Efficient and Precise Points-to Analysis:
Modeling the Heap by Merging Equivalent Automata

Tian Tan, Yue Li and Jingling Xue

PLDI 2017
June, 2017

A New
Points-to Analysis Technique
for
Object-Oriented Programs

Points-to Analysis

- Determines
  - “which objects a variable can point to?”

Uses of Points-to Analysis

Clients

- Security analysis
- Bug detection
- Compiler optimization
- Program verification
- Program understanding
- …

Tools

Soot DroidSafe

Doop Chord

WALA
T. J. WATSON LIBRARIES FOR ANALYSIS

…

Uses of Points-to Analysis

Clients

- Security analysis
- Bug detection
- Compiler optimization
- Program verification
- Program understanding
- …

Tools

- Soot
- DroidSafe
- Doop
- Chord
- WALA — T. J. Watson Libraries for Analysis
- …

Call Graph

Existing Call Graph Construction

- On-the-fly construction
  (run with points-to analysis)
  - Precise
  - Inefficient

Existing Call Graph Construction

- On-the-fly construction
  (run with points-to analysis)
  - Precise
  - Inefficient

- 3-object-sensitive points-to analysis
  - Very precise
  - Adopted by, e.g., DOOP DroidSafe Chord

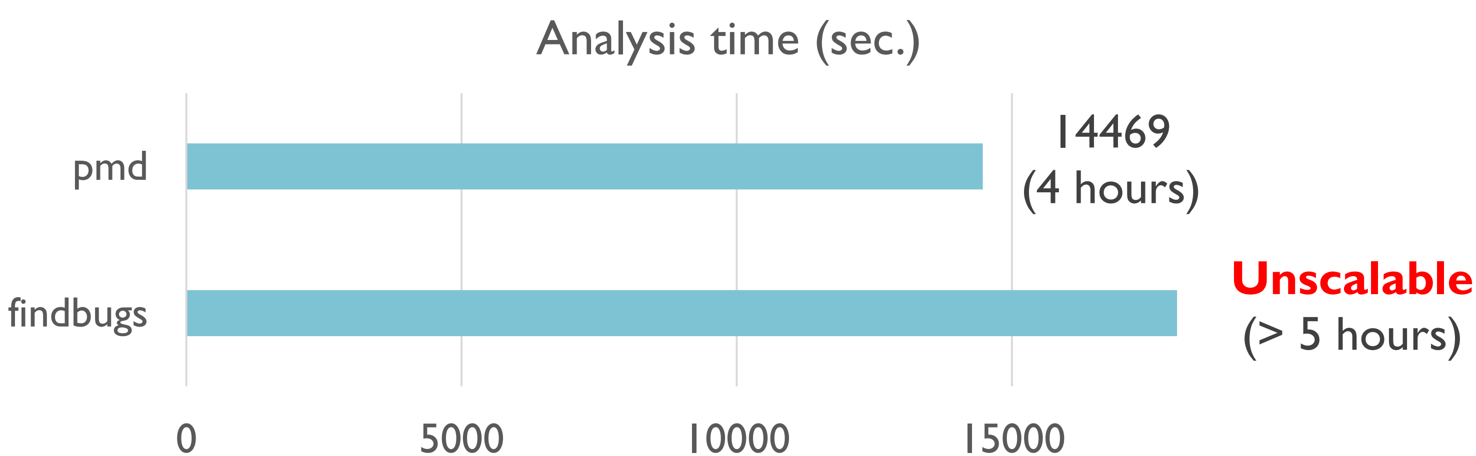
3-Object-Sensitive Points-to Analysis

- Analyze Java programs
  - Intel Xeon E5 3.70GHz, 128GB of memory
  - Time budget: 5 hours (18000 secs)

DOOP

3-Object-Sensitive Points-to Analysis

- Analyze Java programs
  - Intel Xeon E5 3.70GHz, 128GB of memory
  - Time budget: 5 hours (18000 secs)



Two Mainstreams of Points-to Analysis Techniques

- Model control-flow

- Model data-flow

Two Mainstreams of Points-to Analysis Techniques

- Model control-flow
  - Context-sensitivity
    - Call-site-sensitivity (PLDI'04, PLDI'06)
    - Object-sensitivity (ISSTA'02, TOSEM'05, SAS'16)
    - Type-sensitivity (POPL'11)
    - …
- Model data-flow

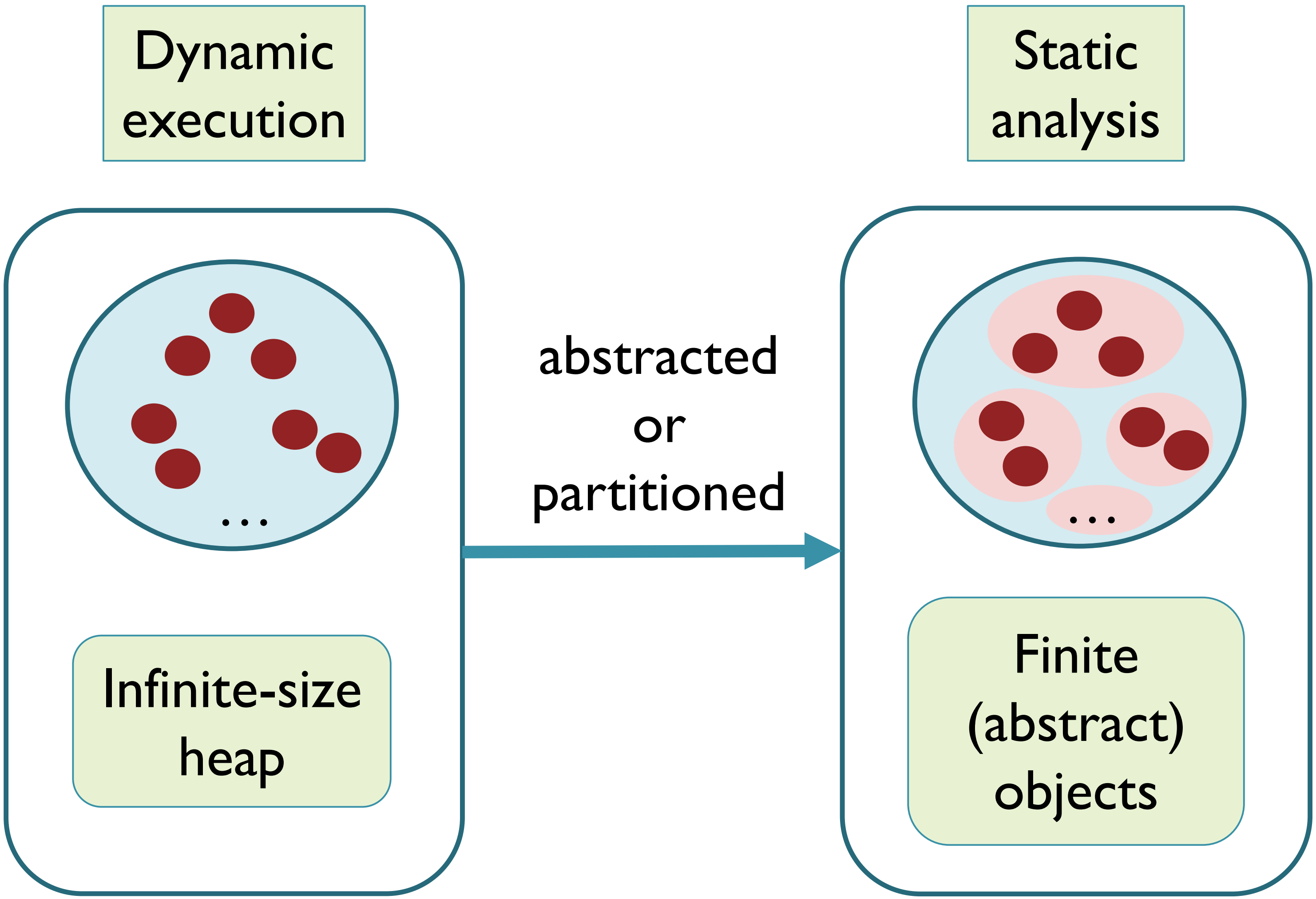
Two Mainstreams of Points-to Analysis Techniques

- Model control-flow
  - Context-sensitivity
    - Call-site-sensitivity (PLDI'04, PLDI'06)
    - Object-sensitivity (ISSTA'02, TOSEM'05, SAS'16)
    - Type-sensitivity (POPL'11)
    - …
- Model data-flow
  - Heap abstraction
    - Allocation-site abstraction
    - Type-based abstraction
    - …

Two Mainstreams of Points-to Analysis Techniques

- Model control-flow
  - Context-sensitivity
    - Call-site-sensitivity (PLDI'04, PLDI'06)
    - Object-sensitivity (ISSTA'02, TOSEM'05, SAS'16)
    - Type-sensitivity (POPL'11)
    - …
- Model data-flow
  - Heap abstraction
    - Allocation-site abstraction
    - Type-based abstraction
    - …

