Pointer analysis

1

Pointer Analysis

- Outline:
  - What is pointer analysis
  - Intraprocedural pointer analysis
  - Interprocedural pointer analysis
    - Andersen and Steensgaard

2

Pointer and Alias Analysis

- Aliases: two expressions that denote the same memory location.
- Aliases are introduced by:
  - pointers
  - call-by-reference
  - array indexing
  - C unions

3

Useful for what?

- Improve the precision of analyses that require knowing what is modified or referenced (eg const prop, CSE …)
- Eliminate redundant loads/stores and dead stores.

```
x := *p;
...
y := *p; // replace with y := x?
```

```
*x := ...;
// is *x dead?
```

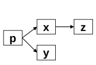
- Parallelization of code
  - can recursive calls to quick_sort be run in parallel? Yes, provided that they reference distinct regions of the array.
- Identify objects to be tracked in error detection tools

```
x.lock();
...
y.unlock(); // same object as x?
```

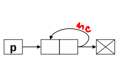
4

Kinds of alias information

- Points-to information (must or may versions)
  - at program point, compute a set of pairs of the form p ! x, where p points to x.
  - can represent this information in a **points-to graph**



- Alias pairs
  - at each program point, compute the set of of all pairs (e_1,e_2) where e_1 and e_2 must/may reference the same memory.
- Storage shape analysis
  - at each program point, compute an abstract description of the pointer structure.



5

Intraprocedural Points-to Analysis

- Want to compute may-points-to information
- Lattice: $D = 2^{\{x \to y \mid x \in Var, y \in Var\}}$

$\sqcup = \cup$

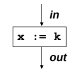
$\sqsubseteq = \subseteq$

$\bot = \emptyset$

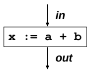
$\top = \{x \to y \mid x \in Var, y \in Var\}$

6

Flow functions



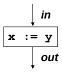
$F_{x := k}(in) =$



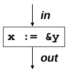
$F_{x := a+b}(in) =$

7

Flow functions



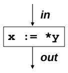
$F_{x := y}(in) =$



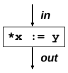
$F_{x := \&y}(in) =$

8

Flow functions



$F_{x := *y}(in) =$



$F_{*x := y}(in) =$

9

Intraprocedural Points-to Analysis

- Flow functions:

$$
\begin{aligned}
kill(x) &= \bigcup_{v \in Vars} \{(x, v)\} \\
F_{x:=k}(S) &= S - kill(x) \\
F_{x:=a+b}(S) &= S - kill(x) \\
F_{x:=y}(S) &= S - kill(x) \cup \{(x, v) \mid (y, v) \in S\} \\
F_{x:=\&y}(S) &= S - kill(x) \cup \{(x, y)\} \\
F_{x:=*y}(S) &= S - kill(x) \cup \{(x, v) | \exists t \in Vars.[(y, t) \in S \wedge (t, v) \in S]\} \\
F_{*x:=y}(S) &= \textbf{let } V := \{v \mid (x, v) \in S\} \textbf{ in} \\
&\quad S - (\textbf{if } V = \{v\} \textbf{ then } kill(v) \textbf{ else } \emptyset) \\
&\quad\quad \cup \{(v, t) \mid v \in V \wedge (y, t) \in S\}
\end{aligned}
$$

10

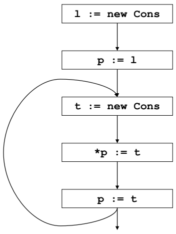
Pointers to dynamically-allocated memory

- Handle statements of the form: **`x := new T`**
- One idea: generate a new variable each time the new statement is analyzed to stand for the new location:

$$F_{x:=new\ T}(S) = S - kill(x) \cup \{(x, newvar())\}$$

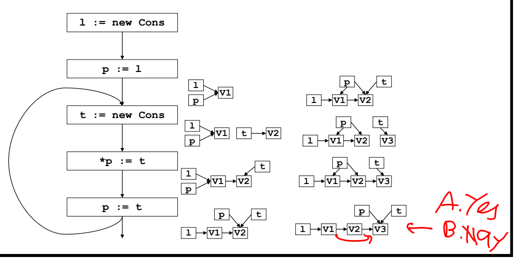
11

Example



12

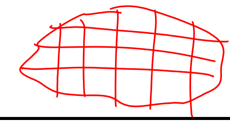
Example solved



13

What went wrong?

