1 3F-1 Bookkeeping

- **0 pts** Correct

Advanced Programming Languages
Homework Assignment 3F and 3C

Mollie Bakal (bakalm)

3/6/21

Logistics. You must work alone. Your name and Michigan email address must appear on the first page of your PDF submission but *may not appear anywhere else*. This is to protect your identity during peer review. The first page of your submission is *not* shared during peer view but all subsequent pages are.

Exercise 3F-1. Bookkeeping [2 points]. These answers should appear on the first page of your submission and are kept private.

1. Indicate in a sentence or two how much time you spent on this homework.
2. Indicate in a sentence or two how difficult you found it subjectively.

Answers to Bookkeepping Time spent varies–I spent a good chunk of a 20-hour round-trip drive thinking about this, but also mentally wandering off and revisiting the same thoughts and staring at Pennsylvania. Let's go with 30 hours outside the trip, plus or minus 5? I found the formal part more difficult than the other homeworks; I am not always the best at intuiting whether or not something is provable, so I vacillated between "it's trivially possible to write rules of inference like this for 3f-3, this is definitely deterministic" and "it's trivially impossible to write those rules, it needs to recurse with a non-fixed number of hypotheses" for way too long.

All subsequent answers should appear after the first page of your submission and may be shared publicly during peer review.

Exercise 3F-1. Regular Expression, Large-Step [10 points]. Regular Expressions are commonly used as abstractions for string matching. Here is an abstract grammar for regular expressions:

e	::=	"x"	singleton — matches the character $\hat{x}$
	|	empty	skip — matches the empty string
	|	$e_1\ e_2$	concatenation — matches e_1 followed by e_2
	|	$e_1 \mid e_2$	or — matches e_1 or e_2
	|	$e*$	Kleene star — matches 0 or more occurrence of e
	|	.	matches any single character
	|	["x" − "y"]	matches any character between $\hat{x}$ and $\hat{y}$ inclusive
	|	$e+$	matches 1 or more occurrences of e
	|	$e?$	matches 0 or 1 occurrence of e

We will call the first five cases the *primary* forms of regular expressions. The last four cases can be defined in terms of the first five. We also give an abstract grammar for strings (modeled as lists of characters):

s	::=	nil	empty string
	|	"x" :: s	string with first character $\hat{x}$ and other characters s

We write "bye" as shorthand for "b" :: "y" :: "e" :: nil. This exercise requires you to give large-step operational semantics rules of inference related to regular expressions matching strings. We introduce a judgment:

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

The interpretation of the judgment is that the regular expression e matches some prefix of the string s, leaving the suffix s' unmatched. If $s' = \text{nil}$ then r matched s exactly. Examples:

$$\vdash \text{"h"}(\text{"e"}+) \text{ matches "hello" leaving "llo"}$$

Note that this operational semantics may be considered *non-deterministic* because we expect to be able to derive all three of the following:

$$\begin{array}{l} \vdash (\text{"h"} \mid \text{"e"})* \text{ matches "hello" leaving "ello"} \\ \vdash (\text{"h"} \mid \text{"e"})* \text{ matches "hello" leaving "hello"} \\ \vdash (\text{"h"} \mid \text{"e"})* \text{ matches "hello" leaving "llo"} \end{array}$$

Here are two rules of inference:

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

Give large-step operational semantics rules of inference for the other three primal regular expressions.

Answer

$$\frac{e1 \text{ matches } s_1 \text{ leaving } s_1' \quad e2 \text{ matches } s_2 \text{ leaving } s_2' \quad s = s1 :: s2 :: s'}{\vdash e1e2 \text{ matches } s \text{ leaving } s'}$$

$$\frac{e1 \text{ matches } s \text{ leaving } s_{e1}' \quad e2 \text{ matches } s \text{ leaving } s_{e2}' \quad s = s2 :: s_{e2}'}{\vdash e1|e2 \text{ matches } s \text{ leaving } s_{e2}'}$$

or

$$\frac{\dfrac{e1 \text{ matches } s \text{ leaving } s_{e1}' \quad e2 \text{ matches } s \text{ leaving } s_{e2}' \quad s = s1 :: s_{e1}'}{\vdash e1|e2 \text{ matches } s \text{ leaving } s_{e1}'}}{\vdash e* \text{ matches } s \text{ leaving } s}$$

or

$$\frac{e \text{ matches } s \text{ leaving } s'' \quad e* \text{ matches } s'' \text{ leaving } s' \quad s = s''' :: s''}{\vdash e* \text{ matches } s \text{ leaving } s'}$$

