1 3F-1 Bookkeeping

- 0 pts Correct

Peer Review ID: 68549283 — enter this when you fill out your peer evaluation via gradescope

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:

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

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

$$\frac{\mathsf{e1\ matches}\ s\ \mathsf{leaving}\ s' \qquad \mathsf{e2\ matches}\ s'\ \mathsf{leaving}\ s''}{\vdash \mathsf{e1\ e2\ matches}\ s\ \mathsf{leaving}\ s''} \qquad \frac{\mathsf{e1\ matches}\ s\ \mathsf{leaving}\ s'}{\vdash \mathsf{e1} \mid \mathsf{e2\ matches}\ s\ \mathsf{leaving}\ s'}$$

$$\frac{\mathsf{e2\ matches}\ s\ \mathsf{leaving}\ s'}{\vdash \mathsf{e1} \mid \mathsf{e2\ matches}\ s\ \mathsf{leaving}\ s'} \qquad \frac{\mathsf{e\ matches}\ s\ \mathsf{leaving}\ s}{\vdash \mathsf{e}*\ \mathsf{matches}\ s\ \mathsf{leaving}\ s}$$

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

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:

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

And use rules of inference like the following:

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

2 3F-2 Regular Expressions, Large Step

- 0 pts Correct

Peer Review ID: 68549283 — enter this when you fill out your peer evaluation via gradescope

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	$::=$	$"\mathtt{x}"$	singleton — matches the character $\hat{x}$
	$\mid$	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 occurrence of e
	$\mid$	$.$	matches any single character
	$\mid$	$["\mathtt{x}" - "\mathtt{y}"]$	matches any character between $\hat{x}$ and $\hat{y}$ inclusive
	$\mid$	$e+$	matches 1 or more occurrences of e
	$\mid$	$e?$	matches 0 or 1 occurrence of e

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

$$\frac{\mathsf{e1\ matches}\ s\ \mathsf{leaving}\ s' \qquad \mathsf{e2\ matches}\ s'\ \mathsf{leaving}\ s''}{\vdash \mathsf{e1\ e2\ matches}\ s\ \mathsf{leaving}\ s''} \qquad \frac{\mathsf{e1\ matches}\ s\ \mathsf{leaving}\ s'}{\vdash \mathsf{e1} \mid \mathsf{e2\ matches}\ s\ \mathsf{leaving}\ s'}$$

$$\frac{\mathsf{e2\ matches}\ s\ \mathsf{leaving}\ s'}{\vdash \mathsf{e1} \mid \mathsf{e2\ matches}\ s\ \mathsf{leaving}\ s'} \qquad \frac{\mathsf{e\ matches}\ s\ \mathsf{leaving}\ s}{\vdash \mathsf{e*\ matches}\ s\ \mathsf{leaving}\ s}$$

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

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:

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

And use rules of inference like the following:

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

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

Creating operational semantics for $e*$ and $e_1\ e_2$ is not possible in this framework, as the rules created would be incomplete.

$$\frac{e \text{ matches } s \text{ leaving } S}{\vdash e* \text{ matches } s \text{ leaving } S} \qquad \frac{e1 \text{ matches } s \text{ leaving } S1}{\vdash e1\ e2 \text{ matches } s \text{ leaving } S1}$$

These two rules are incomplete because for the rules to be complete, another regular expression must be applied to all elements in S (either e again for $e*$ or $e2$ in $e1\ e2$), but to perform this operation a derivation would need to be included in the set constructor which is not allowed in this framework.

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 undecidable: $c \sim c$ 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).

$e_1 \sim e_2$ is undecidable. To prove this, I will reduce regular expression similarity to the halting problem.
Assume a solver exists for regular expression similarity $sim(e1, e2)$ that returns `True` if $e1 \sim e2$ and `False` otherwise. Given Sim, the following program can be constructed, which accepts another program as input and returns whether the input program halts.

```
def ImpToRegex(c,count):
    match (c) with
        | While(b, e) -> ImpToRegex(e, count) + "+ || " + ImpToRegex(e, cou
        | If(b, e1, e2) ->
            (ImpToRegex(e1, count) + " || " + (ImpToRegex(e2, count))
        | Seq(e1, e2) ->
            let s1 = ImpToRegex(e1, count) in
            let s2 = ImpToRegex(e2, count + 1) in
            s1 + " " + s2
        | Assigment(x, n)
        | Skip -> count;

def decideHalt(program) :
    let program_str = ImpToRegex(program, "0")
    return sim(program_str, "a*")
```

3 3F-3 Regular Expressions and Sets

- 0 pts Correct

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

Creating operational semantics for $e*$ and $e_1\ e_2$ is not possible in this framework, as the rules created would be incomplete.

$$\frac{e \text{ matches } s \text{ leaving } S}{\vdash e* \text{ matches } s \text{ leaving } S} \qquad \frac{e1 \text{ matches } s \text{ leaving } S1}{\vdash e1\ e2 \text{ matches } s \text{ leaving } S1}$$

These two rules are incomplete because for the rules to be complete, another regular expression must be applied to all elements in S (either e again for $e*$ or $e2$ in $e1\ e2$), but to perform this operation a derivation would need to be included in the set constructor which is not allowed in this framework.

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 undecidable: $c \sim c$ 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).

