Extra notes on Field-sensitive points-to analysis with inclusion constraints Sept 9, 2015

Week 3 CS6304

The Rountev et al. paper at OOPSLA 2000 extended Andersen's points-to analysis for C programs to a statically typed programming language such as Java. The version of Andersen which was extended was elaborated in papers at PLDI 1998 and POPL 2000 by Alex Aiken's group which used non-standard type inclusion constraints to express the points-to problem formulation. Constraint satisfaction is a solution procedure used to infer (or reconstruct) types in programming languages in which programs do not declare types, but rather infer them from how a variable is used. Two kinds of constraints are used in these specific papers on points-to, namely unification constraints and inclusion constraints. Given a pointer copy statement **p=q** in C, unification constraints will express the fact that after this assignment p and q can point to the same objects, clearly an approximation of what actually happens at runtime. This is written Pts(p) = Pts(q). Using inclusion constraints, after this assignment the constraints will express that the set of objects pointed to by p includes the set of objects pointed to by q. This is written Pts(q) <= Pts(p) (actually <= is written as subset equality or inclusion). Clearly the inclusion constraints are more precise in expressing the semantics of the assignment, but they also are harder to solve. For the same problem using the unification constraints yields a much cheaper to solve and more compact set of constraints, but a more approximate solution.

At the time of the paper, several analysis techniques tended to look at all the code, user code plus libraries. This made scalability of the analysis an issue. As was stated in the presentation, this new points-to analysis used reachability from the program entry points (i.e., main(), methods invoked at JVM startup, class initialization methods e.g., for static fields) to determine which methods' code were analyzed.

In the slides that follow, there are 2 different descriptions of a field-sensitive points-to analysis algorithm – one matches the use of constraints in the paper – the other uses a formulation more like the dataflow analysis equations we have looked at.

Field-sensitive Points-to Analysis (FieldSens)

- Flow-insensitive, context-insensitive extension of Andersen's analysis for C
  - Have to handle dynamic dispatch, fields, and libraries
- Can use a precomputed callgraph or can compute an on-the-fly callgraph from the points-to relations being calculated
- Distinguishes object fields
- Originally formulated as a constraint solution problem -- admits a dataflow formulation too

Rountev, A. Milnova, B. Ryder, "Points-to Analysis for Java Using Annotated Constraints", OOPSLA' 00;
Lahotak and Hendren, "Scaling Java Points-to Analysis using SPARK", CC' 03

Extra notes on Field-sensitive points-to analysis with inclusion constraints **Sept 9, 2015**

The following slides show and example of the field-sensitive points-to algorithm, an explanation of the dataflow effects of each of the 4 types of Java statements that can affect points-to analysis and an outline of how the analysis works described as a worklist algorithm.

Let's walk through the program. We define classes A and B, with B a subclass of A that overrides m(). The program creates new B and X objects. It refers to the B object o_1 through reference variables a and b. The call a.m(x) calls the m method in B because a is pointing to o_1 a B object. This_B.m is associated with o_1, the receiver and q_B.m is associated with o_2 which is the X object used as a parameter.

On the next slide we see Java statements and their points-to effects described. Note that more complex statements can be decomposed into sequences of these statements. The algorithm here is described in words and conditions – there are no actual dataflow functions written.

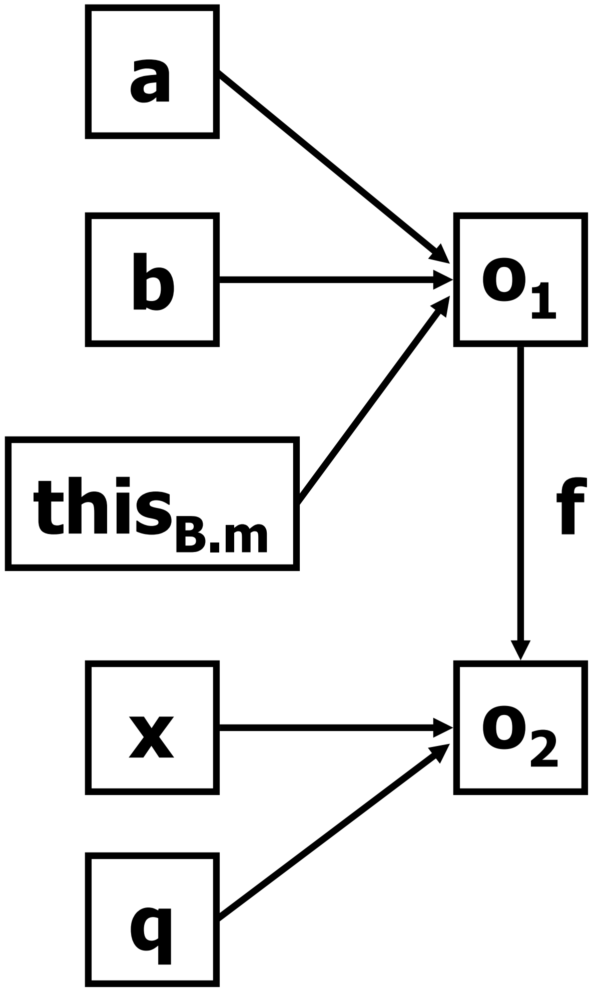
Points-to Analysis in Action

```
class A { void m(X p) {..} }

class B extends A {
    X f;
    void m(X q) {  this.f=q; }
}

B b = new B();
X x = new X();
A a = b;
a.m(x);
```

