Flow-Insensitive Pointer Analysis

Last time

- Interprocedural analysis
- Dimensions of precision (flow- and context-sensitivity)
- Flow-Sensitive Pointer Analysis

Today

- Flow-Insensitive Pointer Analysis

Flow-Insensitive Pointer Analysis

The defining characteristics

- Ignore the control-flow graph, and assume that statements can execute in any order
- Rather than producing a solution for each program point, produce a single solution that is valid for the whole program

Flow-insensitive pointer analyses

- **Andersen-style analysis:** the slowest and most precise
- **Steensgaard analysis:** the fastest and least precise
- All other flow-insensitive pointer analyses are hybrids of these two

Andersen-Style Pointer Analysis [1994]

Basic idea

- View pointer assignments as constraints
- Use these constraints to propagate points-to information

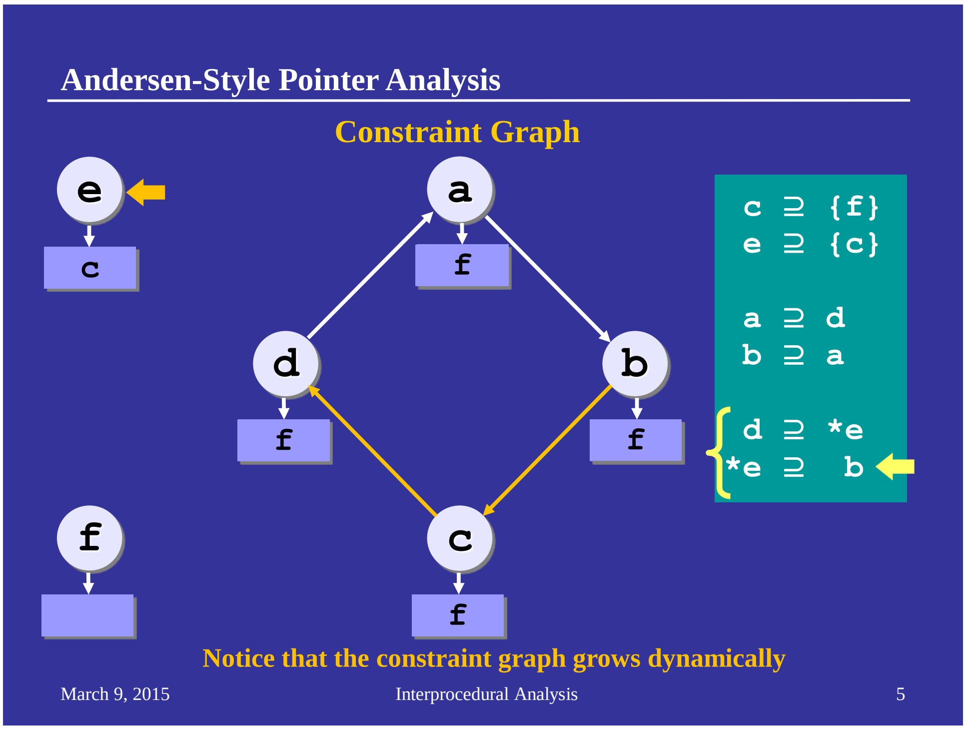
Andersen-Style Pointer Analysis [1994]

