Pointer Analysis

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2017

Outline

- Introduction
- Pointer analysis as a DFA problem
- Design decisions
- Andersen's analysis, Steensgaard's analysis
- Pointer analysis as a graph problem
  - Optimizations
- Pointer analysis as graph rewrite rules
- Applications
- Parallelization
  - Constraint based
  - Replication based

What is Pointer Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

What is Points-to Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

a points to x

What is Points-to Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

a points to x

a and b are aliases

What is Points-to Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

a points to x

a and b are aliases

Is this condition always satisfied?

What is Points-to Analysis?

```
a = &x;
b = a;
if (b == *p) {
    …
} else {
    …
}
```

a points to x

a and b are aliases

Is this condition always satisfied?

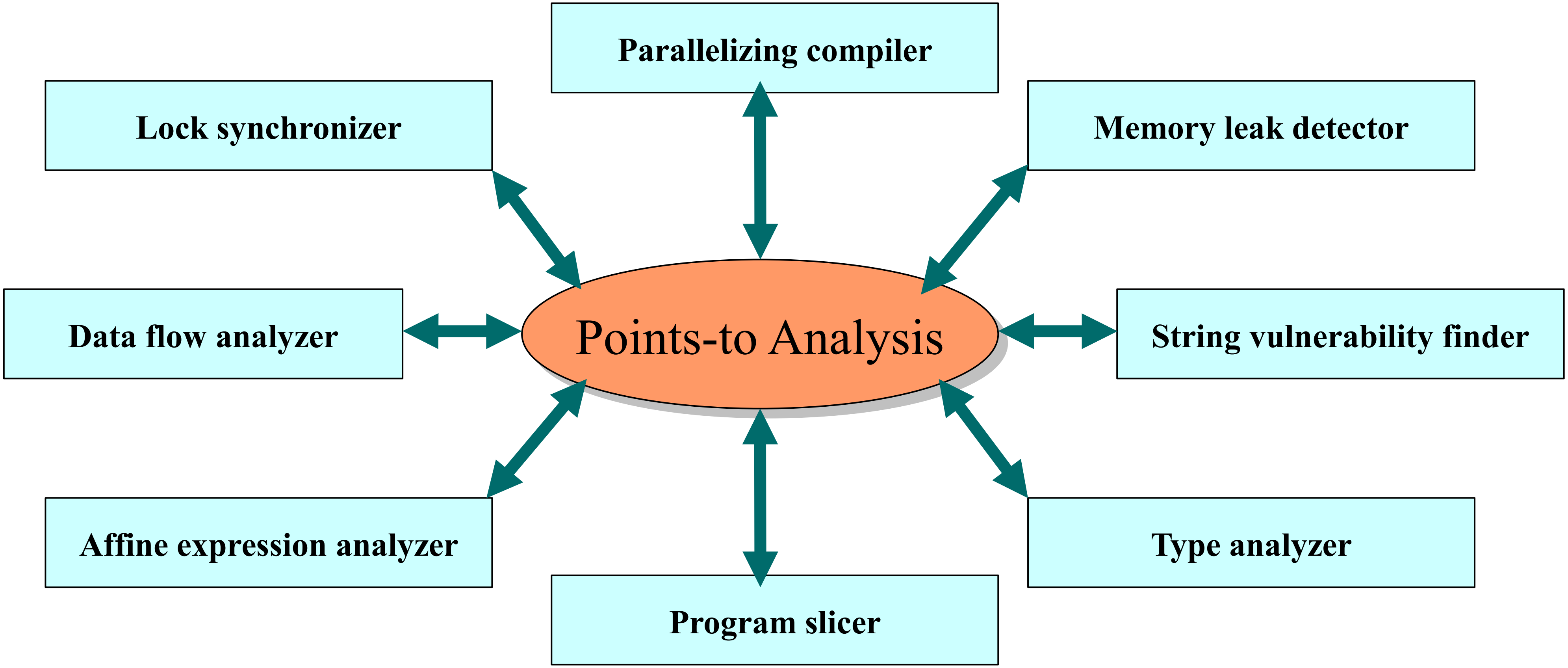
Pointer Analysis is a mechanism to **statically** find out run-time values of a pointer.

Why Points-to Analysis?

- for Parallelization
  - fun(p) || fun(q)
- for Optimization
  - a = p + 2;
  - b = q + 2;
- for Bug-Finding
- for Program Understanding
- ...

