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 – Example 1

Program	**Flow-Sensitive Solution**
`a := &b`	`a → { b }`
`c := a`	`c → { b }`
`a := &d`	`a → { d }`
`e := a`	`e → { d }`

Andersen-style Pointer Analysis – Example 1

Program	Constraints	Points-to Relations
`a := &b`	$a \supseteq \{ b, d \}$	a → { b, d }
`c := a`	$c \supseteq a$	c → { b, d }
`a := &d`	$e \supseteq a$	e → { b, d }
`e := a`		

We've reached a fixed point

Terminology

- **Base constraints**: Used to initialize the points-to sets
  Ex: `a := &b`
  Not needed after initialization
- **Simple constraints:** Involve variable names only
  Ex: `c := a`
- **Complex constraints:** Involve pointer dereferences
  Ex: `*a := c`

Andersen-style Pointer Analysis – Example 2

Program	Constraints	Points-to Relations
`a := &b`	$a \supseteq \{ b \}$	a → { b, d }
`c := &d`	$c \supseteq \{ d \}$	c → { d }
`e := &a`	$e \supseteq \{ a \}$	e → { a }
`f := a`	$f \supseteq a$	f → { b, d }
`*e := c`	$*e \supseteq c$	
	$a \supseteq c$	

Notice that we create the constraint graph dynamically

Andersen-Style Pointer Analysis

Basic idea

– View pointer assignments using a constraint graph

– Propagate points-to relations along the edges of the constraint graph, adding new edges as indirect constraints are resolved

Constraint graph

– One node for each variable

– One directed edge for each constraint

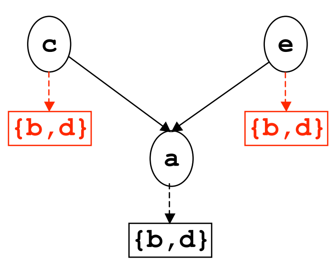
Andersen-style analysis

– Can be reduced to computing the transitive closure of a dynamic graph

– A well-studied problem for which the best known complexity is $O(n^3)$

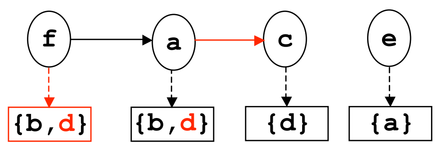
Andersen-style Pointer Analysis – The Constraint Graph

Example 1



Andersen-style Pointer Analysis – The Constraint Graph

Example 2



`[*e ⊇ c]`

Andersen-style Pointer Analysis – Cycle Elimination

Cycle Elimination

- The most important optimization for Andersen-style analysis
- Detect strongly-connected components in the constraint graph
- Collapse them into a single node

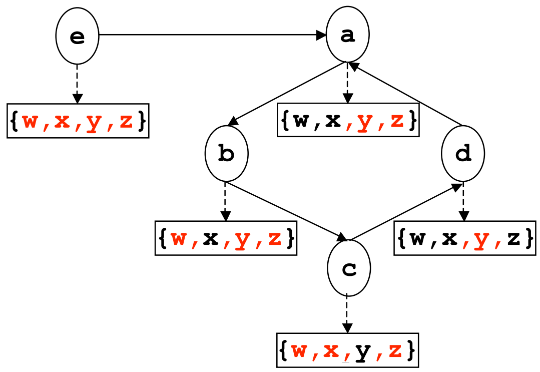
The rationale

- All nodes in the same SCC are guaranteed to have the same points-to relations at the end of the analysis

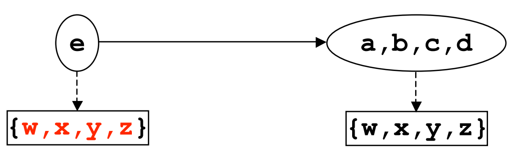
Complication

- Most SCCs are created dynamically during the analysis
- Cycle elimination must be performed dynamically for greatest effect

Andersen-style Pointer Analysis – Cycle Elimination



Andersen-style Pointer Analysis – Cycle Elimination



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

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

- 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.

