Lecture 11: Pointer Analysis (2/2)

Claire Le Goues

15-819O: Program Analysis

Pointer (or points-to) analysis

```
I ::= ...
    | p := & x
    | p := q
    | *p := q
    | p := *q
```

What memory locations can a pointer expression refer to?

- Alias analysis: When do two pointer expressions refer to the same storage location?
  1. int x;
  2. p = &x;
  3. q = p;
- *p and *q alias, as do x and *p, and x and *q

Problem statement

- Consider flow-insensitive may pointer analysis
- Assume pointers $p,q \in P$ and address-taken variables $a,b \in A$ are disjoint
  - Can transform program to make this true: For any variable v for which this isn't true, add statement pv = &av, replace v with *pv
- Want to compute relation pts : $P \cup A \rightarrow 2^A$
  - Essentially *points to* pairs

Andersen-style pointer analysis

- View pointer assignments as *subset constraints*
- Use constraints to propagate points-to information, solve them:

• p = &a;	• p ⊇ {a}
• q = p;	• q ⊇ p
• p = &b;	• p ⊇ {b}
• r = p;	• r ⊇ p

Constraints

- Interprocedural, context-insensitive, flow insensitive:

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p} \; address\text{-}of$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q} \; copy$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q} \; assign$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q} \; dereference$$

Abstracting the rules

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p} \; copy$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r} \; assign$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p} \; dereference$$

Example

```
1. z := 1
2. if cond
3.     p := y
4. else
5.     p = &z;
6. *p := 2
7. print z
```

Dynamic memory allocation?

```
1.  q := malloc1()
2.  p := malloc2()
3.  p := q
4.  r := &p
5.  s := malloc3()
6.  *r := s
7.  t := &s
8.  u := *t
```

$$\frac{}{[\![p := malloc_n()]\!] \hookrightarrow l_n \in p}\ malloc$$

Modeling (dynamic) memory

- Common solution: For each allocation statement, use one node per context.
  - (Note: could choose context-sensitivity for modeling heap locations to be less precise than context-sensitivity for modeling procedure invocation)
- Other solutions:
  - One node for entire heap
  - One node for each type
  - Nodes based on analysis of “shape” of heap

Efficiency

- Constraints generated in O(n) pass over the program.
- Solution size is $O(n^2)$
- Constraints propagated up to n^3 times:
  - O(n) constraints of the form $p \supseteq q$, $*p \supseteq q$, or $p \supseteq *q$.
  - $p \supseteq q$ constraint may fire at most O(n) times, because there are at most O(n) premises of the form $l_x \in p$.
  - $p \supseteq *q$ could cause $O(n^2)$ rule firings, because there are O(n) premises each of the form $l_x \in p$ and $l_r \in q$.
  - With O(n) constraints of the form $p \supseteq *q$ and $O(n^2)$ firings for each, we have $O(n^3)$ constraint firings overall.
  - A similar analysis applies for $*p \supseteq q$ constraints.

Field-sensitivity

- Insensitive analysis (all fields in a struct are equivalent):
  1. p.f = x ;
  2. p.g = y ;
- p.f could point to y??
- More precise to track the contents each field of each abstract location separately (assume can't take the address of a field; true in Java, not C).

Field-sensitivity

- Constraints for fields:

$$\frac{p \supseteq q.f \quad l_q \in q \quad l_f \in l_q.f}{l_f \in p} \; \textit{field-read}$$

$$\frac{p.f \supseteq q \quad l_p \in p \quad l_q \in q}{l_q \in l_p.f} \; \textit{field-assign}$$

- If objects are represented by abstract locations l:

$$\frac{}{[\![p := q.f]\!] \hookrightarrow p \supseteq q.f} \; \textit{field-read}$$

$$\frac{}{[\![p.f := q]\!] \hookrightarrow p.f \supseteq q} \; \textit{field-assign}$$

Steensgaard

- Cubic time is too slow for large programs; Steensgaard proposed a near-linear time constraint based algorithm.
- Challenge 1: space.
  - Pointer analysis inherently requires n^2 space.
  - Why?
- Representation: constaint space for each variable.
- Associate each variable p with an abstract location, and then a single points-to relation between the abstract location and another one to which it may point.
- Key: if it may point to more than one, *unify* multiple possible pointed-to abstract locations.
  - Tradeoff?

Example

1. p := &x
2. r := &p
3. q := &y
4. s := &q
5. r := s

(GRAPHS ON BOARD!)