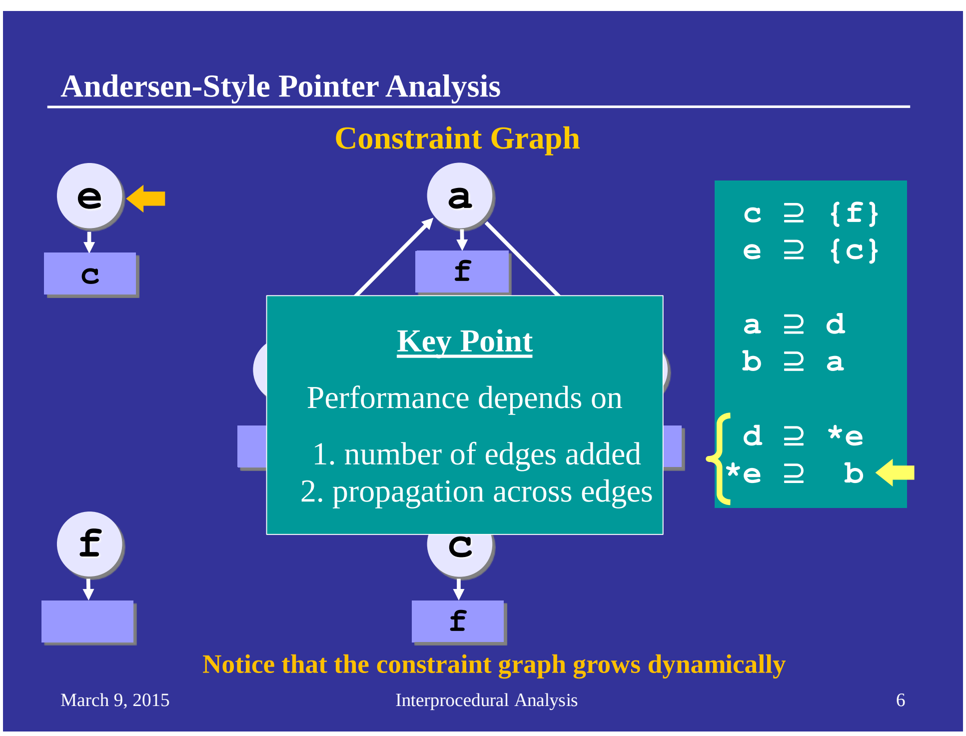
```
void foo()
{
  c = &f;
  e = &c
  b = a;
  if (C) { *e = b; d = *e; a = d; }
}
```

Derive set of constraints on program variables

$$\begin{aligned} c &\supseteq \{f\} \\ e &\supseteq \{c\} \\ a &\supseteq d \\ b &\supseteq a \\ d &\supseteq {*e} \\ {*e} &\supseteq b \end{aligned}$$

Goal: compute the smallest points-to sets that satisfy these constraints


Andersen-Style Pointer Analysis
Constraint Graph
e
c
a
f
d
f
b
f
c
f
f
c ⊇ {f}
e ⊇ {c}
a ⊇ d
b ⊇ a
d ⊇ *e
*e ⊇ b
Notice that the constraint graph grows dynamically
March 9, 2015
Interprocedural Analysis
5


Andersen-Style Pointer Analysis
Constraint Graph
e
c
a
f
Key Point
Performance depends on
1. number of edges added
2. propagation across edges
c
f
f
c ⊇ {f}
e ⊇ {c}
a ⊇ d
b ⊇ a
d ⊇ *e
*e ⊇ b
Notice that the constraint graph grows dynamically
March 9, 2015
Interprocedural Analysis
6

Inclusion-based Pointer Analysis

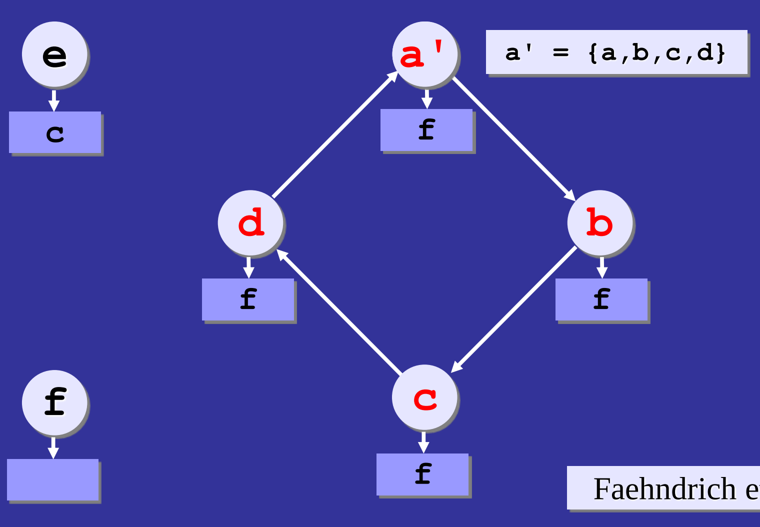
Essentially
- Computes the transitive closure of a dynamic graph

Naïve algorithm doesn't scale— O(n^3)
- Too many edges added
- Quickly runs out of memory

Optimizations
- Cycle detection — PLDI'07
- Location equivalence — SAS'07

March 9, 2015 Interprocedural Analysis 7

Cycle Detection



Faehndrich et al 1998

March 9, 2015 Interprocedural Analysis 8

Cycle Detection

How do we detect cycles?
- They appear dynamically during the analysis
- Check for cycles too often → Too much overhead
- Check for cycles too infrequently → Lost opportunities
- Need to find a sweet spot

