Scaling Java Points-To Analysis Using SPARK

(Soot Pointer Analysis Research Kit)

Ondřej Lhoták and Laurie Hendren

Sable Research Group
McGill University
April 8th, 2003

Problems

- Implementing a points-to analysis to handle the details of Java
  - is a lot of work.
  - is difficult to do correctly.
- Research done on disparate implementations is often incomparable.

Objectives

- Develop a flexible, efficient framework for experimenting with variations in Java points-to analyses
- Demonstrate its usefulness with an empirical comparison of precision and efficiency of some of these variations

Outline

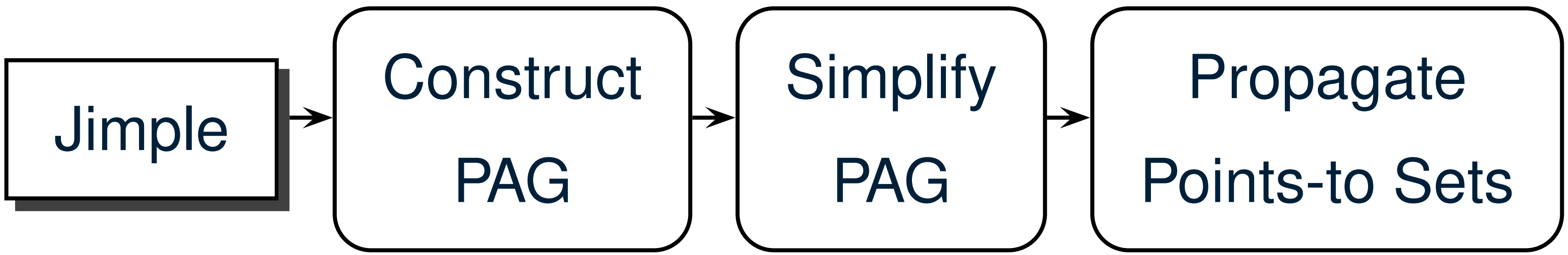
- **Spark overview**
- Empirical study
- Overall performance
- Uses of Spark
- Conclusion

Spark overview

- Part of Soot bytecode transformation and annotation framework [CC 00] [CC 01]
- Initial representation is Soot's Jimple
  - Typed [SAS 00]
  - Three-address (only simple operations)
- Spark internal representation is Pointer Assignment Graph (PAG)
  - Nodes for variables, allocation sites, field references
  - Edges representing subset constraints

Spark overview

- Spark proceeds in three steps:



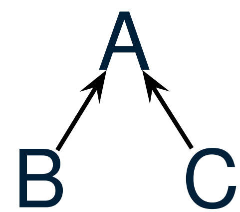
- Analysis variations expressed by building different PAGs for the same code
- This talk concentrates on flow-insensitive, subset-based variations

Empirical study

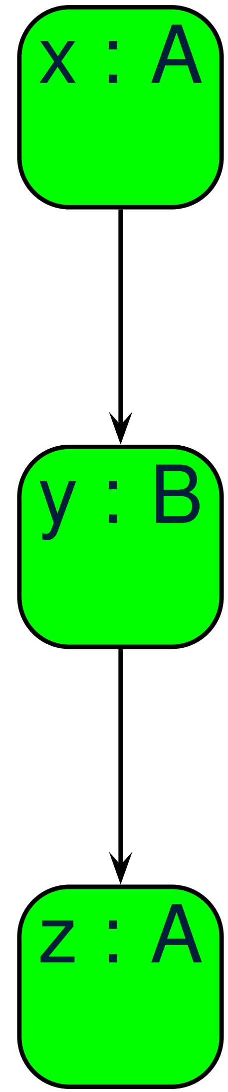
- **Factors affecting precision**
  - **Enforcing declared types**
  - Field reference representation
  - Call graph construction
- Factors affecting only efficiency
  - Pointer assignment graph simplification
  - Set implementation
  - Propagation algorithms

Declared types: ignore

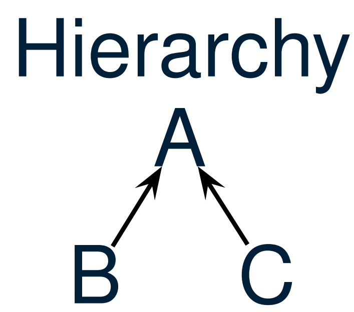
Hierarchy




```
A x, z;
B y;
x = new A();
y = new B();
y = (B) x;
z = y;
```



Declared types: ignore



A x, z;

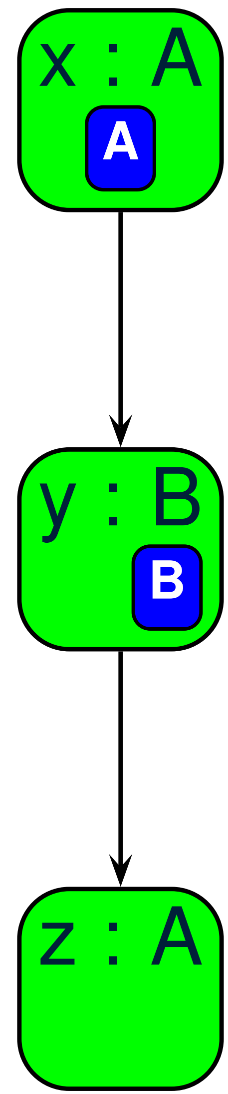
B y;

x = new A();

y = new B();

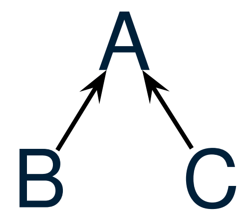
y = (B) x;

z = y;



Declared types: ignore

Hierarchy



A x, z;

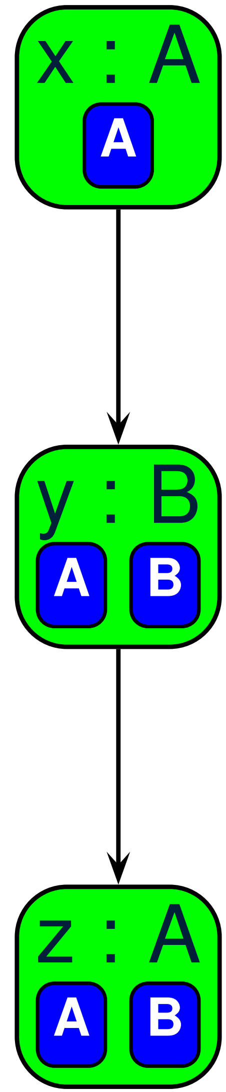
B y;

x = new A();

y = new B();

y = (B) x;

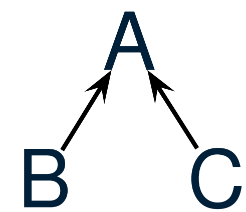
z = y;



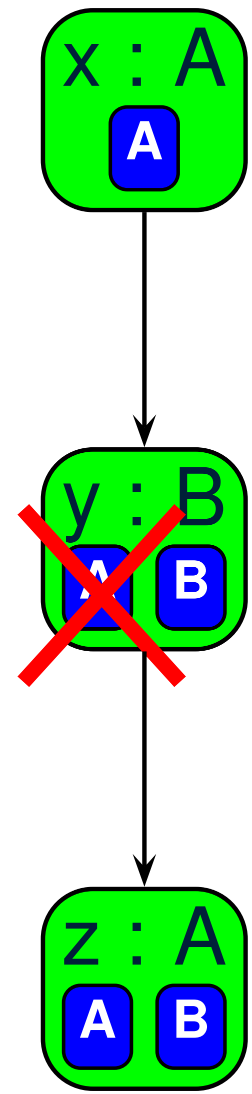
Declared types: enforce after analysis

[OOPSLA 00] [Rountev,Milanova,Ryder 01]

