1 3F-1 Bookkeeping

- **0 pts** Correct

Exercise 3F-2. Regular Expression, Large-Step [10 points].

Rule for e_1e_2

$$\frac{\vdash e_1 \text{ matches } s_1 \text{ @ } s_2 \text{ @ } s'' \text{ leaving } s_2 \text{ @ } s'' \quad \vdash e_2 \text{ matches } s_2 \text{ @ } s'' \text{ leaving } s''}{\vdash e_1e_2 \text{ matches } s_1 \text{ @ } s_2 \text{ @ } s'' \text{ leaving } s'}$$

(where @ is the string concatenation operator.)

Rules for $e_1 \mid e_2$

$$\frac{\vdash e_1 \text{ matches } s_1 \text{ @ } s' \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s_1 \text{ @ } s' \text{ leaving } s'} \qquad \frac{\vdash e_2 \text{ matches } s_2 \text{ @ } s' \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s_2 \text{ @ } s' \text{ leaving } s'}$$

Rules for $e*$

$$\frac{}{\vdash e* \text{ matches } s \text{ leaving } s} \qquad \frac{\vdash e(e*) \text{ matches } s \text{ leaving } s'}{\vdash e* \text{ matches } s \text{ leaving } s'}$$

(One case for matching zero times, another for matching 1+ times)

Exercise 3F-3. 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: $\{x \mid \exists y. \vdash e \text{ matches } x \text{ leaving } y\}$. 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.

Part of doing research is getting stuck. When you get stuck, you must be able to recognize whether "you are just missing something" or "the problem is actually impossible".

I believe that our current framework does not support this "exhaustive" regular expression construct, as there isn't a way to express how to carry forward the resulting set of one regular expression to another without putting regular expressions in the set constructor.

My first attempt at e_1e_2:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \dfrac{s \in \{s_2' \text{ @ } s' | s_2' \text{ @ } s' \in S\}}{\vdash e_2 \text{ matches } s \text{ leaving } S'}}{\vdash e_1e_2 \text{ matches } s \text{ leaving } S'}$$

However, this doesn't produce the whole set, as S' can only tell you something about suffices for a particular $s \in S$ produced by e_1. Therefore, this is not sound.

2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

Exercise 3F-2. Regular Expression, Large-Step [10 points].

Rule for e_1e_2

$$\frac{\vdash e_1 \text{ matches } s_1 \text{ @ } s_2 \text{ @ } s'' \text{ leaving } s_2 \text{ @ } s'' \quad \vdash e_2 \text{ matches } s_2 \text{ @ } s'' \text{ leaving } s''}{\vdash e_1e_2 \text{ matches } s_1 \text{ @ } s_2 \text{ @ } s'' \text{ leaving } s'}$$

(where @ is the string concatenation operator.)

Rules for $e_1 \mid e_2$

$$\frac{\vdash e_1 \text{ matches } s_1 \text{ @ } s' \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s_1 \text{ @ } s' \text{ leaving } s'} \qquad \frac{\vdash e_2 \text{ matches } s_2 \text{ @ } s' \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s_2 \text{ @ } s' \text{ leaving } s'}$$

Rules for $e*$

$$\frac{}{\vdash e* \text{ matches } s \text{ leaving } s} \qquad \frac{\vdash e(e*) \text{ matches } s \text{ leaving } s'}{\vdash e* \text{ matches } s \text{ leaving } s'}$$

(One case for matching zero times, another for matching 1+ times)

Exercise 3F-3. 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: $\{x \mid \exists y. \vdash e \text{ matches } x \text{ leaving } y\}$. 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.

Part of doing research is getting stuck. When you get stuck, you must be able to recognize whether "you are just missing something" or "the problem is actually impossible".

I believe that our current framework does not support this "exhaustive" regular expression construct, as there isn't a way to express how to carry forward the resulting set of one regular expression to another without putting regular expressions in the set constructor.

My first attempt at e_1e_2:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \dfrac{s \in \{s_2' \text{ @ } s' | s_2' \text{ @ } s' \in S\}}{\vdash e_2 \text{ matches } s \text{ leaving } S'}}{\vdash e_1e_2 \text{ matches } s \text{ leaving } S'}$$

However, this doesn't produce the whole set, as S' can only tell you something about suffices for a particular $s \in S$ produced by e_1. Therefore, this is not sound.