Clients of Points-to Analysis

Placement of Points-to Analysis



Parallelizing compiler

Lock synchronizer

Memory leak detector

Data flow analyzer

Points-to Analysis

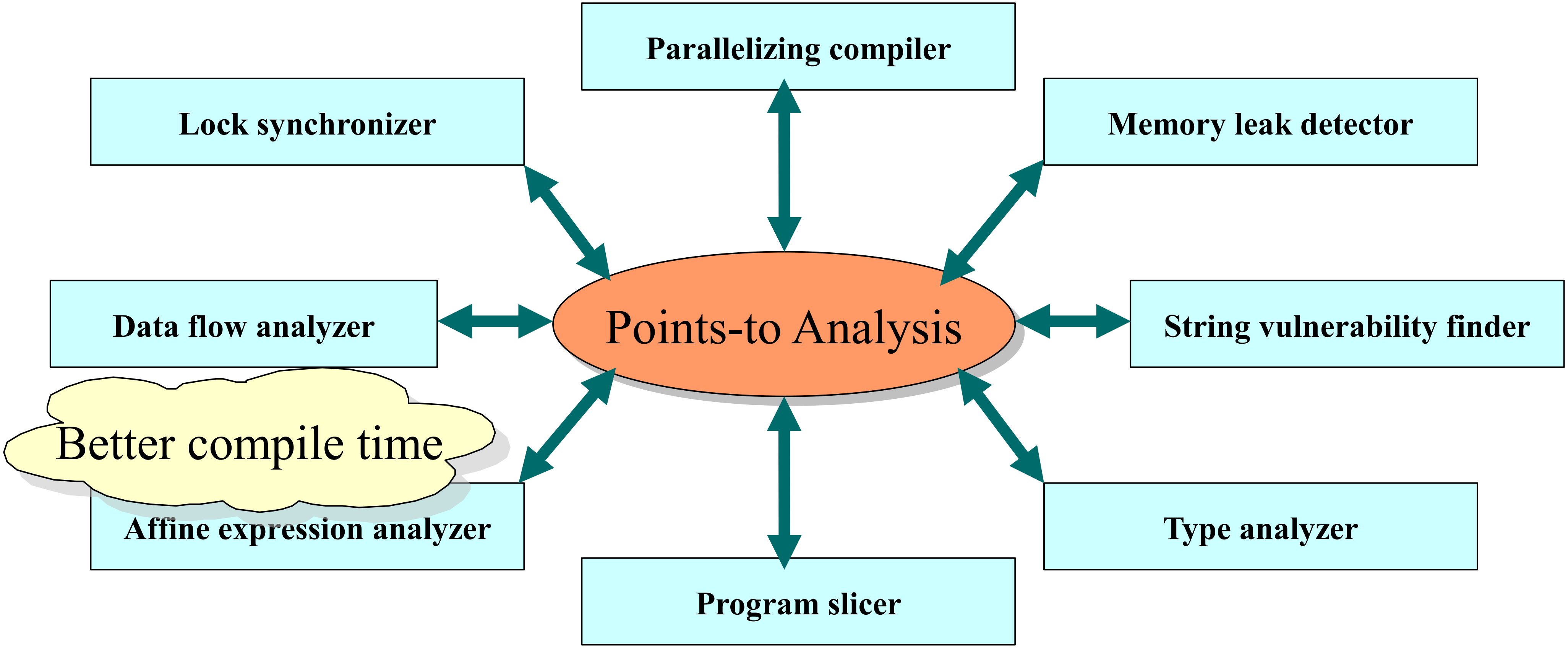
String vulnerability finder

Affine expression analyzer

Type analyzer

Program slicer

Placement of Points-to Analysis



Points-to Analysis

Parallelizing compiler

Lock synchronizer

Memory leak detector

Data flow analyzer

String vulnerability finder

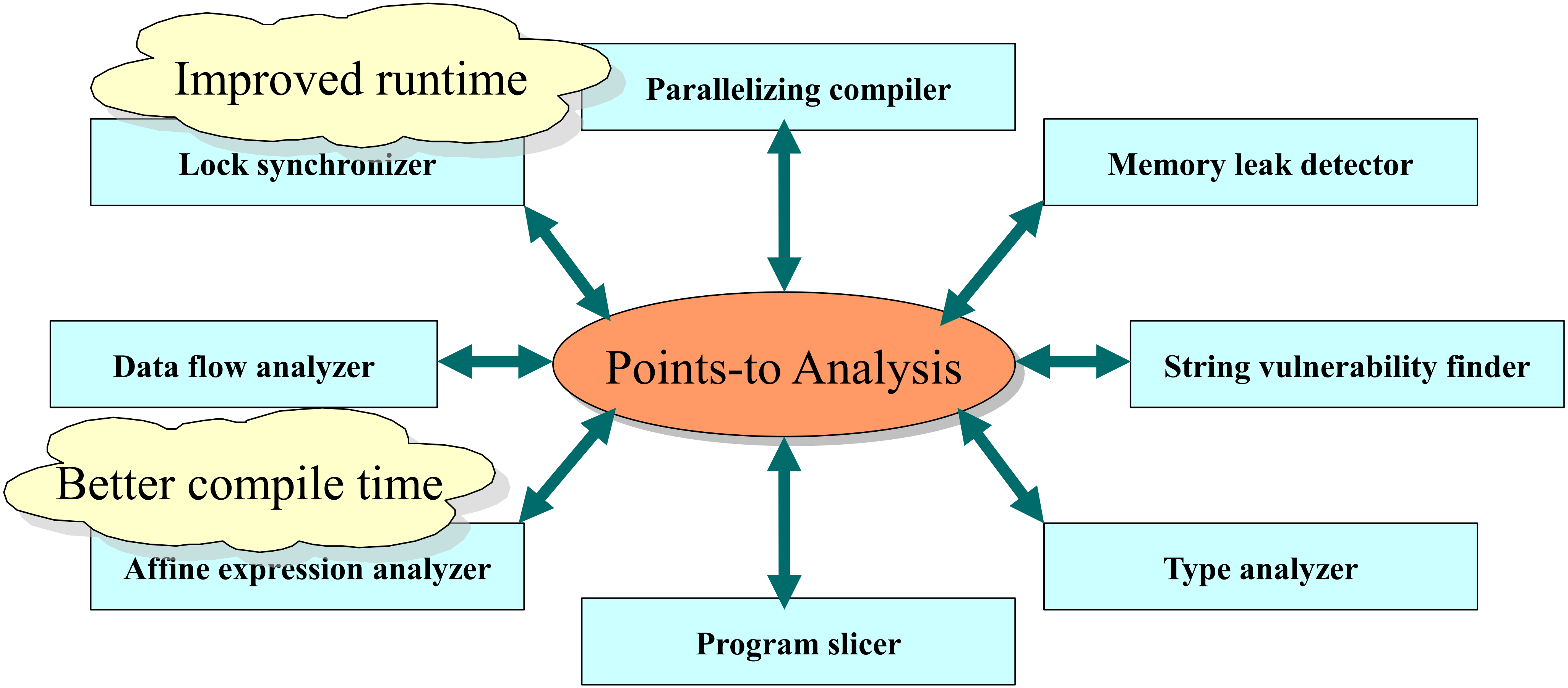
Better compile time

Affine expression analyzer

Type analyzer

Program slicer

Placement of Points-to Analysis



Improved runtime

Parallelizing compiler

Lock synchronizer

Memory leak detector

Data flow analyzer

Points-to Analysis

String vulnerability finder

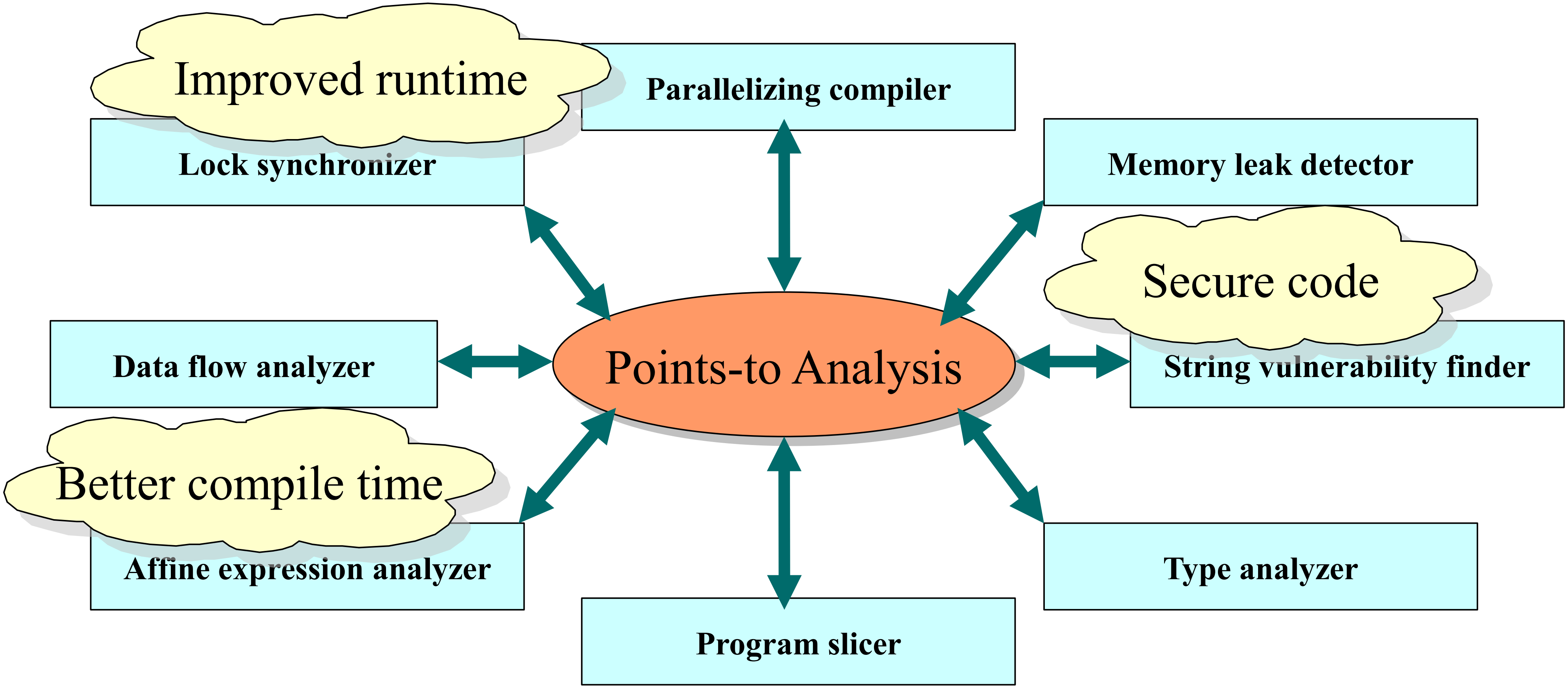
Better compile time

Affine expression analyzer

Type analyzer

Program slicer

Placement of Points-to Analysis



Improved runtime

Parallelizing compiler

Lock synchronizer

Memory leak detector

Secure code

Data flow analyzer

Points-to Analysis

String vulnerability finder

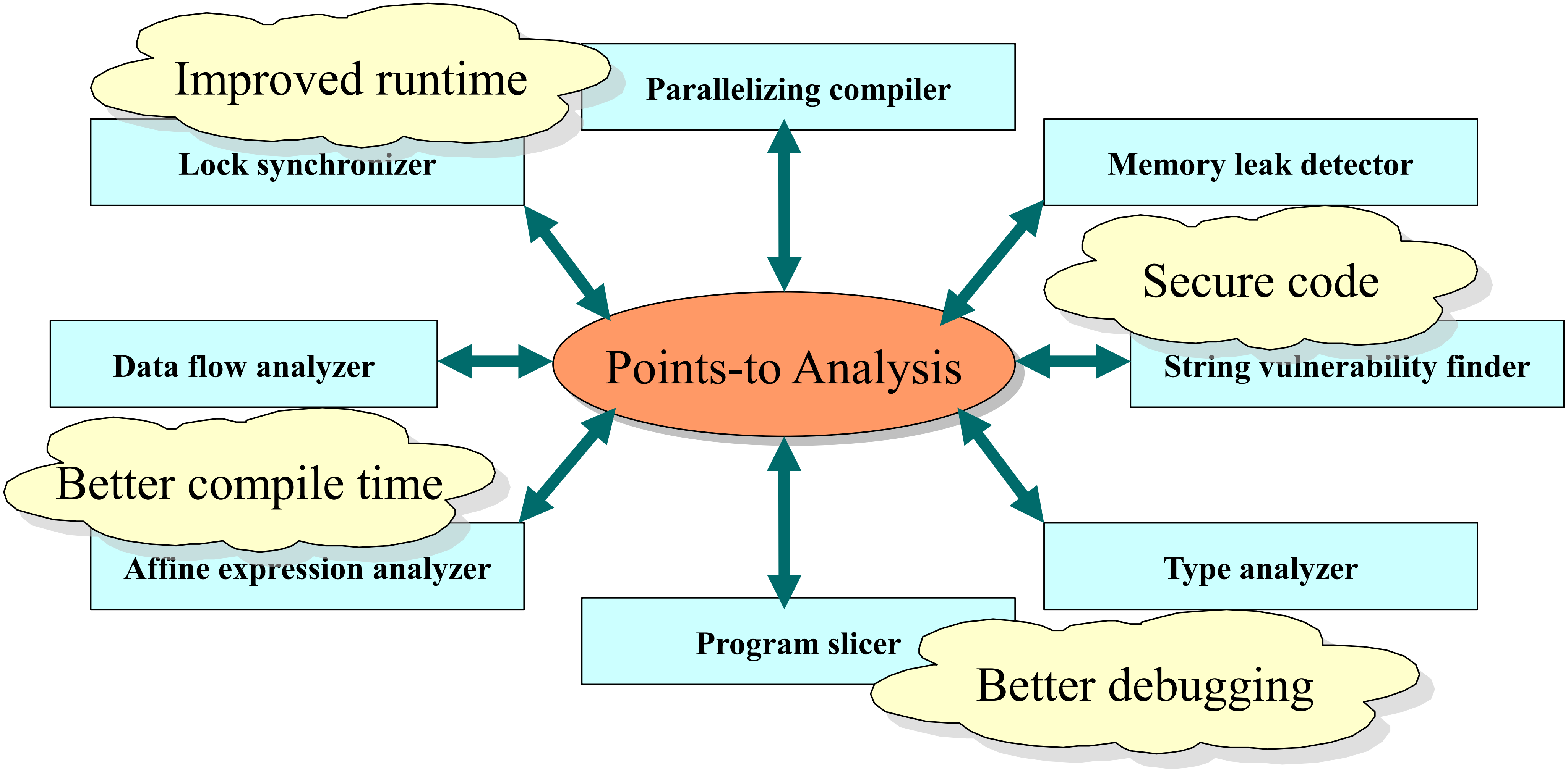
Better compile time

Affine expression analyzer

Type analyzer

Program slicer

Placement of Points-to Analysis