Hierarchy

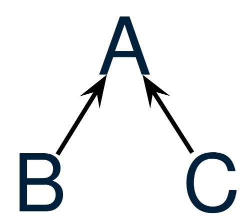



```
A x, z;
B y;
x = new A();
y = new B();
y = (B) x;
z = y;
```



Declared types: enforce during analysis

Hierarchy



A x, z;

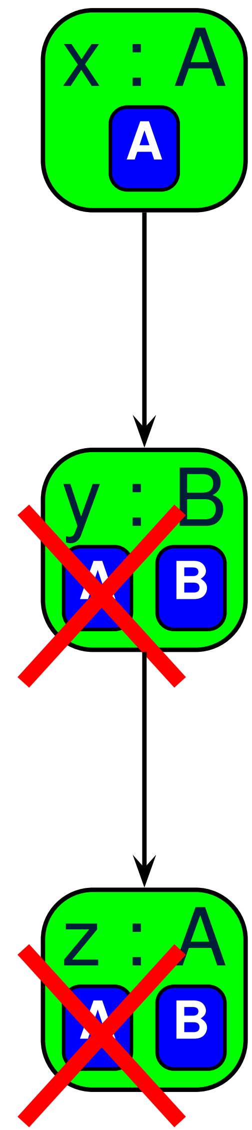
B y;

x = new A();

y = new B();

y = (B) x;

z = y;



Enforcing declared types

- ignoring types produces many large sets (> 1000 elements) of spurious points-to relationships
- in practice, enforcing types after analysis almost as precise as during analysis
- enforcing types during analysis prevents blowup during the analysis

ignore	slow	less precise
after analysis	slow	**more precise**
during analysis	**fast**	**more precise**

Empirical study

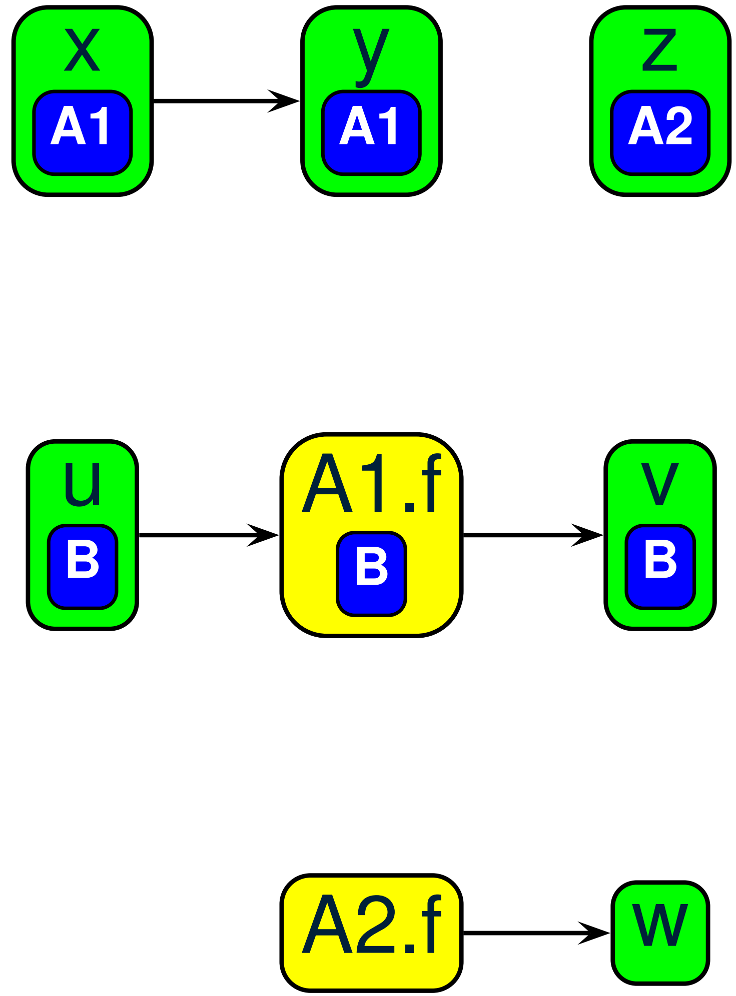
- **Factors affecting precision**
  - Enforcing declared types
  - **Field reference representation**
  - Call graph construction
- Factors affecting only efficiency
  - Pointer assignment graph simplification
  - Set implementation
  - Propagation algorithms

Field representation

- Field references can be represented in different ways:
  - field-sensitive distinguishes fields of different objects
  - field-based ignores the base object, grouping all objects having the field together

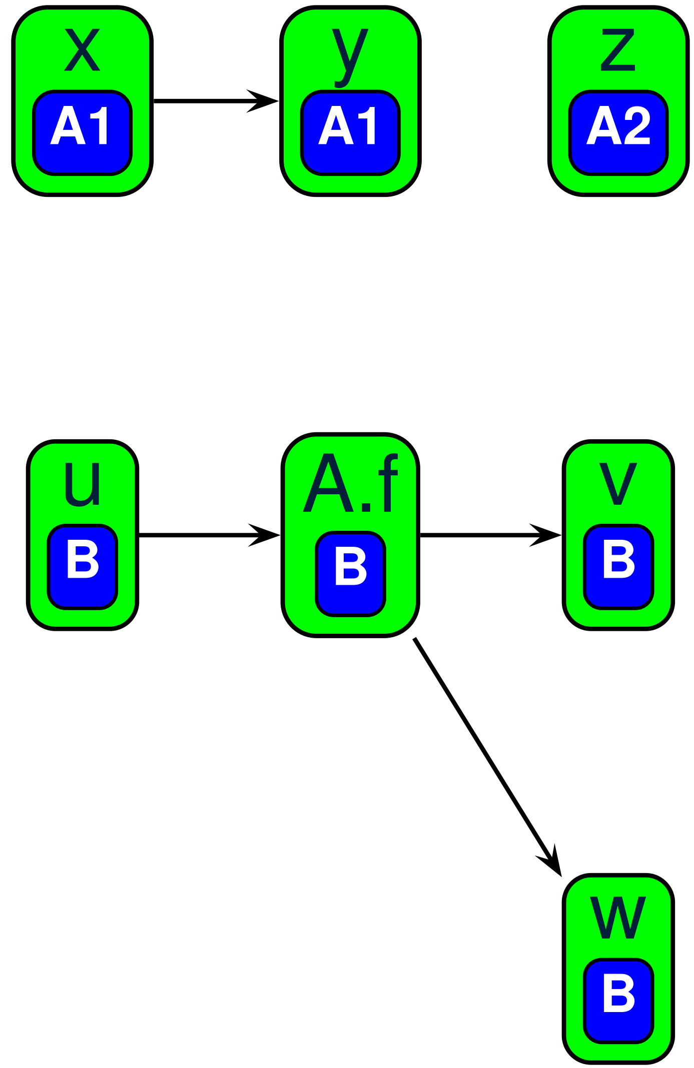
Field-sensitive representation

```
    A x, y, z;
    B u, v, w;
l1: x = new A();
    y = x;
l2: z = new A();
    u = new B();
    x.f = u;
    v = y.f;
    w = z.f;
```