- Lattice infinitely tall!
- We were essentially running the program
- Instead, we need to summarize the infinitely many allocated objects in a finite way
- **New Idea**: introduce summary nodes, which will stand for an entire set of allocated objects.



14

What went wrong?

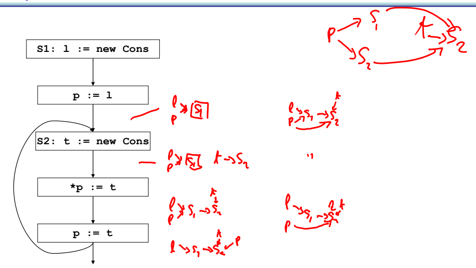
- Example: For each new statement with label L, introduce a summary node loc_L, which stands for the memory allocated by statement L.

$$F_{L:\ x:=new\ T}(S) = S - kill(x) \cup \{(x, loc_L)\}$$

- Summary nodes can use other criterion for merging.

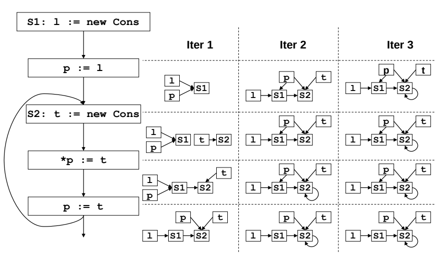
15

Example revisited



16

Example revisited & solved



18

Array aliasing, and pointers to arrays

- Array indexing can cause aliasing:
  - `a[i]` aliases `b[j]` if:
    - a aliases b and i = j
    - a and b overlap, and i = j + k, where k is the amount of overlap.
- Can have pointers to elements of an array
  - `p := &a[i]; ...; p++;`
- How can arrays be modeled?
  - Could treat the whole array as one location.
  - Could try to reason about the array index expressions: array dependence analysis.

19

Fields

- Can summarize fields using per field summary
  - for each field F, keep a points-to node called F that summarizes all possible values that can ever be stored in F
- Can also use allocation sites
  - for each field F, and each allocation site S, keep a points-to node called (F, S) that summarizes all possible values that can ever be stored in the field F of objects allocated at site S.

20

Summary

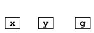
- We just saw:
  - intraprocedural points-to analysis
  - handling dynamically allocated memory
  - handling pointers to arrays
- But, intraprocedural pointer analysis is not enough.
  - Sharing data structures across multiple procedures is one the big benefits of pointers: instead of passing the whole data structures around, just pass pointers to them (eg C pass by reference).
  - So pointers end up pointing to structures shared across procedures.
  - If you don't do an interproc analysis, you'll have to make conservative assumptions functions entries and function calls.

21

Conservative approximation on entry

- Say we don't have interprocedural pointer analysis.
- What should the information be at the input of the following procedure:

```
global g;
void p(x,y) {

   ...

}
```

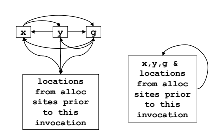


22

Conservative approximation on entry

- Here are a few solutions:

```
global g;
void p(x,y) {

   ...

}
```



- They are all very conservative!
- We can try to do better.

23

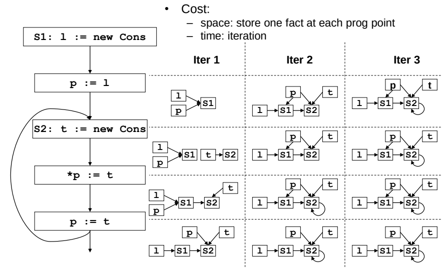
Interprocedural pointer analysis

- Main difficulty in performing interprocedural pointer analysis is scaling
- A single points-to-graph can be O(size of program)

24

Example revisited

- Cost:
  - space: store one fact at each prog point
  - time: iteration



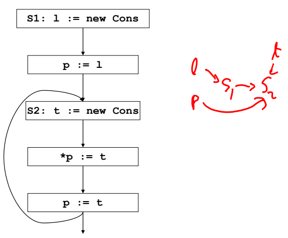
25

New idea: store one dataflow fact

- Store one dataflow fact for the whole program
- Each statement updates this one dataflow fact
  - use the previous flow functions, but now they take the whole program dataflow fact, and return an updated version of it.
- Process each statement once, ignoring the order of the statements
- This is called a flow-insensitive analysis.

