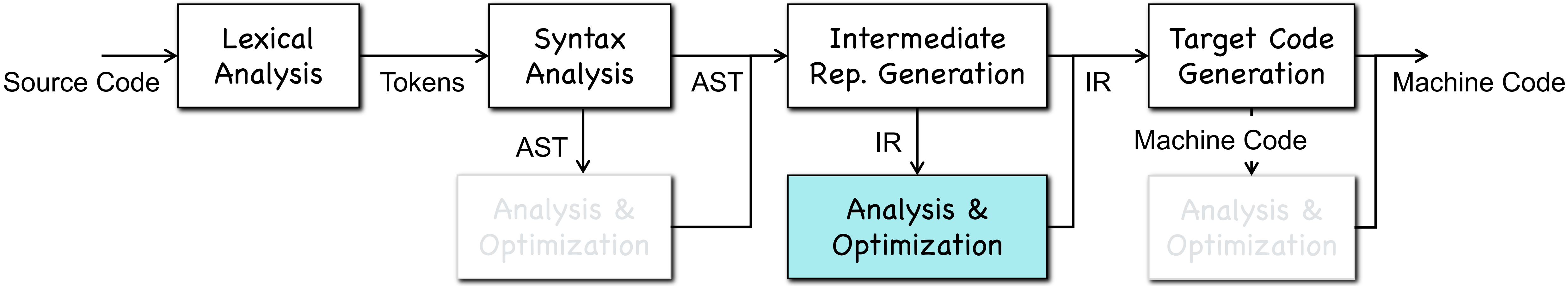
Made by Qingkai Shi
qingkaishi@nju.edu.cn

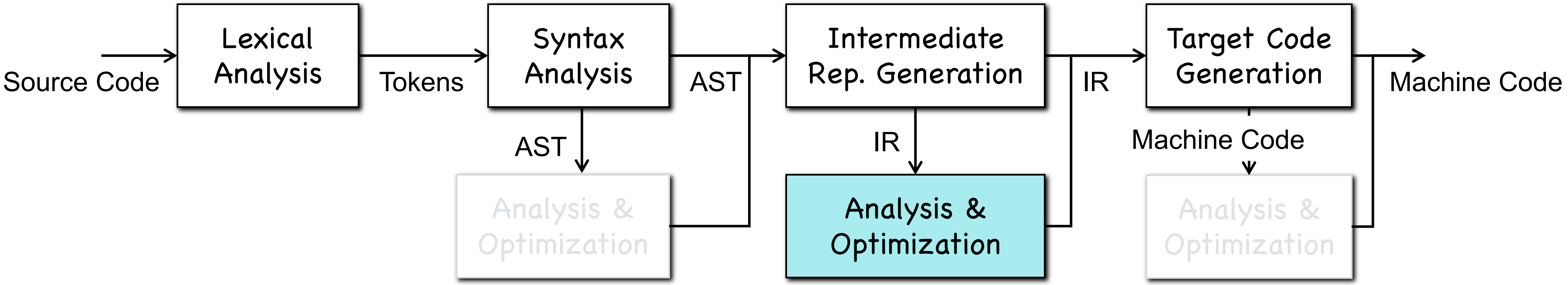
Chapter 12-2
Pointer Alias Analysis

Pointer Analysis



- Why pointer analysis?

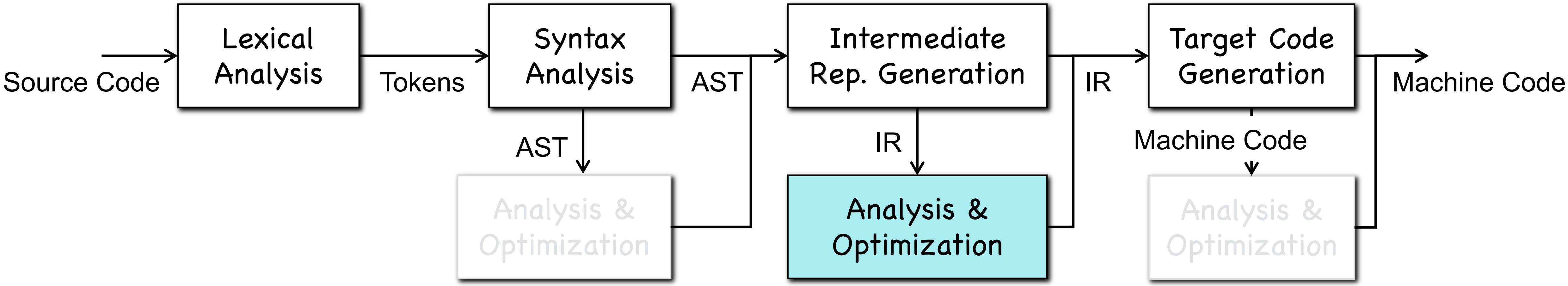
Pointer Analysis



- Why pointer analysis?

```
*p = a; …; …; b = *q; // does b data-depend on a?
```

Pointer Analysis

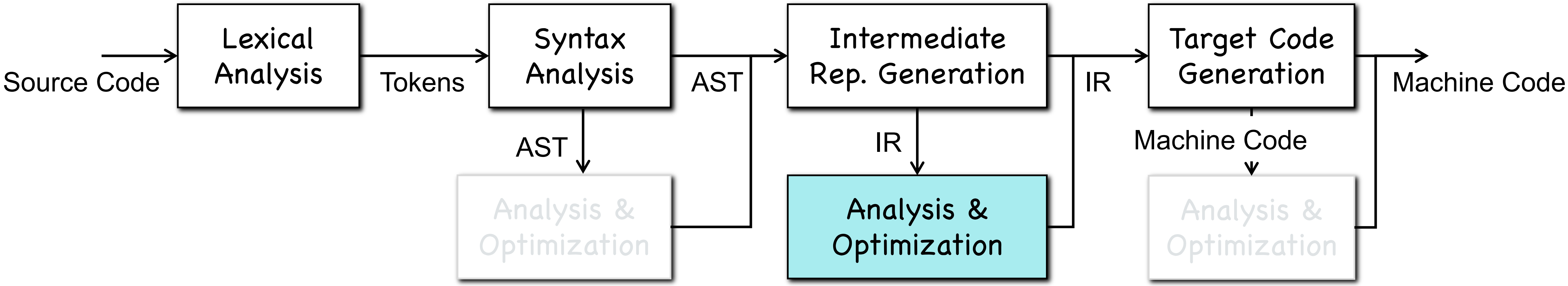


- Why pointer analysis?

```
*p = a; …; …; b = *q; // does b data-depend on a?
```

Pointer analysis is the most fundamental program analysis!

Pointer Analysis



- Why pointer analysis?

```
*p = a; …; …; b = *q; // does b data-depend on a?
```

Pointer analysis is the most fundamental program analysis!

- Flow-sensitivity
- Context-sensitivity
- Path-sensitivity
- Field-sensitivity

Why Pointer Alias Analysis?

- **Aliases**: two expressions that denote the same memory location
- Aliases are introduced by
  - references/pointers
  - function call
  - array indexing
  - C unions
  - …

Why Pointer Alias Analysis?

- **Aliases**: two expressions that denote the same memory location

- Aliases are introduced by
  - references/pointers
  - function call
  - array indexing
  - C unions
  - …

```
int x,y;
int *p = &x;
int *q = &y;
int *r = p;
int **s = &q;
```

Why Pointer Alias Analysis?

- **Aliases**: two expressions that denote the same memory location

- Aliases are introduced by
  - references/pointers
  - function call
  - array indexing
  - C unions
  - …

```
int x,y;
int *p = &x;
int *q = &y;
int *r = p;
int **s = &q;
```

Why Pointer Alias Analysis?

- Improving the precision of **compiler optimizations** that require knowing what is modified or referenced

Why Pointer Alias Analysis?

- Improving the precision of **compiler optimizations** that require knowing what is modified or referenced

```
x = 3;
*p = 4;
y = x; // can constant 3 propagate here
```

Constant Propagation

Why Pointer Alias Analysis?

- Improving the precision of **compiler optimizations** that require knowing what is modified or referenced

```
x = 3;
*p = 4;
y = x; // can constant 3 propagate here
```

Constant Propagation

```
t = a + b;
*p = t
y = a + b; // is a + b available?
```

Available Expression

Why Pointer Alias Analysis?

- Improving the precision of **compiler optimizations** that require knowing what is modified or referenced

```
x = 3;
*p = 4;
y = x; // can constant 3 propagate here
```

Constant Propagation

```
t = a + b;
*p = t
y = a + b; // is a + b available?
```

Available Expression

- Error detection

```
x.lock();
...
y.unlock(); // same object as x?
```

PART I: Pointer Alias Analysis

May/Must Pointer Alias Analysis

- **May- (Must-)** analysis checks if two expressions **may (must)** denote the same memory location.

May/Must Pointer Alias Analysis

- **May- (Must-)** analysis checks if two expressions **may (must)** denote the same memory location.

- May analysis – Must-not analysis
  - If a *may-analysis* says ‘no’, it means two expressions must not denote the same memory location.
  - Useful in compiler optimization.

May/Must Pointer Alias Analysis

- **May- (Must-)** analysis checks if two expressions **may (must)** denote the same memory location.

- May analysis – Must-not analysis
  - If a *may-analysis* says 'no', it means two expressions must not denote the same memory location.
  - Useful in compiler optimization.

```
x = 3;
*p = 4;
y = x; // can constant 3 propagate here
```

Constant Propagation

```
t = a + b;
*p = t
y = a + b; // a + b is available?
```

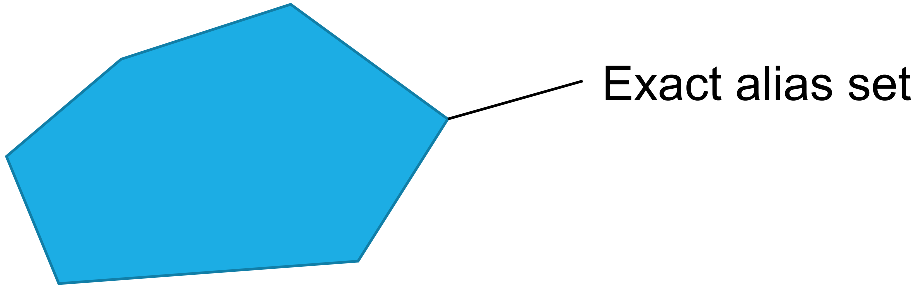
Available Expression

May/Must Pointer Alias Analysis

- Sound/Complete or Over-approximation/Under-approximation
- May analysis provides a sound (or over-approximated) alias set

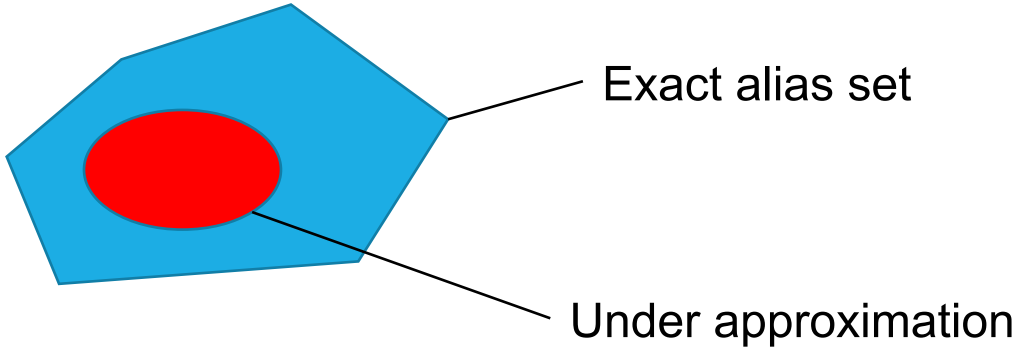
May/Must Pointer Alias Analysis

- Sound/Complete or Over-approximation/Under-approximation
- May analysis provides a sound (or over-approximated) alias set



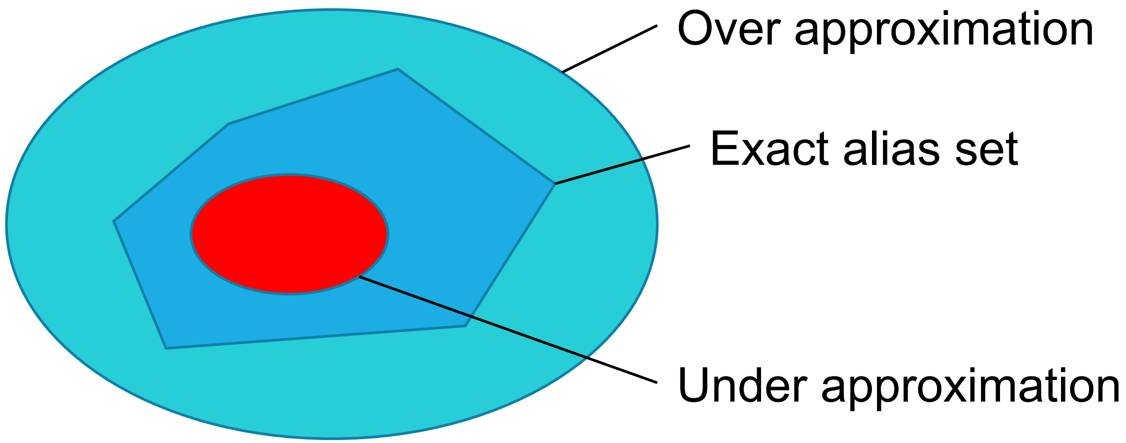
May/Must Pointer Alias Analysis

- Sound/Complete or Over-approximation/Under-approximation
- May analysis provides a sound (or over-approximated) alias set



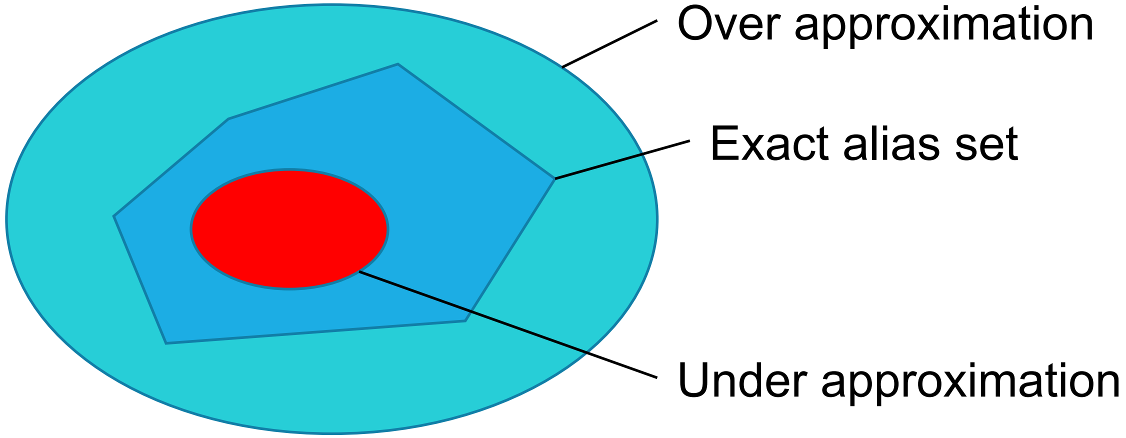
May/Must Pointer Alias Analysis

- Sound/Complete or Over-approximation/Under-approximation
- May analysis provides a sound (or over-approximated) alias set