Exercise 3F-2. Regular Expression and Sets [5 points]. We want to update our operational semantics for regular expressions to capture multiple suffices. We want our new operational semantics to be deterministic — it return the set of all possible answers from the single-answer operational semantics above. We introduce a new judgment:

$$\vdash e \text{ matches } s \text{ leaving } S$$

And use rules of inference like the following:

$$\frac{}{\vdash \text{"x" matches } s \text{ leaving } \{s' \mid s = \text{"x"} :: s'\}} \qquad \frac{}{\vdash \text{empty matches } s \text{ leaving } \{s\}}$$

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s \text{ leaving } S'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } S \cup S'}$$

You must do one of the following:

- *either* give operational semantics rules of inference for $e*$ and e_1e_2. You may *not* place a derivation inside a set constructor, as in: $\{x \mid \exists y.\ \vdash e \text{ matches } x \text{ leaving } y\}$. Each inference rules must have a finite and fixed set of hypotheses.
- *or* argue in one or two 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 show that each one is either unsound or incomplete with respect to our intuitive notion of regular expression matching.

Part of doing research is getting stuck. When you get stuck, you must be able to recognize whether "you are just missing something" or "the problem is actually impossible".

2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

Answer

$$\frac{e1 \text{ matches } s_1 \text{ leaving } s_1' \quad e2 \text{ matches } s_2 \text{ leaving } s_2' \quad s = s1 :: s2 :: s'}{\vdash e1e2 \text{ matches } s \text{ leaving } s'}$$

$$\frac{e1 \text{ matches } s \text{ leaving } s_{e1}' \quad e2 \text{ matches } s \text{ leaving } s_{e2}' \quad s = s2 :: s_{e2}'}{\vdash e1|e2 \text{ matches } s \text{ leaving } s_{e2}'}$$

or

$$\frac{\dfrac{e1 \text{ matches } s \text{ leaving } s_{e1}' \quad e2 \text{ matches } s \text{ leaving } s_{e2}' \quad s = s1 :: s_{e1}'}{\vdash e1|e2 \text{ matches } s \text{ leaving } s_{e1}'}}{\vdash e* \text{ matches } s \text{ leaving } s}$$

or

$$\frac{e \text{ matches } s \text{ leaving } s'' \quad e* \text{ matches } s'' \text{ leaving } s' \quad s = s''' :: s''}{\vdash e* \text{ matches } s \text{ leaving } s'}$$

Exercise 3F-2. Regular Expression and Sets [5 points]. We want to update our operational semantics for regular expressions to capture multiple suffices. We want our new operational semantics to be deterministic — it return the set of all possible answers from the single-answer operational semantics above. We introduce a new judgment:

$$\vdash e \text{ matches } s \text{ leaving } S$$

And use rules of inference like the following:

$$\frac{}{\vdash \text{"x" matches } s \text{ leaving } \{s' \mid s = \text{"x"} :: s'\}} \qquad \frac{}{\vdash \text{empty matches } s \text{ leaving } \{s\}}$$

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s \text{ leaving } S'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } S \cup S'}$$

You must do one of the following:

- *either* give operational semantics rules of inference for $e*$ and e_1e_2. You may *not* place a derivation inside a set constructor, as in: $\{x \mid \exists y.\ \vdash e \text{ matches } x \text{ leaving } y\}$. Each inference rules must have a finite and fixed set of hypotheses.
- *or* argue in one or two 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 show that each one is either unsound or incomplete with respect to our intuitive notion of regular expression matching.

Part of doing research is getting stuck. When you get stuck, you must be able to recognize whether "you are just missing something" or "the problem is actually impossible".

