1 3F-1 Bookkeeping

- 0 pts Correct

Exercise 3F-1. Regular Expression, Large-Step [10 points]. Give large-step operational semantics rules of inference for the other three primal regular expressions.

$$\frac{\vdash \mathsf{e}_1 \text{ matches } s \text{ leaving } s'' \quad \vdash \mathsf{e}_2 \text{ matches } s'' \text{ leaving } s'}{\vdash \mathsf{e}_1\mathsf{e}_2 \text{ matches } s \text{ leaving } s'}$$

The above rule is for the expression e_1e_2. It first evaluates e_1 on the original s, leaving s'', and then it evaluates e_2 on that s'', the result of that being the final result s'.

$$\frac{\vdash \mathsf{e}_1 \text{ matches } s \text{ leaving } s'}{\vdash \mathsf{e}_1|\mathsf{e}_2 \text{ matches } s \text{ leaving } s'}$$

$$\frac{\vdash \mathsf{e}_2 \text{ matches } s \text{ leaving } s'}{\vdash \mathsf{e}_1|\mathsf{e}_2 \text{ matches } s \text{ leaving } s'}$$

The above two rules are for $e_1|e_2$. Either e_1 matches with s or e_2 does.

$$\frac{\vdash \texttt{empty} \text{ matches } s \text{ leaving } s}{\vdash \mathsf{e}* \text{ matches } s \text{ leaving } s}$$

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

The above two rules are for $e*$. There are two possibilities here, either there are 0 matches for e or there are more than 0. In the first case (the first rule), it simply leaves the original string s, it uses the rule for *empty* in the hypotheses. In the second case, we match e on s leaving s'', and then recursively call $e*$ on s''.

2 3F-2 Regular Expressions, Large Step

- 0 pts Correct

Exercise 3F-2. Regular Expression and Sets [5 points]. 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: $\{x \mid \exists y.\ \vdash e \textsf{ matches } x \textsf{ leaving } y\}$. 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.

It is not possible to provide operational semantics rules of inference given the syntax. Specifically the note "Each inference rule must have a finite and fixed set of hypotheses" makes this task impossible.

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } \{s'\} \quad \vdash e_2 \textsf{ matches } s' \textsf{ leaving } S}{\vdash e_1e_2 \textsf{ matches } s \textsf{ leaving } S}$$

The above is an attempted rule for e_1e_2. In the rule, we match e_1 with s leaving a set of one of the possible results of the match. That one suffix is then used with e_2, with that returning the set of all possible suffixes. This rule is technically sound, the result it captures is true, but it isn't complete since the first half of the hypotheses only looks at one of the possible suffixes and ignores the rest.

$$\frac{\vdash \textit{empty} \textsf{ matches } s \textsf{ leaving } \{s\} \quad \vdash e \textsf{ matches } s \textsf{ leaving } \{s'\} \quad \vdash e* \textsf{ matches } s' \textsf{ leaving } S}{\vdash e* \textsf{ matches } s \textsf{ leaving } \{s\} \cup S}$$

The above is an attempted rule for $e*$. Here, we have three hypotheses. The first captures the case where there are 0 matches for e, which will return a set that is just the original string s. The next two hypotheses are for the second case where there are more than 0 matches. The first here matches e on s, leaving a set of just one of the possible suffixes, and then recursively calls $e*$ on that one suffix to produce S. Again, this rule is sound as the set of suffixes it captures are all true, but it is not complete for the same reason as the previous rule. The intermediate hypotheses only focuses on one of the possible suffixes, which means won't be able to capture all true rules.

3 3F-3 Regular Expressions and Sets

- 0 pts Correct

Exercise 3F-3. Equivalence [7 points]. We can determine the equivalence between two regular expressions by looking at the inversions of the derivations of the rules for the expressions and comparing the elements of the final suffix sets between the two expressions(this is similar to proving command equivalence in Lecture 6 Slide 44). We can do this analysis because we define these suffix sets to be deterministic, all cases will be covered for expressions like $e1|e2$ or $e*$. To prove inequality, we simply need to come up with an example string s that would produce different results.

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.

The time sink from these test cases has to do with the Arith.arith method, the function responsible for determining if there are satisfying assignments for the arithmetic expressions. This function essentially takes a brute force method where it tries every possible assignment for every aexp with values ranging from -127 to 128. The runtime for this approach is 256^n, where n is the number of aexp in the model. In order to speed this up, a new algorithm would need to be implemented to perform this task. Something like the Simplex algorithm would be much faster as it could drastically reduce the search space of possible assignments.

In addition to this method being slow, there's also an obvious weakness that would prevent it from even producing the correct answer. The function can only check values between -127 and 128; if a satisfying assignment were to fall outside of the bounds then Arith.arith would fail to recognize it. For example, the following SAT formula would cause the program to return the incorrect answer: $(x = 200)$. This clearly has a satisfying assignment (just assign the value 200 to x), but the solver as is will return "Unsatisfiable."

4 3F-4 Equivalence

- 0 pts Correct

Exercise 3F-3. Equivalence [7 points]. We can determine the equivalence between two regular expressions by looking at the inversions of the derivations of the rules for the expressions and comparing the elements of the final suffix sets between the two expressions(this is similar to proving command equivalence in Lecture 6 Slide 44). We can do this analysis because we define these suffix sets to be deterministic, all cases will be covered for expressions like $e1|e2$ or $e*$. To prove inequality, we simply need to come up with an example string s that would produce different results.

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.

The time sink from these test cases has to do with the Arith.arith method, the function responsible for determining if there are satisfying assignments for the arithmetic expressions. This function essentially takes a brute force method where it tries every possible assignment for every aexp with values ranging from -127 to 128. The runtime for this approach is 256^n, where n is the number of aexp in the model. In order to speed this up, a new algorithm would need to be implemented to perform this task. Something like the Simplex algorithm would be much faster as it could drastically reduce the search space of possible assignments.

In addition to this method being slow, there's also an obvious weakness that would prevent it from even producing the correct answer. The function can only check values between -127 and 128; if a satisfying assignment were to fall outside of the bounds then Arith.arith would fail to recognize it. For example, the following SAT formula would cause the program to return the incorrect answer: $(x = 200)$. This clearly has a satisfying assignment (just assign the value 200 to x), but the solver as is will return "Unsatisfiable."

5 3F-5 SAT Solving

- 0 pts Correct

Peer Review ID: 68556509 — enter this when you fill out your peer evaluation via gradescope