Field-based representation

```
    A x, y, z;
    B u, v, w;
l1: x = new A();
    y = x;
l2: z = new A();
    u = new B();
    x.f = u;
    v = y.f;
    w = z.f;
```



Field representation

- Field-sensitive requires iterating
- Field-based less precise, but possible in a single iteration
- Clever propagation algorithm can make speed difference very small

field-based	**very fast**	less precise
field-sensitive	**almost as fast**	**more precise**

Empirical study

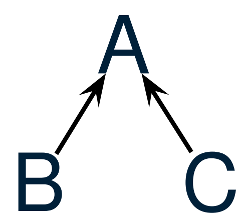
- **Factors affecting precision**
  - Enforcing declared types
  - Field reference representation
  - **Call graph construction**
- Factors affecting only efficiency
  - Pointer assignment graph simplification
  - Set implementation
  - Propagation algorithms

Call graph construction

- An approximation of the call graph is required for points-to analysis
- It can be built
  - ahead-of-time using an analysis such as Class Hierarchy Analysis
  - on-the-fly during the analysis as actual types of receivers are computed

Call graph construction: CHA

Hierarchy



class B

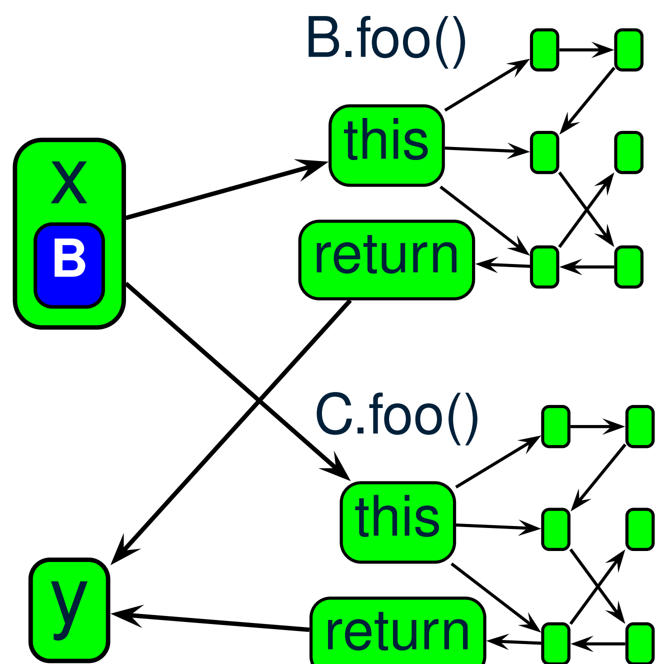
{ foo() { . . . } }

class C

{ foo() { . . . } }

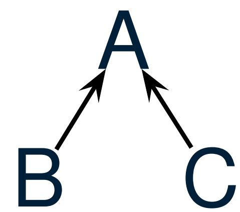
A x = new B();

A y = x.foo();



Call graph construction: on-the-fly

Hierarchy



class B

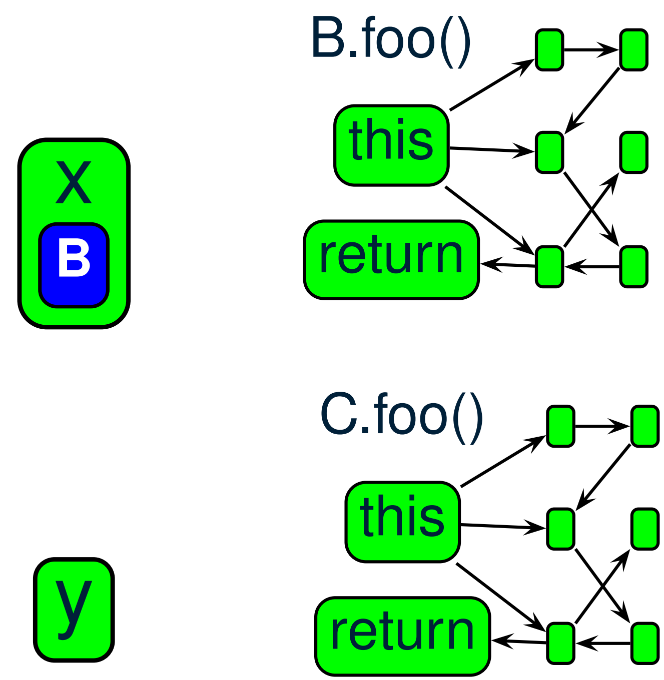
{ foo() { . . . } }

class C

{ foo() { . . . } }

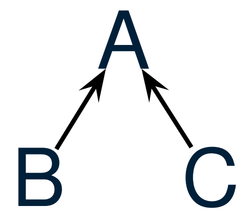
A x = new B();

A y = x.foo();



Call graph construction: on-the-fly

Hierarchy



class B

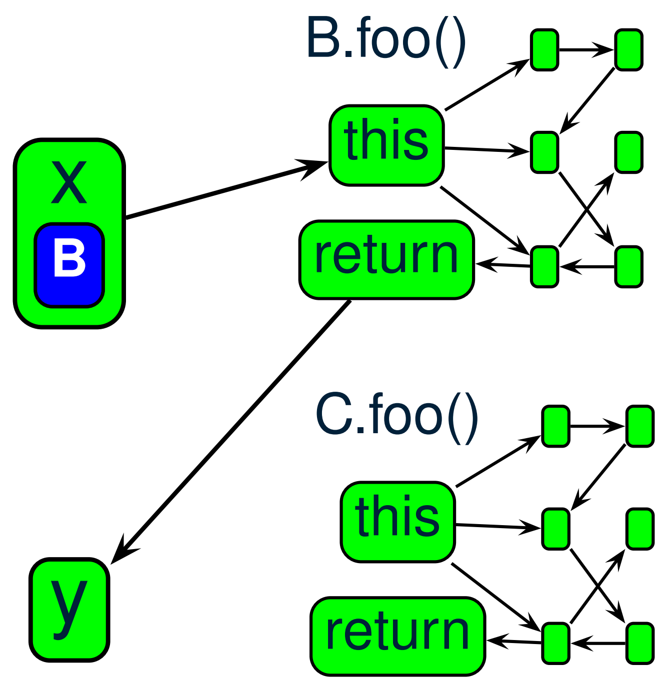
{ foo() { . . . } }

class C

{ foo() { . . . } }

A x = new B();

A y = x.foo();



Call graph construction

- Building call graph on-the-fly requires adding edges during propagation
  - requires more iteration
  - reduces simplification opportunities before propagation
- CHA call graph includes more spurious, unreachable methods than on-the-fly

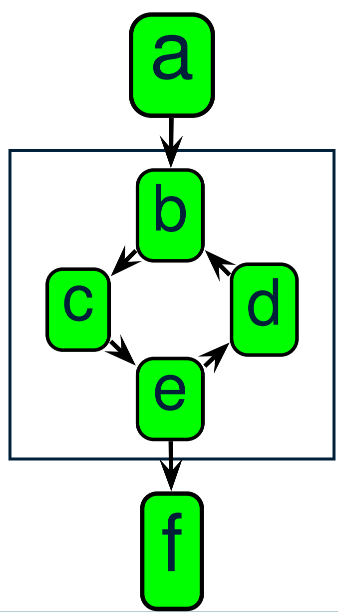
CHA	**fast**	less precise
on-the-fly	slow	**more precise**

