CS6843: Program Analysis

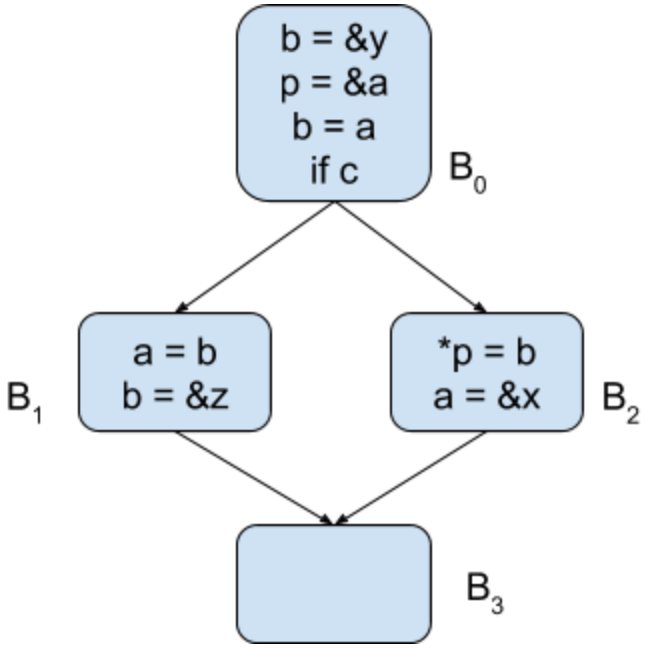
Week 4: 30th Jan - 3rd Feb

INSTRUCTOR: Prof. Rupesh Nasre
Scribes: Kritin Sai (CS14B062) and Sri Harsha (CS14B027)

Modeling Pointer Analysis as a DFA (contd.)

Example explained in class:

Consider a program having a Control flow graph as follows:



To find the points-to sets (which are the data flow facts here) at the IN and OUT points of all the basic blocks (flow sensitively):

- Initially, IN and OUT sets are empty, i.e. the points-to sets of all pointers are empty.
- The GEN and KILL sets are dynamic and could change with iterations depending on the changing IN sets.
    - For a flow sensitive analysis, different copies of points-to sets (IN, OUT sets in particular, as GEN, KILL sets are generated dynamically depending on IN and OUT) need to be maintained for each pointer for each basic block.
    - If a single points-to set is maintained for each pointer variable, common over all basic blocks, then the resulting analysis would be partially flow

insensitive as edges would be added from the processed basic blocks to the ones which need to be processed resulting in imprecision.

    - In partially flow insensitive analysis, flow is maintained inside basic blocks, but not across basic blocks.

- We first consider the GEN sets of each basic block (KILL sets can be ignored as the IN sets are empty initially), which, in the first iteration are as follows:

  $GEN_1(B_0)$ = {~~b → y~~, p → a}; b → y gets killed as b = a and a's points-to set is initially empty.

  $GEN_1(B_1)$ = {b → z};

  $GEN_1(B_2)$ = {a → x};

- If we calculate OUT sets first and then IN sets:

  From **OUT(B) = GEN(B) ∪ (IN(B) - KILL(B)):**

  $OUT_1(B_0) = GEN_1(B_0)$

  $OUT_1(B_1) = GEN_1(B_1)$

  $OUT_1(B_2) = GEN_1(B_2)$

  $OUT_1(B_3) = GEN_1(B_3) = \{\}$

  From **IN(B) =** ∪**OUT(P)**, where P is a predecessor of B:

  $IN_1(B_0) = \{\}$

  $IN_1(B_1) = OUT_1(B_0)$

  $IN_1(B_2) = OUT_1(B_0)$

  $IN_1(B_3) = OUT_1(B_1) \cup OUT_1(B_2)$

- The GEN and KILL sets of the 2nd iteration (GEN_2, $KILL_2$) are as follows:

  $GEN_2(B_0) = GEN_1(B_0)$; Same as first as it's IN set is the same

$GEN_2(B_1) = \{b \rightarrow z\}$

$GEN_2(B_2) = \{a \rightarrow x\}$; As p → a, and *p = b, a → d ∋ b → d would be generated (which is a → {}), but it would be killed internally by a → x

The following table shows the sets in each iteration till fixed point (calculating outs' before ins'):

	in_1	out_1	in_2	out_2	in_3	out_3
B_0	{}	{p → a}	{}	{p → a}	{}	{p → a}
B_1	{p → a}	{b → z}	{p → a}	{p → a, b → z}	{p → a}	{p → a, b → z}
B_2	{p → a}	{a → x}	{p → a}	{a → x, p → a}	{p → a}	{a → x, p → a}
B_3	{b → z, a → x}	{}	{p → a, b → z, a → x}	{p → a, b → z, a → x}	{p → a, b → z, a → x}	{p → a, b → z, a → x}

- The order in which the basic blocks are processed influences the running time, in the above example we have processed them in a breadth first order. They could also be processed in a depth first order.
- We could also propagate only the differences in the IN, OUT sets for faster running time.

Design Decisions

- Heap modelling: We could assume that calls like *malloc* use only a finite number of locations for allocating memory, this assumption is useful when *malloc* is in a loop and the total number of locations allocated is only known at runtime.
- Set Implementation: We could implement sets using a sequence of bits, each bit representing an element. A set bit means the corresponding element is present.

- Call graph, function pointers: This refers to the interdependence between performing context sensitive analysis and pointer analysis for function pointers (Refer week 3 scribes section 2.5)
- Array indices: When an assignment is made to *a[i]*, we could either assume that it affects the elements at all the indices of the array, or keep track of *i*'s value using other analyses and modify the points-to set of the respective elements accordingly (this would require us to treat each element in the array as a different variable).
- Analysis Dimensions: We need to model our analysis based on our requirements of time, memory and precision. To get better at one of them we need to tradeoff the other two.

Analysis Dimensions

Flow Sensitivity

- A flow sensitive analysis keeps track of the points-to set of each pointer at the entry and exit of each basic block in the program.
- A flow insensitive analysis would output a single points-to set for each pointer for the entire program. It is a sound, conservative, possibly imprecise approximation of flow sensitive analysis.
- A flow insensitive analysis is equivalent to a flow sensitive analysis where every basic block is connected to every other basic block and every statement is in a separate basic block.
- The flow insensitive solution can be computed by ordering the basic blocks in some cyclical order, and repeatedly applying the transfer function without killing points-to information (i.e. OUT(B) = GEN(B) ∪ IN(B), in the case of reaching definitions analysis and pointer analysis), till there is no change in the IN and OUT sets.
- An example showing the difference between the solutions:

  Flow sensitive solution:

  At L0, a → x, and

  at L1, a → y.

```
L0: a = &x;
L1: a = &y;
```

Flow insensitive solution:

a → {x, y} in the program.

Context Sensitivity

- In a context sensitive solution, the points-to sets of the appropriate variables are specified at each calling context of every function. The time and space complexity of a context sensitive solution can be exponential in the number of functions.
- In a context insensitive solution, the points-to of appropriate variables are specified for each function, ignoring the calling context. The solution is a sound, conservative and imprecise approximation of the context sensitive solution. Space complexity is always linear in the number of functions.
- A context insensitive solution can be interprocedural or intraprocedural. In interprocedural analysis the values passed to a procedure by other procedures are tracked and used. Functions are treated independent of other functions in intraprocedural analysis.
- For example, in the program:

```
main() {
   L0: fun(&x);
   L1: fun(&y);
}

fun(int *a) {
   b = a;
}
```

  - Context Sensitive solution:
    b → x along main → fun at L0
    b → y along main → fun at L1.
  - Context insensitive solution:
    - Interprocedural: b → {x,y} in the program.
    - Intraprocedural: b → ⊥

