Static Program Analysis

Part 9 – pointer analysis

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University

Agenda

- **Introduction to points-to analysis**
- Andersen’s analysis
- Steensgaards’s analysis
- Interprocedural points-to analysis
- Null pointer analysis
- Flow-sensitive points-to analysis

Analyzing programs with pointers

How do we perform e.g. constant propagation analysis when the programming language has pointers?
(or object references?)

```
...
*x = 42;
*y = -87;
z = *x;
// is z 42 or -87?
```

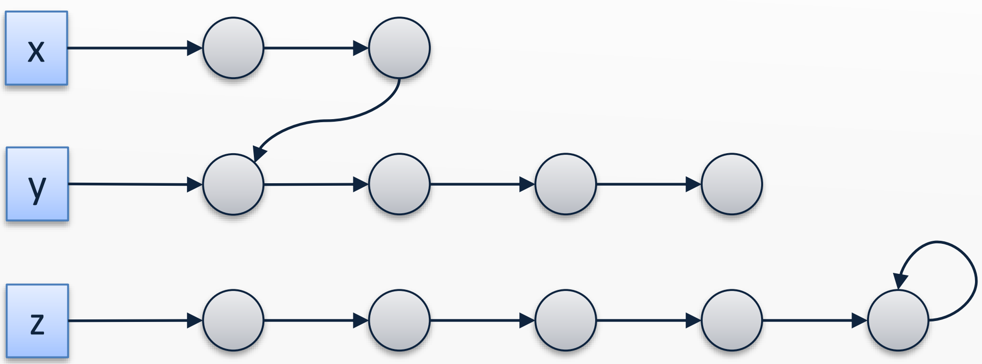
$E \rightarrow$ &X
  | `alloc` E
  | *E
  | `null`
  | …

$S \rightarrow$ *X = E;
  | …

$E \rightarrow E(\ E, \ldots, E\ )$

Heap pointers

- For simplicity, we ignore records
  - `alloc` then only allocates a single cell
  - only linear structures can be built in the heap



- Let's at first also ignore function pointers
- We still have many interesting analysis challenges...

Pointer targets

- The fundamental question about pointers:
  What locations can they point to?

- We need a suitable abstraction
- The set of (abstract) cells, *Cells*, contains
  - `alloc-`*i* for each allocation site with index *i*
  - *X* for each program variable named *X*
- This is called ***allocation site abstraction***
- The set of all (abstract) locations, *Locs*, contains &*p* for every cell *p*
- Each abstract cell may correspond to many concrete memory cells at runtime

Points-to analysis

- Determine for each pointer variable *X* the set *pt*(*x*) of the cells *X* may point to

```
...
*x = 42;
*y = -87;
z = *x;
// is z 42 or -87?
```

- A *conservative* ("may points-to") analysis:
  - the set may be too large
  - can show absence of aliasing: $pt(X) \cap pt(Y) = \emptyset$
- We'll focus on *flow-insensitive* analyses:
  - take place on the AST
  - before or together with the control-flow analysis

Obtaining points-to information

- An almost-trivial analysis (called *address-taken*):
  - include all `alloc-`*i* cells
  - Include the *X* cell if the expression &*X* occurs in the program
- Improvement for a typed language:
  - eliminate those cells whose types do not match
- This is sometimes good enough
  - and clearly very fast to compute

Pointer normalization

- Assume that all pointer usage is normalized:
  - $X = \texttt{alloc}\ \ P$ where P is `null` or an integer constant
  - $X = \&Y$
  - $X = Y$
  - $X = {*Y}$
  - $*X = Y$
  - $X = \texttt{null}$
- Simply introduce lots of temporary variables…
- All sub-expressions are now named
- We choose to ignore the fact that the cells created at variable declarations and `alloc`s are uninitialized

Agenda

- Introduction to points-to analysis
- **Andersen’s analysis**
- Steensgaards’s analysis
- Interprocedural points-to analysis
- Null pointer analysis
- Flow-sensitive points-to analysis

Andersen's analysis (1/2)

