What does this function do?

```
bool foo(char *a, char *b){
   *a = ‘a’;
   *b = ‘b’
   return *a == ‘a’;
}
```

Can we optimize it as following?

```
bool foo(char *a, char *b){
   return true;
}
```

Alias Analysis

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Alias Analysis

- Alias: when two different pointers may point to same location in memory

- Alias analysis: When do two variables alias with each other?

- Formally:
  - At a program point, do any pair of pointer-typed variables, p, q, point to the same memory location?

Why do Alias Analysis

- Eliminate dead stores

```
*x = …
//*x is dead
```

- Avoid redundant load/stores

```
x = *p
…
y = *p
```

- Decide if we can parallelize
- Do LICM (loop invariant code motion) in the presence of stores in the loop body
- Error detection

```
x.lock()
..
y.unlock()
```

Alias Analysis - Challenges

- Undecidable
- May alias (i.e. sometimes)
  - Answer to “does p and q alias with each other” is “yes” or “no” or “maybe”
- Inter-procedural Analysis Required
  - Scalability is an issue

Content

- Alias Analysis
  - Analysis of programs with Pointers (slides courtesy: Prof. Milind Kulkarni and Prof. Keshav Pingali)
    - Flow-sensitive, flow-insensitive algorithms
      - Heap allocation: slides courtesy: Prof. Sorin Lerner, CSE231, UCSD
    - Inter-Procedural Analysis

Analysis of programs with pointers

Simple example

x := 5	S1
ptr := @x	S2
*ptr := 9	S3
y := x	S4
program	dependences

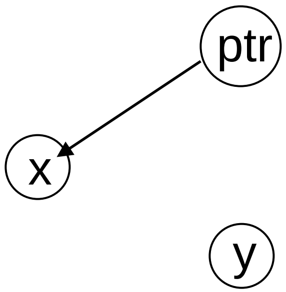
- What are the dependences in this program?
- Problem: just looking at variable names will not give you the correct information
  - After statement S2, program names “x” and “*ptr” are both expressions that refer to the same memory location.
  - We say that ptr points-to x after statement S2.
- In a C-like language that has pointers, we must know the points-to relation to be able to determine dependences correctly

Program model

- For now, only types are int and int*
- No heap
  - All pointers point to only to stack variables
- No procedure or function calls
- Statements involving pointer variables:
  - address: x := &y
  - copy: x := y
  - load: x := *y
  - store: *x := y
- Arbitrary computations involving ints

Points-to relation

- Directed graph:
  - nodes are program variables
  - edge (a,b): variable a points-to variable b

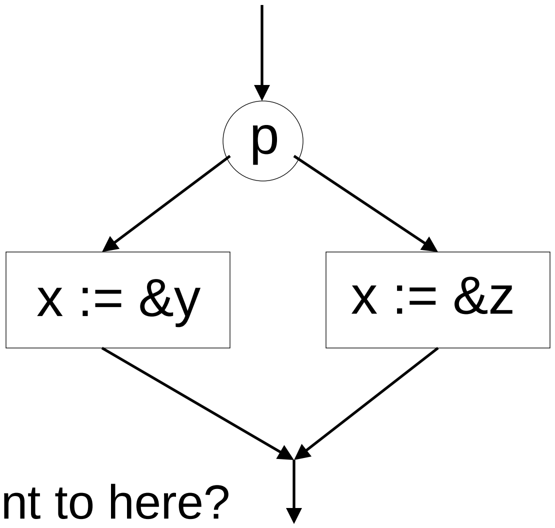


- Can use a special node to represent NULL
- Points-to relation is different at different program points

Points-to graph

- Out-degree of node may be more than one
  - if points-to graph has edges (a,b) and (a,c), it means that variable a may point to either b or c
  - depending on how we got to that point, one or the other will be true
  - path-sensitive analyses: track how you got to a program point (we will not do this)

```
if (p)
    then x := &y
    else x := &z
…..
```



What does x point to here?

Ordering on points-to relation

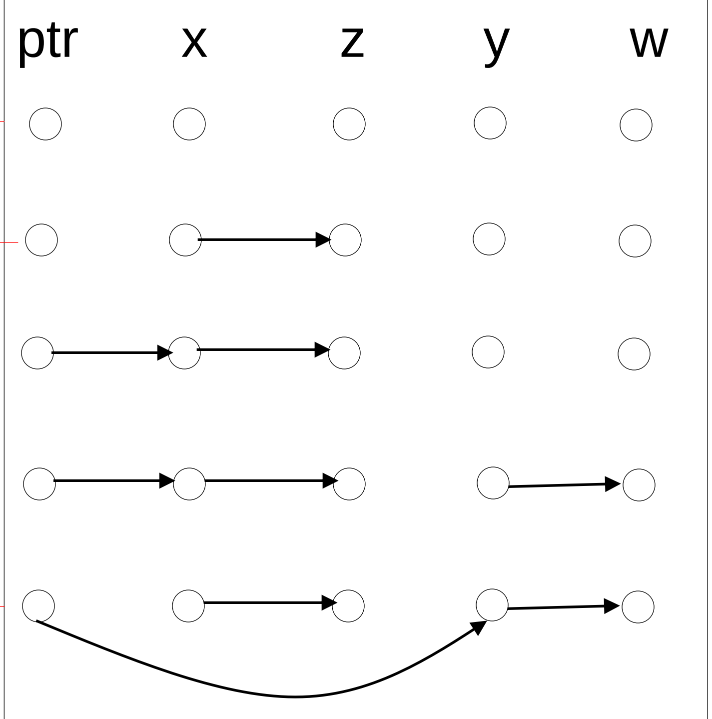
- Subset ordering: for a given set of variables
  - Least element is graph with no edges
  - G1 <= G2 if G2 has all the edges G1 has and maybe some more
- Given two points-to relations G1 and G2
  - G1 U G2: least graph that contains all the edges in G1 and in G2

