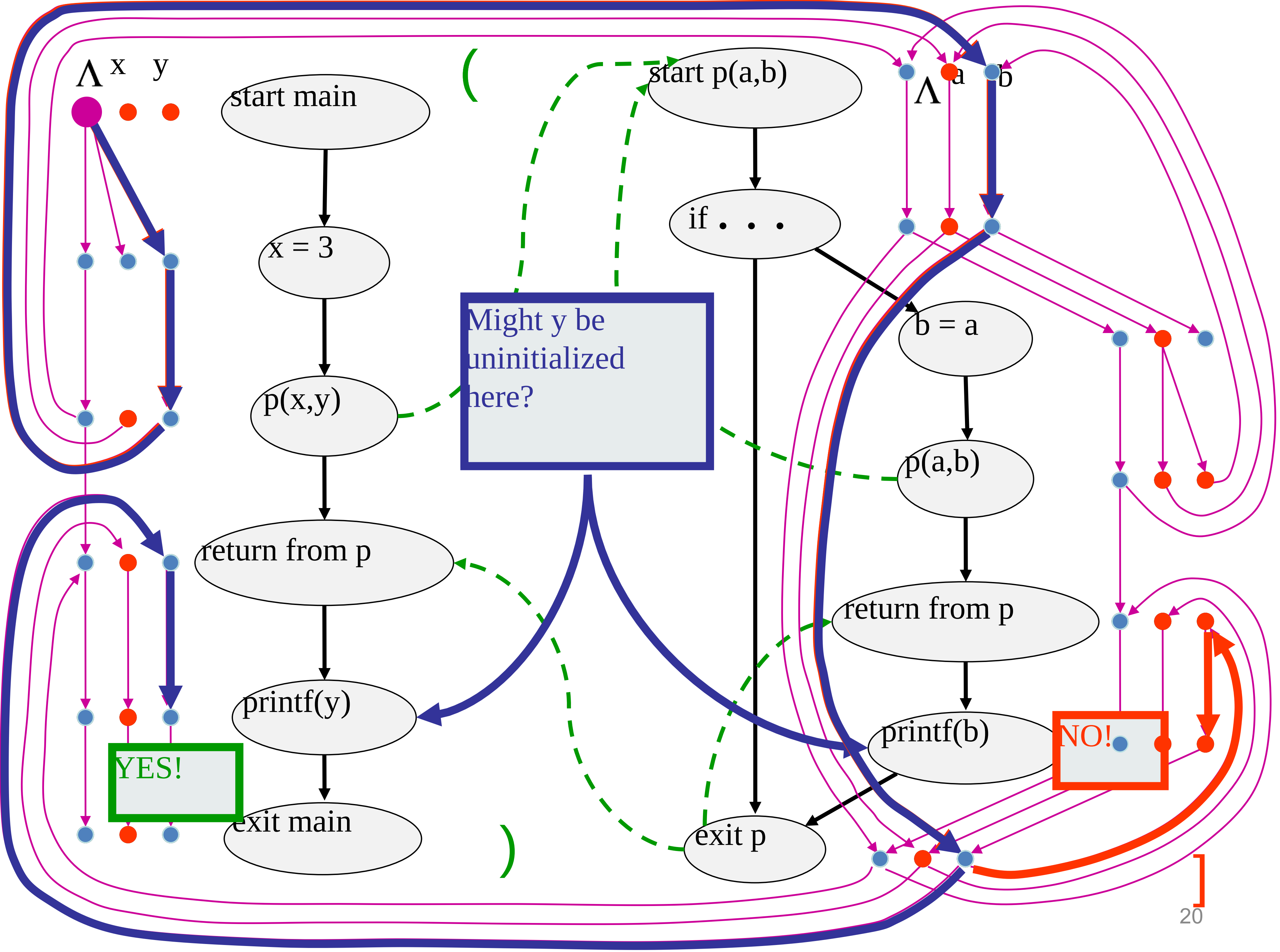
Composing Dataflow Functions

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

$$f_2 = \lambda V.\text{if } b \in V \text{ then } \{c\} \text{ else } \phi$$



$$f_2 \circ f_1(\{a, c\}) = \boxed{\{c\}}$$



Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
YES!
Might y be uninitialized here?
start p(a,b)
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p
Λ a b
NO!

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

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 $\langle start_{main}, \Lambda \rangle$ to $\langle n, d \rangle$

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

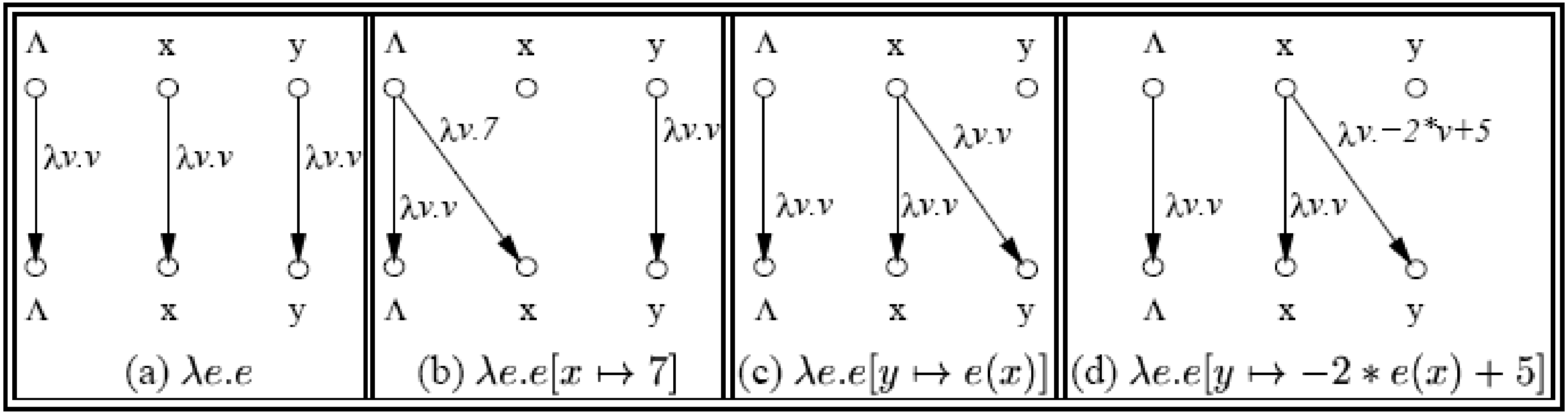
- Consider the constant propagation lattice
- The value of every variable y at the program exit can be represented by:

  $y = \sqcup \{(a_x x + b_x) \mid x \in Var_* \} \sqcup c$

  $a_x, c \in Z \cup \{\bot, \top\} \qquad b_x \in Z$

- Supports efficient composition and "functional" join
  - [z := a * y + b]
  - What about [z:=x+y]?

Linear constant propagation



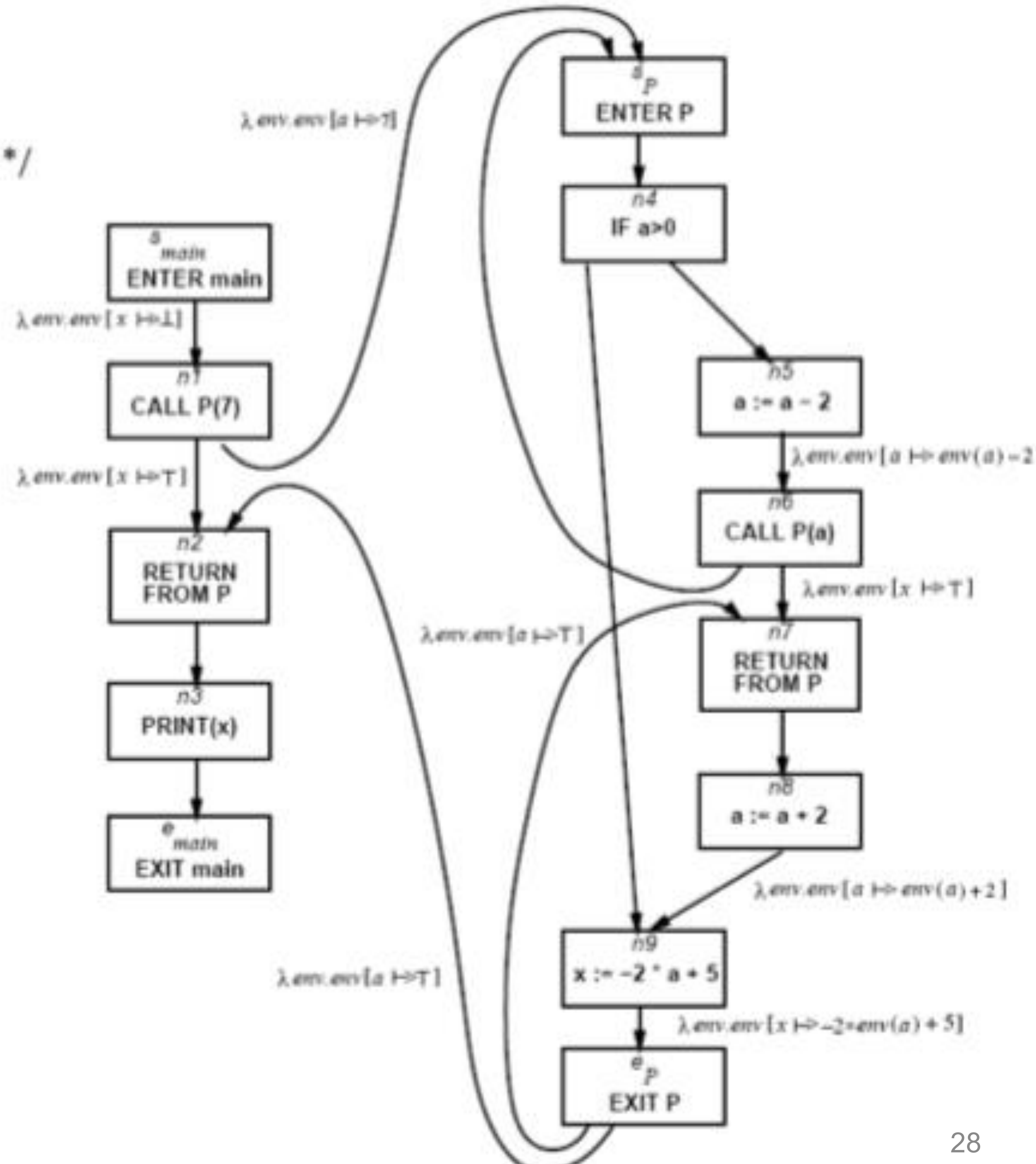
Point-wise representation of environment transformers

