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?)

```
&Id
alloc E
*Exp
null
```

```
Exp → ...

Stm → ...

| *ld = Exp;
```

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

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

• Let’s also ignore functions as values for now
• 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, *Cell*, 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

```c
... *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. ![c]: Cell → P(Cell)

• Generate constraints:
  • ![X] = alloc P: alloc-i ∈ ![X]
  • ![X] = &Y: Y ∈ ![X]
  • ![X] = Y: ![Y] ⊆ ![X]
  • ![X] = *Y: c ∈ ![Y] ⇒ ![c] ⊆ ![X] for each c ∈ Cell
  • *X = Y: c ∈ ![X] ⇒ ![Y] ⊆ ![c] for each c ∈ Cell
  • ![X] = 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:
  \[ pt(X) = \lceil X \rceil \]

• 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

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

Cell = {p, q, x, y, z, alloc-1}
Applying Andersen

• Generated constraints:

\[
\begin{align*}
\text{alloc-1} & \in \lbrack p \rbrack \\
\lbrack y \rbrack & \subseteq \lbrack x \rbrack \\
\lbrack z \rbrack & \subseteq \lbrack x \rbrack \\
c & \in \lbrack p \rbrack \Rightarrow \lbrack z \rbrack \subseteq \lbrack c \rbrack \text{ for each } c \in Cell \\
\lbrack q \rbrack & \subseteq \lbrack p \rbrack \\
y & \in \lbrack q \rbrack \\
c & \in \lbrack p \rbrack \Rightarrow \lbrack c \rbrack \subseteq \lbrack x \rbrack \text{ for each } c \in Cell \\
z & \in \lbrack p \rbrack
\end{align*}
\]

• Smallest solution:

\[
\begin{align*}
pt(p) &= \{ \text{alloc-1}, y, z \} \\
pt(q) &= \{ y \} \\
pt(x) &= pt(y) = pt(z) = \emptyset
\end{align*}
\]
A specialized cubic solver

- At each load/store instruction, instead of generating a conditional constraint for each cell, generate a single universally quantified constraint:
  - $t \in \llbracket x \rrbracket$
  - $\llbracket x \rrbracket \subseteq \llbracket y \rrbracket$
  - $\forall t \in \llbracket x \rrbracket: \llbracket t \rrbracket \subseteq \llbracket y \rrbracket$
  - $\forall t \in \llbracket x \rrbracket: \llbracket y \rrbracket \subseteq \llbracket t \rrbracket$

- Whenever a token is added to a set, lazily add new edges according to the universally quantified constraints
- Note that every token is also a constraint variable here
- Still cubic complexity, but faster in practice
A specialized cubic solver

- $x.\text{sol} \subseteq T$: the set of tokens for $x$ (the bitvectors)
- $x.\text{succ} \subseteq V$: the successors of $x$ (the edges)
- $x.\text{from} \subseteq V$: the first kind of quantified constraints for $x$
- $x.\text{to} \subseteq V$: the second kind of quantified constraints for $x$
- $W \subseteq T \times V$: a worklist (initially empty)

Implementation: SpecialCubicSolver
A specialized cubic solver

- \( t \in [x] \)
  ```
  addToken(t, x)
  propagate()
  ```

- \([x] \subseteq [y]\)
  ```
  addEdge(x, y)
  propagate()
  ```

- \( \forall t \in [x]: [t] \subseteq [y]\)
  ```
  add y to x.from
  for each t in x.sol
    addEdge(t, y)
  propagate()
  ```

- \( \forall t \in [x]: [y] \subseteq [t]\)
  ```
  add y to x.to
  for each t in x.sol
    addEdge(y, t)
  propagate()
  ```

addToken(t, x):
  ```
  if t \notin x.sol
    add t to x.sol
    add (t, x) to W
  ```

addEdge(x, y):
  ```
  if x \neq y \land y \notin x.succ
    add y to x.succ
    for each t in x.sol
      addToken(t, y)
  ```

propagate():
  ```
  while W \neq \emptyset
    pick and remove (t, x) from W
    for each y in x.from
      addEdge(t, y)
    for each y in x.to
      addEdge(y, t)
    for each y in x.succ
      addToken(t, y)
  ```
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:
  • \( X = \text{alloc } P: \) alloc-\( i \in [X] \)
  • \( X = \&Y: \) \( Y \in [X] \)
  • \( X = Y: \) \([X] = [Y] \)
  • \( X = *Y: \) \( c \in [Y] \Rightarrow [c] = [X] \) for each \( c \in Cell \)
  • \( *X = Y: \) \( c \in [X] \Rightarrow [Y] = [c] \) for each \( c \in Cell \)

• Extra constraints:
  \[ c_1, c_2 \in [c] \Rightarrow [c_1] = [c_2] \text{ and } [c_1] \cap [c_2] \neq \emptyset \Rightarrow [c_1] = [c_2] \]
  (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 \( c \in Cell \) (avoids unifying if \( Y \) is never a pointer)
Steensgaard’s analysis

- Reformulate as term unification
- Generate constraints:
  - \( X = alloc \ P: \) \( \llbracket X \rrbracket = \uparrow \llbracket alloc - i \rrbracket \)
  - \( X = \& Y: \) \( \llbracket X \rrbracket = \uparrow \llbracket Y \rrbracket \)
  - \( X = Y: \) \( \llbracket X \rrbracket = \llbracket Y \rrbracket \)
  - \( X = * Y: \) \( \llbracket Y \rrbracket = \uparrow \alpha \land \llbracket X \rrbracket = \alpha \) where \( \alpha \) is fresh
  - \( * X = Y: \) \( \llbracket X \rrbracket = \uparrow \alpha \land \llbracket Y \rrbracket = \alpha \) where \( \alpha \) is fresh

- Terms:
  - term variables, e.g. \( \llbracket X \rrbracket, \llbracket alloc - i \rrbracket, \alpha \) (each representing the possible values of a cell)
  - a single (unary) term constructor \( \uparrow t \) (representing pointers)
  - each \( \llbracket c \rrbracket \) 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 \( pt(X) = \{ c \in Cell \mid \llbracket X \rrbracket = \uparrow \llbracket c \rrbracket \} \)
- Note that there is only one kind of term constructor, so unification never fails
Applying Steensgaard

- Generated constraints (as sets or terms, respectively):

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc-1 ∈ [p]</td>
<td>[p] = ↑{alloc-1}</td>
</tr>
<tr>
<td>[y] = [x]</td>
<td>[y] = [x]</td>
</tr>
<tr>
<td>[z] = [x]</td>
<td>[z] = [x]</td>
</tr>
<tr>
<td>c ∈ [p] ⇒ [z] = [c] for each c ∈ Cell</td>
<td>[p] = ↑α_1, [z] = α_1</td>
</tr>
<tr>
<td>[q] = [p]</td>
<td>[q] = [p]</td>
</tr>
<tr>
<td>y ∈ [q]</td>
<td>[q] = ↑[y]</td>
</tr>
<tr>
<td>c ∈ [p] ⇒ [c] = [x] for each c ∈ Cell</td>
<td>[p] = ↑α_2, [x] = α_2</td>
</tr>
<tr>
<td>z ∈ [p]</td>
<td>[p] = ↑[z]</td>
</tr>
<tr>
<td>+ the extra constraints</td>
<td></td>
</tr>
</tbody>
</table>

- Smallest solution:

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

...
Another example

Andersen:

\[ a1 = \&b1; \]
\[ b1 = \&c1; \]
\[ c1 = \&d1; \]
\[ a2 = \&b2; \]
\[ b2 = \&c2; \]
\[ c2 = \&d2; \]

Steensgaard:

\[ a1 \]
\[ b1 \]
\[ c1 \]
\[ d1 \]
\[ a2 \]
\[ b2 \]
\[ c2 \]
\[ d2 \]
Recall our type analysis...

• Focusing on pointers...

• Constraints:
  - \( X = \text{alloc } P: \) \([X] = \uparrow[P]\)
  - \( X = \&Y: \) \([X] = \uparrow[Y]\)
  - \( X = Y: \) \([X] = [Y]\)
  - \( X = \ast Y: \) \(\uparrow[X] = [Y]\)
  - \( \ast X = Y: \) \([X] = \uparrow[Y]\)

• 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:
  \[(\ast\ast\ast x)(1, 2, 3)\]

• 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

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

• 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 the set \textit{Cell}
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:

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

• (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