A.m() not analyzed because it's unreachable.



Rules of Algorithm

- 4 types of reference assignment statements
  - Each has points-to effects
    - Allocation: p=new X()
      - Adds o_X to Pts(p)
    - Copy: p = q
      - Pts(p) $\subseteq$ Pts(q) (i.e., If o $\in$Pts(q), then o $\in$Pts(p))
    - Field store: p.f = q
      - If o$\in$Pts(p) and r $\in$Pts(q), then r $\in$ Pts(o.f)
    - Field load: p = q.g
      - If o $\in$ Pts(q) and oo $\in$ Pts(o.g) then oo $\in$ Pts(p)
- Algorithm described as construction of a points-to graph
  - Nodes: reference variables and fields
  - Edges: ref ---> object or field labelled: obj--->obj

FieldSens Algorithm

Initially

- **Make a list of all reference assignments excluding allocations**
- **Process allocations first and create the corresponding points-to relations**
- **Create a worklist intialized with all the reference variables and fields**
  - **As the algorithm proceeds, any reference variable (or object field) whose points-to set has changed will be put on the worklist**

FieldSens Algorithm, cont

Repeat until the worklist is empty

- Remove a variable from the worklist and iterate through the statements (involving this variable)
  - Calculate points-to relations implied by the assignment statement; if this changes the points-to set of a variable or object field, add it to the worklist (Note: if p points to o and o.f' s points-to set is changed, <p.f= >, then we record o.f as a variable whose points-to set has changed.)

Differences with Algm for C

- **Fields cannot be ignored**
  - **Field-sensitive versus field-based**
- **No explicit address operator & in Java**
- **Reachability of code**
    - **On-the-fly call graph construction versus use of a static approx call graph**
    - **Possibly multiple entries to call graph (e.g., static initializers, thread start methods, finalizers, etc)**
- **Need to know entire set of classes that will ever be loaded during execution**
    - **Because of reflection**
    - **Large Java libraries vs. smaller C libraries**
- **Can use strong typing to filter out spurious points-to relations**

Field-sensitive Points-to Analysis w Inclusion Constraints

- **Based on Andersen's points-to analysis**
- **Define and solve a system of annotated set-inclusion constraints - different from dataflow formuation**
  - **Handles virtual calls by simulation of run-time method lookup**
  - **Models the fields of objects**
  - **Extended BANE (UC Berkeley) constraint solver**
- **Analyzes only possibly executed code**
  - **Ignores unreachable code from libraries**

Rountev, A. Milnova, B. Ryder, "Points-to Analysis for Java Using Annotated Constraints" OOPSLA'00

Extra notes on Field-sensitive points-to analysis with inclusion constraints **Sept 9, 2015**

These slides present a simplified version of the constraints used in the OOPSLA 2000 paper Rountev et al. to show more clearly how the constraints capture the same actions as the dataflow formulation. The two annotated constraint slides introduces simpler representations. For example, V_p represents the points-to set of the reference variable p; V_o represents a set of points-to set of an abstract object o (one for each field of object o). Note that abstract object o is used by the analysis to represent all the actual objects possibly created at a particular creation site (i.e., a new statement in the program). The analysis represents object o by the term ref(o, V_o). Then we have how to represent edges in the points to graph by these constraint terms….