Empirical study

- Factors affecting precision
  - Enforcing declared types
  - Field reference representation
  - Call graph construction
- **Factors affecting only efficiency**
  - **Pointer assignment graph simplification**
  - Set implementation
  - Propagation algorithms

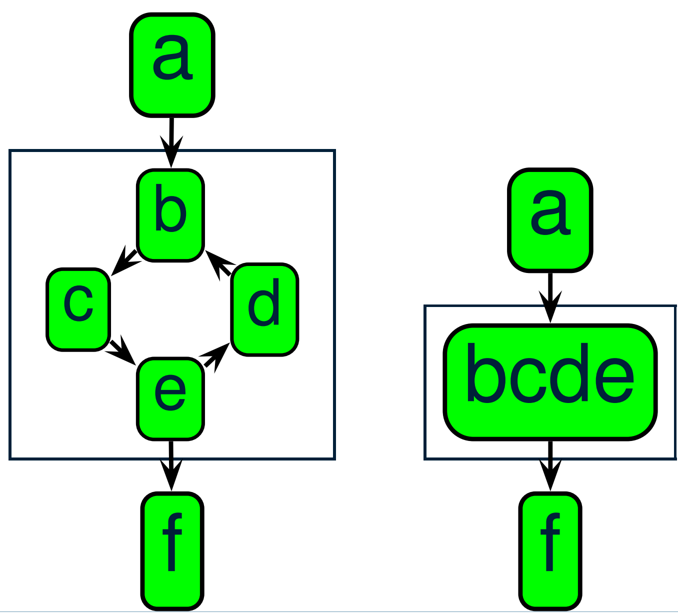
Pointer assignment graph simplification

- Groups of nodes can be merged
  [Rountev,Chandra 00]
  - strongly-connected components
  - single-entry subgraphs



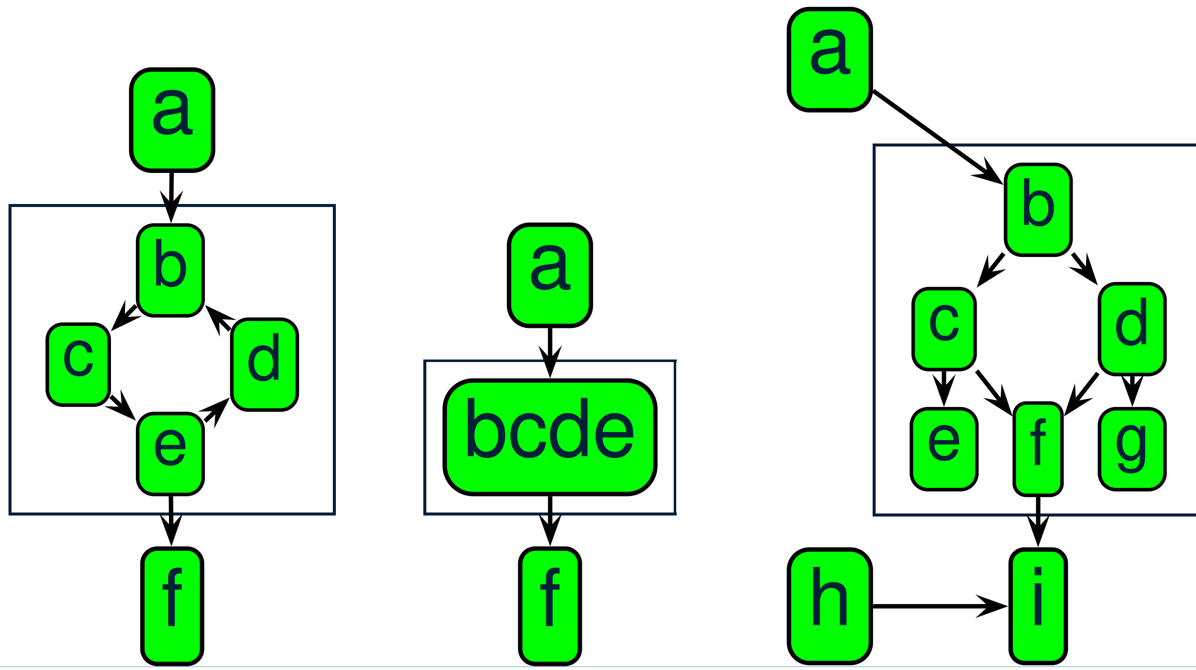
Pointer assignment graph simplification

- Groups of nodes can be merged
  [Rountev,Chandra 00]
  - strongly-connected components
  - single-entry subgraphs



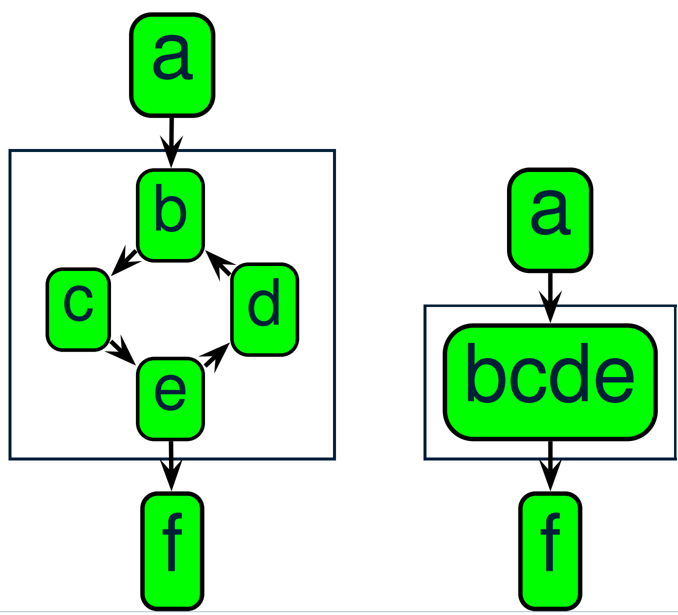
Pointer assignment graph simplification

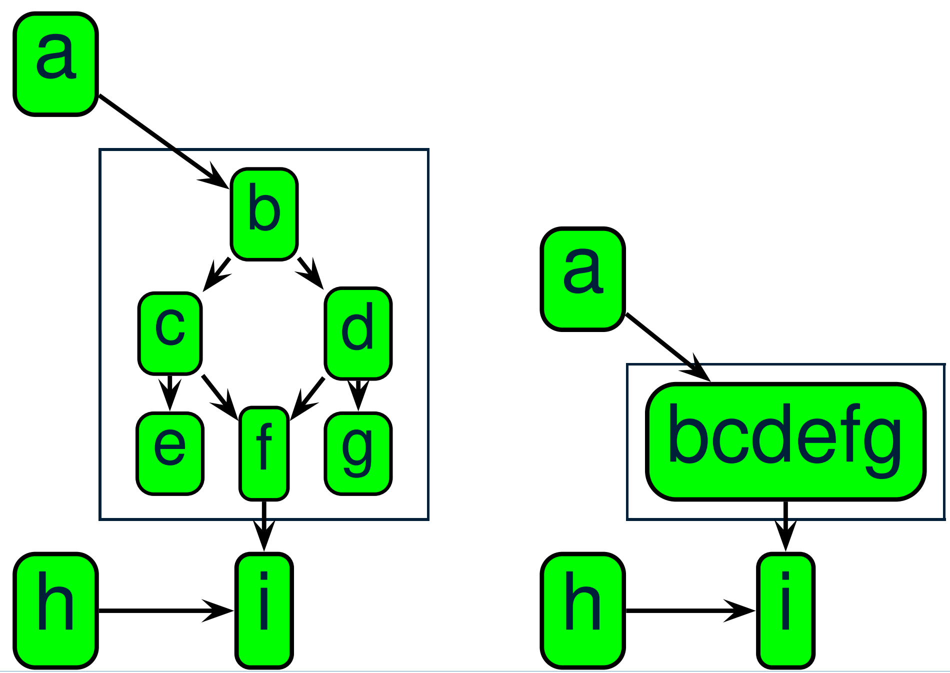
- Groups of nodes can be merged
  [Rountev,Chandra 00]
  - strongly-connected components
  - single-entry subgraphs



