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 pointer analysis
• Andersen’s analysis
• Steensgaard’s analysis
• Interprocedural pointer analysis
• Records and objects
• Null pointer analysis
• Flow-sensitive pointer analysis
Analyzing programs with pointers

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

\[ ... \]
\[ \ast x = 42; \]
\[ \ast y = -87; \]
\[ z = \ast x; \]
\[ // is z 42 or -87? \]

\[ Exp \rightarrow ... \]
\[ | \text{alloc } E \]
\[ | \&Id \]
\[ | \ast Exp \]
\[ | \text{null} \]

\[ Stm \rightarrow ... \]
\[ | \ast Id = Exp; \]
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 functions as values
• We still have many interesting analysis challenges...
Pointer targets

• The fundamental question about pointers:
  *What cells 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*

• 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

- 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

...  
*x* = 42;  
*y* = -87;  
z = *x;  
// is z 42 or -87?
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:
  • $X = \text{alloc } P$ where $P$ is null or an integer constant
  • $X = &Y$
  • $X = Y$
  • $X = *Y$
  • $*X = Y$
  • $X = \text{null}$

• 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 are uninitialized (otherwise it is impossible to get useful results from a flow-insensitive analysis)
Agenda

• Introduction to pointer analysis
• Andersen’s analysis
• Steensgaard’s analysis
• Interprocedural pointer analysis
• Records and objects
• Null pointer analysis
• Flow-sensitive pointer analysis
Andersen’s analysis (1/2)

• For every cell \(c\), introduce a constraint variable \([c]\) ranging over sets of cells, i.e. \([\cdot] : Cells \rightarrow \mathcal{P}(Cells)\)

• Generate constraints:
  • \(X = \text{alloc} P: \) \(\text{alloc} - i \in [X]\)
  • \(X = \&Y: \) \(Y \in [X]\)
  • \(X = Y: \) \([Y] \subseteq [X]\)
  • \(X = \text{xor} Y: \) \(c \in [Y] \Rightarrow [c] \subseteq [X]\) for each \(c \in Cells\)
  • \(*X = Y: \) \(c \in [X] \Rightarrow [Y] \subseteq [c]\) for each \(c \in Cells\)
  • \(X = \text{null}: \) (no constraints)

(For the conditional constraints, there’s no need to add a constraint for the cell \(x\) if \&\(x\) does not occur in the program)
Andersen’s analysis (2/2)

• The points-to map is defined as:
  \[ pt(X) = \llbracket X \rrbracket \]

• 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 \textit{directional}
  – models the direction of the flow of values in assignments
Example program

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

$Cells = \{ p, q, x, y, z, alloc-1 \}$
Applying Andersen

• Generated constraints:

\[ alloc-1 \in [p] \]
\[ [y] \subseteq [x] \]
\[ [z] \subseteq [x] \]
\[ c \in [p] \Rightarrow [z] \subseteq [\alpha] \text{ for each } c \in Cells \]
\[ [q] \subseteq [p] \]
\[ y \in [q] \]
\[ c \in [p] \Rightarrow [\alpha] \subseteq [x] \text{ for each } c \in Cells \]
\[ z \in [p] \]

• Smallest solution:

\[ pt(p) = \{ alloc-1, y, z \} \]
\[ pt(q) = \{ y \} \]
\[ pt(x) = pt(y) = pt(z) = \emptyset \]
Agenda

- Introduction to pointer analysis
- Andersen’s analysis
- Steensgaard’s analysis
- Interprocedural pointer analysis
- Records and objects
- Null pointer analysis
- Flow-sensitive pointer analysis
Steensgaard’s analysis

- View assignments as being bidirectional

- Generate constraints:
  - \( X = \text{alloc } P: \) \( \text{alloc}_i \in [X] \)
  - \( X = \& Y: \) \( Y \in [X] \)
  - \( X = Y: \) \( [X] = [Y] \)
  - \( X = \* Y: \) \( c \in [Y] \Rightarrow [c] = [X] \) for each \( c \in \text{Cells} \)
  - \( \* X = Y: \) \( c \in [X] \Rightarrow [Y] = [c] \) for each \( c \in \text{Cells} \)

