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?)

\[ E \rightarrow &X \]
\[ \mid \text{alloc } E \]
\[ \mid *E \]
\[ \mid \text{null} \]
\[ \mid \ldots \]

\[ S \rightarrow *X = E; \]
\[ \mid \ldots \]

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

\[
\cdots
\begin{align*}
&*x = 42; \\
&*y = -87; \\
&z = *x; \\
&// \ is \ z \ 42 \ or \ -87?
\end{align*}
\]
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
- Steensgaards’s analysis
- Interprocedural points-to analysis
- Null pointer analysis
- Flow-sensitive points-to analysis
Andersen’s analysis (1/2)

• For every cell $c$, introduce a constraint variable $⟦c⟧$ ranging over sets of cells, i.e. $⟦·⟧ : Cells → 2^{Cells}$

• Generate constraints:
  • $X = alloc P$: $alloc - i ∈ ⟦X⟧$  
  • $X &= Y$: $Y ∈ ⟦X⟧$  
  • $X = Y$: $⟦Y⟧ ⊆ ⟦X⟧$  
  • $X = ∗Y$: $c ∈ ⟦Y⟧ ⇒ ⟦c⟧ ⊆ ⟦X⟧$ for each $c ∈ Cells$  
  • $∗X = Y$: $c ∈ ⟦X⟧ ⇒ ⟦Y⟧ ⊆ ⟦c⟧$ for each $c ∈ Cells$  
  • $X = null$: (no constraints)
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;
```

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

- Generated constraints:

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

- 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:
  • $X = \text{alloc}\ P$: $\text{alloc}\ -i \in [X]$
  • $X = &Y$: $Y \in [X]$
  • $X = Y$: $[X] = [Y]$
  • $X = *Y$: $c \in [Y] \Rightarrow [c] = [X]$ for each $c \in \text{Cells}$
  • $*X = Y$: $c \in [X] \Rightarrow [Y] = [c]$ for each $c \in \text{Cells}$

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

• Reformulate as term unification

• Generate constraints:
  • $X = \text{alloc} \ P$: $\llbracket X \rrbracket = \&\llbracket \text{alloc-}i \rrbracket$
  • $X = \&Y$: $\llbracket X \rrbracket = \&\llbracket Y \rrbracket$
  • $X = Y$: $\llbracket X \rrbracket = \llbracket Y \rrbracket$
  • $X = \ast Y$: $\llbracket Y \rrbracket = \&\alpha \land \llbracket X \rrbracket = \alpha$ where $\alpha$ is fresh
  • $\ast X = Y$: $\llbracket X \rrbracket = \&\alpha \land \llbracket Y \rrbracket = \alpha$ 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 $\& t$ (representing pointers)
  – $\llbracket X \rrbracket$ 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 $\text{pt}(X) = \{ c \in Cells \mid \llbracket X \rrbracket = \&\llbracket c \rrbracket \}$

• Note that there is only one kind of term constructor, so unification never fails.
Applying Steensgaard

- Generated constraints:

  \[
  \text{alloc-1} \in \llbracket p \rrbracket \\
  \llbracket y \rrbracket = \llbracket x \rrbracket \\
  \llbracket z \rrbracket = \llbracket x \rrbracket \\
  \alpha \in \llbracket p \rrbracket \Rightarrow \llbracket z \rrbracket = \llbracket \alpha \rrbracket \\
  \llbracket q \rrbracket = \llbracket p \rrbracket \\
  y \in \llbracket q \rrbracket \\
  \alpha \in \llbracket p \rrbracket \Rightarrow \llbracket \alpha \rrbracket = \llbracket x \rrbracket \\
  z \in \llbracket p \rrbracket \\
  + \text{the extra constraints}
  \]

- 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;
b1 = &c2;

Steensgaard:

a1 → b1 → c1 → d1
a2 → b2 → c2 → d2
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:
  $\&t_1 = \&t_2 \implies t_1 = t_2$

- 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

• 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

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

• \( y \) may be a variable whose value is a function pointer
• Assume that all return statements are of the form

\[
\text{return } z; 
\]

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

• Include all function names in 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:

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

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

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

• (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
- We can also track the set of reachable contexts for each function (like the use of lifted lattices earlier): $Contexts \rightarrow \text{lift}(Cells \rightarrow 2^{Cells})$
- Does this still fit into the cubic 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

```c
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 \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 $Null$:

\[
\begin{array}{c}
? \\
\mid \\
NN
\end{array}
\]

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:

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

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

• For operations involving pointers:
  • $X = \text{alloc } P$: \(\llbracket v \rrbracket = ???\) 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:
  • \(\llbracket v \rrbracket = JOIN(v)\)
Null analysis constraints

• For a heap store operation $\ast X = Y$ 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:

\[
\ast X = Y: \quad \llbracket v \rrbracket = store(\text{JOIN}(v), X, Y)
\]

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

\[
X = *Y: \quad \llbracket v \rrbracket = \text{load}(\text{JOIN}(v), X, Y)
\]

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

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

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

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

```
\texttt{\textbf{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

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

Andersen generates:

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

\[ [p=\text{alloc null}] = \bot[p \leftrightarrow \text{NN}, \text{alloc-1} \leftrightarrow ?] \]
\[ [q=&p] = [p=\text{alloc null}][q \leftrightarrow \text{NN}] \]
\[ [n=null] = [q=&p][n \leftrightarrow?] \]
\[ [*q=n] = [n=null][p \leftrightarrow [n=null](p) \cup [n=null](n)] \]
\[ [*p=n] = [*q=n][\text{alloc-1} \leftrightarrow [*q=n](\text{alloc-1}) \cup [*q=n](n)] \]
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 \(\text{alloc}-i\) cells
Agenda

• 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 $2^{\text{Cells} \times \text{Cells}}$ ordered under subset inclusion
  (or alternatively, $\text{Cells} \rightarrow 2^{\text{Cells}}$)

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

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

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

note: weak update!

• For all other CFG nodes:
  • $\llbracket v \rrbracket = JOIN(v)$
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: $pt(x) = \{ z \}$

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

\[ Count = 0 \]
\[ 1 \quad >1 \]
Constraints

- $x = \text{alloc } P$: ...
- $*x = y$: ...
- ...

54
Better results

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

• Thus, alloc-2 nodes form a self-loop
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;
} 
```