Question assigned to the following page: 2

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]. Here are the rules of inference:

$$\textsc{Concatenation}\quad \frac{\vdash e_1 \text{ matches } s \text{ leaving } s'' \qquad \vdash e_2 \text{ matches } s'' \text{ leaving } s'}{\vdash (e_1 e_2) \text{ matches } s \text{ leaving } s'}$$

For the concatenation rule, we first ensure that e_1 is matched, leaving s'', and then e_2 is matched, leaving s'.

$$\textsc{Or-1}\quad \frac{\vdash e_1 \text{ matches } s \text{ leaving } s'}{\vdash (e_1|e_2) \text{ matches } s \text{ leaving } s} \qquad \textsc{Or-2}\quad \frac{\vdash e_2 \text{ matches } s \text{ leaving } s'}{\vdash (e_1|e_2) \text{ matches } s \text{ leaving } s}$$

For the *or* case(|), we have two rules: when e_1 matches s,or similarly when e_2 matches s leaving s'.

$$\textsc{Kleene-Empty}\quad \frac{}{\vdash e* \text{ matches } s \text{ leaving } s'} \qquad \textsc{Kleene}\quad \frac{\vdash e \text{ matches } s \text{ leaving } s' \qquad \vdash e* \text{ matches } s \text{ leaving } s'}{\vdash e* \text{ matches } s \text{ leaving } s'}$$

For the kleene star case, we may have $e*$ match zero or more occurrences of e expressed using the | operator.

Question assigned to the following page: 3

Exercise 3F-2. Regular Expression and Sets [5 points]. Here are the attempted inference rules for concatenation and the Kleene Star.

$$\textsc{Concat}\quad \frac{\vdash e_1 \text{ matches } s \text{ leaving } S \qquad \{\vdash \forall s' \in S\} e_2 \text{ matches } s' \text{ leaving } S'}{\vdash e_1 e_2 \text{ matches } s \text{ leaving } S'}$$

$$\textsc{Kleene-1}\quad \frac{\vdash \text{empty matches } s \text{ leaving } \{s\}}{\vdash e* \text{ matches } s \text{ leaving } S'} \qquad \textsc{Kleene-2}\quad \frac{\vdash \{\forall\ s' \in S\} e* \text{ matches } s \text{ leaving } S'}{\vdash e* \text{ matches } s \text{ leaving } S'}$$

The key restriction in this problem is having a finite and fixed set of hypotheses (for the inference rules) given a language with infinite member string. Therefore, when we use $\forall$to quantify over the strings in the set of permissible strings, we do not necessarily guarantee that the inference rules have finite and fixed set of hypotheses. Both of these clash with the exercise's rule that each inference rule must have a *fixed* and finite number of premises, and cannot embed existential/universal quantification or set-building in the rule itself.

The challenge arises with the constructs of concatenation (e_1e_2) and Kleene star ($e*$). Each of these can potentially match infinitely many prefixes of a given string s, and so we would need rules that somehow collect all possible remainders in a single inference step—yet the exercise forbids placing set-comprehensions or unbounded quantifications in the premises or conclusion. This limitation makes it impossible to write finite-premise rules that correctly and completely describe the semantics of concatenation and Kleene star.

To see why, consider concatenation: we would need to capture all pairs of matches ($s' \in S_1, s'' \in S''$) where S_1 is the set of leftovers from matching e_1 and S_2 is the set of leftovers from then matching e_2. A single finite rule with no set-comprehension cannot do this. Similarly, Kleene star demands potentially repeated matches of e and an aggregation of suffixes from all possible ways of iterating. Expressing that within a single inference step—without "looping back" or enumerating an unbounded set of cases—cannot be done. Thus, under the imposed constraints, no system of rules can be both finite-premise and yield all suffix sets deterministically for those two operators.

Question assigned to the following page: 4

Exercise 3F-3. Equivalence [7 points]. I believe that $e_1 \sim e_2$ is decidable for this language of regular expressions. For a regular expression e, the set of all possible "remainders" after matching a prefix of s is determined exactly by the set of all prefixes that e can match.

Let $S_e(s)$ denote the set of all suffixes, given by the set $\{s_s \mid s = s_p :: s_s \wedge s_p \in L(e)\}$, where s_s & s_p denote the suffix and prefix respectively and $L(e)$ is the language of regular expression e.

Requiring $S_e(s) = S_d(s)$ for every string s is equivalent to the notion of being able to split every string s = uv, such that $u \in L(e) \iff u \in L(d)$. So, then the condition: $e_1 \sim e_2 \iff L(e_1) = L(e_2)$.

So we can follow this procedure:

- Convert each e_i to an NFA (nondeterministic finite automaton).
- Convert each NFA to a DFA (deterministic finite automaton) by the subset construction.
- Minimize each DFA (or else compare them without explicit minimization, using a standard graph-product approach).
- Check language equivalence of the two resulting DFAs (for instance, by checking whether their symmetric difference accepts any string).

So, we can do this in polynomial time in the sizes of the NFA/DFA making it decidable.

Question assigned to the following page: 5

Exercise 3F-4. SAT Solving [6 points]. The slow performance of these test cases reveals fundamental inefficiencies in the DPLL(T) implementation. The current architecture follows a naive "check-and-backtrack" approach where the SAT solver generates complete assignments before consulting the theory solver. This means the SAT solver might explore many theory-inconsistent assignments before finding a valid solution. The test cases (x > y) && (y > z) && (z = 10) && (x < 13/12) are particularly challenging because they combine transitive arithmetic relationships with tight numerical bounds, creating a large search space that the current implementation must explore exhaustively.

The most serious issue lies in the arithmetic theory solver (`arith.ml`), which uses a brute-force enumeration approach to find satisfying assignments. For each variable, it tries every possible value within the bounded range [-127, 128], leading to exponential complexity $O(256^n)$ where n is the number of variables. This approach completely ignores the structure of the arithmetic constraints and fails to employ any form of constraint propagation or early pruning.

Instead of waiting for the SAT solver to propose assignments, the arithmetic solver should proactively propagate constraints (e.g., deducing x > 10 from z = 10). Additionally, it should perform stronger transitive reasoning (i.e., if x > y and y > z, then immediately infer x > z), reducing the number of Boolean branches that need to be explored. Another key optimization is early conflict detection—for example, if x < 12 is added but x > 12 has already been derived, the solver should immediately return UNSAT, avoiding wasted search effort.