Lecture 14

Pointer Analysis

• Basics
• Design Options
• Pointer Analysis Algorithms
• Pointer Analysis Using BDDs
• Probabilistic Pointer Analysis

[ALSU 12.4, 12.6-12.7]
Pros and Cons of Pointers

• Many procedural languages have pointers
  – e.g., C or C++: int *p = &x;

• Pointers are powerful and convenient
  – can build arbitrary data structures

• Pointers can also hinder compiler optimization
  – hard to know where pointers are pointing
  – must be conservative in their presence

• Has inspired much research
  – analyses to decide where pointers are pointing
  – many options and trade-offs
  – open problem: a scalable accurate analysis
I. Pointer Analysis Basics: Aliases

• Two variables are aliases if:
  – they reference the same memory location
• More useful:
  – prove variables reference different locations

What are the Alias sets?

```c
int x, y;
int *p = &x;
int *q = &y;
int *r = p;
int **s = &q;
```
The Pointer Alias Analysis Problem

• Decide for every pair of pointers at every program point:
  – do they point to the same memory location?
• A difficult problem
  – shown to be undecidable by Landi, 1992
• Correctness:
  – report all pairs of pointers which do/may alias
• Ambiguous:
  – two pointers which may or may not alias
• Accuracy/Precision:
  – how few pairs of pointers are reported while remaining correct
  – i.e., reduce ambiguity to improve accuracy
Many Uses of Pointer Analysis

• Basic compiler optimizations
  – register allocation, CSE, dead code elimination, live variables, instruction scheduling, loop invariant code motion, redundant load/store elimination

• Parallelization
  – instruction-level parallelism
  – thread-level parallelism

• Behavioral synthesis
  – automatically converting C-code into gates

• Error detection and program understanding
  – memory leaks, wild pointers, security holes
Challenges for Pointer Analysis

- **Complexity**: huge in space and time
  - compare every pointer with every other pointer
  - at every program point
  - potentially considering all program paths to that point
- **Scalability vs accuracy trade-off**
  - different analyses motivated for different purposes
  - many useful algorithms (adds to confusion)
- **Coding corner cases**
  - pointer arithmetic (*p++), casting, function pointers, long-jumps
- **Whole program?**
  - most analysis algorithms require the entire program
  - library code? optimizing at link-time only?
II. Pointer Analysis: Design Options

• Representation
• Heap modeling
• Aggregate modeling (e.g., arrays, structs)
• Flow sensitivity
• Context sensitivity
Representation

• Track pointer aliases
  - <*a, b>, <*a, e>, <b, e>, <**a, c>, <**a, d>, ...
  - More precise, less efficient

• Track points-to information
  - <a, b>, <b, c>, <b, d>, <e, c>, <e, d>
  - Less precise, more efficient. Why?

```
a = &b;
b = &c;
b = &d;
e = b;
```
Heap Modeling Options

- Heap merged
  - i.e. “no heap modeling”
- Allocation site (any call to malloc/calloc)
  - Consider each to be a unique location
  - Doesn’t differentiate between multiple objects allocated by the same allocation site
- Shape analysis
  - Recognize linked lists, trees, DAGs, etc.
Aggregate Modeling Options

**Arrays**

- Elements are treated as individual locations
- Treat entire array as a single location
- Treat first element separate from others

**Structures**

- Elements are treated as individual locations (“field sensitive”)
- Treat entire structure as a single location

What are the trade-offs?
Flow Sensitivity Options

- **Flow insensitive**
  - The order of statements doesn’t matter
    - Result of analysis is the same regardless of statement order
    - Uses a single global state to store results as they are computed
    - Fast, but not very accurate

- **Flow sensitive**
  - The order of the statements matter
  - Need a control flow graph
  - Must store results for each program point
  - Improves accuracy

- **Path sensitive**
  - Each path in a control flow graph is considered
  - If-then-else implies mutually exclusive paths
Flow Sensitivity Example

*(assuming allocation-site heap modeling)*

Flow Insensitive

\[ S7: a = b; \]

Flow Sensitive

\[ S7: a = \text{malloc}(...); \]