May/Must Pointer Alias Analysis

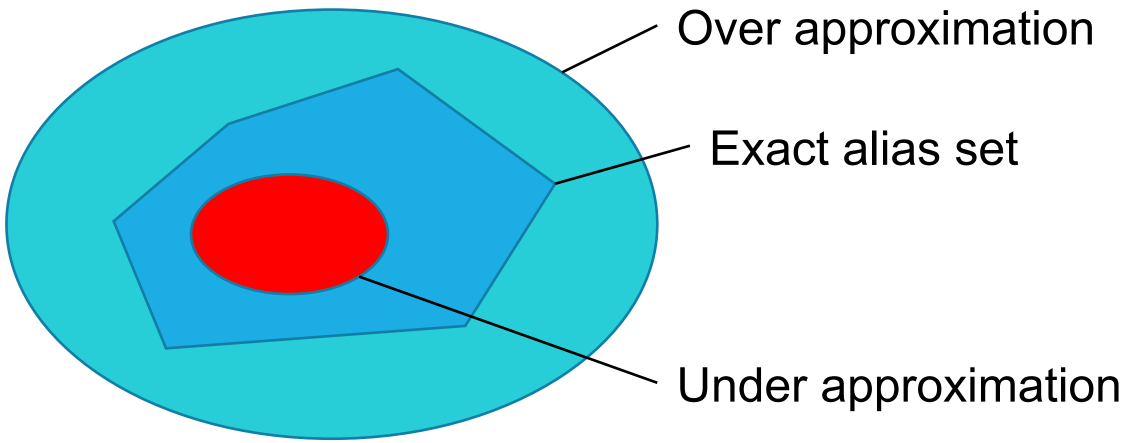
- Sound/Complete or Over-approximation/Under-approximation
- May analysis provides a sound (or over-approximated) alias set



The simplest sound alias analysis is to return true for all may-alias queries

May/Must Pointer Alias Analysis

- Sound/Complete or Over-approximation/Under-approximation
- May analysis provides a sound (or over-approximated) alias set



The simplest sound alias analysis is to return true for all may-alias queries

- Tradeoff between soundness and precision (i.e., completeness)

Flow-Sensitive Pointer Analysis

- Flow-sensitive pointer analysis computes **for each program point** what memory locations a pointer expression may refer to

Flow-Sensitive Pointer Analysis

- Flow-sensitive pointer analysis computes **for each program point** what memory locations a pointer expression may refer to
- Flow-sensitive pointer analysis is **(traditionally) too expensive** to perform for whole program

Flow-Sensitive Pointer Analysis

- Flow-sensitive pointer analysis computes **for each program point** what memory locations a pointer expression may refer to
- Flow-sensitive pointer analysis is **(traditionally) too expensive** to perform for whole program
- There have been a few studies for flow-sensitive or even path-sensitive analysis

Flow-Sensitive Pointer Analysis

- Flow-sensitive pointer analysis computes **for each program point** what memory locations a pointer expression may refer to
- Flow-sensitive pointer analysis is **(traditionally) too expensive** to perform for whole program
- There have been a few studies for flow-sensitive or even path-sensitive analysis

[1] Falcon: A Fused Approach to Path-Sensitive Data Dependence Analysis
Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang, PLDI 2024.

[2] Value-Flow-Based Demand-Driven Pointer Analysis for C and C++
Yulei Sui and Jingling Xue, IEEE Transactions on Software Engineering 2018.

Flow-Insensitive Pointer Analysis

- Flow-insensitive pointer analyses for whole program analyses

Andersen's Algorithm	**Steensgaard's Algorithm**
Presented by Andersen in 1994	Published in POPL 1996
Nearly cubic complexity	Almost linear, $O(n\alpha(n))$

A Small Pointer Language

- y = &x (address-taken)
- y = x (assignment)
- *y = x (store statement)
- y = *x (load statement)

A Small Pointer Language

- y = &x (address-taken)
- y = x (assignment)
- *y = x (store statement)
- y = *x (load statement)

- **pts(x)** --- the points-to set of x
  - pts(x) = { y, z } --- x may point to either y or z

Andersen’s Algorithm

Statement	Constraint	Shorthand
y = &x		
y = x		
*y = x		
y = *x		

Andersen's Algorithm

Statement	Constraint	Shorthand
y = &x	pts(y) ⊇ {x}	
y = x		
*y = x		
y = *x		

Andersen’s Algorithm

Statement	Constraint	Shorthand
y = &x	pts(y) ⊇ {x}	
y = x	pts(y) ⊇ pts(x)	
*y = x		
y = *x		

Andersen's Algorithm

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x		$pts(*y) \supseteq pts(x)$
y = *x		

Andersen's Algorithm

Statement	Constraint	Shorthand
y = &x	pts(y) ⊇ {x}	
y = x	pts(y) ⊇ pts(x)	
*y = x	∀ v ∈ pts(y): pts(v) ⊇ pts(x)	pts(*y) ⊇ pts(x)
y = *x		

Andersen's Algorithm

Statement	Constraint	Shorthand
y = &x	pts(y) ⊇ {x}	
y = x	pts(y) ⊇ pts(x)	
*y = x	∀ v ∊ pts(y): pts(v) ⊇ pts(x)	pts(*y) ⊇ pts(x)
y = *x		pts(y) ⊇ pts(*x)

Andersen's Algorithm

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall v \in pts(y): pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall v \in pts(x): pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

Andersen's Algorithm

Inclusion-based pointer analysis

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall\ v \in pts(y)\text{:}\ \ pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall\ v \in pts(x)\text{:}\ \ pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

Andersen’s Algorithm

Inclusion-based pointer analysis

Statement	Constraint	Shorthand
y = &x	pts(y) ⊇ {x}	
y = x	pts(y) ⊇ pts(x)	
*y = x	∀ v ∊ pts(y): pts(v) ⊇ pts(x)	pts(*y) ⊇ pts(x)
y = *x	∀ v ∊ pts(x): pts(y) ⊇ pts(v)	pts(y) ⊇ pts(*x)

```
p = &a;
q = p;
p = &b;
r = p;
```

Andersen’s Algorithm

Inclusion-based pointer analysis

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall\ v \in pts(y):\ pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall\ v \in pts(x):\ pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

```
p = &a;
q = p;
p = &b;
r = p;
```

➡

```
pts(p) ⊇ { a };
pts(q) ⊇ pts(p);
pts(p) ⊇ { b };
pts(r) ⊇ pts(p);
```

Andersen’s Algorithm

Inclusion-based pointer analysis

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall\ v \in pts(y)$: $pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall\ v \in pts(x)$: $pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

```
p = &a;
q = p;
p = &b;
r = p;
```

```
pts(p) ⊇ { a };
pts(q) ⊇ pts(p);
pts(p) ⊇ { b };
pts(r) ⊇ pts(p);
```

```
pts(p) = { a, b };
```

Andersen’s Algorithm

Inclusion-based pointer analysis

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall\ v \in pts(y)$: $pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall\ v \in pts(x)$: $pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

```
p = &a;
q = p;
p = &b;
r = p;
```

```
pts(p) ⊇ { a };
pts(q) ⊇ pts(p);
pts(p) ⊇ { b };
pts(r) ⊇ pts(p);
```

```
pts(p) = { a, b };
pts(q) = { a, b };
pts(r) = { a, b };
```

Andersen's Algorithm

Inclusion-based pointer analysis

Statement	**Constraint**	**Shorthand**
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall\ v \in pts(y):\ \ pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall\ v \in pts(x):\ \ pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

```
p = &a;
q = p;
p = &b;
r = p;
```

➡

```
pts(p) ⊇ { a };
pts(q) ⊇ pts(p);
pts(p) ⊇ { b };
pts(r) ⊇ pts(p);
```

➡

```
pts(p) = { a, b };
pts(q) = { a, b };
pts(r) = { a, b };
pts(a) = pts(b) = {};
```

Andersen's Algorithm

Inclusion-based pointer analysis

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall v \in pts(y)$: $pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall v \in pts(x)$: $pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

```
p = &a;
q = p;
p = &b;
r = p;
```

```
pts(p) ⊇ { a };
pts(q) ⊇ pts(p);
pts(p) ⊇ { b };
pts(r) ⊇ pts(p);
```

```
pts(p) = { a, b };
pts(q) = { a, b };
pts(r) = { a, b };
pts(a) = pts(b) = {};
```

Over-approximation!
Sound result!

Andersen's Algorithm

Inclusion-based pointer analysis

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall\, v \in pts(y)$: $pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall\, v \in pts(x)$: $pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

```
p  =  &a;
q  =  &b;
*p =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s =  r;
```

?

```
p = &a;
q = p;
p = &b;
r = p;
```

⇨

```
pts(p) ⊇ { a };
pts(q) ⊇ pts(p);
pts(p) ⊇ { b };
pts(r) ⊇ pts(p);
```

⇨

```
pts(p) = { a, b };
pts(q) = { a, b };
pts(r) = { a, b };
pts(a) = pts(b) = {};
```

Over-approximation!
Sound result!

Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  = &a;
q  = &b;
*p = q;
r  = &c;
s  = p;
t  = *p;
*s = r;
```

Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  =  &a;
q  =  &b;
*p =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s =  r;
```

⇨

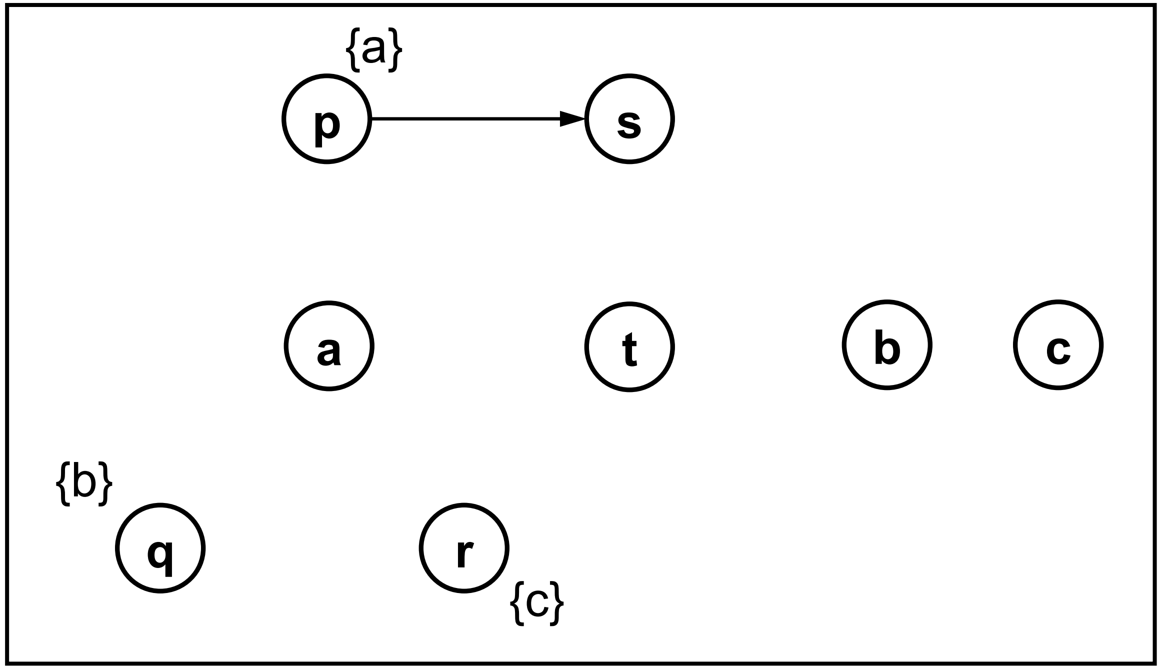
```
pts(p) ⊇ {a};
pts(q) ⊇ {b};
...
pts(r) ⊇ {c};
pts(s) ⊇ pts(p);
...
...
```

Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  =  &a;
q  =  &b;
*p =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s =  r;
```

```
pts(p) ⊇ {a};
pts(q) ⊇ {b};
...
pts(r) ⊇ {c};
pts(s) ⊇ pts(p);
...
...
```

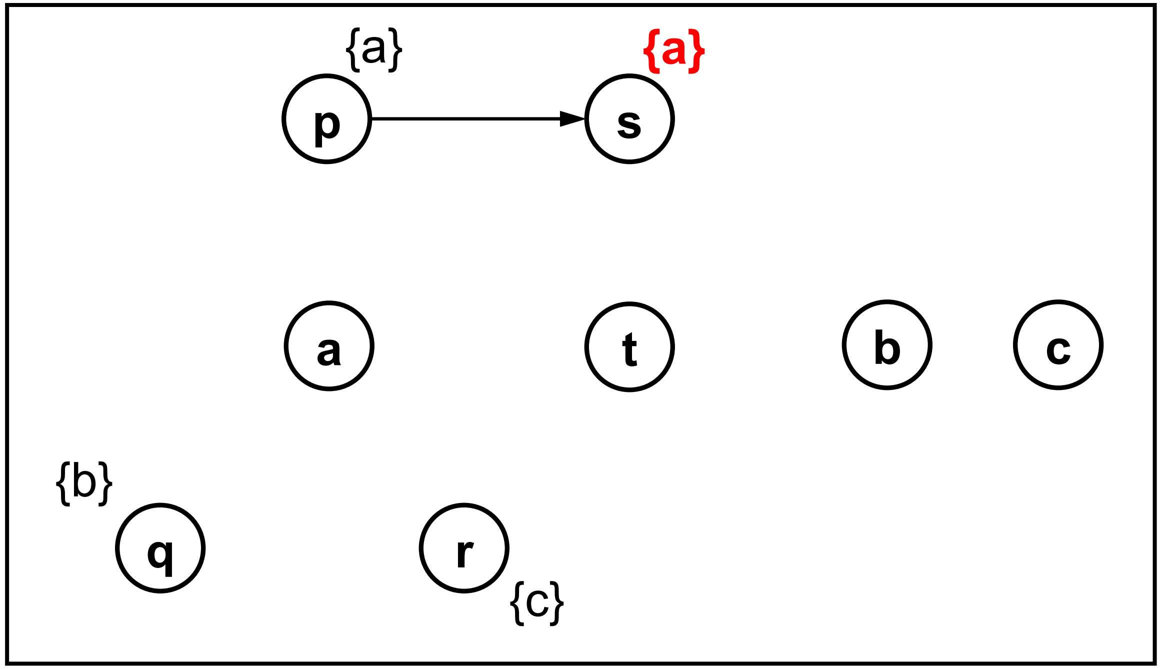


Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  =  &a;
q  =  &b;
*p =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s =  r;
```

```
pts(p) ⊇ {a};
pts(q) ⊇ {b};
...
pts(r) ⊇ {c};
pts(s) ⊇ pts(p);
...
...
```

