17-355/17-665/17-819: Program Analysis

Spring 2022 Midterm Exam Study Guide

Rohan Padhye and Isabella Laybourn

Name: _____

Andrew ID: _____

Study Guide Instructions: This study guide is intended to help you prepare for the midterm. It consists of several types of material:

1. Material providing content or context that will be on the test, such as the background on non-deterministic language constructs (Question 1, Operational Semantics) and the background on alias pairs analysis (Question 2). This material is provided so you can familiarize yourself with it ahead of time, and not have to wade through background material and wrap your head around it in a limited timespan.
2. Question *templates* that specify a type of question that may or may not be asked, but without particulars. You can practice answering those kinds of questions or otherwise making sure you understand how to answer such questions.
3. Concrete questions, which may or may not be on the exam, or may be similar to those that will be on the exam.

We cannot promise to have full coverage of all material in the course so far, or all of the questions that may ultimately be on the exam. However, we have attempted to be thorough, and have tried to give you a sample of the *types* of questions we will be asking about the material in the course. We expact that if you carefully study this material and the lecture notes, you will be well-prepared for the exam.

Question 1: Operational Semantics (0 points)

*Note: the exam **will** contain questions based on this idea of nondeterministic language constructs, and so we include this material in the guide so you can wrap your head around how to reason about them ahead of time.*

Recall from lecture and homework assignments, we previously proved WHILE to be deterministic when execution terminates:

$$\forall S \in \texttt{Stmt} \,.\quad \forall E, E', E'' \in \texttt{Var} \to \mathbb{Z} \,.\qquad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \quad\Rightarrow\quad E' = E''$$

(a) First, consider an extension to WHILE called $\text{WHILE}_{\texttt{rand}}$, which adds a new boolean expression called `rand()` that can evaluate to *either* `true` or `false` at random each time it is evaluated. We define the big-step semantics for this construct using two rules:

$$\frac{}{\langle E, \texttt{rand()}\rangle \Downarrow \texttt{true}}\ \textit{big-rand-true} \qquad \frac{}{\langle E, \texttt{rand()}\rangle \Downarrow \texttt{false}}\ \textit{big-rand-false}$$

Your task is to prove that $\text{WHILE}_{\texttt{rand}}$ is non-deterministic. To do this, you must show that:

$$\exists S \in \texttt{Stmt} \,.\quad \exists E, E', E'' \in \texttt{Var} \to \mathbb{Z} \,.\quad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \wedge E' \neq E''$$

In other words, you will construct a $\text{WHILE}_{\texttt{rand}}$ program S and pick an initial environment E such that the evaluation of $\langle E, S\rangle$ will result in at least two distinct final states.

i. Construct the $\text{WHILE}_{\texttt{rand}}$ program S.

ii. Specify the environment E.

iii. Give an environment E' such that $\langle E, S\rangle \Downarrow E'$.

iv. Give a distinct environment E'' such that $\langle E, S\rangle \Downarrow E''$, where $E' \neq E''$.

(b) Now, consider a different extension to WHILE called $\text{WHILE}_{\|}$, which adds a new statement type $S_1 \parallel S_2$. This construct is similar to sequencing ($S_1; S_2$), with the difference that the order of evaluation is non-deterministic. We again define two rules for this construct:

$$\frac{\langle E, S_1\rangle \Downarrow E' \quad \langle E', S_2\rangle \Downarrow E''}{\langle E, S_1 \parallel S_2\rangle \Downarrow E''}\ \textit{big-par-1} \qquad \frac{\langle E, S_2\rangle \Downarrow E' \quad \langle E', S_1\rangle \Downarrow E''}{\langle E, S_1 \parallel S_2\rangle \Downarrow E''}\ \textit{big-par-2}$$

Your task is to prove that $\text{WHILE}_{\|}$ is non-deterministic. To do this, you must again show that:

$$\exists S \in \texttt{Stmt}\ .\ \ \exists E, E', E'' \in \texttt{Var} \to \mathbb{Z}\ .\ \ \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \wedge E' \neq E''$$

i. Construct the $\text{WHILE}_{\|}$ program S.

ii. Specify the environment E.

iii. Give an environment E' such that $\langle E, S\rangle \Downarrow E'$.

iv. Give a distinct environment E'' such that $\langle E, S\rangle \Downarrow E''$, where $E' \neq E''$.

(c) Finally, consider a multi-threaded extension to WHILE3ADDR called $\text{WHILE3ADDR}_{\texttt{fork}}$ which introduces two instructions "`fork` m" (where m is an integer) and "`halt`". The semantics of "n : `fork` m" are such that a new thread of execution is spawned starting at instruction m, and the current thread continues execution from $n+1$. The `halt` instruction stops the current thread. A $\text{WHILE3ADDR}_{\texttt{fork}}$ program executes by non-deterministically choosing an active thread and evaluating the instruction corresponding to its program counter. All threads share a common global state. A $\text{WHILE3ADDR}_{\texttt{fork}}$ program starts

with one thread executing at instruction 1, and ends when all threads have finished execution.