p points-to abstract object o is represented by ref(o,V_o) within V_p

abstract object o_1 points to abstract object o_2 through its f field is represented by

ref(o_2, V_o_2) within sub f V_o_1

Annotated Constraints

- **Form:** $L \subseteq_a R$
  - L **and** R **denote set expressions**
  - **Annotation** a**: additional information (e.g., object fields)**
- **Kinds of set expressions** L **and** R
  - **Set variables: represent points-to sets**
  - ***ref* terms: represent objects**
  - **Other kinds of expressions**

Set variables and *ref* terms

- Set variables represent points-to sets
  - For each reference variable p: V_P
  - For each object o: V_o
- Object o is denoted by term $\mathit{ref}(o, V_o)$

$$\begin{tikzcd}
\boxed{p} \arrow[r] & \boxed{o} \arrow[r, Rightarrow] & \boxed{ref(o,V_o) \subseteq V_p} \\
\boxed{o_1} \arrow[r, "f"] & \boxed{o_2} \arrow[r, Rightarrow] & \boxed{ref(o_2,V_{o_2}) \subseteq_f V_{o_1}}
\end{tikzcd}$$

Extra notes on Field-sensitive points-to analysis with inclusion constraints **Sept 9, 2015**

Now to work through an example:

Constraint generation:

p = new A(); generates the constraint **ref(o_1,V_o_1) within V_p** where o_1 is the new A abstract object

q= new B(); generates the constraint **ref(o_2, V_o_2) within V_q** where o_2 is the new B abstract object

p.f=q ; generates 2 constraints because we need to represent all the objects that p can point to. So W is used

as a placeholder for each object that p points to represented by **V_p within proj(ref,W)**

then we need a 2nd constraint that expressed the fact that the points-to set of W's f field is enlarged by

those objects pointed to by q: **V_q within sub f W**

Constraint resolution:

Top 4 constraints were generated on the last page; consider them numbered 1-4, and then number each constraint in order 5-6. We derive constraint 5 by combining constraints 1 and 3 and simplify the condition on W, according to the rules in the paper. The shape of this rule may not be intuitive but it is important so that the resulting constraint combines well with constraint 4. Line 6 results from combining line 4 with line 5 and clearly states that the points-to set of q is contained in the points to set of o_1.f

As Krishnan explained in class (and as in the paper) there are rules about when you can combine constraints with annotations on their within operator and about how to simplify the constructed constraints (i.e., those with proj and re in them)

As I said in class, this nomenclature is used for all sorts of problems besides type inference… and there are efficient solvers for these sorts of constraints – which is why people pose their problem using constraints and then wait for a simplified constraint set to appear from which you can read the answer. Constraint-based formulations are still used today, especially to prototype new analyses.

Example: Accessing Fields

```
p = new A();
q = new B();
p.f = q;
```

$$\mathrm{ref}(o_1, V_{o_1}) \subseteq V_p$$

$$\mathrm{ref}(o_2, V_{o_2}) \subseteq V_q$$

$$V_p \subseteq \mathrm{proj}(\mathrm{ref}, W)$$

$$V_q \subseteq_f W$$

$$\begin{tikzcd} \boxed{p} \arrow[r] & \boxed{o_1} \arrow[d, "f"] \\ \boxed{q} \arrow[r] & \boxed{o_2} \end{tikzcd}$$

Think of W as representing V_{o1} and V_{o3} when p points to o1, o3

Constraint generation

Example: Solving Constraints

$ref(o_1, V_{o_1}) \subseteq V_p$

Constraint resolution

$ref(o_2, V_{o_2}) \subseteq V_q$

$V_p \subseteq proj(ref, W)$

$V_q \subseteq_f W$

$W \subseteq V_{o_1}$

$V_q \subseteq_f V_{o_1}$

$ref(o_2, V_{o_2}) \subseteq_f V_{o_1}$

$o_1.f$ points to o_2

Example: Virtual Calls

Constraint generation from actual call:

p.m(x); $\Rightarrow$ $V_P \subseteq_{\boxed{m}} \boxed{lam(V_x)}$

Constraint resolution:

receiver object **o** $\Rightarrow$ $ref(\boxed{o}, V_O) \subseteq V_P$

Actual method called, **A.m(z)** $\Rightarrow$ $V_x \subseteq V_z$

$ref(o, V_O) \subseteq V_{this(A.m)}$