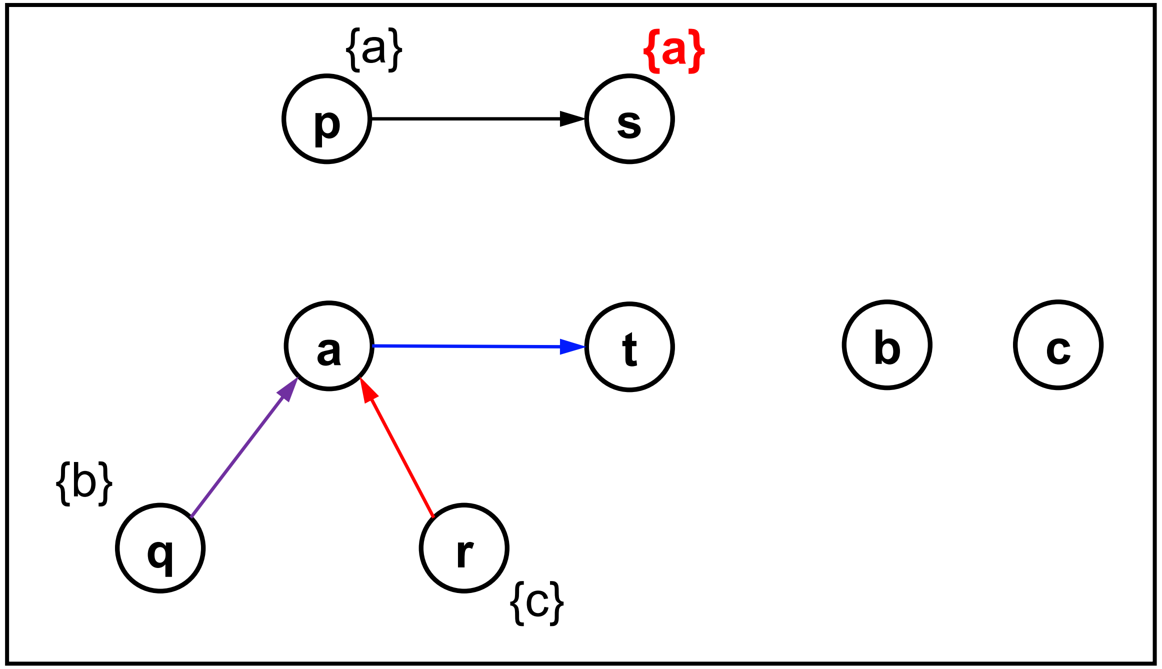


Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  =  &a;
q  =  &b;
*p =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s =  r;
```

```
...
...
pts(*p) ⊇ pts(q)
...
...
pts(t) ⊇ pts(*p)
pts(*s) ⊇ pts(r)
```

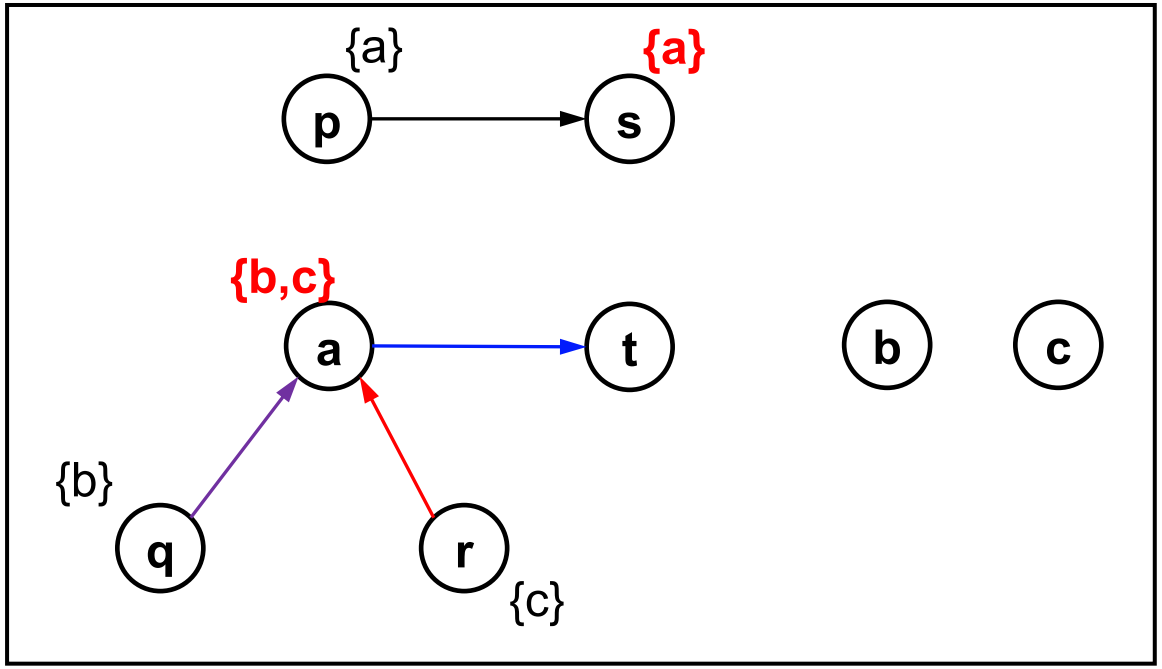


Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  =  &a;
q  =  &b;
*p =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s =  r;
```

```
...
...
pts(*p) ⊇ pts(q)
...
...
pts(t) ⊇ pts(*p)
pts(*s) ⊇ pts(r)
```

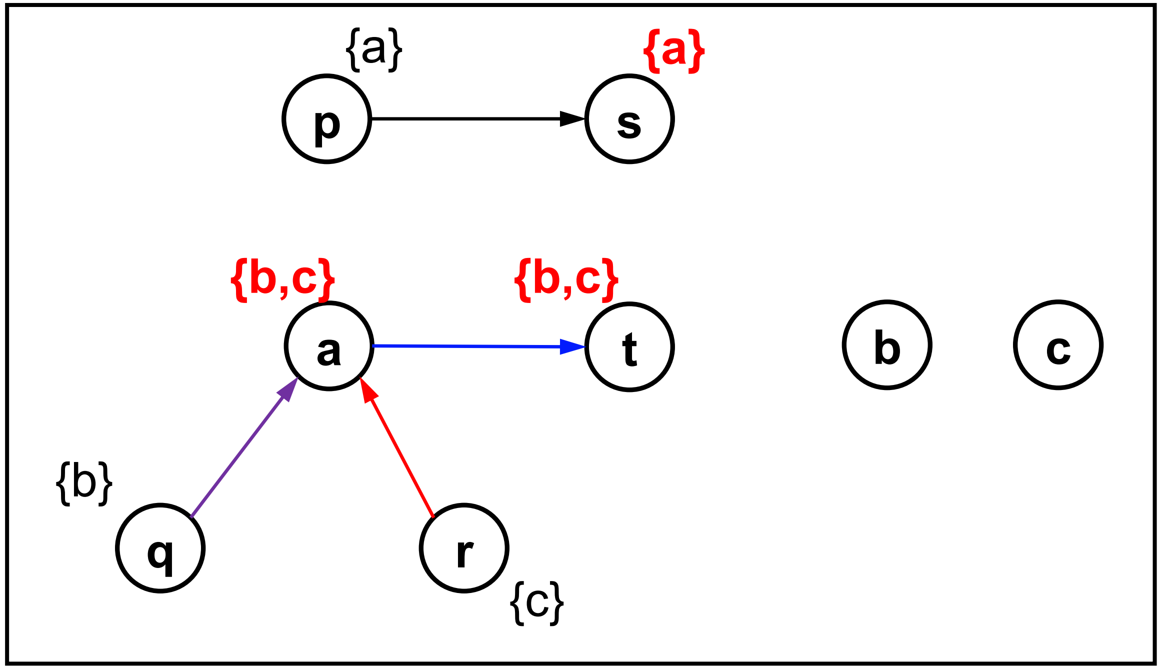


Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  =  &a;
q  =  &b;
*p =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s =  r;
```

```
...
...
pts(*p) ⊇ pts(q)
...
...
pts(t) ⊇ pts(*p)
pts(*s) ⊇ pts(r)
```

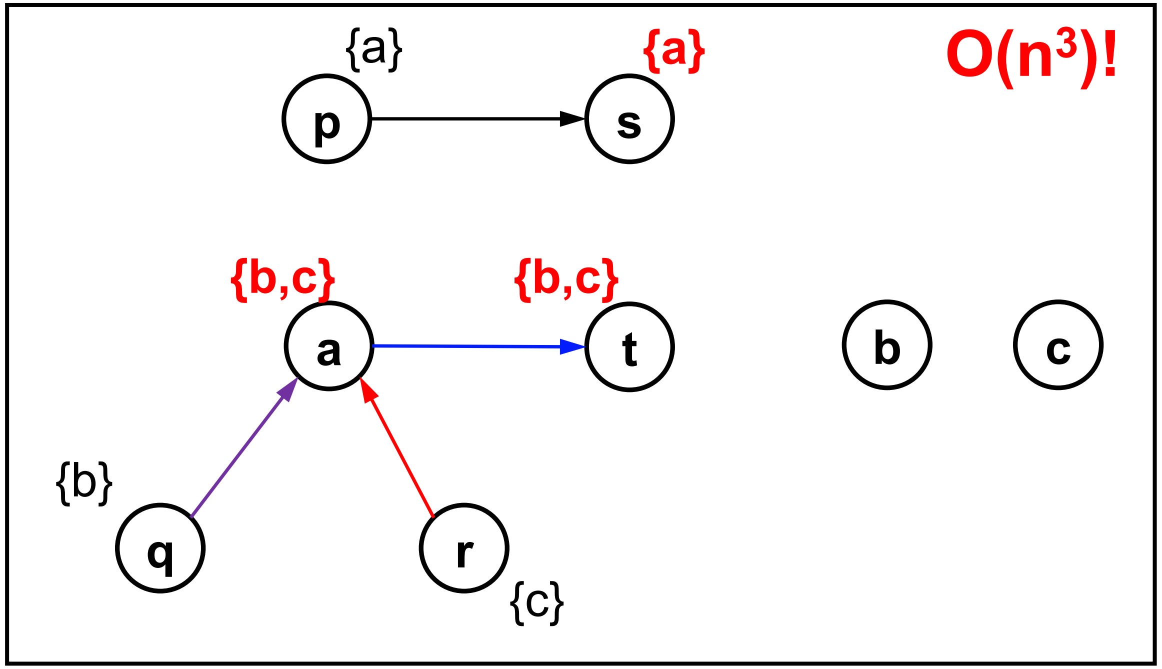


Andersen's Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  =  &a;
q  =  &b;
*p =  q;
r  =  &c;
s  =  p;
t  =  *p;
*s =  r;
```

```
...
...
pts(*p) ⊇ pts(q)
...
...
pts(t) ⊇ pts(*p)
pts(*s) ⊇ pts(r)
```

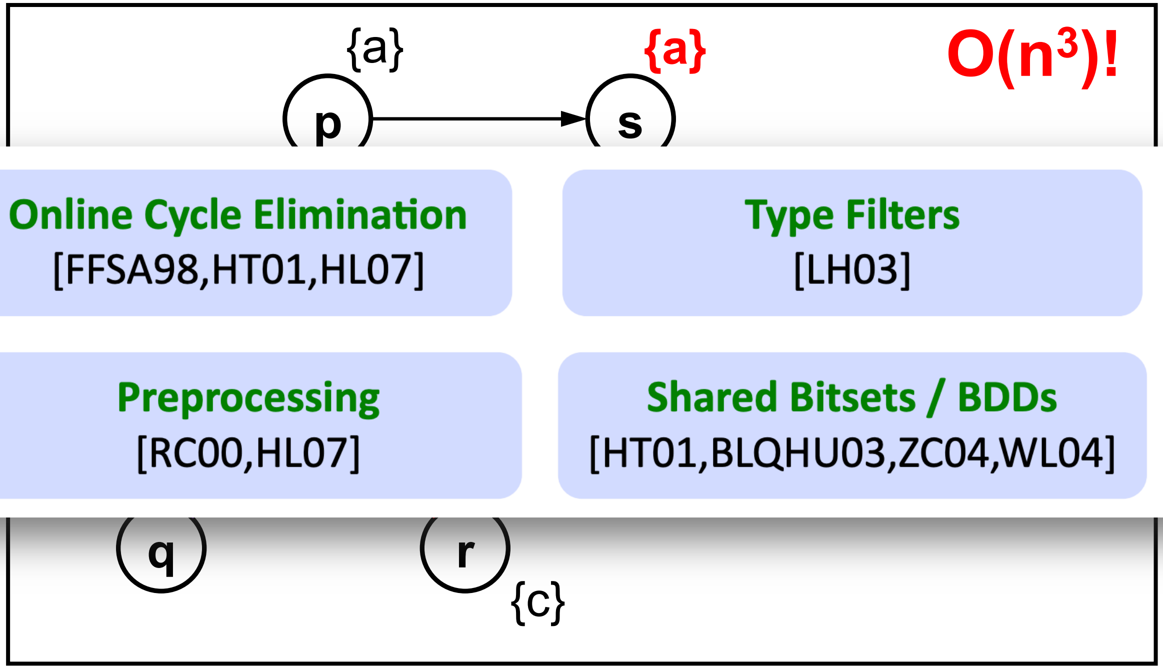


Andersen’s Alg. as Graph Closure

- Each graph node x denotes the points-to set $pts(x)$
- Solving the set constraints via a dynamic transitive closure

```
p  = &a;
q  = &b;
*p = q;
r  = &c;
s  = p;
t  = *p;
*s = r;
```

```
...
...
pts(*p) ⊇ pts(q)
...
...
pts(t) ⊇ pts(*p)
pts(*s) ⊇ pts(r)
```



Online Cycle Elimination
[FFSA98,HT01,HL07]

Type Filters
[LH03]

Preprocessing
[RC00,HL07]

Shared Bitsets / BDDs
[HT01,BLQHU03,ZC04,WL04]

Steensgaard's Algorithm

Inclusion-based pointer analysis (Andersen's Algorithm)

Statement	Constraint	Shorthand
y = &x	pts(y) ⊇ {x}	
y = x	pts(y) ⊇ pts(x)	
*y = x	∀ v ∊ pts(y): pts(v) ⊇ pts(x)	pts(*y) ⊇ pts(x)
y = *x	∀ v ∊ pts(x): pts(y) ⊇ pts(v)	pts(y) ⊇ pts(*x)

Steensgaard’s Algorithm

Inclusion-based pointer analysis (Andersen’s Algorithm)

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) \supseteq pts(x)$	
*y = x	$\forall\ v \in pts(y)$: $pts(v) \supseteq pts(x)$	$pts(*y) \supseteq pts(x)$
y = *x	$\forall\ v \in pts(x)$: $pts(y) \supseteq pts(v)$	$pts(y) \supseteq pts(*x)$

Unification-based pointer analysis (Steensgaard’s Algorithm)

Statement	Constraint	Shorthand
y = &x	$pts(y) \supseteq \{x\}$	
y = x	$pts(y) = pts(x)$	
*y = x	$\forall\ v \in pts(y)$: $pts(v) = pts(x)$	$pts(*y) = pts(x)$
y = *x	$\forall\ v \in pts(x)$: $pts(y) = pts(v)$	$pts(y) = pts(*x)$

南京大学编译技术及软件安全研究组

Steensgaard's Algorithm

Inclusion-based pointer analysis (Andersen's Algorithm)

Statement	Constraint	Shorthand
y = &x	pts(y) ⊇ {x}	
y = x	pts(y) ⊇ pts(x)	
*y = x	∀ v ∊ pts(y): pts(v) ⊇ pts(x)	pts(*y) ⊇ pts(x)
y = *x	∀ v ∊ pts(x): pts(y) ⊇ pts(v)	pts(y) ⊇ pts(*x)

Unification-based pointer analysis (Steensgaard's Algorithm)

Statement	Constraint	Shorthand
y = &x	pts(y) ⊇ {x}	
y = x	pts(y) = pts(x)	
*y = x	∀ v ∊ pts(y): pts(v) = pts(x)	pts(*y) = pts(x)
y = *x	∀ v ∊ pts(x): pts(y) = pts(v)	pts(y) = pts(*x)

Steensgaard’s Algorithm

- **Points-to graph**: x → y means y ∊ pts(x).