Answer This is impossible in the given framework, with the fixed and finite hypotheses– the Kleene star has some finite number of iterations for every non-empty match, as strings are finite and every non-empty match expression must match some combination of characters, removing at least one character from the string; eventually, either the nil string will be left or some combination of characters which the Kleene star cannot match. However, the number of iterations of that which must be run are unknown, because this has a tree structure, so the number of hypotheses are not set. Here is what some bad, non-recursive rules could look like.

$$\frac{\text{e matches } s \text{ leaving } \{s'\}}{\vdash \text{e}* \text{ matches } s \text{ leaving } \{s'\}}$$

This is incomplete: it only matches the outermost, and does not match repeats. Similarly, an e1 e2 rule which has a fixed number of hypotheses in s_1 after e1 to reason about would not be able to reason about e2.

$$\frac{e1 \text{ matches } s_1 \text{ leaving } \{s_1'\} \quad e2 \text{ matches } s_1' \text{ leaving } \{s_2'\}}{\vdash e1e2 \text{ matches } s \text{ leaving } \{s' | s' = s_2'\}}$$

An attempt at functional rules for e1 e2 and e* is below:

$$\frac{\dfrac{e1 \text{ matches } s_1 \text{ leaving } \{s_1'\} \quad e2 \text{ matches } s_2 \text{ leaving } \{s_2'\}}{\vdash e1e2 \text{ matches } s \text{ leaving } \{s' | s' = s_2' \text{ iffs}_2\text{in } \{s_1'\}\}}}{\vdash \text{empty}* \text{ matches } s \text{ leaving } \{s\}}$$

$$\frac{\text{e matches } s \text{ leaving } \{s'\}}{\vdash \text{e}* \text{ matches } s \text{ leaving } e * \{s'\}}$$

Exercise 3F-3. Equivalence [7 points]. In the class notes (usually marked as "optional material" for the lecture component of the class but relevant for this question) we defined an equivalence relation $c_1 \sim c_2$ for IMP commands. Computing equivalence turned out to be undecideable: $c \sim c$ iff c halts. We can define a similar equivalence relation for regular expressions: $e_1 \sim e_2$ iff $\forall s \in S.\ \vdash e_1 \text{ matches } s \text{ leaving } S_1 \ \wedge\ \vdash e_2 \text{ matches } s \text{ leaving } S_2 \implies S_1 = S_2$ (note that we are using an "updated" operational semantics that returns the set of all possible matched suffices, as in the previous problem).

You must *either* claim that $e_1 \sim e_2$ is undecideable by reducing it to the halting problem *or* explain in two or three sentences how to compute it. You may assume that I the reader is familiar with the relevant literature.

Answer I believe that this is possible–unlike possible sequences of IMP commands, strings are finite. The e* command is analogous to the while loop in that it is the only command which can produce itself in its output, causing a loop from a finite input in IMP; however, as the string is finite, if we use my suggested rule to short-circuit empty* all other Kleene star matches and expressions must terminate. We can show equality in the same way that IMP would; structural induction on derivations should work the same way.

Exercise 3C. SAT Solving. Download the Homework 3 code pack from the course web page. Update the skeletal SMT solver so that it correctly integrates the given DPLL-style

3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

Answer This is impossible in the given framework, with the fixed and finite hypotheses– the Kleene star has some finite number of iterations for every non-empty match, as strings are finite and every non-empty match expression must match some combination of characters, removing at least one character from the string; eventually, either the nil string will be left or some combination of characters which the Kleene star cannot match. However, the number of iterations of that which must be run are unknown, because this has a tree structure, so the number of hypotheses are not set. Here is what some bad, non-recursive rules could look like.

$$\frac{\text{e matches } s \text{ leaving } \{s'\}}{\vdash \text{e}* \text{ matches } s \text{ leaving } \{s'\}}$$

This is incomplete: it only matches the outermost, and does not match repeats. Similarly, an e1 e2 rule which has a fixed number of hypotheses in s_1 after e1 to reason about would not be able to reason about e2.

$$\frac{e1 \text{ matches } s_1 \text{ leaving } \{s_1'\} \quad e2 \text{ matches } s_1' \text{ leaving } \{s_2'\}}{\vdash e1e2 \text{ matches } s \text{ leaving } \{s' | s' = s_2'\}}$$

