

Pointer analysis

Pointer Analysis

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

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

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.

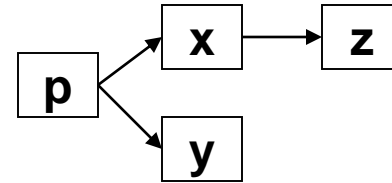
```
x := *p;                               *x := ...;
...                                       // is *x dead?
y := *p; // replace with y := x?
```

- 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?
```

Kinds of alias information

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

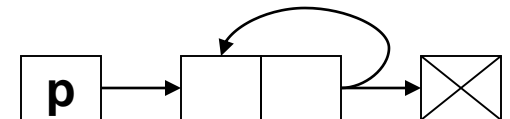


$\{ p \rightarrow x, x \rightarrow z, p \rightarrow y \}$

- Alias pairs
 - at each program point, compute the set 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.



Intraprocedural Points-to Analysis

- Want to compute may-points-to information

- Lattice: $\mathbb{D} = 2^{\{x \rightarrow y \mid x \in \text{Var}, y \in \text{Var}\}}$

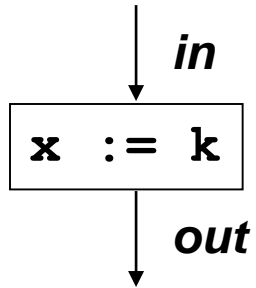
$$\perp = \emptyset$$

$$\sqsubseteq = \subseteq$$

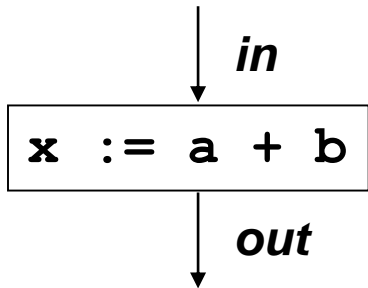
$$\top = \text{Var} \rightarrow \text{Var}$$

$$\mathbb{D} = \{x \rightarrow y \mid x \in \text{Var}, y \in \text{Var}\}$$

Flow functions

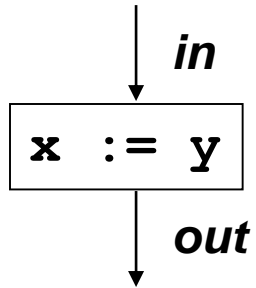


$$F_{x := k}(\text{in}) =$$

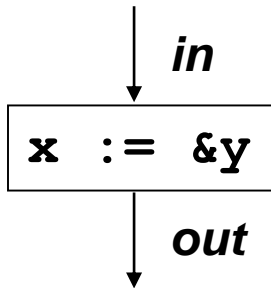


$$F_{x := a+b}(\text{in}) =$$

Flow functions

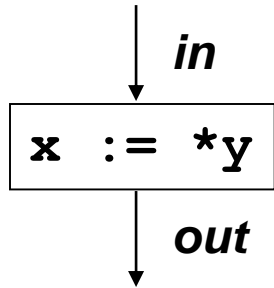


$$F_{x := y}(\text{in}) =$$

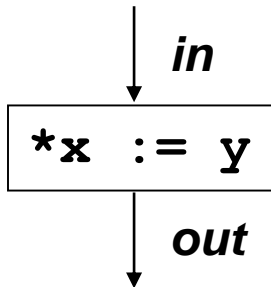


$$F_{x := \&y}(\text{in}) =$$

Flow functions



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



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

Intraprocedural Points-to Analysis

- Flow functions:

$$kill(x) = \bigcup_{v \in Vars} \{(x, v)\}$$

$$F_{x:=k}(S) = S - kill(x)$$

$$F_{x:=a+b}(S) = S - kill(x)$$

$$F_{x:=y}(S) = S - kill(x) \cup \{(x, v) \mid (y, v) \in S\}$$

$$F_{x:=\&y}(S) = S - kill(x) \cup \{(x, y)\}$$

$$F_{x:=*y}(S) = S - kill(x) \cup \{(x, v) \mid \exists t \in Vars. [(y, t) \in S \wedge (t, v) \in S]\}$$

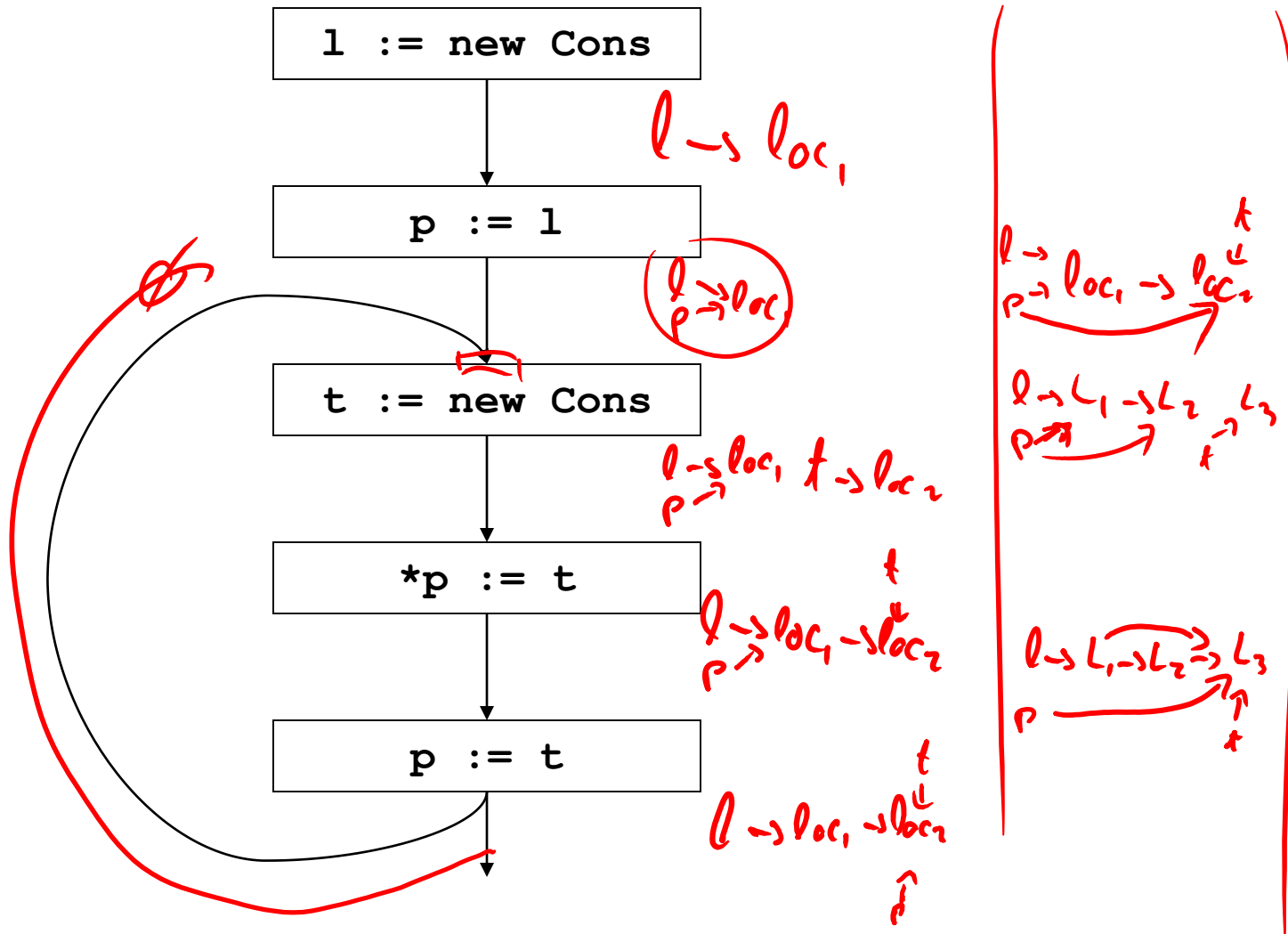
$$F_{*x:=y}(S) = \text{let } V := \{v \mid (x, v) \in S\} \text{ in} \\ S - (\text{if } V = \{v\} \text{ then } kill(v) \text{ else } \emptyset) \\ \cup \{(v, t) \mid v \in V \wedge (y, t) \in S\}$$

Pointers to dynamically-allocated memory

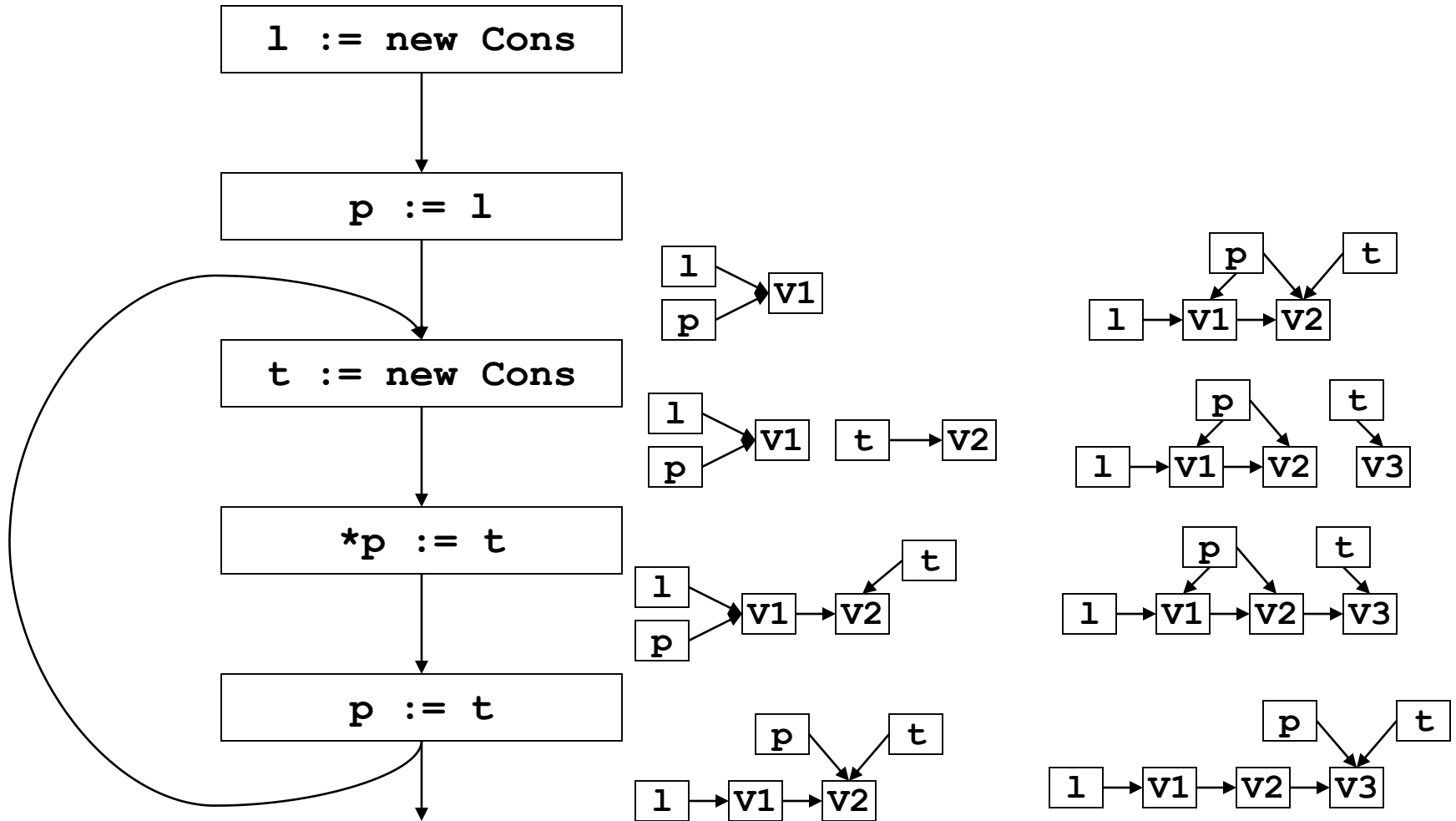
- Handle statements of the form: **$x := \text{new } T$**
- One idea: generate a new variable each time the new statement is analyzed to stand for the new location:

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

Example

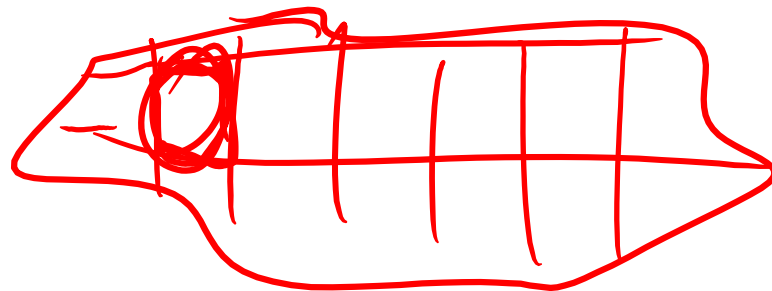


Example solved



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 a whole class of allocated objects.



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.

Example revisited

S1: l := new Cons

p := l

S2: t := new Cons

*p := t

p := t

$l \rightarrow S_1$
 $p \rightarrow$

$l \rightarrow S_1$ $t \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

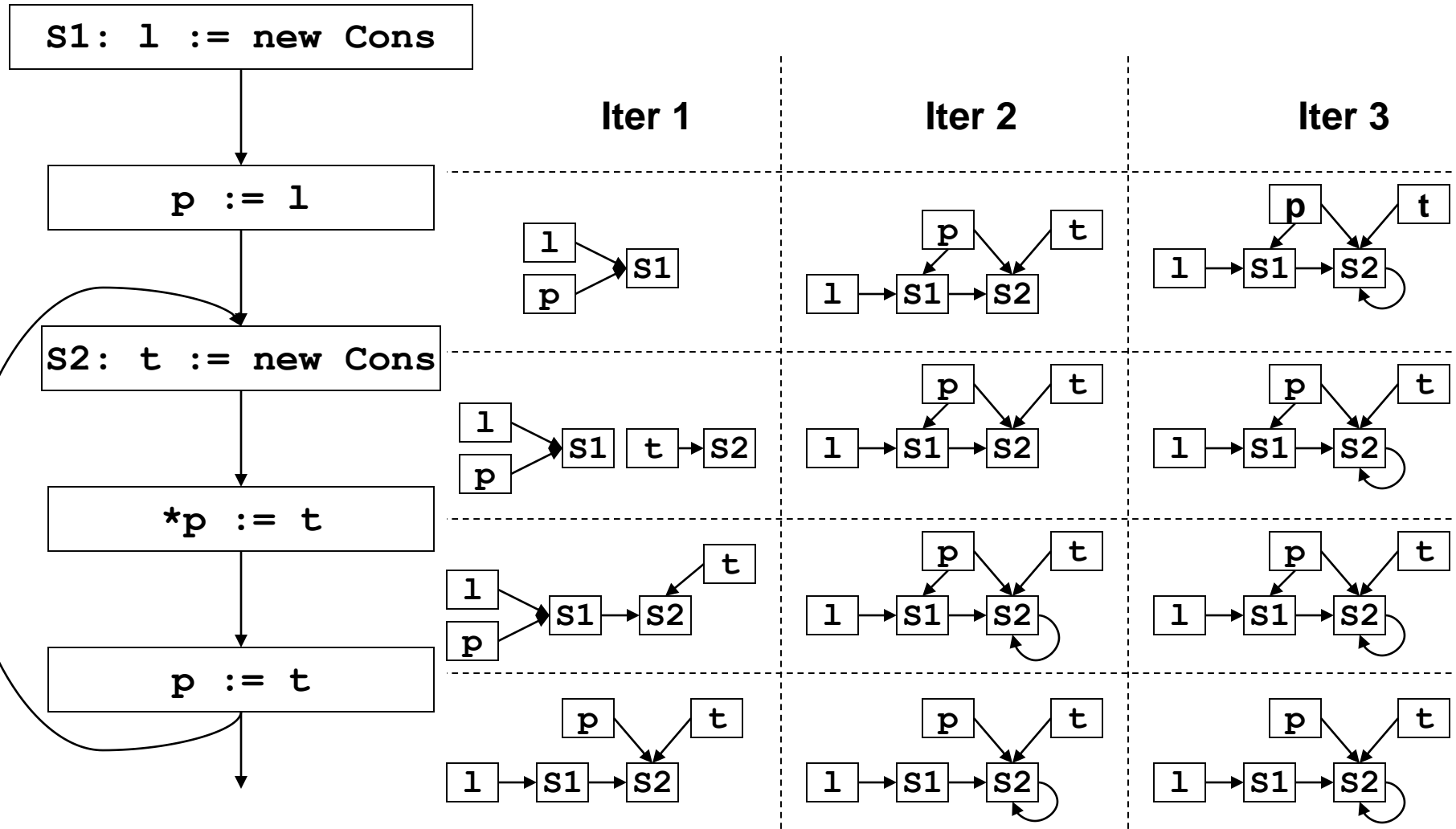
$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

$l \rightarrow S_1 \rightarrow S_2$
 $p \rightarrow$

Example revisited & solved



Array aliasing, and pointers to arrays

- Array indexing can cause aliasing:
 - $\mathbf{a[i]}$ aliases $\mathbf{b[j]}$ if:
 - \mathbf{a} aliases \mathbf{b} and $i = j$
 - \mathbf{a} and \mathbf{b} overlap, and $i = j + k$, where k is the amount of overlap.
- Can have pointers to elements of an array
 - `$\mathbf{p := \&a[i]; \dots; p++}$` ;
- 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.

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 .

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.

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) {  
    ...  
}
```