Path Sensitivity

- A path sensitive solution considers the conditions in if statements, for loops and while loops (unlike a flow sensitive and path insensitive solution which does not use the conditions but considers the branching of control flow at the statements). The points-to sets are given for all possible paths of execution of the program.
- Example:

```
if (a == 0)
   b = &x;
else
   b = &y;
```

- Path sensitive: b → x when a is 0, b → y otherwise.
- Path insensitive: b → {x,y} in the program.
- Practically the length of the path is not very high and we could use a finite length for performing a limited path sensitive analysis.
- But for end goals like formally verifying that a program meets its specifications, we need to use a strict path sensitive approach.

Field Sensitivity

- In a field sensitive solution, the points-to sets of all the fields of aggregate data structures (like structs, unions, arrays) are found.
- The field insensitive solution merges the fields into a single variable, this reduces the number of variables tracked during the analysis and reduces precision.
- Example:

```
struct T S;
s.a = &x;
s.b = &y;
```

- Field sensitive: s.a → x, s.b → y
- Field insensitive: s → {x,y}

For typed pointer analysis with different sized types, we could keep track of the size of the memory locations pointed to, along with the addresses, as *partial alias* could arise (corresponding memory locations could partially overlap).

Andersen's analysis

- Flow sensitive analysis is computationally expensive, hence we need a faster algorithm, trading precision for speed.
- It is a subset based, constraint based, flow insensitive analysis.
- It considers the set of all the statements in the program, and for each statement $a = b$, a constraint $ptsto(a) \supseteq ptsto(b)$ is added.
- The above constraints should be satisfied minimally to get the points-to sets of the variables.
- To get the solution, the statements are arbitrarily ordered and the initial points-to sets are set to {}, we iterate over each statement in the considered order, for each statement, we add minimal elements to the points-to set of the pointer on the LHS, so that the constraint associated with the statement is satisfied. Once this is over, we check if the obtained solution satisfies the constraint, if it doesn't, we repeat the iteration over all statements till it does (which it will as the number of pointers is finite and every pointer pointing to every other pointer is a solution and the size of at least one points-to set would increase in an iteration if the obtained solution in that iteration doesn't satisfy the constraints).

Q) How is it different from flow insensitive DFA analysis ?

A) Flow insensitive DFA analysis gives a partially flow insensitive solution as the order of statements within basic blocks is preserved, andersen's analysis is a completely flow insensitive solution as it does not consider any order amongst the statements.

- Example 1 (discussed in class):

Program	Constraints

`a = &x;`
`b = &y;`
`p = &a;`
`c = b;`
`*p = c;`
	ptsto(a) ⊇ {x}
	ptsto(b) ⊇ {y}
	ptsto(p) ⊇ {a}
	ptsto(c) ⊇ ptsto(b)
	ptsto(*p) ⊇ ptsto(c)

Fixed point

Pointers	Iteration 0	Iteration 1	Iteration 2
a	{}	{x, y}	
b	{}	{y}	
c	{}	{y}	
p	{}	{a}	
x	{}		
y	{}		

After the 1st iteration, a should point to x and y due to constraints **ptsto(a) ⊇ {x}** and **ptsto(*p) ⊇ ptsto(c)** (ptsto(a) ⊇ ptsto(b) ⊇ {y}). This is imprecise as in flow sensitive analysis a → x would have been killed by a → y.

- Example 2 (Last two statements interchanged):

Program	Constraints

`a = &x;`
`b = &y;`
`p = &a;`
`*p = c;`
`c = b;`
	ptsto(a) ⊇ {x}
	ptsto(b) ⊇ {y}
	ptsto(p) ⊇ {a}
	ptsto(*p) ⊇ ptsto(c)
	ptsto(c) ⊇ ptsto(b)

Fixed point

Pointers	Iteration 0	Iteration 1	Iteration 2	Iteration 3
a	{}	{x}	{x, y}	
b	{}	{y}	{y}	
c	{}	{y}	{y}	
p	{}	{a}	{a}	
x	{}			

y	{}			

From this example, it can verified that the order of the statements does not affect the final solution obtained. (This could be because there is a unique minimal solution to the problem)

The order of statements could affect the efficiency though, as if *def(x)* is after *use(x)*, the extra pointee due to *def(x)* would only reach *use(x)* in the next iteration.