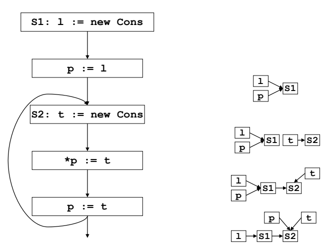
26

Flow insensitive pointer analysis



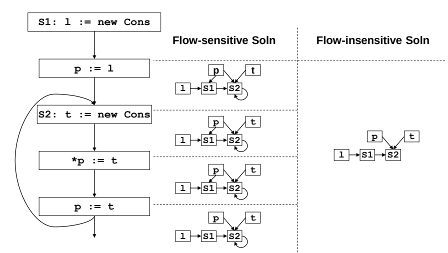
27

Flow insensitive pointer analysis



28

Flow sensitive vs. insensitive



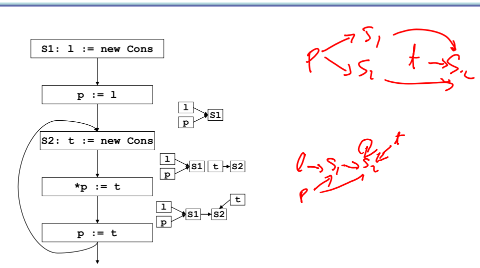
29

What went wrong?

- What happened to the link between p and S1?
  - Can't do strong updates anymore!
  - Need to remove all the kill sets from the flow functions.
- What happened to the self loop on S2?
  - We still have to iterate!

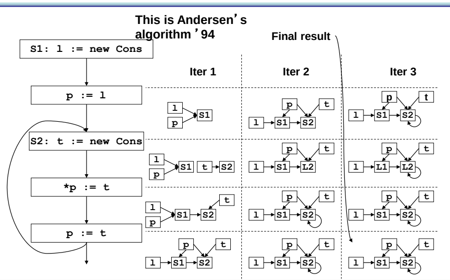
30

Flow insensitive pointer analysis: fixed



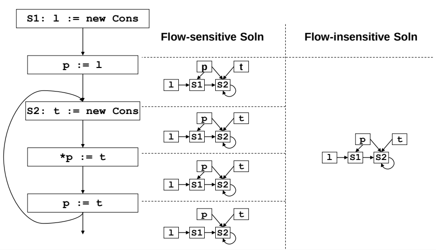
31

Flow insensitive pointer analysis: fixed



32

Flow sensitive vs. insensitive, again



33

Flow insensitive loss of precision

- Flow insensitive analysis leads to loss of precision!

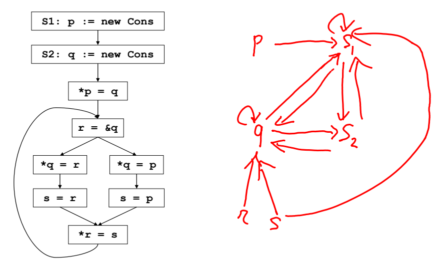
```
main() {
   x := &y;

   ...

   x := &z;
}
```

Flow insensitive analysis tells us that x may point to z here!

- However:
  - uses less memory (memory can be a big bottleneck to running on large programs)
  - runs faster

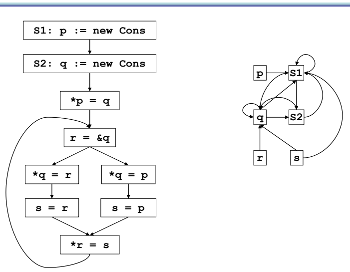
34

In Class Exercise!



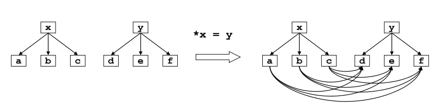
35

In Class Exercise! solved



36

Worst case complexity of Andersen

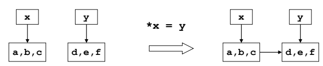


Worst case: N^2 per statement, so at least N^3 for the whole program. Andersen is in fact $O(N^3)$

37

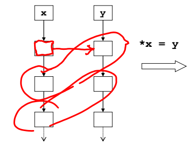
New idea: one successor per node

- Make each node have only one successor.
- This is an invariant that we want to maintain.