Path Sensitive

\[ S7: \ldots = *a; \]
Context Sensitivity Options

• Context insensitive/sensitive (interprocedural analysis)
  – whether to consider different calling contexts
  – e.g., what are the possibilities for $p$ at S6?

```c
int a, b, *p;
int main()
{
  S1: f();
  S2: p = &a;
  S3: g();
}

int f()
{
  S4: p = &b;
  S5: g();
}

int g()
{
  S6: ... = *p;
}
```

Context Insensitive:

Context Sensitive:
### Pointer Alias Analysis Algorithms

**Extensive Literature:**

- “Context-sensitive interprocedural points-to analysis in the presence of function pointers”, Emami et al., PLDI 1994
- “Points-to analysis in almost linear time”, Steensgaard, POPL 1996
- “Which pointer analysis should I use?”, Hind et al., ISSTA 2000
- “Pointer analysis: haven't we solved this problem yet?”, Hind, PASTE 2001
- ...
- “Introspective analysis: context-sensitivity, across the board”, Smaragdakis et al., PLDI 2014
- “Sparse flow-sensitive pointer analysis for multithreaded programs”, Sui et al., CGO 2016
- “Symbolic range analysis of pointers”, Paisante et al., CGO 2016
Address Taken

• Basic, fast, ultra-conservative algorithm
  – flow-insensitive, context-insensitive
  – often used in production compilers
• Algorithm:
  – Generate the set of all variables whose addresses are assigned to another variable.
  – Assume that any pointer can potentially point to any variable in that set.
• Complexity: O(n) - linear in size of program
• Accuracy: very imprecise
T *p, *q, *r;

int main() {
    S1: p = alloc(T);
    f();
    g(&p);
    S4: p = alloc(T);
    S5: ... = *p;
}

void f() {
    S6: q = alloc(T);
    g(&q);
    S8: r = alloc(T);
}

g(T **fp) {
    T local;
    if(...) 
    s9: p = &local;
}
Andersen’s Algorithm

- **Flow-insensitive, context-insensitive, iterative**
- **Representation:**
  - one points-to graph for entire program
  - each node represents exactly one location
- For each statement, build the points-to graph:

<table>
<thead>
<tr>
<th>y = &amp;x</th>
<th>y points-to x</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = x</td>
<td>if x points-to w then y points-to w</td>
</tr>
<tr>
<td>*y = x</td>
<td>if y points-to z and x points-to w then z points-to w</td>
</tr>
<tr>
<td>y = *x</td>
<td>if x points-to z and z points-to w then y points-to w</td>
</tr>
</tbody>
</table>

- Iterate until graph no longer changes
- Worst case **complexity**: \( O(n^3) \), where \( n \) = program size
Andersen Example

T *p, *q, *r;

int main() {
    p = alloc(T);
    f();
    g(&p);
    p = alloc(T);
    ... = *p;
}

void f() {
    q = alloc(T);
    g(&q);
    r = alloc(T);
}

g(T **fp) {
    T local;
    if(...) 
    p = &local;
}
Steensgaard’s Algorithm

- **Flow-insensitive, context-insensitive**
- **Representation:**
  - A compact points-to graph for entire program
    - Each node can represent *multiple locations*
    - But can only point to one other node
      - I.e. every node has a fan-out of 1 or 0
- **Union-find** data structure implements fan-out
  - “Unioning” while finding eliminates need to iterate
- **Worst case complexity:** $O(n)$
- **Precision:** Less precise than Andersen’s
T *p, *q, *r;

int main() {
    S1: p = alloc(T);
    f();
    g(&p);
    S4: p = alloc(T);
    S5: ... = *p;
}

void f() {
    S6: q = alloc(T);
    g(&q);
    S8: r = alloc(T);
}

g(T **fp) {
    T local;
    if(...) 
    s9: p = &local;
}
Example with Flow Sensitivity

```c
T *p, *q, *r;

int main() {
    S1: p = alloc(T);
    f();
    g(&p);
    S4: p = alloc(T);
    S5: ... = *p;
}

void f() {
    S6: q = alloc(T);
    g(&q);
    S8: r = alloc(T);
}

g(T **fp) {
    T local;
    if(...) 
    s9: p = &local;
}
```

\[ P_{S5} = \]

\[ P_{S9} = \]
References:

• “Cloning-based context-sensitive pointer alias analysis using binary decision diagrams”, Whaley and Lam, PLDI 2004
• “Symbolic pointer analysis revisited”, Zhu and Calman, PDLI 2004
• “Points-to analysis using BDDs”, Berndl et al, PDLI 2003
Binary Decision Diagram (BDD)