- For every cell c, introduce a constraint variable $[\![c]\!]$ ranging over sets of locations, i.e. $[\![\cdot]\!]: \mathit{Cells} \rightarrow 2^{\mathit{Locs}}$

- Generate constraints:
  - $X =$ `alloc` P: `&alloc-`$i \in [\![X]\!]$
  - $X = \&Y$: $\&Y \in [\![X]\!]$
  - $X = Y$: $[\![Y]\!] \subseteq [\![X]\!]$
  - $X = {*}Y$: $\&\alpha \in [\![Y]\!] \Rightarrow [\![\alpha]\!] \subseteq [\![X]\!]$
  - ${*}X = Y$: $\&\alpha \in [\![X]\!] \Rightarrow [\![Y]\!] \subseteq [\![\alpha]\!]$
  - $X =$ `null`:   (no constraints)

Andersen's analysis (2/2)

- The points-to map is defined as:

  $$pt(X) = \{\ \alpha \in Cells \mid \&\alpha \in [\![X]\!]\ \}$$

- The constraints fit into the cubic framework ☺
- Unique minimal solution in time $O(n^3)$
- In practice, for Java: $O(n^2)$

- The analysis is flow-insensitive but *directional*
  – models the direction of the flow of values in assignments

Example program

```
var p,q,x,y,z;
p = alloc null;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

Applying Andersen

- Generated constraints:

$\&\mathtt{alloc\text{-}1} \in [\![p]\!]$
$[\![y]\!] \subseteq [\![x]\!]$
$[\![z]\!] \subseteq [\![x]\!]$
$\&\alpha \in [\![p]\!] \Rightarrow [\![z]\!] \subseteq [\![\alpha]\!]$
$[\![q]\!] \subseteq [\![p]\!]$
$\&y \in [\![q]\!]$
$\&\alpha \in [\![p]\!] \Rightarrow [\![\alpha]\!] \subseteq [\![x]\!]$
$\&z \in [\![p]\!]$

- Smallest solution:

$pt(\mathrm{p}) = \{\ \mathtt{alloc\text{-}1}, \mathrm{y}, \mathrm{z}\ \}$
$pt(\mathrm{q}) = \{\ \mathrm{y}\ \}$

Agenda

- Introduction to points-to analysis
- Andersen’s analysis
- **Steensgaards’s analysis**
- Interprocedural points-to analysis
- Null pointer analysis
- Flow-sensitive points-to analysis

Steensgaard's analysis

- View assignments as being bidirectional
- Generate constraints:
  - $X = \texttt{alloc}\ P$: $[\![X]\!] = \&[\![\texttt{alloc-}i]\!]$
  - $X = \&Y$: $[\![X]\!] = \&[\![Y]\!]$
  - $X = Y$: $[\![X]\!] = [\![Y]\!]$
  - $X = {}^*Y$: $[\![Y]\!] = \&\alpha \land [\![X]\!] = \alpha$ where α is fresh
  - ${}^*X = Y$: $[\![X]\!] = \&\alpha \land [\![Y]\!] = \alpha$ where α is fresh
- Terms:
  - term variables, e.g. $[\![X]\!]$, $[\![\texttt{alloc-}i]\!]$, α (each representing the possible values of a cell)
  - a single (unary) term constructor $\&t$ (representing the location of the cell that t represents)
  - $[\![X]\!]$ is now a term variable, not a constraint variable holding a set of locations
- Fits with our unification solver! (union-find…)
- The points-to map is defined as $\text{pt}(X) = \{ c \in \mathit{Cells} \mid [\![X]\!] = \&[\![c]\!] \}$
- Note that there is only one kind of term constructor, so unification never fails

Applying Steensgaard

- Generated constraints:

$\&\texttt{alloc-1} \in [\![p]\!]$
$[\![y]\!] = [\![x]\!]$
$[\![z]\!] = [\![x]\!]$
$\&\alpha \in [\![p]\!] \Rightarrow [\![z]\!] = [\![\alpha]\!]$
$[\![q]\!] = [\![p]\!]$
$\&y \in [\![q]\!]$
$\&\alpha \in [\![p]\!] \Rightarrow [\![\alpha]\!] = [\![x]\!]$
$\&z \in [\![p]\!]$
+ the extra constraints

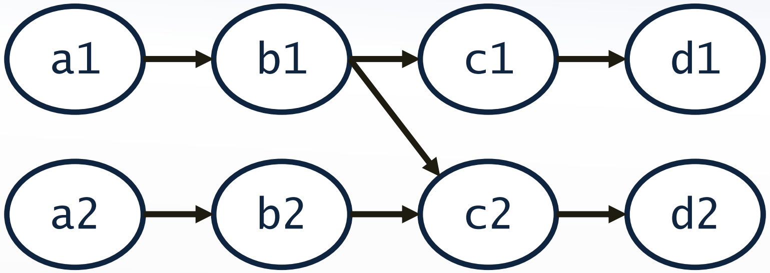
- Smallest solution:

$pt(p) = \{ \texttt{alloc-1}, y, z \}$
$pt(q) = \{ \texttt{alloc-1}, y, z \}$

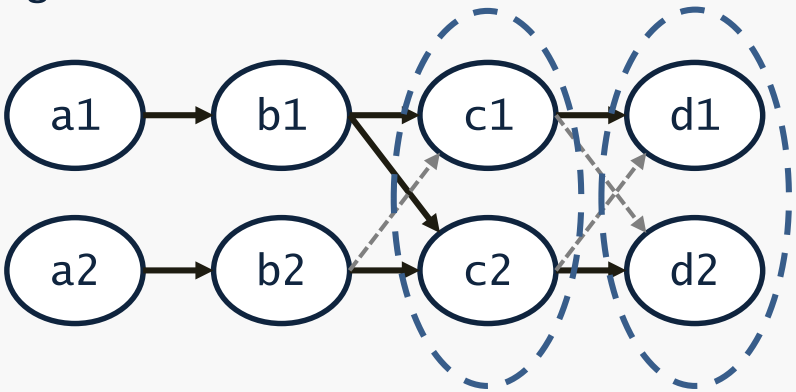
Another example