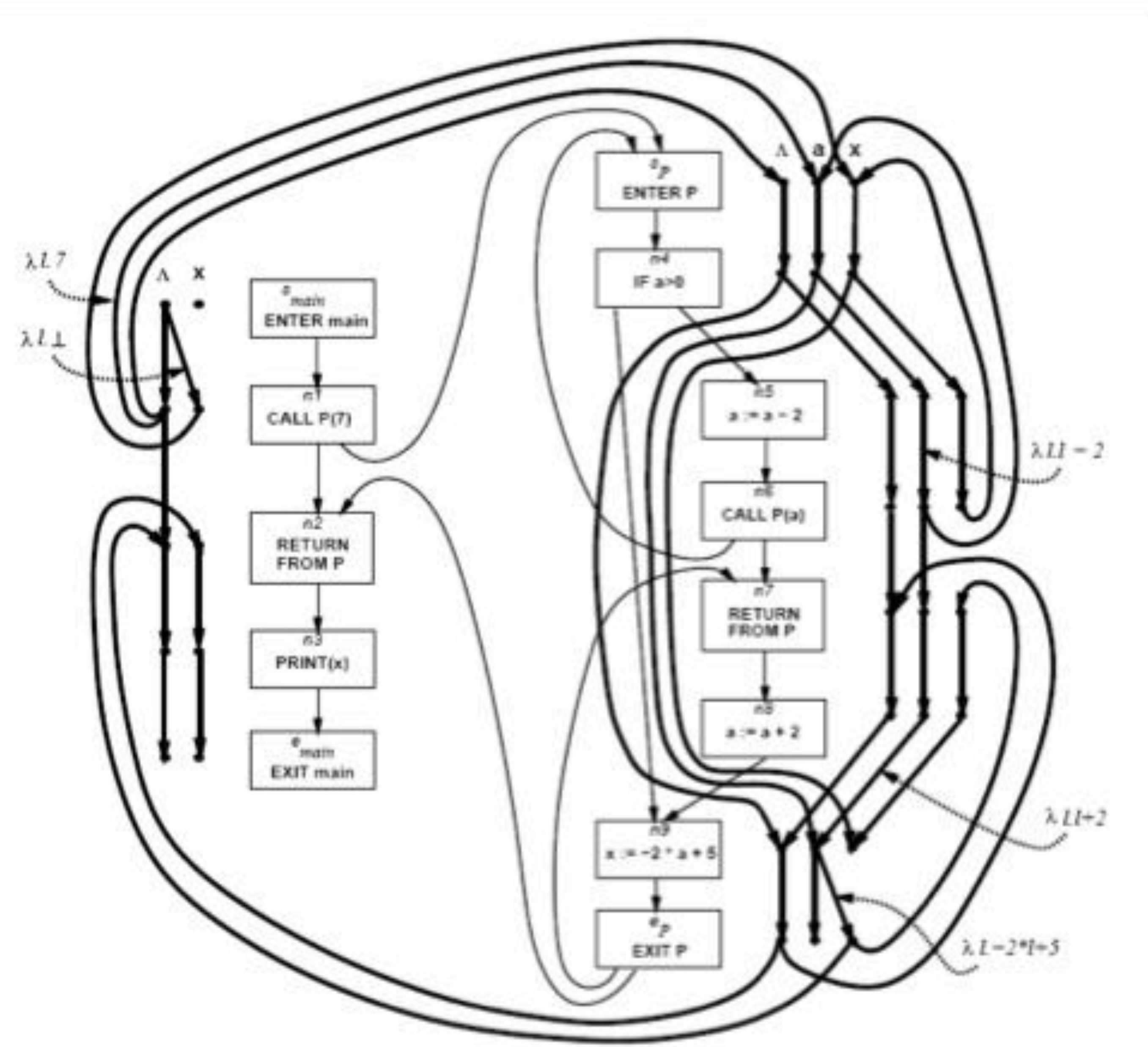
IDE Analysis

- Point-wise representation closed under composition
- CFL-Reachability on the exploded graph
- Compose functions

```
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
```




λl.7
λl.⊥
Λ
x
s_{main}
ENTER main
n1
CALL P(7)
n2
RETURN FROM P
n3
PRINT(x)
e_{main}
EXIT main
s_P
ENTER P
n4
IF a>0
n5
a := a - 2
n6
CALL P(a)
n7
RETURN FROM P
n8
a := a + 2
n9
x := -2 * a + 5
e_P
EXIT P
Λ
a
x
λl.l - 2
λl.l+2
λl.-2*l+5

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

Challenges in Interprocedural Analysis

- Respect call-return mechanism
- Handling recursion
- Local variables
- Parameter passing mechanisms
- The called procedure is not always known
- The source code of the called procedure is not always available

A trivial treatment of procedure

- Analyze a single procedure
- After every call continue with conservative information
  - Global variables and local variables which “may be modified by the call” have unknown values
- Can be easily implemented
- Procedures can be written in different languages
- Procedure inline can help

Disadvantages of the trivial solution

- Modular (object oriented and functional) programming encourages small frequently called procedures
- Almost all information is lost

Bibliography

- **Textbook 2.5**
- Patrick Cousot & Radhia Cousot. Static determination of dynamic properties of recursive procedures In *IFIP Conference on Formal Description of Programming Concepts*, E.J. Neuhold, (Ed.), pages 237-277, St-Andrews, N.B., Canada, 1977. North-Holland Publishing Company (1978).
- **Two Approaches to interprocedural analysis by Micha Sharir and Amir Pnueli**
- 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*

