1 3F-1 Bookkeeping

- 0 pts Correct

Peer Review ID: 68538793 — enter this when you fill out your peer evaluation via gradescope

Exercise 3F-2. Regular Expression, Large-Step

$$\begin{array}{lcll}
e & ::= & \text{"x"} & \text{singleton — matches the character } \hat{x} \\
 & | & \mathsf{empty} & \text{skip — matches the empty string} \\
 & | & e_1\ e_2 & \text{concatenation — matches } e_1 \text{ followed by } e_2 \\
 & | & e_1 \mid e_2 & \text{or — matches } e_1 \text{ or } e_2 \\
 & | & e* & \text{Kleene star — matches 0 or more occurrence of } e \\
 & & & \\
 & | & . & \text{matches any single character} \\
 & | & [\text{"x"} - \text{"y"}] & \text{matches any character between } \hat{x} \text{ and } \hat{y} \text{ inclusive} \\
 & | & e+ & \text{matches 1 or more occurrences of } e \\
 & | & e? & \text{matches 0 or 1 occurrence of } e
\end{array}$$

Concatenation:

Concatenation is simply applying the match twice, switching the string and the match character (or set of characters) . We see if we can match e_1. If we do we look for a match of e_2 on the remaining sub-string.

$$\frac{\vdash \mathsf{e_1\ matches}\ s\ \mathsf{leaving}\ s' \qquad \vdash \mathsf{e_2\ matches\ s'\ leaving\ s''}}{\vdash \mathsf{e_1e_2\ matches}\ s\ \mathsf{leaving}\ s''}$$

Or:

The OR operation is just two separate applications of the same rule. It's clearly symmetrical as we either match e_1 or e_2 (or none). The key here is we do not generate an additional s", instead we search the same string s for both cases producing only s'.

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

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

Kleene star (*):

The Kleene star is a little more challenging. I break this into two cases, match zero, match many. We begin with match zero.

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

Okay that was easy. Lets look at the harder case. We want to match each occurrence. I could match the string, produce s', produce the next string and match s", but this is all looking very similar to a"while" expression. We want to keep matching"while" there's more sub-string to search. We can basically copy paste the while rule as they hinge on the same recursive idea.

$$\frac{\vdash \mathsf{e\ matches}\ s\ \mathsf{leaving}\ s' \qquad \mathsf{e{*}\ matches}\ s'\ \mathsf{leaving}\ s''}{\vdash \mathsf{e{*}\ matches}\ s\ \mathsf{leaving}\ s''}$$

2 3F-2 Regular Expressions, Large Step

- 0 pts Correct

Peer Review ID: 68538793 — enter this when you fill out your peer evaluation via gradescope

These look very similar to while **false** and while **true**. Very suspicious!

Exercise 3F-3.

Concatenation:

1. This new structure requires that we match a single string, leaving a set of strings. Trying to do this type of judgement on concatenation cannot be done. A simple way to illustrate this:

$$\frac{\vdash \mathsf{e_1 \ matches} \ s \ \mathsf{leaving} \ S' \qquad \vdash \mathsf{e_2 \ matches \ S' \ leaving \ S''}}{\vdash \mathsf{e_1 e_2 \ matches} \ s \ \mathsf{leaving} \ S''}$$

   (1) Is clearly incorrect, we do not generate the intermediate term s', and it's not possible to consider a match on the set S'. Assuming s' is an element of the set S' doesn't help us because this fails the "finite" requirement. There are an unknown number of elements in S'.

2. $$\frac{\vdash \mathsf{e_1 \ matches} \ s \ \mathsf{leaving} \ S' \qquad \vdash \mathsf{e_2 \ matches \ s \ leaving \ S''}}{\vdash \mathsf{e_1 e_2 \ matches} \ s \ \mathsf{leaving} \ S''}$$

   (2) Has the merit that s exists and produces real sets S' and S". The issue is that this is an incorrect implementation of concatenation. S" is not the set of strings which are matched by e_1e_2.

3. $$\frac{\vdash \mathsf{e_1 \ matches} \ s \ \mathsf{leaving} \ S' \qquad \forall s' \in S' \quad \vdash \mathsf{e_2 \ matches} \ \mathsf{s'} \ \mathsf{leaving \ S''}}{\vdash \mathsf{e_1 e_2 \ matches} \ s \ \mathsf{leaving} \ S''}$$

   (3) Has the benefits of being correct, but is essentially a rewrite of (1) and as the second hypothesis is quantified and not finite.