Another attempt:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s \text{ leaving } \{s' | s_2 \ @ \ s' \in S\}}{\vdash e_1 e_2 \text{ matches } s \text{ leaving } \{s' | s_2 \ @ \ s' \in S\}}$$

In this case, I tried express the fact that e_2 must happen only after e_1, but somehow I feel it's a bit circular, since in order to know e_2's output, you need to somehow know what it does with s after e_1 through some other means.

Exercise 3F-4. Equivalence [7 points]. I believe that equivalence is undecidable in our implementation of regular expressions. The reason is that not all regular expressions will terminate. Take for example the regular expression `("a"*)*`, which matches zero or more instances of (zero or more instances of the character "a"). Assuming my semantics for the Kleene star (3F-2) are correct, this regular expression may become stuck, since matching it on the string `"a"` can cause it to match zero times, infinitely many times. It is possible to have the "inner" Kleene star not make any "progress" on the string such that. Therefore, given an arbitrary number of runs of the "outer" Kleene star, you can always add one more run that either matches or doesn't match. Although it's probably apparent that the resulting set of matches is $\{\emptyset,$ `"a"`$\}$, a truly "exhaustive" search would never terminate due to the "add-one-more loop" situation described previously—what if the next loop makes it behave differently? It won't, but what if? And if not, what if we add another star?

Therefore, since we cannot guarantee that all regular expressions in our implementation terminate, equivalence is undecidable since it reduces to the halting problem.

Exercise 3F-5. SAT Solving [6 points]. The last two tests took a long time because have arithmetic constraints involving more than two variables. An example could be:

```
x = y && x = z && z = 128
```

This is because the arithmetic theory prover uses "brute force" to test relations between variables, so the time to solve is potentially exponential in the number of variables.

The module I would rewrite first would be the arithmetic solver, namely the portion that performs the bounded search. The way it's implemented seems a lot like plain DPLL where the numbers are all treated as 8-bit collections of booleans—there isn't any acknowledgment of their higher-level properties. I believe a more efficient approach would be to implement some kind of "symbolic execution" for integers i.e. basic algebra substitution methods. The key would be to apply some of the algebraic "intuition" we humans use when solving simple algebra equations. For instance, even if I expand the above equation to a hundred variables all equalling `x`, most people could very quickly guess that the equation is still satisfiable. One approach could be to find the variable with concrete constraints, or those with the most constraints, and try to express other variables in terms of this one. We would need to write a set of inference rules to capture the algebraic relationships between integers, and use those to possibly infer the values (or at least the valid intervals) of variables. This is starting to sound a bit like refinement type inference...

3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

Another attempt:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s \text{ leaving } \{s' | s_2 \ @ \ s' \in S\}}{\vdash e_1 e_2 \text{ matches } s \text{ leaving } \{s' | s_2 \ @ \ s' \in S\}}$$

In this case, I tried express the fact that e_2 must happen only after e_1, but somehow I feel it's a bit circular, since in order to know e_2's output, you need to somehow know what it does with s after e_1 through some other means.

Exercise 3F-4. Equivalence [7 points]. I believe that equivalence is undecidable in our implementation of regular expressions. The reason is that not all regular expressions will terminate. Take for example the regular expression `("a"*)*`, which matches zero or more instances of (zero or more instances of the character "a"). Assuming my semantics for the Kleene star (3F-2) are correct, this regular expression may become stuck, since matching it on the string `"a"` can cause it to match zero times, infinitely many times. It is possible to have the "inner" Kleene star not make any "progress" on the string such that. Therefore, given an arbitrary number of runs of the "outer" Kleene star, you can always add one more run that either matches or doesn't match. Although it's probably apparent that the resulting set of matches is $\{\emptyset,$ `"a"`$\}$, a truly "exhaustive" search would never terminate due to the "add-one-more loop" situation described previously—what if the next loop makes it behave differently? It won't, but what if? And if not, what if we add another star?

Therefore, since we cannot guarantee that all regular expressions in our implementation terminate, equivalence is undecidable since it reduces to the halting problem.