Improved runtime

Parallelizing compiler

Lock synchronizer

Memory leak detector

Secure code

Data flow analyzer

Points-to Analysis

String vulnerability finder

Better compile time

Affine expression analyzer

Type analyzer

Program slicer

Better debugging

Points-to Analysis

A C program can be normalized to contain only four types of pointer-manipulating statements or constraints.

Points-to constraints

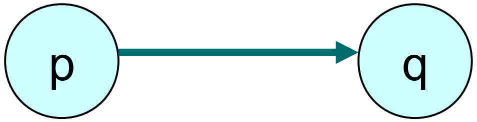
p = &q	address-of
p = q	copy
p = *q	load
*p = q	store

Points-to Analysis

A C program can be normalized to contain only four types of pointer-manipulating statements or constraints.

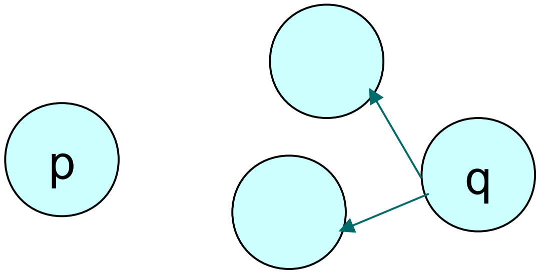
Points-to constraints

p = &q	**address-of**
p = q	copy
p = *q	load
*p = q	store



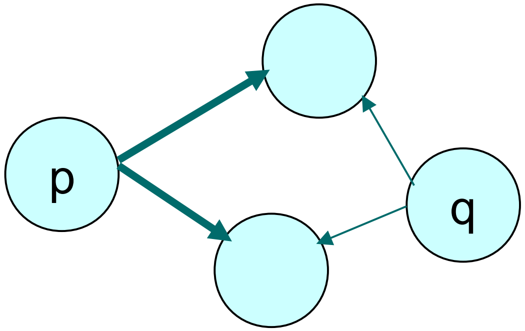
Points-to Analysis

A C program can be normalized to contain only four types of pointer-manipulating statements or constraints.



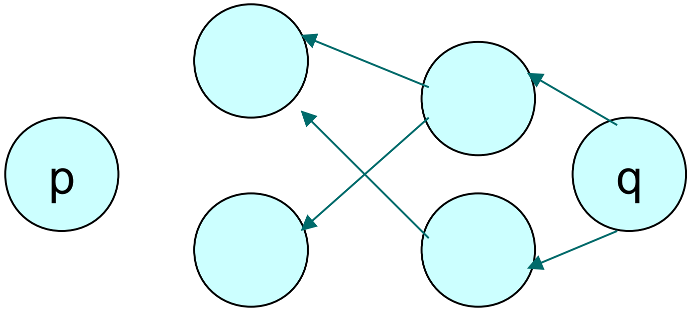
Points-to constraints

p = &q	address-of
p = q	**copy**
p = *q	load
*p = q	store



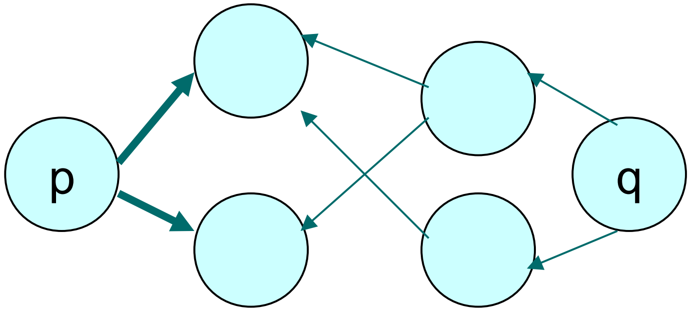
Points-to Analysis

A C program can be normalized to contain only four types of pointer-manipulating statements or constraints.



