Pointer Analysis

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2015

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

Points-to Analysis as a Graph Problem

Each pointer as a node, directed edge p → q indicates points-to set of q is a subset of that of p.

Input: set C of points-to constraints

Process address-of constraints

Add edges to constraint graph G using copy constraints

repeat

  Propagate points-to information in G

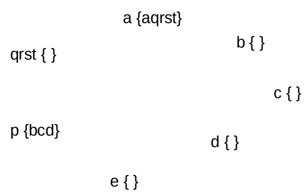
  Add edges to G using load and store constraints

until fixpoint

Points-to Analysis as a Graph Problem

*e = c, c = *a, e = d, b = a, *a = p

Initially, a→{a,q,r,s,t}, p→{b,c,d}

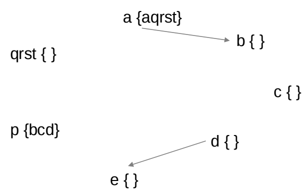


Points-to Analysis as a Graph Problem

*e = c, c = *a, e = d, b = a, *a = p

Initially, a→{a,q,r,s,t}, p→{b,c,d}

Iteration 0



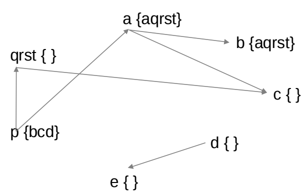
e = d
b = a
----
*e = c
c = *a
*a = p

Points-to Analysis as a Graph Problem

*e = c, c = *a, e = d, b = a, *a = p

Initially, a→{a,q,r,s,t}, p→{b,c,d}

Iteration 1



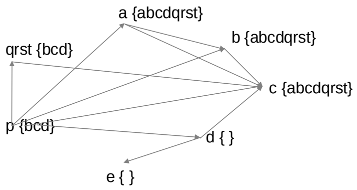
e = d
b = a
----
*e = c
c = *a
*a = p

Points-to Analysis as a Graph Problem

*e = c, c = *a, e = d, b = a, *a = p

Initially, a→{a,q,r,s,t}, p→{b,c,d}

Iteration 2



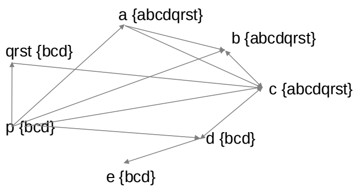
e = d
b = a
----
*e = c
c = *a
*a = p

Points-to Analysis as a Graph Problem

*e = c, c = *a, e = d, b = a, *a = p

Initially, a→{a,q,r,s,t}, p→{b,c,d}

Iteration 3



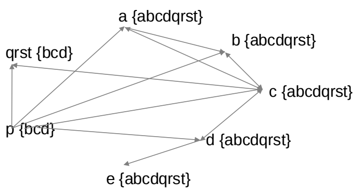
e = d
b = a
----
*e = c
c = *a
*a = p

Points-to Analysis as a Graph Problem

*e = c, c = *a, e = d, b = a, *a = p

Initially, a→{a,q,r,s,t}, p→{b,c,d}

Iteration 4



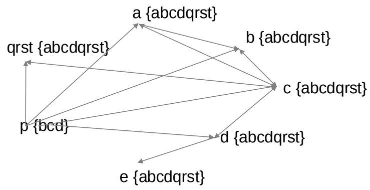
e = d
b = a
----
*e = c
c = *a
*a = p

Points-to Analysis as a Graph Problem

*e = c, c = *a, e = d, b = a, *a = p

Initially, a→{a,q,r,s,t}, p→{b,c,d}

Iteration 5: fixed-point



e = d
b = a
----
*e = c
c = *a
*a = p

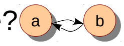
Why a Graph Formulation?

- A naïve formulation offers no benefits over the constraint-based formulation.
- We need to exploit structural properties of the constraint graph for efficient execution.
  - Online cycle detection
  - Online dominator detection
  - Propagation order: Topological sort, Depth first

Pointer Equivalence

- Two pointers are equivalent if they have the same points-to sets. Simple.
- If we identify such pointers *before* computing their points-to information, we can reduce the number of pointers tracked during the analysis.
- Now let's go back to the constraint graph.

Why a Graph Formulation?

- If the program contains statements a = b, b = a, what can you say about the points-to sets of a and b at the fixed-point?
- How does the constraint graph look like? 

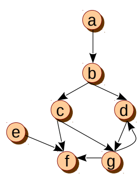
- How about a = b, b = c, c = a?
- How about a = c, b = *p, c = b?

Online Cycle Detection

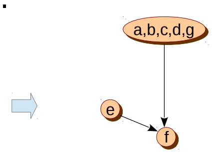
- Edges get added to the graph dynamically.
- So, cycle detection is performed online.
- Cycles are collapsed – usually replaced with a representative.
- Can use union-find.

