Static Program Analysis Book Outline

1. Chapter 1 🡪 Introduction
   1. Main Idea
   2. Section Ideas
      1. Why do you use static analysis?
         1. Finding program optimization possibilities
         2. Proving correctness of programs
         3. Providing developer support in IDE’s, making sure the coder can use certain functions/variables when they attempt to
      2. The need for approximation
         1. Rice (1953) states that any interesting feature/behavior of a program is undecidable
            1. Shown to be true by contradiction proof involving a hypothetical machine which would be able to solve the halting problem if said feature was known before and after running
         2. Thus, we can only take stabs at parts of a program, and this leaves room for making better approximations as methods become more intricate
         3. We use conservative approximations so that we always have a possible margin of error, and do not allow for high levels of false positives in the rejection schema
         4. Program Soundness
            1. An analyzer is sound if it never gives an incorrect result
            2. If a verification tool never misses an error it is designed to detect, but also gives off false positives
            3. An automated testing tool in which all reported errors are genuine, but it might not find all errors in the program
   3. Examples for Mastery
   4. Remaining Questions about Material
      1. What happened in 1.3 with the Universal Turing Machine part…
2. Chapter 2 🡪 A Tiny Imperative Programming Language
   1. Main Idea
      1. Setting up a small, contained language to use as an introductory testing ground for analysis techniques
      2. By partitioning parts of TIP, we can focus on the use cases of distinct types of analysis techniques
   2. Section Ideas
      1. Basic Syntax
         1. Int values, variables, expressions, and input from user
         2. Statements allowed -> var = expressions, output expression, dual statements, if else, while
         3. Functions -> name, param list followed by local variable definition, statements, and return expression
            1. Function calls are technically a new subtype of the general expression from above
         4. Records 🡪 collection of fields with name/value pairs
            1. Immutable for simplicity of language
         5. Pointers
            1. Allocate memory in heap
            2. Create a pointer to that allocation
            3. Dereferences the memory at a pointer
            4. Null Value
            5. Requires addition of pointer assignment statement (\*X = E)
         6. Functions can be called as first class values, further extending the expression definition to E -> E(E,…,E) which is not at all confusing notation for anyone…
            1. You can have a function named as an expression, whose parameters are all expressions
            2. This distinction allows for dealing with the main challenges regarding methods and other higher order functions in language use
         7. Programs are now just a collection of functions
            1. Complete program requires a main function as initiator
            2. Inputs are supplied at the beginning of the input stream, and the output is a value appended on the output stream
      2. Normalization
         1. Transformation of complicated programs into syntactically simpler ones using fresh variable names to disambiguate parts
            1. Using an increasing number scheme, can make all distinct fresh variable names unique (Problem 2.5)
      3. Abstract Syntax Trees
         1. A representation that allows for flow-insensitive analysis including type analysis, control flow analysis (ch9), and pointer analysis (ch10)
            1. Will only be covering type analysis for upcoming thurs meeting (6/4), revisit this bullet on future chapters
      4. Control Flow Graphs
         1. Representation which allows for flow sensitive analysis
            1. Order of statements matters in deciding specific things about the program
            2. Used in dataflow analysis (ch5)
            3. Multiple function cfg’s in ch8/9
         2. Directed graph, assumed to have single entry and single exit point
         3. In a fully normalized program, each individual node represents a single distinct statement
   3. Examples for Mastery
      1. Not exactly mastery, but they left out less than, greater or equal and less or equal from the basic expressions list, do those get assumed given the sole inclusions of greater than and equivalence?
      2. There are no Boolean operations, even though they are defining 0 as false and all other int values as true. Adding in bool operations to the expressions list would be a worthwhile extension
         1. Oh hey, look at that, exercise 3.18 mentions the lack of bools
      3. Ex 2.7 The CFG of a do while would have a check at the end of the instruction set, instead of the beginning like the while loop
   4. Remaining Questions about Material
      1. 2.3 Normalize x = (\*\*f)(g()+H())
         1. T1 = g() + h()
         2. \*t2 = \*f
         3. \*t3 = t2
         4. X = t3(t1)
      2. Normalizing double pointer equations (example 2.4)
         1. \*\*x = \*\*y
         2. \*t1 = y
         3. \*t2 = t1
         4. \*t3 = x
         5. \*t4 = t3
         6. T4 = t2
         7. OR
         8. T1 = \*y
         9. T2 = \*t1
         10. T3 = \*x
         11. T4 = \*t3
         12. T4 = t2?
3. Chapter 3 🡪 Type Analysis
   1. Main Idea
      1. Useful approximation tool to allow for basic correctness checking in a program
      2. Construction of type constraints serves as a conservative approximation for checking a program for type errors as defined in the base language
   2. Section Ideas
      1. Main Types in TIP
         1. Int, pointer, function
            1. Function is defined as collection of types for params outputting a single type
         2. Free type variables alpha, which are the most general representation of a var which has a non-required type
      2. Constraint Variables and Type Constraints
         1. For every local variable, function parameter and function name X, we use a type variable [X], and for every non identifier expression we use a [E]
         2. E would be referring to a full node in a AST, not the specific syntax
         3. See page 20 for a full list of TIP variable and expression type constraints
         4. Subterm constraints
            1. For a given term constructor, the subterms must all match
      3. Unification Algorithm
         1. Using the “familiar” union find data structure (disjoint set data structure) with every node having only one edge to its parent node.
            1. If a node points to itself after unification, it is called a root
         2. There are three main operations before the unify function can be used
            1. MakeSet(x) – creates a singleton node with the parent pointing back to itself
            2. Find(X) -- Finds the type of a node by recursively following the parent chain
            3. Union(x,y) – finds the type of nodes x and y, and makes y the parent of x unless they are equiv types
         3. The unify algo makes specific care to always have a definied type be the second entry into any union call, so that all nodes will always have their parent defined as a proper type, or a link closer to a proper type
         4. See GitHub Unification Algo pictures for working out the solutions to the constraint equations for short function on pages 20-21
      4. Record Types
         1. Adds a new type to the existing list, which has individual fields with their own name and type inside the record
            1. Different field names define different record types
         2. To compare record types using the unify algo from before, we make sure that every single record has every single field present in the program, even if most of those fields would be useless in a specific record call. Since we have the free type variable alpha, we can just say that a record field has a type of alpha if it doesn’t make a difference in a unify
      5. Limitations of Type Analysis
         1. This method described above is flow insensitive, it doesn’t take into account the order of statements given that the type constraints are derived from an unordered AST
         2. It also runs into problems with polymorphic types, and recursion can cause difficulties as well
            1. If a function is polymorphic and recursive, type analysis becomes undecidable in the general case
   3. Examples for Mastery
      1. Ex 3.5
         1. ((int)🡪int) 🡪 ( (int,int)🡪int )
         2. Function which takes in param (int)🡪 int
            1. Let the input param be of the form f(x) = y
         3. Outputs value ( (int,int)🡪int)
            1. The output would be of the form R(y,y) = z
         4. Overall function would be Q(f(x)) = Z where
            1. q is defined as