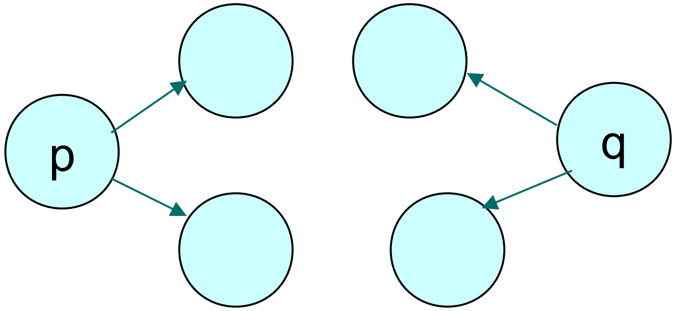
Points-to constraints

p = &q	address-of
p = q	copy
p = *q	**load**
*p = q	store



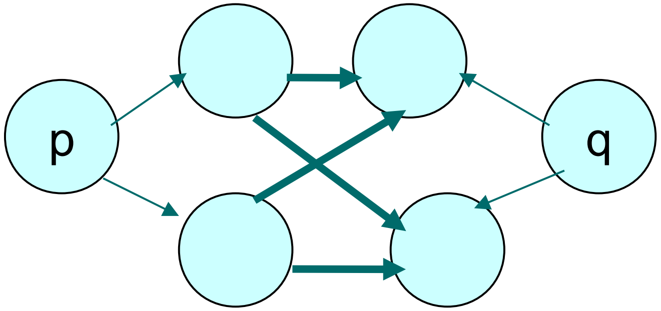
Points-to Analysis

A C program can be normalized to contain only four types of pointer-manipulating statements or constraints.



Points-to constraints

p = &q	address-of
p = q	copy
p = *q	load
***p = q**	**store**



Definitions

- Points-to analysis computes points-to information for each pointer.
- Alias analysis computes aliasing information for all pointers.
- Aliasing information can be computed using points-to information, but not vice versa.
- Clients often query for aliasing information, but storing it is expensive $O(n^2)$, hence frameworks store points-to information.
- If a→x, x is often called a pointee of a.

Points-to information

a → {x, y}
b → {y, z}
c → {z}

Aliasing information

	a	b	c
a	--	Yes	No
b	--	--	Yes
c	--	--	--

Nomenclature

- Pointer analysis: Ambiguous usage in literature. We will use it to refer to both points-to analysis and alias analysis.
- In the context of Java-like languages, it is called reference analysis.
- Also called as heap analysis.

Algebraic Properties

- Aliasing relation is reflexive, symmetric, but not transitive.
- Points-to relation is neither reflexive, nor symmetric, not even transitive.
- The points-to relation induces a restricted DAG for strictly typed languages.

Cyclic Dependence

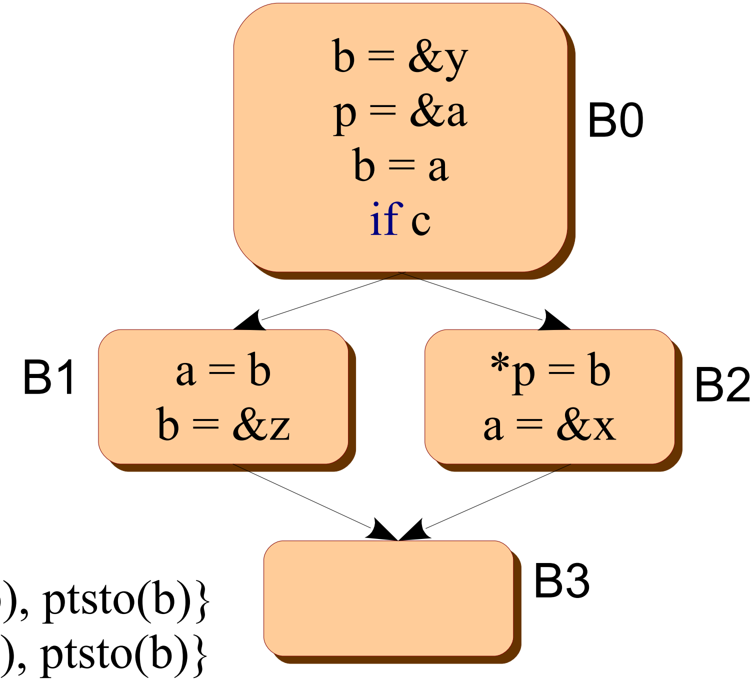
- Call graph ↔ function pointers
- Optimization ↔ points-to information

This slide is erroneous.

As a DFA

a = &x: gen{a → x}
a = b: gen{a → x} if {b → x}
a = *p: gen{a → x} if {p → b → x}
*p = a: gen{b → x} if {p → b and a → x}
 kill{b → x} if {p → b and b → x}

In(B) = U *Out(P) where P ∈ Pred(B)*
Out(B) = *Gen(B)* U *(In(B) – Kill(B))*



B0
b = &y
p = &a
b = a
if c

B1
a = b
b = &z

B2
*p = b
a = &x

B3

gen(B0) = {p→a, b→x if a→x}
gen(B1) = {a→x if b→x, b→z}
gen(B2) = {a→x,m→n if p→m and b→n and m ≠ a}
gen(B3) = { }

kill(B0) = {ptsto(p), ptsto(b)}
kill(B1) = {ptsto(a), ptsto(b)}
kill(B2) = {ptsto(ptsto(p)), ptsto(a)}
kill(B3) = { }

	in1	out1	in2	out2	in3	out3
B0	{}	{p→a}	{}	{p→a,b→{x,z}}	{}	{p→a,b→{x,z}}
B1	{}	{b→z}	out1(B0)	{p→a,a→{x,z},b→{z}}	out2(B0)	{p→a,a→{x,z},b→{z}}
B2	{}	{a→x}	out1(B0)	{p→a,a→{x},b→{x,z}}	out2(B0)	{p→a,a→{x},b→{x,z}}
B3	{}	{}	out1(B1) U out1(B2)	{p→a,a→{x,z},b→{x,z}}	out2(B1) U out2(B2)	{p→a,a→{x,z},b→{x,z}}

As a DFA: Notes

- Gen and Kill are dynamic (not fixed before analysis).
- Gen/Kill and Points-to Information are cyclically dependent.
- Single copy of a variable leads to imprecision.
  - e.g., a's points-to set doesn't reach B0 in any execution, but the analysis treats it otherwise.

Classwork

a = &x: gen{a → x}

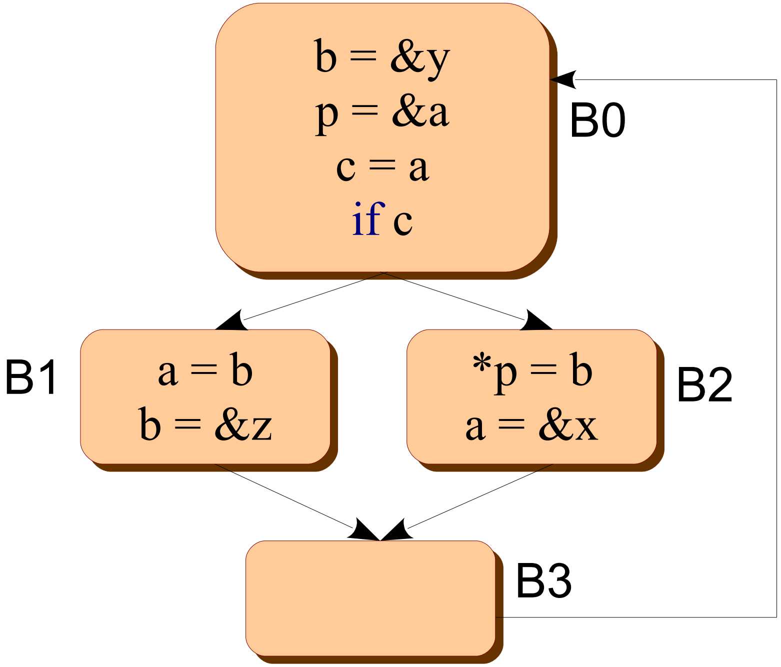
a = b: gen{a → x} if {b → x}

a = *p: gen{a → x} if {p → b → x}

*p = a: gen{b → x} if {p → b and a → x}
kill{b → x} if {p → b and b → x}

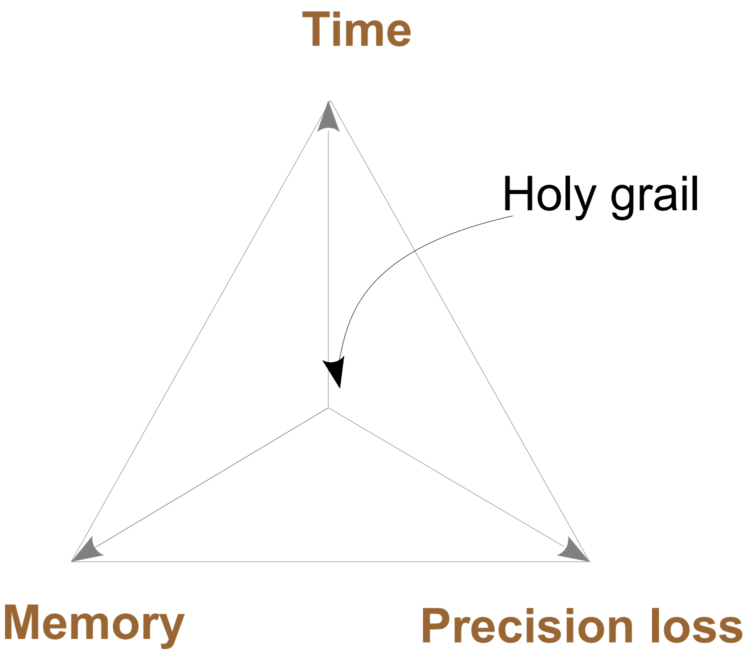
In(B) = U *Out(P) where P* ∈ *Pred(B)*

Out(B) = *Gen(B)* U *(In(B) – Kill(B))*



Design Decisions

- Analysis dimensions
- Heap modeling
- Set implementation
- Call graph, function pointers
- Array indices



Analysis Dimensions

An analysis's precision and efficiency is guided by various design decisions.