Pointer assignment graph simplification

- Groups of nodes can be merged [Rountev,Chandra 00]
  - strongly-connected components
  - single-entry subgraphs

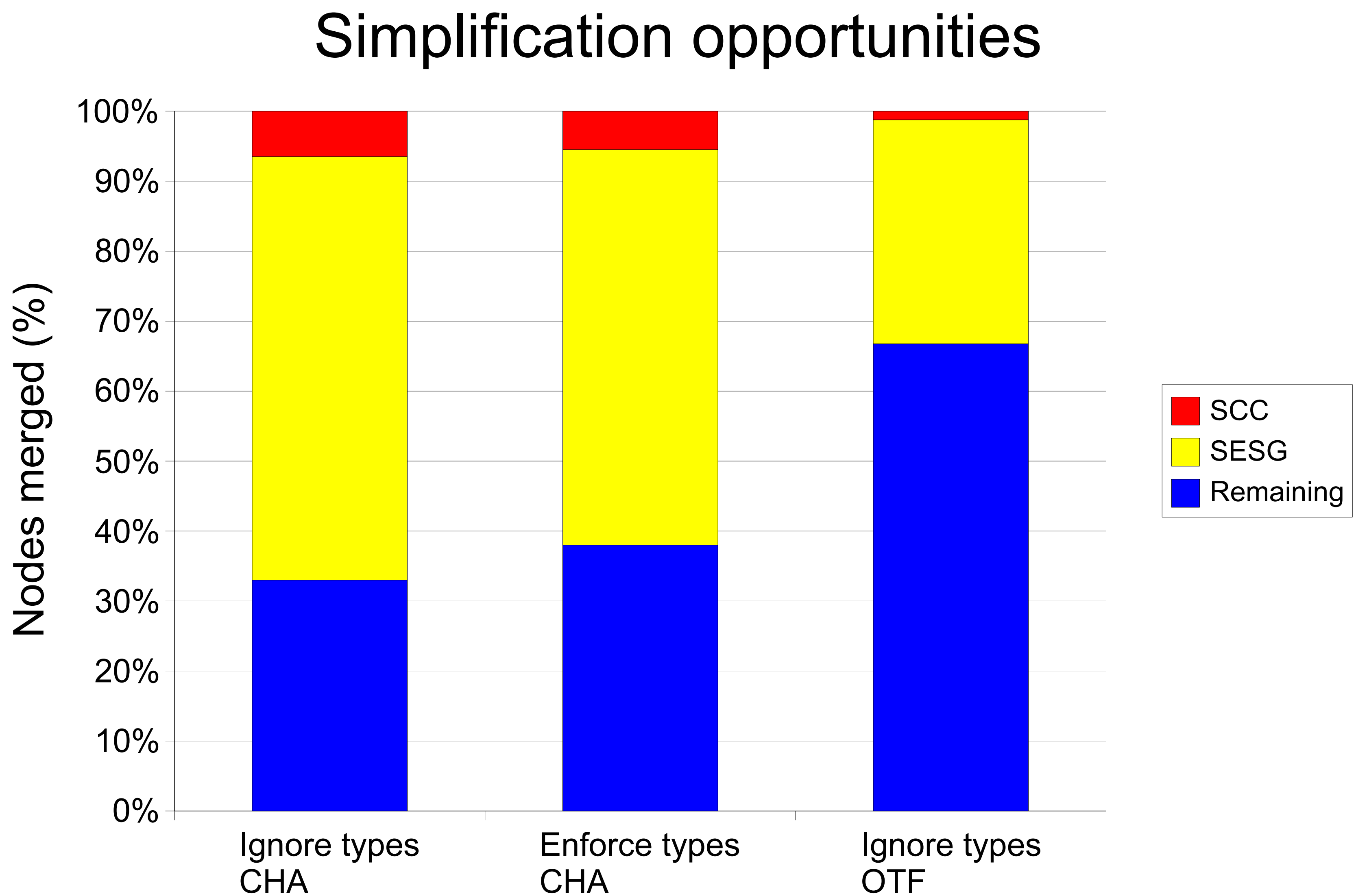




Pointer assignment graph simplification

- Factors limiting simplification opportunities
  - Enforcing declared types changes points-to sets
  - On-the-fly call graph eliminates edges from initial pointer assignment graph

Pointer assignment graph simplification



Empirical study

- Factors affecting precision
  - Enforcing declared types
  - Field reference representation
  - Call graph construction
- **Factors affecting only efficiency**
  - Pointer assignment graph simplification
  - **Set implementation**
  - Propagation algorithms

Set implementation

- **hash** Using java.util.HashSet
- **array** Sorted array, binary search

a	b	d

- **bit** Bit vector

a	b	c	d	e	f	g	h	i	j	. . .	x	y	z
1	1	0	1	0	0	0	0	0	0	. . .	0	0	0

- **hybrid**
  - Array for small sets
  - Bit vector for large sets

Set implementation

hash	slow	large
array	slow	**small**
bit	**fast**	large
hybrid	**fast**	**small**

- In the above table,
  - slow is up to 100 times slower than **fast**
  - large is up to 3 times larger than **small**
- Set implementation is **very** important

Empirical study

- Factors affecting precision
  - Enforcing declared types
  - Field reference representation
  - Call graph construction
- **Factors affecting only efficiency**
  - Pointer assignment graph simplification
  - Set implementation
  - **Propagation algorithms**

Propagation algorithms: iterative

```
repeat
    for each edge e
        propagate along e;
    end for
until no change
```

- Slightly more complicated to handle
  - field references
  - on-the-fly call graph

Propagation algorithms: worklist

while worklist not empty **do**
  remove node *n* from worklist;
  for each edge *e* starting at *n*
    propagate along *e*;
    add all affected nodes to worklist;
  end for
end while

Propagation algorithms: worklist

while worklist not empty **do**
  remove node *n* from worklist;
  for each edge *e* starting at *n*
    propagate along *e*;
    add all affected nodes to worklist;
  end for
end while

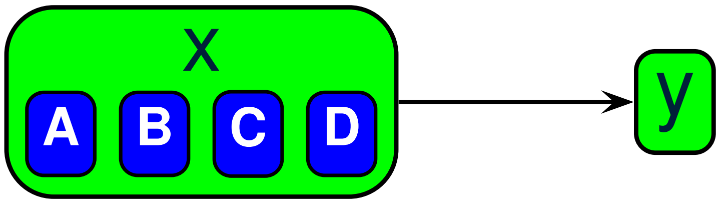
- With field references, difficult to determine affected nodes
- Very costly to determine **all** affected nodes due to of aliasing