- Classwork:

Program	Constraints

`*p = c;`
`b = &y;`
`b = *p;`
`p = &a;`
`a = &x;`
`*p = c;`
`c = p;`
`c = &z;`
	ptsto(*p) ⊇ ptsto(c)
	ptsto(b) ⊇ {y}
	ptsto(b) ⊇ ptsto(*p)
	ptsto(p) ⊇ {a}
	ptsto(a) ⊇ {x}
	ptsto(*p) ⊇ ptsto(c)
	ptsto(c) ⊇ ptsto(p)
	ptsto(c) ⊇ {z}

Fixed point

Pointers	Iteration 0	Iteration 1	Iteration 2	Iteration 3
a	{}	{x}	{a, x, z}	
b	{}	{y}	{a, x, y, z}	
c	{}	{a, z}	{a, z}	
p	{}	{a}	{a}	
x	{}			
y	{}			

Optimizations

- Duplicates can be removed (as it is a set of statements).
- Constraints can be reordered to reduce the number of iterations. (like shifting *def(x)* before *use(x)*; this might not be possible to do for all variables if there are cyclic dependencies. For example: a = b; b = a;).
- Address-of constraints need to be processed only once as they would not add any pointees to the sets in later iterations.
- Propagate differences in the sets after the end of iteration k-1 and k to iteration k+1, **ptsto(x)** at the end of iteration k+1 would be $\mathbf{ptsto_{k+1}(x)} \cup \mathbf{ptsto_{k}(x)} \cup \mathbf{ptsto_{k-1}(x)} \cup \mathbf{...} \cup \mathbf{ptsto_{1}(x)}$.

An optimized version of the above classwork program:

Program	Optimized version

`*p = c;`
`b = &y;`
`b = *p;`
`p = &a;`
`a = &x;`
`*p = c;`
`c = p;`
`c = &z;`
	`b = &y;`
	`p = &a;`
	`a = &x;`
	`c = p;`
	`c = &z;`
	`*p = c;`
	`b = *p;`

Fixed point is reached after one iteration for the optimized version.

Complexity

- Space complexity: $O(n^2)$, (theoretically worst case occurs when every pointer points-to every other pointer, $n * (n-1)$ space is required).
- Time complexity: Between any two pointers **a**, **b**, for the constraint **a = b**, we propagate at most $O(n)$ information (if **b** was pointing to all other pointers), at most $O(n)$ times (if **b**'s points-to set was increasing by 1 each time), there could be at most $O(n^2)$ constraints in the program (**c * n²**, where **c** > 2, due to the load and store operations), which results in a $O(n^4)$ complexity.

- Complexity when using difference propagation: If we propagate only the differences, then we would propagate at most $O(n)$ information (no propagation of information if points-to set remains the same) between any two pointers, so complexity would be $O(n^3)$
- Practically, as points-to sets are sparse, it is observed that the time complexity is close to $O(n^2)$ or $O(n * log(n))$.

Steensgaard's Analysis

- Though Andersen's analysis is faster than flow sensitive analysis, for sufficiently large code, even it is computationally expensive.
- Steensgaard's analysis is faster and more imprecise than Andersen's analysis.
- It is a unification based, flow insensitive analysis.
- For each statement **p = q** in the program, **p**'s points-to set and **q**'s points-to set are merged.
- Time complexity is almost linear: $O(n * \alpha(n))$, where $\alpha(n)$ is the inverse ackermann function.
- The process of finding the points-to set can be visualized as merging nodes in a graph whose nodes are sets containing variables and edges represent points-to information.
- Assumptions:
    - Every pointer initially points-to a non-empty set. (**p → {*p}**)
    - Two points-to edges cannot emanate from a node in the graph.