`x`

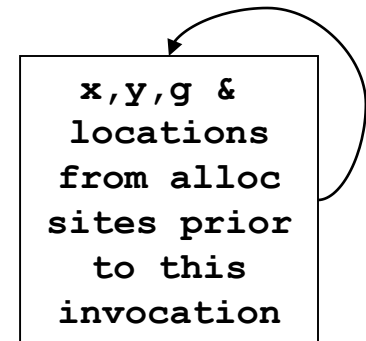
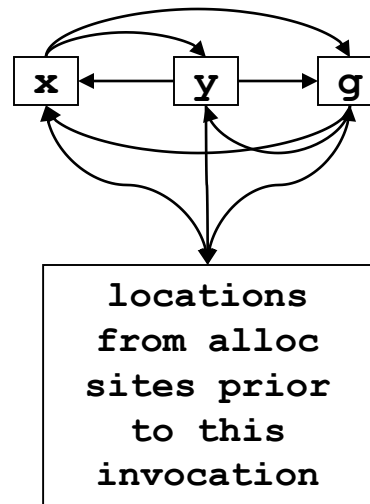
`y`

`g`

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.

Interprocedural pointer analysis

- Main difficulty in performing interprocedural pointer analysis is scaling
- One can use a top-down summary based approach (Wilson & Lam 95), but even these are hard to scale

Example revisited

150 TB

$5M \times 314 \times 100$

• Cost:

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

S1: l := new Cons

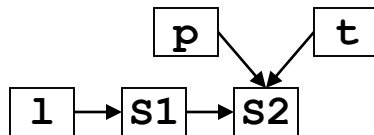
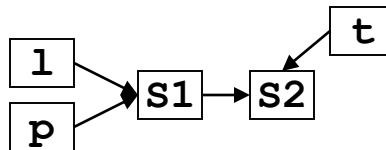
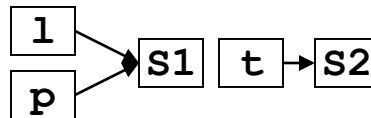
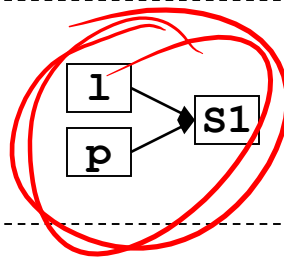
p := l

