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

Fall 2020 Midterm Exam # 1

Claire Le Goues and Jeremy Lacomis

Name: ______________________________

Andrew ID: ______________________________

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

1. Material providing content or context that will actually be on the test, especially the background on regular expressions (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. Questions 1(a) and 1(b) are good examples. You can practice answering those kinds of questions by filling in reasonable examples for the blanks, or otherwise making sure you understand how to answer this kind of question.
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	Points	Score
Operational Semantics	0	
Analysis Specification	0	
Soundness	0	
Interprocedural Analysis	0	
Total:	0	

Question 1: Operational Semantics (0 points)

Note: the exam ***will*** *contain questions based on this idea of specifying inference rules for regular expressions, and so we include this material in the guide so you can wrap your head around how regular expressions work and could be precisely specified in advance.*

Consider the following abstract grammar for regular expressions:

e	$::=$	"x"	singleton — matches the character x
	$\mid$	`empty`	skip — matches the empty string
	$\mid$	$e_1 e_2$	concatenation — matches e_1 followed by e_2
	$\mid$	$e_1 \mid e_2$	or — matches e_1 or e_2
	$\mid$	$e*$	Kleene star —matches 0 or more occurrences of e

We also give an abstract grammar for strings (modeled as lists of characters; we write "bye" as shorthand for "b" :: "y" :: "e" :: nil):

s	$::=$	nil	empty string
	$\mid$	"x" :: s	string with first character *x*, and other characters s

We introduce a new judgement to give large-step operational semantics rules of inference for regular expressions matching strings:

$$\vdash e \texttt{ matches } s \texttt{ leaving } s'$$

A regular expression e applied to string s means that e matches some prefix of s, leaving the suffix s' unmatched. If $s' = nil$, then e matched s exactly; if $s' = s$, then e does not match any part of s. For example: $\vdash$ "h" `matches`"$hello$" `leaving` "$ello$"; the concatenation construct means that $\vdash$ "he" `matches`"$hello$" `leaving` "llo".Note that this semantics may be non-deterministic, because we can derive all of the following:

$$\vdash (\text{"}h\text{"} \mid \text{"}e\text{"}) * \texttt{ matches } \text{"}hello\text{"} \texttt{ leaving } \text{"}ello\text{"}$$
$$\vdash (\text{"}h\text{"} \mid \text{"}e\text{"}) * \texttt{ matches } \text{"}hello\text{"} \texttt{ leaving } \text{"}hello\text{"}$$
$$\vdash (\text{"}h\text{"} \mid \text{"}e\text{"}) * \texttt{ matches } \text{"}hello\text{"} \texttt{ leaving } \text{"}llo\text{"}$$

Here are two of the simpler rules of inference for regular expressions:

$$\frac{s = \text{"}x\text{"} :: s'}{\vdash \text{"}x\text{"} \texttt{ matches } s \texttt{ leaving } s'}\ \textit{singleton} \qquad \frac{}{\vdash \texttt{empty matches } s \texttt{ leaving } s}\ \textit{skip}$$

(a) Precisely specify the BLANK construct(s) via one or more large-step operational semantics inference rules.

(b) Consider these potential rules of inference for the BLANK construct:

$$\frac{premises - 1}{conclusion - 1}\ \textit{rule-1} \qquad \frac{premises - 2}{conclusion - 2}\ \textit{rule-2-WRONG}$$

i. Recall that a logical system is *complete* if every true statement is provable; it is *sound* if every provable statement is true. Assuming the other rules are correct, the `rule-2-WRONG` rule makes our overall system:

○ Unsound

○ Incomplete

ii. Prove it, by giving either an example of a true statement that cannot be proven with this rule, or a provable judgement that is untrue.

iii. Write a correct rule for BLANK.