- Extra constraints:
  \[ c_1, c_2 \in [c] \Rightarrow [c_1] = [c_2] \quad \text{and} \quad [c_1] \cap [c_2] \neq \emptyset \Rightarrow [c_1] = [c_2] \]
  (whenever a cell may point to two cells, they are essentially merged into one)

- Steensgaard’s original formulation uses conditional unification for \( X = Y: \)
  \( c \in [Y] \Rightarrow [X] = [Y] \) for each \( c \in \text{Cells} \) (avoids unifying if \( Y \) is never a pointer)
Steensgaard’s analysis

• Reformulate as term unification
• Generate constraints:
  • \( X = \text{alloc} \ P \): \( \lbrack X \rbrack = \uparrow \lbrack \text{alloc}-i \rbrack \)
  • \( X = \&Y \): \( \lbrack X \rbrack = \uparrow \lbrack Y \rbrack \)
  • \( X = Y \): \( \lbrack X \rbrack = \lbrack Y \rbrack \)
  • \( X = Y \): \( \lbrack Y \rbrack = \uparrow \alpha \land \lbrack X \rbrack = \alpha \) where \( \alpha \) is fresh
  • \( *X = Y \): \( \lbrack X \rbrack = \uparrow \alpha \land \lbrack Y \rbrack = \alpha \) where \( \alpha \) is fresh

• Terms:
  – term variables, e.g. \( \lbrack X \rbrack \), \( \lbrack \text{alloc}-i \rbrack \), \( \alpha \) (each representing the possible values of a cell)
  – each a single (unary) term constructor \( \uparrow t \) (representing pointers)
  – each \( \lbrack c \rbrack \) is now a term variable, not a constraint variable holding a set of cells

• Fits with our unification solver! (union-find...)
• The points-to map is defined as \( \text{pt}(X) = \{ c \in Cells \mid \lbrack X \rbrack = \uparrow \lbrack c \rbrack \} \)
• Note that there is only one kind of term constructor, so unification never fails
Applying Steensgaard

• Generated constraints:

```plaintext
alloc-1 ∈ [p]
[y] = [x]
[z] = [x]
α ∈ [p] ⇒ [z] = [α]
[q] = [p]
y ∈ [q]
α ∈ [p] ⇒ [α] = [x]
z ∈ [p]
+ the extra constraints
```