```
a1 = &b1;
b1 = &c1;
c1 = &d1;
a2 = &b2;
b2 = &c2;
c2 = &d2;
b1 = &c2;
```

Andersen:



Steensgaard:



Recall our type analysis…

- Focusing on pointers…
- Constraints:
  - $X = \texttt{alloc}\ P$: $[\![X]\!] = \&[\![P]\!]$
  - $X = \&Y$: $[\![X]\!] = \&[\![Y]\!]$
  - $X = Y$: $[\![X]\!] = [\![Y]\!]$
  - $X = {*}Y$: $\&[\![X]\!] = [\![Y]\!]$
  - ${*}X = Y$: $[\![X]\!] = \&[\![Y]\!]$
- Implicit extra constraint for term equality:

$$\&t_1 = \&t_2 \Rightarrow t_1 = t_2$$

- Assuming the program type checks, is the solution for pointers the same as for Steensgaard's analysis?

Agenda

- Introduction to points-to analysis
- Andersen’s analysis
- Steensgaards’s analysis
- **Interprocedural points-to analysis**
- Null pointer analysis
- Flow-sensitive points-to analysis

Interprocedural points-to analysis

- If function pointers are distinct from heap pointers:
  - first run a CFA
  - then run Andersen or Steensgaard

- But in TIP both kinds may be mixed together:

```
(***x)(1,2,3)
```

- In this case the CFA and the points-to analysis must happen *simultaneously*!

Function call normalization

- Assume that all function calls are of the form

  $x = y(a_1 , \ldots , a_n)$

- *y* may be a variable whose value is a function pointer
- Assume that all return statements are of the form

  `return` z`;`

- As usual, simply introduce lots of temporary variables…

- Include all function names in *Locs*

CFA with Andersen

- For the function call

  $$x = y(a_1, \ldots, a_n)$$

  and every occurrence of

  $$f(x_1, \ldots, x_n)\ \{ \ldots \texttt{return}\ z; \}$$

  add these constraints:

  $$f \in [\![f]\!]$$

  $$f \in [\![y]\!] \Rightarrow ([\![a_i]\!] \subseteq [\![x_i]\!] \text{ for i=1,...,}n \wedge [\![z]\!] \subseteq [\![x]\!])$$

- (Similarly for simple function calls)
- Fits directly into the cubic framework!

Andersen's analysis is already closely connected to control-flow analysis!

CFA with Steensgaard

- For the function call

  $$x = y(a_1, \ldots, a_n)$$

  and every occurrence of

  $$f(x_1, \ldots, x_n)\ \{ \ldots \texttt{return}\ z;\ \}$$

  add these constraints:

  $$f \in [\![f]\!]$$

  $$f \in [\![y]\!] \Rightarrow ([\![a_i]\!] = [\![x_i]\!] \text{ for i=1,...,}n \wedge [\![z]\!] = [\![x]\!])$$

- (Similarly for simple function calls)
- Fits into the unification framework, but requires a generalization of the ordinary union-find solver

Context-sensitive pointer analysis

- Generalize the abstract domain $Cells \to 2^{Locs}$ to
  $Contexts \to Cells \to 2^{Locs}$
  (or equivalently: $Cells \times Contexts \to 2^{Locs}$)
  where *Contexts* is a (finite) set of call contexts
- As usual, many possible choices of *Contexts*
  - recall the call string approach and the functional approach
- Also need to track the set of reachable contexts for each function (like the use of lifted lattices earlier)

- Does this still fit into the cubic solver?

Context-sensitive pointer analysis

```
foo(a) {
  return *a;
}

bar() {
  ...
  x = alloc null; // alloc-1
  y = alloc null; // alloc-2
  *x = alloc null; // alloc-3
  *y = alloc null; // alloc-4
  ...
  q = foo(x);
  w = foo(y);
  ...
}
```

Are q and w aliases?

Context-sensitive pointer analysis

```
mk() {
  return alloc null; // alloc-1
}

baz() {
  var x,y;
  x = mk();
  y = mk();
  ...
}
```

Are x and y aliases?

Context-sensitive pointer analysis

- We can go one step further and introduce *context-sensitive heap* (a.k.a. *heap cloning*)
- Let each abstract cell be a pair of
  - `alloc-`*i* (the `alloc` with index *i*) or *X* (a program variable)
  - a heap context from a (finite) set *HeapContexts*