Heap Abstraction



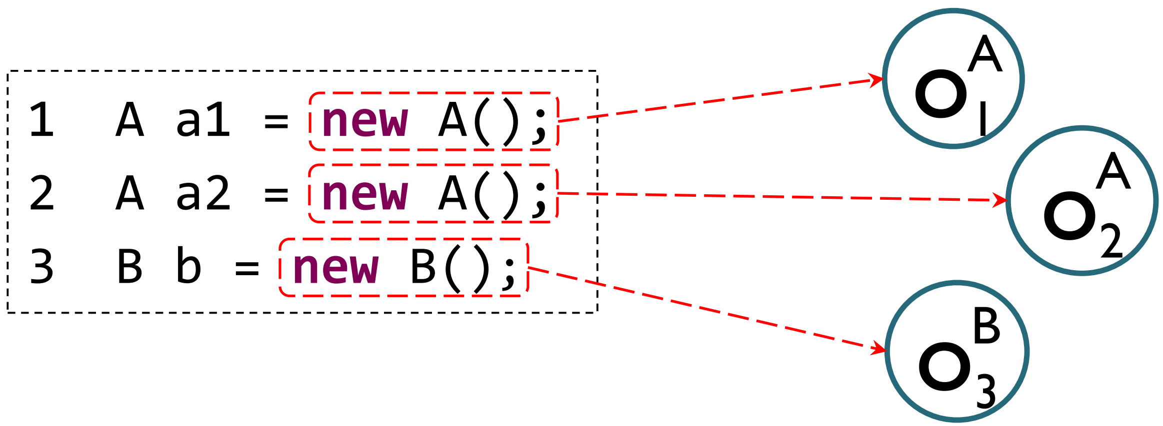
Allocation-Site Abstraction

- One object per **allocation site**

```
A a1 = new A();
A a2 = new A();
B b = new B();
```

Allocation-Site Abstraction

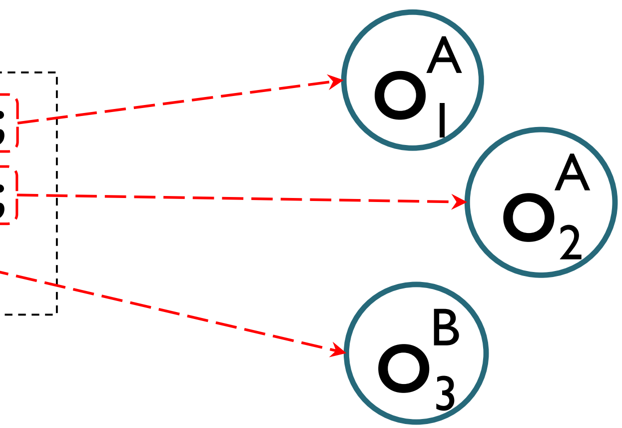
- One object per **allocation site**



Allocation-Site Abstraction

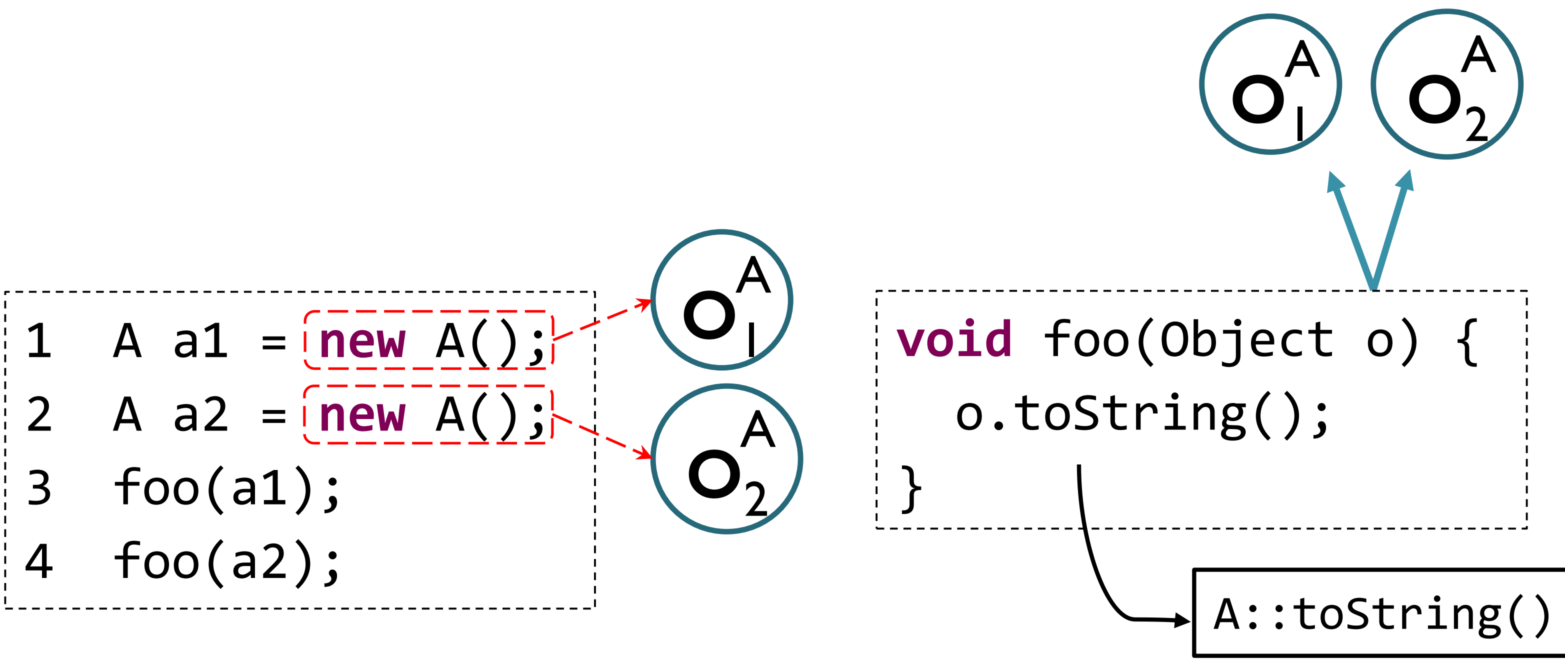
- One object per **allocation site**
  - Adopted by **all** mainstream points-to analyses

```
A a1 = new A();
A a2 = new A();
B b = new B();
```



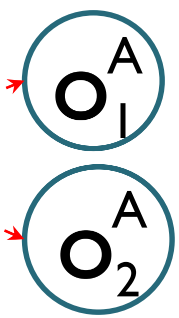
Allocation-Site Abstraction

- **Over-partition** for call graph construction

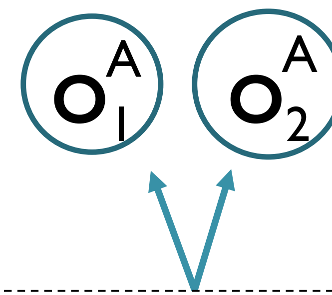


Allocation-Site Abstraction

- **Over-partition** for type-dependent clients
  - Call graph construction
  - Devirtualization
  - May-fail casting
  - …




```
A a1 = new A();
A a2 = new A();
foo(a1);
foo(a2);
```




```
void foo(Object o) {
  o.toString();
  A a = (A) o;
}
```

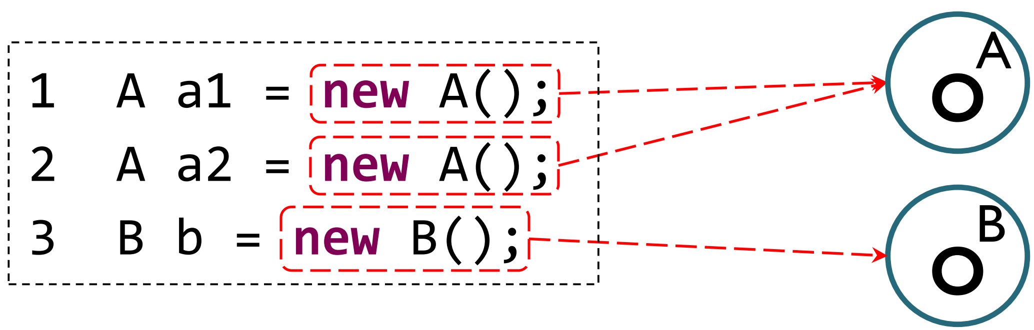
Type-Based Abstraction

- One object per **type**

```
A a1 = new A();
A a2 = new A();
B b = new B();
```