S2: t := new Cons

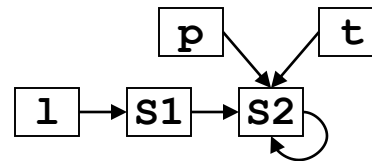
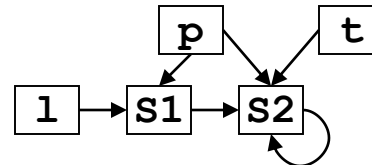
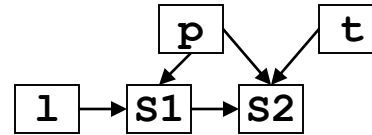
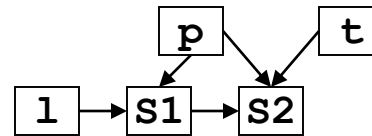
*p := t

p := t

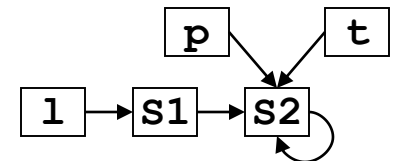
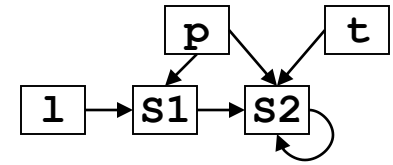
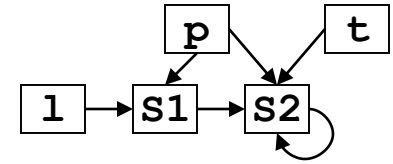
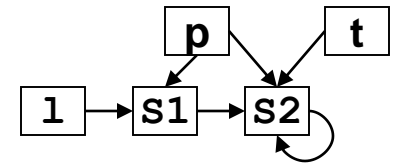
Iter 1



Iter 2



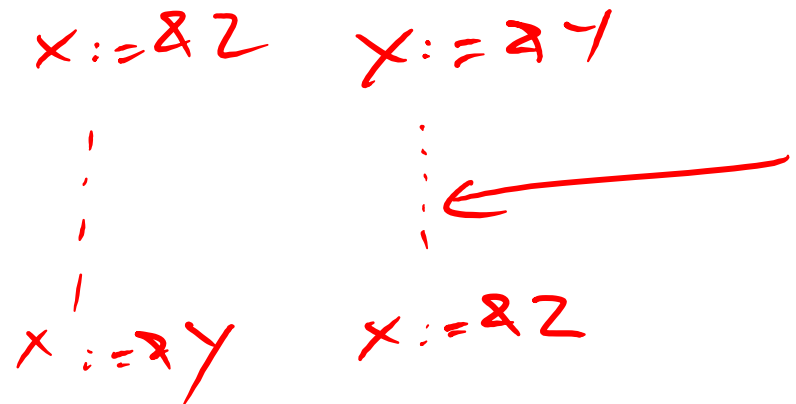
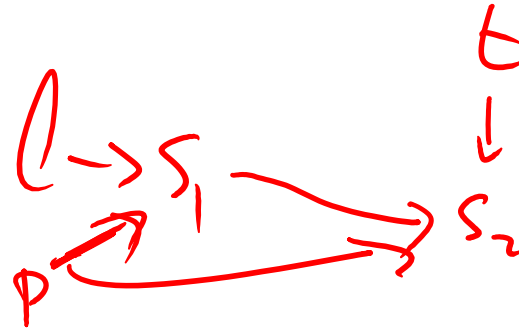
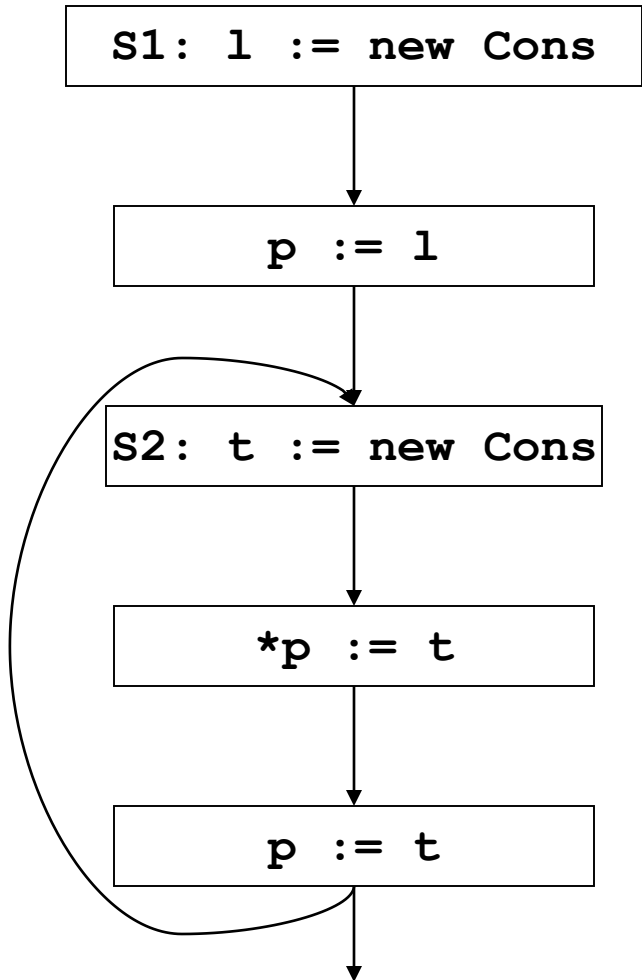
Iter 3



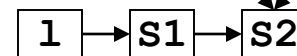
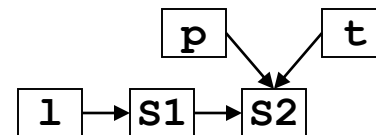
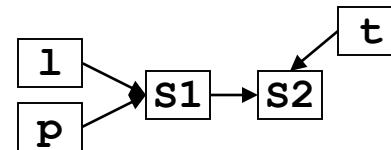
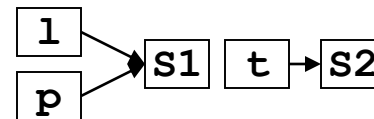
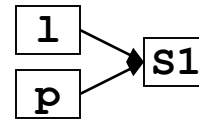
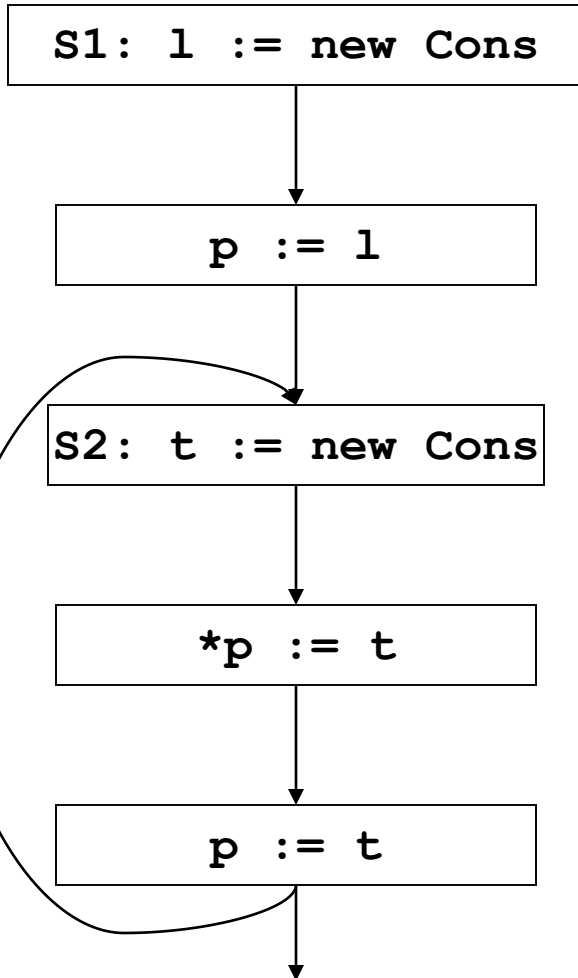
New idea: store one dataflow fact

- 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.

Flow insensitive pointer analysis



Flow insensitive pointer analysis



Flow sensitive vs. insensitive

S1: l := new Cons

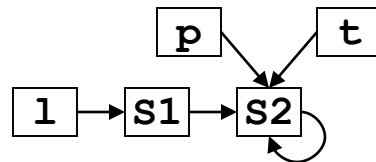
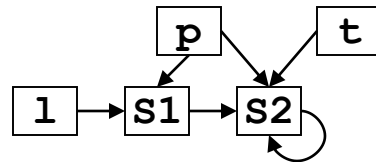
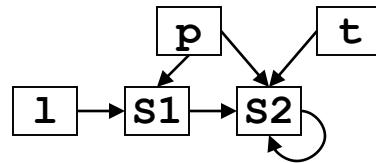
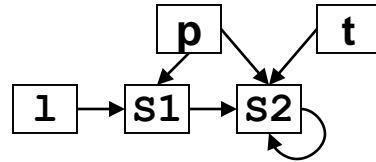
p := l

S2: t := new Cons

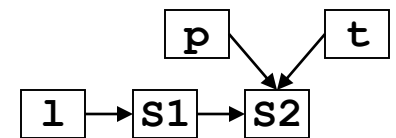
*p := t

p := t

Flow-sensitive Soln



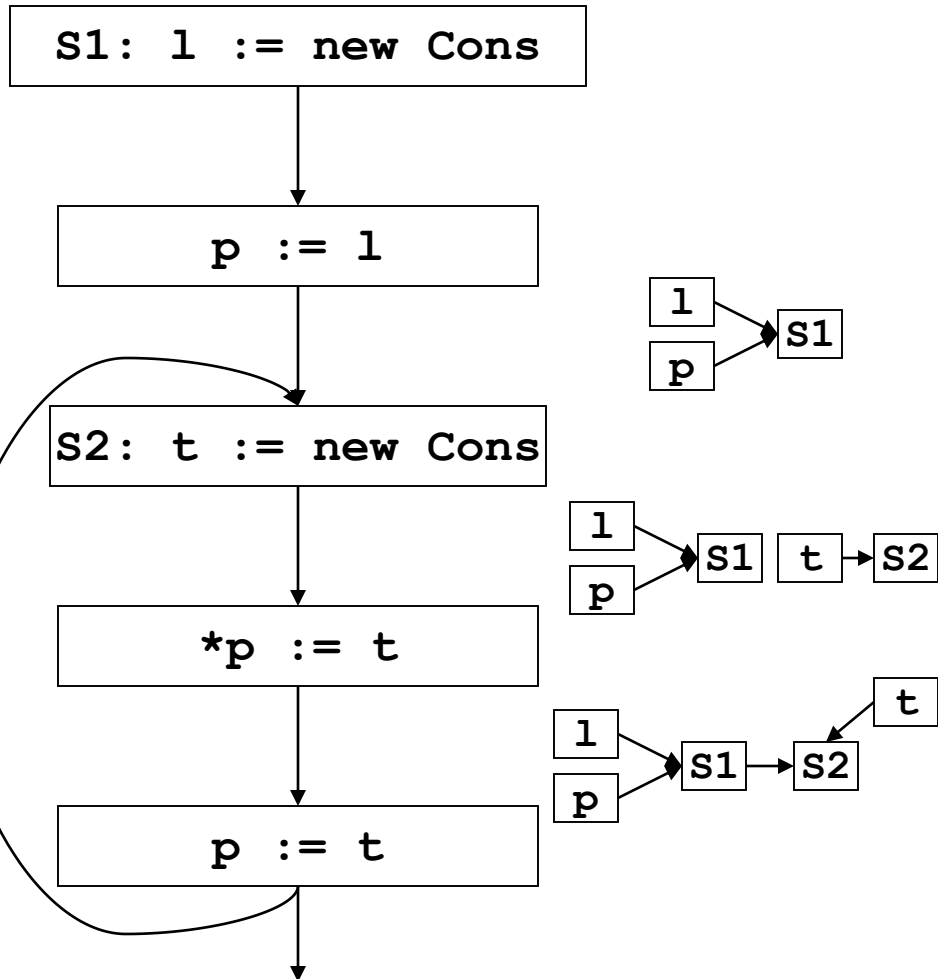
Flow-insensitive Soln



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!