Exercise 3F-5. SAT Solving [6 points]. The last two tests took a long time because have arithmetic constraints involving more than two variables. An example could be:

```
x = y && x = z && z = 128
```

This is because the arithmetic theory prover uses "brute force" to test relations between variables, so the time to solve is potentially exponential in the number of variables.

The module I would rewrite first would be the arithmetic solver, namely the portion that performs the bounded search. The way it's implemented seems a lot like plain DPLL where the numbers are all treated as 8-bit collections of booleans—there isn't any acknowledgment of their higher-level properties. I believe a more efficient approach would be to implement some kind of "symbolic execution" for integers i.e. basic algebra substitution methods. The key would be to apply some of the algebraic "intuition" we humans use when solving simple algebra equations. For instance, even if I expand the above equation to a hundred variables all equalling `x`, most people could very quickly guess that the equation is still satisfiable. One approach could be to find the variable with concrete constraints, or those with the most constraints, and try to express other variables in terms of this one. We would need to write a set of inference rules to capture the algebraic relationships between integers, and use those to possibly infer the values (or at least the valid intervals) of variables. This is starting to sound a bit like refinement type inference...

4 3F-4 Equivalence

- **0 pts** Correct

Another attempt:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \vdash e_2 \text{ matches } s \text{ leaving } \{s' | s_2 \ @ \ s' \in S\}}{\vdash e_1 e_2 \text{ matches } s \text{ leaving } \{s' | s_2 \ @ \ s' \in S\}}$$

In this case, I tried express the fact that e_2 must happen only after e_1, but somehow I feel it's a bit circular, since in order to know e_2's output, you need to somehow know what it does with s after e_1 through some other means.

Exercise 3F-4. Equivalence [7 points]. I believe that equivalence is undecidable in our implementation of regular expressions. The reason is that not all regular expressions will terminate. Take for example the regular expression `("a"*)*`, which matches zero or more instances of (zero or more instances of the character "a"). Assuming my semantics for the Kleene star (3F-2) are correct, this regular expression may become stuck, since matching it on the string `"a"` can cause it to match zero times, infinitely many times. It is possible to have the "inner" Kleene star not make any "progress" on the string such that. Therefore, given an arbitrary number of runs of the "outer" Kleene star, you can always add one more run that either matches or doesn't match. Although it's probably apparent that the resulting set of matches is $\{\emptyset,$ `"a"`$\}$, a truly "exhaustive" search would never terminate due to the "add-one-more loop" situation described previously—what if the next loop makes it behave differently? It won't, but what if? And if not, what if we add another star?

Therefore, since we cannot guarantee that all regular expressions in our implementation terminate, equivalence is undecidable since it reduces to the halting problem.

Exercise 3F-5. SAT Solving [6 points]. The last two tests took a long time because have arithmetic constraints involving more than two variables. An example could be:

```
x = y && x = z && z = 128
```

This is because the arithmetic theory prover uses "brute force" to test relations between variables, so the time to solve is potentially exponential in the number of variables.

The module I would rewrite first would be the arithmetic solver, namely the portion that performs the bounded search. The way it's implemented seems a lot like plain DPLL where the numbers are all treated as 8-bit collections of booleans—there isn't any acknowledgment of their higher-level properties. I believe a more efficient approach would be to implement some kind of "symbolic execution" for integers i.e. basic algebra substitution methods. The key would be to apply some of the algebraic "intuition" we humans use when solving simple algebra equations. For instance, even if I expand the above equation to a hundred variables all equalling `x`, most people could very quickly guess that the equation is still satisfiable. One approach could be to find the variable with concrete constraints, or those with the most constraints, and try to express other variables in terms of this one. We would need to write a set of inference rules to capture the algebraic relationships between integers, and use those to possibly infer the values (or at least the valid intervals) of variables. This is starting to sound a bit like refinement type inference...

One of the issues I noticed with the arithmetic theory prover is that the numbers are limited to a very small finite subset of the integers. For instance, we would expect `x = 128 && x - y = -127` to be satisfiable (since integer subtraction is closed), but it's not since y would have to take on a value outside of the arithmetic solver's bounds. This is a direct side-effect of needing to constrain the bounds due to the brute force approach discussed earlier.

5 3F-5 SAT Solving

- **0 pts** Correct