Type-Based Abstraction

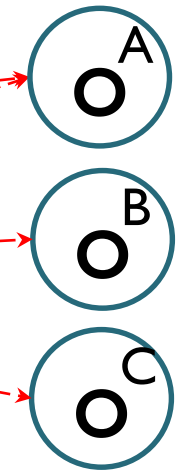
- One object per **type**



Type-Based Abstraction

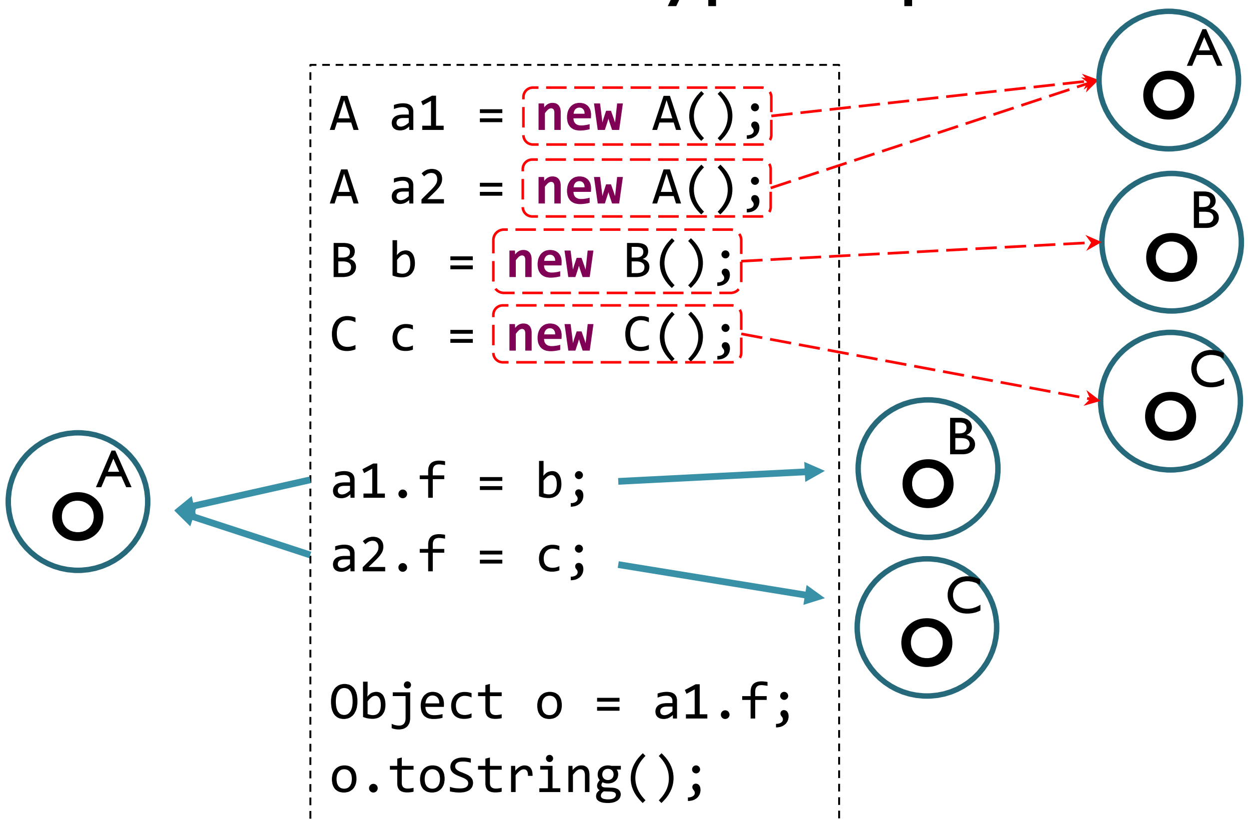
- Precision loss for type-dependent clients

```
A a1 = new A();
A a2 = new A();
B b = new B();
C c = new C();

a1.f = b;
a2.f = c;

Object o = a1.f;
o.toString();
```



