Thursday, January 24, 2019

E6998: Program Analysis + ML

Introduction

Nathan Reitinger

I. **logistics**
   A. not a general purpose ML class
   B. goal is to look at ML techniques in new domain
      1. most ML progress comes from vision, text, robotics, speech recognition
         a) really hard to write logic, hard to write programs that can react to environment in these settings
      2. apply ML (or know more about it) to program analysis
         a) program analysis used formal logic initially
         b) focus on formal logic part that is used for program analysis
            (1) what is program analysis
            (2) what are existing techniques for program analysis
            (3) why ML can help
   C. this originally was a seminar course
      1. first 5-6 lecture, will be traditional lecture
      2. then you are expected to read paper and do presentation
         a) individually or in groups of 2
            (1) read 2-3 papers in a particular area and then summarize them
      3. project
         a) pick whatever project you want
            (1) if it is single person project then the expectation is lower
         b) project doesn't have to be programming heavy
            (1) it should be a new approach to program analysis
         c) he expects a pretty thorough study of literature
            (1) summarize why approach is good
            (2) then do small experiment to check feasibility
      4. class participation is standard
         a) appreciate scribes
            (1) a little extra credit (if you do it, he grades it, as long as you don't screw up—2-3 pages)
               (a) should connect the "big vision" between papers
   D. there will be a TA
   E. doesn't plan to use up entire lecture period, so don't be worried if we finish early —3:20 may be a standard finishing time

II. **program analysis** (*takes program and takes property*[1])
   A. given program (language doesn't matter) ask questions about program behavior (answer questions about behavior)

[1] x is greater/lesser/equal to 0

1. what does this program do
2. what are the basic primitives (mov, pop, push)
   a) arithmetic operations (add, subtract, divide, multiply)
   b) control[2] flow

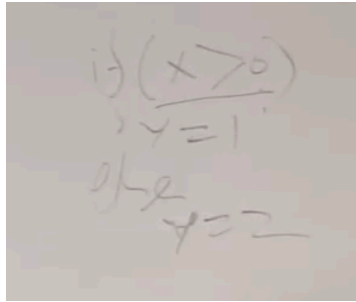
B. what are the instruction classes
   1. control flow (where execution flow will go)
   2. update variables (i.e., updating value at physical address)
      a) state: set of all variables of program

C. program analysis questions
   1. given program, will this program output "1"
   2. whether variable in program can ever be set to "0" across all executions
      a) is there a path through which an intermediate variable can be set to *x*
   3. how many paths are there through which a variable will be >0
   4. ask about some property of program
   5. build analyzer (another program) which takes target program in and answers these questions

D. why is program analysis hard
   1. abuts halting problem (undecidability)
      a) turing machine: abstraction which summarizes all sets that can be computed
      b) halting problem says can there be a turing machine that can take any program and tell whether it will "halt" for all inputs to that program
         (1) we are not interested in value of variable, but interested in "time" across all paths
      c) answer is no. because the question asks if it can be done for *all* programs (all programs written for turing machine and all inputs)
         (1) proof by contradiction: let's imagine there is such a program (can tell halt or no halt); then we can also say that the target program can discover this introspection and add a while(1) loop
      d) but even if you assume well-behaved program, the analysis is exponential (path explosion) in the size of the program and the size of the input
         (1) why is it exponential
            (a) decision point: control flow branch
            (b) if you stack branches, the spread is exponential
               i) nested if-else
               ii) while loop == toast
                  (1) the program can loop an arbitrary number of times
   2. you want to check a property across all paths, *all* paths is the hardness
      a) for most useful programs, they have exponential paths through program
         (1) a path is a single execution (you cannot take all directions on one path)

---

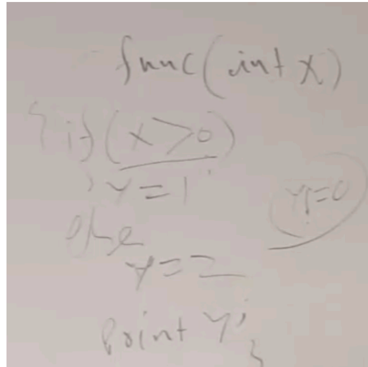
[2] sequence in which program is executed

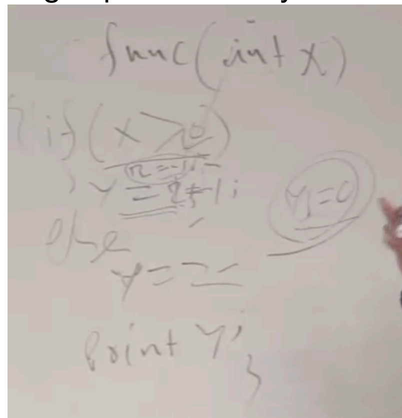
(2) 

E. why study program analysis
  1. bug finding
     a) [my own example] having a guarantee on security (this outcome will never happen in the program) — i.e., formal methods
  2. optimization (kill unused branches)
     a) you may find automated optimization — if variable never used, chop it off
     b) keep highly used variables in registers
        (1) compiler will check on unreachable code (comments, unused parts of libraries) and remove it for optimization
            (a) hardness of this approach
                i) if you execute all possible paths to learn information about the program, then you will never have false negatives or false positives
                   (1) you can't get both to zero with exponentially growing control flow
                ii) tradeoff: because exponential time is there, we can only play with the false positive/negative
  3. privacy
     a) given application from app store. and it asks for [my example] HIV status. how do you know it won't leak that information to someone else who you don't trust
        (1) if the data is copied as is and leaked, it is easy to detect because you know your HIV status (you know what this information is)
        (2) but a smart program would not release that data in the raw
            (a) so how do you know it is leaking if it perturbed the sensitive input
                i) can the sensitive input affect any of the branching points
                   (1) if it affects a branching point then it is leaking information
  4. auto-fix program
     a) what if there was a null pointer found, you could auto-find these instances and make sure that the pointer never gets set to null in all paths
        (1) knuth insight: "Beware of bugs in the above code; I have only proved it correct, not tried it"

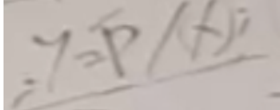
III. **how to solve the exploding paths problem**

A. other ideas
  1. this is the program

2. 