```
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

• Generalize the abstract domain $\text{Cell} \rightarrow \mathcal{P}(\text{Cell})$ to $\text{Context} \rightarrow \text{Cell} \rightarrow \mathcal{P}(\text{Cell})$

( or equivalently: $\text{Cell} \times \text{Context} \rightarrow \mathcal{P}(\text{Cell})$)

where $\text{Context}$ is a (finite) set of call contexts

• As usual, many possible choices of the set $\text{Context}$
  – 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):

  $\text{Context} \rightarrow \text{lift}(\text{Cell} \rightarrow \mathcal{P}(\text{Cell}))$

• Does this still fit into the cubic solver?
Context-sensitive pointer analysis

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

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

Are `x` and `y` aliases? 

\[ [x] = \{alloc-1\} \]
\[ [y] = \{alloc-1\} \]
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\_i (the alloc with index i) or X (a program variable)
  – a heap context from a (finite) set HeapContext

• This allows abstract cells to be named by the source code allocation site and (information from) the current context

• One choice:
  – set HeapContext = Context
  – 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 $Context = \{\varepsilon, c_1, c_2\}$, and $HeapContext = Context$

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

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

Are $x$ and $y$ aliases? 

$$[x] = \{(alloc-1, c_1)\}$$

$$[y] = \{(alloc-1, c_2)\}$$
Agenda

• 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

\[
Ex \rightarrow \ldots
\]

| \{ Id:Exp, \ldots, Id:Exp \} |
| Exp . 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

• $\llbracket \cdot \rrbracket : (Cell \cup (Cell \times Field)) \to \mathcal{P}(Cell)$
  where $Field$ is the set of field names in the program