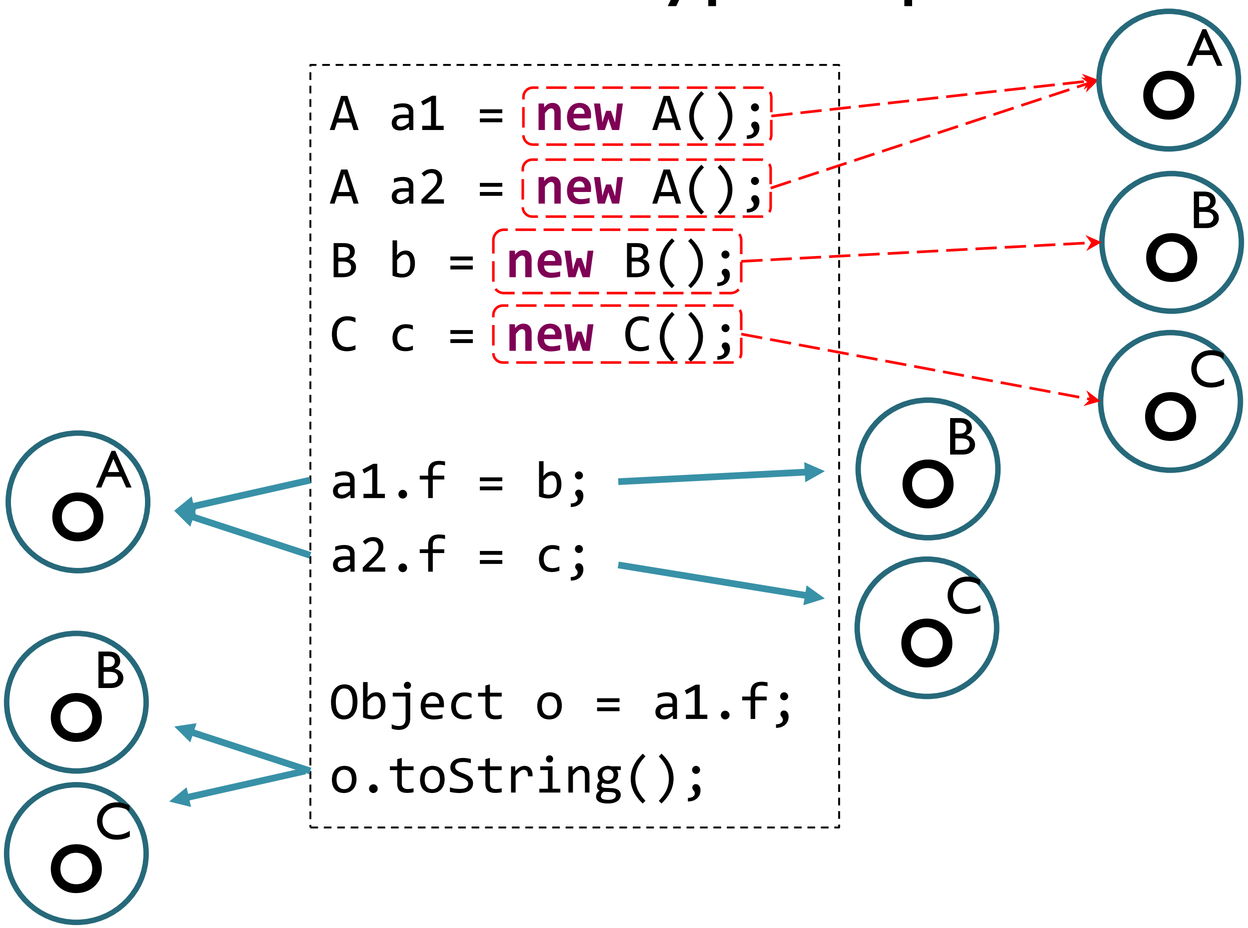
Type-Based Abstraction

- Precision loss for type-dependent clients



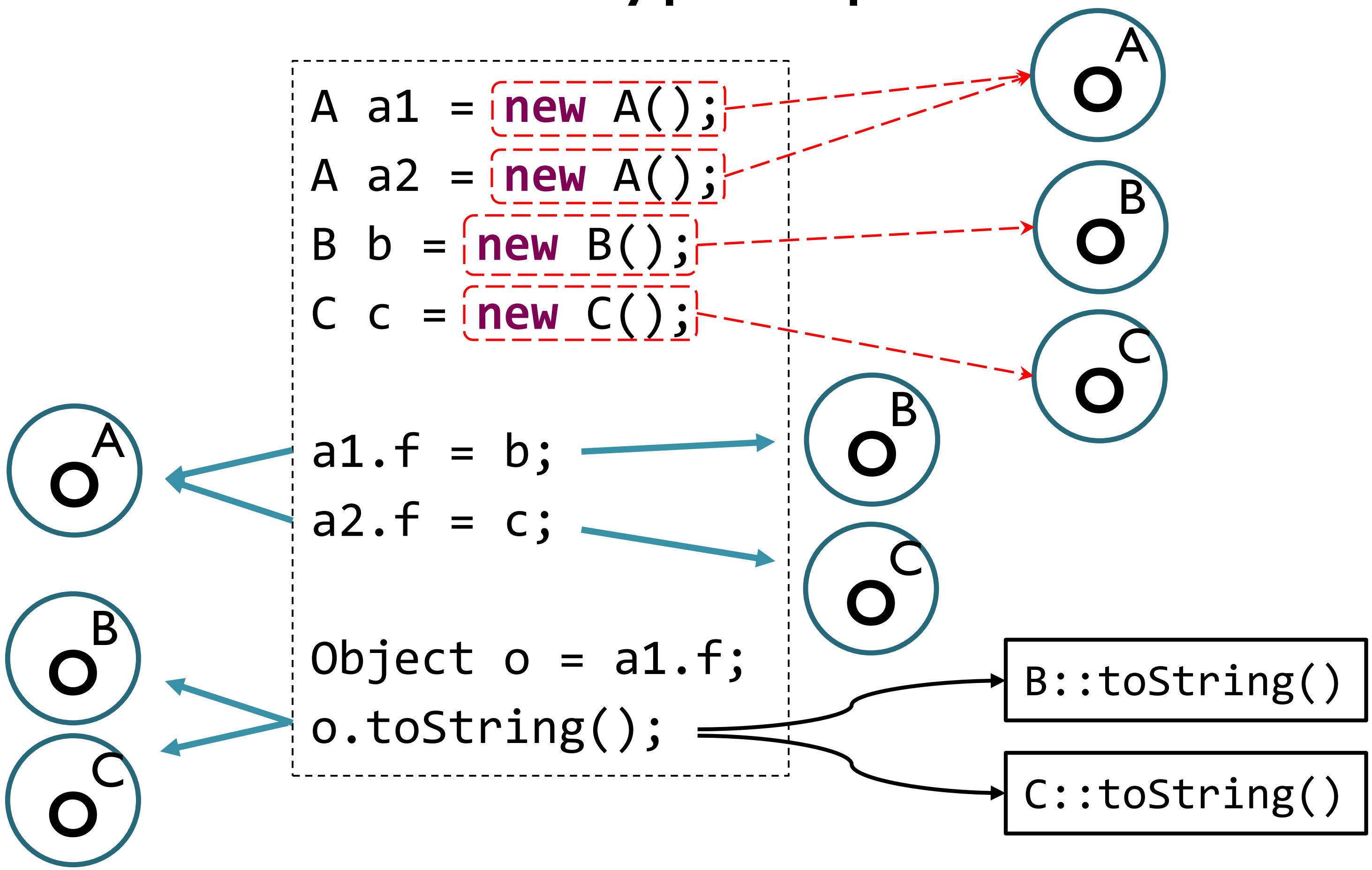
Type-Based Abstraction

- Precision loss for type-dependent clients



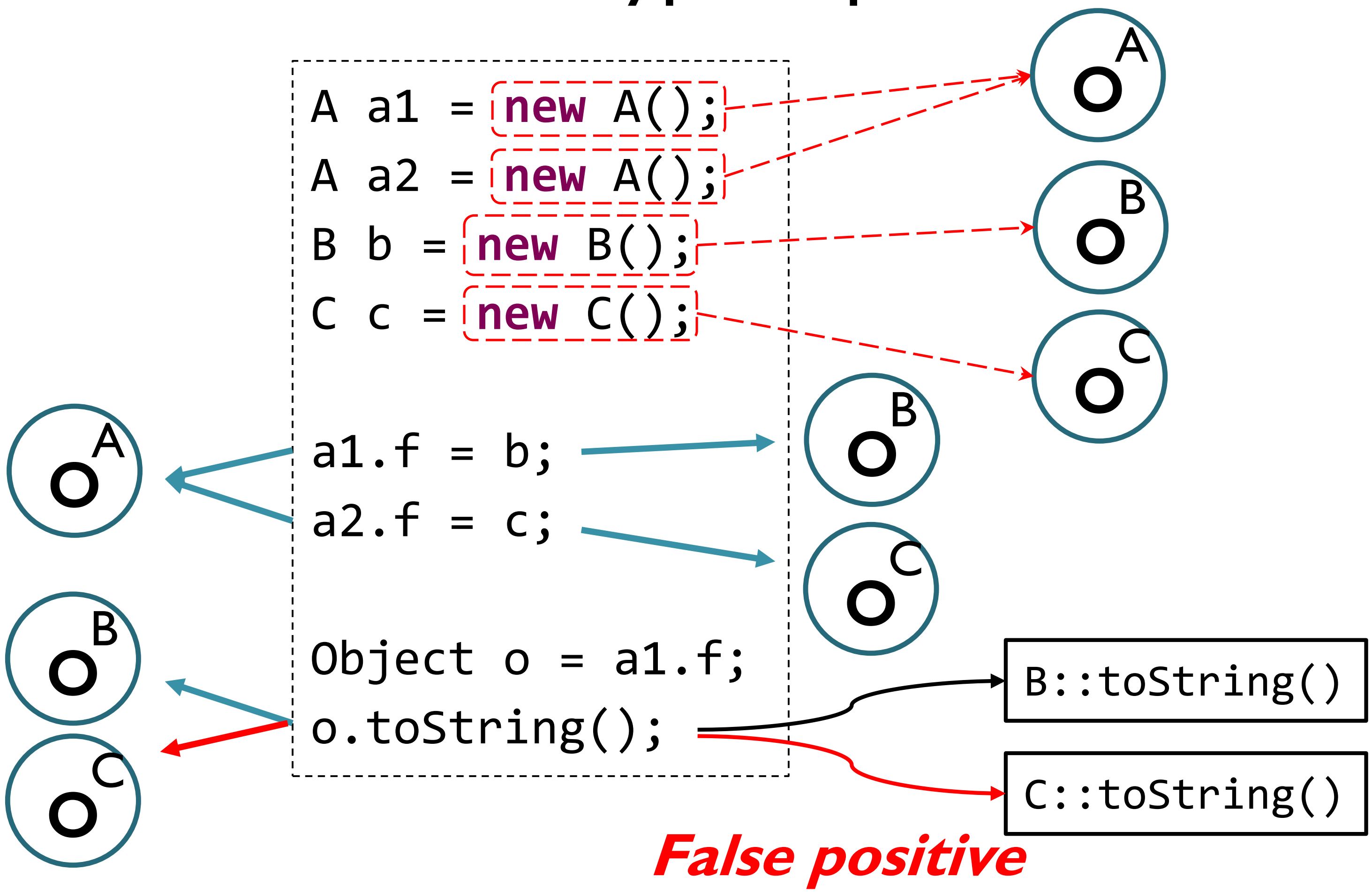
Type-Based Abstraction

- Precision loss for type-dependent clients



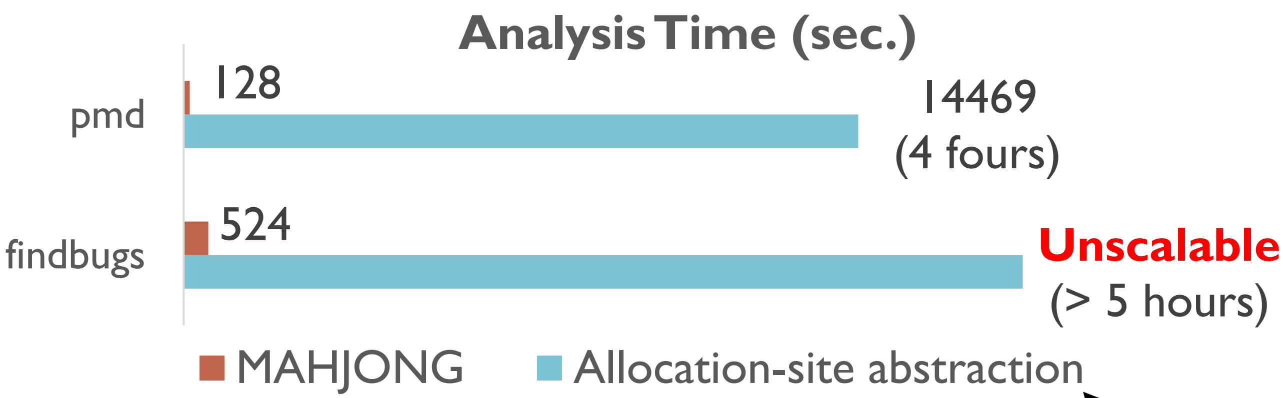
Type-Based Abstraction

- Precision loss for type-dependent clients



Our Goal:
Improve Efficiency
Preserve Precision

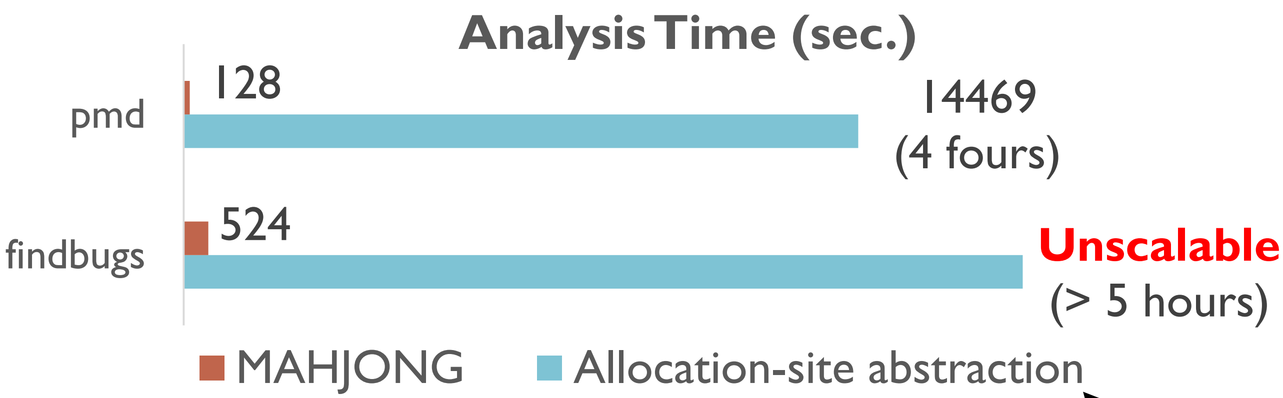
MAHJONG: A New Heap Abstraction



Adopted by **all** mainstream points-to analyses

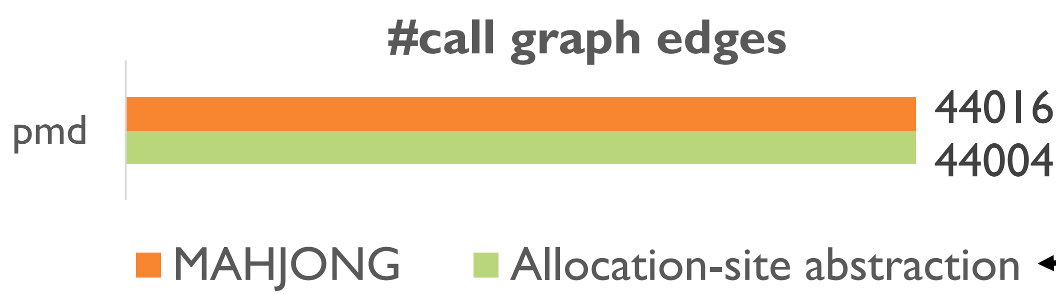
Improve Efficiency

MAHJONG: A New Heap Abstraction



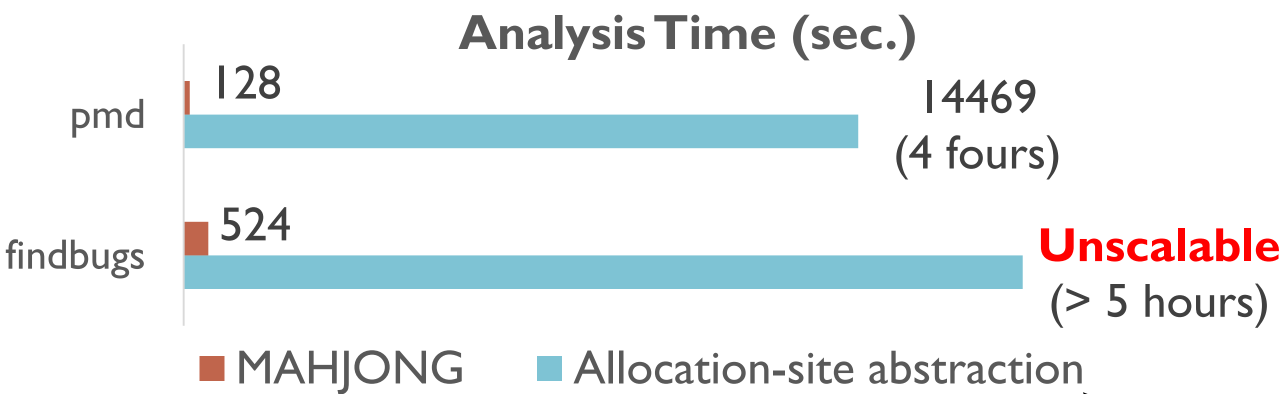
Improve Efficiency

Adopted by **all** mainstream points-to analyses



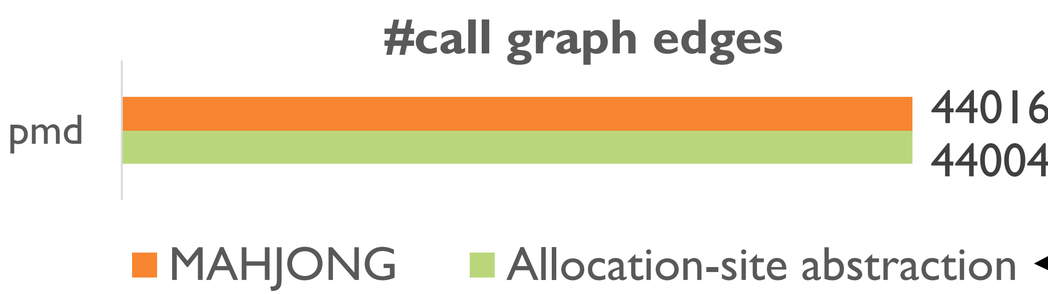
Preserve Precision

MAHJONG: A New Heap Abstraction



Improve Efficiency