Overview

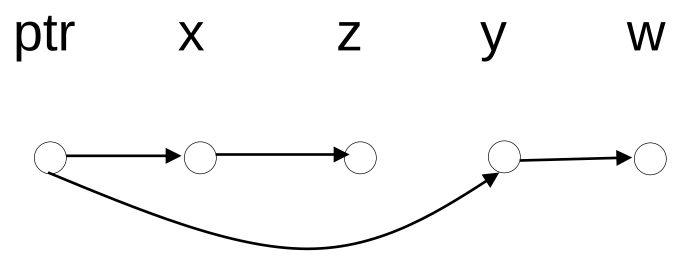
- We will look at three different points-to analyses.
- Flow-sensitive points-to analysis
  - Dataflow analysis
  - Computes a different points-to relation at each point in program
- Flow-insensitive points-to analysis
  - Computes a single points-to graph for entire program
  - Andersen's algorithm
    - Natural simplification of flow-sensitive algorithm
  - Steensgard's algorithm
    - Nodes in tree are equivalence classes of variables
      - if x may point-to either y or z, put y and z in the same equivalence class
    - Points-to relation is a tree with edges from children to parents rather than a general graph
    - Less precise than Andersen's algorithm but faster

Example

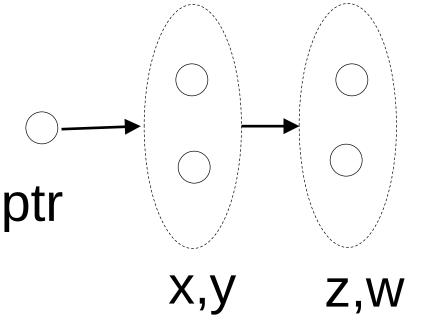
```
x := &z

ptr := @x

y := @w

ptr := @y
```



Flow-sensitive algorithm



Andersen's algorithm



Steensgard's algorithm

Notation

- Suppose S and S1 are set-valued variables.
- S ← S1: strong update
  - set assignment
- S U← S1: weak update
  - set union: this is like S ← S U S1

Flow-sensitive algorithm

Dataflow equations

- Forward flow, any path analysis
- Confluence operator: G1 U G2
- Statements

G

x := &y

G’ = G with pt’(x) $\leftarrow$ {y}

G

x := *y

G’ = G with pt’(x) $\leftarrow$ U pt(a)
for all a in pt(y)

G

x := y

G’ = G with pt’(x) $\leftarrow$ pt(y)

G

*x := y

G’ = G with pt’(a) U$\leftarrow$ pt(y)
for all a in pt(x)

Dataflow equations (contd.)

G

x := &y

G' = G with pt'(x) ← {y}

G

x := *y

G' = G with pt'(x) ← U pt(a)
for all a in pt(y)

G

x := y

G' = G with pt'(x) ← pt(y)

G

*x := y

G' = G with pt'(a) U← pt(y)
for all a in pt(x)

strong updates

weak update (why?)

Strong vs. weak updates

- Strong update:
  - At assignment statement, you know precisely which variable is being written to
  - Example: x := ….
  - You can remove points-to information about x coming into the statement in the dataflow analysis.
- Weak update:
  - You do not know precisely which variable is being updated; only that it is one among some set of variables.
  - Example: *x := …
  - Problem: at analysis time, you may not know which variable x points to (see slide on control-flow and out-degree of nodes)
  - Refinement: if out-degree of x in points-to graph is 1 and x is known not be nil, we can do a strong update even for *x := …

Try it yourself

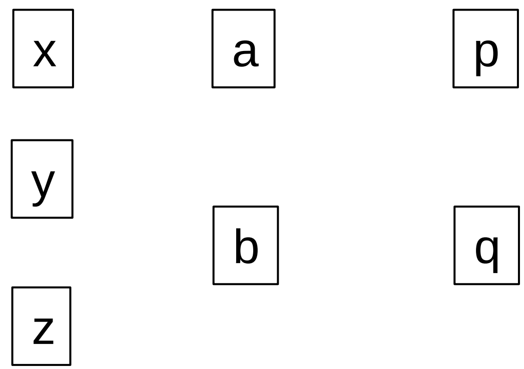
Show the points-to graph after line 6 after running a flow-sensitive analysis on the following code.

```
1: x = &a;
2: y = x;
3: x = &b;
4: a = &p;
5: b = &q;
6: z = *y;
```

Try it yourself - solution

before:

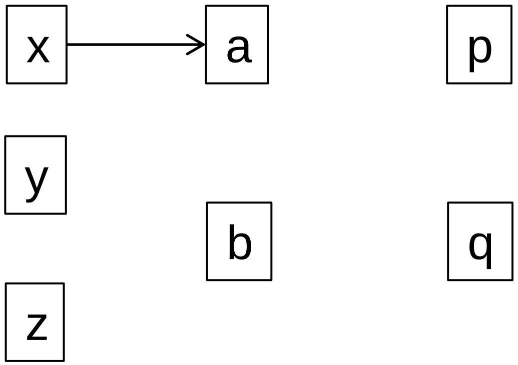
```
1: x = &a;
```



Try it yourself- solution

after:

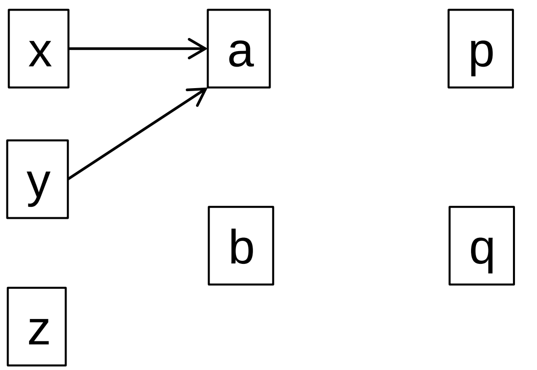
```
1: x = &a;
```



Try it yourself- solution

after:

```
1: x = &a;
2: y = x;
```

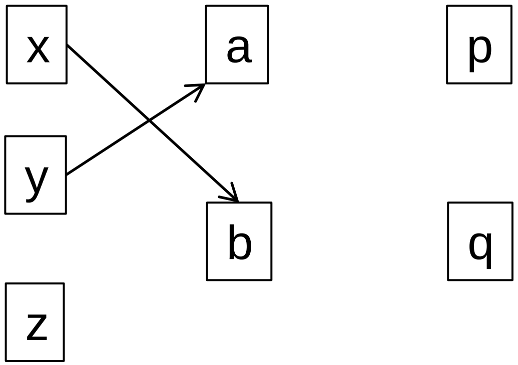


Try it yourself- solution

after:

```
1: x = &a;
2: y = x;
3: x = &b;
```

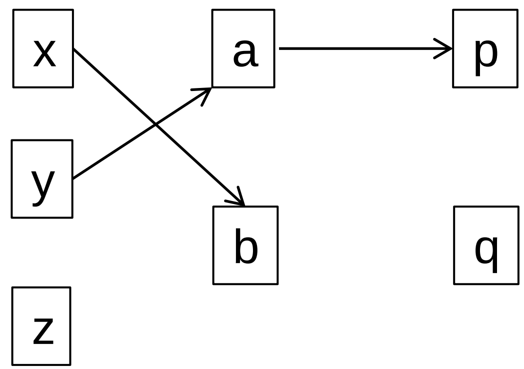
Note the strong update – x now points to b instead of a



Try it yourself- solution

after:

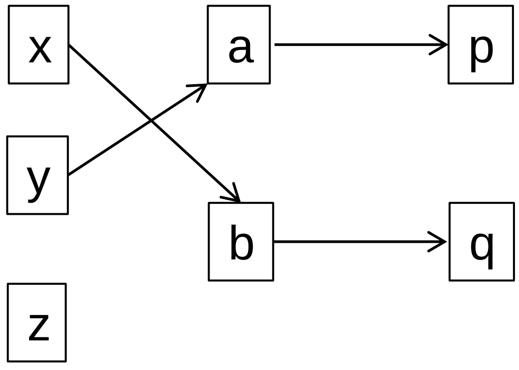
```
1:  x = &a;
2:  y = x;
3:  x = &b;
4:  a = &p;
```



Try it yourself- solution

after:

```
1: x = &a;
2: y = x;
3: x = &b;
4: a = &p;
5: b = &q;
```

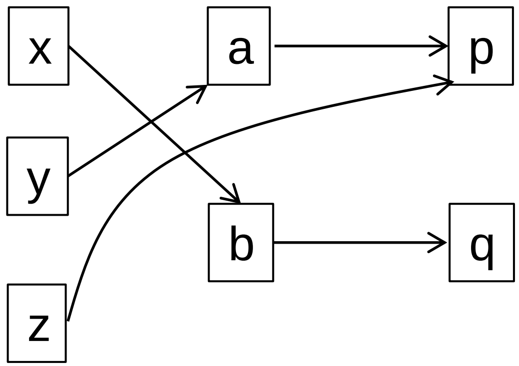


Try it yourself- solution

after:

```
1: x = &a;
2: y = x;
3: x = &b;
4: a = &p;
5: b = &q;
6: z = *y;
```

Why? See dataflow equations on slide 11



Flow-insensitive algorithms

Flow-insensitive analysis

- Flow-sensitive analysis computes a different graph at each program point.
- This can be quite expensive.
- One alternative: flow-insensitive analysis
  - Intuition:compute a points-to relation which is the least upper bound of all the points-to relations computed by the flow-sensitive analysis
- Approach:
  - Ignore control-flow
  - Consider all assignment statements together
    - replace strong updates in dataflow equations with weak updates
  - Compute a single points-to relation that holds regardless of the order in which assignment statements are actually executed