Flow insensitive pointer analysis: fixed



Flow insensitive pointer analysis: fixed

This is Andersen's algorithm '94

Final result

S1: l := new Cons

p := l

S2: t := new Cons

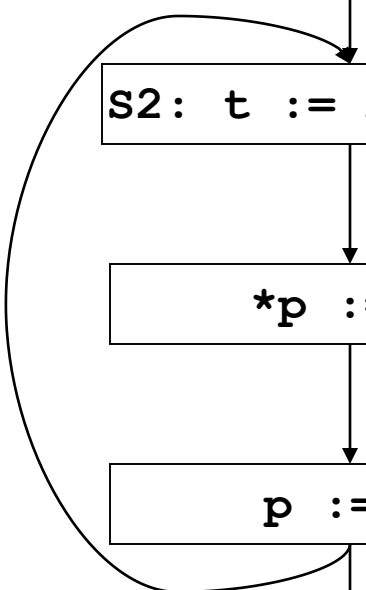
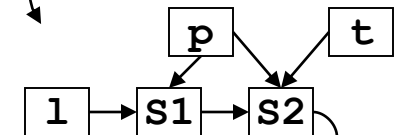
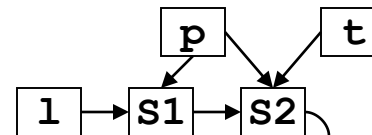
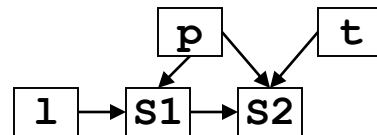
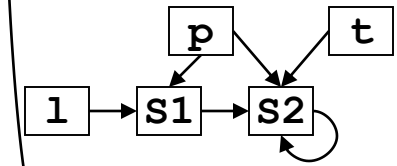
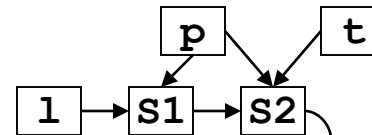
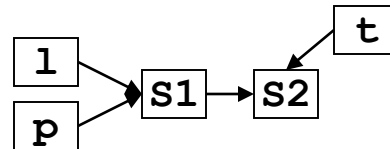
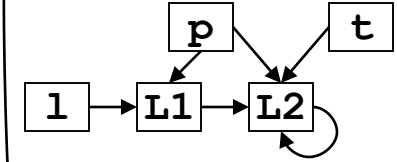
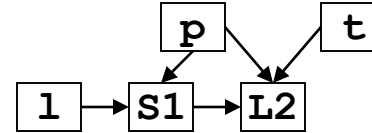
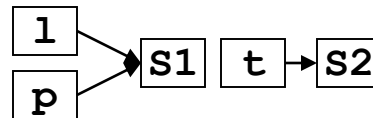
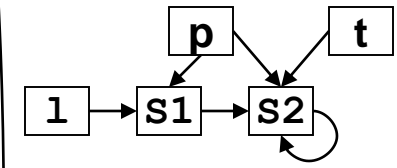
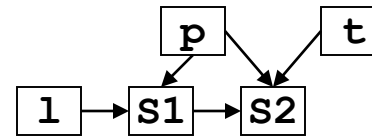
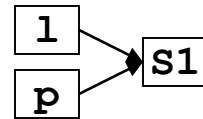
*p := t

p := t

Iter 1

Iter 2

Iter 3



Flow sensitive vs. insensitive, again

S1: l := new Cons

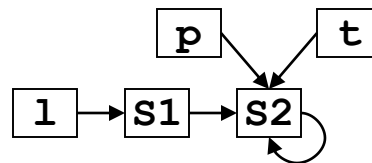
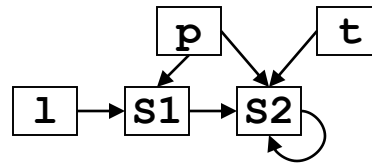
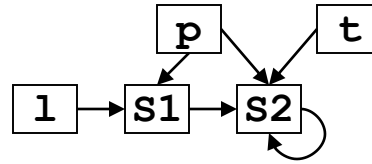
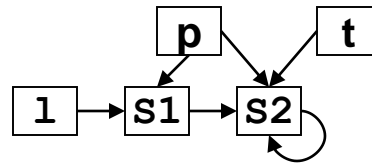
p := l

S2: t := new Cons

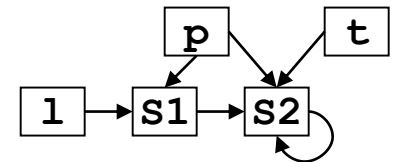
*p := t

p := t

Flow-sensitive Soln



Flow-insensitive Soln



Flow insensitive loss of precision

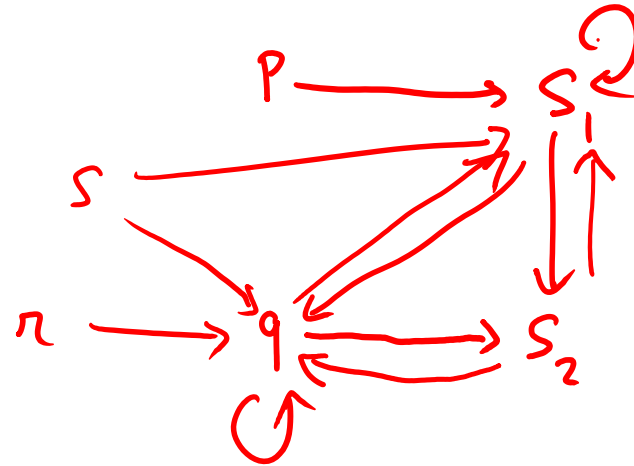
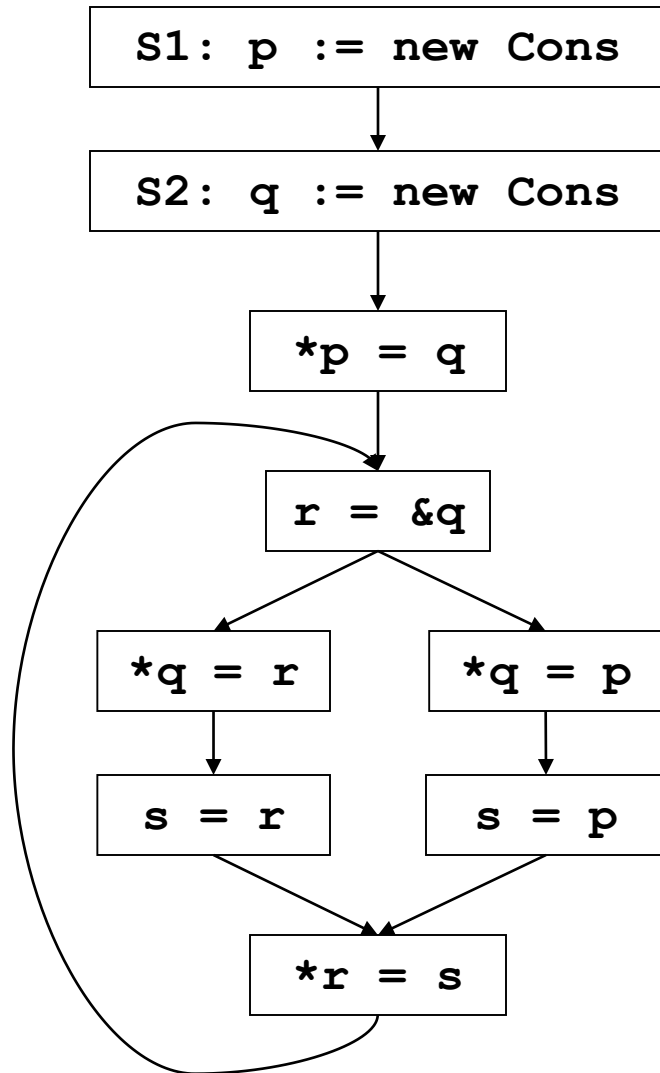
- Flow insensitive analysis leads to loss of precision!

```
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

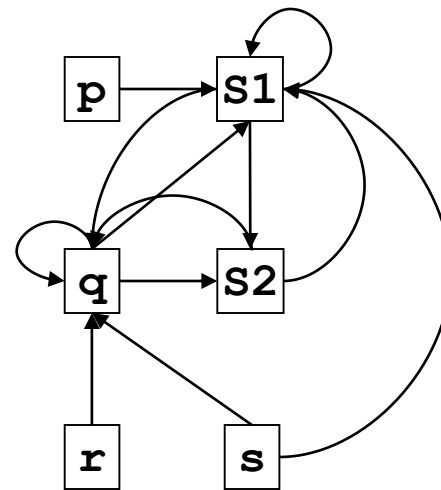
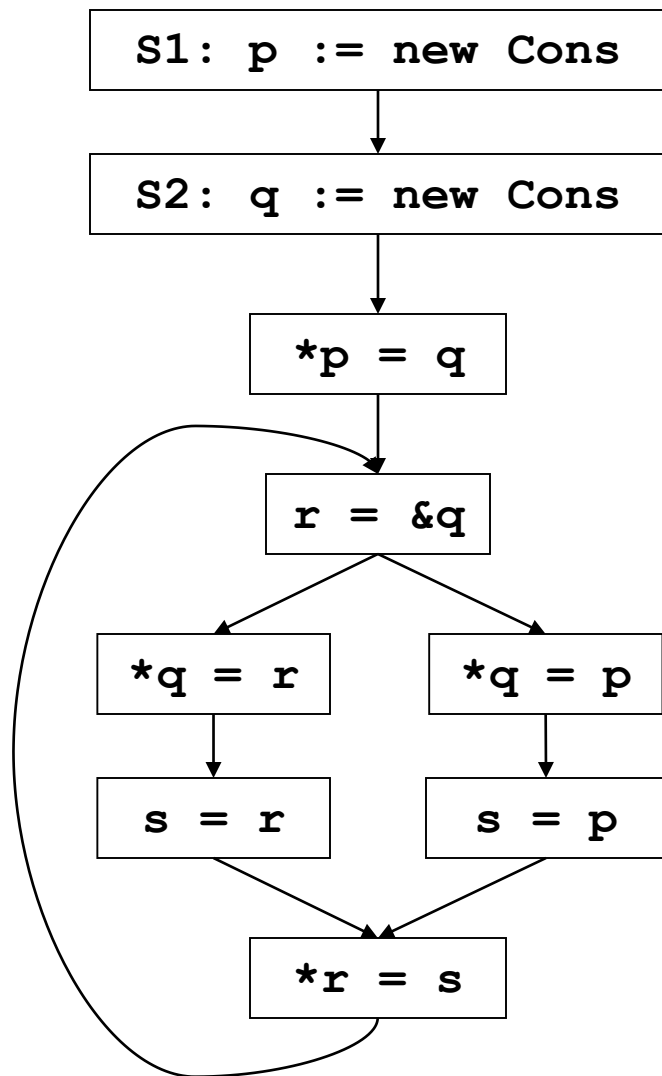
In Class Exercise!



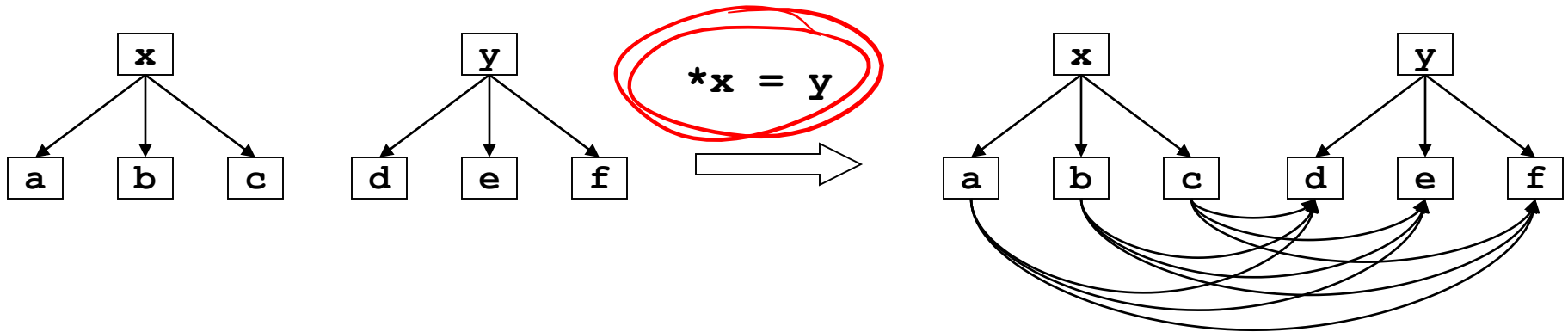
```
f() {  
  A  
  B  
  C  
}
```