• Smallest solution:

```plaintext
pt(p) = { alloc-1, y, z }
pt(q) = { alloc-1, y, z }
...```
Another example

Andersen:

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

Steensgaard:
Recall our type analysis...

• Focusing on pointers...
• Constraints:
  • \( X = \text{alloc} \ P: \ [X] = \uparrow[P] \)
  • \( X = \&Y: \ [X] = \uparrow[Y] \)
  • \( X = Y: \ [X] = [Y] \)
  • \( X = *Y: \ \uparrow[X] = [Y] \)
  • \( *X = Y: \ [X] = \uparrow[Y] \)
• Implicit extra constraint for term equality:
  \( \uparrow t_1 = \uparrow t_2 \Rightarrow t_1 = t_2 \)

• Assuming the program type checks, is the solution for pointers the same as for Steensgaard’s analysis?
Agenda

- Introduction to pointer analysis
- Andersen’s analysis
- Steensgaard’s analysis
- Interprocedural pointer analysis
- Records and objects
- Null pointer analysis
- Flow-sensitive pointer analysis
Interprocedural pointer analysis

• In TIP, function values and pointers may be mixed together:

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

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

• The idea: Treat function values as a kind of pointers
Function call normalization

• Assume that all function calls are of the form

\[ X = X_0(X_1, \ldots, X_n) \]

• \( y \) may be a variable whose value is a function pointer

• Assume that all return statements are of the form

\[ \text{return } X'; \]

• As usual, simply introduce lots of temporary variables...

• Include all function names in \textit{Cells}
CFA with Andersen

• For the function call
  \[ X = X_0(X_1, \ldots, X_n) \]
  and every occurrence of
  \[ f(X'_1, \ldots, X'_n) \{ \ldots \text{return } X' \}; \]
  add these constraints:

  \[
  f \in \llbracket f \rrbracket
  \]
  \[
  f \in \llbracket X_0 \rrbracket \Rightarrow (\llbracket X_i \rrbracket \subseteq \llbracket X'_i \rrbracket \text{ for } i=1,\ldots,n \land \llbracket X' \rrbracket \subseteq \llbracket X \rrbracket)
  \]

• (Similarly for simple function calls)
• Fits directly into the cubic framework!
CFA with Steensgaard

- For the function call
  \[ X = X_0(X_1, \ldots, X_n) \]
  and every occurrence of
  \[ f(X'_1, \ldots, X'_n) \{ \ldots \text{return } X'\; \} \]
  add these constraints:

  \[
  f \in \llbracket f \rrbracket \\
  f \in \llbracket X_0 \rrbracket \Rightarrow (\llbracket X_i \rrbracket = \llbracket X'_i \rrbracket \text{ for } i=1,\ldots,n \land \llbracket X' \rrbracket = \llbracket X \rrbracket)
  \]

- (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 \( \text{Cells} \rightarrow \mathcal{P}(\text{Cells}) \) to
  \[ \text{Contexts} \rightarrow \text{Cells} \rightarrow \mathcal{P}(\text{Cells}) \]
  (or equivalently: \( \text{Cells} \times \text{Contexts} \rightarrow \mathcal{P}(\text{Cells}) \))
  where \( \text{Contexts} \) is a (finite) set of call contexts

• As usual, many possible choices of \( \text{Contexts} \)
  – recall the call string approach and the functional approach

• We can also track the set of reachable contexts
  (like the use of lifted lattices earlier):
  \[ \text{Contexts} \rightarrow \text{lift}(\text{Cells} \rightarrow \mathcal{P}(\text{Cells})) \]

• 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

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

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

Are $x$ and $y$ aliases? $[x] = \{\text{alloc-1}\}$ $[y] = \{\text{alloc-1}\}$
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\textsubscript{–}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
Context-sensitive pointer analysis with heap cloning

Assuming we use the call string approach with k=1, so Contexts = {ε, c1, c2}, and HeapContexts = Contexts

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

baz() {
    var x, y;
    x = mk(); // c1
    y = mk(); // c2
    ...
}
```

Are x and y aliases?  

\[
[x] = \{(alloc-1, c1)\} \\
[y] = \{(alloc-1, c2)\}
\]
- Introduction to pointer analysis
- Andersen’s analysis
- Steensgaard’s analysis
- Interprocedural pointer analysis
- Records and objects
- Null pointer analysis
- Flow-sensitive pointer analysis
Records in TIP

\[
\text{Exp} \rightarrow \ldots \\
\mid \{ \text{Id} : \text{Exp}, \ldots, \text{Id} : \text{Exp} \} \\
\mid \text{Exp} \cdot \text{Id}
\]

- Field write operations: see SPA...
- Values of record fields cannot themselves be records
- After normalization:
  - \( X = \{ F_1 : X_1, \ldots, F_k : X_k \} \)
  - \( X = \text{alloc} \{ F_1 : X_1, \ldots, F_k : X_k \} \)
  - \( X = Y.F \)

Let us extend Andersen’s analysis accordingly...
Constraint variables for record fields

- $\llbracket \cdot \rrbracket : (Cells \cup (Cells \times Fields)) \rightarrow \mathcal{P}(Cells)$
  where $\mathcal{P}(Cells)$ is the set of field names in the program

- Notation: $\llbracket c.f \rrbracket$ means $\llbracket (c, f) \rrbracket$
Analysis constraints

• \( X = \{ F_1 : X_1, \ldots, F_k : X_k \} : \ [X_1] \subseteq [X.F_1] \land \ldots \land [X_k] \subseteq [X.F_k] \)