Adopted by **all** mainstream points-to analyses



Preserve Precision

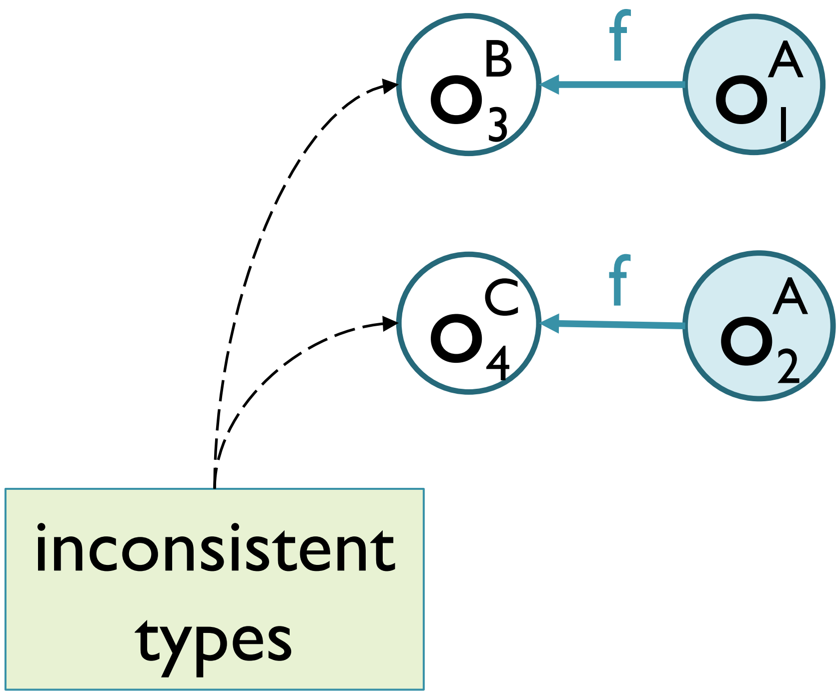
How?

Merging Objects —alleviate→ Over-Partition

Blindly Merging Objects —cause→ Precision Loss

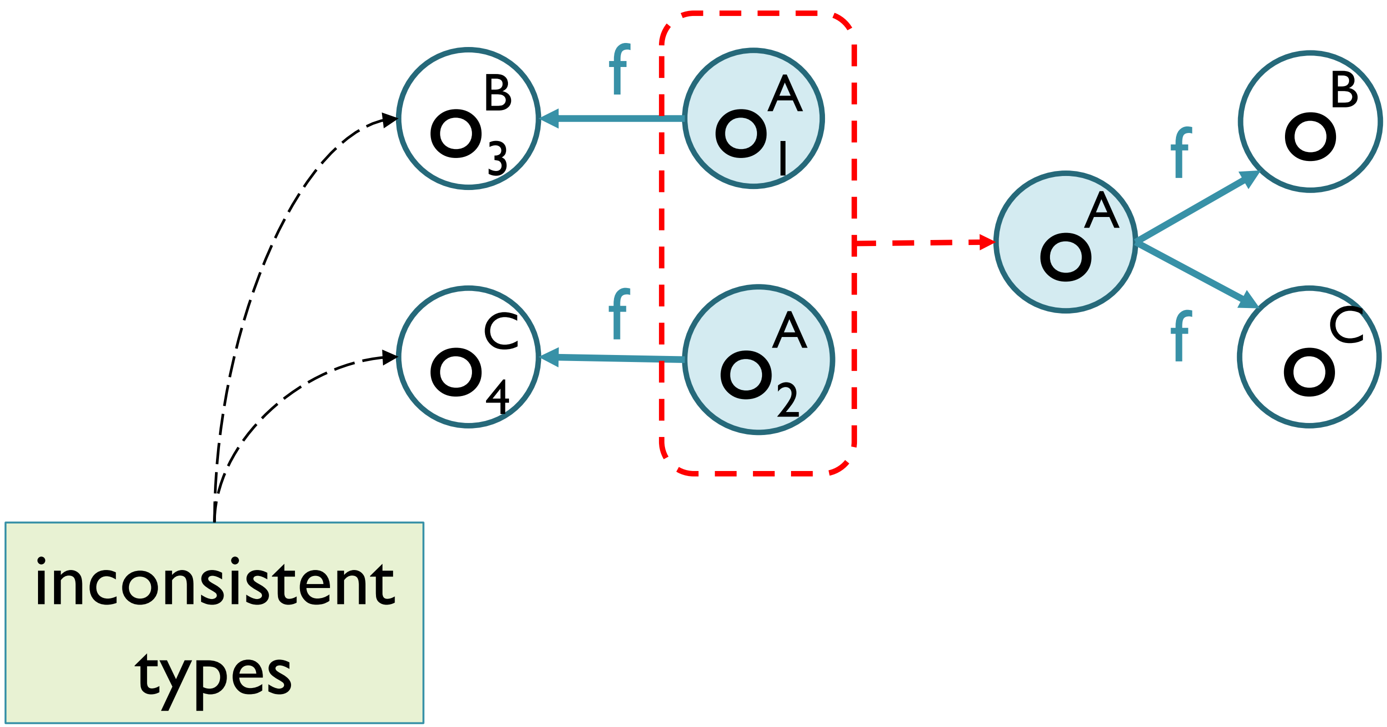
Merging Objects —alleviate→ Over-Partition

Blindly Merging Objects —cause→ Precision Loss



Merging Objects —alleviate→ Over-Partition

Blindly Merging Objects —cause→ Precision Loss



Type-Consistent Objects

- Definition

  O_i^T and O_j^T are type-consistent objects,

  if for every sequence of field names,

  $\overline{f} = f_1.f_2.\cdots.f_n$:

  $O_i^T.\overline{f}$ and $O_j^T.\overline{f}$ point to the objects of the same types.

Type-Consistent Objects

- Definition

  O_i^T and O_j^T are type-consistent objects, if for every sequence of field names,

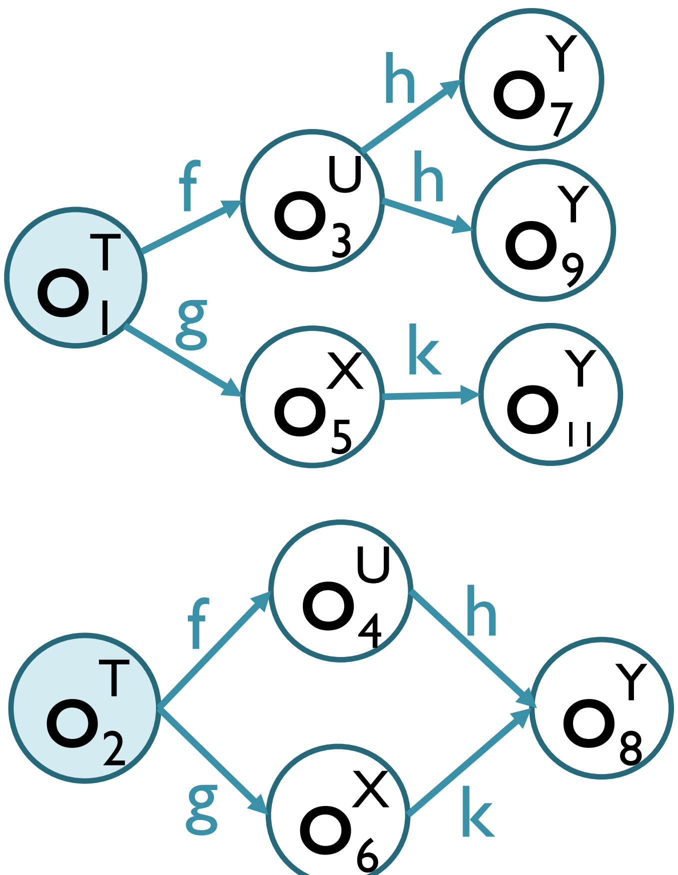
  $\overline{f} = f_1.f_2.\cdots.f_n$:

  $O_i^T.\overline{f}$ and $O_j^T.\overline{f}$ point to the objects of the same types.

