Pointer Analysis

Wei Le

February 5, 2020

Why we should care?

```
1. X = 5
2. *p = ...    // p may or may not point to X
3. ... = X
```

Constant propagation: assume p does point to X (i.e., in statement 3, X cannot be replaced by 5).

Dead Code Elimination: assume p does not point to X (i.e., statement 1 cannot be deleted).

What is Pointer Analysis? [Hind2001]

Pointer analysis statically determines:

- the possible runtime values of a pointer: what storage locations a pointer can point to
- there are certain models can represent the storage locations:
  - In Microsoft Phoenix, memory locations are labeled with tags (numbers), each tag represents an object
  - an entire heap is an object

What is Alias Analysis? [Hind2001]

Alias analysis statically determines

- when two pointer variables or memory references refer to the same memory location (global, stack storage, heap)
- produces alias (points-to) relations

Input: int x; p = &x; q = p;
Output:

- alias pairs: *p and *q, *p and x, *q and x [Shapiro&Horwitz97]
- equivalence set: {*p, x, *q}
- points to pairs: $p \rightarrow x$, $q \rightarrow x$ [Emami94]

*Note: pointer analysis, alias analysis, points-to analysis often are used interchangeably

May and Must Aliasing

- *May aliasing*: aliasing that may occur during execution (e.g., if (c) p = &i)
- *Must aliasing*: aliasing that must occur during execution (e.g., p = &i)
- Easiest alias analysis: nothing must alias, everything may alias

How Hard Is This Problem?

- Undecidable [Landi1992] [Ramalingan1994]
- Approximation algorithms, worst-case complexity, range from almost linear to exponential [Hind2001]
- Research started in late 1970s, still not perfectly solved

Pointer analysis: design decisions

flow-insensitive, *context-insensitive* pointer analysis:

- *flow-insensitive analysis*: the order of statements is not considered
- Compute solutions (points-to information) for the entire function rather than at a program point
  - what are the alias pairs for the function?
  - what are the alias pairs at this program point?
- Calling context is not considered – it does not matter where the function is invoked in the program, the alias information computed for the function is the same
- There can be flow-insensitive and context-sensitive analysis, why?

Andersen-Style Pointer Analysis [Andersen1994]

- First for C programs (1994) later for Java
- View pointer assignments as *subset constraints*:
  - base – pointer initialization
  - simple – variable names,
  - complex – pointer dereferences

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
Simple	a = b	a ⊇ b	pts(a) ⊇ pts(b)
Complex	a = *b	a ⊇ *b	∀v∈pts(b). pts(a) ⊇ pts(v)
Complex	*a = b	*a ⊇ b	∀v∈pts(a). pts(v) ⊇ pts(b)

Andersen-Style Pointer Analysis Algorithm

Step 1: map to subset constraints
Step 2: construct the *constraint graphs* – dynamic graph (changes over analysis of the program)

- Node: representing points-to set for each variable, e.g., pts(p), pts(a)
- Edge: directed, represent a subset constraint

Andersen-Style Pointer Analysis Algorithm

Assgmt.	Constraint	Meaning	Edge
a = &b	$a \supseteq \{b\}$	$b \in pts(a)$	no edge
a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$	$b \rightarrow a$
a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$	no edge
*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$	no edge

Andersen-Style Pointer Analysis Algorithm: Example

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

We've reached a fixed point

Andersen-Style Pointer Analysis Algorithm

Step 3. compute a transitive closure to propagate points-to relations along the edges of the constraint graphs ($O(n^3)$ complexity):

- A relation is an ordered pair:

$$R \subseteq X \times Y = \{(x, y) | x \in X, y \in Y\}$$

- A relation is transitive if for all values a, b, c: a R b and b R c implies a R c
- the *transitive closure* of a binary relation R on a set X is the smallest relation on X that contains R and is transitive.
- Example: if X is a set of airports and x R y means "there is a direct flight from airport x to airport y" (for x and y in X), then the transitive closure of R on X is the relation R+ such that x R+ y means "it is possible to fly from x to y in one or more flights". Informally, the transitive closure gives you the set of all places you can get to from any starting place.

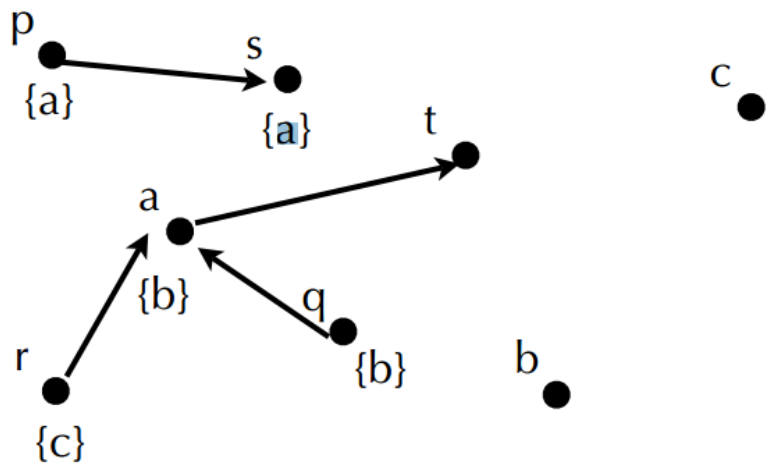
Andersen-Style Pointer Analysis Algorithm

- Initialize graph and points to sets using base and simple constraints
- Let W = { v | pts(v) $\neq \varnothing$ } (all nodes with non-empty points to sets)
- While W not empty
  - v $\leftarrow$ select from W
  - for each a $\in$ pts(v) do
    - for each constraint p $\supseteq$ *v
      - add edge a $\rightarrow$ p, and add a to W if edge is new
    - for each constraint *v $\supseteq$ q
      - add edge q $\rightarrow$ a, and add q to W if edge is new
  - for each edge v $\rightarrow$ q do
    - pts(q) = pts(q) $\cup$ pts(v), and add q to W if pts(q) changed