• \( X = \text{alloc} \{ F_1 : X_1, \ldots, F_k : X_k \} : \ \text{alloc}-i \in [X] \land \ [X_1] \subseteq [\text{alloc}-i.F_1] \land \ldots \land [X_k] \subseteq [\text{alloc}-i.F_k] \)

• \( X = Y.F : \ [Y.F] \subseteq [X] \)

• \( X = Y : \ [Y] \subseteq [X] \land [Y.F] \subseteq [X.F] \) for each \( F \in \text{Fields} \)

• \( X = \ast Y : \ c \in [Y] \Rightarrow ([c] \subseteq [X] \land [c.F] \subseteq [X.F]) \) for each \( c \in \text{Cells} \) and \( F \in \text{Fields} \)

• \( \ast X = Y : \ c \in [X] \Rightarrow ([Y] \subseteq [c] \land [Y.F] \subseteq [c.F]) \) for each \( c \in \text{Cells} \) and \( F \in \text{Fields} \)

See example in SPA
Objects as mutable heap records

```
Exp → ...
  | Id
  | alloc { Id: Exp, ..., Id: Exp }
  | (*Exp).Id
  | null

Stm → ...
  | Id = Exp;
  | (*Exp).Id = Exp;
```

- E.X in Java corresponds to (*E).X in TIP (or C)
- Can only create pointers to heap-allocated records (=objects), not to variables or to cells containing non-record values
Agenda

• Introduction to pointer analysis
• Andersen’s analysis
• Steensgaard’s analysis
• Interprocedural pointer analysis
• Records and objects
• Null pointer analysis
• Flow-sensitive pointer analysis
Null pointer analysis

• Decide for every dereference \(*p\), is \(p\) different from null?

• (Why not just treat null as a special cell 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 not null” and ? represents “maybe null”

• Use for every program point the map lattice:

$$Cells \rightarrow Null$$

(here for TIP without records)
• For every CFG node, \( v \), we have a variable \( \llbracket v \rrbracket \):
  – a map giving abstract values for all cells at the program point *after* \( v \)

• Auxiliary definition:

\[
JOIN(v) = \bigsqcup_{w \in \text{pred}(v)} \llbracket w \rrbracket
\]

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

• For operations involving pointers:
  • $X = \text{alloc } P$: $[v] = ???$ where $P$ is null or an integer constant
  • $X = &Y$: $[v] = ???$
  • $X = Y$: $[v] = ???$
  • $X = *Y$: $[v] = ???$
  • $*X = Y$: $[v] = ???$
  • $X = \text{null}$: $[v] = ???$

• For all other CFG nodes:
  • $[v] = \text{JOIN}(v)$
Null analysis constraints

• For a heap store operation $^X X = Y$ 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:

$$^X X = Y: \quad \llbracket v \rrbracket = store(\text{JOIN}(v), X, Y)$$

where $store(\sigma, X, Y) = \sigma[\alpha \mapsto \sigma(\alpha) \cup \sigma(Y)]$

$$\alpha \in pt(X)$$
Null analysis constraints

• For a heap load operation $X = \star Y$ 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:

$$X = \star Y: \quad [v] = \text{load}(\text{JOIN}(v), X, Y)$$

where $\text{load}(\sigma, X, Y) = \sigma[X \mapsto \bigsqcup \sigma(\alpha)]$

$$\alpha \in \text{pt}(Y)$$
Strong and weak updates

The abstract cell alloc-1 corresponds to *multiple concrete cells*
Strong and weak updates

```plaintext
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;
```

The points-to set for x contains multiple abstract cells

is C null here?
Null analysis constraints

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

- 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:** \( \sigma[c \mapsto \sigma(c) \cup \textit{new-value}] \)
  
  – 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

```
\[ \text{function } f(b_1, \ldots, b_n) \]
```

```
x = f(E_1, \ldots, E_n);
result = E;
```
Using the null analysis

- The pointer dereference \( *p \) is “safe” at entry of \( v \) if
  \[
  JOIN(v)(p) = \text{NN}
  \]

