CS711 Advanced Programming Languages

Pointer Analysis
Overview and Flow-Sensitive Analysis

Radu Rugina

8 Sep 2005
Pointer Analysis

• Informally: determine where pointers (or references) in the program may point to.

• Significant amount of research in past 15 years
  – … still going

• It is a fundamental problem in program analysis
  – Required by virtually all other analyses, optimizations, program understanding tools, bug-finding tools, etc.
  – Worst-case assumptions are too conservative
    • Especially for type-unsafe languages (e.g., C)
Points-To vs. Alias Analysis

• Points-to analysis: Compute the set of memory locations that each pointer may point to.
  – Hence, a may analysis
  – E.g., pt(x)={z,t}, pt(t)={u}, pt(y)={z}
  – Essentially, a points-to graph

\[
x \rightarrow t \rightarrow u
\]
\[
y \rightarrow z
\]

• (Pointer) alias analysis computes alias pairs
  – E.g. (*x,z), (*x,t), (*t,u), (**x,u), (*y,z)
  – Points-to graphs = a compact representation of alias pairs
  – Used in older analyses, e.g., [LR92]
Classifying Points-To Analyses

• **Flow-sensitivity**
  – Flow analyses
    • compute a points-to graph at each program point
  – Flow-insensitive analyses
    • Assignments can execute in any order, any number of times
    • Obviously models program execution
    • A points-to graph for the entire program
    • Two main kinds:
      – Steensgaard, a.k.a. unification-based
      – Andersen, a.k.a. inclusion-based

• **Context-sensitivity**
  – Distinguish the behavior of a function based on its calling context
Classifying Points-To Analyses

- Context Insensitive
  - [And94]
  - [Das00]
  - [SH98]
  - [Ste96]
  - [SGSB05]

- Context Sensitive
  - [RH98]
  - [FRD00]
  - [DLFR01]
  - [WL04]
  - [WL95]
  - [And94]
  - [BLQ+03]
  - [Ruf95]

C analyses (yellow)
Java analyses (green)
Points-To Analysis

• “compute set of locations where each pointer may point to”

• Ambiguities:
  – What are locations?
  – What about heap-allocated pointers?
  – What about aggregate structures: records, arrays, etc?
  – What about different instances of the same variable?

• We’re missing a notion of memory abstraction
Memory Model

• An abstraction of the memory
  – Map concrete locations to “abstract locations/nodes”
    • One abstract node may represent one or more concrete memory locations
    • Approximate unbounded concrete program state using a finite abstraction
  – Analysis clients need to know about this abstraction
  – Difficult to compare (results for) different abstractions
Heap Abstraction

- **Heap abstraction**
  - Typically: one abstract node for each allocation site
  - Think: “one global variable per malloc”

\[
12: x = \text{malloc}(\ldots)
\]

- **Alternatives:**
  - Less precise: one node for the entire heap
  - More precise: different nodes for locations allocated in different calling contexts
    - Aka “context-sensitive heap abstraction”
    - Think malloc wrappers

- **Model is imprecise for recursive structures**
  - Shape analysis is significantly more precise here
Records and Structures

• **Option A:** Model each field of each struct variable
  – A.k.a. “field-sensitive”. Think “x.f”
    
    ```
    struct { int a, b; } x, y;  
    x.a  x.b  
    y.a  y.b  
    ```

• **Option B:** Merge all fields of each struct variable
  – A.k.a. “field-independent”, “field-insensitive”. Think “x.*”
    
    ```
    struct { int a, b; } x, y;  
    x.*  y.*  
    ```

• **Option C:** Model each field of all struct variables
  – A.k.a. “field-based”. Think “*.f”
    
    ```
    struct { int a, b; } x, y;  
    *.a  *.b  
    ```
Unions

• Unions are type-unsafe
  – Sound approach: merge all fields
    • As in “field-independent” (B)

    union { int a; char b; } x; \rightarrow \textcolor{blue}{x.*}

  – Unsound approach: assume fields don’t interfere
    • As in “field-sensitive” (A)

    union { int a; char b; } x; \rightarrow \textcolor{blue}{x.a} \quad \textcolor{blue}{x.b}
Arrays