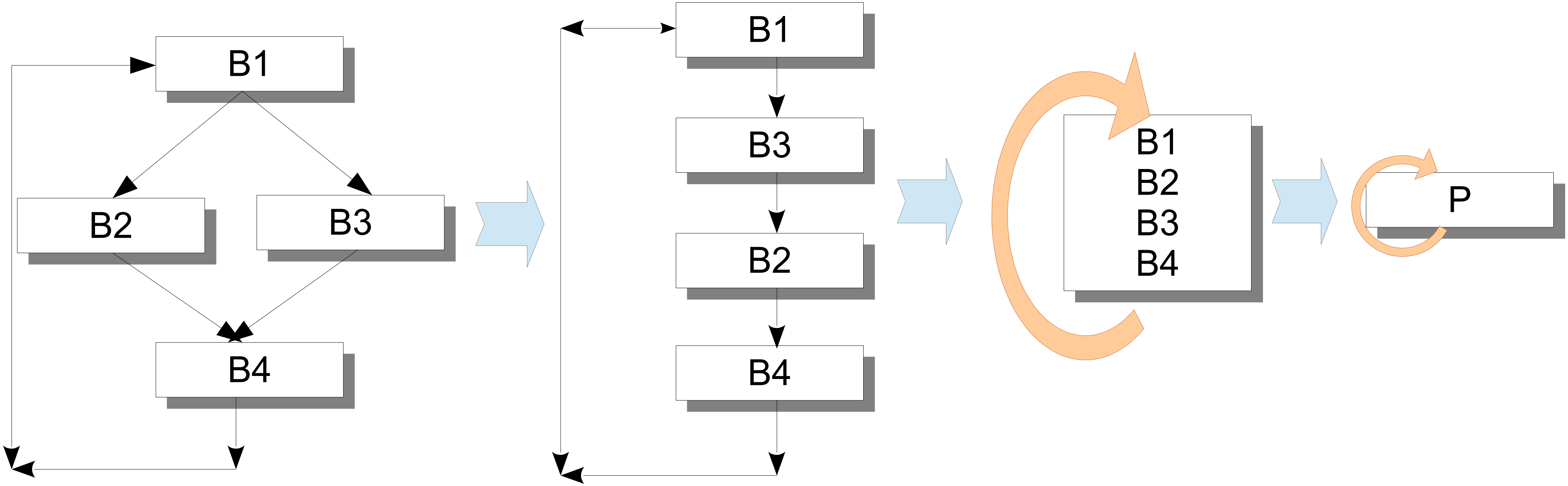
- Flow-sensitivity
- Context-sensitivity
- Path-sensitivity
- Field-sensitivity

Flow-sensitivity

```
L0:  a = &x;
L1:  a = &y;
L2:  ...
```

Flow-sensitive solution: *at L1 a points to x, at L2 a points to y*
Flow-insensitive solution: *in the program a's points-to set is {x, y}*

Flow-insensitive analyses ignore the control-flow in the program.



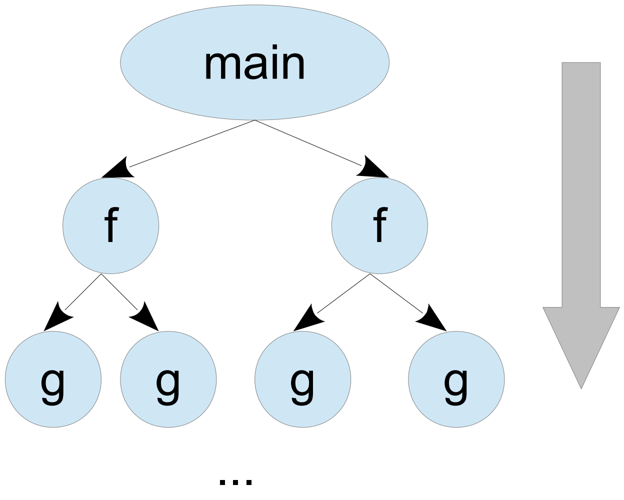
Context-sensitivity

```
main() {
        L0:  fun(&x);
        L1:  fun(&y);
}
```

```
fun(int *a) {
        b = a;
}
```

Context-sensitive solution:
b points to x along L0, b points to y along L1

Context-insensitive solution:
b's points-to set is {x, y} in the program



Exponential Number of contexts

Along main-f1-g1, …
Along main-f1-g2, …
Along main-f2-g1, …
Along main-f2-g2, ...

Exponential time requirement

Exponential storage requirement

Context-sensitivity

```
main() {
      L0:  fun(&x);
      L1:  fun(&y);
}
```

```
fun(int *a) {
      b = a;
}
```

Context-sensitive solution:

b points to x along L0, b points to y along L1

Context-insensitive solution:

Inter-procedural → *b's points-to set is {x, y} in the program*

intra-procedural → *b's points-to set is {all address-taken variables}*

Path-sensitivity

```
if (a == 0)
        b = &x;
else
        b = &y;
```

Path-sensitive solution:
b points-to x when a is 0, b points-to y when a is not 0

Path-insensitive solution:
b's points-to set is {x, y} in the program

```
if (c1)
   while (c2) {
      if (c3)
         …
      else
        for (; c4; )
           ...
   }
else
    ...
```

c1 and c2 and c3, …
c1 and c2 and !c3 and c4, …
c1 and c2 and !c3 and !c4, …
c1 and !c2, …
!c1 ...
...

Field-sensitivity

```
struct T s;

s.a = &x;
s.b = &y;
```

Field-sensitive solution:
s.a points-to x, s.b points-to y

Field-insensitive solution:
s's points-to set is {x, y}

Aggregates are collapsed into a single variable.
e.g., arrays, structures, unions.

This reduces the number of variables tracked during the analysis and reduces precision.

Andersen's Analysis

- Inclusion-based / subset-based / constraint-based analysis
- Flow-insensitive analysis

For a statement $p = q$,
create a constraint $\mathrm{ptsto}(p) \supseteq \mathrm{ptsto}(q)$

where p is of the form *a, a, and q is of the form *a, a, &a.

Solving these inclusion constraints results into the points-to solution.

Andersen's Analysis: Example

Program

```
a = &x;
b = &y;
p = &a;
c = b;
*p = c;
```

Constraints

$\mathrm{ptsto}(a) \supseteq \{x\}$

$\mathrm{ptsto}(b) \supseteq \{y\}$

$\mathrm{ptsto}(p) \supseteq \{a\}$

$\mathrm{ptsto}(c) \supseteq \mathrm{ptsto}(b)$

$\mathrm{ptsto}(*p) \supseteq \mathrm{ptsto}(c)$

fixed-point

Pointers	Iteration 0	Iteration 1	Iteration 2
a	{ }	{x, y}	
b	{ }	{y}	
c	{ }	{y}	
p	{ }	{a}	
x	{ }		
y	{ }		

Imprecision

Andersen's Analysis: Modified Example

Program

```
a = &x;
b = &y;
p = &a;
*p = c;
c = b;
```

Constraints

$\text{ptsto}(a) \supseteq \{x\}$

$\text{ptsto}(b) \supseteq \{y\}$

$\text{ptsto}(p) \supseteq \{a\}$

$\text{ptsto}(*p) \supseteq \text{ptsto}(c)$

$\text{ptsto}(c) \supseteq \text{ptsto}(b)$

Order does not matter
for correctness,
but it does matter
for efficiency.

fixed-point

Pointers	Iteration 0	Iteration 1	Iteration 2	Iteration 3
a	{ }	{x}	{x, y}	
b	{ }	{y}		
c	{ }	{y}		
p	{ }	{a}		
x	{ }			
y	{ }			

Andersen's Analysis: Classwork

Program

```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```

Constraints

ptsto(*p) ⊇ ptsto(c)
ptsto(b) ⊇ {y}
ptsto(b) ⊇ ptsto(*p)
ptsto(p) ⊇ {a}
ptsto(a) ⊇ {x}
ptsto(*p) ⊇ ptsto(c)
ptsto(c) ⊇ ptsto(p)
ptsto(c) ⊇ {z}

fixed-point

Pointers	Iteration 0	Iteration 1	Iteration 2	Iteration 3
a	{ }	{x}	{a, x, z}	
b	{ }	{y}	{a, x, y, z}	
c	{ }	{a, z}	{a, z}	
p	{ }	{a}	{a}	
x	{ }			
y	{ }			
z				

Andersen's Analysis: Optimizations

- Avoid duplicates
- Reorder constraints
- Process address-of constraints once
- Difference propagation

Andersen's Analysis: Complexity

- Total information computed (storage) = $O(n^2)$
- From each pointer
  - To each other pointer
    - Propagate O(n) information
      - O(n) times

$O(n^4)$ Naive

- From each pointer
  - To each other pointer
    - Propagate O(n) information

$O(n^3)$ Difference Propagation

Open: Can you reduce the gap between storage and time complexities?

Steensgaard's Analysis

- Unification-based
- Almost linear time $O(m\alpha(m))$
- More imprecise

For a statement p = q, merge the points-to sets of p and q.

In subset terms, $\text{ptsto}(p) \supseteq \text{ptsto}(q)$ **and** $\text{ptsto}(q) \supseteq \text{ptsto}(p)$ with a single representative element.

Steensgaard's Analysis: Example

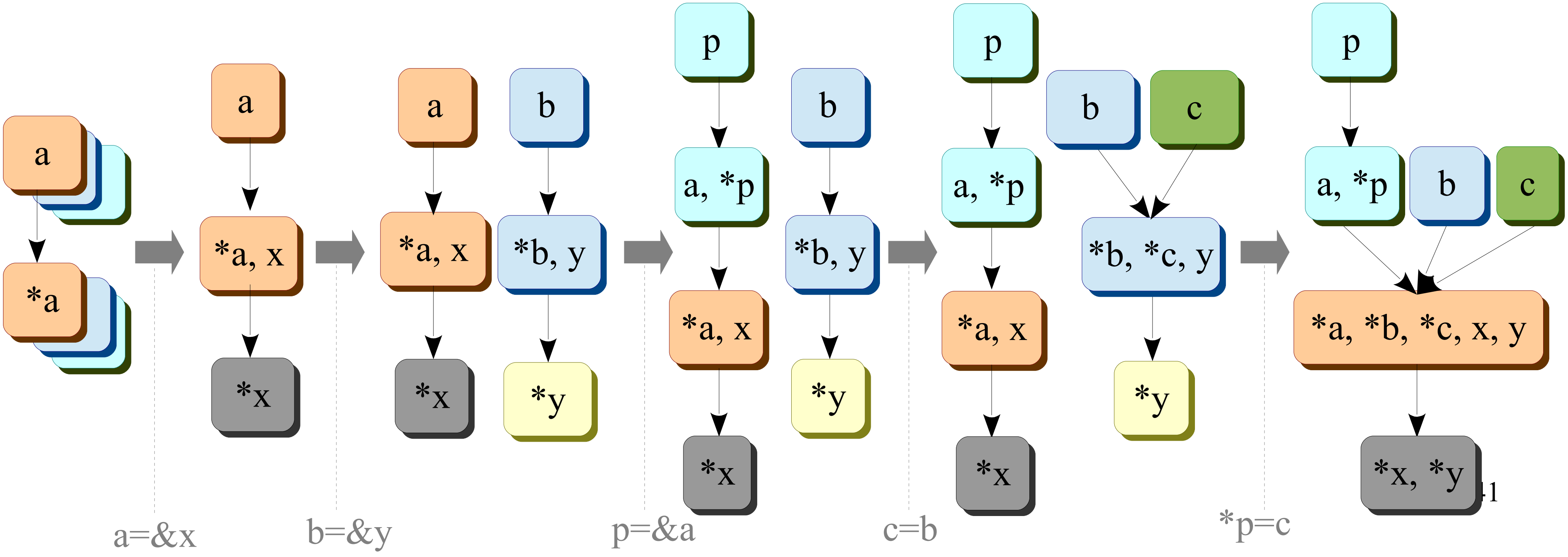
Program	Andersen's	Steensgaard's
a = &x;	a → {x, y}	a → {x, y}
b = &y;	b → {y}	b → {x, y}
p = &a;	c → {y}	c → {x, y}
c = b;	p → {a}	p → {a}
*p = c;		

Pointers	Iteration 0	Iteration 1
a	{*a}	{*a, *b, *c, x, y}
b	{*b}	{*a, *b, *c, x, y}
c	{*c}	{*a, *b, *c, x, y}
p	{*p}	{*p, a}
x	{*x}	
y	{*y}	

Only one iteration

Steensgaard's Hierarchy

Program	Andersen's	Steensgaard's
a = &x;	a → {x, y}	a → {x, y}
b = &y;	b → {y}	b → {x, y}
p = &a;	c → {y}	c → {x, y}
c = b;	p → {a}	p → {a}
*p = c;		



Classwork

Program