It seems there is a specific problem preventing us from writing these deterministic rules of the form: match string leaving set. The problem is trying to transfer information from the result of applying e_1 to the input of e_2. I see this as a type error, where we're trying to fit a square peg in a round hole (and the hole radii isn't a factor of $\sqrt{2}$ larger). The results are in the form of a set, but the input must be a single string. Therefore under the constraints provided I posit it is not possible to write a rule which describes concatenation.

Kleene star (*):

The Kleene star for no matches is easy again.

$$\frac{}{\vdash \mathsf{e*\ matches}\ s\ \mathsf{leaving}\ \{s\}}$$

The problem arises in the following rules. We want something like:

$$\frac{\vdash \mathsf{e\ matches}\ s\ \mathsf{leaving}\ S' \qquad \forall s' \in S' \quad \mathsf{e*\ matches}\ s'\ \mathsf{leaving}\ S''}{\vdash \mathsf{e*\ matches}\ s\ \mathsf{leaving}\ s''}$$

Immediately, we see this is the same problem we saw earlier. We can't get this square (the set S') peg into the round hole (the single string matched by: e* matches s' leaving S"). Therefore by the same logic we can conclude it's not possible.

Exercise 3F-4.

Equivalence:

Three sentence answer: We can transform any regular expression to a DFA (deterministic finite automaton), which once composed we can minimize causing the DFA to be unique. This means any two expressions $e_1 \sim e_2$ will have $DFA_{minimized_{e_1}} = DFA_{minimized_{e_2}}$. Therefore we do not need to compute the possibly infinite sets S_1 and S_2 as we can prove that $e_1 \equiv e_2$ by only comparing their minimized DFA.

Some context that is longer than three sentences: A minimized DFA represents the least processing steps needed to pattern match. There are existing methods to reduce an arbitrary regular expression to a DFA and methods to minimize an arbitrary DFA. Because we can reduce regular expressions to DFA form, we know they do not form a Turing Complete language. This means we have (unfortunately) not solved the halting problem, but we can compare our expressions.

Exercise 3F-5.

T_1

```
(((a * a) + (b * b)) = (c * c)) && (a > 128) && (b > 128) && (c > 128)
```

$T_{1.1}$

```
(((a * a) + (b * b)) = (c * c)) && (a > 128) && (b > 128) && (c > 128)
```

T_2

```
(((a * a) + (b * b)) = (c * c)) && (a > -127) && (b > -127) && (c > -127)
```

$T_{2.1}$

3 3F-3 Regular Expressions and Sets

- 0 pts Correct

Kleene star (*):

The Kleene star for no matches is easy again.

$$\frac{}{\vdash \mathsf{e*\ matches}\ s\ \mathsf{leaving}\ \{s\}}$$

The problem arises in the following rules. We want something like:

$$\frac{\vdash \mathsf{e\ matches}\ s\ \mathsf{leaving}\ S' \qquad \forall s' \in S' \quad \mathsf{e*\ matches}\ s'\ \mathsf{leaving}\ S''}{\vdash \mathsf{e*\ matches}\ s\ \mathsf{leaving}\ s''}$$

Immediately, we see this is the same problem we saw earlier. We can't get this square (the set S') peg into the round hole (the single string matched by: e* matches s' leaving S"). Therefore by the same logic we can conclude it's not possible.

Exercise 3F-4.

Equivalence:

Three sentence answer: We can transform any regular expression to a DFA (deterministic finite automaton), which once composed we can minimize causing the DFA to be unique. This means any two expressions $e_1 \sim e_2$ will have $DFA_{minimized_{e_1}} = DFA_{minimized_{e_2}}$. Therefore we do not need to compute the possibly infinite sets S_1 and S_2 as we can prove that $e_1 \equiv e_2$ by only comparing their minimized DFA.

Some context that is longer than three sentences: A minimized DFA represents the least processing steps needed to pattern match. There are existing methods to reduce an arbitrary regular expression to a DFA and methods to minimize an arbitrary DFA. Because we can reduce regular expressions to DFA form, we know they do not form a Turing Complete language. This means we have (unfortunately) not solved the halting problem, but we can compare our expressions.

Exercise 3F-5.

T_1

```
(((a * a) + (b * b)) = (c * c)) && (a > 128) && (b > 128) && (c > 128)
```

$T_{1.1}$

```
(((a * a) + (b * b)) = (c * c)) && (a > 128) && (b > 128) && (c > 128)
```

T_2

```
(((a * a) + (b * b)) = (c * c)) && (a > -127) && (b > -127) && (c > -127)
```

$T_{2.1}$

4 3F-4 Equivalence

- 0 pts Correct

Kleene star (*):

The Kleene star for no matches is easy again.

$$\frac{}{\vdash \mathsf{e*\ matches}\ s\ \mathsf{leaving}\ \{s\}}$$

The problem arises in the following rules. We want something like:

$$\frac{\vdash \mathsf{e\ matches}\ s\ \mathsf{leaving}\ S' \qquad \forall s' \in S' \quad \mathsf{e*\ matches}\ s'\ \mathsf{leaving}\ S''}{\vdash \mathsf{e*\ matches}\ s\ \mathsf{leaving}\ s''}$$

Immediately, we see this is the same problem we saw earlier. We can't get this square (the set S') peg into the round hole (the single string matched by: e* matches s' leaving S"). Therefore by the same logic we can conclude it's not possible.

