Lecture #43: Topics in Static Analysis, Contd.

• We looked into using it to check correctness, but static analyses also useful in program optimization.

• **Problem:** Consider this loop in Java:

  ```java
  for (i = 1; i < A.length; i += 1)
      A[i] += A[i-1];
  ```

• Java guarantees that array bounds are never violated, as if we had written:

  ```java
  for (i = 1; i < A.length; i += 1)
      if (i < 0 || i >= A.length || i-1 < 0 || i-1 >= A.length)
          throw new ArrayIndexOutOfBoundsException();
      else
          A[i] += A[i-1];
  ```

• But conditionals in a tight loop like this are not good on some architectures at least.
Eliminating Array-Bounds Checks

- Let's rewrite our problem a little:

```java
i = 1
while (i < A.length) {
    assert i ≥ 0 ∧ i < A.length ∧ i - 1 ≥ 0 ∧ i - 1 < A.length;
    A[i] += A[i-1];
    i = i+1
}
```

- (Where this `assert` has teeth—blows up if false).

- All we know is that at top of loop, `i < A.length`, and (from semantics of Java) that `A.length` >= 0.

- See if assertion simplifies to true:

```plaintext
0 ≤ A.length ∧ i < A.length
⇒ i ≥ 0 ∧ i < A.length ∧ i - 1 ≥ 0 ∧ i - 1 < A.length
≡
i - 1 ≥ 0
```

- So not quite. We don’t need to test against `A.length`, but the test `i - 1 ≥ 0` remains.
So, Is It True Anyway?

- So now we have
  
i = 1
  while (i < A.length) {
    assert $i - 1 \geq 0$;
    A[i] += A[i-1];
    i = i+1
  }

- Can we show the assertion to be an invariant?

- Pretty obvious, really. Must show
  
  $\{\text{true}\} \ i = 1 \ \{i - 1 \geq 0\}$, and
  
  $\{i - 1 \geq 0\} \ A[i] += A[i-1]; \ i = i+1 \ \{i - 1 \geq 0\}$

- We haven't given a postcondition for the loop, so just take it to be \text{true}, making the usual exit condition for the \textbf{while} rule trivial.

- The assertions above can be verified mechanically.

- So the \textbf{assert} is proved, and the check is unnecessary.
Java Byte-Code Verification

- Java .class files essentially contain machine code (called bytecode) for a standard virtual machine, implementable by interpreter or byte-code to machine-code compiler.

- Programs may be transmitted as byte codes; no need for source.

- Java's security features (are supposed to) allow one to run untrusted code, and be assured that it is limited in what it can do.

- But Java runtime system's integrity, including these security features, depends on the bytecode's behaving themselves, not writing to invalid locations, etc.

- How do we know a class file downloaded from the net behaves?

- Java system verifies byte-code files prior to loading them. How?
Things to Verify

• Class files must have right format, instructions must be properly formed. Easily checked.
• Jumps must not go beyond bounds of method; method must terminate in return. Also easy.
• Instance variable and instance method references must be selected from proper types.
• Argument types to calls and return types must be valid.
• Assignments to instance variables must have valid types.
• Access privileges observed, exceptions properly declared.
• These things can be verified if we can tell the types in local variables and on the stack.
Validating Types

- Java virtual machine is a stack machine with registers (any number) for local variables and compiler temporaries.
- Type information comes from information in the class files about types of instance variables, types of formal parameters, and return types of methods.
- First $N$ local variables initially contain `this` and parameters; their types are known.
- But nothing explicit in the class files about types on the stack, or in other local variables.
- So we use a form of abstract interpretation, execute program, but keep type information instead of actual values.
Basic Blocks

• Easy enough to divide method’s bytecode into basic blocks.
• Within one basic block, simulate the stack and locals.
• Suppose we know initially that stack is empty and local variables v0, v1, and v2 have types List, int, and String.

• Then after executing
  
  push v0
  push v1
  call List.get(int)->Object

  cast->String
  store v3
  push v3
  etc.

• We know types on stack and in locals at each point. All necessary checks possible.
Joins

• This leaves the problem of what to do when a basic block has more than one predecessor. What’s in its initial starting stack and variables?

• Basic idea is to join the data.

• Example:

  BLOCK1:
  ...
  jump BLOCK3  # stack: String, int, int; v3: ArrayList

  BLOCK2:
  ...
  # stack: Object, Object, int; v3: String

  BLOCK3:  # stack: ? v3: ?

• We require that the two stacks have the same depth.

• We can “join” types of corresponding stack elements and of v3’s to get something that is valid however we get to BLOCK3:

  BLOCK3:  # stack: Object, int, int; v3: Object

• It’s OK to treat anything as an int, since that never breaks anything.
Analysis Challenges

• The perennial problem: are x.y and z.y the same variable?
• Impact on whether one has to re-fetch z.y, or whether one can execute pieces of a program in parallel, if one accesses x.y and the other accesses z.y
• Similar problem in arrays: can A[i] and A[j] access the same location?
• Impact on parallelization, vectorization, and many optimizations.
• In a language where pointer x might access storage on another processor, need to know if x really is just a local pointer. Impact on speed.
• Storage analysis: is this storage location now garbage? Idea is to cut down on garbage collection time by doing some of it statically.