Question assigned to the following page: 2

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.

Solution:

concatenation: $$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s' \quad \vdash e_2 \text{ matches } s' \text{ leaving } s''}{\vdash e_1 e_2 \text{ matches } s \text{ leaving } s''}$$

or: $$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s'}{\vdash e_1 | e_2 \text{ matches } s \text{ leaving } s'}$$

$$\frac{\vdash e_2 \text{ matches } s \text{ leaving } s'}{\vdash e_1 | e_2 \text{ matches } s \text{ leaving } s'}$$

Questions assigned to the following page: 3 and 2

Kleene star: $\dfrac{}{\vdash e* \text{ matches } s \text{ leaving } s}$

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

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:

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

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

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".

Solution: We cannot give an operational semantics rule of inference for $e*$ with a finite set of hypotheses, because $e*$ can match with 0 or any number of occurrence of e. So then our inference rule must have an infinite number of hypotheses, one for each number of es that we could match on.

$$\frac{\vdash e \text{ matches } s \text{ leaving } S_1, \vdash ee \text{ matches } s \text{ leaving } S_2, \vdash eee \text{ matches } s \text{ leaving } S_3 \ldots}{\vdash e* \text{ matches } s \text{ leaving } \{s \cup_i S_i\}}$$

We cannot give an operational semantics rule of inference for e_1e_2 because we do not have a fixed set of hypotheses. This is because for each suffix e_1 could possible leave behind, we have to check what e_2 would leave behind. Following this rule below.

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S, \forall s_i \in S \vdash e_2 \text{ matches } s \text{ leaving } S_i}{\vdash e_1e_2 \text{ matches } s \text{ leaving } \{S \cup_i S_i\}}$$

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 $\forall s \in S.\ \vdash e_1 \text{ matches } s \text{ leaving } S_1 \wedge \vdash$

Peer Review ID: 310860985 — enter this when you fill out your peer evaluation via gradescope

Question assigned to the following page: 4

e_2 matches s 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 from 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.

Solution: $e_1 \sim e_2$ is undecideable. We will reduce the halting problem to determining if $e_1 \sim e_2$.

We will assume that we have a blackbox solver who's input is two regular expressions, e_1, e_2, and who's output is True or False depending on wether the two regular expressions are equivalent as defined above. Now, suppose we are given an instance of input to the halting problem, aka a program. Let us assume our program is in IMP. IMP is Turing complete so it suffices as the language of our program.

We take our IMP program, and derive a regex expression from it. Specifically, we scan the program from left to right. For, any assignment command "x :=e" we evaluate e until we get an integer value or variable made of characters. Then we replace "x := e" with the int/chars e evaluates to. We replace any series of commands $c_1; c_2$ wi the regex we evaluate for c_1 concatenated with the regex we evaluate for c_2. For while loops we evaluate the body and add a kleene star. And for if statements we evaluate both possible commands until we reach our explicit characters and then or them together. Since each possible command except $x := e$ recursively includes a command, we will eventually reach our $x := e$ case, find the corresponding chaacters, and then glue them together with ors/stars/concatenation as specified above. These rules follow the equivalences below:

$$r_1 r_2 \equiv c_1; c_2$$

$$r_1* \equiv while()do\ c_1$$

$$r_1|r_2 \equiv if()c_1\ else\ c_2$$

$$e \equiv x := e$$

Next, we take our regex expression as defined above and give two instances of it to our blackbox. If the blackbox outputs True, that the duplicated regex expression is equivalent to our original regex expression, then our IMP program halts. Otherwise it loops forever. Since our blackbox can confirm that the two regex expressions are equivalent, it must be able to compute the sets they leave for all possible $s \in S$. So for any starting state, we can figure out what the regex expression leaves. So the corresponding program must halt, as the regex matching must eventually stop and leaving something behind.

We have decided the halting problem, therefore we have reached contradiction and our original assumption that equivalence for regex is decidable (via some blackbox) was incorrect.

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 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.

Question assigned to the following page: 5

Solution: see autograder

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.

Solution: The last two included test cases contain 3 variables from the theory, unlike the other tests containing at most 2, and are both conjunctions of unit clauses only. Following the algorithm on the DPLL(T) lecture slides, we end up adding all the unit clauses to our model, and we must call Arith.arith, our bounded integer constraint solver. This constraint solver is quite inefficent as it checks all possible values for each variable, from -127 to 128, so the running time of the constrain solver is at least 256^n where n is the number of variables. So, adding an additional variable increases the running time noticably. To improve the Arith.arith module, I would limit the bounds of the exhaustive search. For example, for test 35 when given $x < 12$, the solver could test values of $x \in [-127, 12)$ instead of $[-127, 128]$.

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.

Peer Review ID: 310860985 — enter this when you fill out your peer evaluation via gradescope