Propagation algorithms: **worklist**

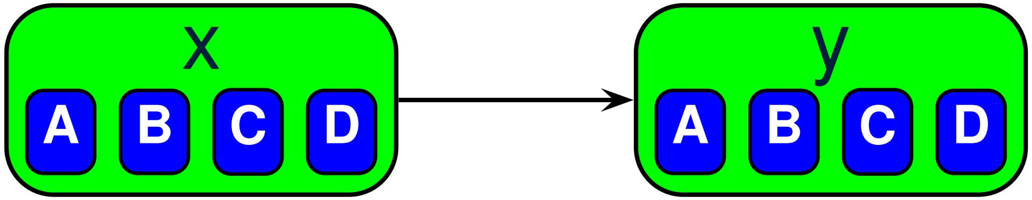
```
repeat
  while worklist not empty do
    remove node n from worklist;
    for each edge e starting at n
      propagate along e;
      add most affected nodes to worklist;
    end for
  end while
  propagate along all field reference edges;
until no change
```

- Solution: find **most** affected nodes, and add outer loop to handle missed nodes

Propagation algorithms: incremental worklist

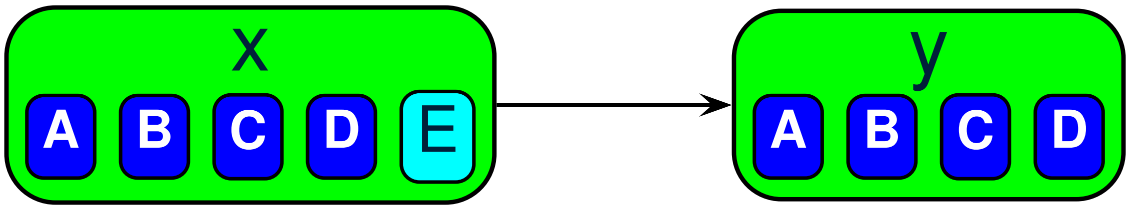


Propagation algorithms: incremental worklist



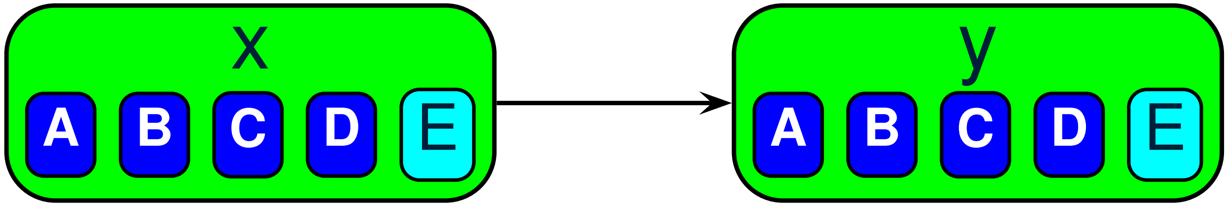
- 1st iteration: propagate {A, B, C, D}

Propagation algorithms: incremental worklist



- 1st iteration: propagate {A,B,C,D}
- add E to x

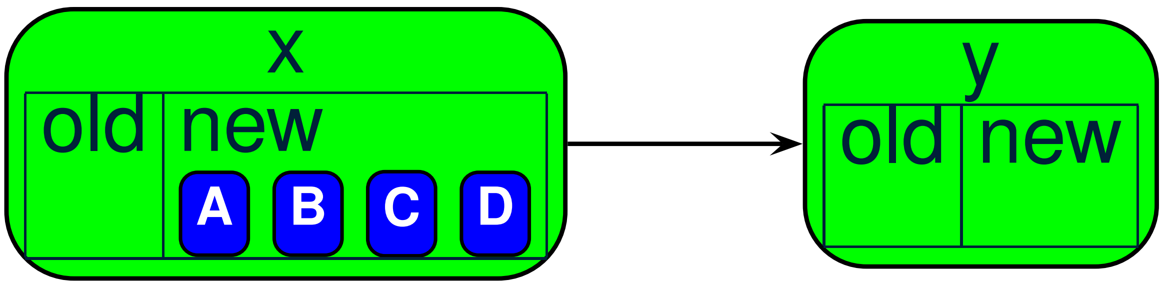
Propagation algorithms: incremental worklist



- 1st iteration: propagate {A, B, C, D}
- add E to x
- 2nd iteration: propagate {A, B, C, D, E}

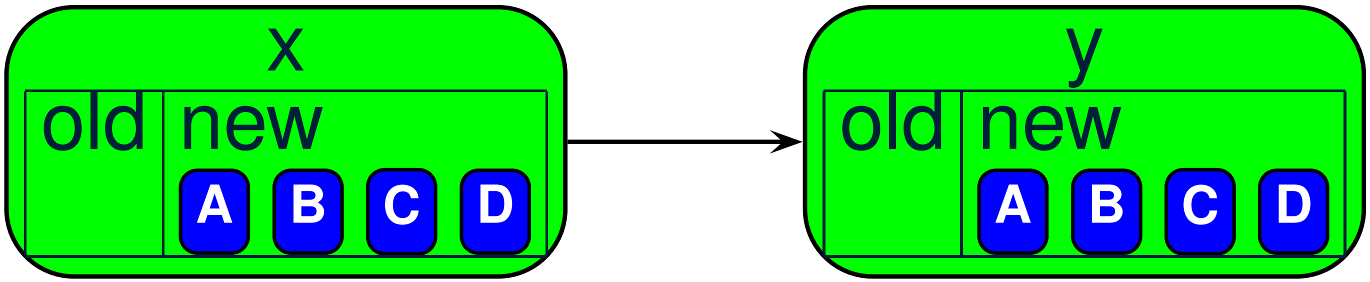
Propagation algorithms: incremental worklist

- Idea: split sets into new and old part



Propagation algorithms: incremental worklist

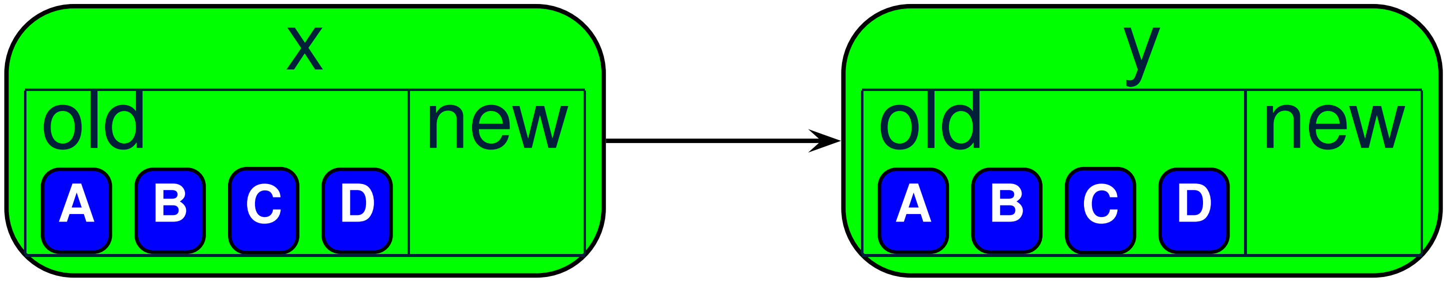
- Idea: split sets into new and old part