MAHJONG only merges type-consistent objects

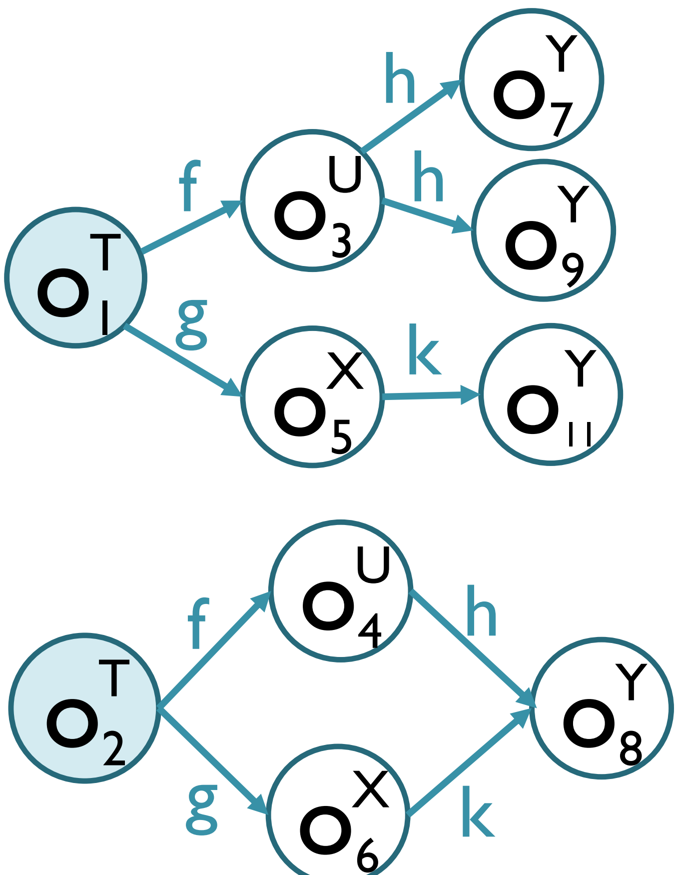
Type-Consistent Objects

- Example



Type-Consistent Objects

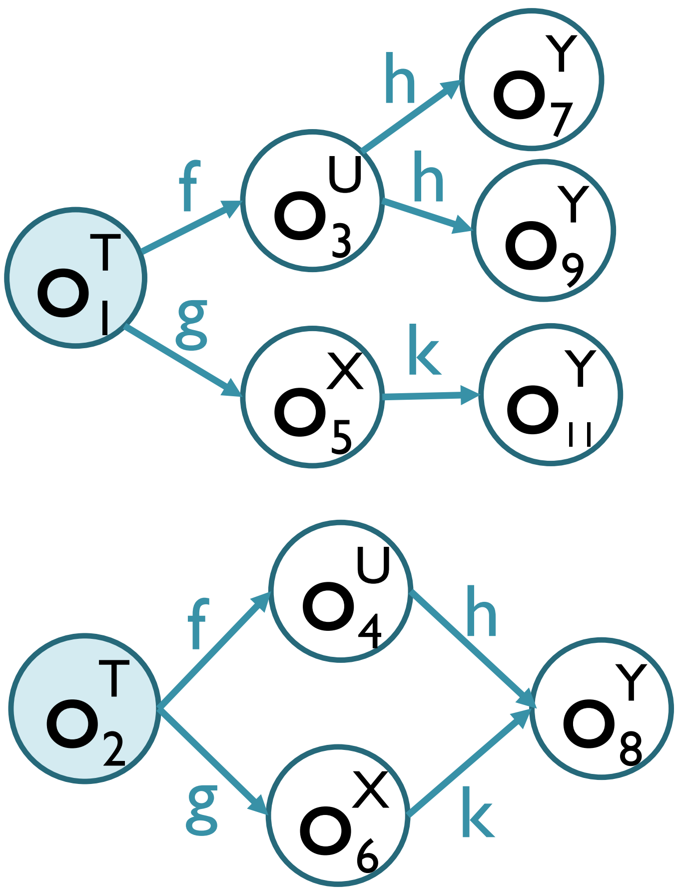
- Example



	o_1^T	o_2^T
.f	U	U
.f.h	Y	Y
.g	X	X
.g.k	Y	Y

Type-Consistent Objects

- Example



∵

	O_1^T	O_2^T
.f	U	U
.f.h	Y	Y
.g	X	X
.g.k	Y	Y

∴

O_1^T and O_2^T are type-consistent objects

How to Check Type-Consistency?

Our Solution: Sequential Automata

Check **Type-Consistency** of **Objects** ⟷ Test **Equivalence** of **Automata**

Sequential Automata

- 6-tuple (Q, Σ, δ, q_0, Γ, γ), where:
  - Q is a set of states
  - Σ is a set of input symbols
  - δ is the next-state map: $Q \times \Sigma \rightarrow \mathcal{P}(Q)$
  - q_0 is the initial state
  - Γ is a set of output symbols
  - γ is the output map: $Q \rightarrow \Gamma$

Check **Type-Consistency** of **Objects** ⟷ Test **Equivalence** of **Automata**

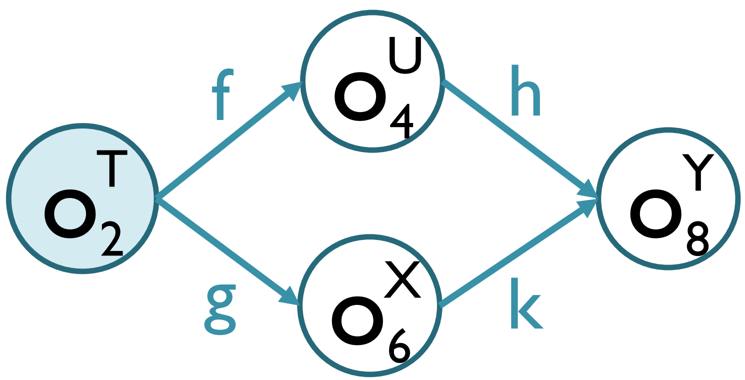
How?

Objects

- A set of objects
- A set of field names
- The field points-to map
- The object to be checked
- A set of types
- The object-to-type map

Automata

- Q: a set of states
- Σ: a set of input symbols
- δ: the next-state map
- q_0: the initial state
- Γ: a set of output symbols
- γ: the output map



Objects

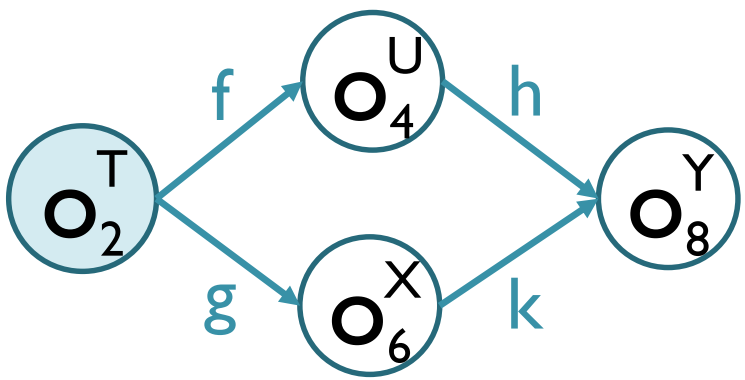
- **A set of objects**
- A set of field names
- The field points-to map
- The object to be checked
- A set of types
- The object-to-type map

Automata

- **Q: a set of states**
- Σ: a set of input symbols
- δ: the next-state map
- q_0: the initial state
- Γ: a set of output symbols
- γ: the output map

objects ↔ states

$O_2^T, O_4^U, O_6^X, O_8^Y$



Objects

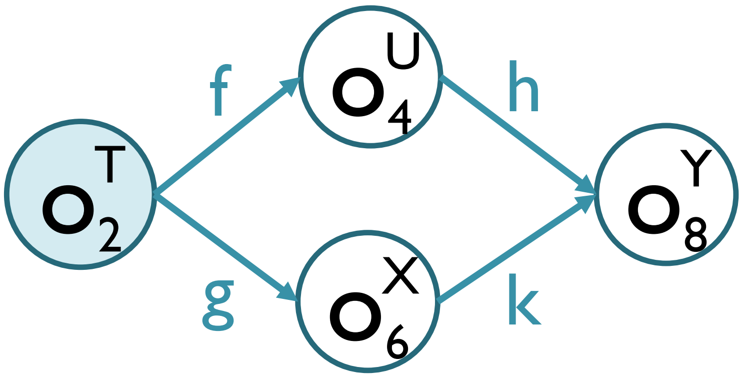
- A set of objects
- **A set of field names**
- The field points-to map
- The object to be checked
- A set of types
- The object-to-type map

