Pointer Analysis
Lecture 2

G. Ramalingam
Microsoft Research, India
&
K. V. Raghavan
Correctness and precision of Algorithm A
Enter: The French Recipe (Abstract Interpretation)

Concrete (Collecting) Domain
• A semi-lattice \((C, \cup)\)
• Transfer Functions
  • For every statement \(st\), \(CS'[st]: C \rightarrow C\)

Abstract Domain
• A semi-lattice \((A, \sqcup)\)
• Transfer Functions
  • For every statement \(st\), \(AS[st]: A \rightarrow A\)

\(C \equiv 2^{\text{DataState}}\)
\(A \equiv \text{AbsDataState}\)
Points-To Analysis (Abstract Interpretation)

\[ \alpha(Y) = \{ p. \{ x | \text{exists } s \in Y. s(p) = x \} \} \]

\[ \gamma(a) = \{ (p \rightarrow v, q \rightarrow w) | v \in a(p), w \in a(q) \} \]

(assuming \( p, q \) are the pointer variables in the program)
Approximating Transformers: Correctness Criterion

It can be shown that for any statement $st$, and for any $a_1 \in A$

$$AS[st](a_1) \geq \alpha (CS[st](\gamma(a_1)))$$
Transfer function illustration

\[
\begin{align*}
\text{CS'} & \quad \ast y = \ast b; \\
\text{AS} & \quad \ast y = \ast b;
\end{align*}
\]
Transfer function illustration
Transfer function illustration

\[ x: \&y \quad y: \&x \quad z: \&a \]
\[ x: \&y \quad y: \&z \quad z: \&a \]
\[ x: \&y \quad y: \&c \quad z: \&a \]

\[ x: \{\&y\} \quad y: \{\&x, \&z\} \quad z: \{\&a\} \]

\[ x: \{\&y, \&b, \&c\} \quad y: \{\&x, \&z\} \quad z: \{\&a, \&b, \&c\} \]

\[ x: \{\&y, \&c\} \quad y: \{\&x, \&z\} \quad z: \{\&a, \&c\} \]
Is The Precise Solution Computable?

• Claim: The set RS(u) of reachable concrete states (for our language) is precisely computable.
  – (However, Algorithm A is imprecise)

• Note: This is true for any collecting semantics with a finite state space.
• This is true only for restricted language!
Precise Points-To Analysis: Computational Complexity

• What’s the complexity of the least-fixed point computation using the collecting semantics?

• The worst-case complexity of computing reachable states is exponential in the number of variables.
  – Can we do better?

• Theorem: Computing precise may-point-to is PSPACE-hard even if we have only two-level pointers.
Precise Points-To Analysis: Caveats

• Theorem: Precise may-alias analysis is undecidable in the presence of dynamic memory allocation.
  – Add "x = new/malloc ()" to language
  – State-space becomes infinite

• Digression: Integer variables + conditional-branching involving integer variables also makes any precise analysis undecidable.
Dynamic Memory Allocation

- $s: x = \text{new}() / \text{malloc}()$
- Assume, for now, that allocated object stores one pointer
  - $s: x = \text{malloc}()$ \text{sizeof(void*)}
- Introduce a pseudo-variable $V_s$ to represent objects allocated at statement $s$, and use previous algorithm
  - treat $s$ as if it were “$x = &V_s$”
  - also track possible values of $V_s$
- allocation-site based approach
Dynamic Memory Allocation: Example

x = new;
y = x;
*y = &b;
*y = &a;
Example illustrating need for weak updates on summary objects

- Key aspect: $V_s$ represents a set of objects (locations), not a single object
  - referred to as a summary object (node)
  - if $x \rightarrow \{V_s\}$, to be safe "$*x = ..$" still needs weak update!

```c
do {
  s:  x = new;

  x -> \{V_s\} is correct here
  *x = &a;
  V_s ->\{a\} is correct here
}

x -> \{V_s\} is correct here
*x = &b;
V_s ->\{a,b\} is correct answer here!
```
Other Aspects