- 1st iteration: propagate {A,B,C,D}

Propagation algorithms: incremental worklist

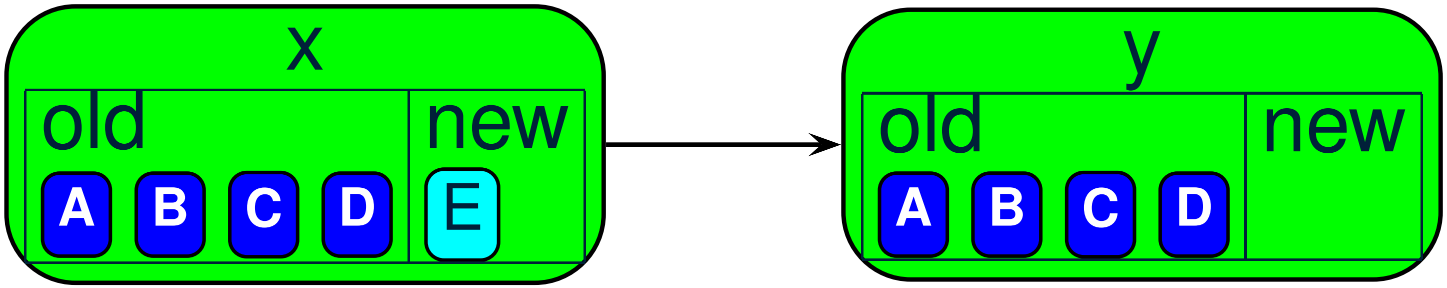
- Idea: split sets into new and old part



- 1st iteration: propagate {A, B, C, D}
- flush new to old

Propagation algorithms: incremental worklist

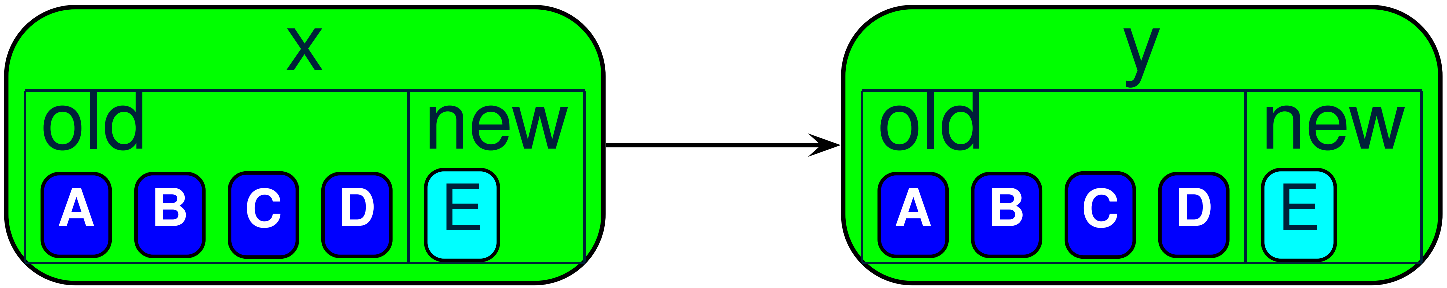
- Idea: split sets into new and old part



- 1st iteration: propagate {A,B,C,D}
- flush new to old
- add E to x

Propagation algorithms: incremental worklist

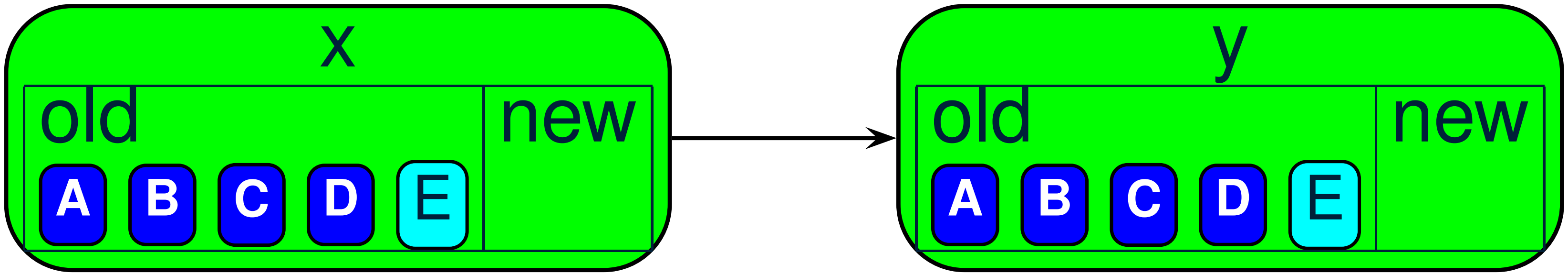
- Idea: split sets into new and old part



- 1st iteration: propagate {A,B,C,D}
- flush new to old
- add E to x
- 2nd iteration: propagate {E}

Propagation algorithms: incremental worklist

- Idea: split sets into new and old part



- 1st iteration: propagate {A,B,C,D}
- flush new to old
- add E to x
- 2nd iteration: propagate {E}
- flush new to old

Propagation algorithms

- When to use worklist?
  - Always, about twice as fast as iterative
- When to use incremental worklist?
  - Always, except with CHA call graph field-based analysis, in which there is not enough iteration

Summary of findings

- Declared types should be enforced during propagation for a scalable analysis
- Hybrid set implementation much faster than others, up to 2 orders of magnitude, with reasonable memory consumption
- Field-based can be done in one iteration, but field-sensitive with worklist algorithm is almost as fast and slightly more precise
- Tradeoff: On-the-fly call graph slower but more precise than ahead-of-time CHA call graph

Outline

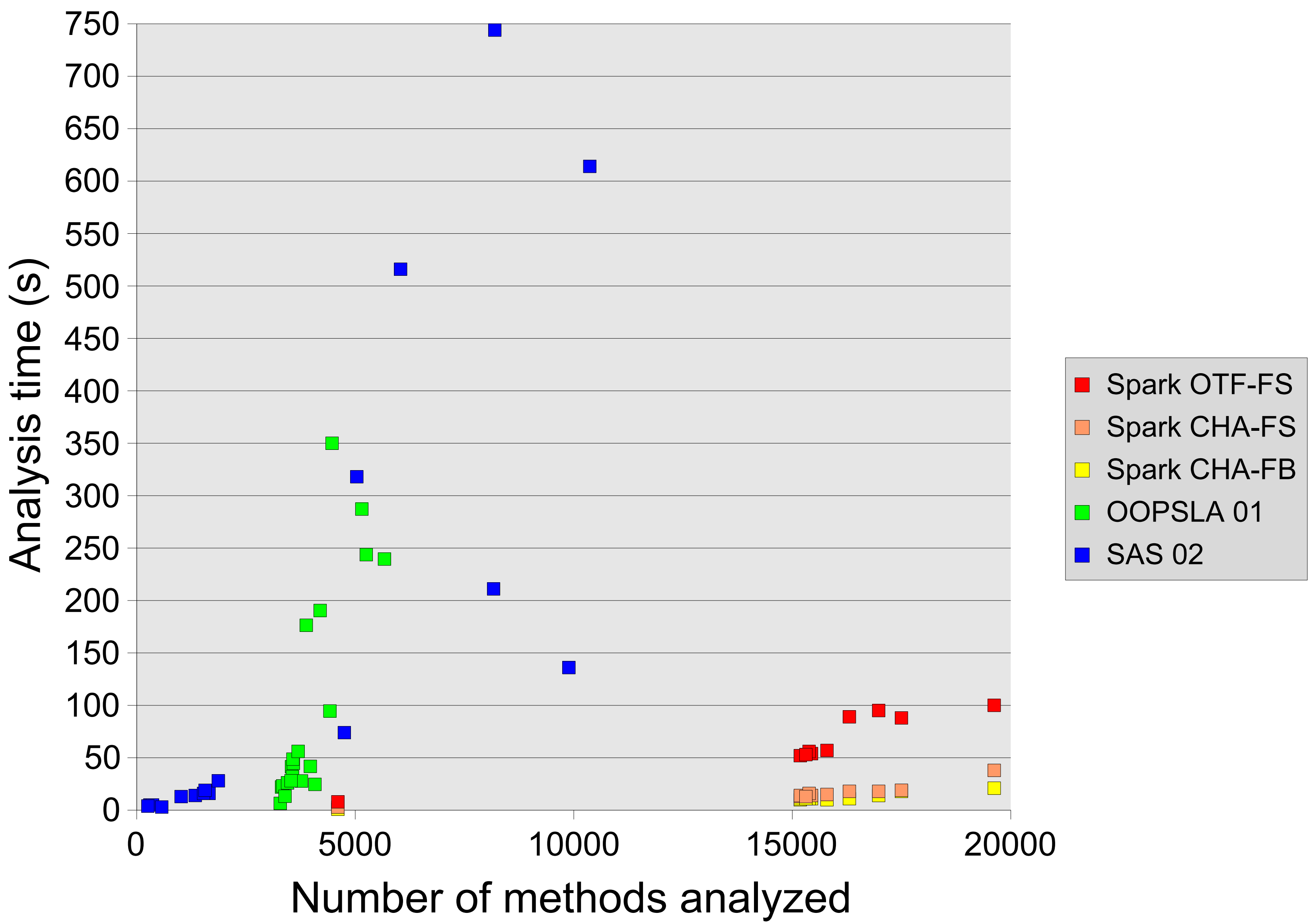
- Spark overview
- Empirical study
- **Overall performance**
- Uses of Spark
- Conclusion