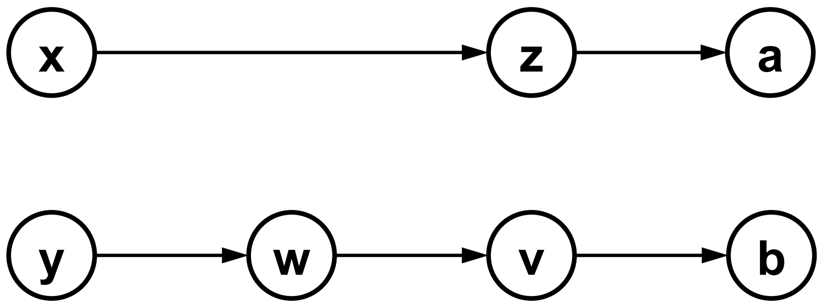
Steensgaard's Algorithm

- **Points-to graph**: x → y means y ∈ pts(x).

Steensgaard's algorithm: One node, one out-going edge!

Steensgaard's Algorithm

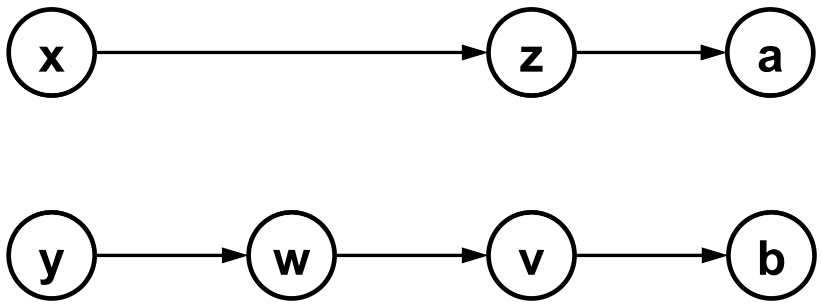
- **Points-to graph**: x → y means y ∈ pts(x).



Steensgaard's algorithm: One node, one out-going edge!

Steensgaard's Algorithm

- **Points-to graph**: x → y means y ∈ pts(x).

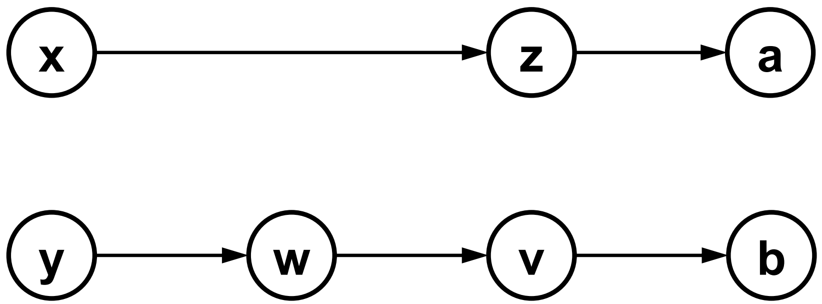


Steensgaard's algorithm: One node, one out-going edge!

x = *y

Steensgaard’s Algorithm

- **Points-to graph**: x → y means y ∈ pts(x).



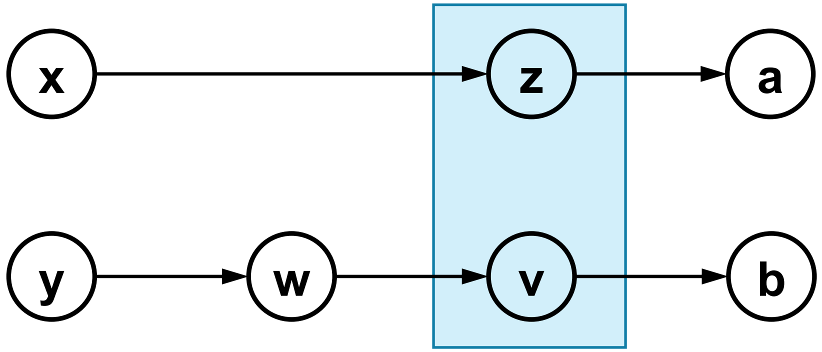
Steensgaard’s algorithm: One node, one out-going edge!

x = *y

pts(x) = pts(*y)

Steensgaard's Algorithm

- **Points-to graph**: x → y means y ∊ pts(x).



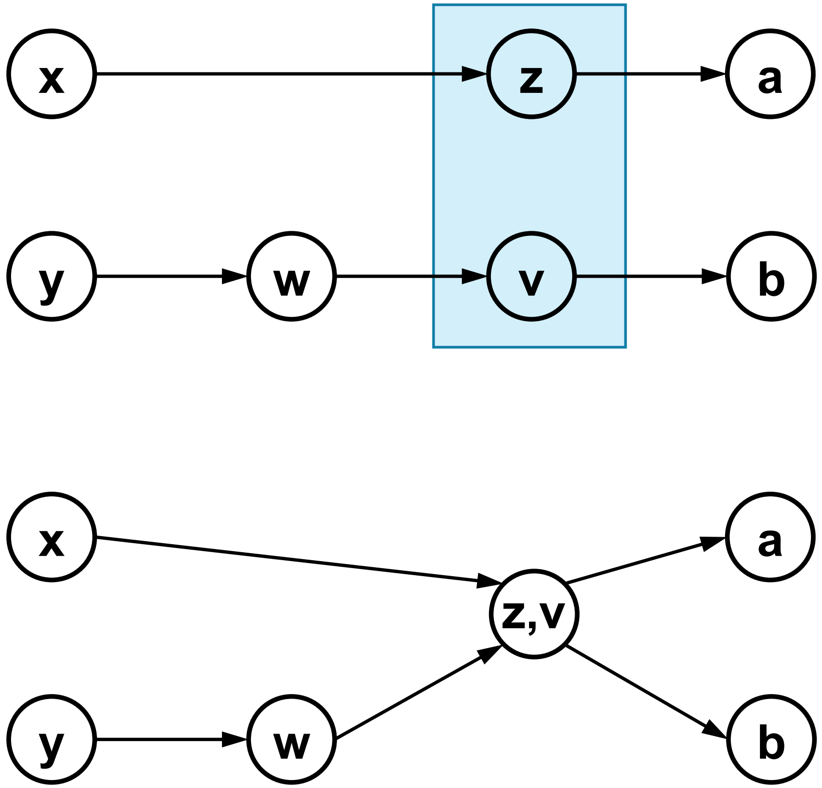
Steensgaard's algorithm: One node, one out-going edge!

x = *y

pts(x) = pts(*y)

Steensgaard's Algorithm

- **Points-to graph**: x → y means y ∊ pts(x).



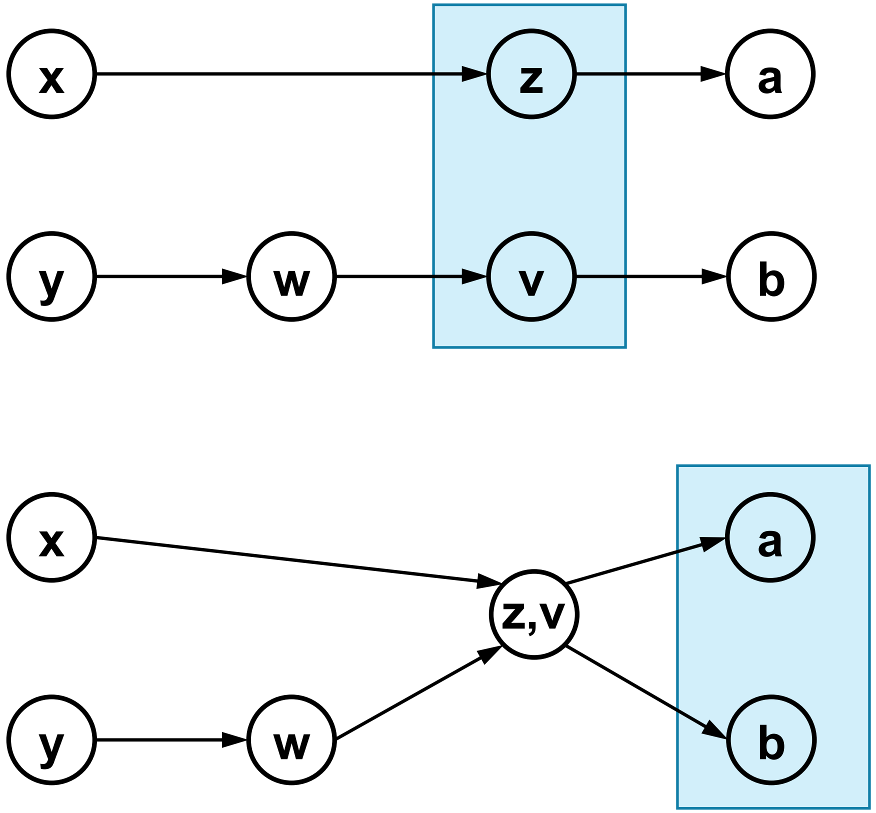
Steensgaard's algorithm: One node, one out-going edge!

x = *y

pts(x) = pts(*y)

Steensgaard's Algorithm

- **Points-to graph**: x → y means y ∈ pts(x).



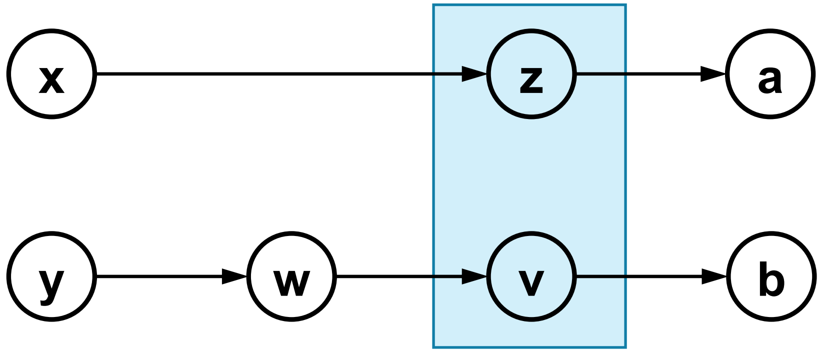
Steensgaard's algorithm: One node, one out-going edge!

⇦ x = *y

pts(x) = pts(*y)

Steensgaard's Algorithm

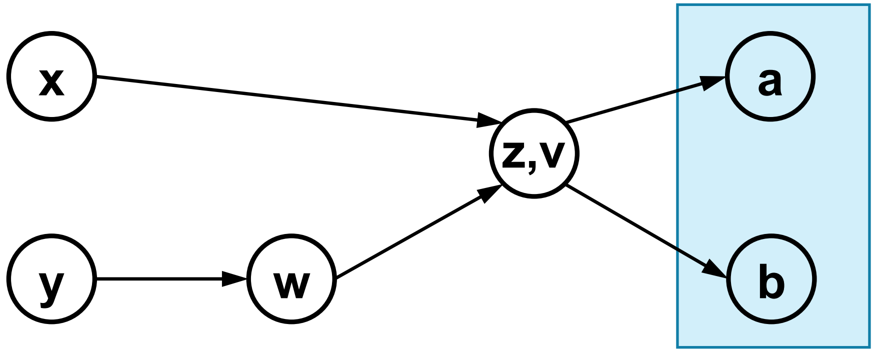
- **Points-to graph**: x → y means y ∊ pts(x).

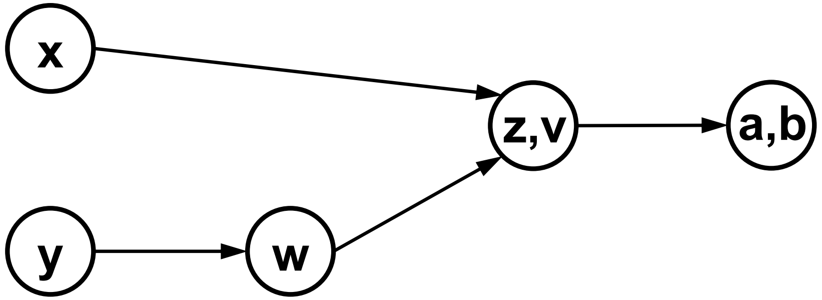


Steensgaard's algorithm: One node, one out-going edge!

x = *y

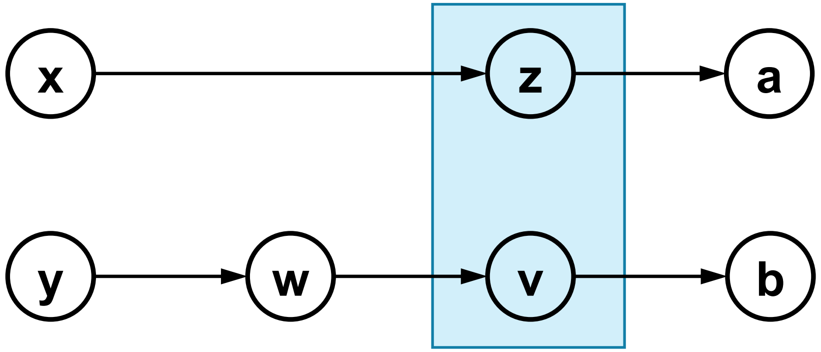
pts(x) = pts(*y)





Steensgaard’s Algorithm

- **Points-to graph**: x → y means y ∊ pts(x).

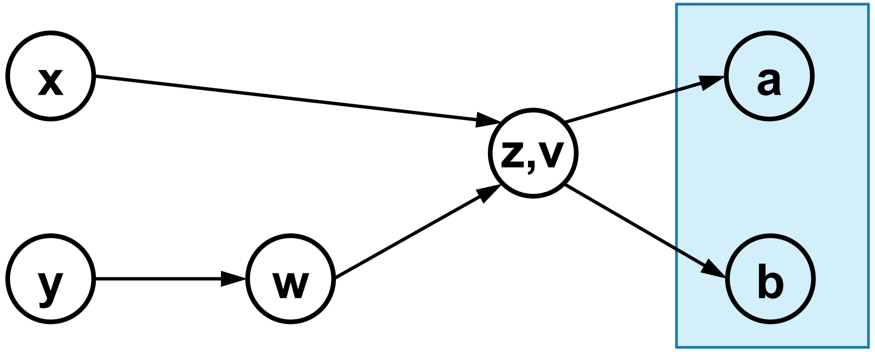


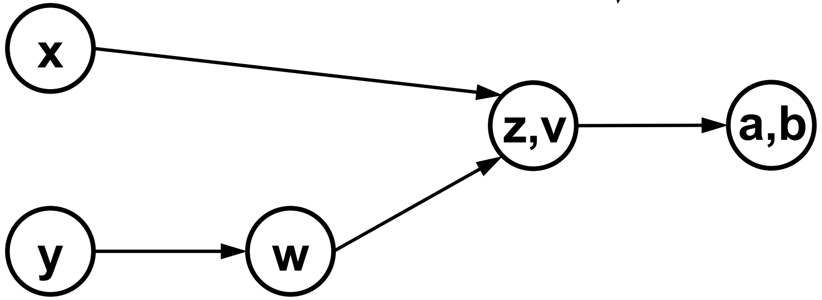
Steensgaard’s algorithm: One node, one out-going edge!

x = *y

pts(x) = pts(*y)