Two solutions **[Hardekopf & Lin ‘07]**
- Lazy Cycle Detection
- Hybrid Cycle Detection

March 9, 2015 Interprocedural Analysis 9

Lazy Cycle Detection

Fact
- Cycles cause identical points-to sets

Heuristic
- Identical points-to sets indicate possible cycles
- Don’t look for a cycle unless we have evidence that one might exist
- Perform cycle detection when two nodes have identical points-to sets

Result
- Faster than all previous cycle detection schemes
- See paper for details

March 9, 2015 Interprocedural Analysis 10

Hybrid Cycle Detection

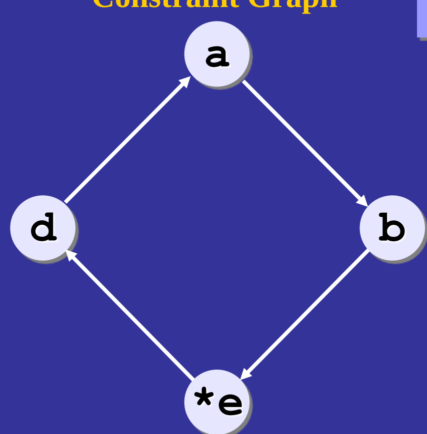
	few cycles	many cycles
cheap	Before the Analysis	Hybrid Cycle Detection
expensive		During the Analysis

Idea: Pre-process the constraint graph with an offline component to make the online component more efficient

Two components
- Offline component (before the analysis)
- Online component (during the analysis)

Hybrid Cycle Detection– Offline Component

Constraint Graph



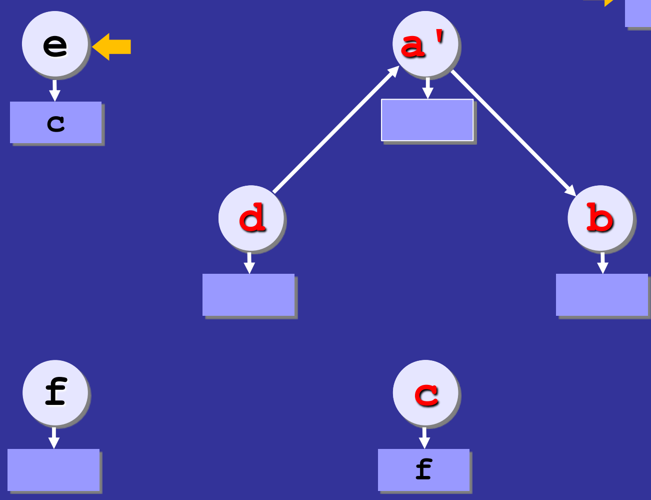
`*e → {a,b,d}`

```
c  ⊇ {f}
e  ⊇ {c}

a  ⊇ d
b  ⊇ a

d  ⊇ *e
*e ⊇  b
```