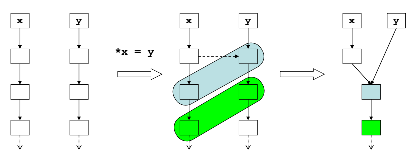
38

More general case for `*x = y`

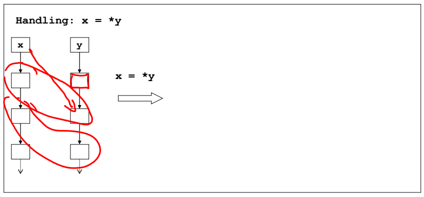


39

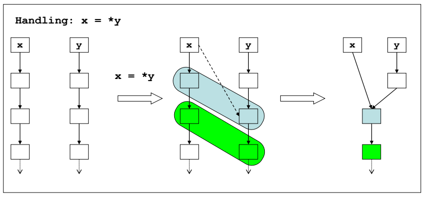
More general case for `*x = y`



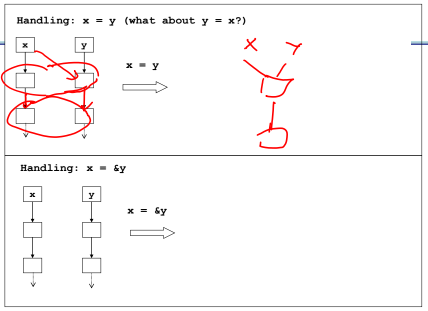
40



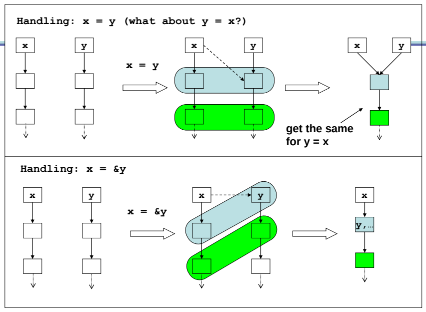
41



42

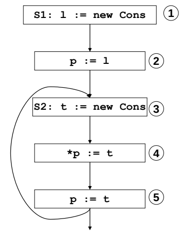


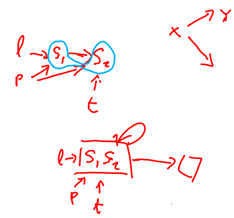
43



44

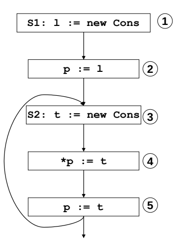
Our favorite example, once more!

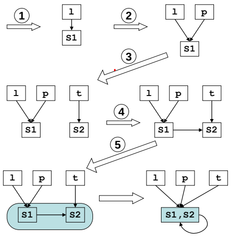




45

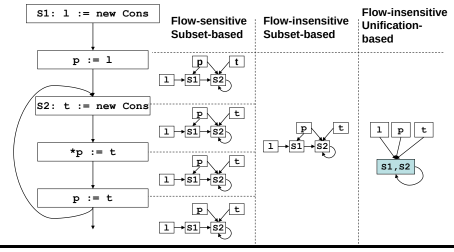
Our favorite example, once more!





46

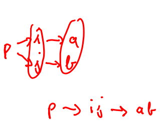
Flow insensitive loss of precision



47

Another example

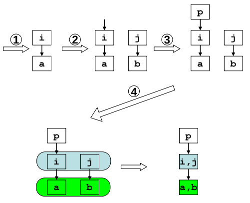
```
bar() {
 ① i := &a;
 ② j := &b;
 ③ foo(&i);
 ④ foo(&j);
   // i pnts to what?
   *i := ...;

}

void foo(int* p) {
   printf("%d",*p);
}
```



48

Another example

```
bar() {
 ① i := &a;
 ② j := &b;
 ③ foo(&i);
 ④ foo(&j);
   // i pnts to what?
   *i := ...;

}

void foo(int* p) {
   printf("%d",*p);
}
```



49

Almost linear time

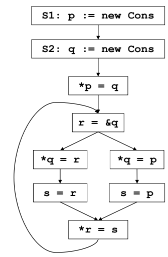
- Time complexity: O(Nα(N, N))

  inverse Ackermann function

- So slow-growing, it is basically linear in practice
- For the curious: node merging implemented using UNION-FIND structure, which allows set union with amortized cost of O(α(N, N)) per op. Take CSE 202 to learn more!

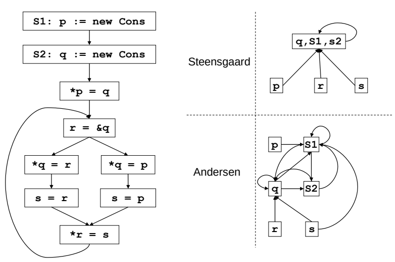
50

In Class Exercise!



51

In Class Exercise! solved



52

Advanced Pointer Analysis

- Combine flow-sensitive/flow-insensitive
- Clever data-structure design
- Context-sensitivity

53