Points-To Analysis
Meeting 21, CSCI 5535, Spring 2010
Guest Lecture: Manu Sridharan

What is points-to analysis?
• Informally: analysis determining what locations (objects) pointers can point to

```
Program
main() {
  x = &a;
  y = &b;
  z = x;
}

Result
pt(x) = {a}, pt(y) = {b}, pt(z) = {a}
```

Importance of points-to analysis

Verification
Is \{ *x = 10 \} \*y = 3 \{ *x = 10 \} valid?
If x and y cannot point to same location, then yes

Optimization
Can
a = *x; \*y = 4; b = *x
be optimized to
a = *x; \*y = 4; b = a?
If x and y cannot point to same location, then yes

Control Flow
Can l.add() invoke ArrayList.add()?
If I can point to an ArrayList object, then yes

Lecture overview
• What we’ll cover
  – Definition / complexity of several points-to analysis variants
  – Andersen’s analysis via CFL-reachability
  – Some issues with handling method calls
  – Refinement-based analysis
• What we’ll skip (lack of time)
  – Shape analysis (Evan will cover later in semester)
  – Many other optimizations, control-flow analysis of functional languages, ...

Formal definition (for C)

Points-to analysis: Given a program and two variables p and q, points-to analysis checks if p can point to q in some program execution [Chakaravarthy03]
– malloc creates a fresh, unnamed variable

Alias analysis: check if p1 and p2 can point to q simultaneously in some execution
– We’ll focus on points-to
For now, assume no procedure calls

Soundness and precision
• If p can point to q in some execution, a sound analysis will always report it
  – Analysis may be over-approximate, reporting p can point to q even if it cannot
• A precise analysis is sound but not over-approximate
  – Yields exact answer given program semantics
• Precise analysis for C is undecidable
  – Or, for any Turing-complete language
  – foo(TuringMachineM, TMInput i) \{ run M on i; p = &q; \}

Bottom line: to obtain decidability and efficiency, must approximate program semantics
Approximation 1: Path Insensitivity

- Treat all branches as non-deterministic
  - Given if (c) then p; else q, always assume either p or q can execute
  - Must still respect execution order (flow sensitive)

- Complexity
  - With dynamic memory (malloc), undecidable
    - See [Ramalingam94,Chakaravarthy03]
  - Without dynamic memory, PSPACE-complete [MD00]
    - Even with just one procedure!
- **Bottom line:** need to approximate more

Approximation 2: Flow Insensitivity

- Assume statements can execute in any order
  - With possible repetition
  - Assume control-flow graph is complete

- Complexity
  - With dynamic memory (malloc), decidability unknown (!)
  - Without dynamic memory, NP-Hard [Horwitz97]
- **Bottom line:** need even more approximation

Simultaneity

**Semantics of pointer accesses**

- **Pointer Read**
  - \( x = *y \)
  - *Note: black arrows must occur simultaneously*

- **Pointer Write**
  - \( *x = y \)

**Issue:** Some relations cannot arise simultaneously

**Statement set (flow insensitive):**

\[
\begin{align*}
(a=\&c);b=\&a; & c=\&b; b=\&a; & *b=\&c \\
\text{b points to c:} & a=\&c; b=\&a; & a \text{ points to b:} & c=\&b; b=\&a; & *b=\&c \\
\text{But not both!}
\end{align*}
\]

Approximation 3: Andersen’s

- Assumes discovered points-to relations can all occur simultaneously
  - Hence, less precise handling of pointer accesses

- **Challenge:** express as approximate semantics?

- Breaks up multi-level derefs
  - \( **x \) becomes temp = \( *x \), *temp = \( y \)
  - Again, imprecision due to simultaneity reasoning (\( **x \) does two derefs atomically)

- Heap abstraction? Other? (I don’t know)

- **Complexity:** \( O(N^2) \); much better!

Andersen’s for Java: The Basics

- Four statement types
  - new: \( x = \text{ new Obj()} \)
  - assign: \( x = y \)
  - getfield: \( x = y.f \)
  - putfield: \( x.f = y \)

- Single abstract location for each new
  - Represents objects allocated by all executions
  - For more precise treatment, **shape analysis**
CFL-Reachability

- Nodes represent variables/abstract locations
- Edges represent statements

Points-to analysis paths:
- flowsTo-path from o to x: \( o \in pt(x) \)
- alias-path from x to y: \( pt(x) \cap pt(y) \neq \emptyset \)

More on CFL-Reachability

- Several variants
  - All-pairs: find all pairs of nodes connected by valid paths
  - Single-source: find all nodes to which source is connected by valid path
- General algorithm \( O(N^3) \)
  - \( N \) is number of nodes
  - Faster algorithms for special cases (see [RHS95])
  - Specialized algorithm needed to scale pointer analysis
- For more details, see [Reps98]

Andersen’s Analysis in CFL-Reachability

- \( x = \text{new} \ \text{Obj}(); // o_1 \)
- \( z = \text{new} \ \text{Obj}(); // o_2 \)
- \( w = x; \)
- \( y = x; \)
- \( y.f = z; \)
- \( v = w.f; \)

What about alias?