A Semantics for Procedure Local Heaps and its Abstractions

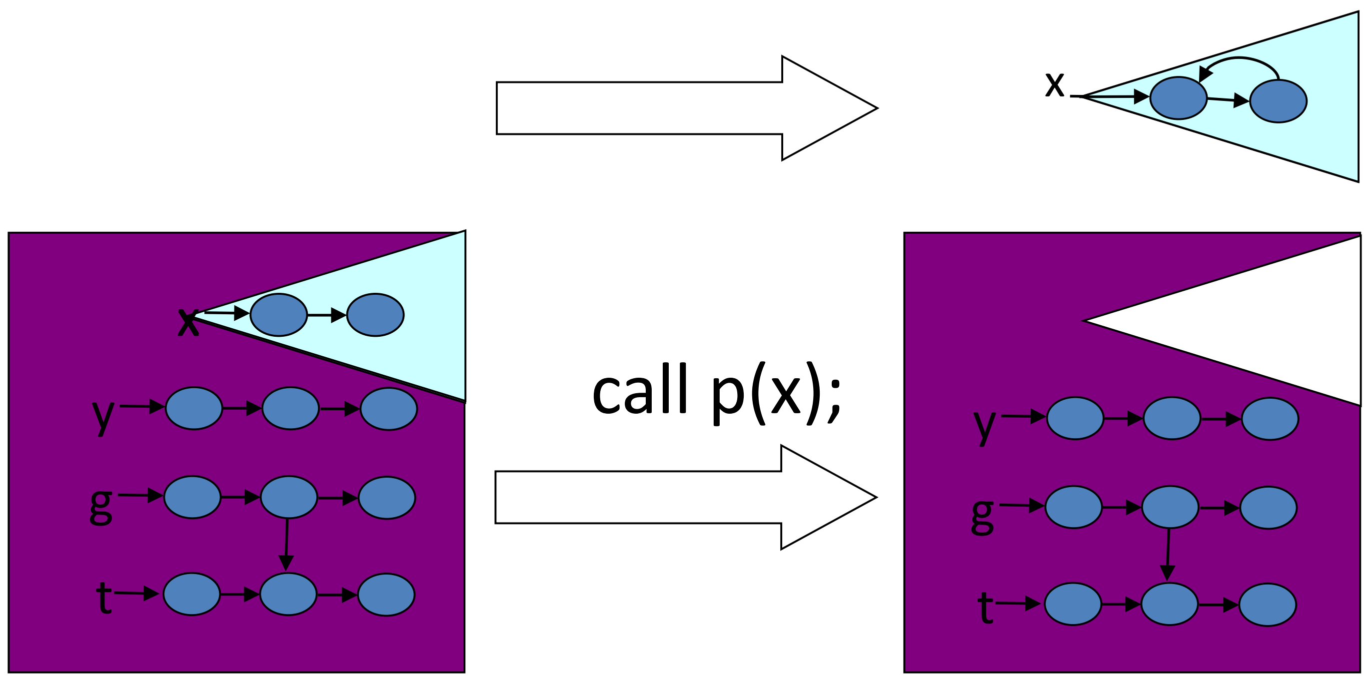
Noam Rinetzky Tel Aviv University
Jörg Bauer Universität des Saarlandes
Thomas Reps University of Wisconsin
Mooly Sagiv Tel Aviv University
Reinhard Wilhelm Universität des Saarlandes

Motivation

- Interprocedural shape analysis
  - Conservative static pointer analysis
  - Heap intensive programs
    - Imperative programs with procedures
    - Recursive data structures
- Challenge
  - Destructive update
  - Localized effect of procedures

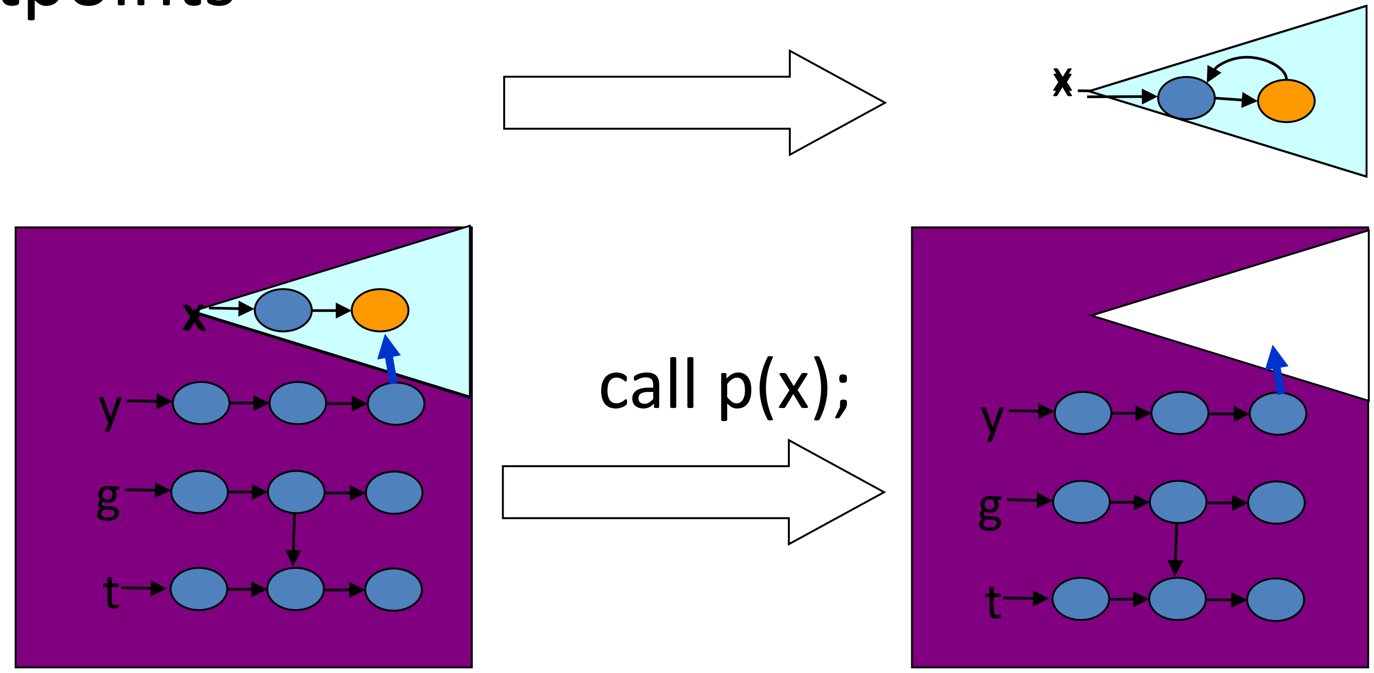
Main idea

- Local heaps



Main idea

- Local heaps
- Cutpoints

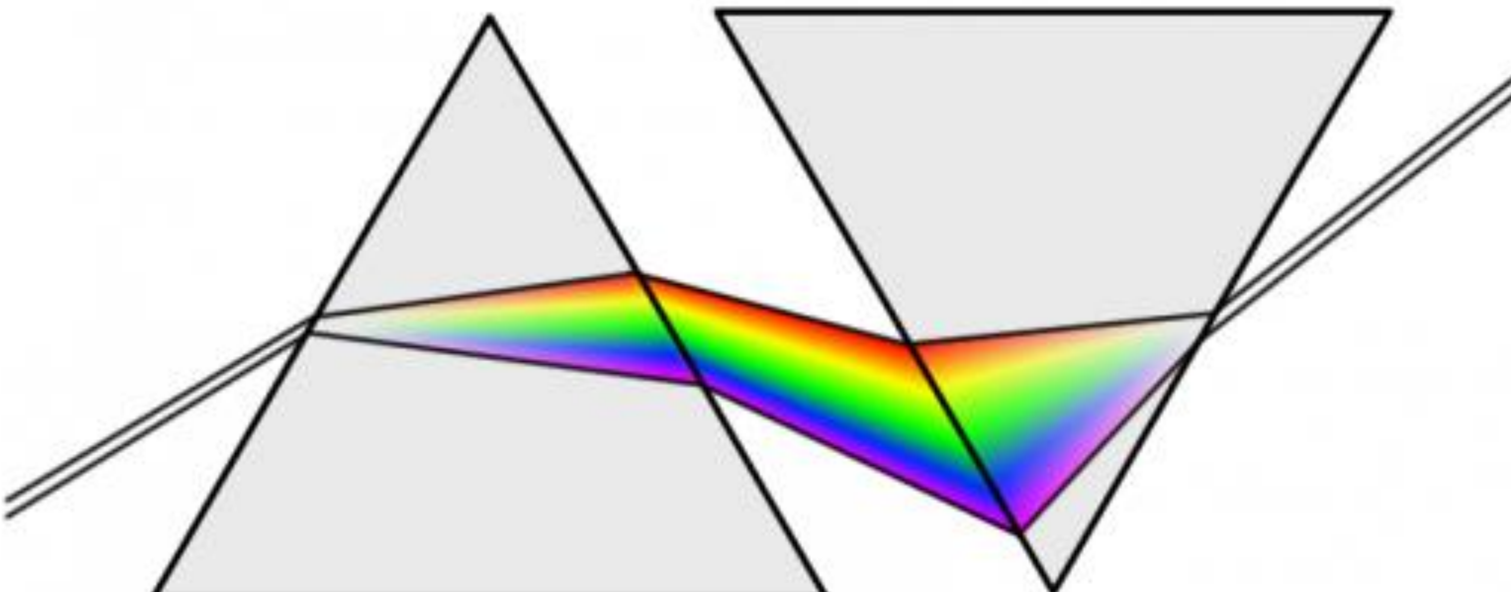


Numerical Analysis

Abstract Interpretation [Cousot'77]

- Mathematical foundation of static analysis

Widening/Narrowing



How can we prove this automatically?

```
public void loopExample() {
    int x = 7;
    while (x < 1000) {
        ++x;
    }
    if (!(x == 1000))
        error("Unable to prove x == 1000!");
}
```

