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
• Steensgaard’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?)

\[
\begin{align*}
E & \rightarrow \&X \\
& \mid \text{alloc } E \\
& \mid \ast E \\
& \mid \text{null} \\
& \mid \ldots
\end{align*}
\]

\[
\begin{align*}
S & \rightarrow \ast X = E; \\
& \mid \ldots
\end{align*}
\]

\[
E \rightarrow E( E, ..., E )
\]

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

• 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
• 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
Agenda

• Introduction to points-to analysis
• Andersen’s analysis
• Steensgaard’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. $⟦·⟧: \text{Cells} \to 2^{\text{Cells}}$

• Generate constraints:
  • $X = \text{alloc } P$: $\text{alloc}_i \in [X]$
  • $X = &Y$: $Y \in [X]$
  • $X = Y$: $[Y] \subseteq [X]$
  • $X = \ast Y$: $\alpha \in [Y] \Rightarrow [\alpha] \subseteq [X]$ for each $\alpha \in \text{Cells}$
  • $\ast X = Y$: $\alpha \in [X] \Rightarrow [Y] \subseteq [\alpha]$ for each $\alpha \in \text{Cells}$
  • $X = \text{null}$: (no constraints)
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 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;
```
Applying Andersen

- Generated constraints:

\[
\begin{align*}
\text{alloc-1} & \in [p] \\
[y] & \subseteq [x] \\
[z] & \subseteq [x] \\
\alpha & \in [p] \Rightarrow [z] \subseteq [\alpha] \\
[q] & \subseteq [p] \\
y & \in [q] \\
\alpha & \in [p] \Rightarrow [\alpha] \subseteq [x] \\
z & \in [p]
\end{align*}
\]

- Smallest solution:

\[
\begin{align*}
pt(p) & = \{ \text{alloc-1}, y, z \} \\
pt(q) & = \{ y \}
\end{align*}
\]
Agenda

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

• View assignments as being bidirectional

• Generate constraints:
  • $X = \text{alloc } P$: $\text{alloc} - i \in \llbracket X \rrbracket$
  • $X = &Y$: $Y \in \llbracket X \rrbracket$
  • $X = Y$: $\llbracket X \rrbracket = \llbracket Y \rrbracket$
  • $X = \ast Y$: $\alpha \in \llbracket Y \rrbracket \Rightarrow \llbracket \alpha \rrbracket = \llbracket X \rrbracket$
  • $\ast X = Y$: $\alpha \in \llbracket X \rrbracket \Rightarrow \llbracket Y \rrbracket = \llbracket \alpha \rrbracket$

• Extra constraints:
  $$t_1, t_2 \in \llbracket t \rrbracket \Rightarrow \llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket \text{ and } \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket \neq \emptyset \Rightarrow \llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket$$
  (whenever a cell may point to two cells, they are effectively merged into one)

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

- Reformulate as term unification
- Generate constraints:
  - \( X = \text{alloc} \ P : \ \ [X] = \&[\text{alloc}-i] \)
  - \( X = \&Y : \ \ [X] = \&[Y] \)
  - \( X = Y : \ \ [X] = [Y] \)
  - \( X = \*Y : \ \ [Y] = \&\alpha \ \land \ \ [X] = \alpha \) where \( \alpha \) is fresh
  - \( \*X = Y : \ \ [X] = \&\alpha \ \land \ \ [Y] = \alpha \) where \( \alpha \) is fresh
- Terms:
  - term variables, e.g. \([X] , \&[\text{alloc}]-i] , \alpha \) (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 cells
- Fits with our unification solver! (union-find...)
- The points-to map is defined as \( \text{pt}(X) = \{ c \in \text{Cells} \mid [X] = \&[c] \} \)
- Note that there is only one kind of term constructor, so unification never fails.

16
Applying Steensgaard

- Generated constraints:

  \[
  \begin{align*}
  \text{alloc}-1 & \in [p] \\
  [y] & = [x] \\
  [z] & = [x] \\
  \alpha & \in [p] \Rightarrow [z] = [\alpha] \\
  [q] & = [p] \\
  y & \in [q] \\
  \alpha & \in [p] \Rightarrow [\alpha] = [x] \\
  z & \in [p] \\
  \text{+ the extra constraints}
  \end{align*}
  \]

- Smallest solution:

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

Andersen:

\[
\begin{align*}
  a1 &= &b1; \\
  b1 &= &c1; \\
  c1 &= &d1; \\
  a2 &= &b2; \\
  b2 &= &c2; \\
  c2 &= &d2; \\
  b1 &= &c2;
\end{align*}
\]

Steensgaard:

\[
\begin{align*}
  a1 &= &b1; \\
  b1 &= &c1; \\
  c1 &= &d1; \\
  a2 &= &b2; \\
  b2 &= &c2; \\
  c2 &= &d2; \\
  b1 &= &c2;
\end{align*}
\]
Recall our type analysis...

- Focusing on pointers...
- Constraints:
  - $X = \text{alloc } P$: $[X] = &[P]$  
  - $X = &Y$: $[X] = &[Y]$  
  - $X = Y$: $[X] = [Y]$  
  - $X = *Y$: $&[X] = [Y]$  
  - $*X = Y$: $[X] = &[Y]$  
- 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
- Steensgaard’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 \textit{simultaneously}!
Function call normalization

• Assume that all function calls are of the form

\[ x = y(a_1, ..., a_n) \]

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

• Assume that all return statements are of the form

\[ \text{return } z; \]

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

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

• For the function call
  \[ x = y(a_1, \ldots, a_n) \]
  and every occurrence of
  \[ f(x_1, \ldots, x_n) \{ \ldots \text{return } z; \} \]
  add these constraints:

  \[
  f \in [f] \\
  f \in [y] \Rightarrow ([a_i] \subseteq [x_i] \text{ for } i=1,\ldots,n \land [z] \subseteq [x])
  \]

• (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
  \[ f(x_1, \ldots, x_n) \{ \ldots \text{ return } z; \} \]
  add these constraints:
  \[
  f \in \llbracket f \rrbracket \\
  f \in \llbracket y \rrbracket \Rightarrow \left( [a_i] = [x_i] \text{ for } i=1,\ldots,n \land [z] = [x] \right)
  \]

• (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 $Cells \rightarrow 2^{Cells}$ to $Contexts \rightarrow Cells \rightarrow 2^{Cells}$
  (or equivalently: $Cells \times Contexts \rightarrow 2^{Cells}$)
  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

- 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