For example, consider the program:

$$
\begin{array}{ll}
1: & x := 1 \\
2: & y := 2 \\
3: & \text{fork } 6 \\
4: & \text{print } x \\
5: & \text{halt} \\
6: & \text{print } y \\
7: & \text{halt}
\end{array}
$$

When the first thread reaches instruction 3, it forks off a second thread. The first thread continues to print 1 and then stops. The second thread prints 2 and then stops. Because of the non-deterministic order of thread interleaving, this program may either print 1 then 2, or it may print 2 then 1.

To account for multi-threaded execution in the program semantics, we change the configuration c to be of the form $c \in E \times \mathbb{N}^*$, where $\mathbb{N}^*$ refers to a sequence of zero or more program counters for all the currently executing threads. The initial configuration is $c_0 = \langle E, (1) \rangle$, and the program halts when reaching a final configuration of the form $\langle E', () \rangle$ (where $()$ is the empty sequence).

i. When executing the sample program above, with the empty initial environment $E_0 = \{\}$, what is the configuration immediately after evallating instruction 3?

ii. Recall that an execution trace of program P is a sequence of configurations $c_0, c_1, \ldots$, such that $P \vdash c_i \rightsquigarrow c_{i+1}$. When starting with the empty environment $E_0 = \{\}$, how many distinct traces can the execution of the above program have? For each trace, list down the sequence of program counters being evaluated (e.g. "1, 2, 3, 4, ...").

iii. Draw the control-flow graph (CFG) for the above program. Remember that a CFG edge exists between nodes m and n iff the evaluation of m can be immediately followed by the evaluation of n in any trace of program execution.

iv. Complete the small-step semantics rule for the constant-variable assignment instruction ($x := m$) executing in one of k active threads.

$$\frac{k \geqslant 1 \quad 1 \leqslant i \leqslant k \quad P[n_i] = x := m}{P \vdash \langle E, (n_1, \ldots, n_i, \ldots, n_k) \rangle \rightsquigarrow} \textit{ threaded-step-const}$$

Question 2: Analysis Specification (0 points)

*Note: the exam **will** contain questions based on this idea of alias pairs analysis. As in these sample questions, it will be a dataflow analysis instead of a flow-insensitive constraint solving problem like Andersen's analysis.*

An alternative to points-to analysis is *alias pairs* analysis, which computes, at each program point, a set of pairs of expressions that may alias one another. An expression is either a variable such as x, or a single dereference of a pointer variable such as *x. We do not track aliased pairs including more dereferences—that is, nothing like ***x. To illustrate, the pair (*x, y) means that x may point to y, whereas the pair (*x, *y) means that x and y may point to the same memory location.

For example, consider the following program:

$$\begin{array}{ll} 1: & s := 2 \\ 2: & x := \&y \\ 3: & y := \&z \\ 4: & t := \&s \\ 5: & w := t \end{array}$$

This analysis would compute the following pair sets immediately after each program location:

location	alias pairs
1	$\varnothing$
2	{ (*x, y) }
3	{ (*x, y) (*y, z) }
4	{ (*x, y) (*y, z) (*t, s) }
5	{ (*x, y) (*y, z) (*t, s) (*w, s) (*w, *t) }

(a) Define a lattice L and analysis information σ for this analysis.

(b) What do *top* and *bottom* correspond to in this lattice? (the answer $\top$ is incorrect).

(c) Assume we have the alias information σ = FOO, and consider analyzing the statement BAR.

  i. Which alias pairs in the state should be *killed* by the statement?

  ii. Which alias pairs should be *generated* by the statement?

(d) Consider the statement FOO. If the alias information before the statement is σ = EXAMPLE1, what is the alias information after the statement?

(e) Imagine you have only two variables x and y in your program. What is the maximum number of alias pairs you can have in the worst case? Do not worry about type safety—assume that all alias pairs that can be represented by the syntax are possible. What is the total size of the lattice?

Question 3: Soundness (0 points)

Imagine we want to extend X analysis to a language with Y. Consider the following *incorrect* flow function:

$$f_{FOO}[\![CODE]\!](\sigma) \quad = \sigma[...update...]$$

This function is incorrect because it does X; to see this, consider code that does Y.

(a) Prove that this flow function is not locally sound.

(b) Specify a correct flow function.

(c) Prove that your new flow function is monotonic.

(d) At a high level, how must we change either the worklist algorithm or the control flow graph to implement a backwards analysis, like in live variables analysis?

Question 4: Interprocedural Analysis (0 points)

Imagine you are would like to implement an interprocedural X analysis. Consider the following simple test code:

...example omitted...

(a) Would porting an intraprocedural analysis and applying it to the interprocedural control flow graph produce satisfactory analysis output on this example? Why or why not?

(b) Provide an example program that demonstrates a case where function inlining is a bad solution to the interprocedural control flow problem, and explain why it shows that.

(c) What is one reason that dynamic dispatch poses a challenge to interprocedural dataflow analysis?

(d) Create a `GETCTX` function for the k-limited value-based context strategy.