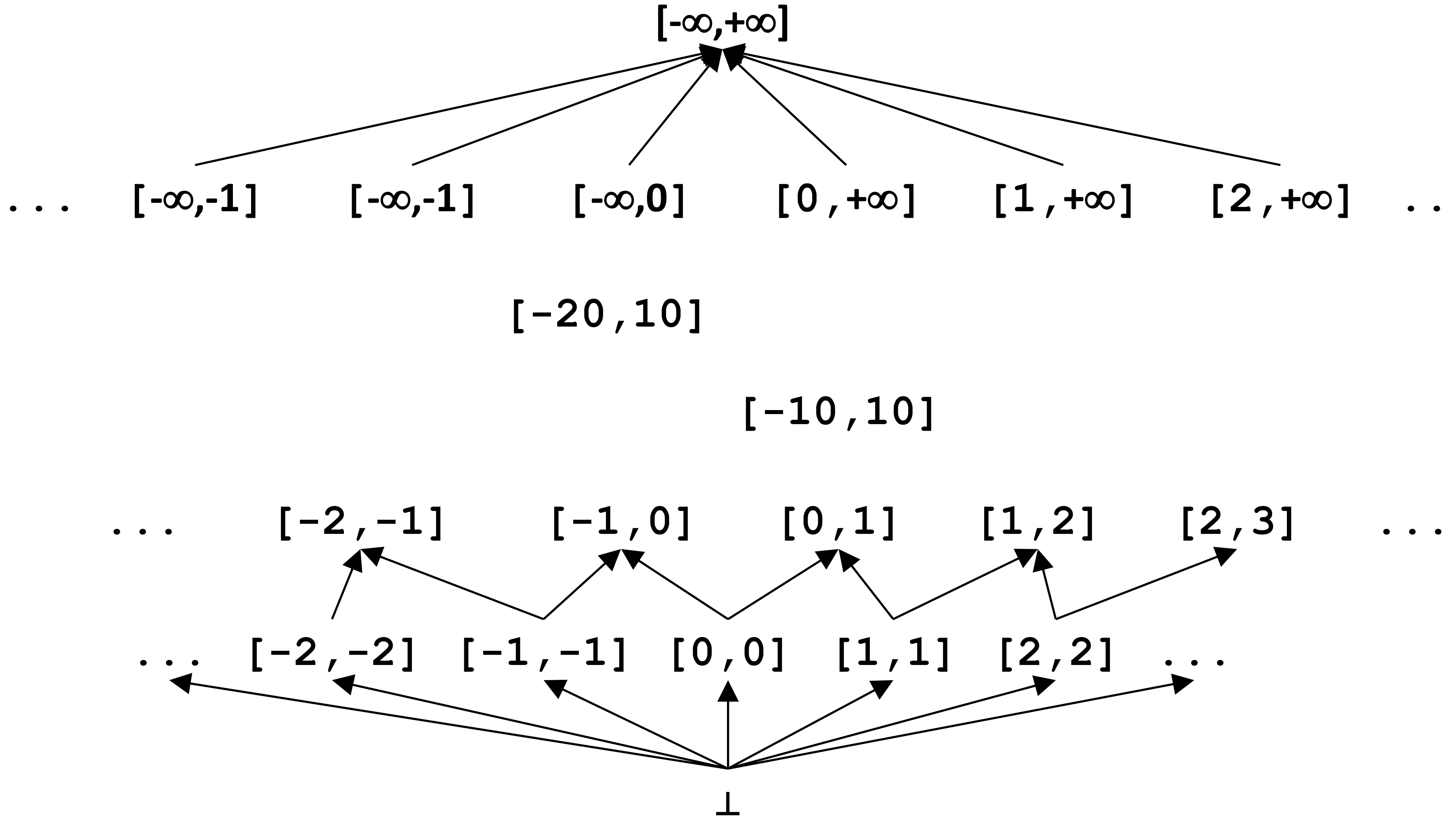
RelProd(CP, VE)

```
Reached fixed-point after 19 iterations.
Solution = {
  V[0] : (true, true)
  V[1] : (true, true)
  V[2] : (x=7, true)
  V[3] : (x=7, true)
  V[4] : (true, true)
  V[7] : (true, true)
  V[5] : (true, true)
  V[6] : (true, true)
  V[8] : (true, true)
  V[9] : (true, true)
  V[10] : (true, true)
  V[12] : (true, true)
  V[11] : (true, true)
}
1 possible errors found.
```

Intervals domain

- One of the simplest numerical domains
- Maintain for each variable *x* an interval [*L*,*H*]
  - *L* is either an integer of -∞
  - *H* is either an integer of +∞
- A (non-relational) numeric domain

Intervals lattice for variable *x*



Intervals lattice for variable *x*

- $D^{int}[x]$ = { (L,H) | L∈-∞,**Z** and H∈**Z**,+∞ and $L \leq H$}
- ⊥
- ⊤=[-∞,+∞]
- ⊑ = ?
  - [1,2] ⊑ [3,4] ?
  - [1,4] ⊑ [1,3] ?
  - [1,3] ⊑ [1,4] ?
  - [1,3] ⊑ [-∞,+∞] ?
- What is the lattice height?

Intervals lattice for variable *x*

- $D^{int}[x] = \{ (L,H) \mid L \in -\infty, \mathbf{Z}$ and $H \in \mathbf{Z}, +\infty$ and $L \leq H \}$
- $\bot$
- $\top = [-\infty, +\infty]$
- $\sqsubseteq$ = ?
  - $[1,2] \sqsubseteq [3,4]$ no
  - $[1,4] \sqsubseteq [1,3]$ no
  - $[1,3] \sqsubseteq [1,4]$ yes
  - $[1,3] \sqsubseteq [-\infty, +\infty]$ yes
- What is the lattice height? Infinite

Joining/meeting intervals

- $[a,b] \sqcup [c,d]$ = ?
  - $[1,1] \sqcup [2,2]$ = ?
  - $[1,1] \sqcup [2, +\infty]$ = ?
- $[a,b] \sqcap [c,d]$ = ?
  - $[1,2] \sqcap [3,4]$ = ?
  - $[1,4] \sqcap [3,4]$ = ?
  - $[1,1] \sqcap [1,+\infty]$ = ?
- Check that indeed $x \sqsubseteq y$ if and only if $x \sqcup y = y$

Joining/meeting intervals

- $[a,b] \sqcup [c,d] = [\min(a,c), \max(b,d)]$
  - $[1,1] \sqcup [2,2] = [1,2]$
  - $[1,1] \sqcup [2,+\infty] = [1,+\infty]$
- $[a,b] \sqcap [c,d] = [\max(a,c), \min(b,d)]$ if a proper interval and otherwise $\perp$
  - $[1,2] \sqcap [3,4] = \perp$
  - $[1,4] \sqcap [3,4] = [3,4]$
  - $[1,1] \sqcap [1,+\infty] = [1,1]$
- Check that indeed $x \sqsubseteq y$ if and only if $x \sqcup y = y$

Interval domain for programs

- $D^{int}[x] = \{ (L,H) \mid L \in -\infty,\mathbf{Z}$ and $H \in \mathbf{Z},+\infty$ and $L \leq H\}$
- For a program with variables $Var=\{x_1,\ldots,x_k\}$
- $D^{int}[Var] = ?$

Interval domain for programs

- $D^{int}[x] = \{ (L,H) \mid L \in -\infty,\mathbf{Z}$ and $H \in \mathbf{Z},+\infty$ and $L \leq H\}$
- For a program with variables $Var=\{x_1,\ldots,x_k\}$
- $D^{int}[Var] = D^{int}[x_1] \times \ldots \times D^{int}[x_k]$
- How can we represent it in terms of formulas?

Interval domain for programs

- $D^{int}[x]$ = { (L,H) | L∈-∞,**Z** and H∈**Z**,+∞ and $L \leq H$}
- For a program with variables *Var*={$x_1,\ldots,x_k$}
- $D^{int}[Var] = D^{int}[x_1] \times \ldots \times D^{int}[x_k]$
- How can we represent it in terms of formulas?
  - Two types of factoids $x \geq c$ and $x \leq c$
  - Example: $S = \wedge\{x \geq 9, y \geq 5, y \leq 10\}$
  - Helper operations
    - c + +∞ = +∞
    - remove(S, x) = S without any x-constraints
    - lb(S, x) =

Assignment transformers

- $[\![x := c]\!]^\# S = ?$
- $[\![x := y]\!]^\# S = ?$
- $[\![x := y+c]\!]^\# S = ?$
- $[\![x := y+z]\!]^\# S = ?$
- $[\![x := y*c]\!]^\# S = ?$
- $[\![x := y*z]\!]^\# S = ?$

Assignment transformers

- $[\![x := c]\!]^\# S$ = remove(S,x) $\cup$ {$x \geq c$, $x \leq c$}
- $[\![x := y]\!]^\# S$ = remove(S,x) $\cup$ {$x \geq$lb(S,y), $x \leq$ub(S,y)}
- $[\![x := y+c]\!]^\# S$ = remove(S,x) $\cup$ {$x \geq$lb(S,y)+c, $x \leq$ub(S,y)+c}
- $[\![x := y+z]\!]^\# S$ = remove(S,x) $\cup$ {$x \geq$lb(S,y)+lb(S,z),
  $x \leq$ub(S,y)+ub(S,z)}
- $[\![x := y*c]\!]^\# S$ = remove(S,x) $\cup$ if c>0 {$x \geq$lb(S,y)*c, $x \leq$ub(S,y)*c}
  else {$x \geq$ub(S,y)*-c, $x \leq$lb(S,y)*-c}
- $[\![x := y*z]\!]^\# S$ = remove(S,x) $\cup$?