Andersen's algorithm

- Statements

weak updates only

G

x := &y

G = G with pt(x) U← {y}

G

x := *y

G = G with pt(x) U← pt(a)
for all a in pt(y)

G

x := y

G = G with pt(x) U← pt(y)

G

*x := y

G = G with pt(a) U← pt(y)
for all a in pt(x)

Andersen's algorithm formulated using set constraints

- Statements

$$pt : \text{var} \rightarrow 2^{\text{var}}$$

x := &y

$$y \in pt(x)$$

x := *y

$$\forall a \in pt(y).pt(x) \supseteq pt(a)$$

x := y

$$pt(x) \supseteq pt(y)$$

*x := y

$$\forall a \in pt(x).pt(a) \supseteq pt(y)$$

Steensgard's algorithm

- Flow-insensitive
- Computes a points-to graph in which there is no fan-out
  - In points-to graph produced by Andersen's algorithm, if x points-to y and z, y and z are collapsed into an equivalence class
  - Less accurate than Andersen's but faster
- We can exploit this to design an $O(N*\alpha(N))$ algorithm, where N is the number of statements in the program.

Steensgard's algorithm using set constraints

- Statements

$$pt : \text{var} \rightarrow 2^{\text{var}}$$

No fan-out $\forall x. \forall y, z \in pt(x). pt(y) = pt(z)$

$\boxed{\text{x := \&y}}$

$$y \in pt(x)$$

$\boxed{\text{x := *y}}$

$$\forall a \in pt(y). pt(x) = pt(a)$$

$\boxed{\text{x := y}}$

$$pt(x) = pt(y)$$

$\boxed{\text{*x := y}}$

$$\forall a \in pt(x). pt(a) = pt(y)$$

Try it yourself

Show the points-to graph after line 6 after running a flow-insensitive analysis on the following code.

```
1: x = &a;
2: y = x;
3: x = &b;
4: a = &p;
5: b = &q;
6: z = *y;
```

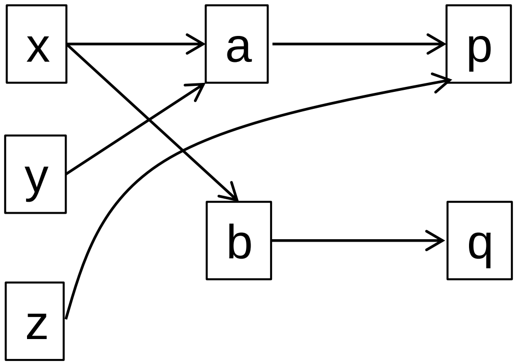
Try it yourself - solution

We do not do strong updates. Processing a statement can only add new edges. We do not remove edges. But we have to process statements repeatedly until the graph converges.

after:

```
1: x = &a;
2: y = x;
3: x = &b;
4: a = &p;
5: b = &q;
6: z = *y;
```

Note x points to both a and b because of weak updat
Do we need to reprocess statements?



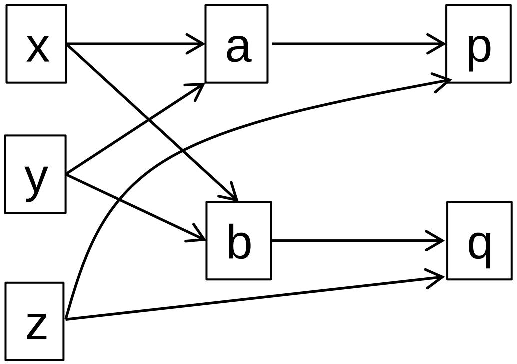
Try it yourself - solution

We do not do strong updates. Processing a statement can only add new edges. We do not remove edges. But we have to process statements repeatedly until the graph converges.

after:

```
1: x = &a;
2: y = x;
3: x = &b;
4: a = &p;
5: b = &q;
6: z = *y;
```

Reprocess all statements.



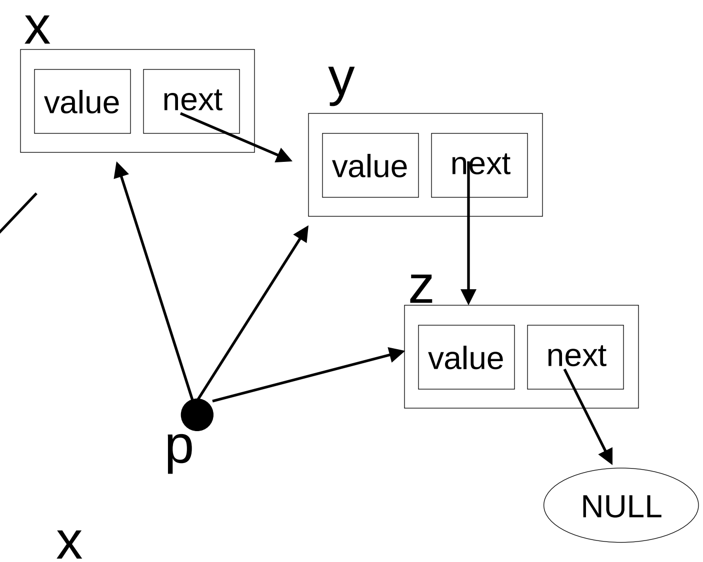
Note y now points to both a and b and also z points to both p and q