```
while(  
  f()  
)
```

In Class Exercise! solved



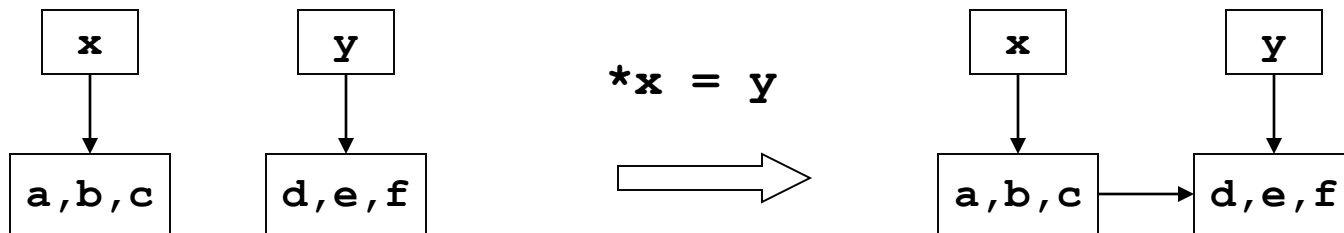
Worst case complexity of Andersen



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

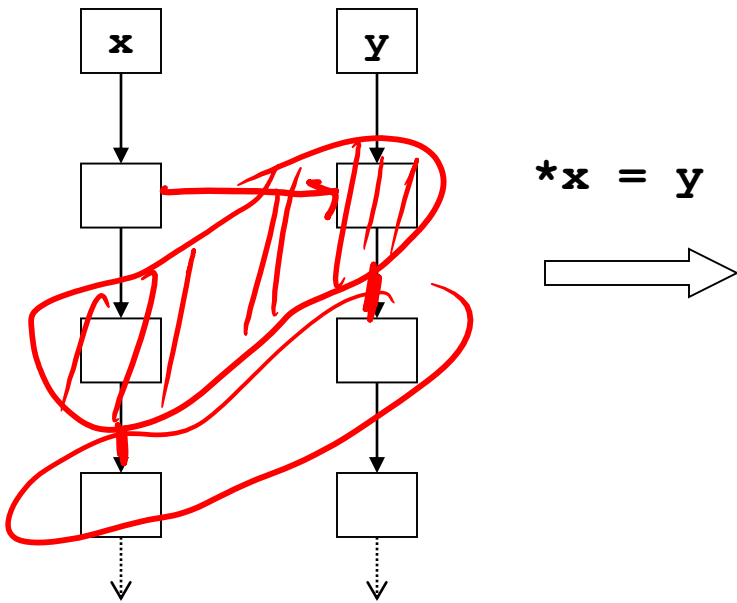
New idea: one successor per node

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

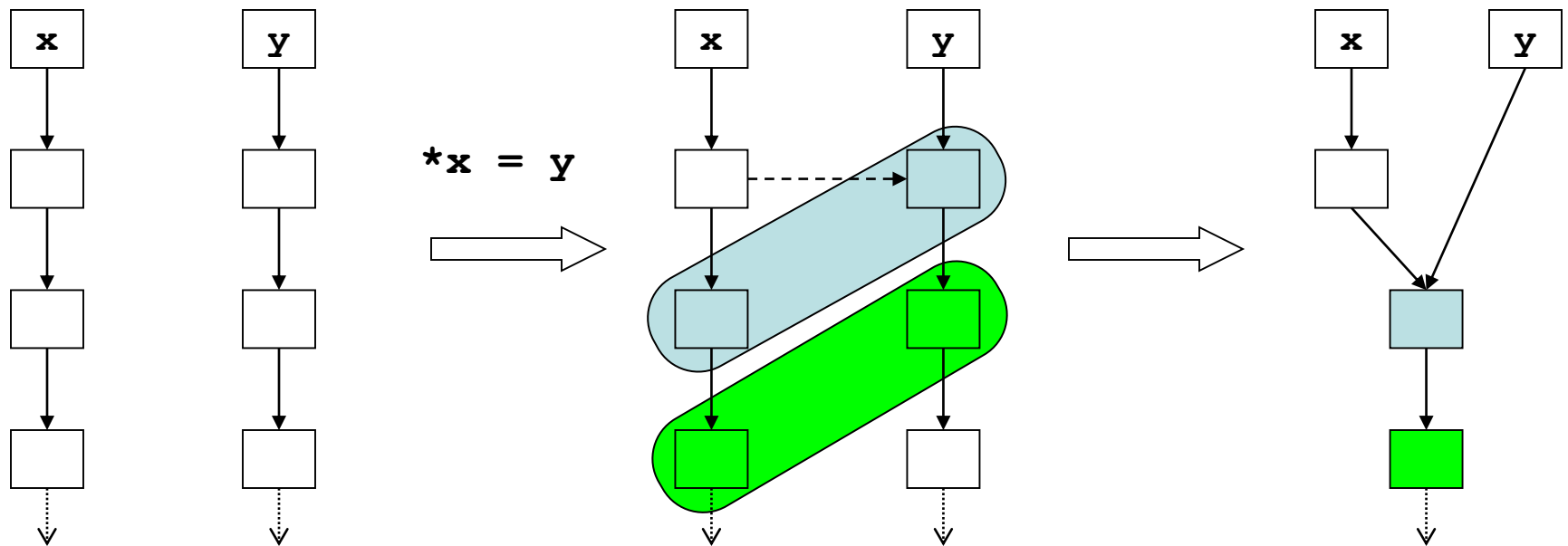


~~$x, y, 2$~~

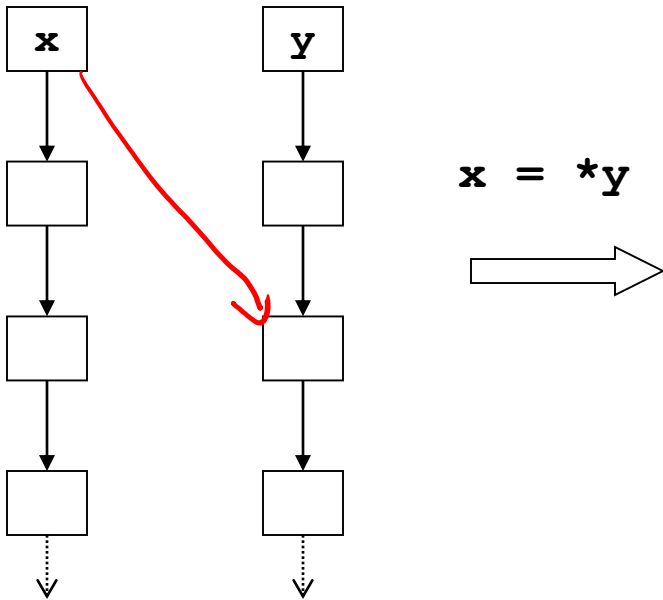
More general case for $*x = y$