• Notation: $\llbracket c \cdot f \rrbracket$ means $\llbracket (c, f) \rrbracket$
Analysis constraints

- $X = \{ F_1:X_1, \ldots, F_k:X_k \}$: $[X_1] \subseteq [X.F_1] \land \cdots \land [X_k] \subseteq [X.F_k]$
- $X = \text{alloc} \{ F_1:X_1, \ldots, F_k:X_k \}$: $\text{alloc} - i \in [X] \land [X_1] \subseteq [\text{alloc} - i.F_1] \land \cdots \land [X_k] \subseteq [\text{alloc} - i.F_k]$
- $X = Y.F$: $[Y.F] \subseteq [X]$

- $X = Y$: $[Y] \subseteq [X] \land [Y.F] \subseteq [X.F]$ for each $F \in \text{Field}$
- $X = \ast Y$: $c \in [Y] \Rightarrow ([c] \subseteq [X] \land [c.F] \subseteq [X.F])$
  for each $c \in \text{Cell}$ and $F \in \text{Field}$
- $\ast X = Y$: $c \in [X] \Rightarrow ([Y] \subseteq [c] \land [Y.F] \subseteq [c.F])$
  for each $c \in \text{Cell}$ and $F \in \text{Field}$

See example in SPA
Objects as mutable heap records

- 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 \texttt{null}?

• (Why not just treat \texttt{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 \textit{Null}:

\[
\begin{array}{c}
\text{?} \\
\text{\_} \\
\text{NN}
\end{array}
\]

where NN represents “definitely not null” and ? represents “maybe null”

• Use for every program point the map lattice:

\[ Cell \rightarrow Null \]

(here for TIP without records)
Setting up

• 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:
  • $X = \text{alloc } P$: $\llbracket v \rrbracket = ???$ where $P$ is null or an integer constant
  • $X = \&Y$: $\llbracket v \rrbracket = ???$
  • $X = Y$: $\llbracket v \rrbracket = ???$
  • $X = *Y$: $\llbracket v \rrbracket = ???$
  • $*X = Y$: $\llbracket v \rrbracket = ???$
  • $X = \text{null}$: $\llbracket v \rrbracket = ???$

• For all other CFG nodes:
  • $\llbracket v \rrbracket = \text{JOIN}(v)$
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 $\text{alloc}-i$ may describe *multiple* concrete cells

• So we settle for **weak** update:

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

where

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

- For a heap load operation $X = \ast Y$ 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:

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

where $\text{load}(\sigma, X, Y) = \sigma[X \mapsto \bigcup_{\alpha \in pt(Y)} \sigma(\alpha)]$
Strong and weak updates

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

```c
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

• $X = \text{alloc } P$: $[v] = JOIN(v)\{X \mapsto \text{NN}, \text{alloc}_i \mapsto ?\}$
• $X = \&Y$: $[v] = JOIN(v)\{X \mapsto \text{NN}\}$
• $X = Y$: $[v] = JOIN(v)\{X \mapsto \text{JOIN}(v)(Y)\}$
• $X = \text{null}$: $[v] = JOIN(v)\{X \mapsto ?\}$

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

could be improved...
Strong and weak updates, revisited

• **Strong update:** \( \sigma[c \mapsto \text{new-value}] \)
  – 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:** \( \sigma[c \mapsto \sigma(c) \cup \text{new-value}] \)
  – 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
  \[\text{JOIN}(v)(p) = \text{NN}\]

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

Andersen generates:

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

Andersen generates:

\[
\begin{align*}
  pt(p) &= \{\text{alloc}-1\} \\
  pt(q) &= \{p\} \\
  pt(n) &= \emptyset
\end{align*}
\]
Generated constraints

\[
\begin{align*}
[p=\text{alloc null}] &= \bot[p \mapsto \text{NN}, \text{alloc-1} \mapsto ?] \\
[q=&p] &= [p=\text{alloc null}][q \mapsto \text{NN}] \\
[n=\text{null}] &= [q=&p][n \mapsto ?] \\
[*q=n] &= [n=\text{null}][p \mapsto [n=\text{null}](p) \cup [n=\text{null}](n)] \\
[*p=n] &= [*q=n][\text{alloc-1} \mapsto [*q=n](\text{alloc-1}) \cup [*q=n](n)]
\end{align*}
\]
Solution

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

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

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

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

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

- 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-i 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 $\mathcal{P}(Cell \times Cell)$ ordered under subset inclusion (or alternatively, $Cell \to \mathcal{P}(Cell)$)

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

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

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

\( \text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket \)
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:
  \[ pt(X) = \{ t \mid (X,t) \in [v] \} \]

• More expensive, but more precise:
  – Andersen: \[ pt(x) = \{ y, z \} \]
  – flow-sensitive: \[ pt(x) = \{ z \} \]
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:
  \[
  \mathcal{P}(Cell \times Cell) \times (Cell \rightarrow \text{Count})
  \]
  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

\[
\text{Count} = \begin{cases} 0, & \text{if } ? = 1 \text{ or } 1 \text{ or } >1 \\ 1 \end{cases}
\]
Better results

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

  ![Graph Diagram]

  • 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

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