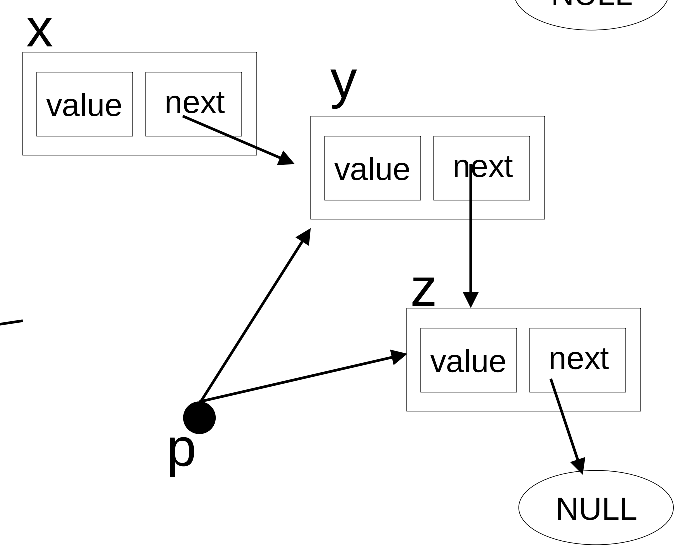
Structures

- Structure types
  - struct cell {int value; struct cell *left, *right;}
  - struct cell x,y;
- Use a "field-sensitive" model
  - x and y are nodes
  - each node has three internal fields labeled value, left, right
- This representation permits pointers into fields of structures
  - If this is not necessary, we can simply have a node for each structure and label outgoing edges with field name

Example

```
int main(void)
        { struct cell {int value;
                       struct cell *next;
                       };
          struct cell x,y,z,*p;
          int sum;

          x.value = 5;
          x.next = &y;
          y.value = 6;
          y.next = &z;
          z.value = 7;
          z.next = NULL;

          p = &x;
          sum = 0;
          while (p != NULL) {
                       sum = sum + (*p).value;
                       p = (*p).next;
                   }
          return sum;
        }
```





Example

```
int main(void)
        { struct cell {int value;
                        struct cell *next;
                        };
          struct cell x,y,z,*p;
          int sum;

          x.value = 5;
          x.next = &y;
          y.value = 6;
          y.next = &z;
          z.value = 7;
          z.next = NULL;

          p = &x;
          sum = 0;
          while (p != NULL) {
                        sum = sum + (*p).value;
                        p = (*p).next;
                    }
          return sum;
        }
```

x.next = &y;

y.next = &z;

z.next = NULL;

p = &x;

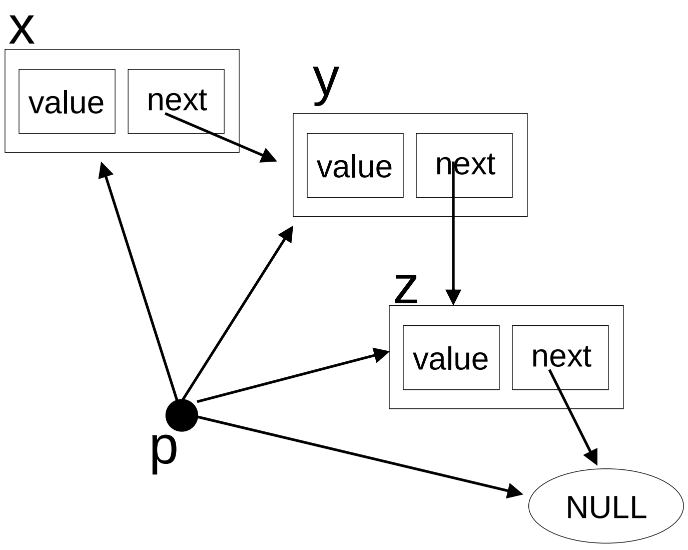
p = (*p).next;

.
.
.

G

Assignments for flow-insensitive analysis

Solution to flow-insensitive equations



- Compare with points-to graphs for flow-sensitive solution
- Why does p point-to NULL in this graph?

Inter-procedural analysis

- What do we do if there are function calls?

```
x1 = &a
y1 = &b
swap(x1, y1)
```

```
x2 = &a
y2 = &b
swap(x2, y2)
```

```
swap (p1, p2) {
      t1 = *p1;
      t2 = *p2;
      *p1 = t2;
      *p2 = t1;
}
```

Two approaches

- Context-sensitive approach:
  - treat each function call separately just like real program execution would
  - problem: what do we do for recursive functions?
    - need to approximate
- Context-insensitive approach:
  - merge information from all call sites of a particular function
  - in effect, inter-procedural analysis problem is reduced to intra-procedural analysis problem
- Context-sensitive approach is obviously more accurate but also more expensive to compute

Context-insensitive approach

```
x1 = &a
y1 = &b
swap(x1, y1)
```

```
x2 = &a
y2 = &b
swap(x2, y2)
```

```
swap (p1, p2) {
        t1 = *p1;
        t2 = *p2;
        *p1 = t2;
        *p2 = t1;
}
```

