**ECE750-T5: Static Analysis for Software Engineering** Spring 2012

# Lecture 2 — May 9, 2012

*Patrick Lam* *version 1*

**Review from last time.** We talked about presentation tips, dataflow analyses, static versus dynamic analyses, and false positives/false negatives (soundness and completeness). Key points:

- presentations: "so what?" versus "what", tailor the presentation to the audience.
- dataflow analysis: come up with a heap abstraction and transfer functions.
- static vs dynamic analysis: static analysis tells you something about all possible executions, while dynamic analysis tells you everything about one specific execution.
- false positives/negatives: static analysis is prone to false negatives, which can be mitigated with better abstractions; dynamic analysis is prone to false positives along unobserved paths.

**Useful resources.** I kept the discussion of dataflow analysis at a very high level last time. You can refer to the PLDI 2003 tutorial at `http://www.sable.mcgill.ca/soot` to find detailed instructions for using Soot's dataflow framework and pointer analysis. Basically, you program the parts I described, and then Soot does the rest of the work of implementing the dataflow analysis. JavaCOP, which I'll describe later, also has a dataflow framework. Ask me for help if you need it.

Another good resource is the document on "Static Program Analysis" by Anders Møller and Michael I. Schwartzbach. I've linked to it from the course webpage.

## Finitization and Unrolling

The two main (related) reasons that the static analysis of programs requires approximations are 1) user input and 2) loops. We'll consider loops here. Static analysis approximates loops by assuming that any number of loop iterations could occur. Instead, one can unroll a loop a finite number of times and check that program properties hold on the unrolled program, avoiding the use of the merge operator on the loop edge.

```
void foo(int N) {
  double j = 1.0;
  for (int i = 1; i < N; i++)
    j /= i;
}
```

$\Longrightarrow$

```
void fooUnrolled(int N) {
  double j = 1.0;
  j /= 1;
  j /= 2;
  j /= 3;
}
```

Unrolling is particularly useful for model checking; one can check that the program behaves properly as long as it it iterates fewer than $N$ times.

## Type Systems

Most modern programming languages type-check programs to ensure that operands to operations are of appropriate types. For instance, the following line will trigger a compile-time error in sensible languages[1].

```
print (5 - "Hello");
```

In this part of the lecture, I'll first present standard type system notation (which will come up in the papers that you read) and then describe the JavaCOP [MME$^+$10] system for implementing your own pluggable type systems. Such a system could be a suitable course project, if you can think of a sufficiently interesting type system.

**Type system notation.** A type system attempts to assign types to program fragments, starting from the bottom up. Type rules declare conditions under which type assignments are allowed. Here is a simple type rule defining the type of integer constants.

$$\text{[CONST]}\quad \frac{n \in \mathbb{N}}{n : \mathsf{int}}$$

This type rule states that any integer $n$ has type int. We can then allow the addition of two integers:

$$\text{[ADD]}\quad \frac{n : \mathsf{int} \quad m : \mathsf{int}}{n + m : \mathsf{int}}$$

This says that if $n$ and $m$ are both int, then so is the expression $n + m$.

You can define types for basic expressions this way. But to typecheck interesting programs, we need the concept of an *environment*, which stores types for variables. Consider a program $P$ and an environment $E$. Then we can introduce the following rule:

$$\text{[EXP VAR READ]}\quad \frac{E = E_1, \tau\ y, E_2}{P; E \vdash y : \tau}$$

That is, if the environment states that $y$ has type $\tau$, then we can assign type $\tau$ to $y$. Of course, we also need to introduce types for variables into the environment:

$$\text{[METHOD]}\quad \frac{\begin{array}{c} arg_i = cn_i\ vn_i \\ \tau_j = cn_j \quad lv_j = \tau_j\, ln_j \\ E = arg_{0..n}, lv_{n+1..n+l} \\ \forall i \in [1..t].\ P; E \vdash s_i \end{array}}{P; E \vdash mn(arg_{1..n})\ \{s_{1..t}\}}$$

This rule allows the type system to verify a method $mn$ as long as each of the statements $s_i$ in the method type checks. It populates $E$ with the method arguments and local variables.

