Static Program Analysis
Part 10 – pointer analysis

https://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?)

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

```
Exp → ...
| alloc E
| &Id
| *Exp
| null

Stm → ...
| *Id = Exp;
```
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, *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
• Assume that all pointer usage is normalized:
  • \( X = \text{alloc } P \) where \( P \) is \text{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:
  
  • $X = alloc \ P$: $alloc-i \in [X]$
  • $X = &Y$: $Y \in [X]$
  • $X = Y$: $[Y] \subseteq [X]$
  • $X = *Y$: $c \in [Y] \Rightarrow [c] \subseteq [X]$ for each $c \in Cells$
  • $*X = Y$: $c \in [X] \Rightarrow [Y] \subseteq [c]$ for each $c \in Cells$
  • $X = null$: (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 = &z;
q = &y;
x = *p;
p = &z;
```

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

• Generated constraints:

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

• 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:

  - $t \in [x]$
  - $[x] \subseteq [y]$
  - $\forall t \in [x]: [t] \subseteq [y]$
  - $\forall t \in [x]: [y] \subseteq [t]$

- 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

- \( t \in \llbracket x \rrbracket \)
  
  \[
  \text{addToken}(t, x)
  \]
  
  \[
  \text{propagate}()
  \]

- \( \llbracket x \rrbracket \subseteq \llbracket y \rrbracket \)
  
  \[
  \text{addEdge}(x, y)
  \]
  
  \[
  \text{propagate}()
  \]

- \( \forall t \in \llbracket x \rrbracket : \llbracket t \rrbracket \subseteq \llbracket y \rrbracket \)
  
  \[
  \text{add y to } x.\text{from}
  \]
  
  \[
  \text{for each } t \in x.\text{sol}
  \]
  
  \[
  \text{addEdge}(t, y)
  \]
  
  \[
  \text{propagate}()
  \]

- \( \forall t \in \llbracket x \rrbracket : \llbracket y \rrbracket \subseteq \llbracket t \rrbracket \)
  
  \[
  \text{add y to } x.\text{to}
  \]
  
  \[
  \text{for each } t \in x.\text{sol}
  \]
  
  \[
  \text{addEdge}(y, t)
  \]
  
  \[
  \text{propagate}()
  \]

- \( \text{addToken}(t, x) : \)
  
  \[
  \text{if } t \not\in x.\text{sol}
  \]
  
  \[
  \text{add } t \text{ to } x.\text{sol}
  \]
  
  \[
  \text{add } (t, x) \text{ to } W
  \]

- \( \text{addEdge}(x, y) : \)
  
  \[
  \text{if } x \neq y \land y \not\in x.\text{succ}
  \]
  
  \[
  \text{add } y \text{ to } x.\text{succ}
  \]
  
  \[
  \text{for each } t \text{ in } x.\text{sol}
  \]
  
  \[
  \text{addToken}(t, y)
  \]

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

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

• Steensgaard’s original formulation uses conditional unification for $X = Y$: $c \in \llbracket Y \rrbracket \Rightarrow \llbracket X \rrbracket = \llbracket Y \rrbracket$ for each $c \in Cells$ (avoids unifying if $Y$ is never a pointer)
Steensgaard’s analysis

• Reformulate as term unification
• Generate constraints:
  • \( X = \text{alloc} \ P \): \( \llbracket X \rrbracket = \uparrow \llbracket \text{alloc}-i \rrbracket \)
  • \( X = \& Y \): \( \llbracket X \rrbracket = \uparrow \llbracket Y \rrbracket \)
  • \( X = Y \): \( \llbracket X \rrbracket = \llbracket Y \rrbracket \)
  • \( X = * Y \): \( \llbracket Y \rrbracket = \uparrow \alpha \land \llbracket X \rrbracket = \alpha \) 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. \( \llbracket X \rrbracket, \llbracket \text{alloc}-i \rrbracket, \alpha \) (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 \( \text{pt}(X) = \{ c \in Cells \mid \llbracket X \rrbracket = \uparrow \llbracket c \rrbracket \} \)
• Note that there is only one kind of term constructor, so unification never fails
Applying Steensgaard

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

\[
\begin{align*}
\quad & alloc-1 \in [p] \\
\quad & [y] = [x] \\
\quad & [z] = [x] \\
\quad & c \in [p] \implies [z] = [c] \text{ for each } c \in Cells \\
\quad & [q] = [p] \\
\quad & y \in [q] \\
\quad & c \in [p] \implies [c] = [x] \text{ for each } c \in Cells \\
\quad & z \in [p] \\
\quad & + \text{ the extra constraints}
\end{align*}
\]

\[
\begin{align*}
\quad & [p] = \uparrow [alloc-1] \\
\quad & [y] = [x] \\
\quad & [z] = [x] \\
\quad & [p] = \uparrow \alpha_1 \quad [z] = \alpha_1 \\
\quad & [q] = [p] \\
\quad & [q] = \uparrow [y] \\
\quad & [p] = \uparrow \alpha_2 \quad [x] = \alpha_2 \\
\quad & [p] = \uparrow [z]
\end{align*}
\]

• Smallest solution:

\[
\begin{align*}
\quad & pt(p) = \{ alloc-1, y, z \} \\
\quad & pt(q) = \{ alloc-1, y, z \} \\
\quad & \ldots
\end{align*}
\]
Another example

Andersen:

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

Steensgaard:

```
a1 -> b1 -> c1 -> d1
a2 -> b2 -> c2 -> d2
```
Recall our type analysis...

• Focusing on pointers...

• Constraints:
  • $X = \text{alloc } P$: $\llbracket X \rrbracket = \uparrow \llbracket P \rrbracket$
  • $X = &Y$: $\llbracket X \rrbracket = \uparrow \llbracket Y \rrbracket$
  • $X = Y$: $\llbracket X \rrbracket = \llbracket Y \rrbracket$
  • $X = *Y$: $\uparrow \llbracket X \rrbracket = \llbracket Y \rrbracket$
  • $*X = Y$: $\llbracket X \rrbracket = \uparrow \llbracket Y \rrbracket$

• 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

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

• Assume that all return statements are of the form

```
return X';
```

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

• Include all function names in 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!

Andersen’s analysis is already closely connected to control-flow analysis!
CFA with Steensgaard

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

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

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

```c
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 $Cells \rightarrow \mathcal{P}(Cells)$ to $Contexts \rightarrow Cells \rightarrow \mathcal{P}(Cells)$
  (or equivalently: $Cells \times Contexts \rightarrow \mathcal{P}(Cells)$)
  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

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

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

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

int 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\_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 $Contexts = \{\varepsilon, c1, c2\}$, and $HeapContexts = Contexts$

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

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

Are $x$ and $y$ aliases? $[x] = \{(alloc-1, c1)\}$ $[y] = \{(alloc-1, c2)\}$
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

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

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 $Fields$ is the set of field names in the program

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

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

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

• $X = Y. F$: $[Y.F] \subseteq [X]$

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

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

• $* X = Y$: $c \in [X] \Rightarrow ([Y] \subseteq [c] \land [Y.F] \subseteq [c.F])$
  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
• 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:

\[ \text{Cells } \rightarrow \text{Null} \]

(here for TIP without records)
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:

\[ \text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} [w] \]

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

• For operations involving pointers:
  • $X = \text{alloc } P$: $\llbracket v \rrbracket = ???$
  • $X = &Y$: $\llbracket v \rrbracket = ???$
  • $X = Y$: $\llbracket v \rrbracket = ???$
  • $X = *Y$: $\llbracket v \rrbracket = ???$
  • $*X = Y$: $\llbracket v \rrbracket = ???$
  • $X = \text{null}$: $\llbracket v \rrbracket = ???$

  where $P$ is null or an integer constant

• For all other CFG nodes:
  • $\llbracket v \rrbracket = JOIN(v)$
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:

\[
\begin{align*}
\text{\textbf{*X = Y: \[v\] = store(JOIN(v), X, Y)}}
\end{align*}
\]

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:

$$X = \star Y: \quad \llbracket v \rrbracket = load(JOIN(v), X, Y)$$

where

$$load(\sigma, X, Y) = \sigma[X \mapsto \bigsqcup \sigma(\alpha)] \quad \alpha \in pt(Y)$$
Strong and weak updates

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

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

is `d` null here?

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

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

- In each case, the assignment modifies a program variable
- So we can use strong updates, as for heap load operations
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: $\sigma[c \mapsto \sigma(c) \sqcup \text{new-value}]$
  – 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 $JOIN(v)(p) = NN$

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

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

Andersen generates:

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

\[
\begin{align*}
[p=alloc\ \text{null}] &= \bot[p \leftrightarrow NN, alloc-1 \leftrightarrow ?] \\
[q=&p] &= [p=alloc\ \text{null}][q \leftrightarrow NN] \\
[n=null] &= [q=&p][n \leftrightarrow?] \\
[*q=n] &= [n=null][p \leftrightarrow [n=null](p) \sqcup [n=null](n)] \\
[*p=n] &= [*q=n][alloc-1 \leftrightarrow [*q=n](alloc-1) \sqcup [*q=n](n)]
\end{align*}
\]
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-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 $\mathcal{P}(Cells \times Cells)$ 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:
  • $X = \text{alloc } P$: $\llbracket v \rrbracket = \text{JOIN}(v) \downarrow X \cup \{ (X, \text{alloc} - i) \}$
  • $X = \& Y$: $\llbracket v \rrbracket = \text{JOIN}(v) \downarrow X \cup \{ (X, Y) \}$
  • $X = Y$: $\llbracket v \rrbracket = \text{JOIN}(v) \downarrow X \cup \{ (X, t) \mid (Y, t) \in \text{JOIN}(v) \}$
  • $X = * Y$: $\llbracket v \rrbracket = \text{JOIN}(v) \downarrow X \cup \{ (X, t) \mid (Y, s) \in \sigma, (s, t) \in \text{JOIN}(v) \}$
  • $* X = Y$: $\llbracket v \rrbracket = \text{JOIN}(v) \cup \{ (s, t) \mid (X, s) \in \text{JOIN}(v), (Y, t) \in \text{JOIN}(v) \}$
  • $X = \text{null}$: $\llbracket v \rrbracket = \text{JOIN}(v) \downarrow X$

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

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

note: weak update!

• For all other CFG nodes:
  • $\llbracket v \rrbracket = \text{JOIN}(v)$
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:  
    $$pt(x) = \{ z \}$$
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:
  \[ P(\text{Cells} \times \text{Cells}) \times (\text{Cell} \rightarrow \text{Count}) \]
  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

\[ \text{Count} = \begin{cases} 0 & \text{if 0} \\ 1 & \text{if 1} \\ >1 & \text{if >1} \end{cases} \]
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;
}