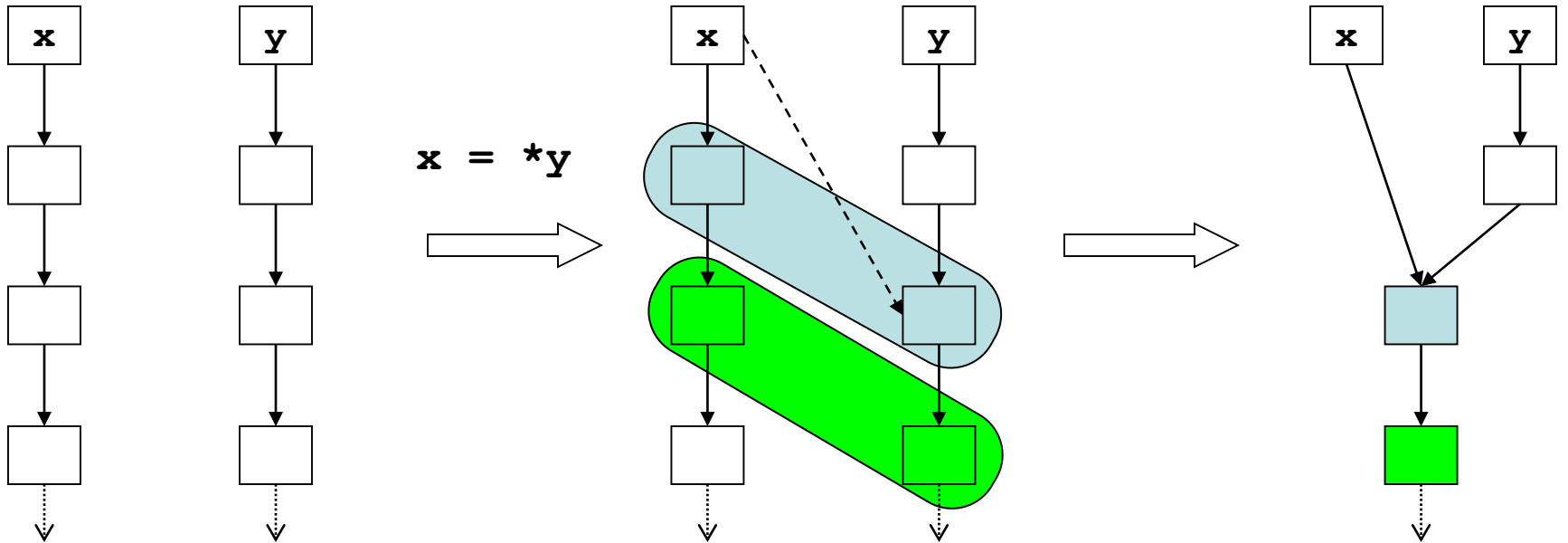
More general case for $*x = y$



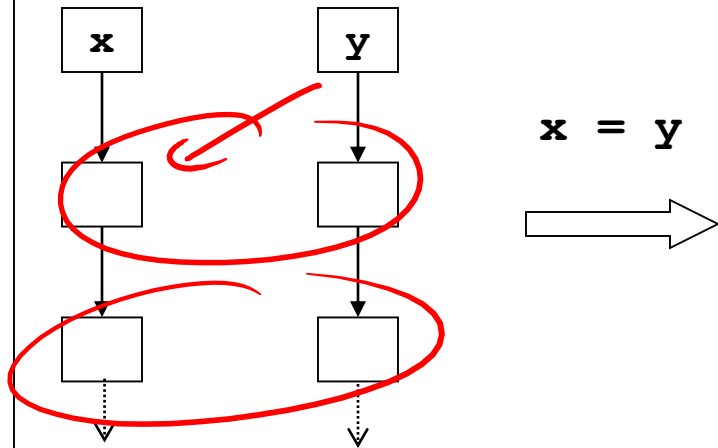
Handling: $x = *y$



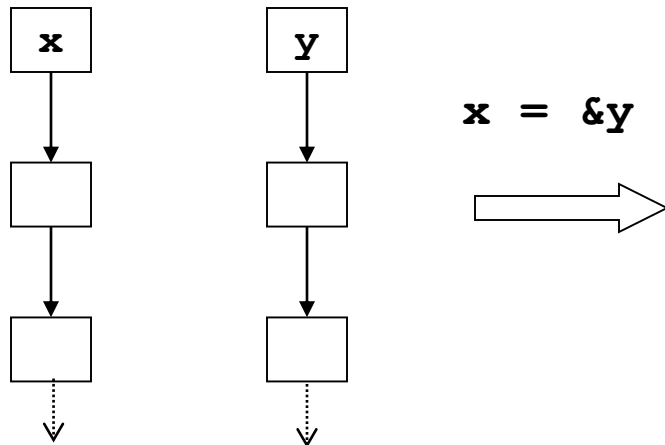
Handling: $x = *y$



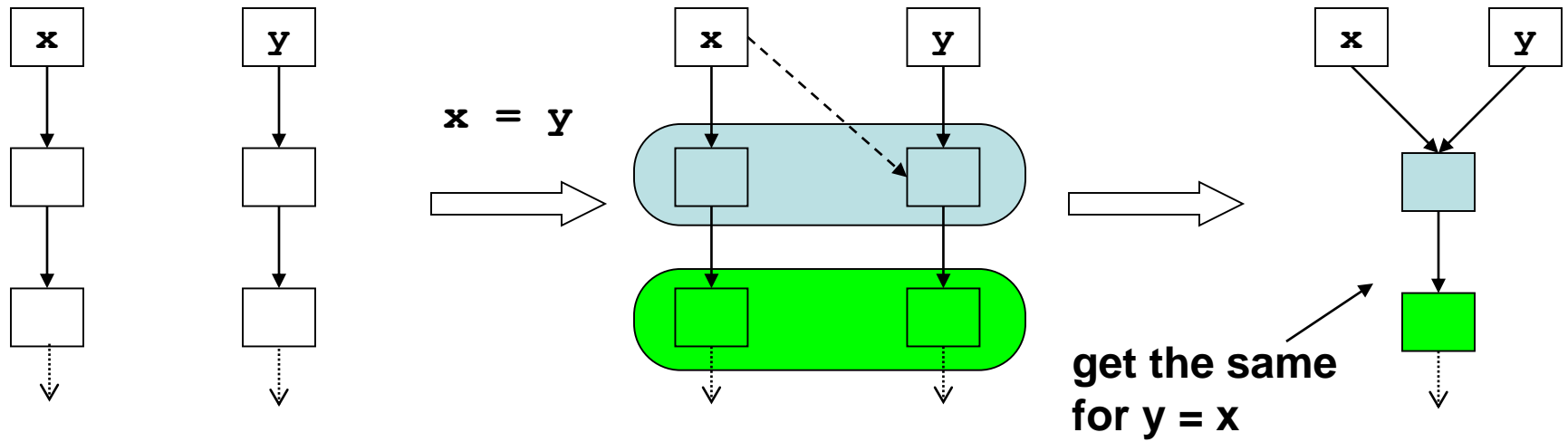
Handling: $x = y$ (what about $y = x$?)



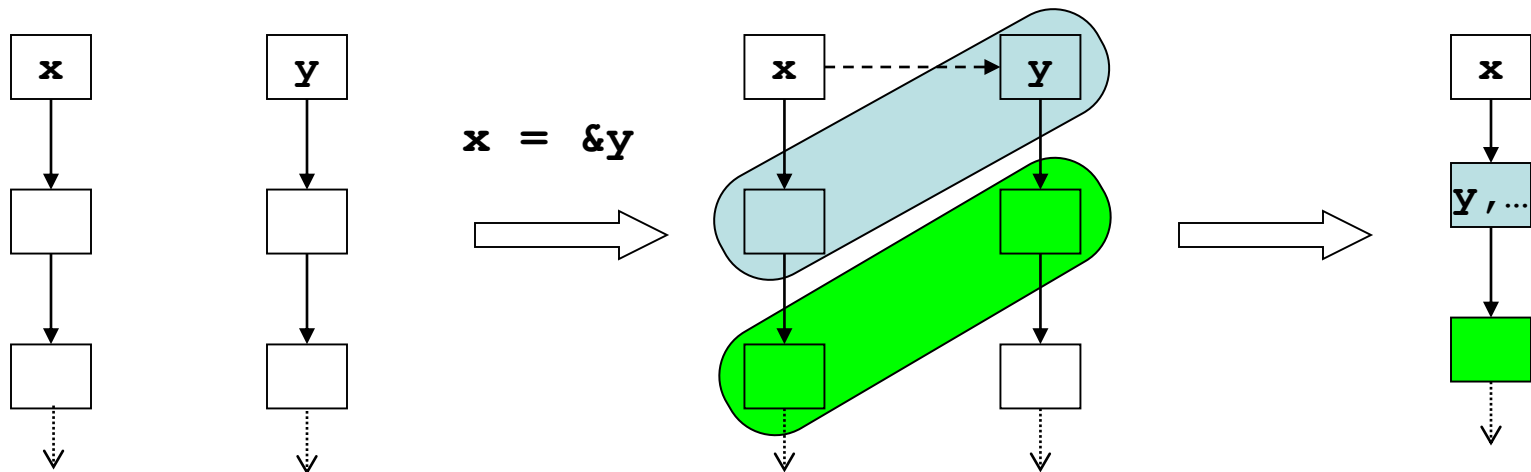
Handling: $x = \&y$



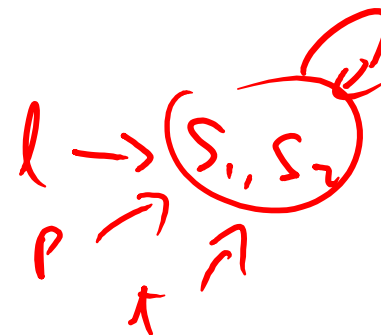
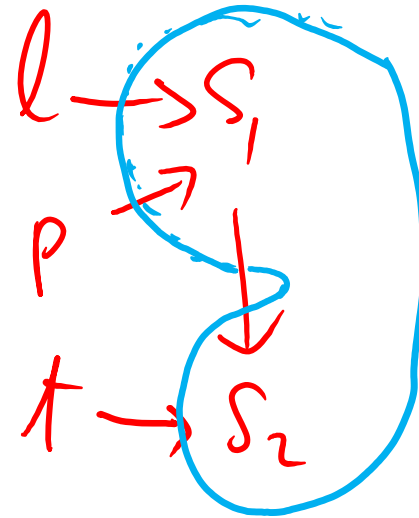
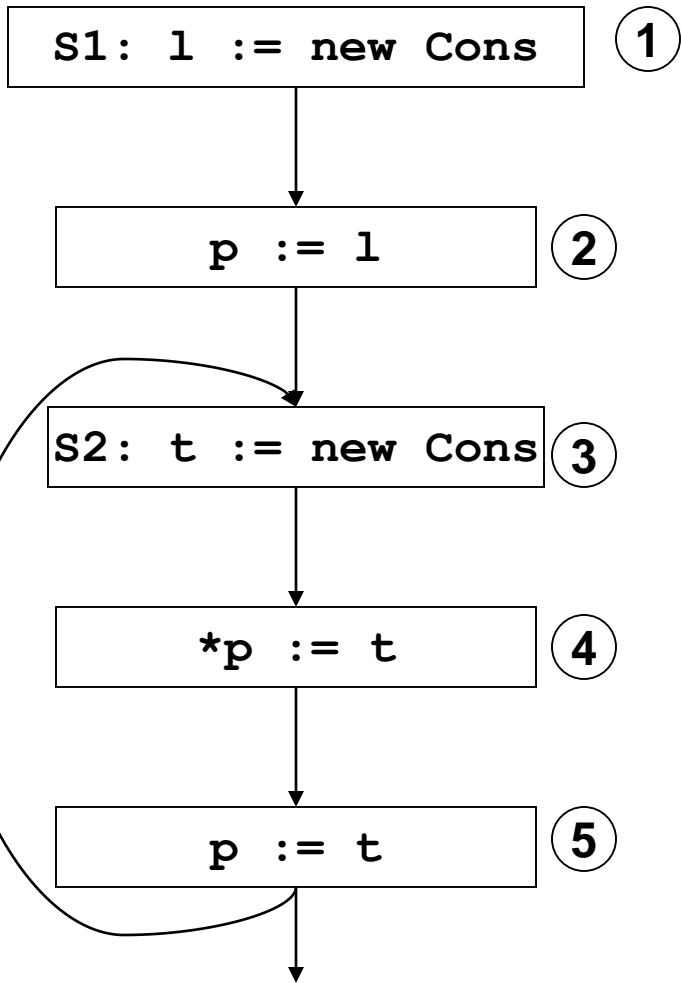