[1]For a good time, try it in PHP.

**Type systems: the bigger picture.** A type system attempts to prove that a program is well-typed by constructing a type judgment for the program as a whole. This will only succeed if each part of the program is well-typed.

Formally, the type system will guarantee that well-typed programs will never get stuck, in the sense that the operational semantics of the program will always be able to make progress. This is useful because typical operational semantics will omit error transitions, preventing the program from doing something nonsensical (e.g. subtracting an integer from a string.)

## JavaCOP

The JavaCOP system allows you to experiment with your own type system extensions, so that you can actually define your own type rules and then check that (possibly-annotated) programs respect these rules.

While the type systems above completely define all allowable expressions in a language (a default-deny discipline), JavaCOP also allows you to state that certain otherwise well-typed Java expressions don't type-check (more like default-allow). For example, you can define a `@NonNull` type which will not accept values that might be `null`. For instance, this would not type-check:

```
@NonNull String x = null;
```

**Annotations.** Non-prehistoric versions of Java include user-defined annotations. Developers can therefore define arbitrary annotations, like `@NonNull`, and use these annotations to augment the program's types.

**Rules.** JavaCOP type systems are based around type rules. Here's an example from their paper:

```
rule checkNonNull(Assign a) {
  where(requiresNonNull(a.lhs)) {
    require(definitelyNotNull(a.rhs)):
      error(a, "Possible null assignment to @NonNull");
  } }
```

This rule states that an assignment where the left-hand side is marked non-null must also have a right-hand side which is not null. These rules depend on predicates.

**Predicates.** Because the same conditions tend to occur in many different rules, JavaCOP allows its users to refactor such conditions by using predicates.

```
declare requiresNonNull(JCTree t) {
  require (t.holdsSymbol && t.getSymbol.hasAnnotation("NonNull"));
}
```

In the event that there are multiple applicable predicate definitions, it suffices that one of them evaluates to `true`. The `definitelyNotNull` predicate also has a definition in the paper.

**Redefining subtyping.** Instead of manually redefining every check in the AST as we've done above, we can move to a higher level and redefine the subtype check throughout the AST.

```
rule checkNonNull(node <<: sym) {
  where(requiresNonNull(sym)) {
    require(definitelyNotNull(node)):
      error(a, "Possible null expression "+node+" used as @NonNull");
  } }
```

There are many JavaCOP features that I won't present here. See the paper for full details. I'll note, in passing, that JavaCOP also supports dataflow analysis, which enables you to incorporate more detailed information than a type system would normally have. In particular, dataflow analysis enables you to verify conditions along paths in the control-flow of a method.

**Exercise.** If you want to investigate JavaCOP, a good introductory exercise would be to write a type system that includes subtypes for positive, negative, and zero `ints`.

## Interprocedural Analyses

I'll continue with a brief introduction to interprocedural analysis. My philosophy is that if one can get good-enough results without having to go interprocedural, that's a win. We'll talk about several ways of reasoning about programs beyond method boundaries.

### Call Graphs (and Pointer Analysis)

A call graph is just a graph where the nodes are methods (or statements) and the edges represent calls and returns. Let's look at a simple example:

```
void main() { bar(); fob(); }
void foo() { bar(); }
void bar() { }
void fob() { if (...) fob(); }
```

If we assume that we have the whole program, then we can omit method `foo()` from the call graph. Complications arise when computing call graphs for OO programs, but also for programs with function pointers. We'll only talk about OO call graphs here.

Dynamic dispatch—where the identity of the callee depends on the run-type type of the receiver object—is why it is hard to build a call graph for an OO language. Good call graphs require good pointer information.

**Pointer analysis.** I'll say more about pointer analysis later on. Right now, what's important to know is that we use pointer analysis to get good information about the run-time type of objects. How does that work?

