Exercise 3F-1

$$\frac{\vdash e_1 \ \texttt{matches} \ s_1 \ \texttt{leaving} \ s_2 \qquad \vdash e_2 \ \texttt{matches} \ s_2 \ \texttt{leaving} \ s_3}{\vdash e_1 e_2 \ \texttt{matches} \ s_1 \ \texttt{leaving} \ s_3} \ \texttt{concat}$$

$$\frac{\vdash e_1 \ \texttt{matches} \ s \ \texttt{leaving} \ s'}{\vdash e_1 \mid e_2 \ \texttt{matches} \ s \ \texttt{leaving} \ s'} \ \texttt{or1}$$

$$\frac{\vdash e_2 \ \texttt{matches} \ s \ \texttt{leaving} \ s'}{\vdash e_1 \mid e_2 \ \texttt{matches} \ s \ \texttt{leaving} \ s'} \ \texttt{or2}$$

$$\frac{}{\vdash e* \ \texttt{matches} \ s \ \texttt{leaving} \ s} \ \texttt{Kleene1}$$

$$\frac{\vdash e \ \texttt{matches} \ s_1 \ \texttt{leaving} \ s_2 \qquad \vdash e* \ \texttt{matches} \ s_2 \ \texttt{leaving} \ s_3}{\vdash e* \ \texttt{matches} \ s_1 \ \texttt{leaving} \ s_3} \ \texttt{Kleene2}$$

Exercise 3F-2

One observation is that for the judgment $\vdash e$ `matches` s `leaving` S, for some e and s, as s is always a finite string by construction, any element $s' \in S$ must be a substring of s, and they share the same suffix up to a certain point, i.e., $s = "x_1" :: "x_2" :: \cdots :: s'$. Therefore, S must be a finite set of strings with size at most $\texttt{length}(s) + 1$.

Tentative rules for concatenation:

$$\frac{\vdash e_1 \texttt{ matches } s_1 \texttt{ leaving } S_2 \qquad \forall s \in S_2. \vdash e_2 \texttt{ matches } s \texttt{ leaving } S_3(s)}{\vdash e_1 e_2 \texttt{ matches } s_1 \texttt{ leaving } \bigcup_{s \in S_2} S_3(s)} \; \texttt{concatS}$$

The second hypothesis is essentially defining one hypothesis for each $s \in S_2$, which seems to violate the requirement that there must be a finite and fixed set of hypotheses. In particular, since S_2 is always a finite sets, there are indeed only finitely many hypotheses in the above rule; while it's ambiguous to say whether this is a "fixed" set of hypotheses, depending on the exact definition of "fixed".

$$\frac{}{\vdash e* \texttt{ matches } s \texttt{ leaving } \{s\}} \; \texttt{KleeneS1}$$

$$\frac{\vdash e \texttt{ matches } s_1 \texttt{ leaving } S_2 \qquad \forall s \in S_2. \vdash e* \texttt{ matches } s \texttt{ leaving } S_3(s)}{\vdash e* \texttt{ matches } s_1 \texttt{ leaving } \bigcup_{s \in S_2} S_3(s)} \; \texttt{KleeneS2}$$

Similarly, if we accept the above uses of set of hypotheses, then the rules for Kleene stars can also be defined as above.

Exercise 3F-3

The equivalence is defined as follow in the problem statement. Let e_1, e_2 be two regular expressions:

$e_1 \sim e_2$ iff $\forall s \in S.(\vdash e_1 \texttt{ matches } s \texttt{ leaving } S_1) \wedge (\vdash e_2 \texttt{ matches } s \texttt{ leaving } S_2) \implies S_1 = S_2$.

One observation is that for the judgment $\vdash e \texttt{ matches } s \texttt{ leaving } S$, for some e and s, as s is always a finite string by construction, any element $s' \in S$ must be a substring of s, and they share the same suffix up to a certain point, i.e., $s = "x_1" :: "x_2" :: \cdots :: s'$. Therefore, S must be a finite set of strings with size at most $\texttt{length}(s) + 1$.

For fixed e and s, we can create a TM to check wether for a substring s' of s, we have $\vdash e_1 \texttt{ matches } s \texttt{ leaving } s'$ in finite steps, by enumerating all possible proof trees. Since there are only finitely many possible combinations of rules (No infinite loops).

Therefore, if S is a finite set of strings, then the whole procedure can be computed by enumerating each $s \in S$.

On the other hand, if S is infinite, i.e. all possible strings of an alphabet, $L*$, then it seems not to be computable, as structral induction on s would not work, and the TM would have to enumerate all possible strings. (Rice's theorem may be relevant?)

Exercise 3F-4

Experiments suggests that the reason of the slowdown is probably the last two test cases actually called the theory solver Arith.arith, even for only once in each.

In addition, since it's essentially a brute-force search, the time complexity is exponential to the number of variables. In these two test cases, there are 3 variables, with means the number of searches could be up to $256^3 \approx 10^7$.

To improve, I would then start with the arithmetic theory module arith.ml. Some nice alternative can be a branch-and-cut based modern integer programming solver, e.g., CPLEX, Gurobi, SCIP. Some relevant packages for OCaml are actually available: `https://ocaml.org/p/lp/0.0.2`, `https://opam.ocaml.org/packages/lp-gurobi/`.

Some potential defects:

- The theory solver is not guaranteed to return a correct answer (model) even it's satisfiable;
- No alerts of the first defect outside the arith module.