Exercise 3F-4.

Equivalence:

Three sentence answer: We can transform any regular expression to a DFA (deterministic finite automaton), which once composed we can minimize causing the DFA to be unique. This means any two expressions $e_1 \sim e_2$ will have $DFA_{minimized_{e_1}} = DFA_{minimized_{e_2}}$. Therefore we do not need to compute the possibly infinite sets S_1 and S_2 as we can prove that $e_1 \equiv e_2$ by only comparing their minimized DFA.

Some context that is longer than three sentences: A minimized DFA represents the least processing steps needed to pattern match. There are existing methods to reduce an arbitrary regular expression to a DFA and methods to minimize an arbitrary DFA. Because we can reduce regular expressions to DFA form, we know they do not form a Turing Complete language. This means we have (unfortunately) not solved the halting problem, but we can compare our expressions.

Exercise 3F-5.

T_1

```
(((a * a) + (b * b)) = (c * c)) && (a > 128) && (b > 128) && (c > 128)
```

$T_{1.1}$

```
(((a * a) + (b * b)) = (c * c)) && (a > 128) && (b > 128) && (c > 128)
```

T_2

```
(((a * a) + (b * b)) = (c * c)) && (a > -127) && (b > -127) && (c > -127)
```

$T_{2.1}$

```
(((a * a) + (b * b) + (l * l)) = (c * c)) && (a > -127) && (b > -127) &&
    (c > -127) && (l > -127)
```

$$T_{2.2}$$

```
((a != a) && (a < 7)) || ((a > 6) && (a < 7))
```

1. These final two test cases require enumeration on three variables. This causes the values required to be checked to be 256*256*256 in the worst case (this occurs if the inequalities on x, y, and z are unsatisfiable). To show this I developed a few test cases T_1, $T_{1.1}$, T_2 and $T_{2.1}$. T_1 and $T_{1.1}$ are UNSAT, but T_2 and $T_{2.1}$ are SAT. When we run these on our solver we find that T_1 takes considerably longer than T_2. This difference is exaggerated when considering $T_{1.1}$ and $T_{2.1}$. While $T_{2.1}$ will terminate producing SAT fairly quickly, I left $T_{1.1}$ running on my PC for over 40 minutes and it did not terminate. This is because DPLL does not handle arithmetic expressions efficiently. We must implement EUF abstractions rather than iterate over the values. Finally, when considering $T_{2.2}$ I found the lack of an abstraction causes the solver to continuously try to satisfy the first two literals, even though arithmetically we can see it's obviously UNSAT. This results in the arithmetic solver being called many tens of thousands of times if not more for a simple query. From these experiments I show that even for small (256 valued) integers the brute force method quickly become intractable. I would rewrite the arithmetic solver, and I argue the best instrument to make the solver efficient is to utilize EUF.

2. I would rewrite the **arith.ml** module. I select this one because once we begin solving arithmetic constraints of any reasonable size this becomes the largest bottleneck. I think the strongest change to this module would be to change it from a bounded model to an EUF based approach. This would allow us to reason about much more complex questions without needing to enumerate the entire bound. Once we can check arithmetic without enumeration we can increase the bounds on what size of variables our model checker can verify without a dramatic increase in run-time.

3. I believe a serious defect is the inability to generate stronger formulas when new constraints are added to the boolean model. I ran this query, T_e, for 12 hours, it never finished. Checking any arithmetic of the form: $arith_{unsat} \vee arith_{unsat}$ results in extremely poor performance. The performance issues arise from many spurious SAT results due to poor linking between the arithmetic theory solver and the boolean SAT solver. The solver should learn from the arithmetic solver that $(a > 6) \rightarrow !(a < 7)$, but it continues to learn terms of the form $(!(a > 6)||(a < 7)||!(a < 7)||(a > 6))$ (this reduces to **true**, but many of the test cases are weak without returning **true** directly). These weak constraints basically add nothing to prune our search space.

   $$T_e$$

   ```
   ((a > 6) && (a < 7)) || ((a > 6) && (a < 7))
   ```

5 3F-5 SAT Solving

- **0 pts** Correct

Peer Review ID: 68538793 — enter this when you fill out your peer evaluation via gradescope