- Merge all array elements together

  ```
  int a[10];
  a[*]
  ```

- Or use a separate abstraction for the first element

  ```
  int a[10];
  a[0]  a[1..10]
  ```
Nested Arrays and Structures

- Recurse through nested structure
  - Merge array elements
  - Separate all structure fields
    - even if structure is nested in an array

```
struct { int a[3], b; } x[3];
```

```
x[0] x[1] x[2]
```
The Flow Analysis

- Program assignments:
  - address-of      copy        load        store
  - \( x = \&y \)      \( x = y \)      \( x = *y \)      \( *x = y \)

- Dataflow information = points-to graphs
  - Use \( pt(x) = \) points-to set of \( x \)

- Merge operator = set union

- Transfer functions
  - \( x = \&y \) : \( pt'(x) = \{ y \} \)
  - \( x = y \) : \( pt'(x) = pt(y) \)
  - \( x = *y \) : \( pt'(x) = U \) pt(z), for all \( z \in pt(y) \)
  - \( *x = y \) : \( pt'(z) U= pt(y) \), for all \( z \in pt(x) \)
Strong vs. Weak Updates

- "strong updates" = update value
- "weak updates" = accumulate value

- Strong updates = more precise
- Weak updates if can’t tell which concrete location is written
  - \( *x = y \)
  - \( x[i] = y \)

- Strong updates = key difference between flow-sensitive and flow-insensitive analyses
Inter-Procedural Analysis [EGH’94]

• Analyze callee for each function call
  – “map” the points-to information in the caller
  – Analyze callee with mapped information
  – “unmap” result and return to caller

• Mapping process:
  – Use “invisible variables” to model variables that are not in the current scope, but accessible through pointers
  – Store mapping information, use it during unmap

```c
foo() { int a, *b = &a;
    bar(&b); }
bar(int** p) { ... }
```

Call site graph: b ← a
Mapped graph: p ← p_1 ← p_2
Mapping info: (b,p_1) (a,p_2)
Invocation Graph

- Use an “invocation graph” for context-sensitivity
  - Unroll call-graph, turn it into a tree

```
main() { g(); g(); }
g()    { f(); }
f()    { ... }
```
Invocation Graph

• Use an “invocation graph” for context-sensitivity
  – For recursion:
    • Use two nodes: “approximate” and “recursive”
    • Perform a fixed-point computation along the back edge
    • Use summaries for each node

```
main() { f(); }
f()    { if (...) g(); }
g()    { f(); }
```
Function Pointers

- **Indirect calls**: a “chicken-and-egg” problem
  - Need points-to information to resolve such calls
  - Need to resolve the calls to compute the points-to info
  - Solution: compute both at the same time
  - Once a call is resolved: analyze each callee, merge the results
Evaluating an Analysis

• What is the right metric?
  – An ongoing debate
  – **Option 1:** size of points-to sets
    • At loads and stores, at indirect calls
    • Difficult to compare analyses that use different abstractions

  – **Option 2:** evaluate effect on analysis clients
    • E.g, how many virtual calls are disambiguated? Or how many false data dependencies are being removed?
    • How much faster do programs run because of a better points-to analysis?
    • How is the false positive ratio improved in a bug-finding tool?
Experiments [EGH’94]

• Programs ranging from 0.1 K to 2.2 K LOC
• Small points-to set sizes at indirect accesses (avg. 1.13)
• Many indirect with one single target (28%)
  – But only 19% where the target is a program variable
• Invocation Graph statistics:
  – Average ratio IG size / call-sites = 1.45 (up to 2.5)
  – Ratio IG size / procedures larger (up to 21)
  – In theory, IG size is exponential
Memoization [WL’95]

• [Wilson,Lam,PLDI’95] “Efficient Context-Sensitive Pointer Analysis for C Programs”
  – Always use procedure summaries (not just for recursion)
    • Called “partial transfer functions” (PTFs)
  – Do not build an Invocation Graph
  – Build “invisible variables” lazily
  – Memory abstraction using triples (b, f, s), with base b, offset f, and stride s
  – Ratio PTFs / procedures : between 1.00 and 1.39
  – Report a program with 37 procedures that generates an invocation graph with more than 700,000 nodes