- This allows abstract cells to be named by the source code allocation site ***and (information from) the current context***
- One choice:
  - set *HeapContexts* = *Contexts*
  - at `alloc`, use the entire current call context as heap context

Agenda

- Introduction to points-to analysis
- Andersen's analysis
- Steensgaards's analysis
- Interprocedural points-to analysis
- **Null pointer analysis**
- Flow-sensitive points-to analysis

Null pointer analysis

- Decide for every dereference `*p`, is `p` different from `null`?
- (Why not just treat null as a special location in an Andersen or Steensgaard-style analysis?)
- Use the monotone framework
  - assuming that a points-to map *pt* has been computed
- Let us consider an intraprocedural analysis

  (i.e. we ignore function calls)

A lattice for null analysis

- Define the simple lattice *Null*:

  ?
  |
  NN

  where NN represents "definitely **n**ot **n**ull"
  and ? represents "maybe null"

- Use for every program point the map lattice:

  Cells $\rightarrow$ *Null*

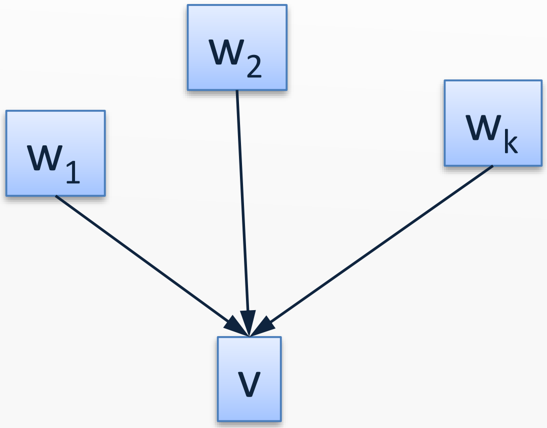
Setting up

- For every CFG node, v, we have a variable ⟦v⟧:
  - a map giving abstract values for all cells at the program point *after* v

- Auxiliary definition:

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

(i.e. we make a *forward* analysis)



Null analysis constraints

- For operations involving pointers:
  - $X = $ `alloc` P: $[\![v]\!] = ???$
  - $X = \&Y$: $[\![v]\!] = ???$
  - $X = Y$: $[\![v]\!] = ???$
  - $X = {*}Y$: $[\![v]\!] = ???$
  - ${*}X = Y$: $[\![v]\!] = ???$
  - $X = $ `null`: $[\![v]\!] = ???$
- For all other CFG nodes:
  - $[\![v]\!] = JOIN(v)$

Null analysis constraints

- For a heap store operation $*X = Y$ we need to model the change of whatever X points to
- That may be *multiple* abstract cells (i.e. the cells $pt(X)$)
- With the present abstraction, each abstract heap cell `alloc-`*i* may describe *multiple* concrete cells
- So we settle for **weak** update:

$$*X = Y: \qquad [\![v]\!] = store(JOIN(v), X, Y)$$

$$\text{where } store(\sigma, X, Y) = \sigma[\alpha \mapsto \sigma(\alpha) \sqcup \sigma(Y)]_{\alpha \in pt(X)}$$

Null analysis constraints

- For a heap load operation $X = *Y$ we need to model the change of the program variable X
- Our abstraction has a *single* abstract cell for X
- That abstract cell represents a *single* concrete cell
- So we can use **strong** update:

$$X = *Y: \qquad [\![v]\!] = load(JOIN(v), X, Y)$$

$$\text{where } \; load(\sigma, X, Y) = \sigma[X \mapsto \bigsqcup_{\alpha \in pt(Y)} \sigma(\alpha)]$$

Strong and weak updates