Andersen-Style Pointer Analysis Algorithm: Example

```
p = &a
q = &b
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```

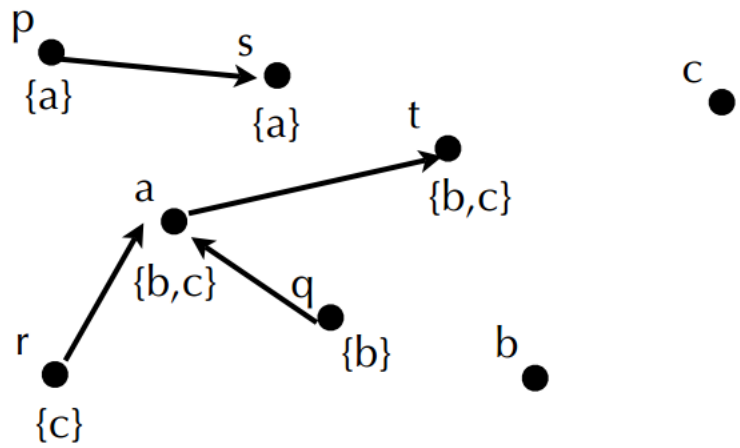
$p \supseteq \{a\}$
$q \supseteq \{b\}$
$*p \supseteq q$
$r \supseteq \{c\}$
$s \supseteq p$
$t \supseteq *p$
$*s \supseteq r$



Andersen-Style Pointer Analysis Algorithm: Example

```
p = &a
q = &b
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```

$p \supseteq \{a\}$
$q \supseteq \{b\}$
$*p \supseteq q$
$r \supseteq \{c\}$
$s \supseteq p$
$t \supseteq *p$
$*s \supseteq r$



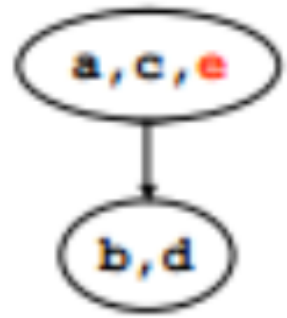
Steensgaard-Style Pointer Analysis [Steensgaard1996POPL] (optional)

- Uses *equality constraints* instead of subset constraints

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	a ⊇ {b}	loc(b) ∈ pts(a)
Simple	a = b	a = b	pts(a) = pts(b)
Complex	a = *b	a = *b	∀v∈pts(b). pts(a) = pts(v)
Complex	*a = b	*a = b	∀v∈pts(a). pts(v) = pts(b)

Steensgaard-Style Pointer Analysis (optional)

- Andersen: many out edges, one variable per node; Steensgaard: one out edge, many variables per node; both are flow-insensitive and context-insensitive; differ in points-to set construction

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

- *Unification based approach*: assignment unifies the graph nodes, e.g., x = y (unified x and y in the same node), also called *union-find algorithm*, nearly linear complexity

Steensgaard-Style Pointer Analysis (optional)

```
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)
```

Andersen vs. Steensgaard Performance

Name	Size (LoC)	Andersen(sec)	Steensgaard(sec)
triangle	1986	2.9	0.8
gzip	4584	1.7	1.1
li	6054	738.5	4.7
bc	6745	5.5	1.6
less	12152	1.9	1.5
make	15564	260.8	6.1
tar	18585	23.2	3.6
espresso	22050	1373.6	10.2
screen	24300	514.5	10.1

75MHz SuperSPARC, 256MB RAM

[Shapiro-Horwitz POPL'97]

Pointer Analysis in Java

- Goal: determine the set of objects pointed to by a reference variable or a reference object field.
- Different languages use pointers differently:
  - Most C programs have many occurrences of the address-of (&) operator in addition to dynamic allocation
  - Java allows no stack-directed pointers, many more dynamic allocation sites than similar-sized C programs
  - Java strongly typed, limits set of objects a pointer can point to (Can improve precision)
  - Larger libraries in Java

Modeling Memory Locations

- For global variables, use a single node
- For local variables, use a single node per context, just one node if context insensitive
- For dynamically allocated memory
  - problem: potentially unbounded locations created at runtime
  - Need to model locations with some finite abstraction

Modeling Dynamic Memory Locations

- For each allocation statement, use one node per context
- One node for entire heap
- One node for each type
- Nodes based on the *shape* of the heap

Recursive Data Structure

- 1-level: combine all elements as one big cell [Emami'93,Wilson et al'95]
- k-limiting: distinguish first k-elements (k is an arbitrary number) [Landi et al'92, Cheng et al'00]
- beyond k-limiting: consider all elements, using some kinds symbolic access paths [Deutch'94]
- *shape analysis*: not only distinguish all elements, but also tell how they are connected (connection analysis) and what their shape is.[Ghiya et al'95]

Shape Analysis

Goal:

- Static code analysis that discovers and verifies properties of linked, dynamically allocated data structures in (usually imperative) computer programs
- Examples: does the data structure have a cycle? two data structures share a piece of memory? does a pointer points to a tree, graph? what are the invariant of the data structure? x and y point to structures that do not share cells?

Applications:

- Verification: detecting memory leaks, proving the absence of mutual exclusive and deadlock
- Optimization: parallel programs

History:

- Originally formulated by Jones and Muchnick, 1981
- Mooly Sagiv, Tom Reps and Reinhard Wihelm, since 1995

References and Further Reading

- Pointer Analysis: Haven't We Solved This Problem Yet?