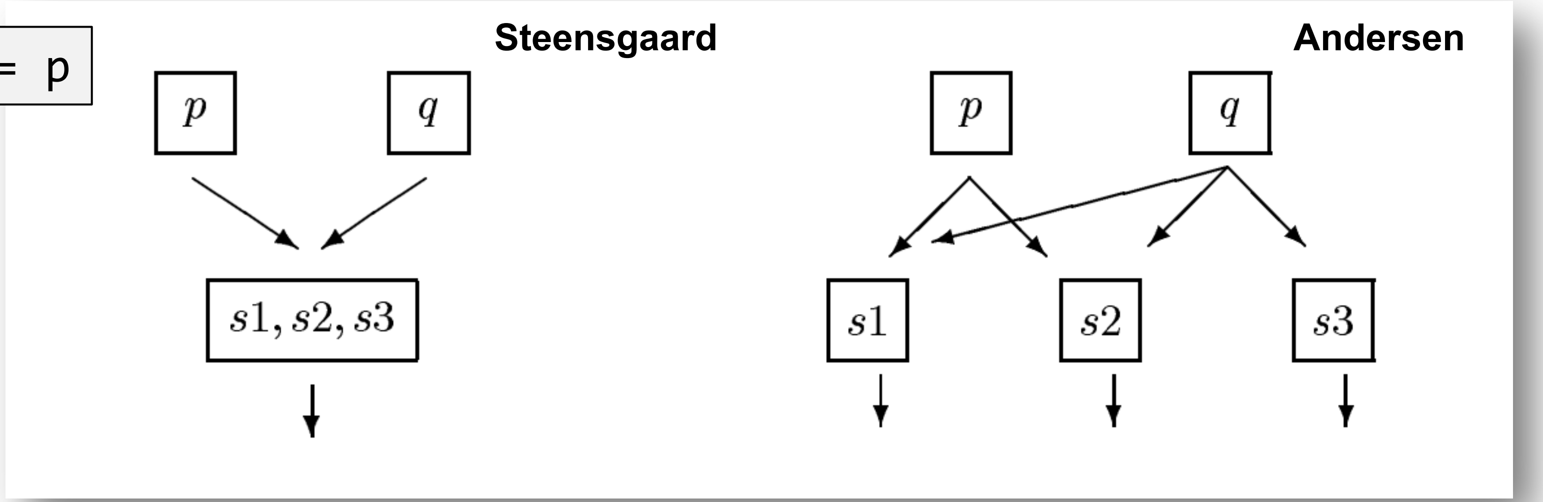
Disjoint-Set
Union-Find





Steensgaard’s Algorithm

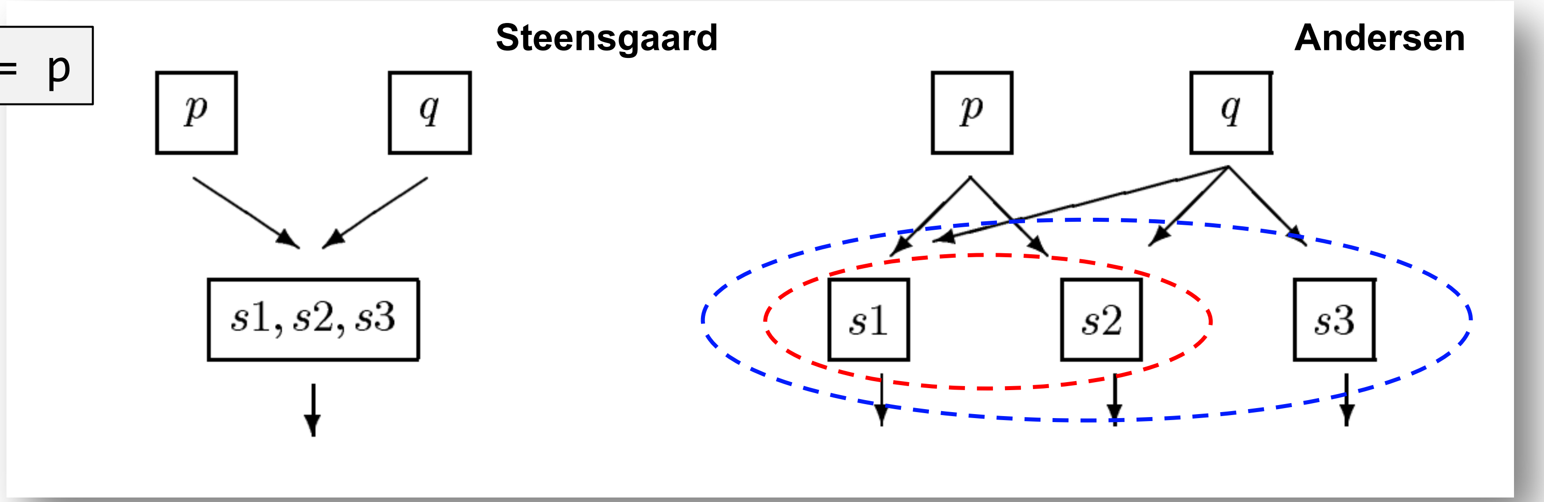
```
q = p
```



南京大学编译技术及软件安全研究组

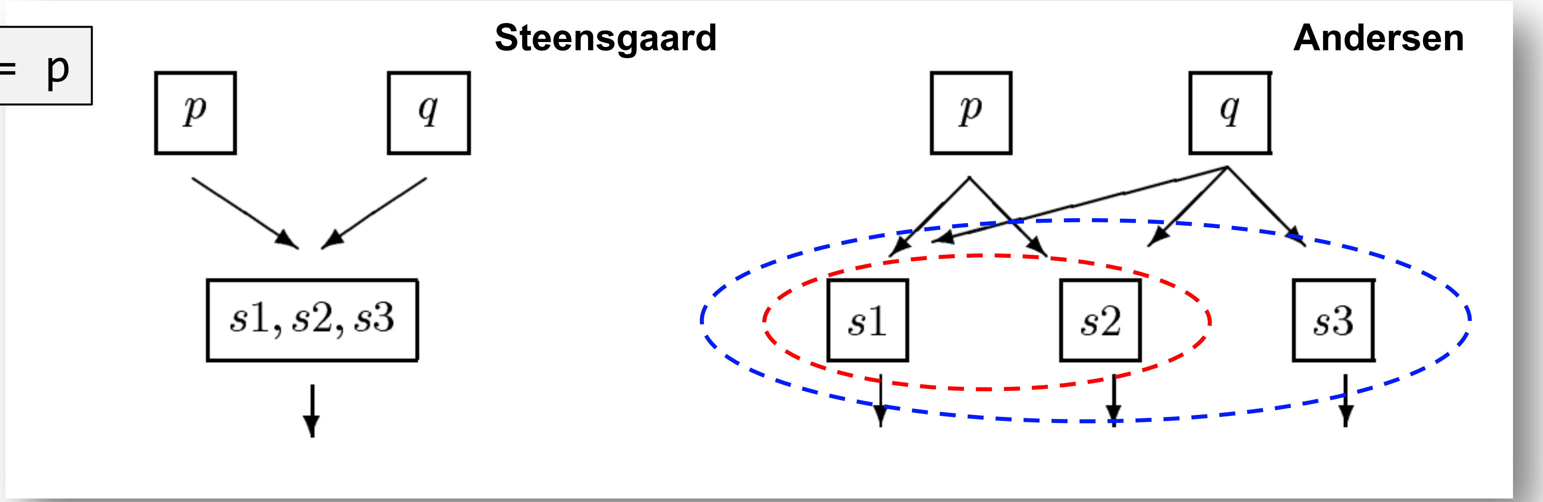
Steensgaard’s Algorithm

q = p



Steensgaard's Algorithm

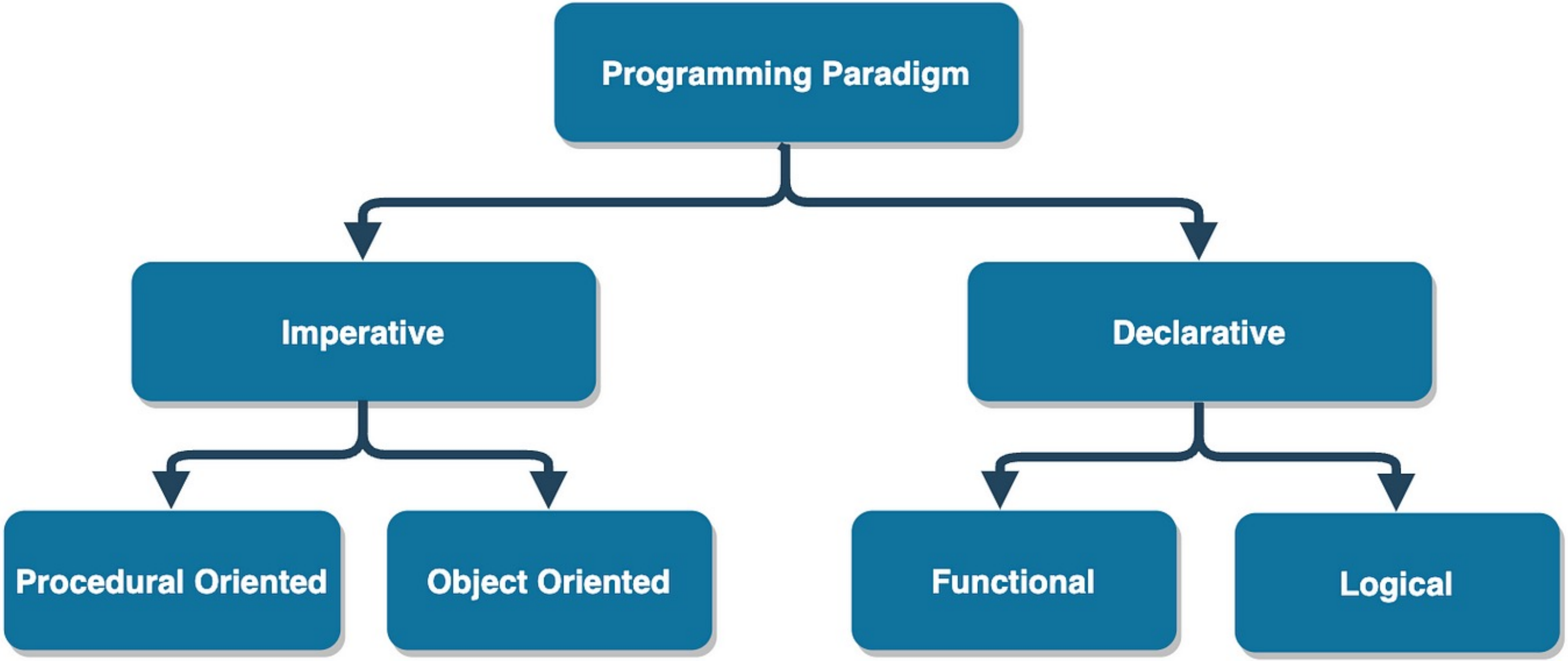
q = p



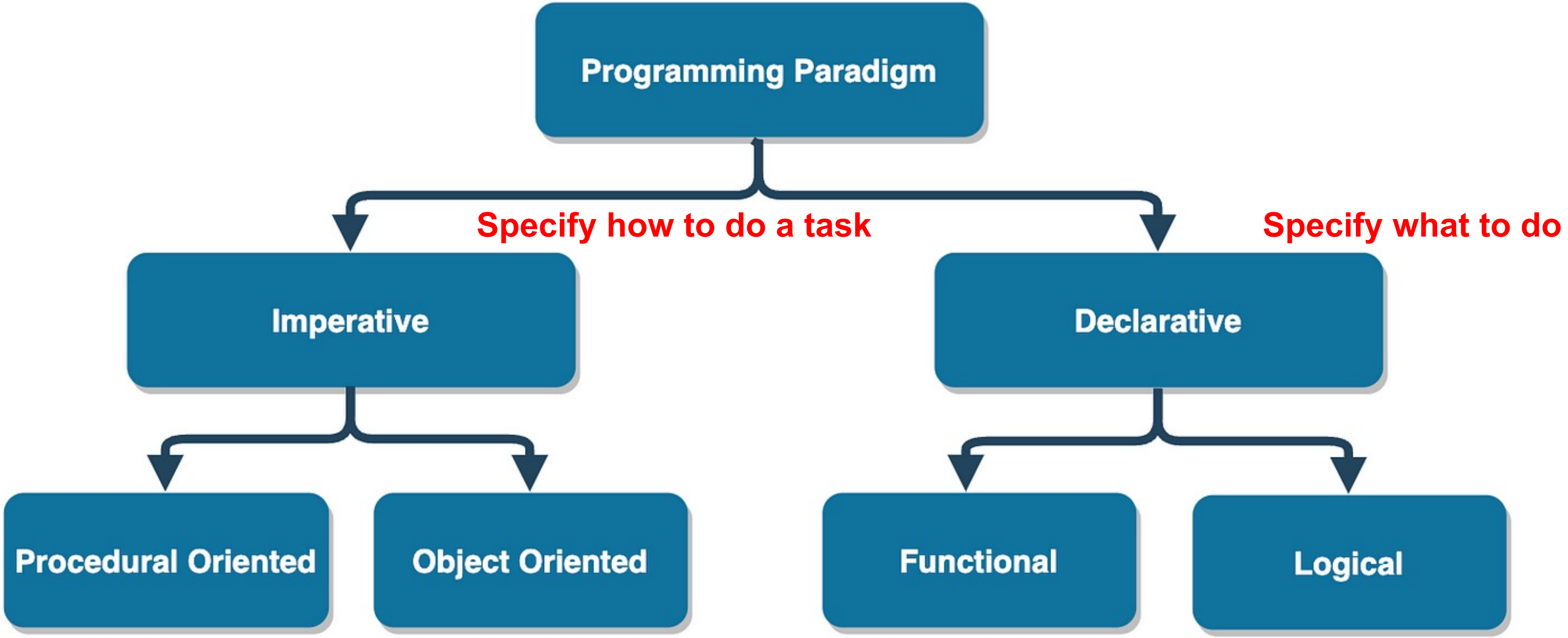
Steensgaard	Andersen
Sound	Sound
Almost linear (more efficient)	Nearly cubic
Unification-based (less precise)	Inclusion-based

PART IV: Datalog-Based DFA

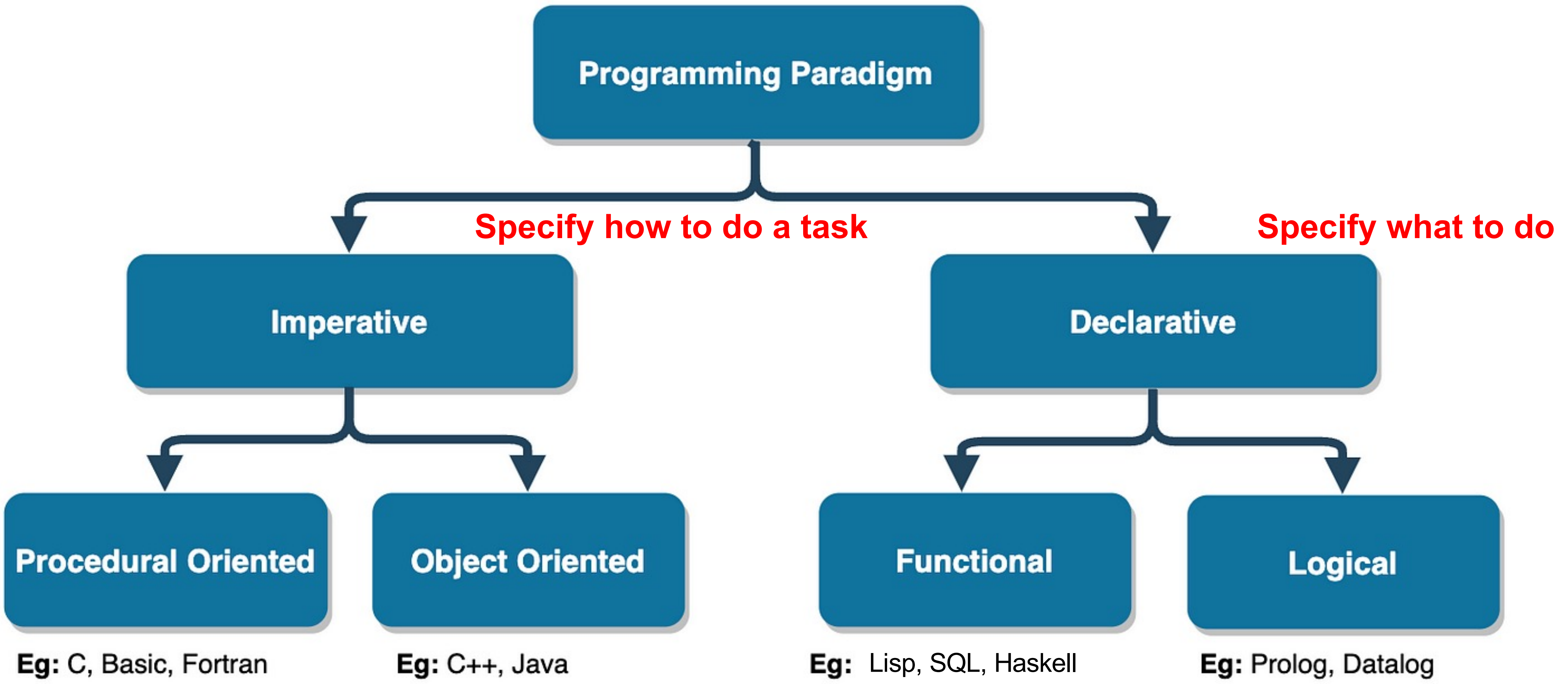
Recap: Programming Paradigm



Recap: Programming Paradigm



Recap: Programming Paradigm



Logic Programming

- Logic programming
  - In a broad sense: the use of mathematical logic for programming