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

\[
p = \text{alloc null}; \\
q = \&p; \\
n = \text{null}; \\
*q = n; \\
*p = n;
\]

Andersen generates:

\[
pt(p) = \{\text{alloc-1}\} \\
pt(q) = \{p\} \\
pt(n) = \emptyset
\]
Generated constraints

\[
[p = \text{alloc null}] = \perp \![p \mapsto \text{NN, alloc-1} \mapsto ?] \\
[q = \& p] = [p = \text{alloc null}][q \mapsto \text{NN}] \\
[n = \text{null}] = [q = \& p][n \mapsto {?}] \\
[* q = n] = [n = \text{null}][p \mapsto [n = \text{null}] (p) \sqcup [n = \text{null}] (n)] \\
[* p = n] = [* q = n][\text{alloc-1} \mapsto [* q = n] (\text{alloc-1}) \sqcup [* q = n] (n)]
\]
Solution

\[ [p=\text{alloc null}] = [p \mapsto \text{NN}, q \mapsto \text{NN}, n \mapsto \text{NN}, \text{alloc-1} \mapsto ?] \]
\[ [q=&p] = [p \mapsto \text{NN}, q \mapsto \text{NN}, n \mapsto \text{NN}, \text{alloc-1} \mapsto ?] \]
\[ [n=\text{null}] = [p \mapsto \text{NN}, q \mapsto \text{NN}, n \mapsto ?, \text{alloc-1} \mapsto ?] \]
\[ [*q=n] = [p \mapsto ?, q \mapsto \text{NN}, n \mapsto ?, \text{alloc-1} \mapsto ?] \]
\[ [*p=n] = [p \mapsto ?, q \mapsto \text{NN}, n \mapsto ?, \text{alloc-1} \mapsto ?] \]

- 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-1 cells
Agenda

• Introduction to pointer analysis
• Andersen’s analysis
• Steensgaard’s analysis
• Interprocedural pointer analysis
• Records and objects
• Null pointer analysis
• Flow-sensitive pointer 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 $\mathcal{P}(Cells \times Cells)$ ordered under subset inclusion
  (or alternatively, $Cells \rightarrow \mathcal{P}(Cells)$)

• 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:
  • \( X = \text{alloc} \ P \): \[ \llbracket v \rrbracket = JOIN(v) \downarrow X \cup \{ (X, \text{alloc} - i) \} \]
  • \( X = \& Y \): \[ \llbracket v \rrbracket = JOIN(v) \downarrow X \cup \{ (X, Y) \} \]
  • \( X = Y \): \[ \llbracket v \rrbracket = JOIN(v) \downarrow X \cup \{ (X, t) | (Y, t) \in JOIN(v) \} \]
  • \( X = \* Y \): \[ \llbracket v \rrbracket = \text{JOIN}(v) \cup \{ (X, t) | (Y, s) \in \sigma, (s, t) \in JOIN(v) \} \]
  • \( \* X = Y \): \[ \llbracket v \rrbracket = \text{JOIN}(v) \cup \{ (s, t) | (X, s) \in \text{JOIN}(v), (Y, t) \in \text{JOIN}(v) \} \]
  • \( X = \text{null} \): \[ \llbracket v \rrbracket = \text{JOIN}(v) \downarrow X \]

where \( \sigma \downarrow X = \{ (s, t) \in \sigma | s \neq X \} \)

\( join(v) = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket \)

note: weak update!

• For all other CFG nodes:
  • \( \llbracket v \rrbracket = \text{JOIN}(v) \)
Example program

```plaintext
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 $v$:
  
  $$pt(X) = \{ t \mid (X,t) \in \llbracket v \rrbracket \}$$

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

```plaintext
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:
  \[ \mathcal{P}(Cells \times Cells) \times (Cell \rightarrow Count) \]
  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
Better results

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

• Thus, alloc-2 cells form a self-loop
• Both alloc-1 and alloc-2 permit strong updates
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;
}