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.

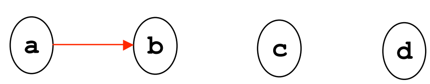
Set Representation

- 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

Steensgaard Pointer Analysis – UNION-FIND

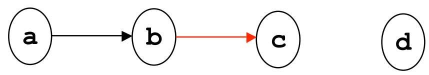
`UNION (a, b)`

- `FIND(a)`
- `FIND(b)`



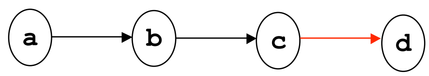
Steensgaard Pointer Analysis – UNION-FIND

`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

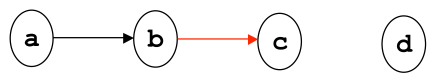
Union-by-rank
- When performing the UNION operation, choose the set representative based on the sizes of the two sets

Steensgaard Pointer Analysis – Path Compression

```
UNION (a, b)
  - FIND(a)
  - FIND(b)
```

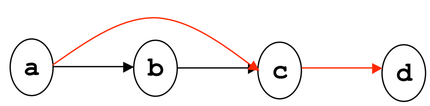
Steensgaard Pointer Analysis – Path Compression

`UNION (a, c)`
- **`FIND(a)`**
- **`FIND(c)`**



Steensgaard Pointer Analysis – Path Compression

`UNION (a, d)`
- **`FIND(a)`**
- **`FIND(d)`**



Steensgaard Pointer Analysis – Union-by-Rank

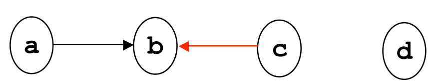
UNION (a, b)

- **FIND(a)**
- **FIND(b)**

Steensgaard Pointer Analysis – Union-by-Rank

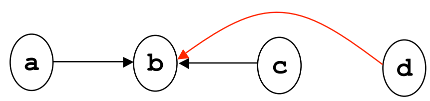
UNION (a, c)

- **FIND(a)**
- **FIND(c)**



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 follow as few parent pointers as possible
- Consider the cost of selecting **d** as the new set representative in this last union operation

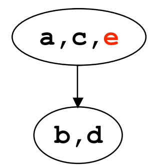
Steensgaard Pointer Analysis – the Algorithm

```
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)
```

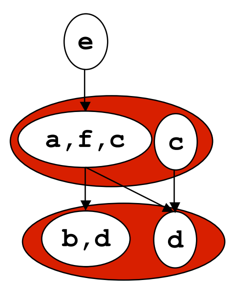
Steensgaard Pointer Analysis – Example 1

Program	Constraints	Points-to Relations
`a := &b`	`a = { b, d }`	
`c := a`	`c = a`	
`a := &d`	`e = a`	
`e := a`		



Steensgaard Pointer Analysis – Example 2

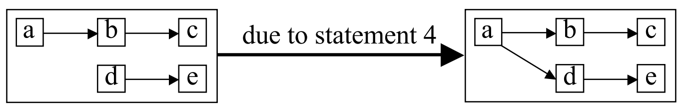
Program	Constraints	Points-to Relations
`a := &b`	`a = { b }`	
`c := &d`	`c = { d }`	
`e := &a`	`e = { a }`	
`f := a`	`f = a`	
`*e := c`	`*e = c`	



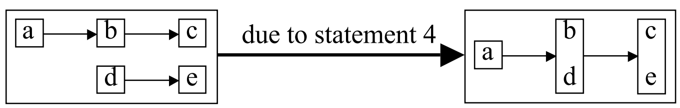
Andersen vs. Steensgaard

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

Andersen-style analysis



Steensgaard analysis



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

Next Time

Lecture

– Context-Sensitive Pointer Analysis