Automata

- Q: a set of states
- **Σ: a set of input symbols**
- δ: the next-state map
- q_0: the initial state
- Γ: a set of output symbols
- γ: the output map

field names ↔ input symbols

f, g, h, k



Objects

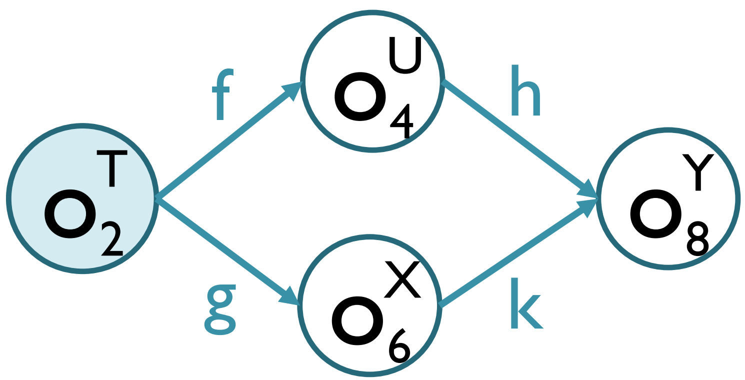
- A set of objects
- A set of field names
- **The field points-to map**
- The object to be checked
- A set of types
- The object-to-type map

Automata

- Q: a set of states
- Σ: a set of input symbols
- **δ: the next-state map**
- q_0: the initial state
- Γ: a set of output symbols
- γ: the output map

field points-to map ↔ next-state map

O_2^T	f	O_4^U
O_2^T	g	O_6^X
O_4^U	h	O_8^Y
O_6^X	k	O_8^Y



Objects

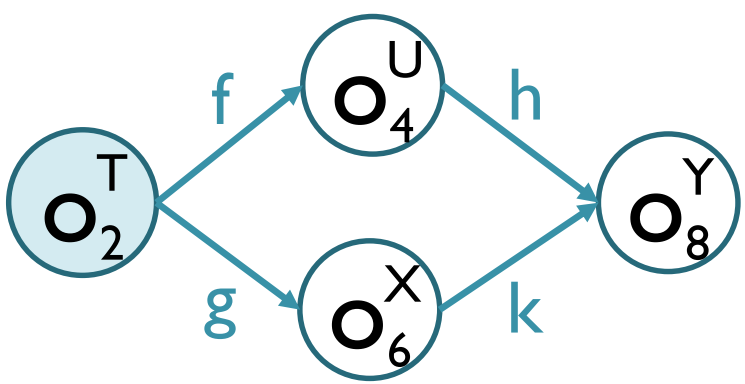
- A set of objects
- A set of field names
- The field points-to map
- **The object to be checked**
- A set of types
- The object-to-type map

Automata

- Q: a set of states
- Σ: a set of input symbols
- δ: the next-state map
- **q_0: the initial state**
- Γ: a set of output symbols
- γ: the output map

checked object ↔ initial state

O_2^T



Objects

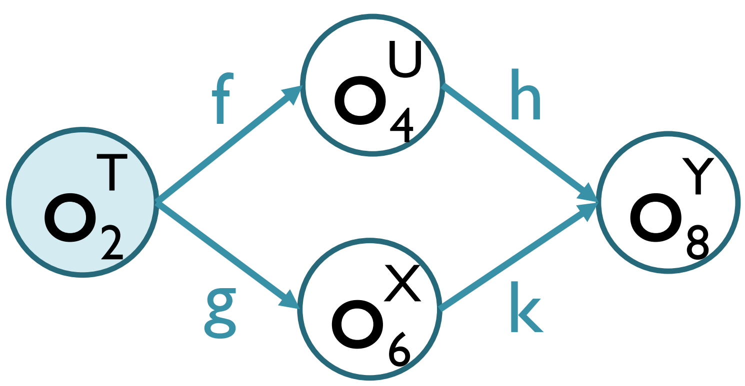
- A set of objects
- A set of field names
- The field points-to map
- The object to be checked
- **A set of types**
- The object-to-type map

Automata

- Q: a set of states
- Σ: a set of input symbols
- δ: the next-state map
- q_0: the initial state
- **Γ: a set of output symbols**
- γ: the output map

types ↔ output symbols

T, U, X, Y



Objects

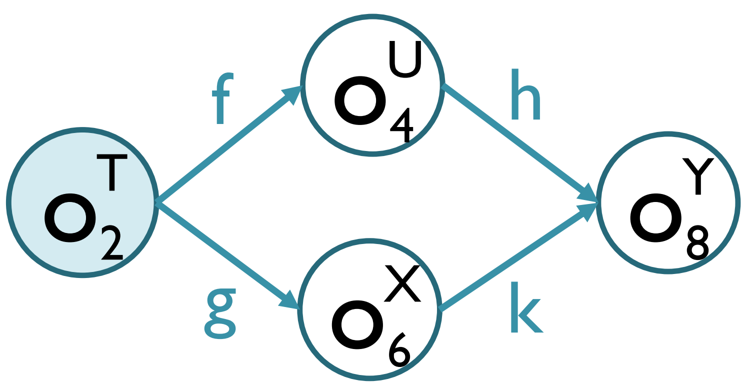
- A set of objects
- A set of field names
- The field points-to map
- The object to be checked
- A set of types
- **The object-to-type map**

Automata

- Q: a set of states
- Σ: a set of input symbols
- δ: the next-state map
- q_0: the initial state
- Γ: a set of output symbols
- **γ: the output map**

object-to-type map ↔ output map

O_2^T	T
O_4^U	U
O_6^X	X
O_8^Y	Y



Objects

- A set of objects
- A set of field names
- The field points-to map
- The object to be checked
- A set of types
- The object-to-type map

Automata

- Q: a set of states
- Σ: a set of input symbols
- δ: the next-state map
- q_0: the initial state
- Γ: a set of output symbols
- γ: the output map

Check **Type-Consistency** of **Objects** ⟷ Test **Equivalence** of **Automata**

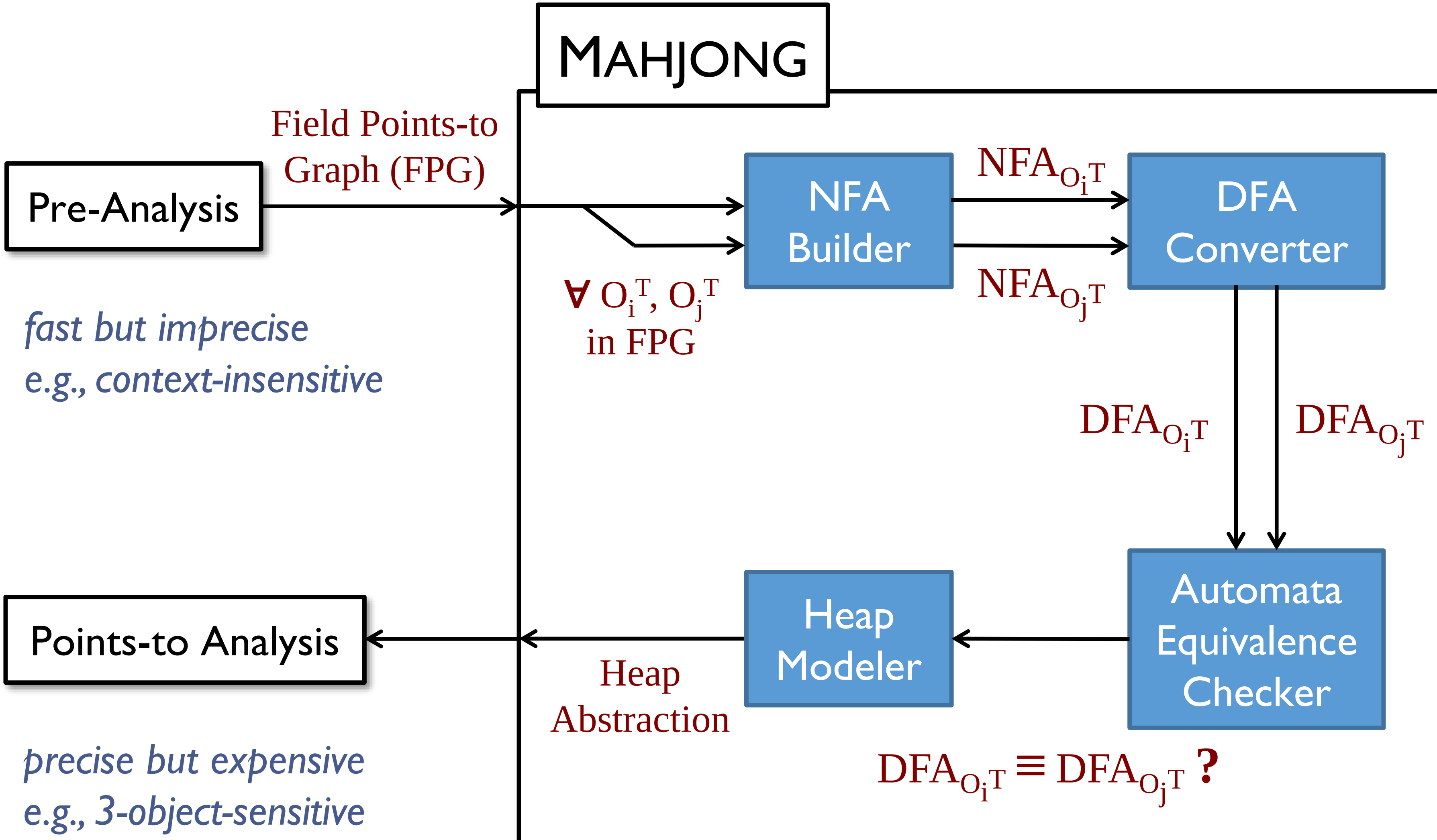
Test Equivalence of Automata

- Hopcroft-Karp algorithm*
  - Almost linear in terms of $|Q_{larger}|$
  - Q_{larger}: set of states of the larger automaton

* *J. E. Hopcroft and R. M. Karp, A linear algorithm for testing equivalence of finite automata, Technical Report 71-114, 1971*