Handling: $x = y$ (what about $y = x$?)



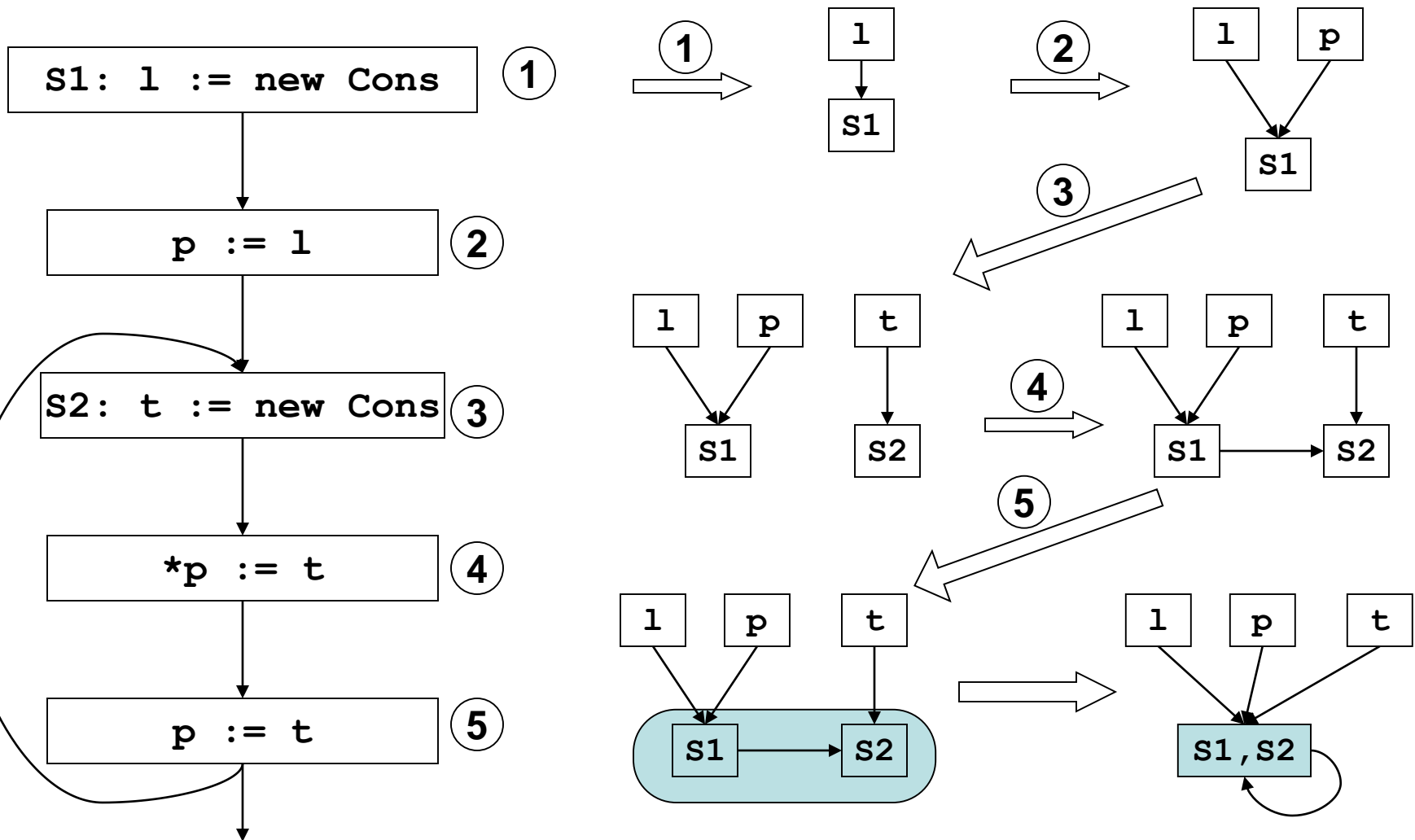
Handling: $x = \&y$



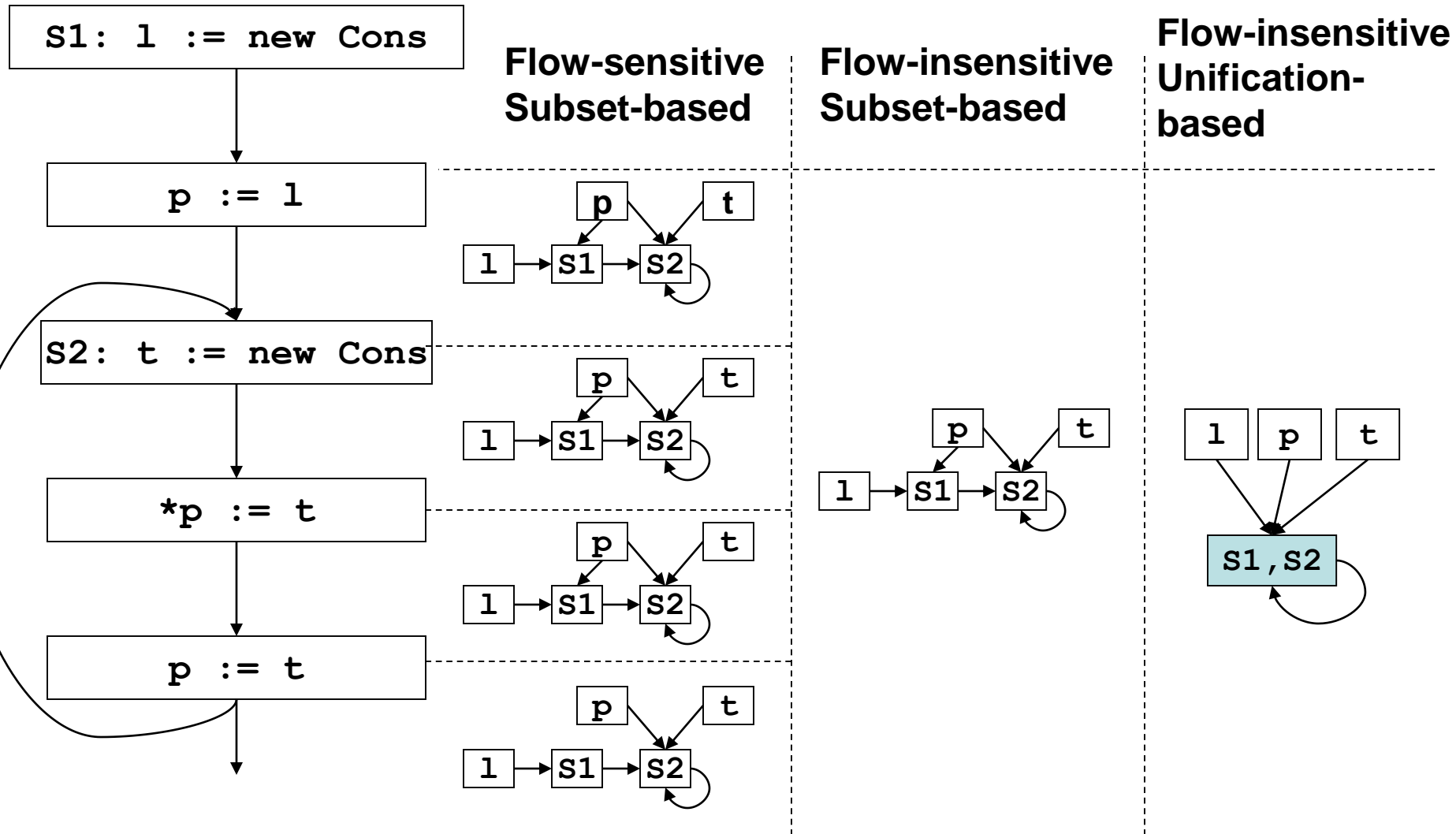
Our favorite example, once more!



Our favorite example, once more!



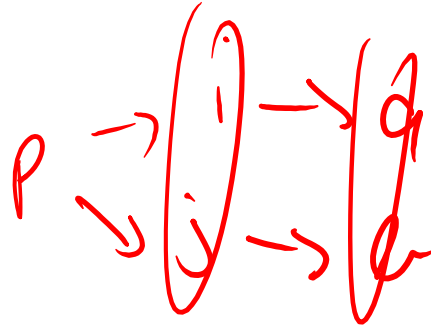
Flow insensitive loss of precision



Another example

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

$p = \&i$
 $p = \&j$

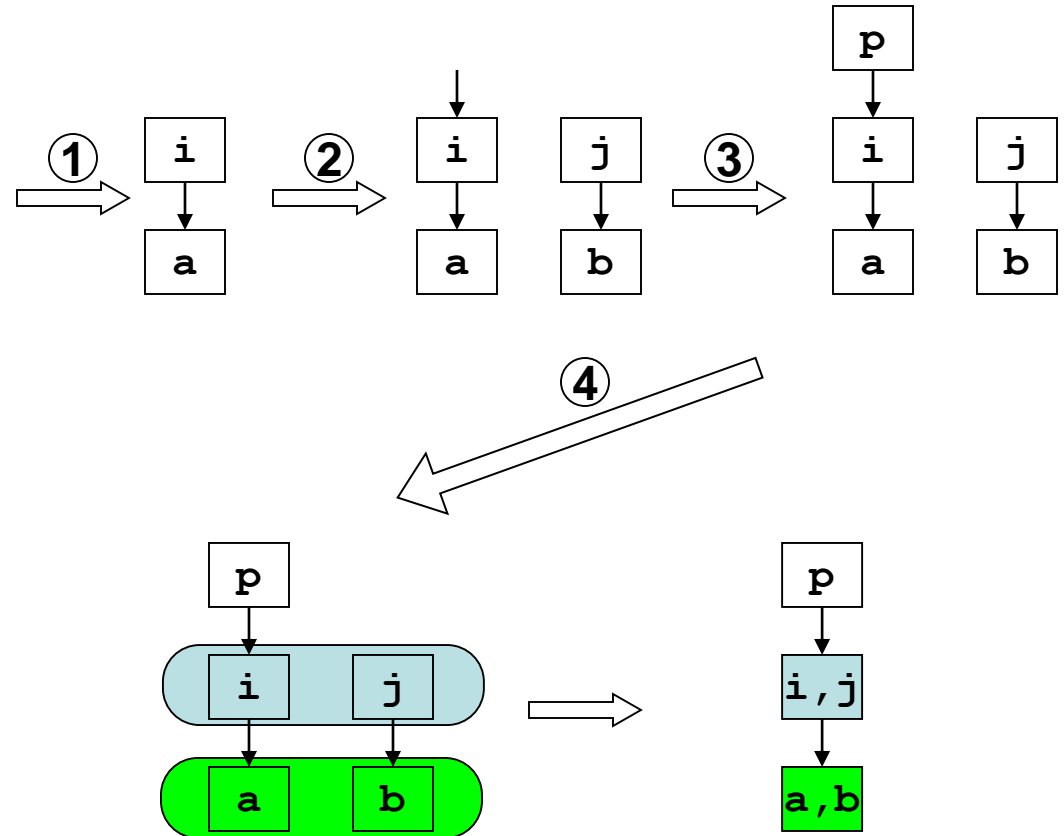