A key challenge in pointer analysis is finitizing the program heap, which is potentially unbounded. Note that, for instance, the number of object allocation sites in a program ("new" expressions) is finite, and every object must have been allocated at some new expression.

```
S x = new S();
T y = new T();
// note: x != y
f(x);

void f(Object a) { a.toString(); }
```

Here, we can see that resolving the call to `a.toString()` requires information on the run-time type of `a`. Knowing that `f()` only gets called with an `S` object allows us to refine the call graph. Note also that refining the call graph also allows us to improve the pointer information.

**Constraint-based pointer analysis.** Let's look at an example run of a pointer analysis from Møller and Schwartzbach. This analysis is based on propagating constraints; it's also known as an Andersen-style analysis. We'll switch to a C-like language here.

```
var p,q,x,y,z;
p = malloc;     //  $m1 \subseteq [p]$
x = y;          //  $[y] \subseteq [x]$
x = z;          //  $[z] \subseteq [y]$
*p = z;         //  $\&y \in [p] \Rightarrow [z] \subseteq [y],\ \&z \in [p] \Rightarrow [z] \subseteq [z]$
p = q;          //  $[q] \subseteq [p]$
q = &y;         //  $\{\&y\} \subseteq [q]$
x = *p;         //  $\&y \in [p] \Rightarrow [y] \subseteq [x],\ \&z \in [p] \Rightarrow [z] \subseteq [x]$
p = &z;         //  $\{\&z\} \subseteq [p]$
```

Solving these constraints gives:

$$[p] = \{m1, \&y, \&z\}; \qquad [q] = \{\&y\}$$

In this course, we won't talk much about how to actually get pointer analysis results, but rather about how to use them. Note also that the two common pointer analysis queries are:

- may $p$ and $q$ point to the same object?
- must $p$ and $q$ point to the same object?

## Summary-Based Approaches

Given a call graph, we can reason about the effects of callees using method summaries.

**Assume/guarantee reasoning.** Given an abstraction, three pieces of information typically suffice to describe a method's interactions with the abstraction: requires, ensures clauses and a frame condition. Here's an example from my thesis.

```
proc removeFirst() returns e : Entry
  requires card(Content)>=1
  modifies Content
  ensures (Content' = Content - e) & card(e)=1 & (e in Content);
```

- The *requires* clause encapsulates the abstract program state that the method may expect to hold upon entry; it is the responsibility of the caller to establish the necessary precondition. Static and dynamic analyses can report errors when preconditions don't hold and attribute the errors to the caller.

  The above example says that `removeFirst()` assumes that the set `Content` has successfully been checked for non-emptiness.

- The *modifies* clause is not strictly necessary, but allows the caller to assume that everything that does not appear in a modifies clause stays the same. Without the modifies clause, the caller must make a worst-case assumption and throw out all information it may have about the program state not mentioned in the requires clause. Separation logic is an elegant notation for obviating the need for modifies clauses.

  The example modifies clause states that only `Content` changes after a call to `removeFirst()`.

- The *ensures* clause expresses constraints on the abstract program state guaranteed by the method upon exit. The caller may assume any conditions expressed in the ensures clause. Analysis systems blame the callee for failing to ensure its stated postconditions.

  The example requires clause states that the returned value is no longer in the new `Content'` set upon exit from `removeFirst()`, but that it previously belonged to the old `Content` set.

**Inferring summaries.** These clauses are a lot to ask of the developer, so one might try to automatically infer summaries. Postconditions (including modifies clauses) can be particularly tractable via static analysis; dynamic analysis helps find preconditions.

## Interprocedural dataflow analysis

An alternative to using a call graph and separate control-flow graphs for each method is to create a single interprocedural control-flow graph, or ICFG. At a method call, the ICFG contains the call graph edges. One can then perform dataflow analysis on the ICFG. Because this graph is quite large, dynamic programming is required. The dataflow analysis algorithm on ICFGs is IFDS.

## References

[MME$^+$10] Shane Markstrum, Daniel Merino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems*, 32(2):Article 4, January 2010.