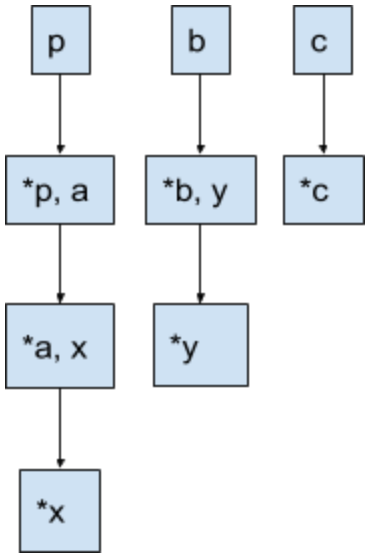
- Example:

Program	Andersen's analysis	Steensgaard's analysis

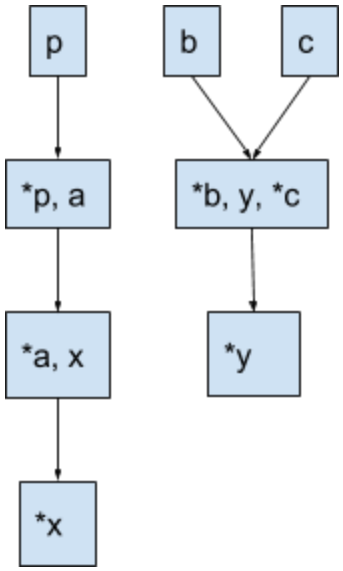
`a = &x;`
`b = &y;`
`p = &a;`
`c = b;`
`*p = c;`
	a → {x, y}
	b → {y}
	c → {y}
	p → {a}
		a → {x, y}
		b → {x, y}
		c → {x, y}
		p → {a}

Pointers	Iteration 0	Iteration 1
a	{*a}	{*a, *b, *c, x, y}
b	{*b}	{*a, *b, *c, x, y}
c	{*c}	{*a, *b, *c, x, y}
p	{*p}	{*p, a}
x	{*x}	
y	{*y}	

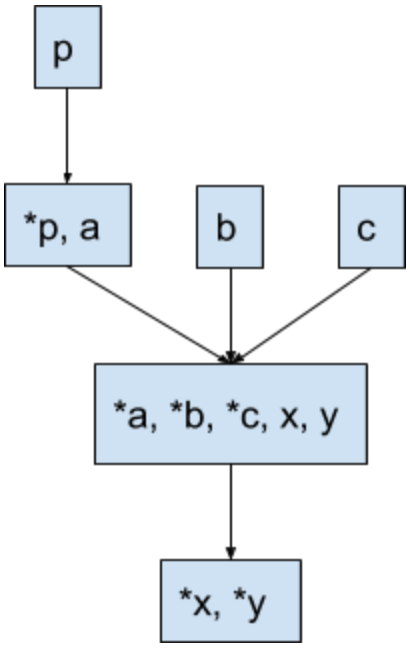
- The pointees in iteration 0 (***a, ...**) can be deleted after the final result is computed.
- After **p = &a** is processed, the graph would be **{p → {*p, a} → {*a, x} → {*x}}, {b → {y, *b} → {*y} }, {c → {*c}}, ...**



- After **c = b** is processed, c and b's points-to nodes would get merged and the graph would be **{p → {*p, a} → {*a, x} → {*x}}, {b → {y, *b, *c} → {*y}}**, c points-to the node which **b** points-to.



- After ***p = c** is processed, the node which ***p** points-to **{*a, x}** is merged with the node which **c** points-to (**{y, *b, *c}**), to get the graph **{p → {*p, a} → {*a, *b, *c, x, y} → {*x, *y}}**, **b** would also point to this merged node.


p
*p, a
b
c
*a, *b, *c, x, y
*x, *y