```
mk() {
  return alloc null; // alloc-1
}


...
a = mk();
b = mk();
*a = alloc null; // alloc-2
n = null;
*b = n; // strong update here would be unsound!
c = *a;
```

is c null here?

The abstract cell `alloc-1` corresponds to *multiple concrete cells*

Strong and weak updates

```
a = alloc null; // alloc-1
b = alloc null; // alloc-2
*a = alloc null; // alloc-3
*b = alloc null; // alloc-4
if (...) {
  x = a;
} else {
  x = b;
}
n = null;
*x = n; // strong update here would be unsound!
c = *x;
```

is c null here?

The points-to set for x contains *multiple abstract cells*

Null analysis constraints

- $X = \texttt{alloc}\ P$: $[\![v]\!] = JOIN(v)[X \mapsto \text{NN}, \texttt{alloc-i} \mapsto ?]$
- $X = \&Y$: $[\![v]\!] = JOIN(v)[X \mapsto \text{NN}]$
- $X = Y$: $[\![v]\!] = JOIN(v)[X \mapsto JOIN(v)(Y)]$
- $X = \texttt{null}$: $[\![v]\!] = JOIN(v)[X \mapsto ?]$

- In each case, the assignment modifies a program variable
- So we can use strong updates, as for heap load operations

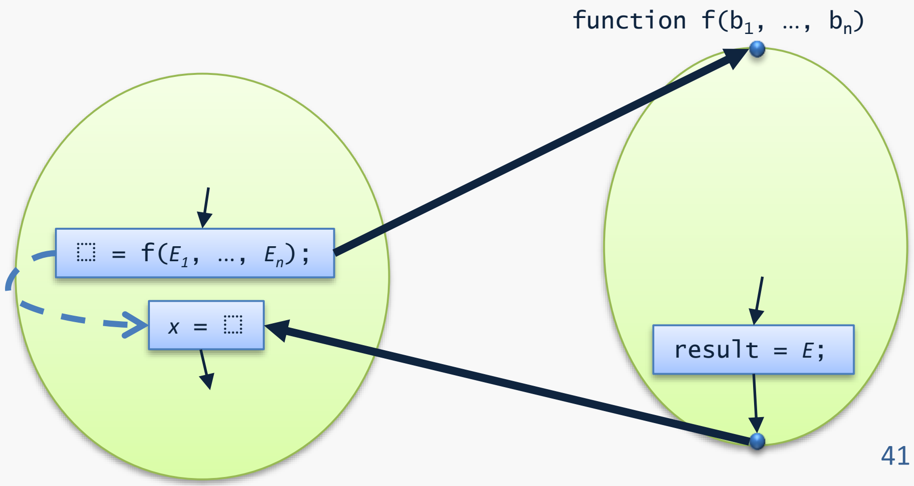
Strong and weak updates, revisited

- Strong update: $\sigma[c \mapsto \textit{new-value}]$
  - possible if *c* is known to refer to a single concrete cell
  - works for assignments to local variables
    (as long as TIP doesn't have e.g. nested functions)

- Weak update: $\sigma[c \mapsto \underline{\sigma(c)} \sqcup \textit{new-value}]$
  - necessary if *c* may refer to multiple concrete cells
  - bad for precision, we lose some of the power of flow-sensitivity
  - required for assignments to heap cells
    (unless we extend the analysis abstraction!)

Interprocedural null analysis

- Context insensitive or context sensitive, as usual…
  - at the after-call node, use the heap from the callee
- But be careful!
  Pointers to local variables may escape to the callee
  - the abstract state at the after-call node cannot simply copy the abstract values for local variables from the abstract state at the call node



Using the null analysis

- The pointer dereference *p is "safe" at entry of v if

  JOIN(v)(p) = NN

- The quality of the null analysis depends on the quality of the underlying points-to analysis

Example program