Online Dominator Detection

- If two nodes in a constraint graph have the same dominator, they are pointer equivalent.
- A dominator and its dominees are pointer equivalent.
- doms is a transitive relation.



b doms g
!(b doms f)
a doms b
By transitivity, a doms g



Offline Variable Substitution

- But some constraints were easy to check for equivalence without running the analysis.
  - a = b, b = a
  - a = *p, *p = a
  - a = b, c = a, c = b and no other incoming edge to c.
- OVS is performed before running pointer analysis.

Propagation Order

- A topological ordering is beneficial for propagating points-to information (wave propagation)
- The information may also be propagated in depth-first manner (deep propagation)
- DP is helpful to reuse the difference in points-to information

How About Constraint Order?

- Given a set of constraints, find an optimal way of evaluating them
- Like most CS problems, this is NP-Complete
- Reducible from Set Cover

Reduction from Set Cover

- Given an instance of Set Cover SC(U, S, K)
  - U: universe of elements
  - S: set of subsets S_i
  - K: some number

  whether there exists a set of K subsets covering U
- Reduce to PTA(C, S, K) where
  - C is a set of copy constraints
  - S is a variable of interest w.r.t. fixed-point
  - K is the number of steps in which the fixed-point is reached

S = {1, 4}, {2, 5}, {2, 4, 5}, {3}
Solution Two: {1, 4}, {2, 4, 5}, {3}
Solution One: {1, 4}, {2, 5}, {3}

SC ⩾ PTA

- SC(U, S, K) ⩾ PTA(C, S, K)
- Linear time reduction
  - for each s ∈ S_i add s to ptsto(S_i)
  - for each set S_i create a copy statement S = S_i
- A solution to PTA ⇒ A solution to SC
- A solution to PTA ⇐ A solution to SC
- Poly-time verification

NP-Hard

NPC

NP

How About Constraint Order?

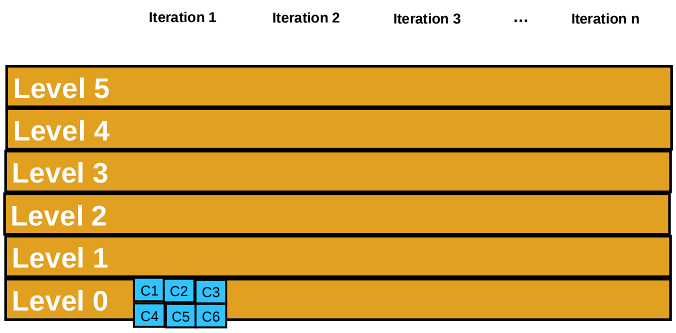
- Given a set of constraints, find an optimal way of evaluating them
- Like most CS problems, this is NP-Complete
- Reducible from Set Cover
- Need to depend upon heuristics

What would be a good heuristic?

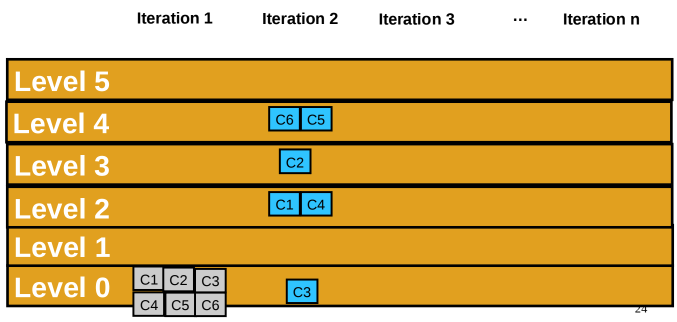
Constraint Priority

- Priority of a constraint in iteration *i* is the amount of new points-to information it adds in iteration *(i – 1)*.
- Constraints are grouped in different priority levels which are ordered based on their priority.
- A constraint may jump across multiple priority levels during the analysis.