```
*p = c;
b = &y;
b = *p;
p = &a;
a = &x;
*p = c;
c = p;
c = &z;
```

Andersen's

a → {a, x, z}
b → {a, x, y, z}
c → {a, z}
p → {a}

Steensgaard's

Steensgaard's Hierarchy

- What is its structure?
- How many incoming edges to each node?
- How many outgoing edges from each node?
- Can there be cycles?
- What happens to p = &p?
- What is the precision difference between Andersen's and Steensgaard's analyses?
- If for each P = Q, we add Q = P and solve using Andersen's analysis, would it be equivalent to Steensgaard's analysis?

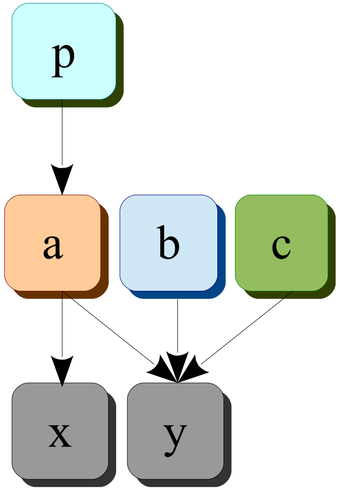
Unifying Model Two

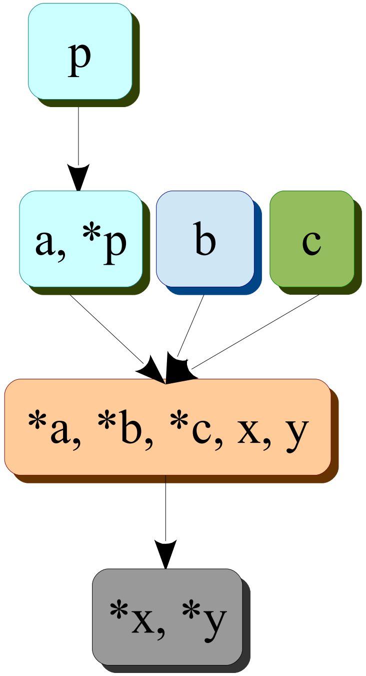
- Steensgaard's hierarchy is characterized by a single outgoing edge.
- Andersen's points-to graph can have arbitrary number of outgoing edges (maximum n).
- Number of edges in between the two provide precision-scalability trade-off.

Unifying Model Two

Program	Andersen's	Steensgaard's

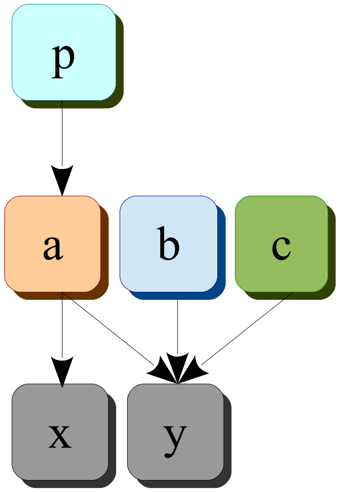
a = &x;
b = &y;
p = &a;
c = b;
*p = c;
	a → {x, y}
	b → {y}
	c → {y}
	p → {a}
		a → {x, y}
		b → {x, y}
		c → {x, y}
		p → {a}

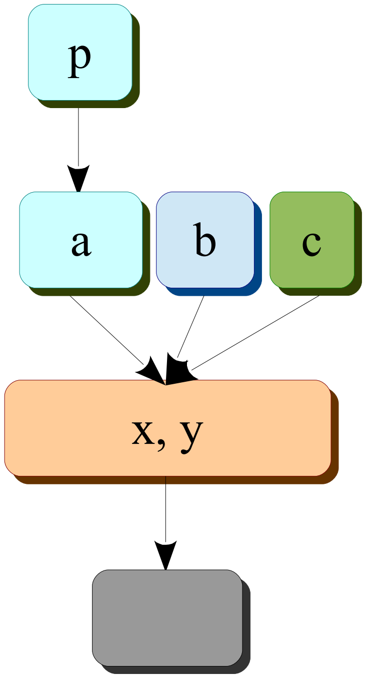




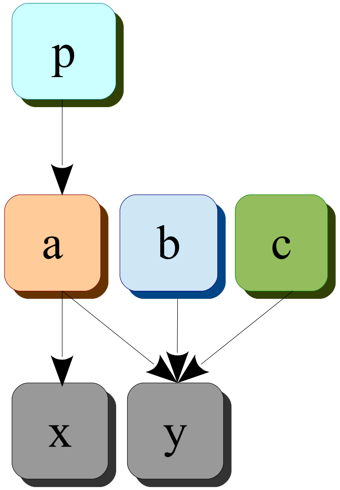
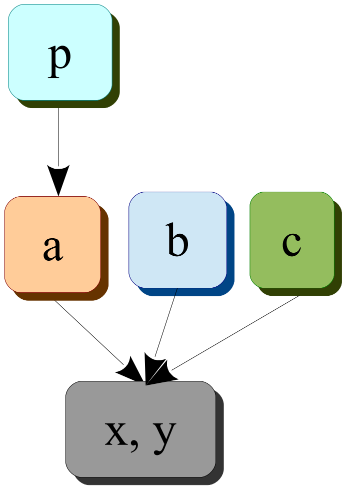
Unifying Model Two

Program	Andersen's	Steensgaard's
a = &x;	a → {x, y}	a → {x, y}
b = &y;	b → {y}	b → {x, y}
p = &a;	c → {y}	c → {x, y}
c = b;	p → {a}	p → {a}
*p = c;		

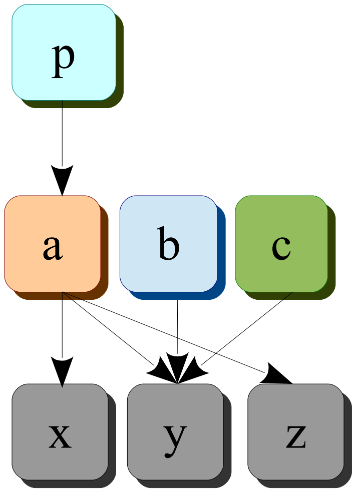
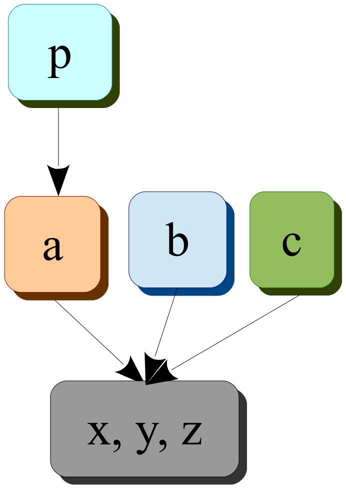




Unifying Model Two

Program	Andersen's	Steensgaard's
a = &x;	a → {x, y}	a → {x, y}
b = &y;	b → {y}	b → {x, y}
p = &a;	c → {y}	c → {x, y}
c = b;	p → {a}	p → {a}
*p = c;		
		

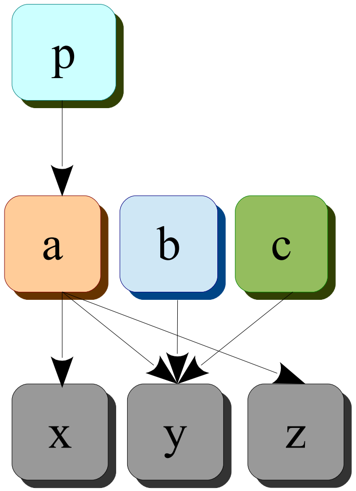
Unifying Model Two

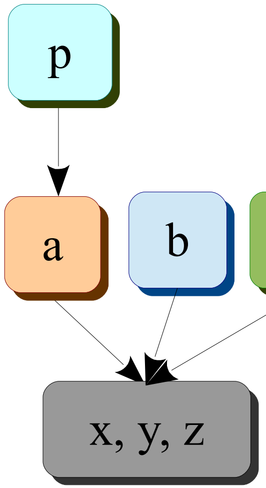
Program	Andersen's	Steensgaard's
a = &x;	a → {x, y, z}	a → {x, y, z}
b = &y;	b → {y}	b → {x, y, z}
p = &a;	c → {y}	c → {x, y, z}
c = b;	p → {a}	p → {a}
*p = c;		
a = &z;		
		

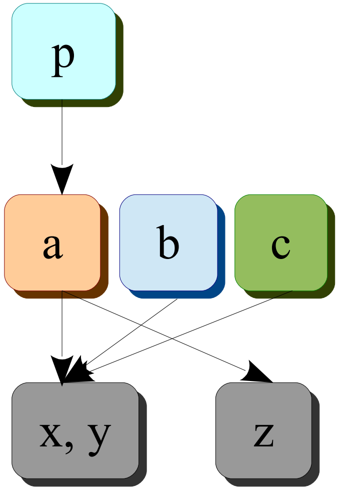
Unifying Model Two

Program	Andersen's	Steensgaard's	In between

a = &x;
b = &y;
p = &a;
c = b;
*p = c;
a = &z;
	a → {x, y, z}
	b → {y}
	c → {y}
	p → {a}
		a → {x, y, z}
		b → {x, y, z}
		c → {x, y, z}
		p → {a}
			a → {x, y, z}
			b → {x, y}
			c → {x, y}
			p → {a}







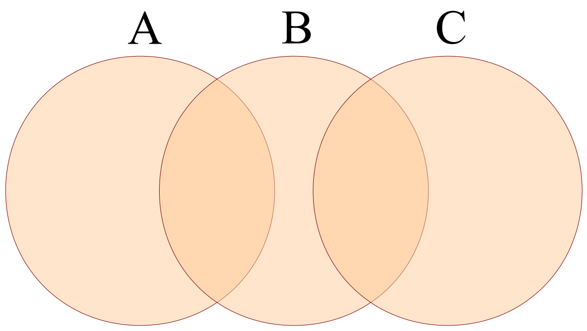
What if x and z are merged?

Unifying Model One

- Steensgaard's unification can be viewed as equality of points-to sets.
- Thus, if $a = b$ merges their points-to sets and $b = c$ merges their points-to sets, then a and c become aliases!
- Remember: aliasing is not transitive.
- So, unification adds transitivity to the aliasing relation.

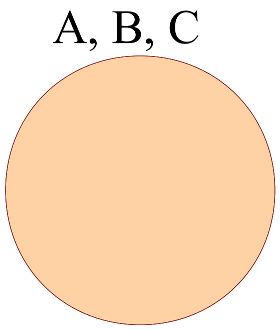
Unifying Model One

Andersen's



Aliasing is non-transitive

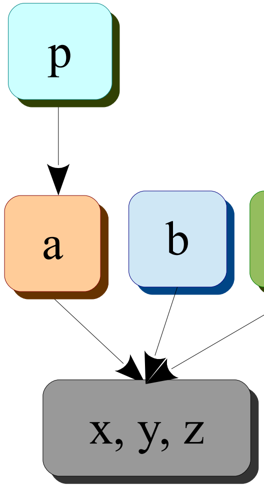
Steensgaard's



Aliasing becomes transitive

Back to Steensgaard's

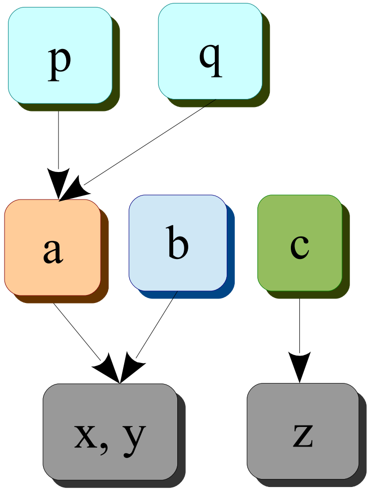
- Aliasing relation is transitive.
- We know that it is also reflexive and symmetric.
- This means aliasing becomes an equivalence relation.
- Steensgaard's unification partitions pointers into equivalent sets.



All predecessors of a node form a partition.
The equivalence sets are {p}, {a, b, c}, {x, y, z}.

Back to Steensgaard's

- Aliasing relation is transitive.
- We know that it is also reflexive and symmetric.
- This means aliasing becomes an equivalence relation.
- Steensgaard's unification partitions pointers into equivalent sets.



All predecessors of a node form a partition.
The equivalence sets are {p, q}, {a, b}, {c}, {x, y}, {z}.

Realizable Facts

Statements	Andersen's points-to
a = &c	a → {b, c}
b = &a	b → {a, b, c}
c = &b	c → {b}
b = a	d → {a, b, c}
*b = c	
d = *a	

A realizability sequence is a sequence of statements such that a given points-to fact is satisfied.

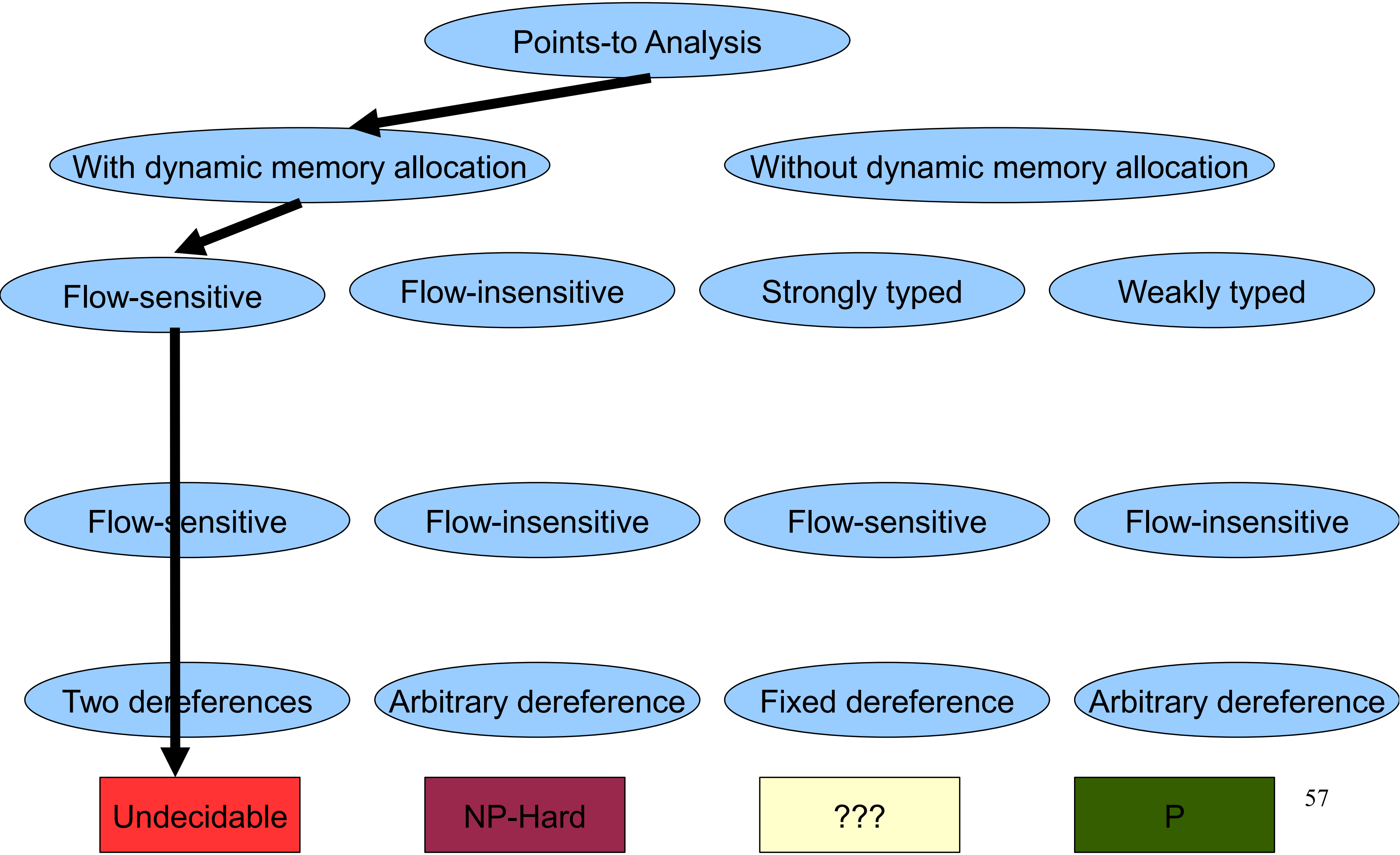
The realizability sequence for b→c is a=&c, b=a.
The realizability sequence for a→b is c=&b, b=&a, *b=c.
Classwork: What is the realizability sequence for d → a?
Classwork: What is the realizability sequence for d → c?
a→b and b→c are realizable individually, but not simultaneously.

