Lecture 26: Pointer Analysis

[Based on slides from R. Bodik]

Administrivia

- HKN survey on Wednesday. Worth 5 points (but you *must* show up!).

Today

- Points-to analysis: an instance of static analysis for understanding pointers
- Andersen's algorithm via deduction
- Implementation of Andersen's algorithm in Prolog

General Goals of Static Analysis

- Determine run-time properties statically at compilation.
- Sample property: "is variable x a constant?"
- Since we don't know the inputs, must consider all possible program executions.
- Conservative (err on the side of caution) for *soundness:*
  - allowed to say x is not a constant when it is,
  - but not that x is a constant when it is not .
- Many clients: optimization, verification, compilation.

Client 1: Optimizing virtual calls in Java

- Motivation: virtual calls are costly, due to method dispatch
- Idea:
  - determine the target of the call statically
  - if we can prove call has a single target method, call the target directly
- declared (static) types of pointer variables not precise enough for this, so, analyze their run-time (dynamic) types.

Client 1: Example

```
class A             { void foo() {...} }
class B extends A   { void foo() {...} }
void bar(A a)       { a.foo() }  // OK to just call B.foo?
B myB = new B();
A myA = myB;
bar(myA);
```

- Declared type of a permits a.foo() to target both A.foo and B.foo.
- Yet we know only B.foo is the target.
- What program property would reveal this fact?

Client 2: Verification of casts

- In Java, casts are checked at run time: (Foo) e translates to

```
if (! (e instanceof Foo))
    throw new ClassCastException()
```

- Java generics help readability, but still cast.
- The exception prevents any security holes, but is expensive.
- Static verification useful to catch bugs.
- Goal: prove that no exception will happen at runtime

Client 2: Example

```
class SimpleContainer { Object a;
    void put (Object o) { a=o; }
    Object get() { return a; }   }
SimpleContainer c1 = new SimpleContainer();
SimpleContainer c2 = new SimpleContainer();
c1.put(new Foo()); c2.put(``Hello'');
Foo myFoo = (Foo) c1.get(); // Check not needed
```

What property will lead to desired verification?

Client 3: Non-overlapping fields in heap

```
E = new Thing (42);
for (j = 0; j < D.len; j += 1) {
    if (E.len >= E.max)) throw new OverflowException ();
    E.data[E.len] = D.data[i];  E.len += 1;
}
```

We assign to E.len, but we don't have to fetch from D.len every time; can save in register.

Pointer Analysis

- To serve these three clients, want to understand how pointers "flow," that is, how they are copied from variable to variable.
- Interested in flow from *producers* of objects (new Foo) to *users* (myFoo.f).
- Complication: pointers may flow via the heap: a pointer may be stored in an object's field and later be read from this field.
- For simplicity, assume we are analyzing Java without reflection, so that we know all fields of an object at compile time.

Analyses

- Client 1: virtual call optimization:
  - which producer expressions new T() produced the values that may flow to receiver p (a consumer) in a call?
  - Knowing producers tells us possible dynamic types of p, and thus also the set of target methods.
- Client 2: cast verification:
  - Same, but producers include expressions (Type) p.
- Client 3: non-overlapping fields: again, same question

Flow analysis as a constant propagation

- Initially, consider only new and assignments p=r:

```
if (...) p = new T1(); else p = new T2();
r = p; r.f();  // what are possible dynamic types of r?
```

- We (conceptually) translate the program to

```
if (...) p = o₁; else p = o₂;
r = p; r.f();  // what are possible symbolic constant values r?
```

Abstract objects

- The o_i constants are called abstract objects
- an abstract object o_i stands for any and all concrete objects allocated at the *allocation site* ('new' expression) with number i.
- When the analysis says a variable p may have value o_7,
- we know p may point to any object allocated at

  $\texttt{new}_7$ `Foo()`

Flow analysis: Add pointer dereferences

```
x = new Obj();      // o1
z = new Obj();      // o2
w = x;
y = x;
y.f = z;
v = w.f;
```

- To propagate the abstract objects through p.f, must keep track of the *heap state*—where the pointers point:
  - y and w point to same object
  - z and y.f point to same object, etc.

Flow-Insensitive Analysis

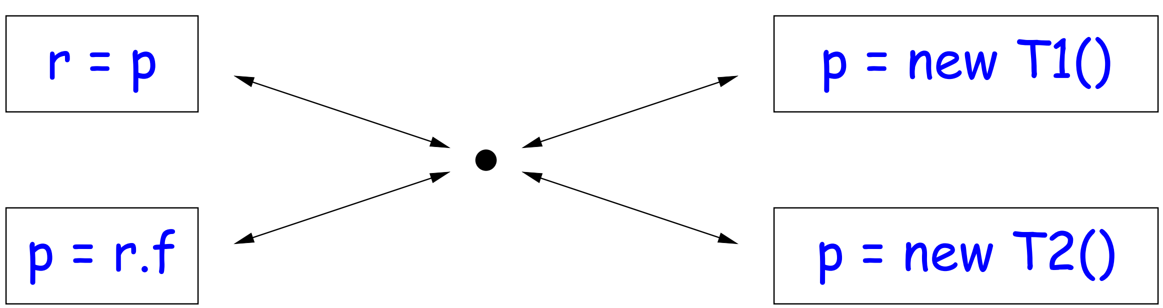
- The heap state may change at each statement, so ideally, track the heap state separately at each program point as in dataflow analysis.
- But to be scalable (i.e. practical), analyses typically don't do it.
- For example, to save space, can collapse all program points into one consequently, they keep a single heap state, and disregard the control flow of the program (*flow-insensitive* analysis):

  assume that statements can execute in any order, and any number of times

- So, flow-insensitive analysis transforms this program

```
if (...) p = new T1(); else p = new T2();
r = p; p = r.f;
```

into this CFG:



Flow-Insensitive Analysis, contd.