$e_1 \sim e_2$ is undecidable. To prove this, I will reduce regular expression similarity to the halting problem.
Assume a solver exists for regular expression similarity $sim(e1, e2)$ that returns `True` if $e1 \sim e2$ and `False` otherwise. Given Sim, the following program can be constructed, which accepts another program as input and returns whether the input program halts.

```
def ImpToRegex(c, count):
    match (c) with
        | While(b, e) -> ImpToRegex(e, count) + "+ || " + ImpToRegex(e, cou
        | If(b, e1, e2) ->
            (ImpToRegex(e1, count) + " || " + (ImpToRegex(e2, count))
        | Seq(e1, e2) ->
            let s1 = ImpToRegex(e1, count) in
            let s2 = ImpToRegex(e2, count + 1) in
            s1 + " " + s2
        | Assigment(x, n)
        | Skip -> count;

def decideHalt(program) :
    let program_str = ImpToRegex(program, "0")
    return sim(program_str, "a*")
```

If such a decidable algorithm as *sim* exists, then the halting problem can be decided by the above algorithm. If the converted regular expression string from the program is similar to the program string "a*", which matches all strings, then the program trace includes an infinite number of matches and does not halt.

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.

The included test cases take comparatively long because the arithmetic solver for the clauses runs comparatively slowly for these tests. In both tests, for satisfiability all four clauses need to be true. The DPLL solver runs relatively fast, since it is simple to find that all singleton clauses need to be true when connected by an "and" operation. The greater time complexity comes from finding variables assignments that satisfy all four arithmetic clauses. The Arith module includes the arith function, which is used to find satisfying variable value assignments for the constraints. Both of these test cases require that x > y and y > z. As the arithmetic solver as given, values for x and y are set to the lower bound (-127) and then all values of z are tried. None of these candidate values for z will work with the current x and y values, since x = y, but each will be tried by the solver. After none of the z mappings work, increasing new values for y will be tried through the upper bound. However, none of these will work since x is the lowest value and there is a constraint of x > y. This time wasting with invalid assignments will also happen with the y > z constraint, since the arithmetic solver works by trying the lowest value first, and tries values for x, y and z in that order. I would make two key changes to module: scan the constraints for any equality relationships with constants before trying assignments, and check the model after each variable assignment, not once every variable has been assigned. The first recommendation would greatly reduce the search space if any variables must be set to a certain constant (ie x = 10), as it allows x to be taken as a constant in the model rather than searching through all possible values. The second recommendation will prune impossible routes early, rather than trying all assignments. This will solve the problem of time wasting in test cases 35 and 36, as the relationship between x and y can be found earlier, before any values of z are tried with any already invalid x and y values.

Bonus Point: An incredibly egregious defect in the original code is that the arithmetic solver does not the model against the constraints until all variables have been set. This results in redundant checks of models where the invalidating variables are not changed. For example, in Test 35, the mappings [x: -127, y:-127] are compared for all possible values of z, even though the mappings of [x: -127, y: -127] will never satisfy the constraints regardless of the value of z given the constraint that x > y.

4 3F-4 Equivalence

- 0 pts Correct

Peer Review ID: 68549283 — enter this when you fill out your peer evaluation via gradescope

If such a decidable algorithm as *sim* exists, then the halting problem can be decided by the above algorithm. If the converted regular expression string from the program is similar to the program string "a*", which matches all strings, then the program trace includes an infinite number of matches and does not halt.

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.

The included test cases take comparatively long because the arithmetic solver for the clauses runs comparatively slowly for these tests. In both tests, for satisfiability all four clauses need to be true. The DPLL solver runs relatively fast, since it is simple to find that all singleton clauses need to be true when connected by an "and" operation. The greater time complexity comes from finding variables assignments that satisfy all four arithmetic clauses. The Arith module includes the arith function, which is used to find satisfying variable value assignments for the constraints. Both of these test cases require that x > y and y > z. As the arithmetic solver as given, values for x and y are set to the lower bound (-127) and then all values of z are tried. None of these candidate values for z will work with the current x and y values, since x = y, but each will be tried by the solver. After none of the z mappings work, increasing new values for y will be tried through the upper bound. However, none of these will work since x is the lowest value and there is a constraint of x > y. This time wasting with invalid assignments will also happen with the y > z constraint, since the arithmetic solver works by trying the lowest value first, and tries values for x, y and z in that order. I would make two key changes to module: scan the constraints for any equality relationships with constants before trying assignments, and check the model after each variable assignment, not once every variable has been assigned. The first recommendation would greatly reduce the search space if any variables must be set to a certain constant (ie x = 10), as it allows x to be taken as a constant in the model rather than searching through all possible values. The second recommendation will prune impossible routes early, rather than trying all assignments. This will solve the problem of time wasting in test cases 35 and 36, as the relationship between x and y can be found earlier, before any values of z are tried with any already invalid x and y values.

Bonus Point: An incredibly egregious defect in the original code is that the arithmetic solver does not the model against the constraints until all variables have been set. This results in redundant checks of models where the invalidating variables are not changed. For example, in Test 35, the mappings [x: -127, y:-127] are compared for all possible values of z, even though the mappings of [x: -127, y: -127] will never satisfy the constraints regardless of the value of z given the constraint that x > y.

5 3F-5 SAT Solving

- 0 pts Correct

Peer Review ID: 68549283 — enter this when you fill out your peer evaluation via gradescope