q (f (x)) = return R(f(x),f(x))

* + - * 1. r is defined as

r(w(v),s(t)) = return w(v)+s(t)

* + 1. GitHub Pictures of unify type analysis on Short function
  1. Remaining Questions about Material

1. Chapter 4 🡪 Lattice Theory
   1. Main Idea
      1. Mathematical basis for static analysis
      2. Construction of an abstract domain of representations of what we are trying to analyze, while allowing for approximation methods by including terms like empty set (for any term which doesn’t follow our requirements) or full set over the domain (for any term which we can not definitively put into a sub category, allowing us to maintain soundness via giving false positives when a program fails due to a lack of strict adherence, or imperfect precision
   2. Section Ideas
      1. Lattices
         1. Defined as a partial ordering of a set, a lattice is a combination between a set and a defined partial order of that set
         2. 3 Properties of a partial order
            1. Reflexivity 🡪 V x € S: x [= x

for all x in s, x is a safe approximation of x

* + - * 1. Transitivity -> V x,y,z € S: x [= y ^ y [= z 🡪 x [= z

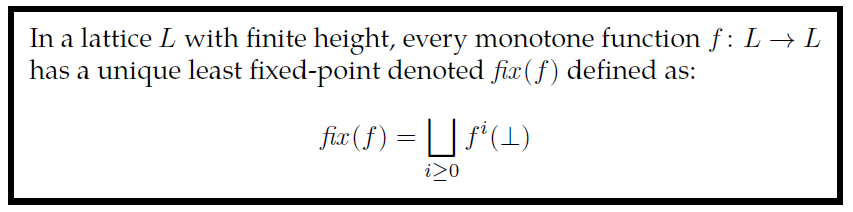
If y is a safe approx. of x, and z is a safe approx. of y, then z is a safe approx. of x