(c) The non-determinism in the above rules is sub-optimal; we would instead prefer operational semantics for a judgement that returns the *set* of all possible suffices. If S is a set of strings s, we could change our previous judgement accordingly, to: $\vdash e$ `matches` s `leaving` S, and then use rules of inference like the following:

$$\frac{}{\vdash \text{“}x\text{”}\ \texttt{matches}\ s\ \texttt{leaving}\ \{s' | s = \text{“}x\text{”} :: s\text{‘}\}}\ singleton' \qquad \frac{}{\vdash \mathit{empty}\ \texttt{matches}\ s\ \texttt{leaving}\ \{s\}}\ empty'$$

$$\frac{\vdash e_1\ \texttt{matches}\ s\ \texttt{leaving}\ S \quad \vdash e_2\ \texttt{matches}\ s\ \texttt{leaving}\ S'}{\vdash e_1 | e_2\ \texttt{matches}\ s\ \texttt{leaving}\ S \cup S'}\ or$$

Do one of the following:

- *either* give operational semantics rules of inference for one of **CONSTRUCT** or **CONSTRUCT** You may not place a derivation inside a set constructor, as in $\{x | \exists y.\ \vdash e\ \texttt{matches}\ x\ \texttt{leaving}\ y\}$. Each inference rules must have a finite and fixed set of hypotheses.
- *or* argue in two–five sentences that it cannot be done correctly in the given framework. Back up your argument by presenting two attempted but “wrong” rules of inference, and demonstrate that each one is incorrect—either unsound or incomplete—with respect to our intuitive notion of regular expression matching.

(d) Your roommate looks over your shoulder as you are reviewing for this exam, and sees the following inference rule for small-step semantics for WHILE3ADDR copy:

$$\frac{P[n] = x := y}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto E[y]], n+1\rangle}\ \textit{step-copy}$$

They ask “Hey why do you need that premise in that rule? You wouldn’t need it in the WHILE equivalent small-step rules...”

(e) Induction on the structure of expressions is sufficient to prove many properties about them (like the one we did in class, showing that the number of literals and variable occurrences in some expressions is L(a) = O(a) + 1). Why, in one sentence, can’t we induct on statement structure to prove most other interesting properties about WHILE (and have to induct on the structure of the derivation, instead)?

Question 2: Analysis Specification (0 points)

Note: the exam ***will*** *contain questions based on this idea of alias pairs analysis, but note that, as in these sample questions, the analysis will not be constraint based (as in, it will be similar to the fixpoint-based approach we started with).*

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.[1]

For example, consider the following program:

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

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

location	alias pairs
1	∅
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). [2].

(c) Define the *ordering relation* between lattice elements. i.e., when is $\sigma_1 \sqsubseteq \sigma_2$?

(d) Define the *join operation* on lattice elements. i.e.,. what is $\sigma_1 \sqcup \sigma_2$?

(e) 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?

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

(g) Assuming monotonic flow functions, will the analysis on the alias pair lattice defined above terminate? Why or why not? Your answer should be precise in terms of bounding the height of the lattice.

(h) Imagine we relax the assumption about the number of dereferences tracked. E.g., consider an alternative lattice that allows a pair (****x,y) (whereas the original analysis only allows pairs like (*x, y). Is the analysis guaranteed to terminate on this new lattice? Why or why not?

[1] We assume for simplicity that the pair (x,y) cannot occur, which is true in C and Java, but not C++.

[2] cf. constant propagation: $\top = \mathcal{Z}$

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) Why is it a good idea to apply a widening operator only at loop heads in the control flow graph?

(e) Why don't we need a widening operator for zero or sign analysis?

(f) Why can't we use the basic operational semantics to reason about the correctness of a reaching definitions analysis?

(g) 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?

________ / 0

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) Imagine you wanted to use default assumptions around function calls.

  i. Provide an example of default assumptions that produces sound but imprecise results for the example.

  ii. Provide an example of default assumptions that produces more precise output on this example.

  iii. Provide an example for which the more precise assumptions produces incorrect dataflow output.

(b) 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?

(c) 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.

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

(e) Assuming a context-sensitivity limit of X, which context-sensitivity-limiting approach would provide more precise results for the following example function:

...example omitted...

○ Call strings

○ Contexts

...possibly: justify your answer...

(f) Consider the following example program written in the functional language we used for CFA:

...example omitted...

Consider the following dataflow analysis information generated according to constraints generated by a (potentially buggy) implementation of 0-CFA analysis with constant propagation:

...example omitted...

Is this dataflow analysis information *acceptable*? Why or why not?

(g) Why do we concern ourselves with termination of the iterative, fixpoint-based dataflow analyses, but not the CFA-style analyses?

_________ / 0

End of exam.