Context-sensitive approach

```
x1 = &a
y1 = &b
swap(x1, y1)
```

```
swap (p1, p2) {
        t1 = *p1;
        t2 = *p2;
        *p1 = t2;
        *p2 = t1;
}
```

```
x2 = &a
y2 = &b
swap(x2, y2)
```

```
swap (p1, p2) {
        t1 = *p1;
        t2 = *p2;
        *p1 = t2;
        *p2 = t1;
}
```

Context-insensitive/Flow-insensitive Analysis

- For now, assume we do not have function parameters
  - this means we know all the call sites for a given function
- Set up equations for binding of actual and formal parameters at each call site for that function
  - use same variables for formal parameters for all call sites
- Intuition: each invocation provides a new set of constraints to formal parameters

Swap example

```
x1 = &a
y1 = &b
p1 = x1
p2 = y1
```

```
x2 = &a
y2 = &b
p1 = x2
p2 = y2
```

```
t1 = *p1;
t2 = *p2;
*p1 = t2;
*p2 = t1;
```

Heap allocation

- Simplest solution:
  - use one node in points-to graph to represent all heap cells
- More elaborate solution:
  - use a different node for each malloc site in the program
- Even more elaborate solution: shape analysis
  - goal: summarize potentially infinite data structures
  - but keep around enough information so we can disambiguate pointers from stack into the heap, if possible

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:

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

- Flow functions:

Where,

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

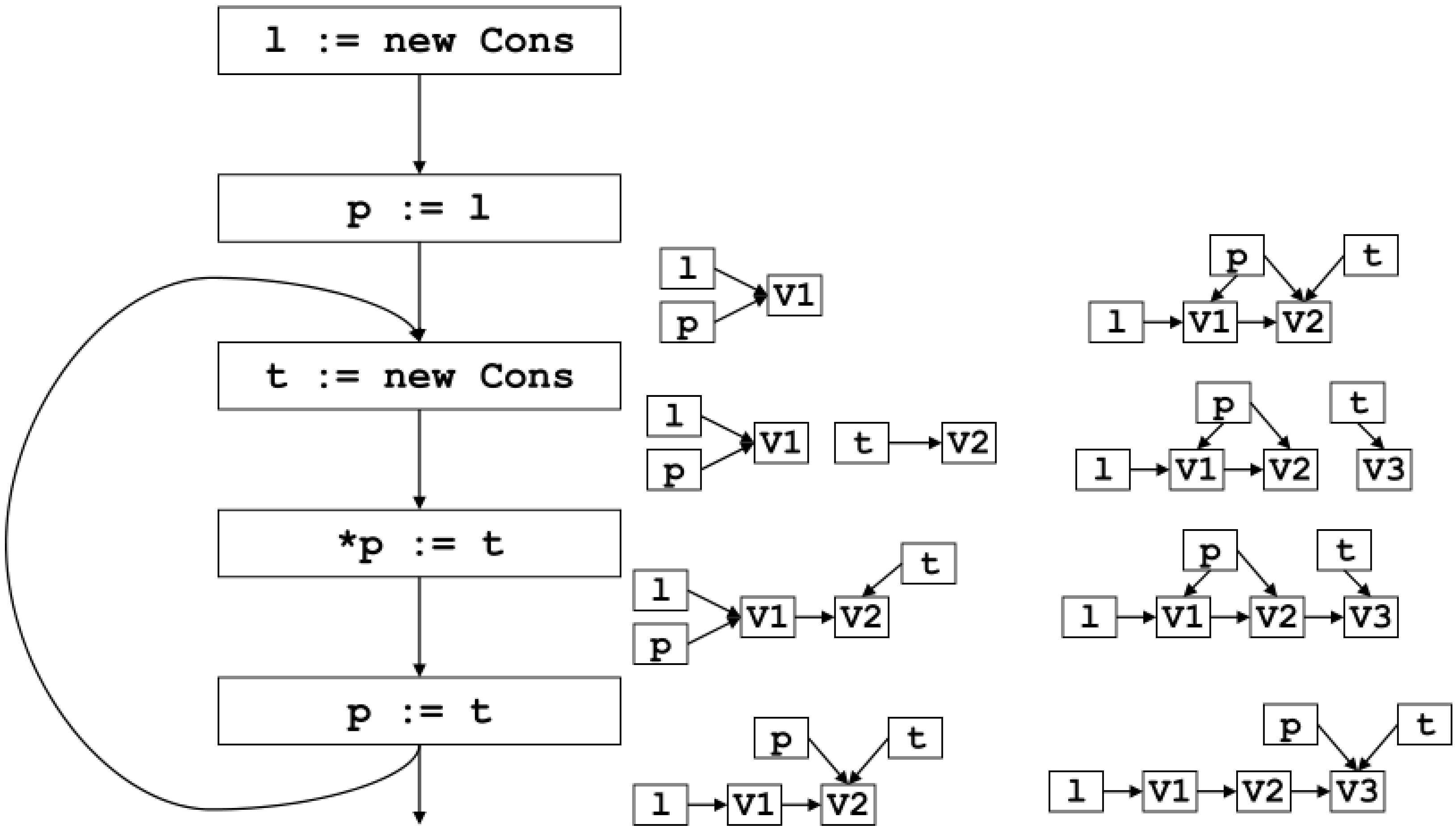
Pointers to dynamically-allocated memory

- 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:

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

Flow-sensitive analysis

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.

