Flow-Insensitive Pointer Analysis

Last time
- Interprocedural analysis
- Dimensions of precision (flow- and context-sensitivity)
- Flow-Sensitive Pointer Analysis

Today
- Flow-Insensitive Pointer Analysis

The defining characteristics
- Ignore the control-flow graph, and assume that statements can execute in any order
- Rather than producing a solution for each program point, produce a single solution that is valid for the whole program

Flow-insensitive pointer analyses
- Andersen-style analysis: the slowest and most precise
- Steensgaard analysis: the fastest and least precise
- All other flow-insensitive pointer analyses are hybrids of these two
Andersen-Style Pointer Analysis [1994]

Basic idea
- View pointer assignments as constraints
- Use these constraints to propagate points-to information

March 9, 2015 Interprocedural Analysis

void foo()
{
    c = &f;
    e = &c
    b = a;
    if (C) { *e = b; d = *e; a = d; }
}

\[ c \supseteq \{f\} \]
\[ e \supseteq \{c\} \]
\[ b \supseteq a \]
\[ d \supseteq *e \]
\[ *e \supseteq b \]

Goal: compute the smallest points-to sets that satisfy these constraints
Andersen-Style Pointer Analysis

**Constraint Graph**

- e \supseteq \{f\}
- e \supseteq \{c\}
- a \supseteq d
- b \supseteq a
- d \supseteq \ast e
- \ast e \supseteq b

Notice that the constraint graph grows dynamically

**Key Point**
Performance depends on
1. number of edges added
2. propagation across edges

March 9, 2015
Interprocedural Analysis
Inclusion-based Pointer Analysis

**Essentially**
- Computes the transitive closure of a dynamic graph

**Naïve algorithm doesn’t scale— O(n^3)**
- Too many edges added
- Quickly runs out of memory

**Optimizations**
- Cycle detection
- Location equivalence

---

Cycle Detection

\[
a' = \{a,b,c,d\}
\]

Faehndrich et al 1998
Cycle Detection

**How do we detect cycles?**
- They appear dynamically during the analysis
- Check for cycles too often $\rightarrow$ Too much overhead
- Check for cycles too infrequently $\rightarrow$ Lost opportunities
- Need to find a sweet spot

**Two solutions** [Hardekopf & Lin ‘07]
- Lazy Cycle Detection
- Hybrid Cycle Detection

Lazy Cycle Detection

**Fact**
- Cycles cause identical points-to sets

**Heuristic**
- Identical points-to sets indicate possible cycles
- Don’t look for a cycle unless we have evidence that one might exist
- Perform cycle detection when two nodes have identical points-to sets

**Result**
- Faster than all previous cycle detection schemes
- See paper for details
Hybrid Cycle Detection

<table>
<thead>
<tr>
<th>cheap</th>
<th>many cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before the Analysis</td>
<td>Hybrid Cycle Detection</td>
</tr>
<tr>
<td>During the Analysis</td>
<td></td>
</tr>
</tbody>
</table>

Idea: Pre-process the constraint graph with an offline component to make the online component more efficient

Two components

- Offline component (before the analysis)
- Online component (during the analysis)

Hybrid Cycle Detection– Offline Component

Constraint Graph

```plaintext
*a* → {a, b, d}
c ⊇ {f}
e ⊇ {c}
a ⊇ d
b ⊇ a
d ⊇ *e
*e ⊇ b
```
Hybrid Cycle Detection—Online Component

Finds cycles at earliest possible opportunity

Never has to traverse the constraint graph
Evaluation

**Compare our work with previous state of the art**
- First need to identify the state of the art
  - Routev et al 2000 (OVS)
  - Heintze et al 2001 (HT)
  - Berndl et al 2003 (BLQ)
  - Pearce et al 2004 (PKH)
- For a fair comparison, we implement algorithms from scratch using the same infrastructure
- Compare analysis time and memory consumption on 10 C benchmarks with 100K – 2M LOC

- Higher represents worse performance
- 4× faster
- 7× less memory
- Scales to 2M LOC
Impact

**Academic**
- PLDI 2007 Best Paper Award
- Raised the bar for empirical evaluation

**Industrial**
- Implemented in gcc and LLVM compilers
- Implemented by Semantic Designs, Inc
  - Some of their software engineering tools can now scale to over 12M lines of C (previously stuck at 1M)

---

Andersen-style Pointer Analysis – Procedure Calls

**Program**

```c
foo(int* x){
    . . .
    return x;
}
```

**Constraints**

```
x \geq b
a \geq x
```

**a := foo(&b)**

**How do we handle procedure calls?**
- Insert constraints for copying actual parameters to formal parameters
- Insert constraints for copying return values
Steensgaard Pointer Analysis [1996]

Basic idea
- Further reduce precision by using equality constraints
- That is, information flows both ways, rather than from the right-hand side to the left-hand side of the constraint

Tradeoffs
- Extremely imprecise
- A system of equality constraints can be solved in near-linear time
- Running time is $O(n \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann’s function.
  - $\alpha(2^{132}) < 4$

Key idea
- The key to this algorithm is the Union-Find data structure.