```
p = alloc null;
q = &p;
n = null;
*q = n;
*p = n;
```

Andersen generates:

$pt(\texttt{p}) = \{\texttt{alloc-1}\}$

$pt(\texttt{q}) = \{\texttt{p}\}$

$pt(\texttt{n}) = \emptyset$

Generated constraints

$[\![$p=alloc null$]\!] = \bot[$p $\mapsto$ NN, alloc-1 $\mapsto$?$]$

$[\![$q=&p$]\!] = [\![$p=alloc null$]\!][$q $\mapsto$ NN$]$

$[\![$n=null$]\!] = [\![$q=&p$]\!][$n $\mapsto$?$]$

$[\![$*q=n$]\!] = [\![$n=null$]\!][$p $\mapsto [\![$n=null$]\!]$(p) $\sqcup$ $[\![$n=null$]\!]$(n)$]$

$[\![$*p=n$]\!] = [\![$*q=n$]\!][$alloc-1 $\mapsto [\![$*q=n$]\!]$(alloc-1) $\sqcup$ $[\![$*q=n$]\!]$(n)$]$

Solution

⟦p=alloc null⟧ = [p ↦ NN, q ↦ NN, n ↦ NN , alloc-1 ↦ ?]
⟦q=&p⟧ = [p ↦ NN, q ↦ NN, n ↦ NN , alloc-1 ↦ ?]
⟦n=null⟧ = [p ↦ NN, q ↦ NN, n ↦ ?, alloc-1 ↦ ?]
⟦*q=n⟧ = [p ↦ ?, q ↦ NN, n ↦ ?, alloc-1 ↦ ?]
⟦*p=n⟧ = [p ↦ ?, q ↦ NN, n ↦ ?, alloc-1 ↦ ?]

- At the program point before the statement `*q=n` the analysis now knows that q is definitely non-null
- … and before `*p=n`, the pointer p is maybe null
- Due to the weak updates for all heap store operations, precision is bad for `alloc-i` locations

Agenda

- Introduction to points-to analysis
- Andersen’s analysis
- Steensgaards’s analysis
- Interprocedural points-to analysis
- Null pointer analysis
- **Flow-sensitive points-to analysis**

Points-to graphs

- Graphs that describe possible heaps:
  - nodes are abstract cells
  - edges are possible pointers between the cells
- The lattice of points-to graphs is $2^{Cells \times Cells}$ ordered under subset inclusion
  (or alternatively, $Cells \rightarrow 2^{Cells}$)
- For every CFG node, v, we introduce a constraint variable ⟦v⟧ describing the state *after* v
- Intraprocedural analysis (i.e. ignore function calls)

Constraints

- For pointer operations:
  - X = `alloc` P: ⟦v⟧ = $JOIN$(v)$\downarrow X \cup$ { (X, `alloc`-i) }
  - X = &Y: ⟦v⟧ = $JOIN$(v)$\downarrow X \cup$ { (X, Y) }
  - X = Y: ⟦v⟧ = $assign$($JOIN$(v), X, Y)
  - X = *Y: ⟦v⟧ = $load$($JOIN$(v), X, Y)
  - *X = Y: ⟦v⟧ = $store$($JOIN$(v), X, Y)
  - X = `null`: ⟦v⟧ = $JOIN$(v)$\downarrow X$

- For all other CFG nodes:
  - ⟦v⟧ = $JOIN$(v)

Auxiliary functions

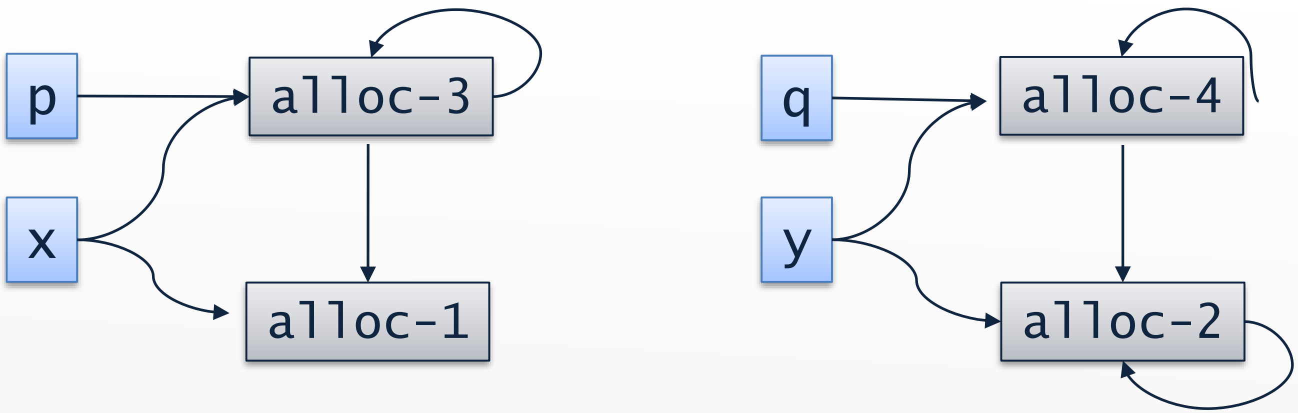
- $JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$
- $\sigma{\downarrow}X = \{ (s,t) \in \sigma \mid s \neq X \}$
- $assign(\sigma, X, Y) = \sigma{\downarrow}X \cup \{ (X, t) \mid (Y, t) \in \sigma \}$
- $load(\sigma, X, Y) = \sigma{\downarrow}X \cup \{ (X, t) \mid (Y, s) \in \sigma, (s, t) \in \sigma \}$
- $store(\sigma, X, Y) = \sigma \cup \{ (s, t) \mid (X, s) \in \sigma, (Y, t) \in \sigma \}$
  - note: weak update!

Example program

```
var x,y,n,p,q;
x = alloc null; y = alloc null;
*x = null; *y = y;
n = input;
while (n>0) {
  p = alloc null; q = alloc null;
  *p = x; *q = y;
  x = p; y = q;
  n = n-1;
}
```

Result of analysis

- After the loop we have this points-to graph:



- We conclude that x and y will always be disjoint

Points-to maps from points-to graphs

- A points-to map for each program point v:

  $$pt(X) = \{\, t \mid (X,t) \in [\![\mathrm{v}]\!] \,\}$$

- More expensive, but more precise:
  - Andersen: $pt(\mathrm{x}) = \{\, \mathrm{y}, \mathrm{z} \,\}$
  - flow-sensitive: $pt(\mathrm{x}) = \{\, \mathrm{z} \,\}$