Logic Programming

- Logic programming
  - In a broad sense: the use of mathematical logic for programming

- Prolog (1972)
  - Use logical rules to specify how mathematical relations are computed
  - A prolog program is a database of logical rules

Logic Programming

- Example:
  - Nanjing is rainy
  - Beijing is rainy
  - Beijing is cold
  - If a city is both rainy and cold, then it is snowy

Logic Programming

- Example:
  - Nanjing is rainy
  - Beijing is rainy
  - Beijing is cold
  - If a city is both rainy and cold, then it is snowy

Facts

Rules

Logic Programming

- Example:
  - Nanjing is rainy
  - Beijing is rainy
  - Beijing is cold
  - If a city is both rainy and cold, then it is snowy

Facts

Rules

  - **Query:** which city is snowy

Logic Programming

- Example:
  - Nanjing is rainy
  - Beijing is rainy
  - Beijing is cold
  - If a city is both rainy and cold, then it is snowy

Facts

Rules

  - **Query:** which city is snowy
  - Search for solutions based on facts and rules

Logic Programming

- Expert systems
- Natural language processing
- Theorem provers
- Reasoning about safety a security
- **Program analysis/Compiler optimization**

What is Datalog?

- Datalog is a subset of Prolog
  - All Datalog programs terminate
  - Ordering of rules does not matter
  - Not Turing complete

What is Datalog?

- Datalog is a subset of Prolog
  - All Datalog programs terminate
  - Ordering of rules does not matter
  - Not Turing complete

- Souffle Datalog engine --- https://souffle-lang.github.io/
  - .decl rainy (c:symbol)
  - .decl cold(c:symbol)
  - .decl snowy(c:symbol)
  - .output snowy

What is Datalog?

- Datalog is a subset of Prolog
  - All Datalog programs terminate
  - Ordering of rules does not matter
  - Not Turing complete

- Souffle Datalog engine --- https://souffle-lang.github.io/
  - .decl rainy (c:symbol)
  - .decl cold(c:symbol)
  - .decl snowy(c:symbol)
  - .output snowy
  - rainy(“Nanjing”)
  - rainy(“Beijing”)
  - cold(“Beijing”)
  - snowy(c) :- rainy(c), cold(c)

Predicate/Atom

- Predicates are N-ary relations
  - predicate(x, y, z)

- Examples
  - rainy(“Nanjing”)
  - rainy(“Beijing”)
  - cold(“Beijing”)

Predicate/Atom

- Predicates are N-ary relations
  - predicate(x, y, z)

- Examples
  - rainy(“Nanjing”)
  - rainy(“Beijing”)
  - cold(“Beijing”)
  - brother(x, y) -- x is y’s brother
  - speaks(x, a) -- x speaks language a

Datalog Programming Model

- A Datalog program is a database of Horn clauses
  - h is true **if** the assumptions l_1 l_2 … l_n are simultaneously true
  - **if** --- a sufficient but not necessary condition

$$h \ :\text{-}\ l_1\ l_2\ \ldots\ l_n$$

Datalog Programming Model

- A Datalog program is a database of Horn clauses
  - h is true **if** the assumptions l_1 l_2 … l_n are simultaneously true
  - **if** --- a sufficient but not necessary condition

```
h :- l1 l2 … ln
```

- **Example:**
  - rainy("Nanjing")
  - snowy(c) :- rainy(c), cold(c)

Datalog Programming Model

- All rules hold for any instantiation of its variables
  - snowy(c) :- rainy(c), cold(c) for all cities like Nanjing, Beijing, etc.…

Datalog Programming Model

- All rules hold for any instantiation of its variables
  - snowy(c) :- rainy(c), cold(c) for all cities like Nanjing, Beijing, etc.…

- Anything not declared is not true

Datalog Programming Model

- All rules hold for any instantiation of its variables
  - snowy(c) :- rainy(c), cold(c) for all cities like Nanjing, Beijing, etc.…
- Anything not declared is not true
- Ordering of rules does not matter for results (Prolog vs. Datalog)

Datalog Programming Model

- All rules hold for any instantiation of its variables
  - snowy(c) :- rainy(c), cold(c) for all cities like Nanjing, Beijing, etc.…
- Anything not declared is not true
- Ordering of rules does not matter for results (Prolog vs. Datalog)
- Rules may be recursive
  - reachable(a, b) :- edge(a, b);
  - reachable(a, c) :- edge(a, b), reachable(b, c)

Datalog Programming Model

- Negation is allowed in assumptions
  - more_than_one_hop(a, b) :- reachable(a, b), ¬edge(a, b)

Datalog Programming Model

- Negation is allowed in assumptions
  - more_than_one_hop(a, b) :- reachable(a, b), ¬edge(a, b)
- All rules must be well-formed. Example of bad rules:
  - more_than_one_hop(a, b) :- ¬edge(a, b)

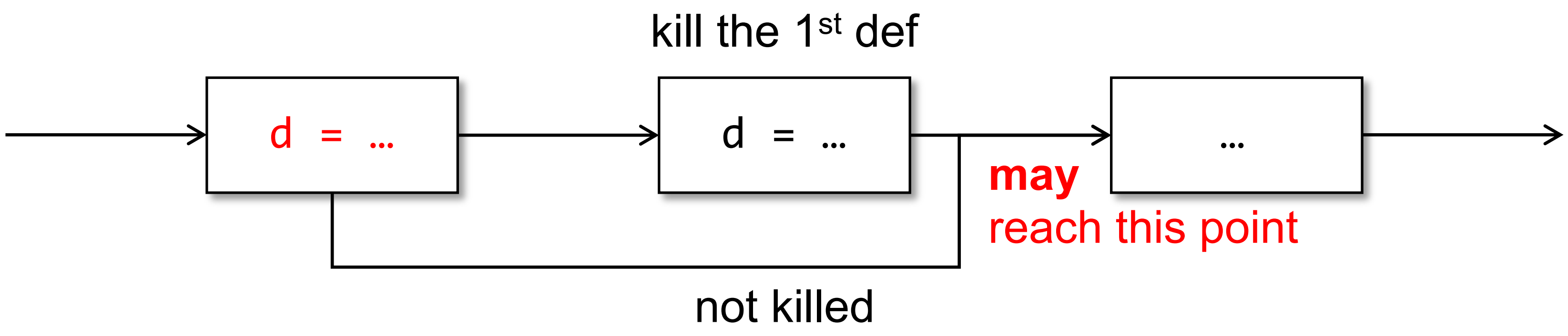
Datalog Programming Model

- Negation is allowed in assumptions
  - more_than_one_hop(a, b) :- reachable(a, b), ¬edge(a, b)
- All rules must be well-formed. Example of bad rules:
  - more_than_one_hop(a, b) :- ¬edge(a, b)
  - **Bad:** a, b on the left do not appear in the positive predicate on the right
  - **Goal:** to ensure termination

Reaching Definitions by Datalog

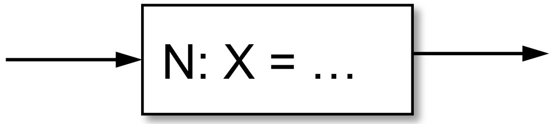
Recap: Reaching Definition

- A definition *d* may reach a program point, if there is a path from *d* to the program point such that *d* is not killed along the path.



Reaching Definitions by Datalog

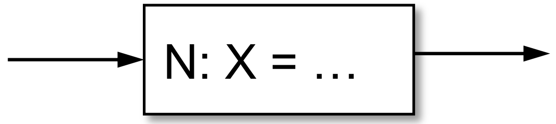
- def(B, N, X)
  - the Nth statement in Block B may define the variable X.



Block B

Reaching Definitions by Datalog

- def(B, N, X)
  - the Nth statement in Block B may define the variable X.

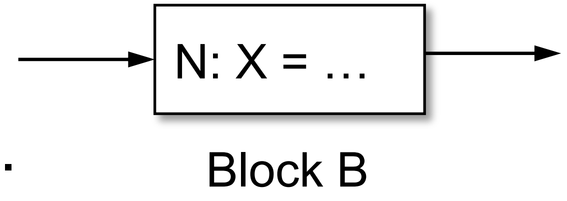


Block B

- succ(B, N, C)
  - block C is a successor of block B, and B has N statements.

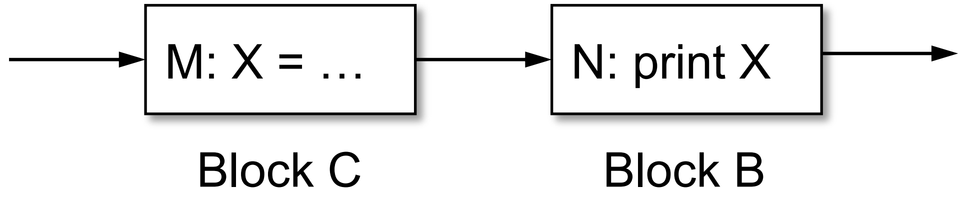
Reaching Definitions by Datalog

- def(B, N, X)
  - the Nth statement in Block B may define the variable X.



- succ(B, N, C)
  - block C is a successor of block B, and B has N statements.

- rd(B, N, C, M, X)
  - the definition of variable X at the Mth statement of block C reaches the Nth statement in B.

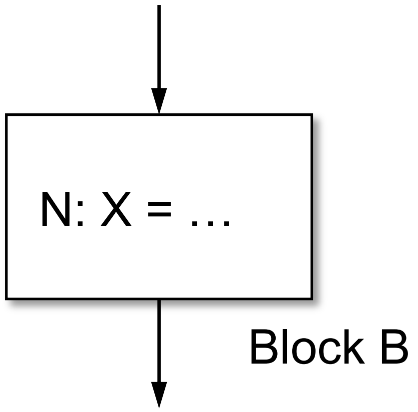


Reaching Definitions by Datalog

- rd(B, N, B, N, X) :- def(B, N, X)
- rd(B, N, C, M, X) :- rd(B, N-1, C, M, X), def(B, N, Y), X≠Y
- rd(B, 0, C, M, X) :- rd(D, N, C, M, X), succ(D, N, B)

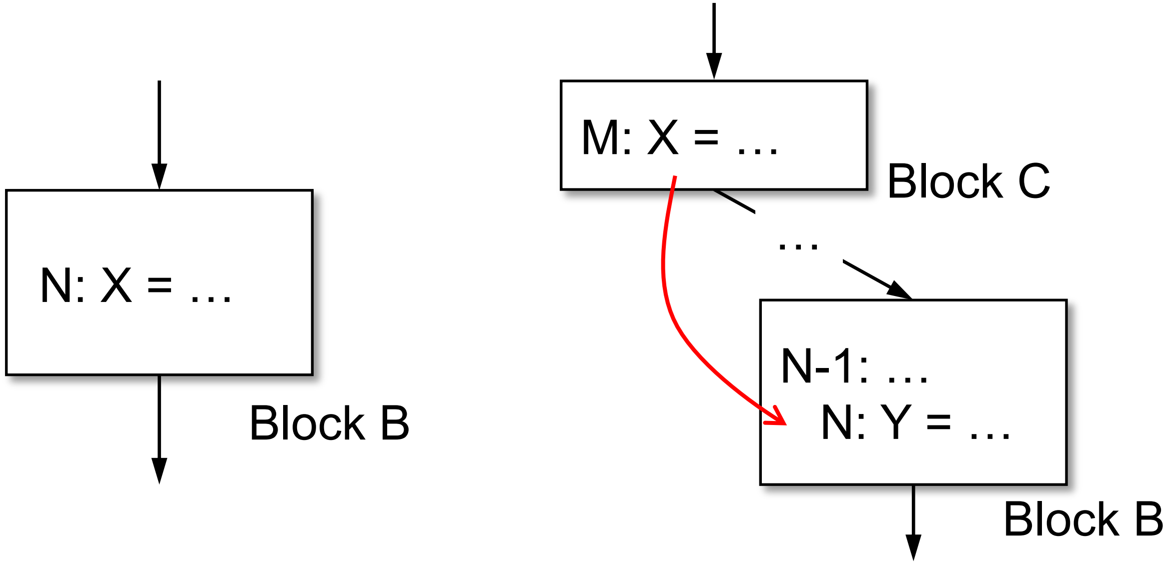
Reaching Definitions by Datalog

- rd(B, N, B, N, X) :- def(B, N, X)
- rd(B, N, C, M, X) :- rd(B, N-1, C, M, X), def(B, N, Y), X≠Y
- rd(B, 0, C, M, X) :- rd(D, N, C, M, X), succ(D, N, B)



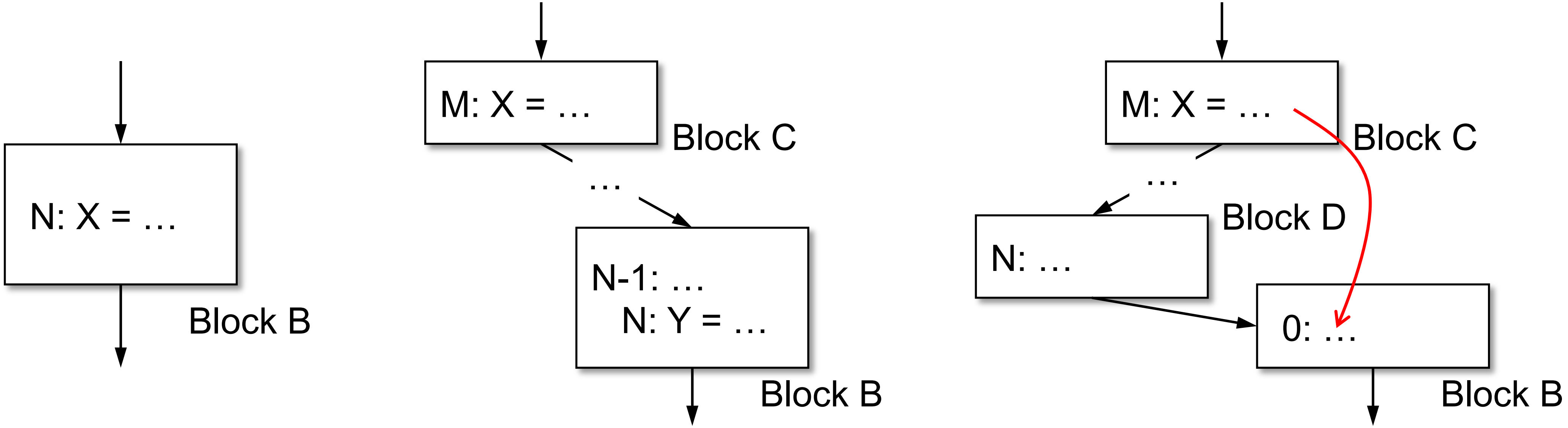
Reaching Definitions by Datalog

- rd(B, N, B, N, X) :- def(B, N, X)
- rd(B, N, C, M, X) :- rd(B, N-1, C, M, X), def(B, N, Y), X≠Y
- rd(B, 0, C, M, X) :- rd(D, N, C, M, X), succ(D, N, B)