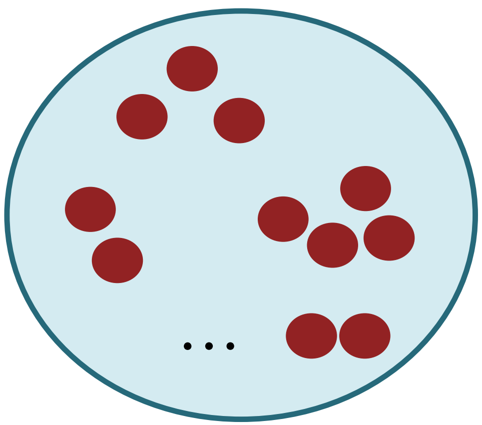
Methodology (MAHJONG)

Overview


MAHJONG
Pre-Analysis
Field Points-to Graph (FPG)
$\forall\ O_i^T, O_j^T$ in FPG
NFA Builder
$NFA_{O_i T}$
$NFA_{O_j T}$
DFA Converter
$DFA_{O_i T}$
$DFA_{O_j T}$
Automata Equivalence Checker
$DFA_{O_i T} \equiv DFA_{O_j T}$?
Heap Modeler
Heap Abstraction
Points-to Analysis
fast but imprecise
e.g., context-insensitive
precise but expensive
e.g., 3-object-sensitive

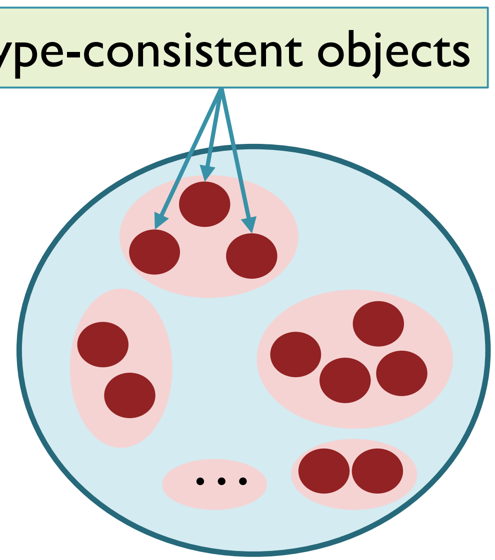
Original

- Allocation-site heap abstraction



New

- MAHJONG heap abstraction



Working with Points-to Analysis

Implementation MAHJONG

- 1500 LOC of Java in total
- Integrated with Doop
- Can also be easily integrated to other points-to analysis frameworks

Evaluation

Evaluation - Research Questions

- RQ1: MAHJONG's effectiveness as a pre-analysis
- RQ2: MAHJONG-based points-to analysis

RQ1: MAHJONG's Effectiveness as A Pre-Analysis

- Efficiency
  - Is MAHJONG lightweight for large programs?
- Heap partitioning
  - Can MAHJONG avoid heap over-partition?

Pre-Analysis: Efficiency

	antlr	fop	luindex	pmd	chart	checkstyle	xalan	bloat	lusearch	JPC	findbugs	eclipse
CI	44.1	34.7	26.2	44.8	37.7	89.6	66.6	38.7	41.4	58.9	90.6	174.1
FPG	1.3	0.7	0.8	1.4	2.4	2.3	3.0	1.2	0.8	2.1	4.6	15.5
MAHJONG	1.3	1.1	1.1	1.5	1.9	4.0	3.1	1.7	1.0	4.5	3.2	21.4
Total	46.7	36.5	28.1	47.7	42.0	95.9	72.7	41.6	43.2	65.5	98.4	211.0

CI: Context-Insensitive points-to analysis
FPG: Read Field Points-to Graph
MAHJONG: Check automata equivalence, build heap abstraction

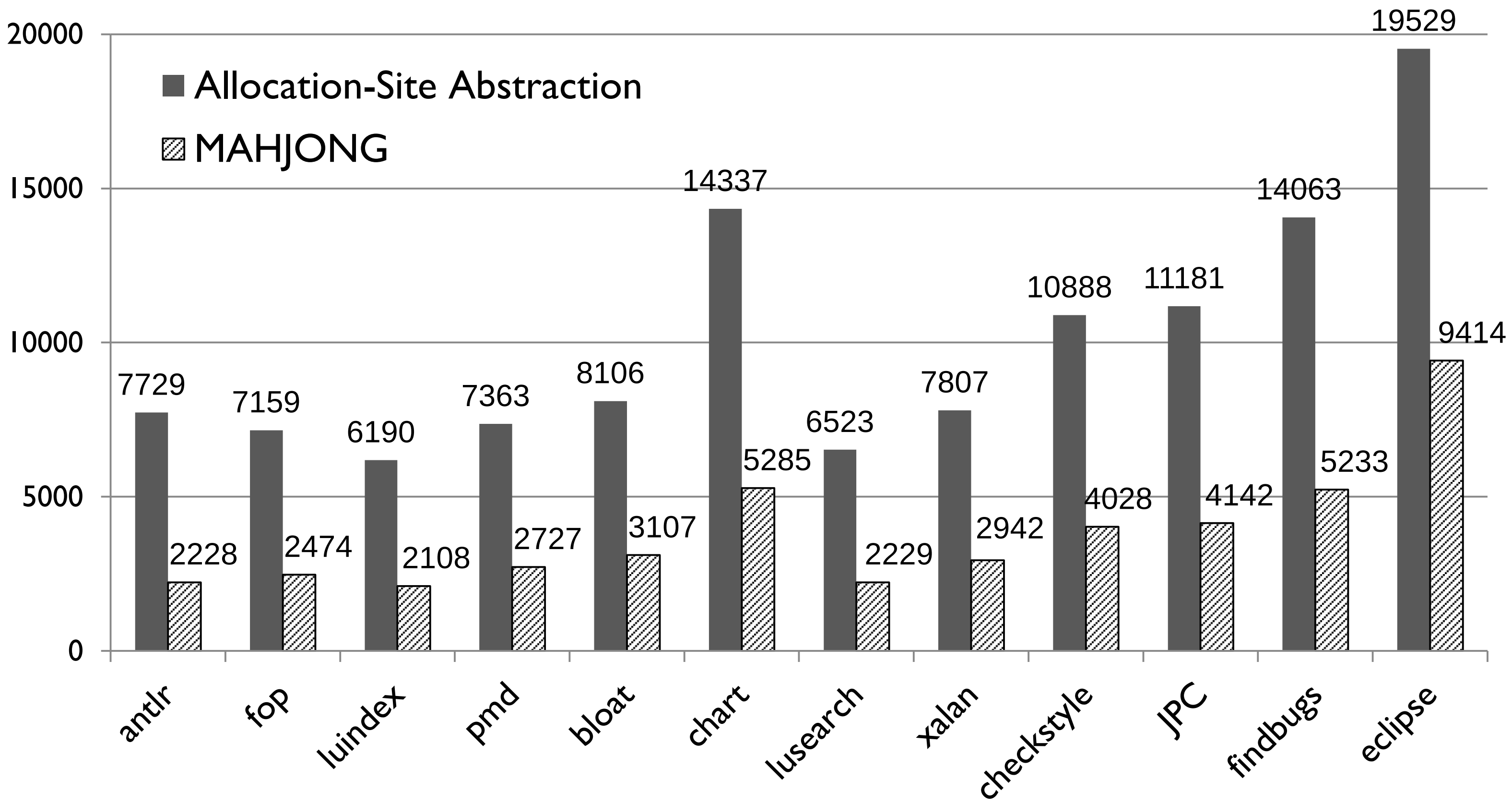
In total: 1 minute

MAHJONG itself: 3.8 seconds

Each program (on average)

Pre-Analysis: Heap Partition

Number of abstract objects created by the allocation-site abstraction and MAHJONG



Average reduction: 62%

RQ2: MAHJONG-Based Points-to Analysis

- Efficiency
  - Can MAHJONG accelerate points-to analysis?
- Precision
  - Can MAHJONG preserve precision for type-dependent clients?

Evaluated Points-to Analyses

- 5 mainstream context-sensitive points-to analyses:

DOOP

1. 2-call-site-sensitive analysis
2. 2-type-sensitive analysis
3. 3-type-sensitive analysis
4. 2-object-sensitive analysis
5. 3-object-sensitive analysis

- Time budget: 5 hours

Evaluated Clients

- Call graph construction
- Devirtualization
- May-fail casting

MAHJONG-Base Points-to Analysis: Results

- Efficiency

  Most precise
  (3-object-sensitive)
  Speedup: **131X**

- Precision

  Call graph: -0.02%
  Devirtualization: -0.29%
  May-fail casting: -0%

MAHJONG-Base Points-to Analysis: Results

- Efficiency

Most precise
(3-object-sensitive)
Speedup: 131X

On average
Speedup: 15X

- Precision

Call graph: -0.02%
Devirtualization: -0.29%
May-fail casting: -0%

Call graph: -0.02%
Devirtualization: -0.18%
May-fail casting: -0.03%

MAHJONG-Base Points-to Analysis: Results

- Efficiency

  Most precise (3-object-sensitive)
  Speedup: 131X

  On average
  Speedup: 15X

- Precision

  Call graph: -0.02%
  Devirtualization: -0.29%
  May-fail casting: -0%

  Call graph: -0.02%
  Devirtualization: -0.18%
  May-fail casting: -0.03%

For `checkstyle`, `xalan`, `lusearch`, `JPC`, `findbugs`
3-object-sensitive analysis:

- **without** MAHJONG, unscalable (> 5 hours)
- **with** MAHJONG, finish in 1min ~ 84 mins (33 minutes on average)

Conclusion

- MAHJONG
  - Improve significantly the efficiency of different point-to analyses
    - Call-site-, object- and type-sensitivity
  - Preserve almost the same precision for type-dependent clients

- Direct impact
  - Benefit many program analyses where call graphs are required

Thank you!