Steensgaard Pointer Analysis – Union-Find

The Union-Find data structure
- Maintains a set of disjoint sets and supports two operations:
  - Find(x) : return the set containing x.
  - Union(x, y) : union the two sets containing x and y.

Set Representation
- Sets are represented by a distinguished element called the set representative
- Each set is an inverted tree, with nodes pointing to their parents and the set representative as the root
Steengaard Pointer Analysis – Union-Find

Union(a, b)
  - Find(a)
  - Find(b)

March 9, 2015  Interprocedural Analysis  21

Steengaard Pointer Analysis – Union-Find

Union(a, c)
  - Find(a)
  - Find(c)

March 9, 2015  Interprocedural Analysis  22
Steensgaard Pointer Analysis – Union-Find

Union(a, d)
- Find(a)
- Find(d)

Union-by-rank
- When performing the Union operation, choose the set representative based on the sizes of the two sets

Path compression
- Avoid redundant searches for the set representative

Two key optimizations
- Path compression
- Union-by-rank
- Together these optimizations yield near-linear time operations
Steensgaard Pointer Analysis – Path Compression

\[ \text{Union}(a, b) \]
- \text{Find}(a)
- \text{Find}(b)

\[ \text{Union}(a, c) \]
- \text{Find}(a)
- \text{Find}(c)
Steensgaard Pointer Analysis – Path Compression

Union(a, d)
- Find(a)
- Find(d)

Steensgaard Pointer Analysis – Union-by-Rank

Union(a, b)
- Find(a)
- Find(b)
Steensgaard Pointer Analysis – Union-by-Rank

Union(a, c)
- Find(a)
- Find(c)

What is the benefit of union-by-rank?
- It ensures that we update as few parent pointers as possible
- Consider the cost of selecting d as the new set representative in this last union operation
Steensgaard Pointer Analysis – Example 1

Program | Constraints | Points-to Relations
---|---|---
a := &b | a = { b, d } | a, c, e
b := a | c = a |
e := a | e = a |

Steensgaard Pointer Analysis – the Algorithm

```
merge(x, y)
{
    x = Find(x); y = Find(y);
    if (x == y) then return;
    Union(x, y);
    merge(points-to(x), points-to(y));
}
```

for each constraint LHS = RHS
merge(LHS, RHS)
Steensgaard Pointer Analysis – Example 2

Program

| a := &b | Constraints: a = { b } |
| c := &d | c = { d } |
| e := &a | e = { a } |
| f := a | f = a |
| *e := c | *e = c |

Points-to Relations

March 9, 2015

Interprocedural Analysis
Andersen vs. Steensgaard

```c
int **a, *b, c, *d, e;
1: a = &b;
2: b = &c;
3: d = &e;
4: a = &d;
```

**Andersen-style analysis**

- A flow graph:
  - Input vertices: `a`, `b`, `c`, `d`, `e`
  - Output vertices: `a`, `b`, `c`, `d`, `e`

  Edges:
  - `a` to `b`
  - `b` to `c`
  - `d` to `c`
  - `a` to `d`

  Output graph:
  - `a` to `b`
  - `b` to `c`
  - `d` to `c`
  - `a` to `d`

**Steensgaard analysis**

- A flow graph:
  - Input vertices: `a`, `b`, `c`, `d`, `e`
  - Output vertices: `a`, `b`, `c`, `d`, `e`

  Edges:
  - `a` to `b`
  - `b` to `c`
  - `d` to `c`
  - `a` to `d`

  Output graph:
  - `a` to `b`
  - `b` to `c`
  - `d` to `c`
  - `a` to `d`

Due to statement 4

The Big Picture

**Precision vs. Performance**

- Steensgaard’s analysis and Andersen’s analysis operate on abstractions of the program text
- Instead of the CFG, they operate on sets
- These abstractions trade off precision for performance
Concepts

**Flow-insensitive pointer analysis**

**Andersen-style analysis**
- Inclusion-based, subset-based
- Compute transitive closure of a dynamic graph
- Constraint graph
- Cycle elimination optimization

**Steensgaard-style analysis**
- Unification-based, equality-based
- Union-find data structure

Next Time

**Lecture**
- Context-Sensitive Pointer Analysis