Binary Decision Tree

Truth Table

BDD

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
BDD-Based Pointer Analysis

• Use a BDD to represent transfer functions
  – encode procedure as a function of its calling context
  – compact and efficient representation
• Perform context-sensitive, inter-procedural analysis
  – similar to dataflow analysis
  – but across the procedure call graph
• Gives accurate results
  – and scales up to large programs
Probabilistic Pointer Analysis

References:

- “A Probabilistic Pointer Analysis for Speculative Optimizations”, DaSilva and Steffan, ASPLOS 2006
- “Compiler support for speculative multithreading architecture with probabilistic points-to analysis”, Shen et al., PPoPP 2003
- “Speculative Alias Analysis for Executable Code”, Fernandez and Espasa, PACT 2002
- “A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion”, Dai et al., CGO 2005
- “Speculative register promotion using Advanced Load Address Table (ALAT)”, Lin et al., CGO 2003
Do pointers a and b point to the same location?
  - Repeat for every pair of pointers at every program point

How can we optimize the “maybe” cases?
Let’s Speculate

- Implement a potentially unsafe optimization
  - Verify and Recover if necessary

```c
int *a, x;
...
while(...) {
    x = *a;
    ...
}
```

*a* is probably loop invariant

```c
int *a, x, tmp;
...
tmp = *a;
while(...) {
    x = tmp;
    ...
}
<verify, recover?>
```
Data Speculative Optimizations

• EPIC Instruction sets
  – Support for speculative load/store instructions (e.g., Itanium)

• Speculative compiler optimizations
  – Dead store elimination, redundancy elimination, copy propagation, strength reduction, register promotion

• Thread-level speculation (TLS)
  – Hardware and compiler support for speculative parallel threads

• Transactional programming
  – Hardware and software support for speculative parallel transactions

Heavy reliance on detailed profile feedback
Can We Quantify “Maybe”? 

- Estimate the potential benefit for speculating:

  - **Recovery penalty** (if unsuccessful)
  - **Expected speedup** (if successful)
  - **Overhead for verify**
  - **Probability of success**

- **Speculate?**

  - **YES**
  - **NO**

Ideally “maybe” should be a probability.
Conventional Pointer Analysis

• Do pointers \(a\) and \(b\) point to the same location?
  – Repeat for every pair of pointers at every program point
Probabilistic Pointer Analysis

- Potential advantage of Probabilistic Pointer Analysis:
  - it doesn’t need to be safe
PPA Research Objectives

• **Accurate points-to probability information**
  – at every static pointer dereference

• **Scalable analysis**
  – Goal: entire SPEC integer benchmark suite

• **Understand scalability/accuracy tradeoff**
  – through flexible static memory model

*Improve our understanding of programs*
Algorithm Design Choices

**Fixed:**
- Bottom Up / Top Down Approach
- Linear transfer functions (for scalability)
- One-level context and flow sensitive

**Flexible:**
- Edge profiling (or static prediction)
- Safe (or unsafe)
- Field sensitive (or field insensitive)
Traditional Points-To Graph

```c
int x, y, z, *b = &x;
void foo(int *a) {
    if(...)
        b = &y;
    if(...)
        a = &z;
    else(...)
        a = b;
    while(...) {
        x = *a;
        ...
    }
}
```

Results are inconclusive
int x, y, z, *b = &x;
void foo(int *a) {
    if(...) \Rightarrow 0.1 \text{ taken (edge profile)}
        b = &y;
    if(...) \Rightarrow 0.2 \text{ taken (edge profile)}
        a = &z;
    else
        a = b;
    while(...) {
        x = *a;
        ...
    }
}
Probabilistic Pointer Analysis Results Summary

- Matrix-based, transfer function approach
  - SUIF/Matlab implementation
- Scales to the SPECint 95/2000 benchmarks
  - One-level context and flow sensitive
- As accurate as the most precise algorithms
- Interesting result:
  - ~90% of pointers tend to point to only one thing
Today’s Class

- Basics
- Design Options
- Pointer Analysis Algorithms
- Pointer Analysis Using BDDs
- Probabilistic Pointer Analysis

Friday’s Class

- Memory Hierarchy Optimization