`assume` transformers

- $[\![$**`assume`** $x=c]\!]^\# S$ = ?
- $[\![$**`assume`** $x<c]\!]^\# S$ = ?
- $[\![$**`assume`** $x=y]\!]^\# S$ = ?
- $[\![$**`assume`** $x\neq c]\!]^\# S$ = ?

assume transformers

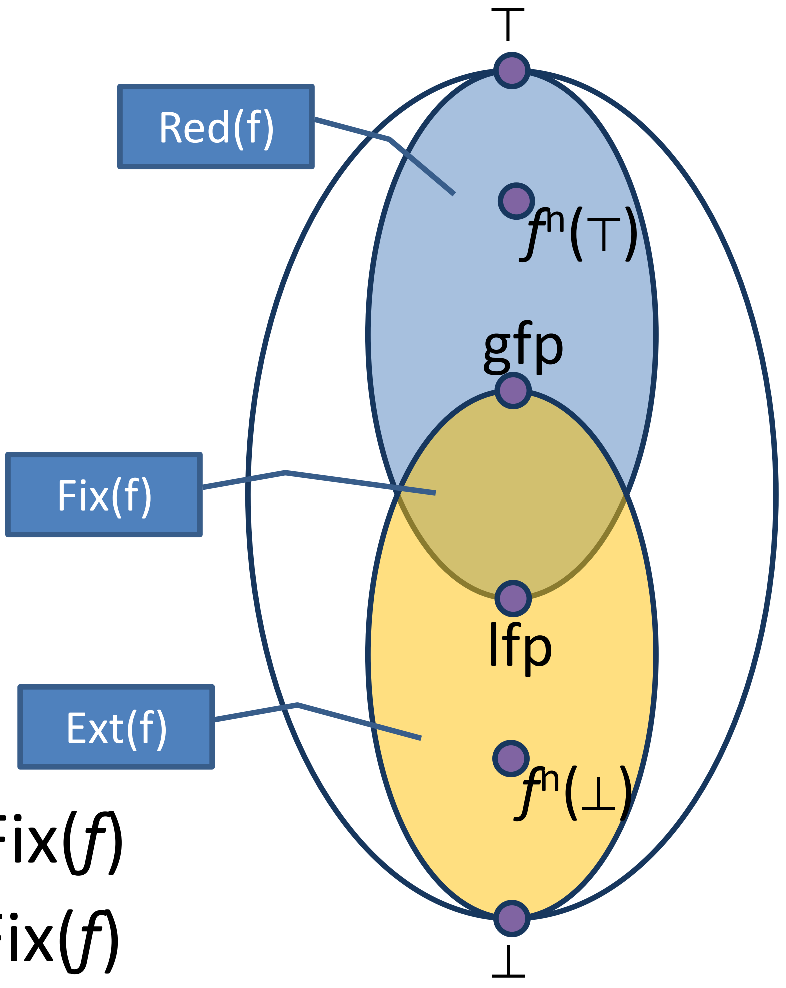
- $[\![$**assume** x=$c]\!]^\#\ S = S \sqcap \{x \geq c,\ x \leq c\}$
- $[\![$**assume** x<$c]\!]^\#\ S = S \sqcap \{x \leq c\text{-}1\}$
- $[\![$**assume** x=$y]\!]^\#\ S = S \sqcap \{x \geq \mathrm{lb}(S,y),\ x \leq \mathrm{ub}(S,y)\}$
- $[\![$**assume** x≠$c]\!]^\#\ S$ = ?

assume transformers

- $[\![$**assume** $x=c]\!]^\# S = S \sqcap \{x \geq c, x \leq c\}$
- $[\![$**assume** $x<c]\!]^\# S = S \sqcap \{x \leq c\text{-}1\}$
- $[\![$**assume** $x=y]\!]^\# S = S \sqcap \{x \geq \mathrm{lb}(S,y), x \leq \mathrm{ub}(S,y)\}$
- $[\![$**assume** $x \neq c]\!]^\# S = (S \sqcap \{x \leq c\text{-}1\}) \sqcup (S \sqcap \{x \geq c+1\})$

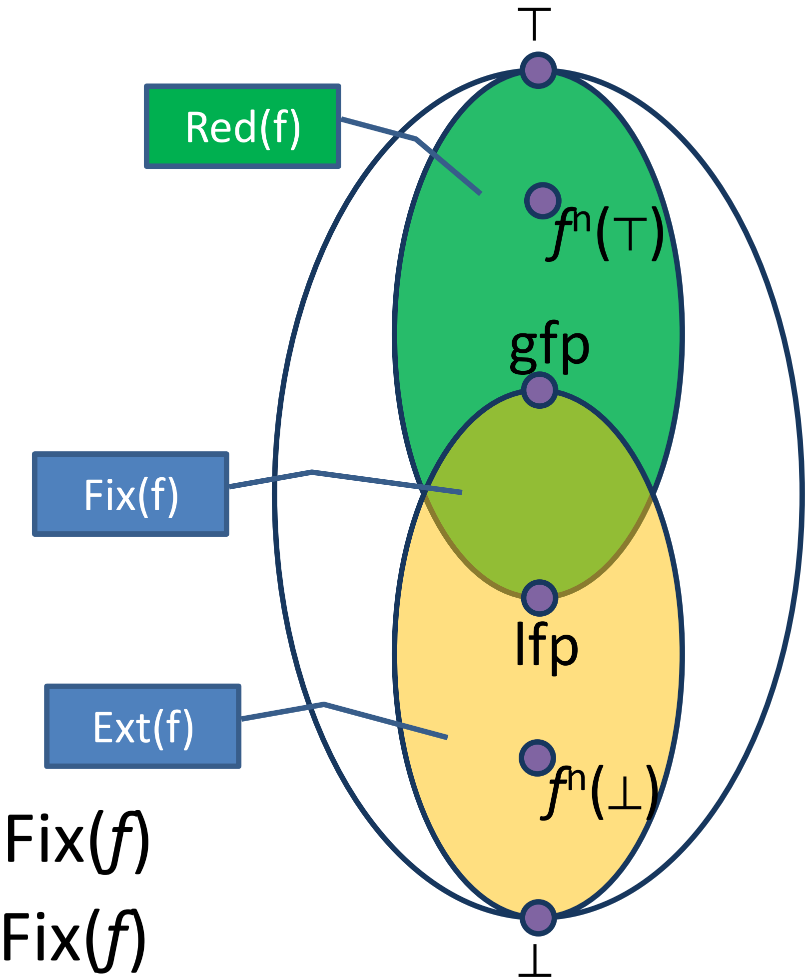
Effect of function *f* on lattice elements

- $L = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
- $f : D \rightarrow D$ **monotone**
- $\text{Fix}(f) = \{ d \mid f(d) = d \}$
- $\text{Red}(f) = \{ d \mid f(d) \sqsubseteq d \}$
- $\text{Ext}(f) = \{ d \mid d \sqsubseteq f(d) \}$
- **Theorem** [Tarski 1955]
  - $\text{lfp}(f) = \sqcap\text{Fix}(f) = \sqcap\text{Red}(f) \in \text{Fix}(f)$
  - $\text{gfp}(f) = \sqcup\text{Fix}(f) = \sqcup\text{Ext}(f) \in \text{Fix}(f)$



Effect of function *f* on lattice elements

- $L = (D, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
- $f : D \to D$ **monotone**
- $\text{Fix}(f) = \{ d \mid f(d) = d \}$
- $\text{Red}(f) = \{ d \mid f(d) \sqsubseteq d \}$
- $\text{Ext}(f) = \{ d \mid d \sqsubseteq f(d) \}$
- **Theorem** [Tarski 1955]
  - $\text{lfp}(f) = \sqcap\text{Fix}(f) = \sqcap\text{Red}(f) \in \text{Fix}(f)$
  - $\text{gfp}(f) = \sqcup\text{Fix}(f) = \sqcup\text{Ext}(f) \in \text{Fix}(f)$



Continuity and ACC condition

- Let $L = (D, \sqsubseteq, \sqcup, \bot)$ be a complete partial order
  - Every ascending chain has an upper bound
- A function f is continuous if for every increasing chain $Y \subseteq D^*$,

$$f(\sqcup Y) = \sqcup\{ f(y) \mid y \in Y \}$$

- L satisfies the ascending chain condition (ACC) if every ascending chain eventually stabilizes:

$$d_0 \sqsubseteq d_1 \sqsubseteq \ldots \sqsubseteq d_n = d_{n+1} = \ldots$$

Fixed-point theorem [Kleene]