```
int *fun(int *a, int *b) {
        int *c;
        if (*a == *b) {
                c = b;
        } else {
                c = a;
        }
        return c;
}
int *g;
void main() {
        int *x, *y, *z, **w;
        int m = 0, n = 1;
        char *str;
        x = &m;
        y = &n;
        str = (char *)malloc(30);
        w = (int *)&str;
        if (m < n) {
                strcpy(str, “m is smaller\n”);
                z = fun(y, x);
        } else {
                printf(“m is >= n\n”);
                w = &x;
                *w = fun(x, y);
        }
        printf(“**w=%d\n”, **w);
}
```

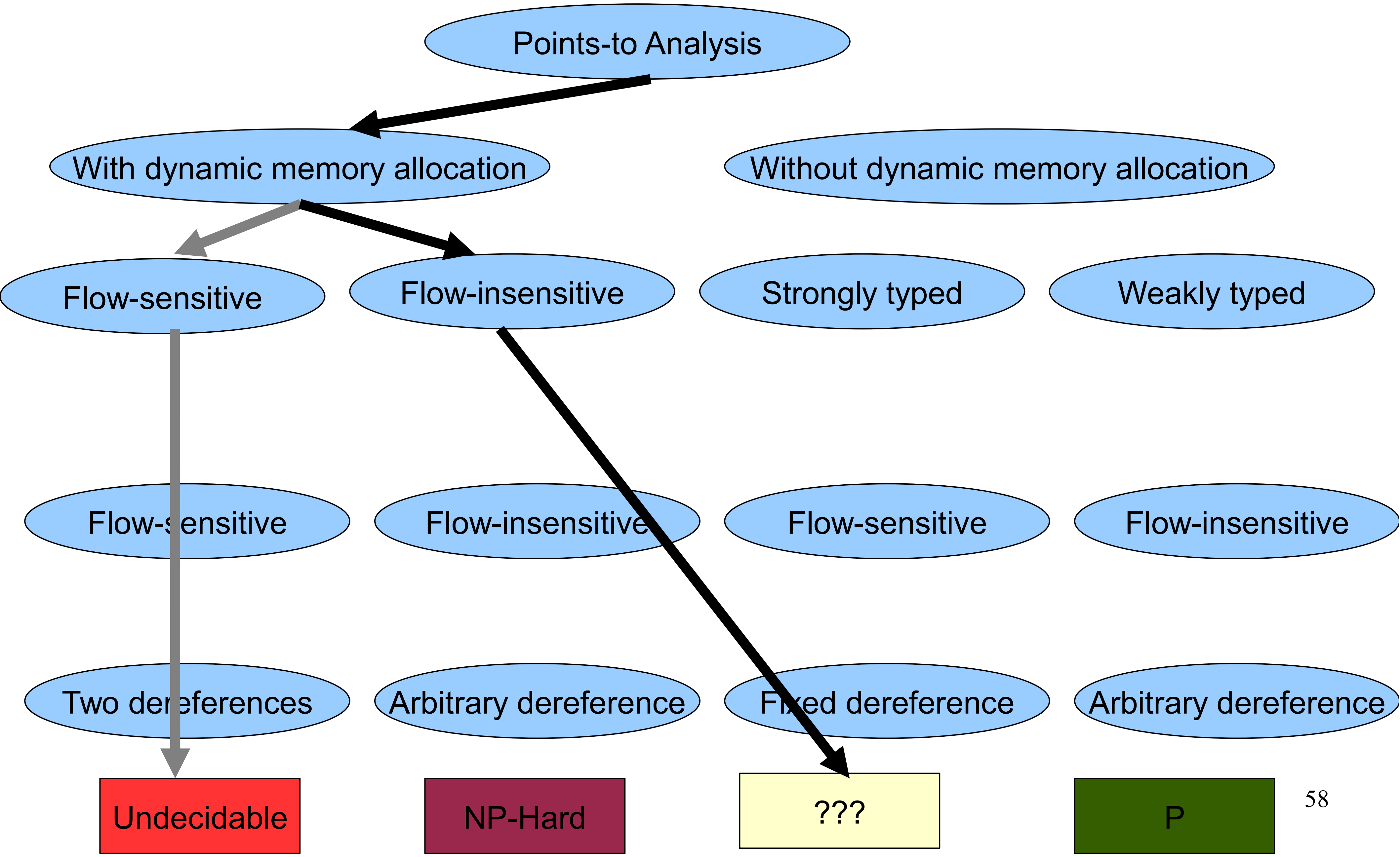
- How do we take care of malloc?
- How do we take care of type-casts?
- Find the set of normalized statements for intra-procedural pointer analysis.
- Perform intra-procedural Andersen's analysis.
- How do we take care of strcpy and printf? How about the global g?
- Perform inter-procedural context-insensitive Andersen's analysis.
- Perform Steensgaard's analysis.

Extra