```javascript
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
- Steensgaard’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 $Null$:

$\begin{array}{c}
? \\
| \\
NN
\end{array}$

where NN represents “definitely not null” 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:

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

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

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

  where $P$ is null or an integer constant

• 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:

\[
^*X = Y: \quad \llbracket v \rrbracket = store(JOIN(v), X, Y)
\]

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 $X = \ast Y$ 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:

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

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

```c
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

```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;
```

The points-to set for `x` contains *multiple abstract cells*
Null analysis constraints

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

• 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: \( \sigma[c \mapsto \emph{new-value}] \)
  – 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: \( \sigma[c \mapsto \sigma(c) \sqcup \emph{new-value}] \)
  – 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

```c
function f(b_1, ..., b_n)

x = f(E_1, ..., E_n);

result = E;
```
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:

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

\[\llbracket p=\text{alloc null} \rrbracket = \bot[p \mapsto \text{NN}, \text{alloc}-1 \mapsto ?] \]
\[\llbracket q=&p \rrbracket = \llbracket p=\text{alloc null} \rrbracket[q \mapsto \text{NN}] \]
\[\llbracket n=\text{null} \rrbracket = \llbracket q=&p \rrbracket[n \mapsto ?] \]
\[\llbracket *q=n \rrbracket = \llbracket n=\text{null} \rrbracket[p \mapsto [n=\text{null}](p) \sqcup [n=\text{null}](n)] \]
\[\llbracket *p=n \rrbracket = \llbracket *q=n \rrbracket[\text{alloc}-1 \mapsto [\*q=n](\text{alloc}-1) \sqcup [\*q=n](n)] \]
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 \(\text{alloc-i}\) locations
• Introduction to points-to analysis
• Andersen’s analysis
• Steensgaard’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 $\llbracket v \rrbracket$ describing the state after $v$

- Intraprocedural analysis (i.e. ignore function calls)
Constraints

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

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

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

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

- \( \text{assign}(\sigma, X, Y) = \sigma \downarrow X \cup \{ (X, t) \mid (Y, t) \in \sigma \} \)

- \( \text{load}(\sigma, X, Y) = \sigma \downarrow X \cup \{ (X, t) \mid (Y, s) \in \sigma, (s, t) \in \sigma \} \)

- \( \text{store}(\sigma, X, Y) = \sigma \cup \{ (s, t) \mid (X, s) \in \sigma, (Y, t) \in \sigma \} \)
  - note: weak update!
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:

```
<table>
<thead>
<tr>
<th>p</th>
<th>alloc-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>alloc-1</td>
</tr>
<tr>
<td>q</td>
<td>alloc-4</td>
</tr>
<tr>
<td>y</td>
<td>alloc-2</td>
</tr>
</tbody>
</table>
```

• 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:
  \[ 2^{Cell \times Cell} \times (Cell \rightarrow Count) \]
  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
Constraints

- $X = \text{alloc } P$: ...
- $*X = Y$: ...
- ...
- ...
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

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