- Let $L = (D, \sqsubseteq, \sqcup, \bot)$ be a complete partial order and a **continuous** function f: $D \to D$ then

$$\text{lfp}(f) = \bigsqcup_{n \in N} f^n(\bot)$$

Resulting algorithm

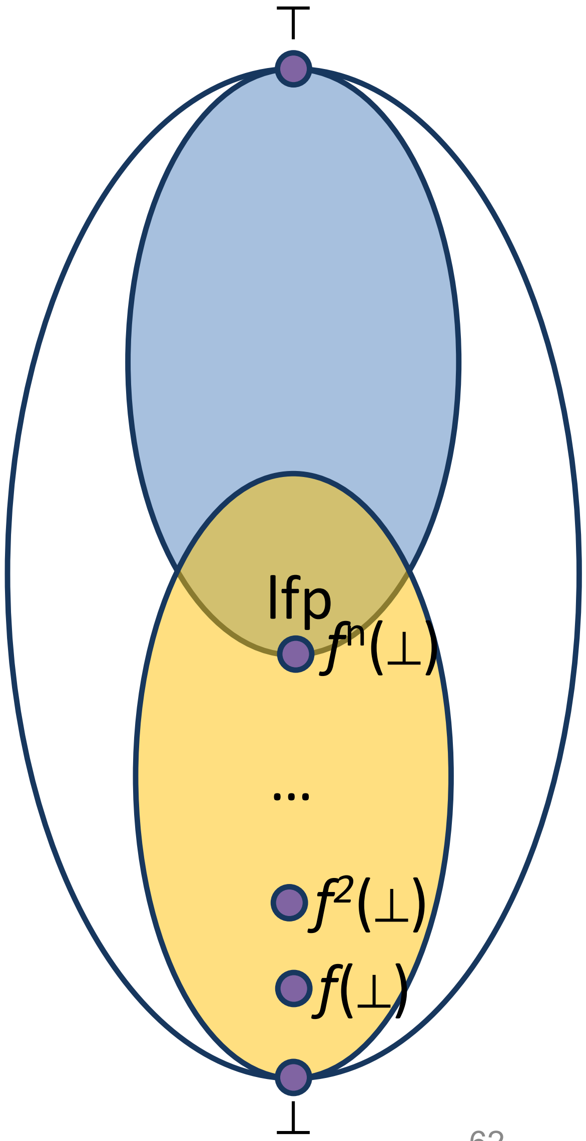
- Kleene's fixed point theorem gives a constructive method for computing the lfp

Mathematical definition

$$\text{lfp}(f) = \bigsqcup_{n \in N} f^n(\bot)$$

Algorithm

$d := \bot$
while $f(d) \neq d$ **do**
    $d := d \sqcup f(d)$
return d



Chaotic iteration

- Input:
  - A cpo $L = (D, \sqsubseteq, \sqcup, \perp)$ satisfying ACC
  - $L^n = L \times L \times \ldots \times L$
  - A monotone function $f : D^n \to D^n$
  - A system of equations $\{ X[\mathrm{i}] \mid f(X) \mid 1 \leq i \leq n \}$
- Output: lfp(f)
- A worklist-based algorithm

```
for i:=1 to n do
   X[i] := ⊥
WL = {1,…,n}
while WL ≠ ∅ do
   j := pop WL // choose index non-deterministically
   N := F[i](X)
   if N ≠ X[i] then
      X[i] := N
      add all the indexes that directly depend on i to WL
      (X[j] depends on X[i] if F[j] contains X[i])
return X
```

Concrete semantics equations

```
public void loopExample() {
R[0]  int x = 7;  R[1]
R[2]  while (x < 1000) {
R[3]      ++x;  R[4]
      }
R[5]  if (!(x == 1000))
R[6]      error("Unable to prove x == 1000!");
}
```

- $R[0] = \{\mathbf{x} \in \mathbf{Z}\}$
  $R[1] = [\![\mathbf{x:=7}]\!]$
  $R[2] = R[1] \cup R[4]$
  $R[3] = R[2] \cap \{s \mid s(x) < 1000\}$
  $R[4] = [\![\mathbf{x:=x+1}]\!]\ R[3]$
  $R[5] = R[2] \cap \{s \mid s(x) \geq 1000\}$
  $R[6] = R[5] \cap \{s \mid s(x) \neq 1001\}$

Abstract semantics equations

```
public void loopExample() {
R[0]  int x = 7;  R[1]
R[2]  while (x < 1000) {
R[3]       ++x;  R[4]
      }
R[5]  if (!(x == 1000))
R[6]       error("Unable to prove x == 1000!");
}
```

- $R[0] = \alpha(\{\mathbf{x} \in \mathbf{Z}\})$
  $R[1] = [\![\mathbf{x:=7}]\!]^{\#}$
  $R[2] = R[1] \sqcup R[4]$
  $R[3] = R[2] \sqcap \alpha(\{s \mid s(x) < 1000\})$
  $R[4] = [\![\mathbf{x:=x+1}]\!]^{\#}\ R[3]$
  $R[5] = R[2] \sqcap \alpha(\{s \mid s(x) \geq 1000\})$
  $R[6] = R[5] \sqcap \alpha(\{s \mid s(x) \geq 1001\}) \sqcup R[5] \sqcap \alpha(\{s \mid s(x) \leq 999\})$

Abstract semantics equations

```
public void loopExample() {
R[0]  int x = 7;  R[1]
R[2]  while (x < 1000) {
R[3]      ++x;  R[4]
      }
R[5]  if (!(x == 1000))
R[6]      error("Unable to prove x == 1000!");
}
```

- $R[0] = \top$
  $R[1] = [7,7]$
  $R[2] = R[1] \sqcup R[4]$
  $R[3] = R[2] \sqcap [-\infty,999]$
  $R[4] = R[3] + [1,1]$
  $R[5] = R[2] \sqcap [1000,+\infty]$
  $R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

Too many iterations to converge

```
Iteration 3981: processing V[8] = Interval[x==1000](V[6]) // if x == 1000 goto return
              V[8] : false
              V[6] : and(x=1000)
              V[8]' : and(x=1000)
              Adding [V[12] = Join_IntervalDomain(V[8], V[10]) // return]
              workSet = {V[12]}
Iteration 3982: processing V[12] = Join_IntervalDomain(V[8], V[10]) // return
              V[12] : false
              V[8] : and(x=1000)
              V[10] : false
              V[12]' : and(x=1000)
              Adding [V[11] = V[12] // return]
              workSet = {V[11]}
Iteration 3983: processing V[11] = V[12] // return
              V[11] : false
              V[12] : and(x=1000)
              V[11]' : and(x=1000)
              Adding []
Reached fixed-point after 3983 iterations.
Solution = {
  V[0] : true
  V[1] : true
  V[2] : and(x=7)
  V[3] : and(x=7)
  V[4] : and(8<=x<=1000)
  V[7] : and(7<=x<=1000)
  V[5] : and(7<=x<=999)
  V[6] : and(x=1000)
  V[8] : and(x=1000)
  V[9] : false
  V[10] : false
  V[12] : and(x=1000)
  V[11] : and(x=1000)
}
0 possible errors found.
Writing to sootOutput\IntervalExample.jimple
Soot finished on Wed Jun 12 06:24:14 IDT 2013
Soot has run for 0 min. 1 sec.
```

How many iterations for this one?

```
public void loopExample2(int y) {
    int x = 7;
    if (x < y) {
        while (x < y) {
            ++x;
        }

        if (x != y)
            error("Unable to prove x = y!");
    }
}
```