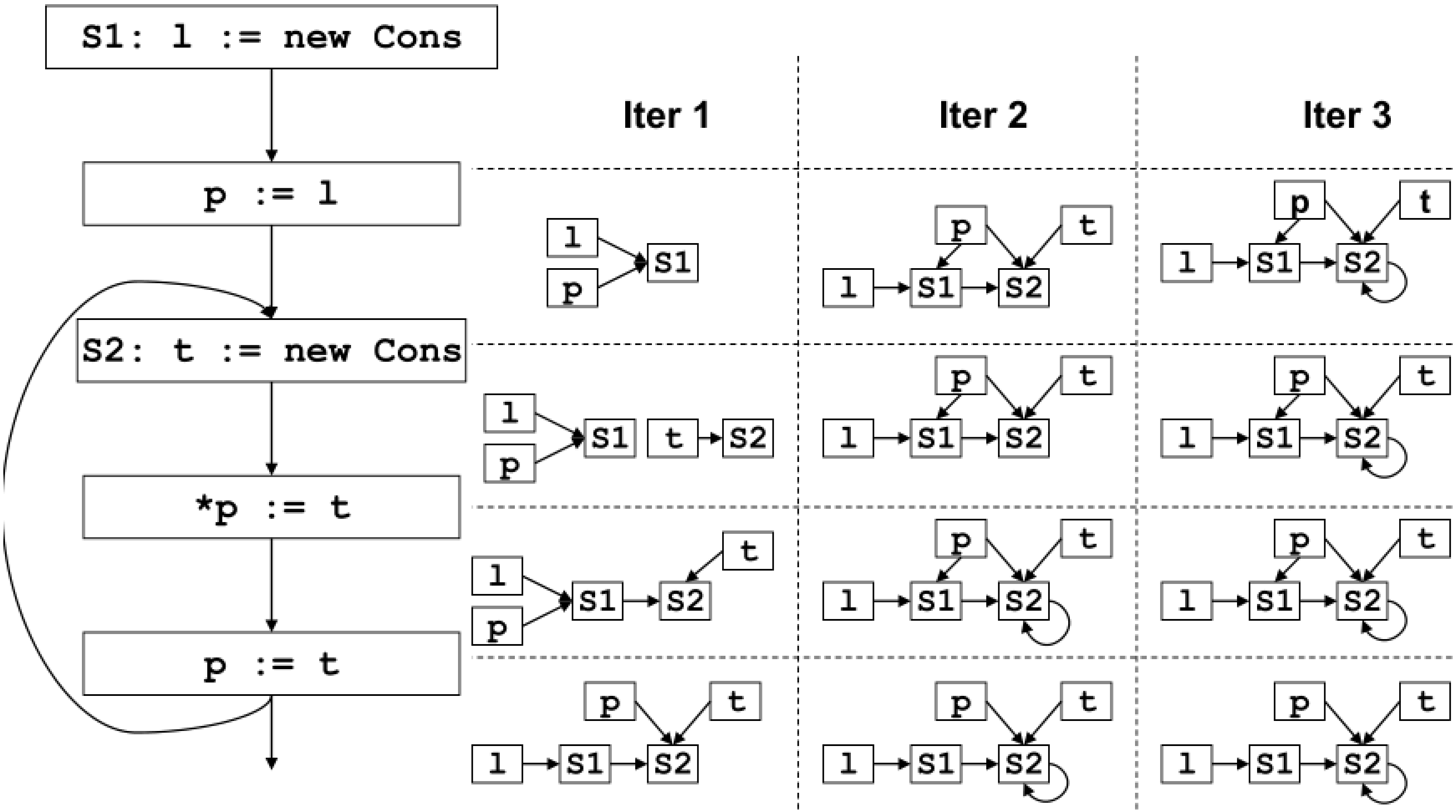
What went wrong?

