Questions assigned to the following page: 2, 3, and 4

1 Regular Expression, Large-Step

Here is the large-step rule for the concatenation,

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

For the or expression we will need two rules,

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

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

And lastly we can add the following rules for the Kleene star,

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

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

2 Regular Expression and Sets

I argue that giving inference rules for $e*$ and e_1e_2 cannot be done correctly in the given framework. Because, e^* can match zero or more occurrences of e, potentially producing infinitely many distinct suffixes. Similarly, $(e_1\ e_2)$ also may produce multiple different remainders, depending on how e_1 matches. We would need a premise saying "for all $s' \in S$, match e_2 on s'," which requires either a set-comprehension or an unbounded number of premises. Therefore, one cannot write the rules for e^* or e_1e_2 without violating the finite-premise or no-set-comprehension restrictions. Let's try a rule in this shape for concatenation,

$$\frac{e_1 \text{ matches } s \text{ leaving } S'}{\vdash e_1e_2 \text{ matches } s \text{ leaving } S}$$

In this case the provided rule is unsound since it is true that e_1 matches the beginning of s if e_1e_2 matches that too however it is not enough to prove that the concatenation of e_1 and e_2 matches s leaving S. Or else we can try a set of rules for Kleene star,

$$\frac{}{\vdash e^* \text{ matches } s \text{ leaving } \{s\}} \qquad \frac{\vdash e \text{ matches } s \text{ leaving } S}{\vdash e^* \text{ matches } s \text{ leaving } S}$$

First one matches the empty string. However the second one only propagates one match of e. Thus it is incomplete relative to the notion of Kleene star.

3 Equivalence

Because every regular expression e can be converted into an NFA that recognizes precisely those prefixes that e can match from the start, the set of all possible "matched prefixes" of e is itself a regular language $L(e)$. Hence to check if e_1 and e_2 leave exactly the same leftover suffixes on every string, one only needs to test if $L(e_1) = L(e_2)$. This last step is decidable by comparing the two regular languages $L(e_1)$ and $L(e_2)$. To do that we can first convert constructed NFAs into DFAs using subset construction and then we can minimize both DFAs to check whether they're isomorphic. If they are isomorphic, the original DFAs accept the same language. Therefore, the given regular expressions will be equivalent.

Question assigned to the following page: 5

4 SAT Solving

The last two test cases included queries that have multiple arithmetic expressions rather than having a propositional expression or a single arithmetic expression. Because DPLL(T) must interleave propositional reasoning with theory checks, a formula that has multiple arithmetic variables and constraints can trigger more back-and-forth between the SAT solver and the arithmetic solver, repeatedly testing partial assignments until it finds (or refutes) a consistent combination. By contrast, purely propositional formulas, for example $(((p \to q) \to p) \to p) \wedge p \wedge q$, never invoke a theory solver and therefore can be solved more quickly. Similarly, a single constraint like $x > 5$ involves only one variable and is checked in a single step, making it far less costly than a system of several interacting inequalities.

I would begin by rewriting the arithmetic module to improve its performance. Currently, it uses a brute-force approach that explores numerous variable combinations, leading to large search times. Instead, I would implement a simplex-like method for finding a feasible solution to the arithmetic constraints and then adapt it for integer programming since we are dealing with a theory of bounded arithmetic. A suitable class of algorithms for this purpose is the cutting-plane methods, which solve a linear program relaxation and iteratively add constraints (cuts) that make the solution go toward integrality without excluding any integer feasible points.

One shortcoming of the program is that it only considers integer values from -127 to 128, which is quite restrictive. For instance, if we try to check whether there exists an x less than -127, the solver returns unsatisfiable. While these bounds might have been chosen to keep the complexity manageable, they limit the solver's usefulness for many real-world applications.