- Motivation: Just "version" of program state, hence less space
- Flow-insensitive analysis is *sound,* assuming we mean that *at least* all possible values of pointer from all possible executions found
- But it is generally *imprecise*:
  - In effect, adds many executions not present in the original program;
  - Does not distinguish value of p at various program points.

Canonical Statements

- Java pointers can be manipulated in complex statements, such as

  p.f().g.arr[i] = r.f.g(new Foo()).h

- To keep complexity under control, prefer a small set of *canonical statements* that accounts for everything our analysis needs to serve as intermediate representation:

```
p = new T()     new
p = r           assign
p = r.f         getfield
p.f = r         putfield
```

- Complex statements can be canonicalized

  p.f.g = r.f $\Longrightarrow$ t1 = p.f; t2 = r.f; t1.g = t2

- Can be done with a syntax-directed translation

Handling of method calls: Arguments and return values

- Translate calls into assignments. For example,

```
Object foo(T x) { return x.f }
r = new T; s = foo(r.g)
```

could translate to

```
foo_retval = x.f;
r = new T; x = r.g; s = foo_retval;
```

(have used flow-insensitivity: order irrelevant)

Handling of method calls: targets of virtual calls

- Call p.f() may call many possible methods
- To do the translation shown on previous slide, must determine what these targets are
- Suggest two simple methods:
  - Use declared type of p.
  - Check whole program to see which types are actually instantiated.

Handling of method calls: arrays

- We collapse all array elements into one.
- Represent this single element by a field named arr, so

  p.g[i] = r becomes p.g.arr = r

Andersen's Algorithm for flow-insensitive points-to analysis

- Goal: computes a binary relation between variables and abstract objects:

  o flowsTo x when abstract object o may be assigned to x.

- (Or, if you prefer, x pointsTo o.)
- Strategy: Deduce the flowsTo relation from program statements:
  - Statements are facts.
  - Analysis is a set of inference rules.
  - flowsTo relation is a set of facts inferred with analysis rules.

Statement facts

We'll write facts in the form x *predicate* y

```
p = new_i T()  ⟹ o_i new p
p = r          ⟹ r assign p
p = r.f        ⟹ r gf(f) p    (get field)
p.f = r        ⟹ r pf(f) p    (put field)
```

and apply these *inference rules:*

- Rule 1) o_i *new p* $\Rightarrow$ o_i *flowsTo* p
- Rule 2) o_i *flowsTo r* $\wedge$ *r assign p* $\Rightarrow$ o_i *flowsTo p*
- Rule 3) o_i *flowsTo a* $\wedge$ *a pf(f) p* $\wedge$ *p alias r* $\wedge$ *r gf(f) b* $\Rightarrow$ o_i *flowsTo b*
- Rule 4) o_i *flowsTo x* $\wedge$ o_i *flowsTo y* $\Rightarrow$ *x alias y*

Meaning of the results

- When the analysis infers *o flowsTo y,* what did we prove?
- Nothing useful, usually, since *o flowsTo y* does not imply that there is a program input for which o will *definitely* flow to y.
- BUT the useful results are places where analysis *does not* infer that *o flowsTo y*:
- In those cases—because the analysis assumes conservatively that o flows to y if there appears to be any possibility of that happening—we can infer that not *o flowsTo y* for all inputs.
- Same arguments apply to alias, pointsTo relations and many other static analyses in general.

Inference Example

The program:

```
x = new Foo(); // o1
z = new Bar(); // o2
w = x;
y = x;
y.f = z;
v = w.f;
```

The six facts:

o_1 *new x*

o_2 *new z*

x assign w

x assign y

z pf(f) y

w gf(f) v

Sample inferences:

o_1 *new x* $\Rightarrow$ o_1 *flowsTo x*

o_2 *new z* $\Rightarrow$ o_2 *flowsTo z*

o_1 *flowsTo x* $\wedge$ *x assign w* $\Rightarrow$ o_1 *flowsTo w*

o_1 *flowsTo x* $\wedge$ *x assign y* $\Rightarrow$ o_1 *flowsTo y*

o_1 *flowsTo y* $\wedge$ o_1 *flowsTo w* $\Rightarrow$ *y alias w*

o_2 *flowsTo z* $\wedge$ *z pf(f) y* $\wedge$ *y alias w* $\wedge$ *w gf(f) v* $\Rightarrow$ o_2 *flowsTo v*

etc.

Inference Example, contd.

- The inference must continue until no more facts can be derived; only then do we know we have performed sound analysis.
- In this example:
  - We have inferred o_2 *flowsTo v*
  - But we have *not* inferred o_1 *flowsTo v*.
  - Hence we know *v* will point only to instances of *Bar* (assuming the example contains the whole program)
  - Thus, casts (Bar) v will succeed
  - Similarly, calls v.f() are optimizable.

Prolog program for Andersen algorithm

```
new(o1,x).        % x=new_1 Foo()
new(o2,z).        % z=new_2 Bar()
assign(x,y).      % y=x
assign(x,w).      % w=x
pf(z,y,f).        % y.f=z
gf(w,v,f).        % v=w.f
flowsTo(O,X) :- new(O,X).
flowsTo(O,X) :- assign(Y,X), flowsTo(O,Y).
flowsTo(O,X) :- pf(Y,P,F), gf(R,X,F), aliasP,R), flowsTo(O,Y).
alias(X,Y)  :- flowsTo(O,X), flowsTo(O,Y).
```

- Prolog's search is too general and potentially expensive.
- Prolog program may in general backtrack (exponential time)
- Fortunately, there are better algorithms as well that operate in polynomial time.