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
   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. Remaining Questions about Material
     1. Recursive typing and the Mu operator, the use of the dot operator in this context

1. Chapter 4 🡪 Lattice Theory
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
2. Chapter 5 🡪 Dataflow Analysis with Monotone Frameworks
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
3. Chapter 6 🡪Widening
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
4. Chapter 7 🡪 Path Sensitivity and Relational Analysis
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
5. Chapter 8 🡪 Interprocedural Analysis
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
6. Chapter 9 🡪 Control Flow Analysis
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
7. Chapter 10 🡪 Pointer Analysis
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material
8. Chapter 11 🡪 Abstract Interpretation
   1. Main Idea
   2. Section Ideas
   3. Examples for Mastery
   4. Remaining Questions about Material