• Field sensitivity
  – Concrete state maps each variable to a variable or address of an object or null, and for each object o and each field f of o, maps \((o,f)\) to the address of some other object (or to null)
  
  – Let PseudoVar be the set of pseudo-variables, let \(\text{Var}' = \text{Var} \cup \text{PseudoVar} \cup \{\text{null}\}\)
  
  – Each AbsDataState maps each variable to a subset of \(\text{Var}'\). Also, for each \(V_s\), and for each field f in the object allocated at site s, maps \((V_s,f)\) to a subset of \(\text{Var}'\).

• Context-sensitivity can be achieved using standard techniques

• Indirect (virtual) function call sites need to be resolved to candidate functions using points-to analysis. And points-to analysis needs calls to be resolved! Therefore, the two have to happen hand in hand.
Andersen’s Analysis

• A flow-insensitive analysis
  – computes a single points-to solution, which over-approximates points-to solutions at all program points
  – ignores control-flow – treats program as a set of statements
  – equivalent to collapsing the given program to have a single program point, and then applying Algorithm A on it.
Andersen's Analysis

If program has statements \( s_1, s_2, \ldots, s_n \), then create collapsed CFG as follows:

After algorithm terminates, final solution at the single program point over-approximates result computed by flow-sensitive analysis at any point.
Example:
Andersen’s Analysis

\[ x = \&a; \]
\[ *x = \&w; \]
\[ y = x; \]
\[ x = \&b; \]
\[ *x = \&t; \]
\[ z = x; \]
Notes about Andersen’s Analysis

• Strong updates never happen in Andersen’s analysis!
  – If $x \rightarrow \{y\}$ and $y \rightarrow \{w\}$ before we process statement "$*x = \&z$", then even if transfer function returns $y \rightarrow \{z\}$, due to subsequent join, $y$ will point to $\{w,z\}$ after this step.

• Flow-insensitive style can be adopted for any analysis, not just for pointer analysis
Why Flow-Insensitive Analysis?

• Reduced space requirements
  – a single points-to solution

• Reduced time complexity
  – no copying of points-to facts
    • individual updates more efficient
  – a cubic-time algorithm

• Scales to millions of lines of code
  – most popular points-to analysis
Andersen’s Analysis: An alternative formulation

1. Introduce a constraint variable \( PT_x \) for each program variable \( x \)
2. Create a constraint from each assignment statement, as follows:
   - \( x = y \): \( PT_x \subseteq PT_y \)
   - \( *x = y \): \( PT_x \subseteq PT_y \), forall variables \( v \) in \( PT_x \)
   - \( x = &y \): \( PT_x \subseteq \{y\} \)
   - \( x = ^{*}y \): \( PT_x \subseteq PT_v \), forall variables \( v \) in \( PT_y \)
3. Find least solution to set of all constraints that were generated above. (A solution is a mapping from constraint variables to sets of program variables.) Emit this least solution as the final solution.
   - Note: Solution \( s_1 \) dominates Solution \( s_2 \) if for each program variable \( v \), \( s_2(PT_v) \subseteq s_1(PT_v) \)

Note: This approach computes exact same result as previous approach that collapses program and then uses Algorithm A.
May-Point-To Analyses

Ideal-May-Point-To

more efficient / less precise

Algorithm A

more efficient / less precise

Andersen’s
Andersen’s Analysis:
Further Optimizations and Extensions

- Fahndrich et al., Partial online cycle elimination in inclusion constraint graphs, PLDI 1998.
- Rountev and Chandra, Offline variable substitution for scaling points-to analysis, 2000.
- M. Hind, Pointer analysis: Haven’t we solved this problem yet?, PASTE 2001.
- Hardekopf and Lin, Exploiting pointer and location equivalence to optimize pointer analysis, SAS 2007.
Context-Sensitivity Etc.

Applications

• Compiler optimizations

• Verification & Bug Finding
  – use in preliminary phases
  – use in verification itself