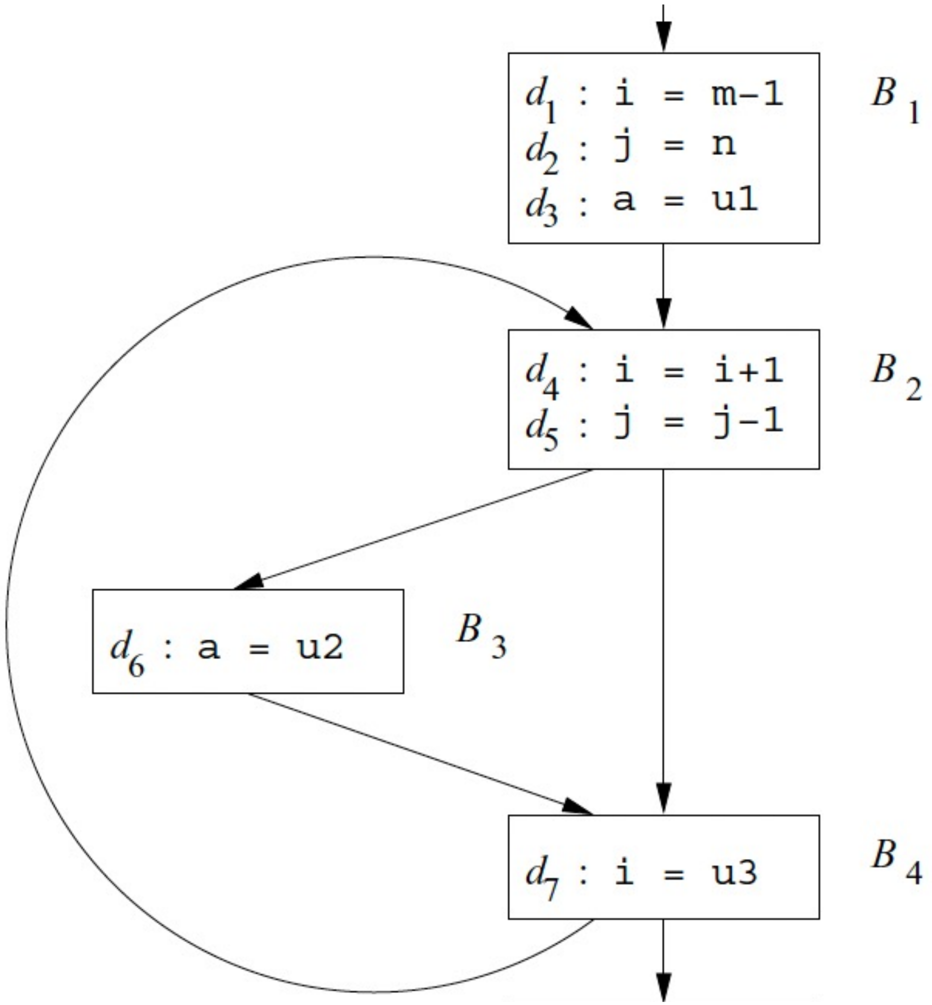


Reaching Definitions by Datalog

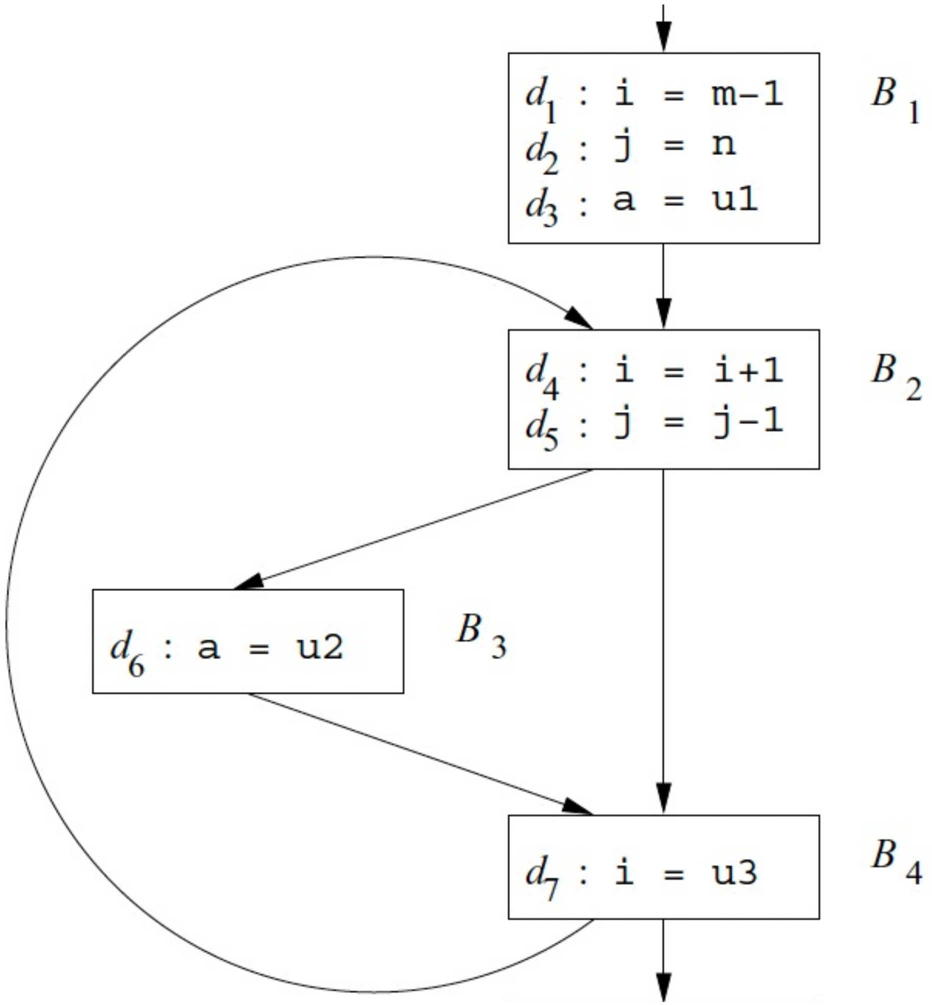
- rd(B, N, B, N, X) :- def(B, N, X)
- rd(B, N, C, M, X) :- rd(B, N-1, C, M, X), def(B, N, Y), X≠Y
- rd(B, 0, C, M, X) :- rd(D, N, C, M, X), succ(D, N, B)



Reaching Definitions by Datalog

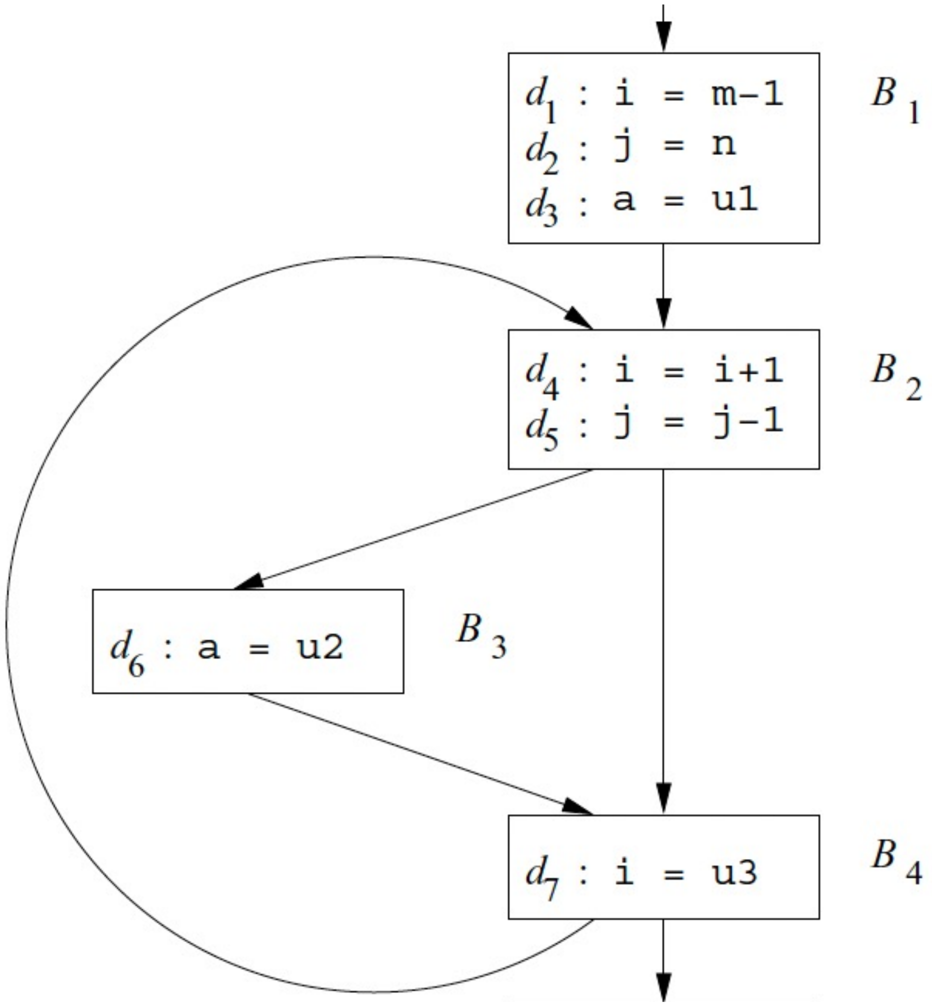


Reaching Definitions by Datalog



- def(B_1, 1, i)
- def(B_1, 2, j)
- def(B_1, 3, a)
- def(B_2, 1, i)
- def(B_2, 2, j)
- def(B_3, 1, a)
- def(B_4, 1, i)

Reaching Definitions by Datalog



- def(B_1, 1, i)
- def(B_1, 2, j)
- def(B_1, 3, a)
- def(B_2, 1, i)
- def(B_2, 2, j)
- def(B_3, 1, a)
- def(B_4, 1, i)

- succ(B_1, 3, B_2)
- succ(B_2, 2, B_3)
- succ(B_2, 2, B_4)
- succ(B_3, 1, B_4)
- succ(B_4, 1, B_2)

Reaching Definitions by Datalog

- rd(B, N, B, N, X) :- def(B, N, X)
- rd(B, N, C, M, X) :- rd(B, N-1, C, M, X), def(B, N, Y), X≠Y
- rd(B, 0, C, M, X) :- rd(D, N, C, M, X), succ(D, N, B)

- def(B_1, 1, i)
- def(B_1, 2, j)
- def(B_1, 3, a)
- def(B_2, 1, i)
- def(B_2, 2, j)
- def(B_3, 1, a)
- def(B_4, 1, i)
- succ(B_1, 3, B_2)
- succ(B_2, 2, B_3)
- succ(B_2, 2, B_4)
- succ(B_3, 1, B_4)
- succ(B_4, 1, B_2)

Reaching Definitions by Datalog

- rd(B, N, B, N, X) :- def(B, N, X)
- rd(B
- rd(B

We just define facts and rules.
Analysis is automatically done by Datalog engines!

- def(B_1, 1, i)
- def(B_1, 2, j)
- def(B_1, 3, a)
- def(B_2, 1, i)
- def(B_2, 2, j)
- def(B_3, 1, a)
- def(B_4, 1, i)
- succ(B_1, 3, B_2)
- succ(B_2, 2, B_3)
- succ(B_2, 2, B_4)
- succ(B_3, 1, B_4)
- succ(B_4, 1, B_2)

Reaching Definitions by Datalog

- rd(B, N, B, N, X) :- def(B, N, X)

We just define facts and rules.
Analysis is automatically done by Datalog engines!

Query Example: rd(B_4, 1, B_1, 1, i)

- def(B_1, 1, i)
- def(B_1, 2, j)
- def(B_1, 3, a)
- def(B_2, 1, i)
- def(B_3, 1, a)
- def(B_4, 1, i)
- succ(B_1, 3, B_2)
- succ(B_2, 2, B_4)
- succ(B_3, 1, B_4)
- succ(B_4, 1, B_2)

Pointer Analysis by Datalog

Pointer Analysis by Datalog

- Four kinds of Java statements and ignore procedural call
- **Object creation.** h: T v = **new** T();
- **Copy.** v = w;
- **Field store.** v.f = w;
- **Field load.** v = w.f;

Pointer Analysis by Datalog

- Four kinds of Java statements and ignore procedural call
- **Object creation.** h: T v = **new** T();
- **Copy.** v = w;
- **Field store.** v.f = w;
- **Field load.** v = w.f;

- pts(V, H) variable V can point to a heap object H
- hpts(H, F, G) field F of heap object H can point to heap obj G

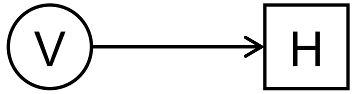
Pointer Analysis by Datalog

$$
\begin{array}{lrll}
1) & pts(V,H) & \texttt{:-} & \text{“}H:\ T\ V\ = \texttt{new}\ T\text{”} \\
2) & pts(V,H) & \texttt{:-} & \text{“}V = W\text{”}\ \& \\
 & & & pts(W,H) \\
3) & hpts(H,F,G) & \texttt{:-} & \text{“}V.F = W\text{”}\ \& \\
 & & & pts(W,G)\ \& \\
 & & & pts(V,H) \\
4) & pts(V,H) & \texttt{:-} & \text{“}V = W.F\text{”}\ \& \\
 & & & pts(W,G)\ \& \\
 & & & hpts(G,F,H)
\end{array}
$$

Pointer Analysis by Datalog

$$
\begin{array}{llll}
1) & pts(V,H) & \texttt{:-} & \text{“}H:\ T\ V\ = \texttt{new}\ T\text{”} \\
2) & pts(V,H) & \texttt{:-} & \text{“}V = W\text{” \&} \\
 & & & pts(W,H) \\
3) & hpts(H,F,G) & \texttt{:-} & \text{“}V.F = W\text{” \&} \\
 & & & pts(W,G)\ \text{\&} \\
 & & & pts(V,H) \\
4) & pts(V,H) & \texttt{:-} & \text{“}V = W.F\text{” \&} \\
 & & & pts(W,G)\ \text{\&} \\
 & & & hpts(G,F,H)
\end{array}
$$

H: T V = new T



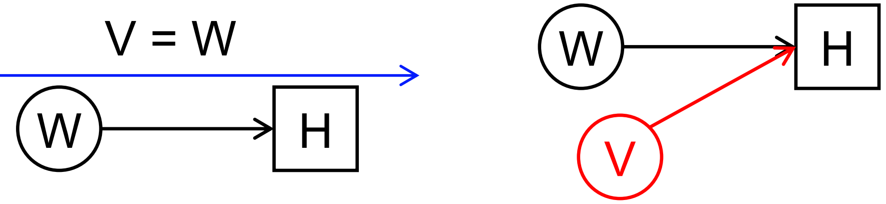
Pointer Analysis by Datalog

1) $pts(V, H) \quad \text{:-} \quad \text{“}H: T\ V = \texttt{new}\ T\text{”}$

2) $pts(V, H) \quad \text{:-} \quad \text{“}V = W\text{”}\ \&\ pts(W, H)$

3) $hpts(H, F, G) \quad \text{:-} \quad \text{“}V.F = W\text{”}\ \&\ pts(W, G)\ \&\ pts(V, H)$