An attempt at functional rules for e1 e2 and e* is below:

$$\frac{\dfrac{e1 \text{ matches } s_1 \text{ leaving } \{s_1'\} \quad e2 \text{ matches } s_2 \text{ leaving } \{s_2'\}}{\vdash e1e2 \text{ matches } s \text{ leaving } \{s' | s' = s_2' \text{ iffs}_2\text{in } \{s_1'\}\}}}{\vdash \text{empty}* \text{ matches } s \text{ leaving } \{s\}}$$

$$\frac{\text{e matches } s \text{ leaving } \{s'\}}{\vdash \text{e}* \text{ matches } s \text{ leaving } e * \{s'\}}$$

Exercise 3F-3. Equivalence [7 points]. In the class notes (usually marked as "optional material" for the lecture component of the class but relevant for this question) we defined an equivalence relation $c_1 \sim c_2$ for IMP commands. Computing equivalence turned out to be undecideable: $c \sim c$ iff c halts. We can define a similar equivalence relation for regular expressions: $e_1 \sim e_2$ iff $\forall s \in S.\ \vdash e_1 \text{ matches } s \text{ leaving } S_1 \ \wedge\ \vdash e_2 \text{ matches } s \text{ leaving } S_2 \implies S_1 = S_2$ (note that we are using an "updated" operational semantics that returns the set of all possible matched suffices, as in the previous problem).

You must *either* claim that $e_1 \sim e_2$ is undecideable by reducing it to the halting problem *or* explain in two or three sentences how to compute it. You may assume that I the reader is familiar with the relevant literature.

Answer I believe that this is possible–unlike possible sequences of IMP commands, strings are finite. The e* command is analogous to the while loop in that it is the only command which can produce itself in its output, causing a loop from a finite input in IMP; however, as the string is finite, if we use my suggested rule to short-circuit empty* all other Kleene star matches and expressions must terminate. We can show equality in the same way that IMP would; structural induction on derivations should work the same way.

Exercise 3C. SAT Solving. Download the Homework 3 code pack from the course web page. Update the skeletal SMT solver so that it correctly integrates the given DPLL-style

4 3F-4 Equivalence

- **0 pts** Correct

CNF SAT solver with the given theory of bounded arithmetic. In particular, you must update only the Main.solve function. Your updated solver must be correct. This notably implies that it must correctly handle all of the included test cases — we use `diff` for some testing, but if you change only the listed method you should end up with the same answers as the reference.

In addition, create an example "tricky" input that can be parsed by our test harness.

Submit your `.ml` and `.input` files.

Exercise 3F-4. SAT Solving [6 points]. Why do the last two included tests take such a comparatively long time? Impress me with your knowledge of DPLL(T) — feel free to use information from the assigned reading or related papers, not just from the lecture slides. I am looking for a reasonably detailed answer. Include a discussion of which single module you would rewrite first to improve performance, as well as how you would change that module.

Potential bonus point: The provided code contains at least one fairly egregious defect. Comment.

Answer The last two tests rely primarily on the theory-checker, and not as much on the more-efficient CNF solver. While SAT is undecidable, the DPLL algorithm usually solves it pretty quickly in practical cases. However, DPLL(T) must call the theory solver after the CNF is run if there are any theory (arithmetic) clauses which may be unsatisfied with the model. In this specific case, this theory-checker is heavily inefficient; it gathers all variables in clauses which it needs to check, then evaluates the model to check every value for every variable until it accepts within the range, from -127 to 128. With three variables, we need to check up to 256^3 values against the model–if all of them return true, then we break by raising an exception, but it may take up to that long. To fix the arithmodel module, we could try constraining the values which we check with. Instead of starting at -127, if an arithmetic clause must be true and has an equality operator, we could start by setting that variable to what it must be equal to. That would save a number of loop iterations. Similarly, if a variable is constrained by a ">" or "<" relation to a number or another variable, it could begin checking validity at that lower or upper bound, respectfully, and save time checking.

I also believe the code might have a defect relating to evaluation of the model. When I ran a variant of test case 27, which was just (x * x == 25), I got back that a satisfiable result was x = -127. I am not certain where this happens; the constraint appears to work: ((x) * (x)) = (25) and the function for evaluating the arithmetic model appears robust–I believe it has to do with the recursive bounded_search removing the variable, although I am not sure.

Submission. Turn in the formal component of the assignment as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment via the `autograder.io` website.

5 3F-5 SAT Solving

- **0 pts** Correct