* + - * 1. Anti-symmetry

For all x,y, if y is a safe approx. of x, and x is a safe approx. of y, x = y

* + - 1. Safe approximations are stating that one set is at least as precise as another, but could be slightly more
      2. Upper Bounds
         1. For all x in X where X is a subset of S, if y is a safe approx. of x, then y is an upper bound (not unique however)
         2. There can be a lowest value for Y (unique) which is called the least upper bound (VERY IMPORTANT IDEA)
         3. The same is true in reverse for lower bounds and greatest lower bound
      3. Height is defined as the longest path from null to full set (ie bottom to top)
    1. Lattice Construction
       1. Powerset lattice is when given a set A, you construct a lattice with the top being A, and the bottom being Null set, and then you enumerate all connective pathways between them where the first row is the individual elements of the set, and subsequent rows are the combinations of those elements (height is cardinality of A)
       2. If A is a set, then flat(A) is the lattice where the top row is T, bottom row is Null set, and the middle row is all individ elems of A (height is 2)
       3. Map Lattices
          1. If A is a Set, and L is a lattice, then we can get a map lattice consisting of all the functions which map A to L, which maintains the ordering of f [= g iff V ai € A: f(ai) [= g(ai)

A function G is a safe approx. of a function f, if and only if for all elements of set A, g(ai) is a safe approximation for f(ai)

* + 1. Equations, Monotonicity, and Fixed Points
       1. Constraint systems
          1. Describe a program with a single variable representing each program variable at every line of the program
          2. You can set up a system which informs you about changes in the variable state as it maps to the given lattice
          3. Since each constraint equation is dependent on the last (in this example) it can be solved by simple substitution, and is also flow sensitive
       2. Monotone
          1. Order preserving 🡪 if y is a safe approximation of x, then f(y) is a safe approximation of f(x)
          2. More precise input does not result in less precise output
       3. Fixed point 🡪 when f(x) = x
          1. Least fixed point 🡪 X is a “least fixed point” if y is a good approximation of x for every fixed point x in f
       4. Fixed Point Theorem:
       5. Naïve Fixed Point Algo
          1. Set x to empty set, while x isn’t f(x), set x to f(x), then return x
          2. This will scale the lattice until it reaches a fixed point, which will be guaranteed if the lattice is finite
          3. Doesn’t exploit special lattice structures
       6. Least Fixed Point will be the most precise solution to the equation system, but will still be a conservative estimate.
          1. I.e. the most semantically precise possible will be below the LFP
  1. Examples for Mastery
     1. 4.8 foo,bar,baz reverse lattice construction in notebook
  2. Remaining Questions about Material
     1. 4.4 right graph
        1. Not a lattice because there aren’t unique lowest upper bounds for certain subsets?

1. Chapter 5 🡪 Dataflow Analysis with Monotone Frameworks
   1. Main Idea
      1. We can represent a program with a CFG and a finite height lattice describing the abstract info we want to assign to the nodes of the cfg
      2. Every cfg node gets assigned a variable which will range over the lattice
      3. We relate nodes to other nodes depending on program construction
         1. Sometimes we need to relate via pred nodes, sometimes succ nodes
      4. If the functions we set up to relate nodes to lattice are all monotone, we can use the fixed point algo to compute a unique least solution
      5. Analysis is considered sound if all solutions to the constraints correspond to correct info
         1. Could be imprecise, but is not wrong (ie, a node (a = 5) is positive, or any, but never defined as negative in our sign analysis
   2. Section Ideas
      1. Sign Analysis Revisited
         1. Using Sign lattice, we want to attach a abstract value to each program variable, so we define a map lattice that takes the set of program vars and maps it onto the sign lattice
         2. Join function for this example
   3. Examples for Mastery
      1. 5.8
         1. (2>0) == 1 🡪 [(2>0)] == 1 since op>(+,0) is 1, and 1 == 1 evals to 1 as the final ans
         2. X-x would eval to -0
         3. ½ would eval to +0
   4. Remaining Questions about Material
2. Chapter 6 🡪Widening
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
3. Chapter 7 🡪 Path Sensitivity and Relational Analysis
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
4. Chapter 8 🡪 Interprocedural Analysis
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
5. Chapter 9 🡪 Control Flow Analysis
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
6. Chapter 10 🡪 Pointer Analysis
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
7. Chapter 11 🡪 Abstract Interpretation
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material