Exercise 3F-1

We introduce one rule for concatenation that first applies e_1 and then applies e_2:

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

We introduce two rules for or, one for each choice of expression to match with:

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

and:

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

We introduce two rules for $e\star$. First, we allow for zero applications of e:

$$\frac{}{\vdash e\star \text{ matches } s \text{ leaving } s}$$

Next, we inductively allow multiple applications of $e\star$:

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

Exercise 3F-2

I believe that this task cannot be accomplished in the current framework. Consider the concatenation e_1e_2. In order to determine the set that e_1e_2 leaves after matching s, we must apply e_2 to every item in the set that e_1 leaves. However, it doesn't seem possible to express this without putting a derivation inside of a set constructor. One attempt at expressing concatenation is the following rule that simply avoids e_2:

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

However, this is unsound as the regular expression ab now matches the string a. Next, we could consider applying both e_1 and e_2 to only S, and taking the intersection:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_1 \text{ matches } s \text{ leaving } S}{\vdash e_1e_2 \text{ matches } s \text{ leaving } S_1}$$

But this rule is both unsound and incomplete. The regular expression ab fails to match the string ab. And the string ab is erroneously matched by the regular expression aa.

Exercise 3F-3

This problem is decideable. We say that a regular expression e "accepts" s if e matches s leaving the empty string. Note that $e_1 \sim e_2$ iff they accept the same set of strings. It's well-known that regular expressions are equivalent to deterministic finite automata (DFA), and so determining if $e_1 \sim e_2$ is equivalent to determining if two DFAs accept the same set of strings. It is a classic result that this can be done with a "pairwise state reachability" strategy if the DFAs are viewed as edge-labelled graphs.

Exercise 3F-4

The last two testcases take a long time to calculate because they contain clauses that are essentially of the form $x > 0$. When faced with such a testcase, the given arithmetic theory solver must iterate all possibilities for x from -128 all the way up until it finds a satisfying assignment at 1. Indeed, the worstcase runtime of our arithmetic solver is $O(256^n)$ where n is the number of variables. If we are willing to relax conditions to allow for rational solutions (e.g., $x > 3 \,\&\&\, x < 4$ becomes satisfiable with $x = 3.5$), this module could be replaced with simplex, drastically decreasing the runtime.

The current DPLL module could be integrated more efficiently with the theory module by creating some kind of `SetTrue` functionality as in "DPLL(T): Fast Decision Procedures" by Ganzinger et al. In this way, the DPLL module could prune arithmetically infeasible branches early instead of fully exploring them and having them refuted only at the leaves of the search tree. However, upon closer examination of the slow cases, there is none to little backtracking done by the DPLL module, so this would yield little improvement, if any.

The only egregious defect in the provided code I can think of is the very limited range allowed for variables. e.g., $x > 200$ is unsatisfiable.