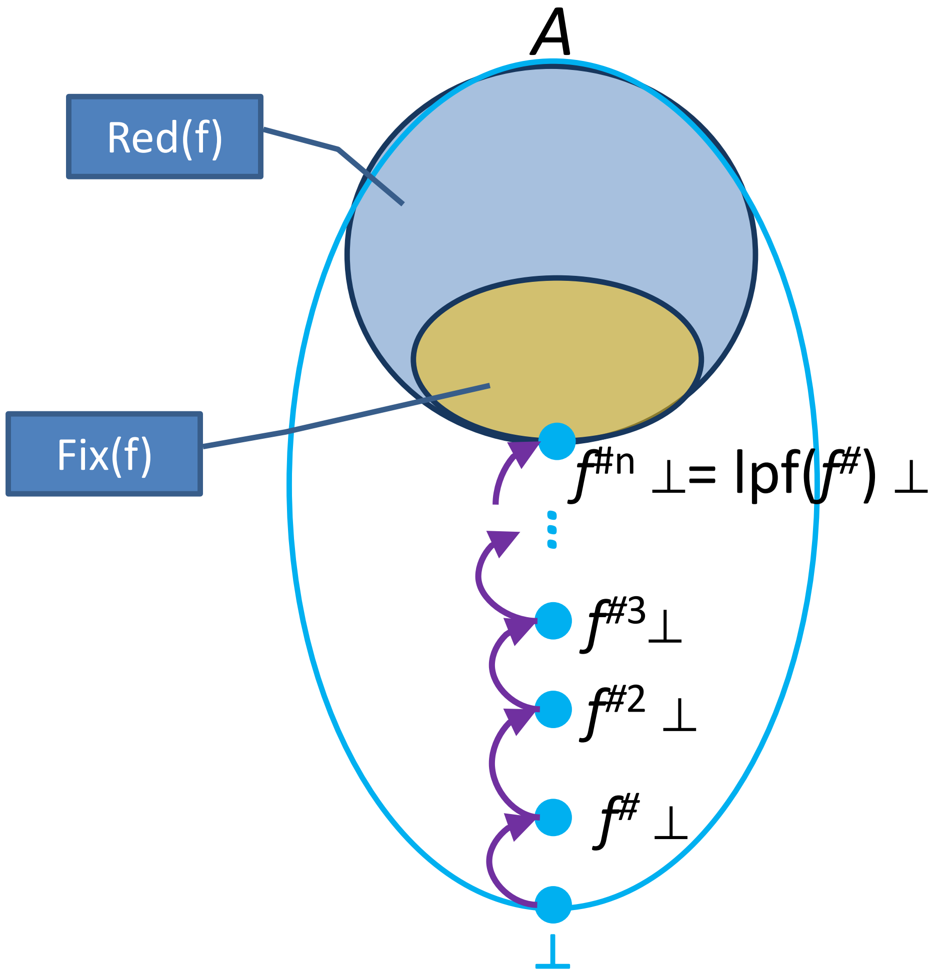
Widening

- Introduce a new binary operator to ensure termination
  - A kind of extrapolation
- Enables static analysis to use infinite height lattices
  - Dynamically adapts to given program
- Tricky to design
- Precision less predictable then with finite-height domains (widening non-monotone)

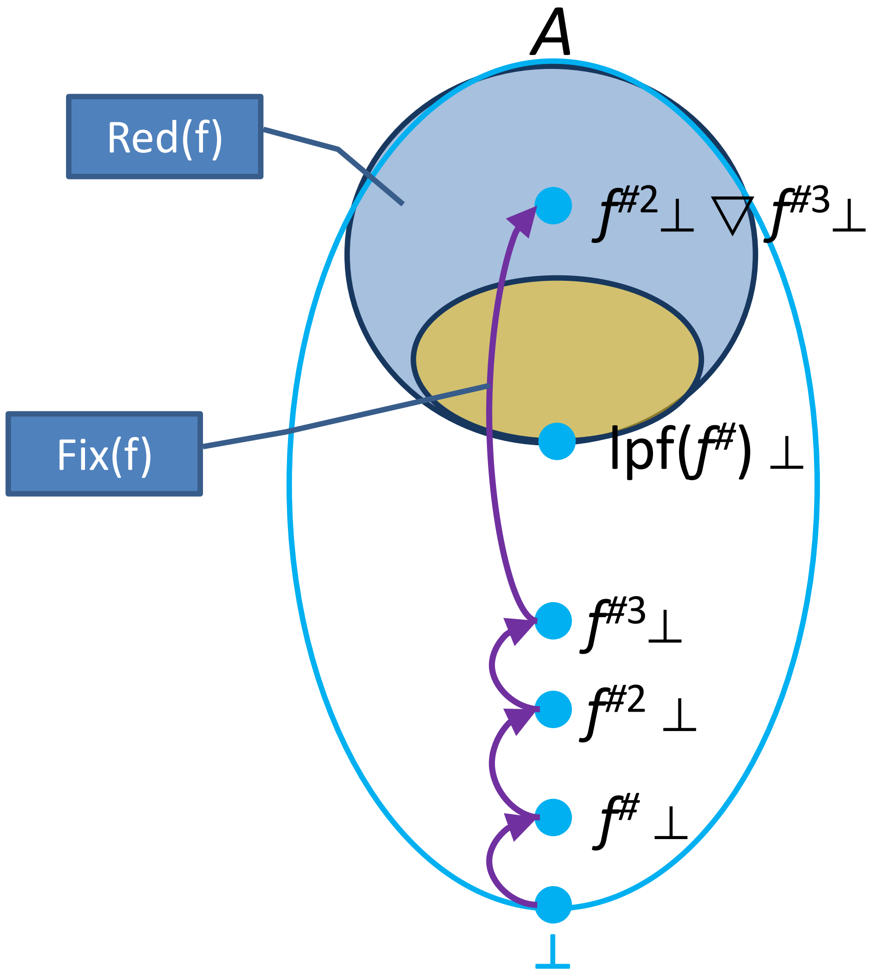
Formal definition

- For all elements $d_1 \sqcup d_2 \sqsubseteq d_1 \triangledown d_2$
- For all ascending chains $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ the following sequence eventually stabilizes
  - $y_0 = d_0$
  - $y_{i+1} = y_i \triangledown d_{i+1}$
- For a monotone function $f : D \rightarrow D$ define
  - $x_0 = \bot$
  - $x_{i+1} = x_i \triangledown f(x_i)$
- Theorem:
  - There exits k such that $x_{k+1} = x_k$
  - $x_k \in Red(f) = \{ d \mid d \in D \text{ and } f(d) \sqsubseteq d \}$

Analysis with finite-height lattice



Analysis with widening



Widening for Intervals Analysis

- $\bot \nabla [c, d] = [c, d]$
- $[a, b] \ \nabla [c, d] = [$

  if $a \leq c$

  then a

  else $-\infty$,

  if $b \geq d$

  then b

  else ∞

Semantic equations with widening

```
public void loopExample() {
R[0]  int x = 7;  R[1]
R[2]  while (x < 1000) {
R[3]      ++x;  R[4]
      }
R[5]  if (!(x == 1000))
R[6]      error("Unable to prove x == 1000!");
}
```

- $R[0] = \top$
  $R[1] = [7,7]$
  $R[2] = R[1] \sqcup R[4]$
  $R[2.1] = R[2.1] \triangledown R[2]$
  $R[3] = R[2.1] \sqcap [-\infty,999]$
  $R[4] = R[3] + [1,1]$
  $R[5] = R[2] \sqcap [1001,+\infty]$
  $R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

Non monotonicity of widening

- $[0,1] \bigtriangledown [0,2] = ?$
- $[0,2] \bigtriangledown [0,2] = ?$

Non monotonicity of widening

- $[0,1] \triangledown [0,2] = [0, \infty]$
- $[0,2] \triangledown [0,2] = [0,2]$