- **Want**: \( x \ alias \ y \Leftrightarrow \exists o. o \ flowsTo x \wedge o \ flowsTo y \)
- **Problem**: need all edges in same direction
- **Solution**: alias \( \Rightarrow \) flowsTo flowsTo
  - flowsTo is inverse of flowsTo
  - Must add inverse edges to graph (e.g., assign)
  - See [SB06] for full grammar

Importance of Handling Method Calls

- Used pervasively, esp. in Java-like languages
- Often deeply related to objects and pointers

```
class ArrayList {
    Object[] elems;
    int i;
    public ArrayList() {
        this.elems = new Object[10];
    }
    public void add(Object o) {
        this.elems[i++] = o;
    }
    public Object get(int i) {
        return this.elems[i];
    }
}
```
Precise Handling of Method Calls

- **Idea**: analyze as if **all method calls inlined**
  - Yields separate copies of local variables / `new` expressions for each possible call
  - Known as **context-sensitive** analysis
- **Problem**: how to handle recursion
  - Full inlining yields an infinite program
  - But, analysis definitions still work fine!
    - Require variables `p` and `q` **up front**: forces choice of inlined copy
    - **Flow-insensitive**: find finite sequence from infinite statement set

Decidability with Context Sensitivity

- **Precise path-insensitive + dynamic memory still undecidable**
  - Already undecidable with just one method
- **Flow-insensitive + dynamic memory + precise calls**: **undecidable**
  - Recall that with one method, decidability unknown
    - Via small modification of [Reps00] proof
    - Even Andersen-style analysis + precise calls is undecidable (details coming up)
- **No dynamic memory**: not well-studied
  - Note that stack frames are a form of dynamic memory

Andersen’s and Calls, Simplified

- Four statement types (ignore fields for now)
  - `new`: `x = new Obj();`
  - `assign`: `x = y`
  - `call`: `x = m(p1, p2, ...)`
  - `return`: `return x`
- **Idea**: use balanced parentheses to match calls and returns
  - Parenls labeled by call site
  - Grammar filters out **unrealizable paths** (method call returning to wrong site)
  - Classic use of CFL-reachability [RHS95]

Matching Calls and Returns: Example

```
id(p) { return p; }  // o1
x = new Obj();        // o2
a = id(x);            // 1
b = id(y);            // 2
```

```
S → SS | (S) | N | ε
N → new | assign
```

Andersen’s and Calls: The Details

- Must allow for **partially balanced** call parens
  - E.g., to handle `makeObj() { return new Obj(); }`
- Handle fields and calls simultaneously via **intersected languages**
  - Enhance `N` production (previous slide) to include all field accesses
  - Points-to analysis must find paths that are both `S` paths (for calls) and `flowsTo` paths (for fields)
- Also need barred edges, etc.; details in [SB06]

Andersen’s and Calls: Decidability

- Analysis requires solving reachability over intersection of two CFLs (`S` and `flowsTo`)`
- But, CFLs are **not closed** under intersection
- In our case, problem is undecidable
  - Proof via reduction from PCP [Reps00]
- Standard approach for decidability: **approximate recursion**
  - Collapse SCCs in call graph (change `{` into `assign`)
  - Yields imprecise handling of recursive calls / returns
**Refinement Overview**

- **Goal**: "good" answers for client with less cost
  - For a verifier client, "no bug" is good answer
  - For query "can x point to o?", good answer is NO
- **Refinement loop**
  - First approximate
  - If requested by client, add targeted precision
  - Continue until (1) good answer, (2) fully precise, or (3) timeout
- **Challenge**: make it work for pointer analysis!

**Approximation via Match Edges**

- Match edges connect matched field parens
  - From source of open to sink of close
  - Initially, all pairs connected
- Use match edges to **skip subpaths**

**Scaling Context-Sensitive Analysis**

- With recursion approximation, context-sensitive Andersen’s is **exponential**
  - Same explosion as from inlining
- Standard approaches to scaling
  - k-limiting (reduced precision)
  - Efficient data structures (e.g., BDDs)
  - Smarter inlining choices (object sensitivity)
- **Bottom-line performance**: minutes of times, GB of memory
  - No good for interactive tools like IDEs

**Single path problem**

- **Problem**: show path is unbalanced
- **Goal**: reduce number of visited edges
- **Insight**: enough to find one unbalanced paren

**Refining the Approximation**

- Refine by removing some match edges
  - Exposes more of original path for checking
- Correctness from proper nesting
  - Traversing match assumes skipped path balanced
  - Must try all outgoing match edges
- Remove where unbalanced parens expected
  - Explore deeper levels of pointer indirection
Refinement With Both Languages

Match edges force approximation of calls
- Can only check calls on match-free subpaths

Match edge removal yields more call checking

Key novelty: refine heap and calls together

Context-Sensitive Analysis Comparison

- Refinement-based analysis gave best precision and performance in practice [5806]
  - Answer for a variable in 1 second, 35 MB of memory (vs. minutes, GB)
  - Precision measured for real clients
- New comparison needed with more recent work [8509]
- Refinement advantages
  - Suitable for interactive tools (like an IDE)
  - Works on huge programs and libraries; exhaustive dies
- Refinement policy easily tuned for different clients
- Drawback of refinement: sensitive to heuristics
- Which match edges should be removed?
- Okay for papers, but can be undesirable in real world

References

- Decidability / complexity
  - [94] [95] [96]: The undecidability of aliasing. TOPLAS 16(5):1467-1473, 1994.
- [100]: Chakraverty, New Results on the Computation and Complexity of Points To Analysis. POPL 2003.
- CFL-Reachability
- Scalable pointer analysis

Summary

- Points-to analysis is essential for modern program analysis
- Precise points-to analysis is hard
- Many approximations needed for tractability
  - Some theoretical questions remain unsolved
- Andersen’s approximation seems to be a good precision / performance compromise
- Tons of scalability work (not discussed here)
- Trends support a client-driven, refinement-based approach
  - Platforms, libraries always getting bigger
  - Interactive tools need better analysis