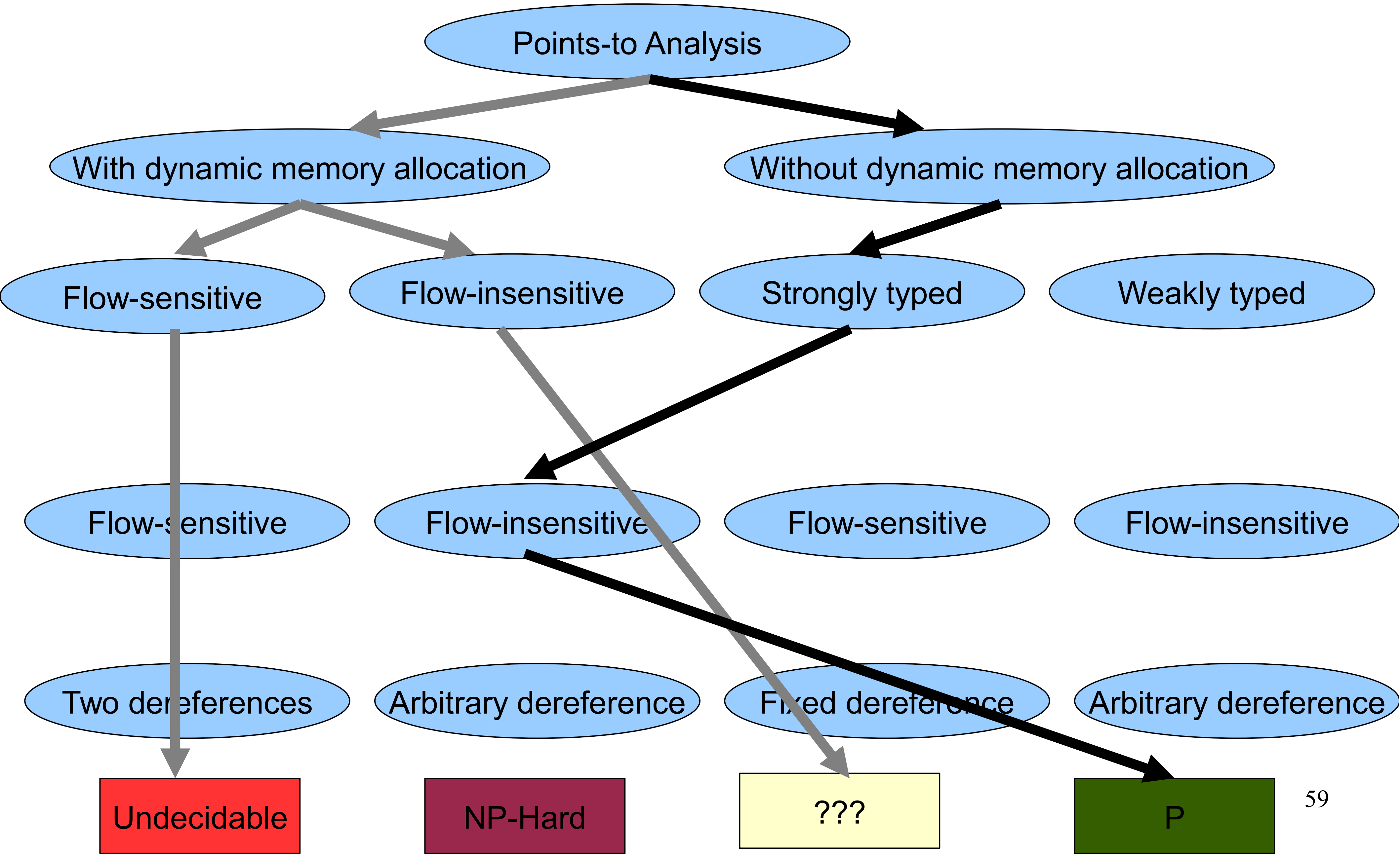
Complexity of Points-to Analysis



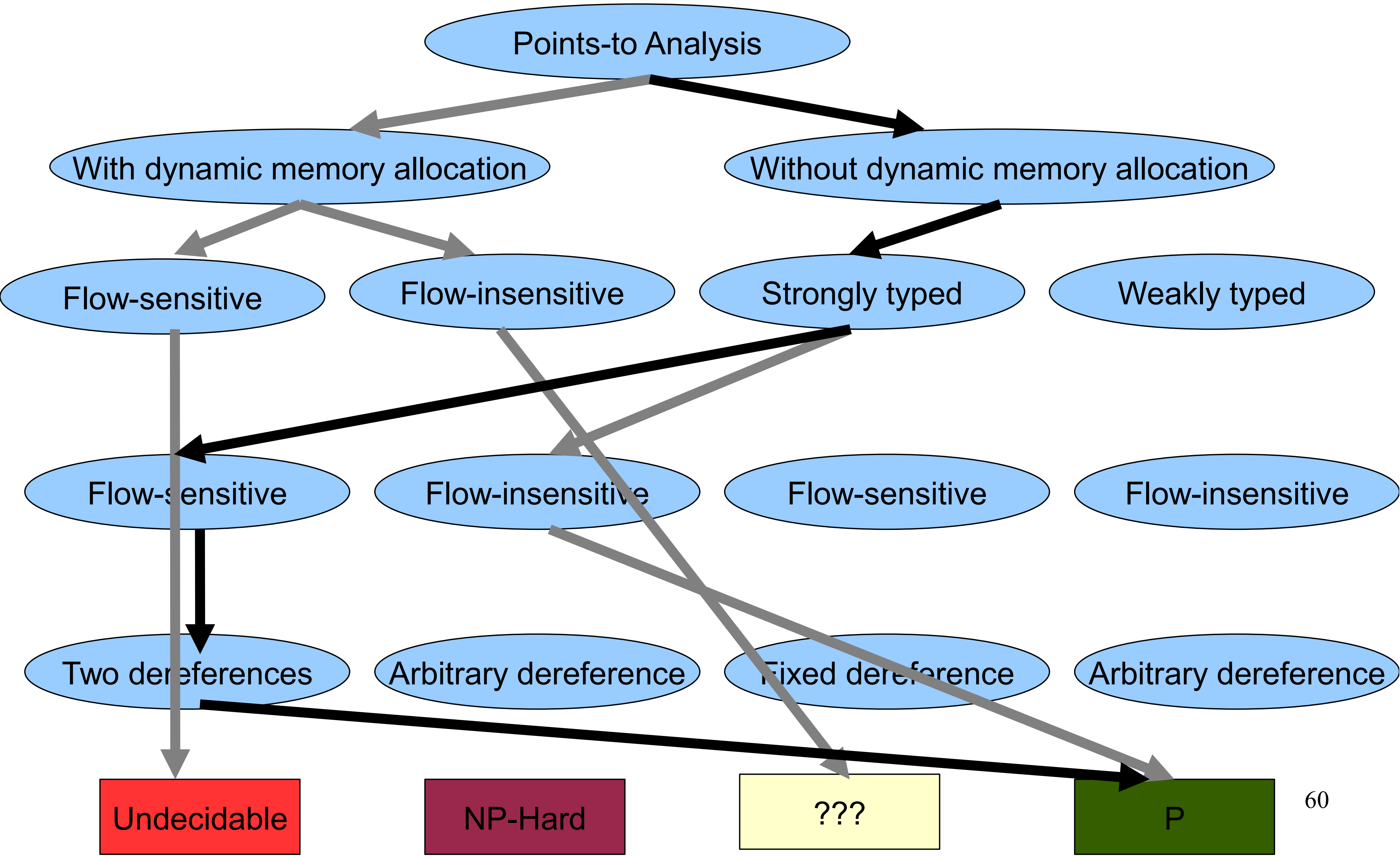
Complexity of Points-to Analysis



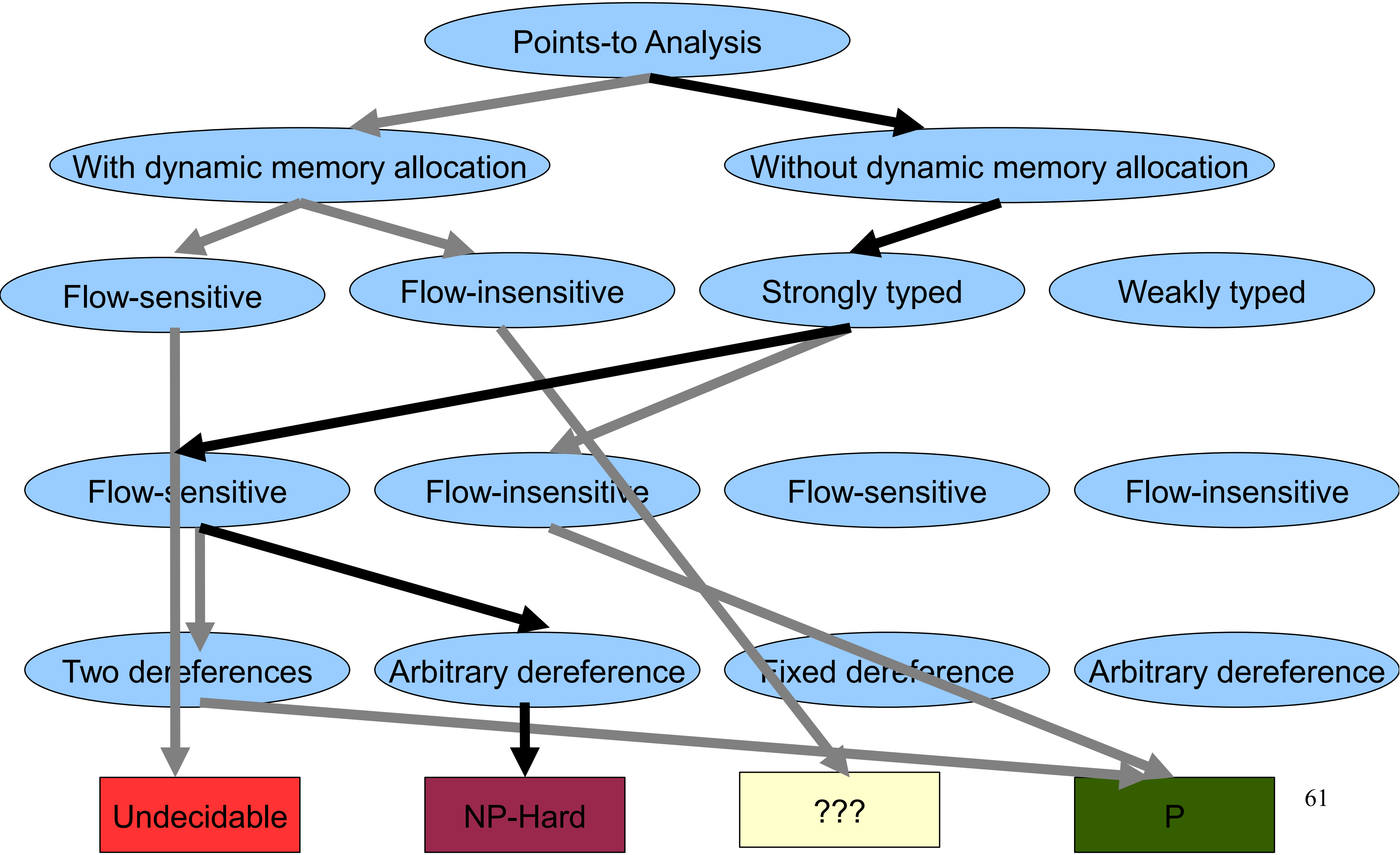
Complexity of Points-to Analysis



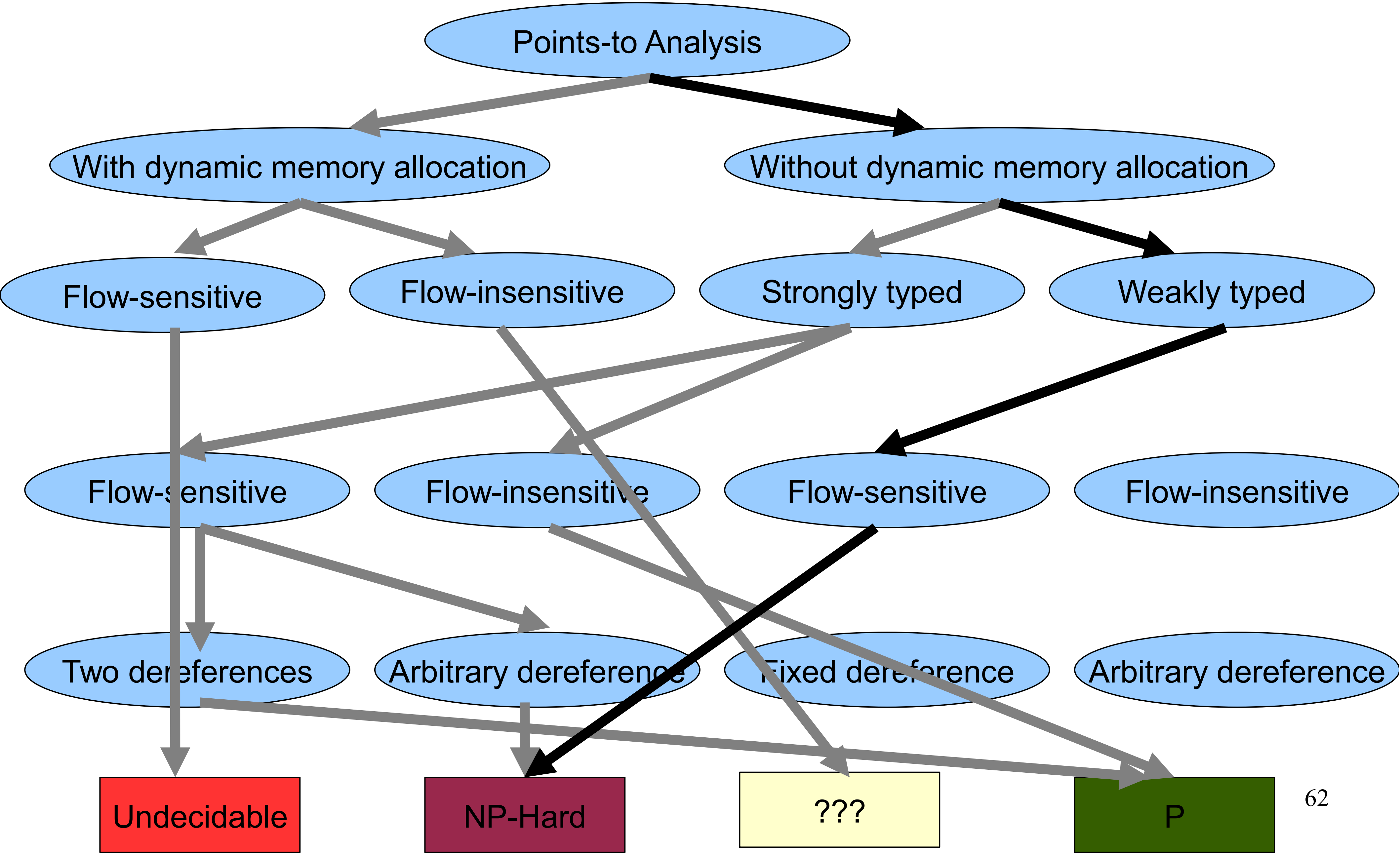
Complexity of Points-to Analysis



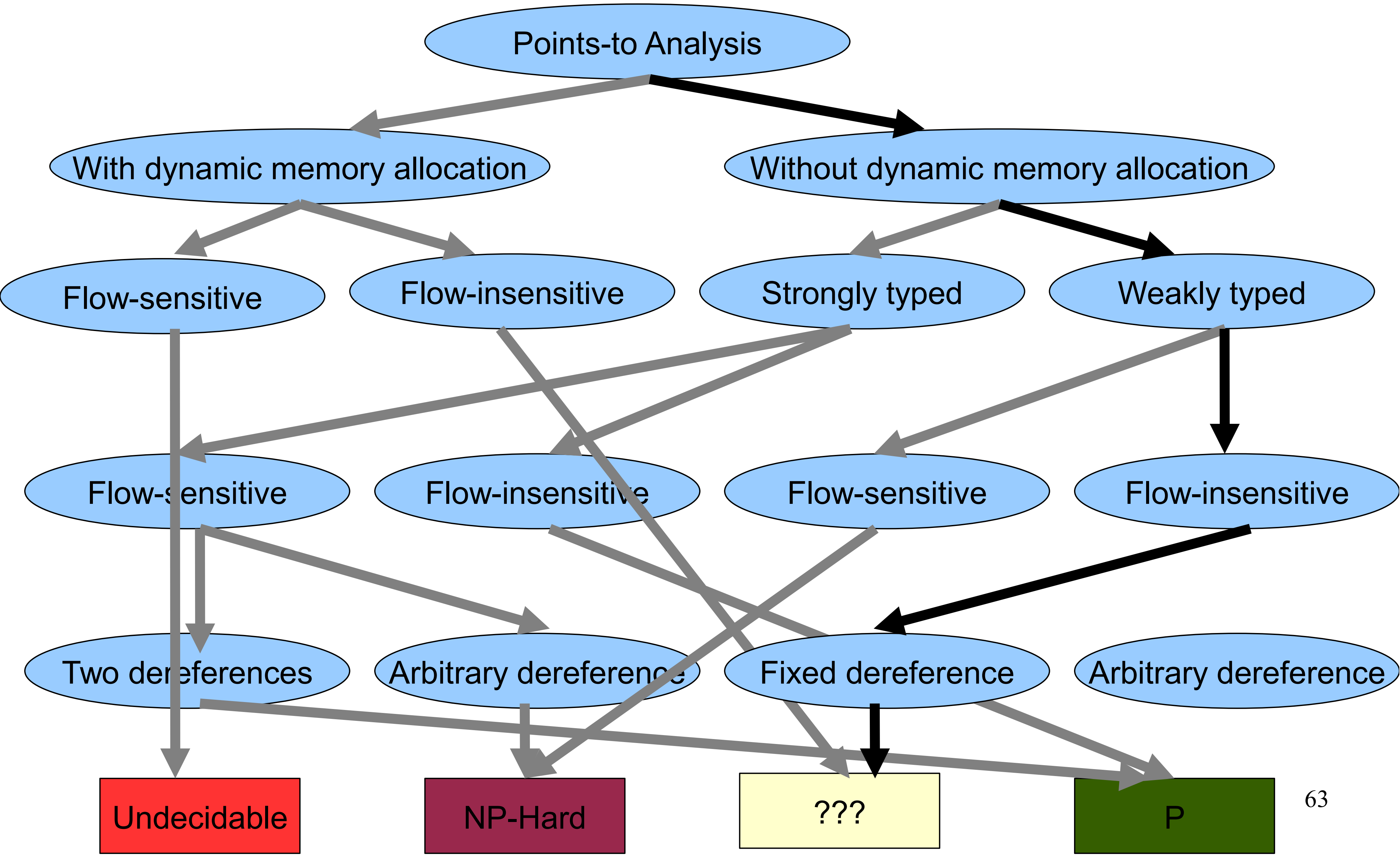
Complexity of Points-to Analysis



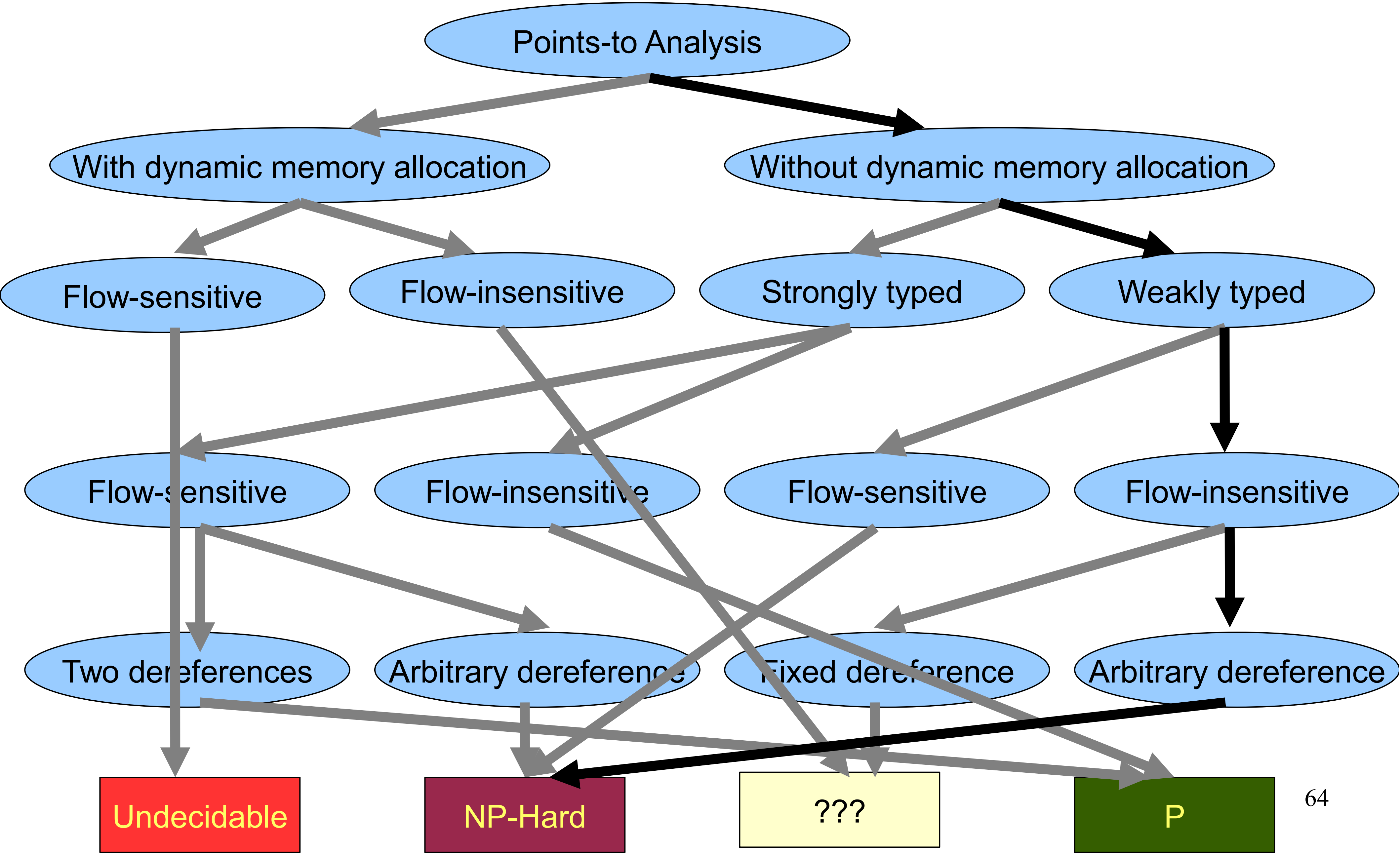
Complexity of Points-to Analysis



Complexity of Points-to Analysis



Complexity of Points-to Analysis



Related Work

Precision

Precision

	Context-Sensitive	**Context-Insensitive**
Flow-Sensitive	Landi, Ryder 92 Choi et al. 93 Emami et al. 94 Reps et al. 95 Hind et al. 99 Kahlon 08	Zheng 98 Hardekopf, Lin 09
Flow-insensitive	Liang, Harrold 99 Whaley, Lam 04 Zhu, Calman 04 Lattner et al. 07	Andersen 94 Steensgaard 96 Shapiro, Horwitz 97 Fahndrich et al. 98 Das 00 Rountev, Chandra 00 Berndl et al. 03 Hardekopf, Lin 07 Pereira, Berlin 09 Mendez-Lojo 10
Surveys	Hind, Pioli 00 Qiang, Wu 06	