Precise definition

$$\frac{}{[\![p := q]\!] \hookrightarrow join(*p, *q)}\ copy$$

$$\frac{}{[\![p := \&x]\!] \hookrightarrow join(*p, x)}\ address\text{-}of$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow join(*p, **q)}\ dereference$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow join(**p, *q)}\ assign$$

Precise definition

- For each abstract location *p*, associate the abstract location that *p* points to, denoted **p*.
- Implemented as a union-find data structure for efficient merging.
- Implicitly invoke *find* on an abstract location before calling *join* on it, or before looking up the location it points to.
- *Join* implements a union operation on the abstract locations. Since we are tracking what each abstract location points to, we must update this information also:

```
1.  join(e1, e2)
2.      if (e1 == e2)
3.          return
4.      e1next = *e1
5.      e2next = *e2
6.      unify(e1, e2)
7.      join(e1next, e2next)
```

Steensgaard's example

```
1. a := &x
2. b := &y
3. if p then
     y := &z
1. else
2.   y := &x
3. c := &y
```

Efficiency

- Each statement is processed once.
- Processing is linear, except for find on union-find (amortized time O(α(n)) each and the join operation.
- Short circuit in Join fails at most O(n) times; when it fails, two abstract locations are unified, at cost O(α(n))
- At most O(n) operations and the amortized cost of each operation is at most $O($\alpha$(n))$, overall running time of the algorithm is near linear: $O(n * α(n))$.
- Space consumption is linear, as no space is used beyond that used to represent abstract locations for all the variables in the program text.
- Steensgaard's algorithm was run on a 1 million line program (Microsoft Word) in 1996; order of magnitude greater scalability than other analyses known at the time.
- Caveat: field-insensitive; making it field-sensitive would mean that it is no longer linear.

Context-sensitivity in Andersen's analysis

- Analyze each function separately for each calling point.
- Track current context, the calling point *n* of the current procedure.
- Track separate values for:
  - each variable x_n according to the calling context n of the procedure defining it,
  - each memory location lk_n according to the calling context n active when that location was allocated at new instruction k.

Context-sensitivity in Andersen's analysis

$$\frac{n \vdash p := \mathbf{new}_k\ A}{l_n^k \in p_n}\ new$$

$$\frac{n \vdash p := q \quad l_n \in q_n}{l_n \in p_n}\ copy$$

$$\frac{n \vdash x.f := y \quad l_x \in x_n \quad l_y \in y_n}{l_y \in l_x.f}\ field\text{-}read$$

$$\frac{n \vdash x := y.f \quad l_y \in y_n \quad l_z \in l_y.f}{l_z \in x_n}\ field\text{-}assign$$

$$\frac{n \vdash f_k(y) \quad l_y \in y_n \quad [\![f(z) = e]\!] \in Program}{l_y \in z_k \quad k \vdash e}\ call$$

Example

```
1.    interface A { void g(); }
2.    class B implements A { void g() { ... } }
3.    class C implements A { void g() { ... } }
4.    class D {
5.        A f(A a1) { return a1; }

6.    // in main()
7.    D d1 = new D();
8.    if (...) {
9.        A x = d1.f(new B());
10.       x.g() // which g is called?
11.   else
12.       A y = d1.f(new C());
13.    y.g() // which g is called?
14.   …
15.   }
```

Solution: Object-sensitive analysis

$$\frac{l \vdash p := \mathbf{new}_k\ A}{l_l^k \in p_l}\ new$$

$$\frac{l \vdash p := q \quad l_l \in q_l}{l_l \in p_l}\ copy$$

$$\frac{l \vdash x.f := y \quad l_x \in x_l \quad l_y \in y_l}{l_y \in l_x.f}\ field\text{-}read$$

$$\frac{l \vdash x := y.f \quad l_y \in y_l \quad l_z \in l_y.f}{l_z \in x_l}\ field\text{-}assign$$

$$\frac{l \vdash x.f(y) \quad l_x \in x_l \quad l_y \in y_l \quad [\![f(z) = e]\!] \in Program}{l_x \in \mathbf{this}_{l_x} \quad l_y \in z_{l_x} \quad l_x \vdash e}\ call$$

Object-sensitive pointer analysis

- Organizing a program around objects makes the objects themselves the most interesting thing to analyze.
- Milanova, Rountev, and Ryder. *Parameterized object sensitivity for points-to analysis for Java*. ACM Trans. Softw. Eng. Methodol., 2005.
  - Context-sensitive interprocedural pointer analysis
  - For context, use stack of receiver objects
- Lhotak and Hendren. *Context-sensitive points-to analysis: is it worth it?* CC 06
  - Object-sensitive pointer analysis more precise than call-stack contexts for Java
  - Likely to scale better
- Bravenboer and Smaragdakis in OOPSLA 2009.
  - generates declarative Datalog code to represent the input program, and a datalog evaluation engine solves what are essentially declarative constraints to get the analysis result.
- Smaragdakis, Bravenboer, and Lhotak, POPL 2011
  - Type-sensitive analysis, tracks only the type of the receiver (and, for depths > 2, type of the object that created the receiver, etc.). Nearly as precise as object-sensitive analysis and much more scalable.

Pointer analysis in Java more generally

- Different languages use pointers differently
- Most C programs have many more occurrences of the address-of (&) operator than dynamic allocation
  - & creates stack-directed pointers; malloc creates heap-directed pointers
- Java allows no stack-directed pointers, many more dynamic allocation sites than similar-sized C programs
  - Java strongly typed, limits set of objects a pointer can point to, which can improve precision
- Call graph in Java depends on pointer analysis, and vice-versa (in context sensitive pointer analysis)
- Dereference in Java only through field store and load
- And more…Larger libraries in Java, more entry points in Java, can't alias fields in Java, ...