Related studies

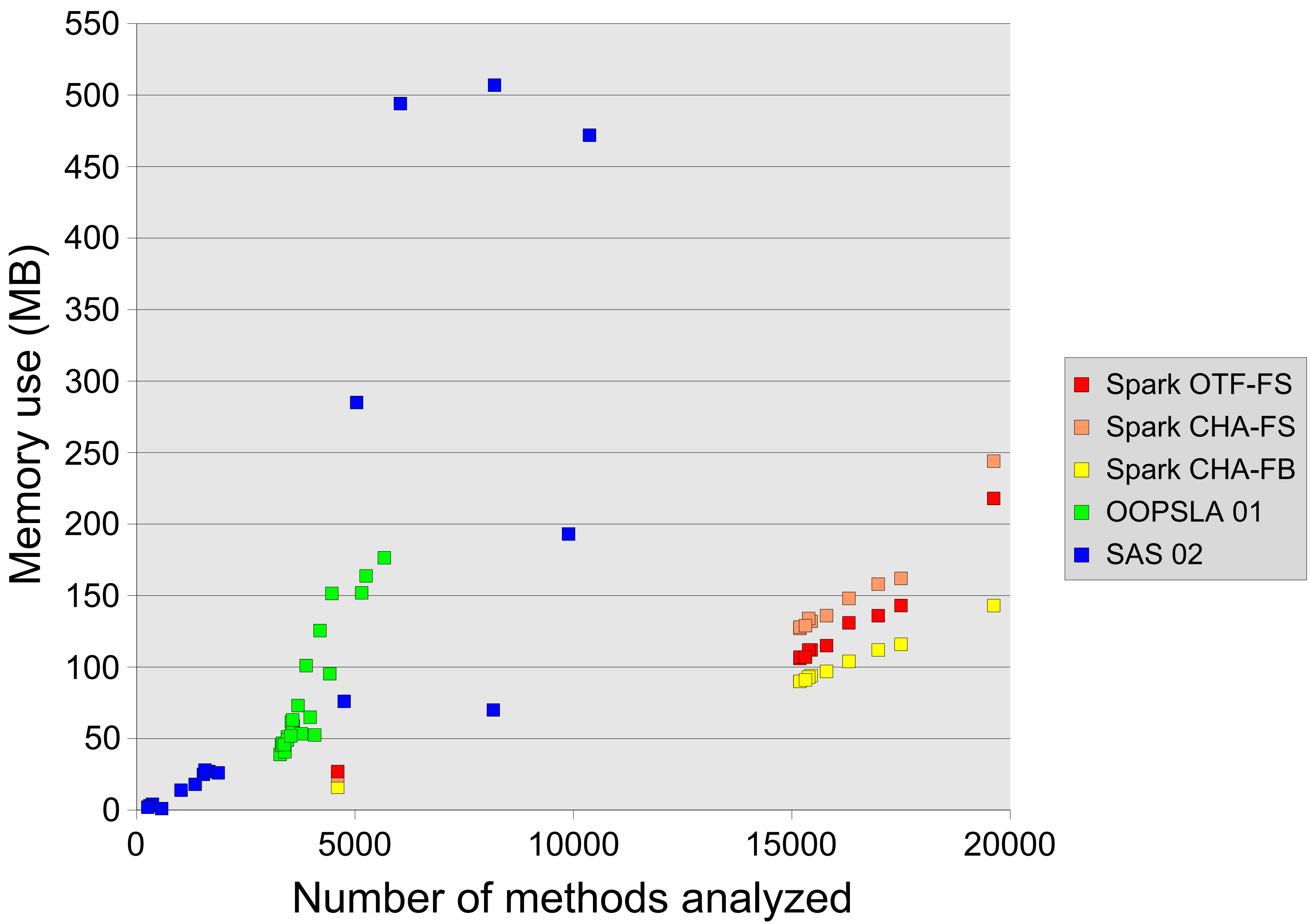
- Rountev, Milanova, Ryder [OOPSLA 01]
  - 360 MHz SPARC, solver written in ML
  - version 1.1.8 library (150 KLOC)
- Whaley, Lam [SAS 02]
  - 2 GHz Pentium, solver written in Java
  - version 1.3.1 library (500 KLOC)
  - optimistic call graph (potentially unsafe)
- **(Spark)** Lhoták, Hendren [CC 03]
  - 1.67 GHz Athlon, solver written in Java
  - version 1.3.1 library (500 KLOC)

Common metric: number of methods analyzed

Overall performance: time



Overall performance: space



Outline

- Spark overview
- Empirical study
- Overall performance
- **Uses of Spark**
- Conclusion

Uses of Spark

- Use points-to and side-effect information in Soot analyses
- Encode in attributes
  - for use in JITs
  - for use in program understanding
- Experiment with points-to algorithms
  - using Spark command-line switches
  - by implementing new algorithms within Spark

Conclusions

- Spark is a flexible and efficient framework for experimenting with variations in Java points-to analyses
- We have demonstrated its usefulness in an empirical study of some of these variations

Ongoing work

- BDD-based solvers [PLDI 03]
- Object-sensitivity [Milanova,Rountev,Ryder 02]
- On-the-fly cycle detection [Heintze,Tardieu 01]
- Shared bit-vector [Heintze,Tardieu 01]

Obtaining Spark

- Spark is part of Soot since version 1.2.4
- Soot is available under the LGPL
  - http://www.sable.mcgill.ca/soot
- Future plans for Soot
  - Major update (version 2.0) in June 2003
  - Tutorial at PLDI