4) $pts(V, H) \quad \text{:-} \quad \text{“}V = W.F\text{”}\ \&\ pts(W, G)\ \&\ hpts(G, F, H)$



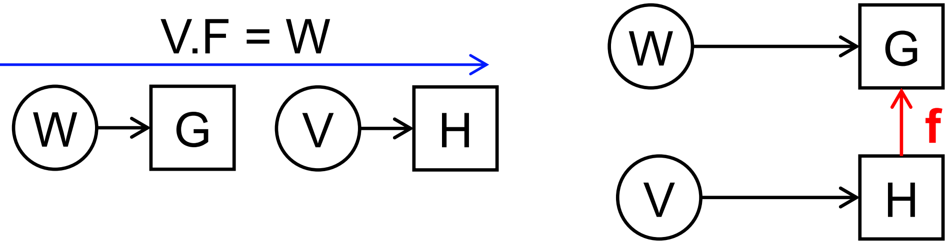
Pointer Analysis by Datalog

1) $pts(V, H)$:- "$H : T\ V = \texttt{new}\ T$"

2) $pts(V, H)$:- "$V = W$" & $pts(W, H)$

3) $hpts(H, F, G)$:- "$V.F = W$" & $pts(W, G)$ & $pts(V, H)$

4) $pts(V, H)$:- "$V = W.F$" & $pts(W, G)$ & $hpts(G, F, H)$



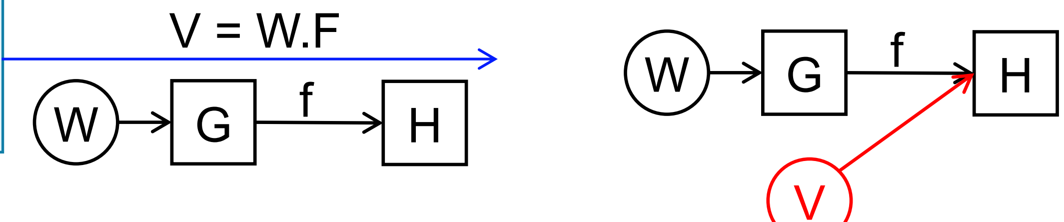
Pointer Analysis by Datalog

1) $pts(V, H) \quad :- \quad$ "$H: T\ V = \texttt{new}\ T$"

2) $pts(V, H) \quad :- \quad$ "$V = W$" & $pts(W, H)$

3) $hpts(H, F, G) \quad :- \quad$ "$V.F = W$" & $pts(W, G)$ & $pts(V, H)$

4) $pts(V, H) \quad :- \quad$ "$V = W.F$" & $pts(W, G)$ & $hpts(G, F, H)$



Interprocedural Pointer Analysis

- Dealing with method invocation in Java programs
- x = y.n(z)

Interprocedural Pointer Analysis

- Dealing with method invocation in Java programs
- x = y.n(z)

- actual(S, I, V): V is the I^{th} argument in call site S
- formal(M, I, V): V is the I^{th} parameter in method M

Interprocedural Pointer Analysis

- Dealing with method invocation in Java programs
- x = y.n(z)

- actual(S, I, V): V is the Ith argument in call site S
- formal(M, I, V): V is the Ith parameter in method M

$$1) \quad invokes(S, N) \quad :- \quad \text{“}S : \ V.N(...)\text{”}$$

Interprocedural Pointer Analysis

- Dealing with method invocation in Java programs
- x = y.n(z)

- actual(S, I, V): V is the I^{th} argument in call site S
- formal(M, I, V): V is the I^{th} parameter in method M

1) $invokes(S, N) \quad :- \quad \text{“}S: \ V.N(...)\text{”}$

2) $pts(V, H) \quad :- \quad invokes(S, M) \ \& \ formal(M, I, V) \ \& \ actual(S, I, W) \ \& \ pts(W, H)$

Interprocedural Pointer Analysis

- Dealing with method invocation in Java programs
- x = y.n(z)

- actual(S, I, V): V is the I^{th} argument in call site S
- formal(M, I, V): V is the I^{th} parameter in method M

$$1)\quad invokes(S, N) \quad :- \quad \text{“}S:\ V.N(...)\text{”}$$

$$2)\quad pts(V, H) \quad :- \quad invokes(S, M)\ \&\ formal(M, I, V)\ \&\ actual(S, I, W)\ \&\ pts(W, H)$$

Just follow the rule for assignment V = W

Interprocedural Pointer Analysis

- Dealing with method invocation in Java programs
- x = y.n(z)

- actual(S, I, V): V is the I^th argument in call site S
- formal(M, I, V): V is the I^th parameter in method M

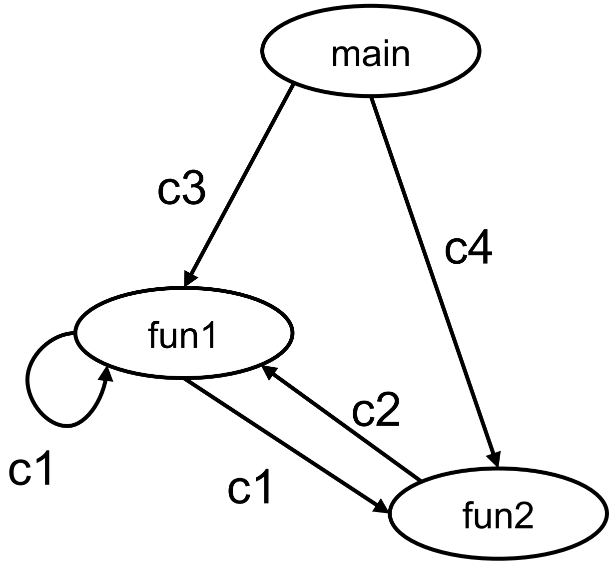
$$1)\quad invokes(S, N) \quad :- \quad \text{“}S : V.N(...)\text{”}$$

$$2)\quad pts(V, H) \quad :- \quad invokes(S, M)\ \&\ formal(M, I, V)\ \&\ actual(S, I, W)\ \&\ pts(W, H)$$

Just follow the rule for assignment V = W

It is **context-insensitive** as we always do the same thing when calling a function, i.e., do not distinguish different call sites of the same function

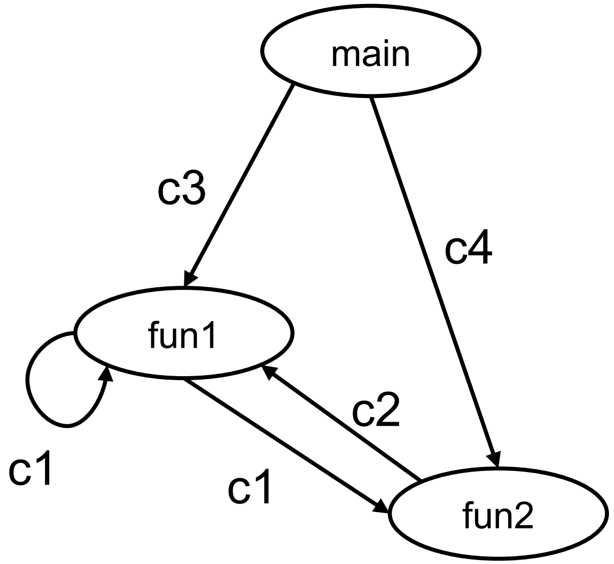
Context-Sensitive Pointer Analysis



Call chains:
- c_3-c_1-…
- c_4-c_2-c_1-…
- …

- Assume we already have the call graph.
- Calling context:
  - call chains, paths in the graph

Context-Sensitive Pointer Analysis

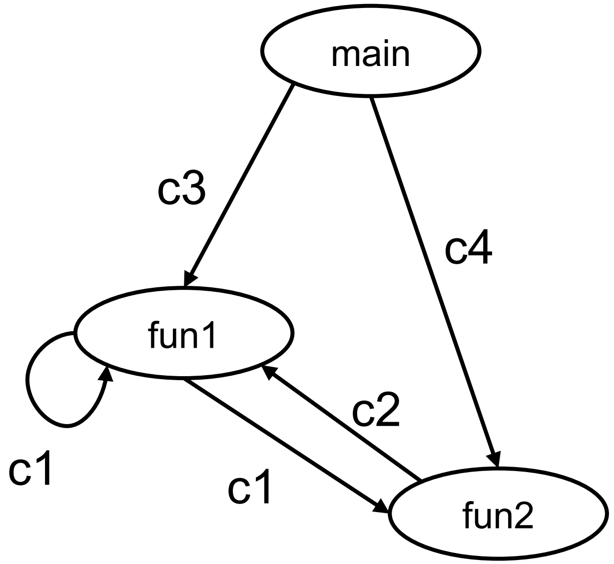


Call chains:

- c_3-c_1-…
- c_4-c_2-c_1-…
- …

- Assume we already have the call graph.
- Calling context:
  - call chains, paths in the graph
- Context-sensitive analysis:
  - analyze functions in different calling context

Context-Sensitive Pointer Analysis



Call chains:
- c_3-c_1-…
- c_4-c_2-c_1-…
- …

- Assume we already have the call graph.
- Calling context:
  - call chains, paths in the graph
- Context-sensitive analysis:
  - analyze functions in different calling context
- **Difficulty**: Infinite # calling contexts
- **Solution**: restrict the length of call chains

Context-Sensitive Pointer Analysis

- invokes(S, **C**, M, **D**)
- call site S in context C, invokes the method M of context D

Context-Sensitive Pointer Analysis

- invokes(S, C, M, D)
- call site S in context C, invokes the method M of context D

call chains of length ≤ a predefined value

Context-Sensitive Pointer Analysis

- invokes(S, $\boxed{\underline{\textbf{C}}}$, M, $\boxed{\underline{\textbf{D}}}$)
- call site S in context C, invokes the method M of context D

call chains of length ≤ a predefined value

- pts(V, **<u>C</u>**, H)
- V points to H in context C

南京大学编译技术及软件安全研究组

Context-Sensitive Pointer Analysis

- invokes(S, C, M, D)
- call site S in context C, invokes the method M of context D

call chains of length ≤ a predefined value

- pts(V, C, H)
- V points to H in context C

- hpts(H, F, G)
- same as before, field F of H points to G

Context-Sensitive Pointer Analysis

	Context-Insensitive	Context-Sensitive
1	pts(V, H) :- "H : T V = new T()"	pts(V, **C,** H) :- "H : T V = new T()", **invoke(H, C, _, _)**

Context-Sensitive Pointer Analysis

	Context-Insensitive	Context-Sensitive
1	pts(V, H) :- "H : T V = new T()"	pts(V, **C,** H) :- "H : T V = new T()", **invoke(H, C, _, _)**
2	pts(V, H) :- "V = W", pts(W, H)	pts(V, **C,** H) :- "V = W", pts(W, **C,** H)

Context-Sensitive Pointer Analysis

	Context-Insensitive	Context-Sensitive
1	pts(V, H) :- "H : T V = new T()"	pts(V, **C,** H) :- "H : T V = new T()", **invoke(H, C, _, _)**
2	pts(V, H) :- "V = W", pts(W, H)	pts(V, **C,** H) :- "V = W", pts(W, **C,** H)
3	hpts(H, F, G) :- "V.F = W", pts(W, G), pts(V, H)	hpts(H, F, G) :- "V.F = W", pts(W, **C,** G), pts(V, **C,** H)

Context-Sensitive Pointer Analysis

	Context-Insensitive	Context-Sensitive
1	pts(V, H) :- "H : T V = new T()"	pts(V, **C,** H) :- "H : T V = new T()", **invoke(H, C, _, _)**
2	pts(V, H) :- "V = W", pts(W, H)	pts(V, **C,** H) :- "V = W", pts(W, **C,** H)
3	hpts(H, F, G) :- "V.F = W", pts(W, G), pts(V, H)	hpts(H, F, G) :- "V.F = W", pts(W, **C,** G), pts(V, **C,** H)
4	pts(V, H) :- "V = W.F", pts(W, G), hpts(G, F, H)	pts(V, **C,** H) :- "V = W.F", pts(W, **C,** G), hpts(G, F, H)

Context-Sensitive Pointer Analysis

	Context-Insensitive	Context-Sensitive
1	pts(V, H) :- "H : T V = new T()"	pts(V, C, H) :- "H : T V = new T()", invoke(H, C, _, _)
2	pts(V, H) :- "V = W", pts(W, H)	pts(V, C, H) :- "V = W", pts(W, C, H)
3	hpts(H, F, G) :- "V.F = W", pts(W, G), pts(V, H)	hpts(H, F, G) :- "V.F = W", pts(W, C, G), pts(V, C, H)
4	pts(V, H) :- "V = W.F", pts(W, G), hpts(G, F, H)	pts(V, C, H) :- "V = W.F", pts(W, C, G), hpts(G, F, H)
5	pts(V, H) :- invokes(S, M), formal(M, I, V), actual(S, I, W), pts(W, H)	pts(V, D, H) :- invokes(S, C, M, D), formal(M, I, V), actual(S, I, W), pts(W, C, H)

PART V: Object Sensitivity

Recap: Context Sensitivity

```
    class List {
        int x;
        void foo() {
c1          print(x);
        }
    }


    class A {
       void bar(List a) {
c2         a.foo();
c3         a.foo();
       }
       …
    }
```

Recap: Context Sensitivity

```
    class List {
        int x;
        void foo() {
c1          print(x);
        }
    }


    class A {
        void bar(List a) {
c2          a.foo();
c3          a.foo();
        }
        …
    }
```

A context-sensitive analysis analyzes the call chains $\mathbf{c_2c_1}$; $\mathbf{c_3c_1}$ separately to distinguish the calling context.

Object Sensitivity

```
    class List {
        int x;
        void foo() {
c1          print(x);
        }
    }


    class A {
        void bar(List a) {
c2          a.foo();
c3          a.foo();
        }
        …
    }
```

A new form of context-sensitivity, especially for **OO programs**. It does not distinguish contexts based on call sites.

Object Sensitivity

```
     class List {
         int x;
         void foo() {
c1           print(x);
         }
     }


     class A {
        void bar(List a) {
c2           a.foo();
c3           a.foo();
        }
        …
     }
```

A new form of context-sensitivity, especially for **OO programs**. It does not distinguish contexts based on call sites.

An object-sensitive analysis will not separately analyze $\mathbf{c_2c_1}$ and $\mathbf{c_3c_1}$, because they use the same object a.

No Silver Bullet

- Object-sensitivity is a tradeoff between precision and efficiency.

Ana Milanova, Atanas Rountev, Barbara Ryder.
Parameterized object sensitivity for points-to analysis for Java.
ACM Transactions on Software Engineering and Methodology, 14(1), 2005.
https://doi.org/10.1145/1044834.1044835

No Silver Bullet

- Object-sensitivity is a tradeoff between precision and efficiency.

Ana Milanova, Atanas Rountev, Barbara Ryder.
Parameterized object sensitivity for points-to analysis for Java.
ACM Transactions on Software Engineering and Methodology, 14(1), 2005.
https://doi.org/10.1145/1044834.1044835

Rice Theorem: It is impossible to decide a property of programs, which depends only on the semantics and not on the syntax, unless the property is trivial (true of all programs, or false of all programs).

THANKS!