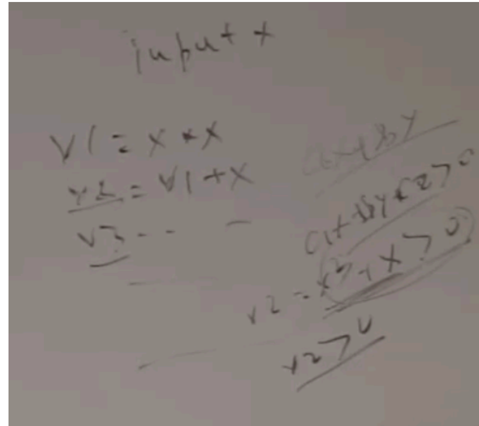
a) we want to ensure that *y* is never set to 0
   (1) **backward execution**—just check wherever *y* is set and see if it is ever set to 0
      (a) bad because it is hard to find concrete input that causes you to set *y* to 0
         i) example: what if y was set to variable z earlier, now we have a large space to analyze as well
         ii) 

      (b) bad because you may find a way that z is set to 0 and therefore y is set to 0, but for other reasons z could never be set to 0
         i) so in the example above, perhaps z is set to -1 at some point in time (which would set y to 0) but this does not mean that `func(int x)` will ever reach the point where y is then set to 0
            (1) to summarize: we can find a situation where our target variable (y) would turn to 0 if z was -1, and it is possible that z could be -1, but z can never be -1 in a way that y would then be 0
   (2) **probabilistic testing**: can answer question with certain probability, and not certainty
      (a) let's just say we test a random amount of times to pick x (between range of $1-2^{64}$), but this is still problematic because the value of x only needs one value to trigger y as 0

- i) just because a bridge collapsed in one scenario, you can say that it likely won't happen—but we can't guarantee anything with this type of strategy
- (b) smart selection of inputs—pick inputs more likely to hit the bugs
  - i) try to bias distribution so that different paths are likely to be taken for different inputs
  - ii) problem here is biasing distribution is hard
    - (1) 
    - (2) biasing x in such a way that you hit different paths is hard (trying to figure out when x is 0 is hard)
      - (a) boolean satisfiability problem: imagine you have boolean variables (true or false) and a formula that connects variables using <and or not>. for a given formula what values should those variables be assigned to make the formula true. in some cases there may be no answer. this problem is NP hard, so it is exponential.

- B. why ML is a good fit for program analysis
  - 1. in a lot of cases, bugs are similar to each other (different programs have similar with code the same bug)
    - a) not all sections of code are the same
    - b) there is no way past exponential paths
    - c) this is like a heuristic
    - d) ML might be able to reason about the code base faster than executing each instruction
      - (1) with static analysis, if you don't find a bug that doesn't mean there is no bug
      - (2) with ML, dynamic analysis, you can search for bugs specifically and therefore narrow the search space smartly to get over the exploded path problem
        - (a) at which level to use ML? an open question

- IV. **simple example of program analysis**
  - A. narrow the space
    - 1. restrict the things that programs can do
      - a) example: no branches—straight line program
        - (1) side effect: not very useful
        - (2) language can only do mathematical operations, and has no memory beyond variables (functions cannot take jumps, it can only move linearly)
        - (3) how would you analyze this
          - (a) question: can a variable be greater or lesser than something
          - (b) answer: you can use linear programming techniques to solve them

i) 

b) example: $v2 = x^2 + x = 0$
  (1) we can try to solve for x, as long as the exponential is not more than 5
    (a) we can use gradient descent[3] to minimize the value of x to see how close we can get to 0
    (b) assumes x is differentiable
      i) x may not be a valid value, what if x is 4.3 (but x is an int)
        (1) you could try and round x, but the issue there is that you may not reach a valid solution

B. say that you are willing to accept false positives
  1. instead of operating on point values, you can keep track of region
    a) you can just keep lower and upper bound of variable
      (1) so define the operations on intervals
      (2) if you had input of [1, 2^64]
      (3) then the first v1 is = x*x so that is [1, 2^64] * [1, 2^64]
      (4) then at the end you can check whether 0 is in your bound
        (a) issue: you can guarantee that there is no violation, but not the reverse (that there is a violation doesn't mean there really is because it might never occur in practice)
        (b) solution: abstract interpretation, keep a fixed number of bounds (split the problem up)

V. **static versus dynamic**

A. dynamic: execute a program—one path at at time
    a) execution: follow each instruction along a path
      (1) if you are following each path one at a time, it is dynamic
      (2) suffers from exponential paths

B. static: there is some way of coalescing the effects of different paths
  1. reasoning about group of paths somehow
  2. you have too many paths, so we have to group them somehow

---

[3] calculate the gradient and decide which next step you want to try based on the gradient