Hybrid Cycle Detection– Online Component



$*e \rightarrow \{a,b,d\}$

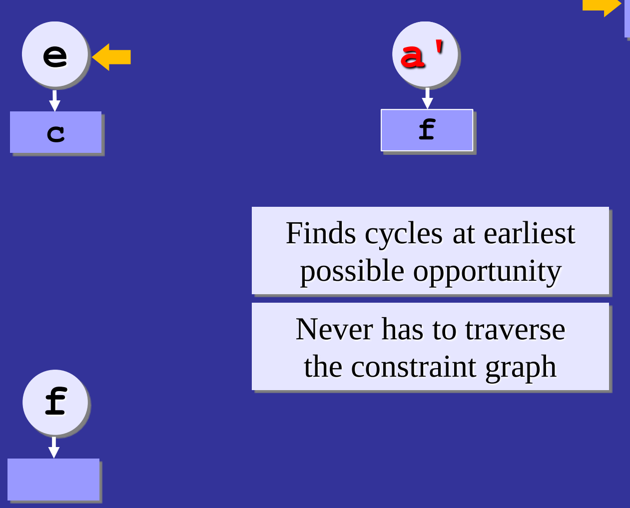
$c \supseteq \{f\}$
$e \supseteq \{c\}$

$a \supseteq d$
$b \supseteq a$

$d \supseteq *e$
$*e \supseteq b$

March 9, 2015 Interprocedural Analysis 13

Hybrid Cycle Detection– Online Component



$*e \rightarrow \{a,b,d\}$

$c \supseteq \{f\}$
$e \supseteq \{c\}$

$a \supseteq d$
$b \supseteq a$

$d \supseteq *e$
$*e \supseteq b$

March 9, 2015 Interprocedural Analysis 14

Evaluation

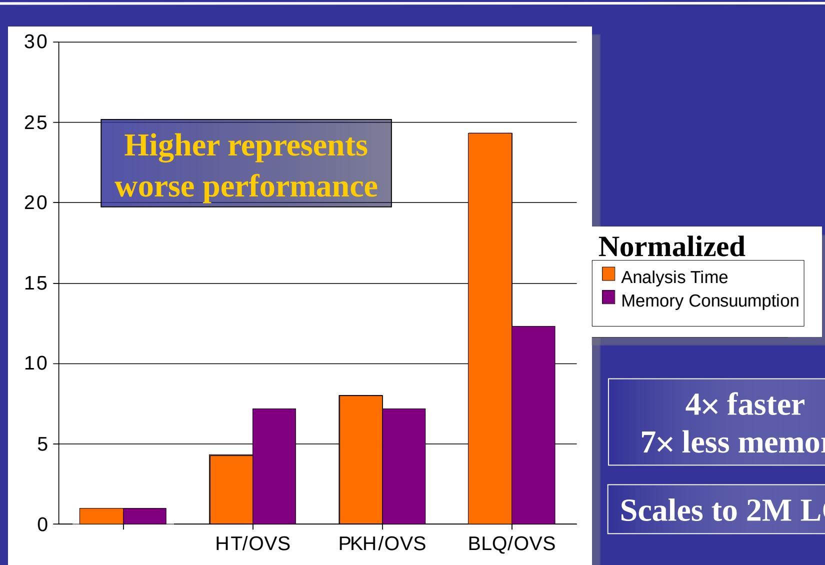
Compare our work with previous state of the art

- First need to identify the state of the art

Rountev et al 2000 (OVS)
Heintze et al 2001 (HT)
Berndl et al 2003 (BLQ)
Pearce et al 2004 (PKH)

- For a fair comparison, we implement algorithms from scratch using the same infrastructure
- Compare analysis time and memory consumption on 10 C benchmarks with 100K – 2M LOC

Evaluation



4× faster
7× less memory

Scales to 2M LOC

Impact

Academic

- PLDI 2007 Best Paper Award
- Raised the bar for empirical evaluation

Industrial

- Implemented in gcc and LLVM compilers
- Implemented by Semantic Designs, Inc
  - Some of their software engineering tools can now scale to over 12M lines of C (previously stuck at 1M)

Andersen-style Pointer Analysis – Procedure Calls

Program

```
foo(int* x){
   . . .
   return x;
}

a := foo(&b)
```

Constraints

$x \supseteq b$

$a \supseteq x$

How do we handle procedure calls?

- Insert constraints for copying actual parameters to formal parameters
- Insert constraints for copying return values

Steensgaard Pointer Analysis [1996]

Basic idea

- Further reduce precision by using equality constraints
- That is, information flows both ways, rather than from the right-hand side to the left-hand side of the constraint

Tradeoffs

- Extremely imprecise
- A system of equality constraints can be solved in near-linear time
- Running time is $O(n \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann's function.
- $\alpha(2^{132}) < 4$

Key idea

- The key to this algorithm is the Union-Find data structure.

March 9, 2015 Interprocedural Analysis 19

Steensgaard Pointer Analysis – Union-Find

The Union-Find data structure

- Maintains a set of disjoint sets and supports two operations:
- Find(x) : return the set containing x.
- Union(x,y) : union the two sets containing x and y.

Set Representation

- Sets are represented by a distinguished element called the set representative
- Each set is an inverted tree, with nodes pointing to their parents and the set representative as the root

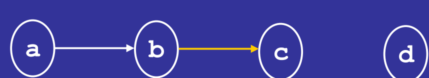
March 9, 2015 Interprocedural Analysis 20

Steensgaard Pointer Analysis – Union-Find

```
Union(a, b)
  – Find(a)
  – Find(b)
```

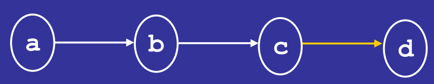
Steensgaard Pointer Analysis – Union-Find

```
Union(a, c)
  – Find(a)
  – Find(c)
```