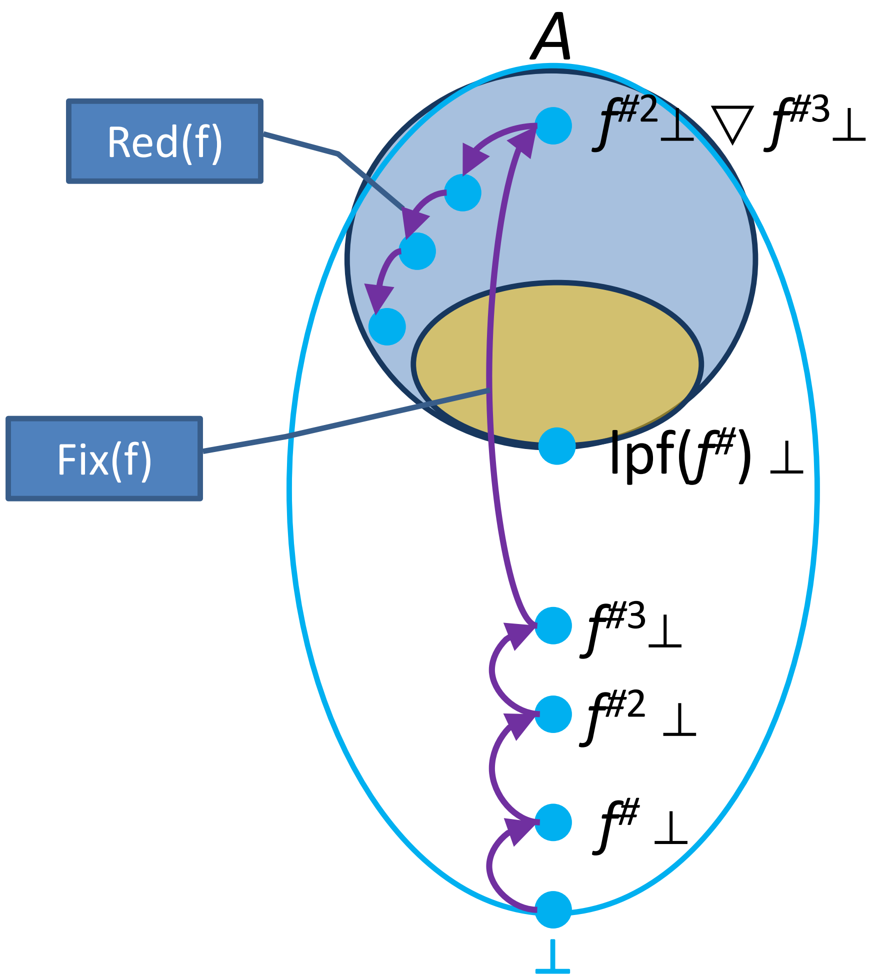
Analysis results with widening

```
Analyzing method loopExample
-------------------------------------------
Solving the following equation system =
V[0] = true // this := @this: IntervalExample
V[1] = AssignTopTransformer(V[0]) // this := @this: IntervalExample
V[2] = AssignConstantToVarTransformer(V[1]) // x = 7
V[3] = V[2] // goto [?= (branch)]
V[4] = AssignAddExprToVarTransformer(V[5]) // x = x + 1
V[7] = JoinLoop_IntervalDomain(V[3], V[4]) // if x < 1000 goto x = x + 1
V[8] = IntervalDomain[Widening|Narrowing](V[8], V[7]) // if x < 1000 goto x = x + 1
V[5] = Interval[x<1000](V[8]) // if x < 1000 goto x = x + 1
V[6] = Interval[x>=1000](V[8]) // if x < 1000 goto x = x + 1
V[9] = Interval[x==1000](V[6]) // if x == 1000 goto return
V[10] = Interval[x!=1000](V[6]) // if x == 1000 goto return
V[11] = V[10] // specialinvoke this.<IntervalExample: void error(java.lang.String)>("Unable to prove x == 1000!")
V[13] = Join_IntervalDomain(V[9], V[11]) // return
V[12] = V[13] // return
```

```
Reached fixed-point after 23 iterations.
Solution = {
  V[0] : true
  V[1] : true
  V[2] : and(x=7)
  V[3] : and(x=7)
  V[4] : and(8<=x<=1000)
  V[7] : and(7<=x<=1000)
  V[8] : and(x>=7)
  V[5] : and(7<=x<=999)
  V[6] : and(x>=1000)
  V[9] : and(x=1000)
  V[10] : and(x>=1001)
  V[11] : and(x>=1001)
  V[13] : and(x>=1000)
  V[12] : and(x>=1000)
}
```

Did we prove it?

Analysis with narrowing



Formal definition of narrowing

- Improves the result of widening
- $y \sqsubseteq x \Rightarrow y \sqsubseteq (x \triangle y) \sqsubseteq x$
- For all decreasing chains $x_0 \sqsupseteq x_1 \sqsupseteq \ldots$ the following sequence is finite
  - $y_0 = x_0$
  - $y_{i+1} = y_i \triangle x_{i+1}$
- For a monotone function $f: D \rightarrow D$ and $x_k \in \mathrm{Red}(f) = \{ d \mid d \in D \text{ and } f(d) \sqsubseteq d \}$ define
  - $y_0 = x$
  - $y_{i+1} = y_i \triangle f(y_i)$
- Theorem:
  - There exits k such that $y_{k+1} = y_k$
  - $y_k \in \mathrm{Red}(f) = \{ d \mid d \in D \text{ and } f(d) \sqsubseteq d \}$

Narrowing for Interval Analysis

- $[a, b] \triangle \bot = [a, b]$
- $[a, b] \triangle [c, d] = [$

    if $a = -\infty$

    then c

    else a,

  if $b = \infty$

    then d

    else b

    ]

Semantic equations with narrowing

```
public void loopExample() {
R[0]  int x = 7; R[1]
R[2]  while (x < 1000) {
R[3]      ++x; R[4]
      }
R[5]  if (!(x == 1000))
R[6]      error("Unable to prove x == 1000!");
}
```

- R[0] = $\top$
  R[1] = [7,7]
  R[2] = R[1] $\sqcup$ R[4]
  R[2.1] = R[2.1] $\triangle$ R[2]
  R[3] = R[2.1] $\sqcap$ [-∞,999]
  R[4] = R[3]+[1,1]
  R[5] = R[2]$^{\#}$ $\sqcap$ [1000,+∞]
  R[6] = R[5] $\sqcap$ [999,+∞] $\sqcup$ R[5] $\sqcap$ [1001,+∞]

Analysis with widening/narrowing

- Two phases
  - Phase 1: analyze with widening until converging
  - Phase 2: use values to analyze with narrowing

```
public void loopExample() {
    int x = 7;
    while (x < 1000) {
        ++x;
    }
    if (!(x == 1000))
        error("Unable to prove x == 1000!");
}
```

Phase 1:

$R[0] = \top$

$R[1] = [7,7]$

$R[2] = R[1] \sqcup R[4]$

$R[2.1] = R[2.1] \bigtriangledown R[2]$

$R[3] = R[2.1] \sqcap [-\infty,999]$

$R[4] = R[3] + [1,1]$

$R[5] = R[2] \sqcap [1001,+\infty]$

$R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

Phase 2:

$R[0] = \top$

$R[1] = [7,7]$

$R[2] = R[1] \sqcup R[4]$

$R[2.1] = R[2.1] \bigtriangleup R[2]$

$R[3] = R[2.1] \sqcap [-\infty,999]$

$R[4] = R[3]+[1,1]$

$R[5] = R[2]^{\#} \sqcap [1000,+\infty]$

$R[6] = R[5] \sqcap [999,+\infty] \sqcup R[5] \sqcap [1001,+\infty]$

Analysis with widening/narrowing

```
Reached fixed-point after 23 iterations.
Solution = {
  V[0] : true
  V[1] : true
  V[2] : and(x=7)
  V[3] : and(x=7)
  V[4] : and(8<=x<=1000)
  V[7] : and(7<=x<=1000)
  V[8] : and(x>=7)
  V[5] : and(7<=x<=999)
  V[6] : and(x>=1000)
  V[9] : and(x=1000)
  V[10] : and(x>=1001)
  V[11] : and(x>=1001)
  V[13] : and(x>=1000)
  V[12] : and(x>=1000)
}

Starting chaotic iteration: narrowing phase...
              workSet = {V[0], V[1], V[2], V[3], V[4], V[7], V[8], V[5], V[6], V[9], V[10], V[11], V[13], V[12]}
Iteration 24: processing V[0] = true // this := @this: IntervalExample
              V[0] : true
              V[0]' : true
              workSet = {V[12], V[1], V[2], V[3], V[4], V[7], V[8], V[5], V[6], V[9], V[10], V[11], V[13]}
```

Analysis results widening/narrowing

```
Iteration 44: processing V[1] = AssignTopTransformer(V[0]) // this := @this: IntervalExample
              V[1] : true
              V[0] : true
              V[1]' : true
Reached fixed-point after 44 iterations.
Solution = {
  V[0] : true
  V[1] : true
  V[2] : and(x=7)
  V[3] : and(x=7)
  V[4] : and(8<=x<=1000)
  V[7] : and(7<=x<=1000)
  V[8] : and(7<=x<=1000)
  V[5] : and(7<=x<=999)
  V[6] : and(x=1000)
  V[9] : and(x=1000)
  V[10] : false
  V[11] : false
  V[13] : and(x=1000)
  V[12] : and(x=1000)
}
0 possible errors found.
Writing to sootOutput\IntervalExample.jimple
Soot finished on Wed Jun 12 06:47:24 IDT 2013
Soot has run for 0 min. 0 sec.
```

Precise invariant

Project

- 1-2 Students in a group
  - 3-4: Bigger projects
- Theoretical + Practical
- Your choice of topic
  - Contact me in 3 weeks
- Submission – 15/Sep
  - Code + Examples
  - Document
  - 15 minutes presentation

Past projects

- JavaScript Dominator Analysis
- Attribute Analysis for JavaScript
- Simple Pointer Analysis for C
- Adding program counters to Past Abstraction (abstraction of finite state machines.)
- Verification of Asynchronous programs
- Verifying SDNs using TVLA
- Verifying independent accesses to arrays in GO

Past projects

- Detecting index out of bound errors in C programs
- Lattice-Based Semantics for Combinatorial Models Evolution
- Verifying sorting programs
- Cross-array sorting (array of arrays) – use for storage systems version management
- Verifying LTL formulae over TVLA structures
- Worst-case memory consumption

Past projects

- Automatic loop parallelization via dependency tracking
- Handling asynchronous calls

Default Project

- Pick a framework
  - LLVM (C) : http://llvm.org/
  - Soot (Java) : https://sable.github.io/soot/

- Analysis:
  - Refined pointer analysis
  - Invent numerical domain