```
x = &y;
x = &z;
```

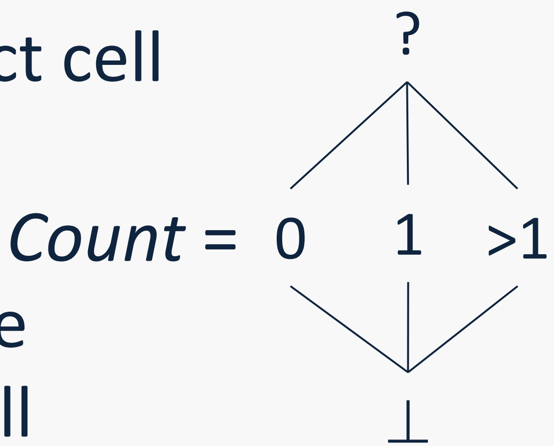
Improving precision with abstract counting

- The points-to graph is missing information:
  - `alloc-2` nodes always form a self-loop in the example

- We need a more detailed lattice:

$$2^{Cell \times Cell} \times (Cell \to Count)$$

  where we for each cell keep track of how many concrete cells that abstract cell describes



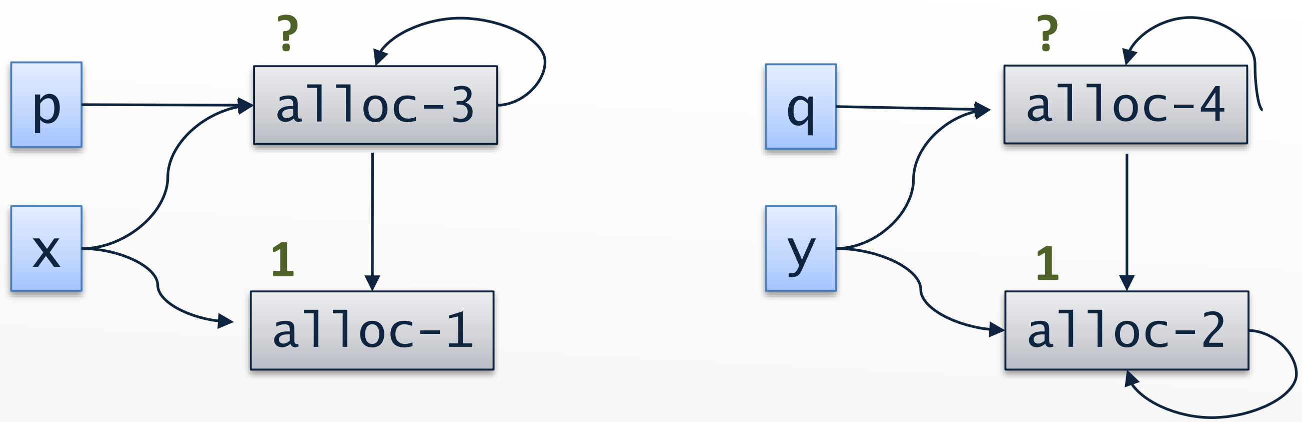
- This permits **strong updates** on those that describe precisely 1 concrete cell

Constraints

- X= `alloc` P: ...
- $*X = Y$: ...
- ...

Better results

- After the loop we have this extended points-to graph:



- Thus, `alloc-2` nodes form a self-loop

Interprocedural shape analysis

New issues to consider:

- parameter passing etc.
- weak updates to stack cells
- escaping of stack cells

Escape analysis

- Perform a points-to analysis
- Look at return expression
- Check reachability in the points-to graph to arguments or variables defined in the function itself

- None of those

  ⇓

  no escaping stack cells

```
baz()  {
  var x;
  return &x;
}

main() {
  var p;
  p=baz();
  *p=1;
  return *p;
}
```