```
void foo(int* p) {  
  printf("%d", *p);  
}
```

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);  
}
```



Almost linear time

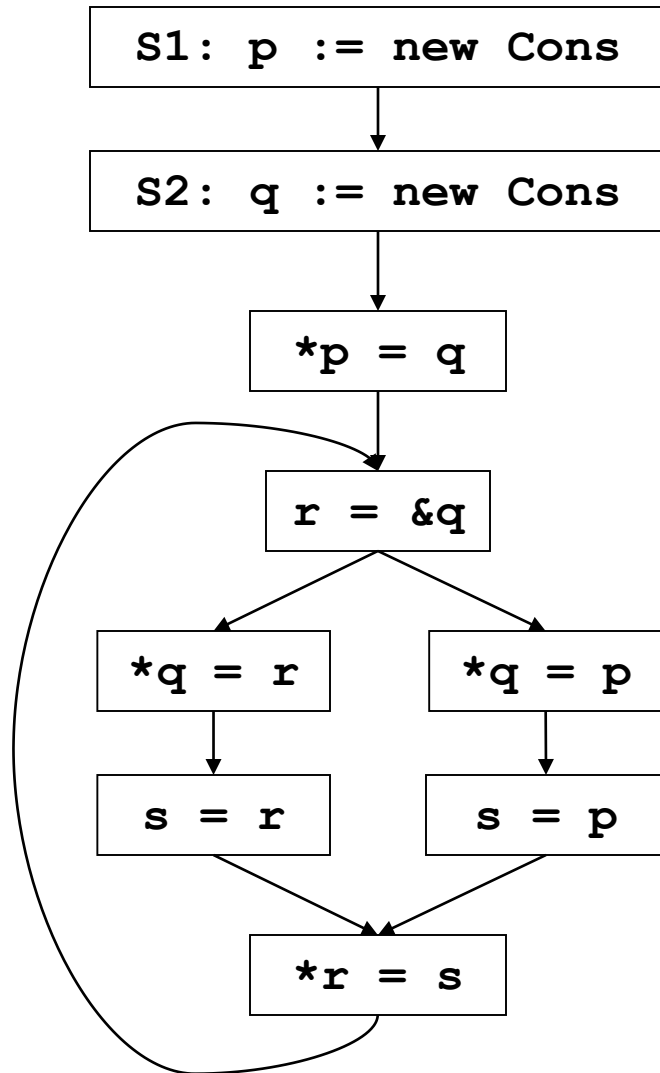
- Time complexity: $O(N\alpha(N, N))$



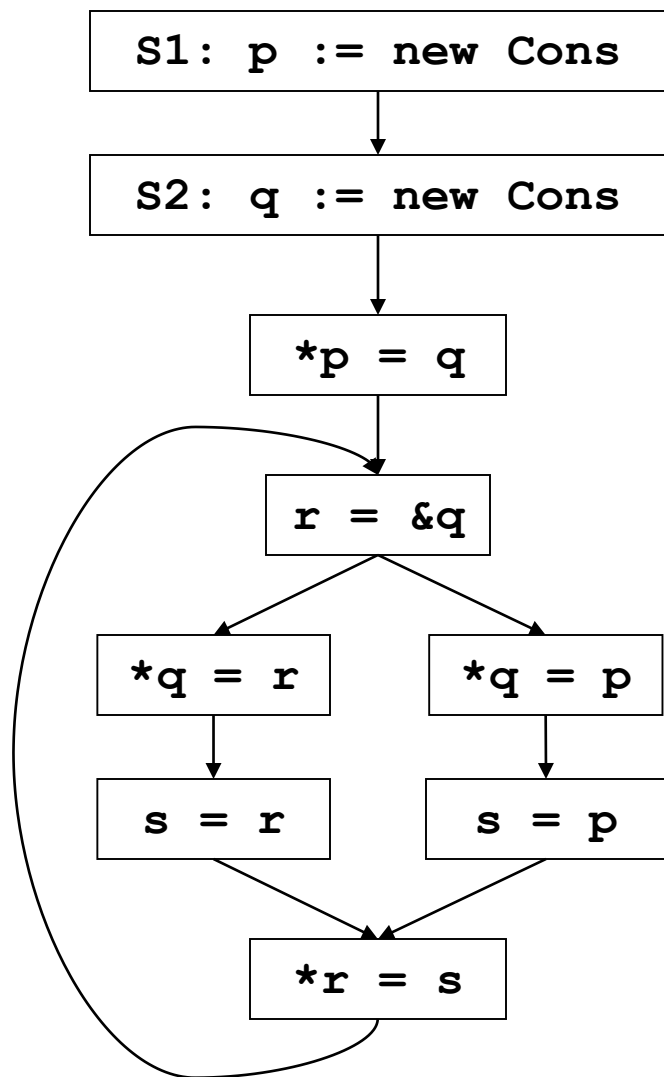
inverse Ackermann
function

- 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 $O(\alpha(N, N))$ per op. Take CSE 202 to learn more!

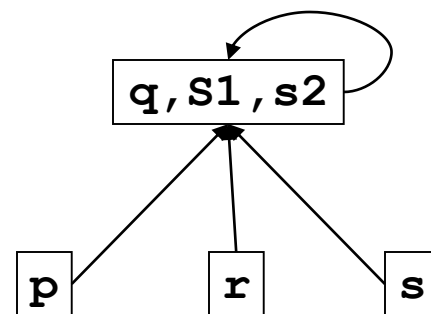
In Class Exercise!



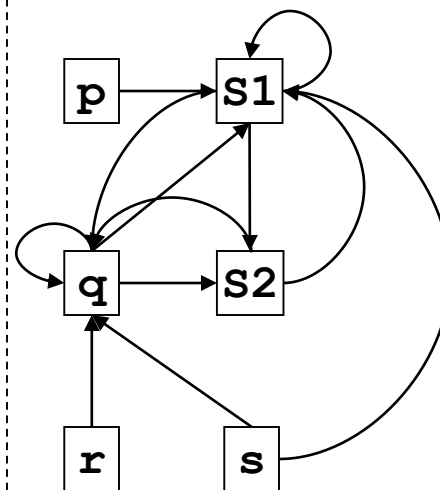
In Class Exercise! solved



Steensgaard



Andersen



Advanced Pointer Analysis

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