Steensgaard Pointer Analysis – Union-Find

`Union(a, d)`
- `Find(a)`
- `Find(d)`



Union-Find Optimizations

Two key optimizations
- Path compression
- Union-by-rank
- Together these optimizations yield near-linear time operations

Path compression
- Avoid redundant searches for the set representative

Union-by-rank
- When performing the Union operation, choose the set representative based on the sizes of the two sets

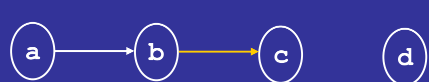
Steensgaard Pointer Analysis – Path Compression

```
Union(a, b)
  – Find(a)
  – Find(b)
```

March 9, 2015 Interprocedural Analysis 25

Steensgaard Pointer Analysis – Path Compression

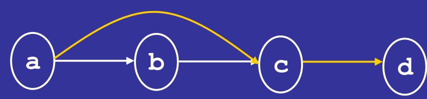
```
Union(a, c)
  – Find(a)
  – Find(c)
```



March 9, 2015 Interprocedural Analysis 26

Steensgaard Pointer Analysis – Path Compression

Union(a, d)
- Find(a)
- Find(d)



March 9, 2015 Interprocedural Analysis 27

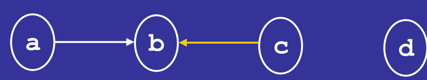
Steensgaard Pointer Analysis – Union-by-Rank

Union(a, b)
- Find(a)
- Find(b)

March 9, 2015 Interprocedural Analysis 28

Steensgaard Pointer Analysis – Union-by-Rank

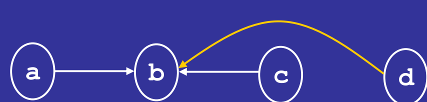
`Union(a, c)`
- `Find(a)`
- `Find(c)`



March 9, 2015 Interprocedural Analysis 29

Steensgaard Pointer Analysis – Union-by-Rank

`Union(a, d)`
- `Find(a)`
- `Find(d)`



What is the benefit of union-by-rank?
- It ensures that we update as few parent pointers as possible
- Consider the cost of selecting **`d`** as the new set representative in this last union operation

March 9, 2015 Interprocedural Analysis 30

Steensgaard Pointer Analysis – Example 1

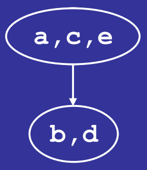
Program

```
a := &b
c := a
a := &d
e := a
```

Constraints

```
a = { b, d }
c = a
e = a
```

Points-to Relations



Steensgaard Pointer Analysis – the Algorithm

```
merge(x, y)
{
    x = Find(x); y = Find(y);
    if (x == y) then return;
    Union(x,y);
    merge(points-to(x),points-to(y));
}

for each constraint LHS = RHS
    merge(LHS,RHS)
```

Steensgaard Pointer Analysis – Example 2

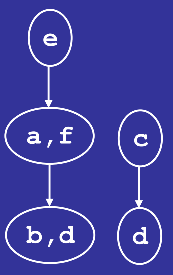
Program

```
a  := &b
c  := &d
e  := &a
f  := a
*e  := c
```

Constraints

```
a = { b }
c = { d }
e = { a }
f = a
*e = c
```

Points-to Relations



Steensgaard Pointer Analysis – Example 2

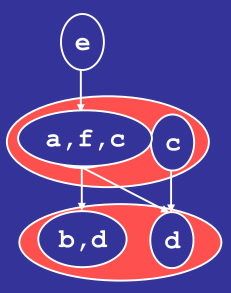
Program

```
a  := &b
c  := &d
e  := &a
f  := a
*e  := c
```

Constraints

```
a = { b }
c = { d }
e = { a }
f = a
*e = c
```

Points-to Relations



Andersen vs. Steensgaard

```
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```

Andersen-style analysis

Steensgaard analysis

The Big Picture

Precision vs. Performance

- Steensgaard's analysis and Andersen's analysis operate on abstractions of the program text
- Instead of the CFG, they operate on sets
- These abstractions trade off precision for performance

Concepts

Flow-insensitive pointer analysis

Andersen-style analysis
- Inclusion-based, subset-based
- Compute transitive closure of a dynamic graph
- Constraint graph
- Cycle elimination optimization

Steensgaard-style analysis
- Unification-based, equality-based
- Union-find data structure

March 9, 2015 Interprocedural Analysis 37

Next Time

Lecture
- Context-Sensitive Pointer Analysis

March 9, 2015 Interprocedural Analysis 38