- 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 & solved



Array aliasing, and pointers to arrays

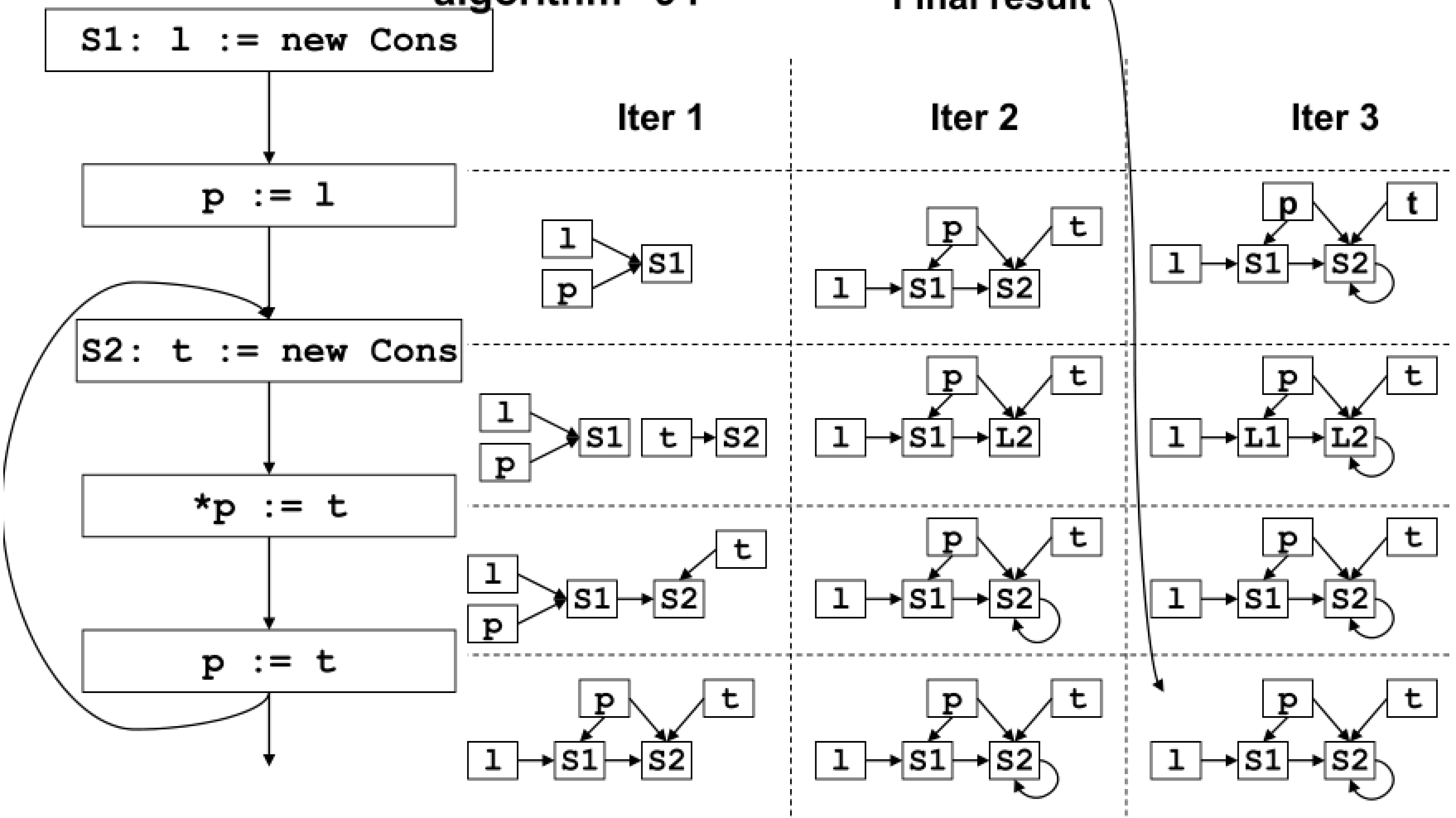
- Array indexing can cause aliasing:
  - **`a[i]`** aliases **`b[j]`** if:
    - a aliases b and i = j
    - a and b overlap, and i = j + k, where k is the amount of overlap.
- Can have pointers to elements of an array
  - **`p := &a[i]; ...; p++;`**
- How can arrays be modeled?
  - Could treat the whole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

Flow-insensitive analysis

Flow insensitive pointer analysis: fixed

This is Andersen's algorithm '94

Final result

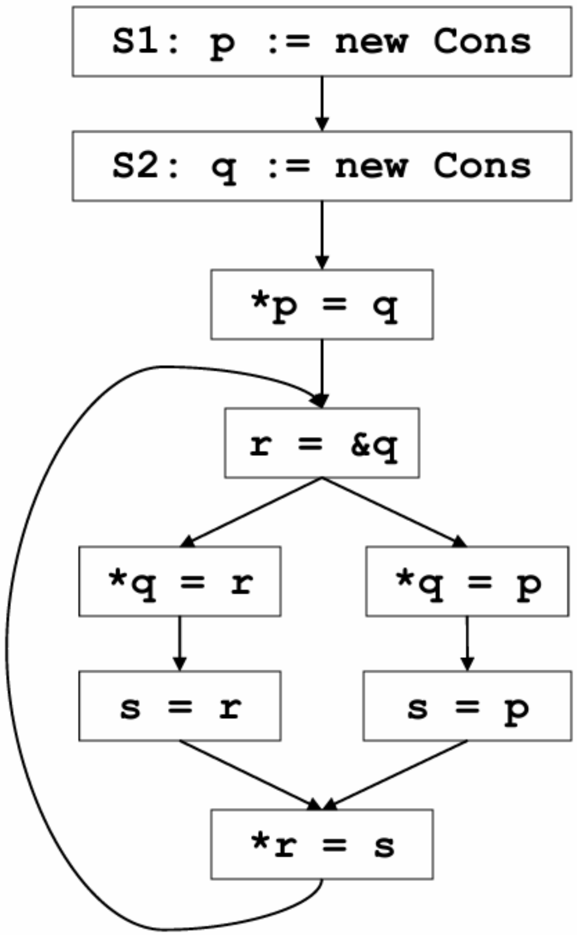


Generalization (Dataflow Analysis)

- **Direction of the analysis?**
  - How does information flow w.r.t. control flow?
- **Join operator?**
  - What happens at merge points? E.g. what operator to use Union or Intersection?
- **Transfer function?**
  - Define sets gen(b), kill(b), IN(b), OUT(b)
- **Initializations?**

What is Lattice (Domain, Join/Meet, LUB/GLB, Top and Bottom) values for Intra-procedural Points-to analysis?

Try it yourself



Show the points to graph after *flow-insensitive* analysis on the shown cfg.