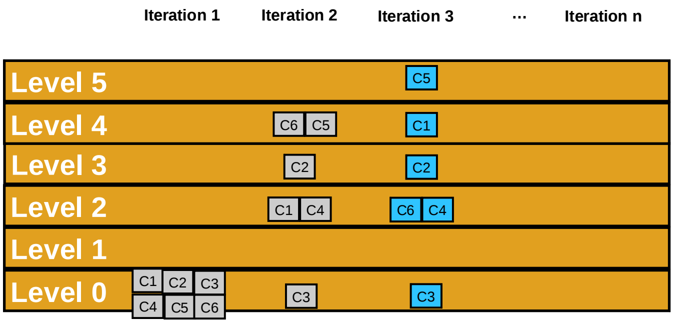
Bucketization



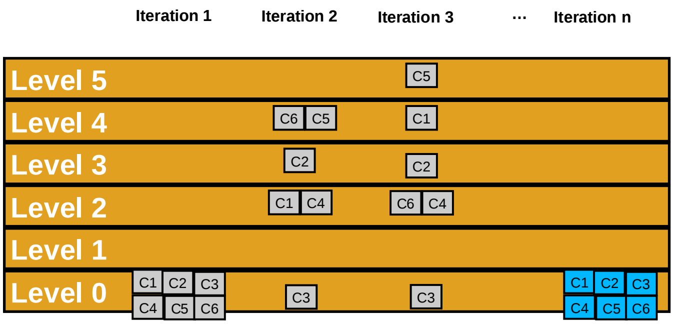
Bucketization



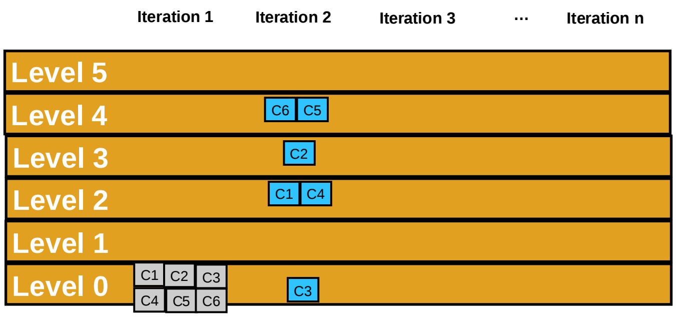
Bucketization



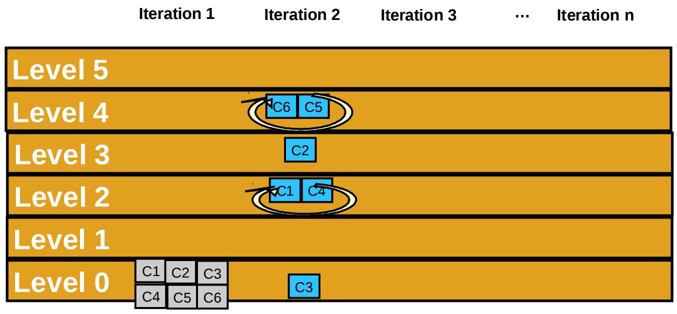
Bucketization



Skewed Evaluation



Skewed Evaluation



Prioritized Points-to Analysis

Processing order

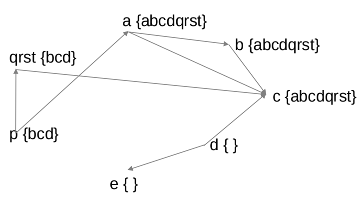
*a = p (18)
c = *a (8)
*e = c (0)

Prioritized Points-to Analysis

Processing order

*a = p (18)
c = *a (8)
*e = c (0)

Priority: Iteration 1

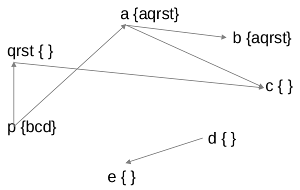


Prioritized Points-to Analysis

Fixed Processing order

*e = c
c = *a
*a = p

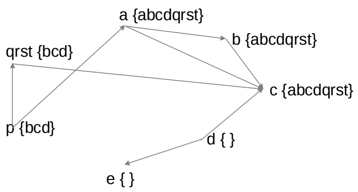
Andersen: Iteration 1



Processing order

*a = p (18)
c = *a (8)
*e = c (0)

Priority: Iteration 1

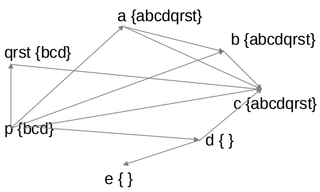


Prioritized Points-to Analysis

Fixed Processing order

*e = c
c = *a
*a = p

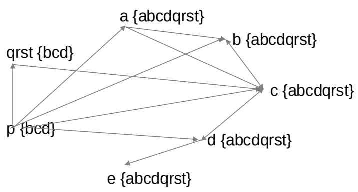
Andersen: Iteration 2



Processing order

*a = p (6)
c = *a (0)
*e = c (10)

Priority: Iteration 2

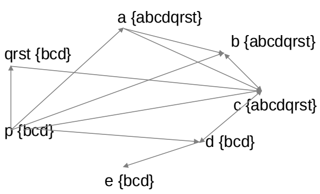


Prioritized Points-to Analysis

Fixed Processing order

*e = c
c = *a
*a = p

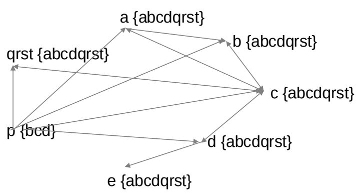
Andersen: Iteration 3



Processing order

*e = c (20)
*a = p (0)
c = *a (0)

Priority: Iteration 3

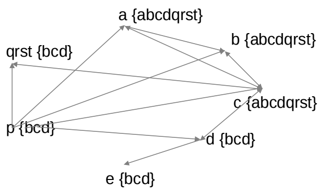


Prioritized Points-to Analysis

Fixed Processing order

*e = c
c = *a
*a = p

Andersen: Iteration 4



Processing order

*e = c (0)
*a = p (0)
c = *a (0)

Priority: fixed-point