1 3F-1 Bookkeeping

- **0 pts** Correct

Peer Review ID: 68408531 — enter this when you fill out your peer evaluation via gradescope

3F-2. We introduce the following large-step operational semantics rules of inference for the other three primal regular expressions:

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } s_1 \quad \vdash e_2 \texttt{ matches } s_1 \texttt{ leaving } s'}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } s'}$$

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

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

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

$$\frac{\vdash e \texttt{ matches } s \texttt{ leaving } s_1 \quad \vdash e* \texttt{ matches } s_1 \texttt{ leaving } s'}{\vdash e* \texttt{ matches } s \texttt{ leaving } s'}$$

3F-3. Within the given framework, it is likely impossible to provide such inference rules. Intuitively, describing the possible suffixes left by the regular expression e_1e_2 requires looking at all suffixes left by e_1, then considering all possible suffixes e_2 leaves when matching any of these, so some judgment including e_2 is necessary in the final set constructor . A similar argument applies to $e*$ looking at e and $e*$ recursively.

Consider, for example, the candidate rule

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S_1 \quad \vdash e_2 \texttt{ matches } s \texttt{ leaving } S_2}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } S_1 \cap S_2}$$

for e_1e_2. This is unsound with respect to the intuitive notion of regular expression matching. Specifically, $\vdash$ "aa" `matches` "a" `leaving` {""} is intuitively false, yet it is derivable since $\vdash$ "a" `matches` "a" `leaving` {""} is derivable and {""} $\cap$ {""} = {""}, so we have the derivation

$$\frac{\vdash \text{"}a\text{"} \texttt{ matches } \text{"}a\text{"} \texttt{ leaving } \{\text{""}\} \quad \vdash \text{"}a\text{"} \texttt{ matches } \text{"}a\text{"} \texttt{ leaving } \{\text{""}\}}{\vdash \text{"}aa\text{"} \texttt{ matches } \text{"}a\text{"} \texttt{ leaving } \{\text{""}\}}.$$

Likewise, consider the candidate rule

$$\frac{\vdash ee* \texttt{ matches } s \texttt{ leaving } S}{\vdash e* \texttt{ matches } s \texttt{ leaving } \{s\} \cup S}$$

for $e*$. This is incomplete with respect to the intuitive notion of regular expression matching. Specifically, $\vdash$ `empty` $*$ `matches` s `leaving` $\{s\}$ is true yet not derivable: whatever rule we have for concatenation would necessarily involve a judgement of `empty`$*$ matched against the suffixes left by `empty`, but this is exactly the initial judgement, reaching an infinite recursion and implying no finite derivation tree is possible. □

Peer Review ID: 68408531 — enter this when you fill out your peer evaluation via gradescope

2 3F-2 Regular Expressions, Large Step

- 0 pts Correct

Peer Review ID: 68408531 — enter this when you fill out your peer evaluation via gradescope

3F-2. We introduce the following large-step operational semantics rules of inference for the other three primal regular expressions:

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } s_1 \quad \vdash e_2 \texttt{ matches } s_1 \texttt{ leaving } s'}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } s'}$$

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

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

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

$$\frac{\vdash e \texttt{ matches } s \texttt{ leaving } s_1 \quad \vdash e* \texttt{ matches } s_1 \texttt{ leaving } s'}{\vdash e* \texttt{ matches } s \texttt{ leaving } s'}$$

3F-3. Within the given framework, it is likely impossible to provide such inference rules. Intuitively, describing the possible suffixes left by the regular expression e_1e_2 requires looking at all suffixes left by e_1, then considering all possible suffixes e_2 leaves when matching any of these, so some judgment including e_2 is necessary in the final set constructor . A similar argument applies to $e*$ looking at e and $e*$ recursively.

Consider, for example, the candidate rule

$$\frac{\vdash e_1 \texttt{ matches } s \texttt{ leaving } S_1 \quad \vdash e_2 \texttt{ matches } s \texttt{ leaving } S_2}{\vdash e_1 e_2 \texttt{ matches } s \texttt{ leaving } S_1 \cap S_2}$$

for e_1e_2. This is unsound with respect to the intuitive notion of regular expression matching. Specifically, $\vdash$ "aa" `matches` "a" `leaving` $\{""\}$ is intuitively false, yet it is derivable since $\vdash$ "a" `matches` "a" `leaving` $\{""\}$ is derivable and $\{""\} \cap \{""\} = \{""\}$, so we have the derivation

$$\frac{\vdash \text{"}a\text{"} \texttt{ matches } \text{"}a\text{"} \texttt{ leaving } \{\text{""}\} \quad \vdash \text{"}a\text{"} \texttt{ matches } \text{"}a\text{"} \texttt{ leaving } \{\text{""}\}}{\vdash \text{"}aa\text{"} \texttt{ matches } \text{"}a\text{"} \texttt{ leaving } \{\text{""}\}} .$$

Likewise, consider the candidate rule

$$\frac{\vdash ee* \texttt{ matches } s \texttt{ leaving } S}{\vdash e* \texttt{ matches } s \texttt{ leaving } \{s\} \cup S}$$

for $e*$. This is incomplete with respect to the intuitive notion of regular expression matching. Specifically, $\vdash$ `empty` $*$ `matches` s `leaving` $\{s\}$ is true yet not derivable: whatever rule we have for concatenation would necessarily involve a judgement of `empty`$*$ matched against the suffixes left by `empty`, but this is exactly the initial judgement, reaching an infinite recursion and implying no finite derivation tree is possible. □

Peer Review ID: 68408531 — enter this when you fill out your peer evaluation via gradescope

3 3F-3 Regular Expressions and Sets

- 0 pts Correct

Peer Review ID: 68408531 — enter this when you fill out your peer evaluation via gradescope

3F-4. Equivalence of regular expressions is decidable. Given regular expressions r_1, r_2, it suffices to determine whether they describe the same language. Since any regular language is matched by a unique DFA with a minimal number of states (up to isomorphism), we can simply compute these minimal DFAs then check that they are isomorphic by looking at all possible relabellings. Concretely, this could be done applying Thompon's construction to get NFAs, applying the Rabin-Scott powerset construction to produce equivalent DFAs, then using Hopcraft's algorithm to minimizes these DFAs. □

3F-5. Considering the last two test cases, they take comparatively longer to run due to the large number of arithmetic variables causing a time-consuming brute force search in the integer arithmetic solver. More concretely, consider the input

$$(x > y) \ \&\& \ (y > z) \ \&\& \ (z = 10) \ \&\& \ (x < 12)$$

for test-35. This input is expanded to be solely in terms of $\leq$, $\geq$, and $=$, resulting in

$$(x \geq y) \ \&\& \ !(x = y) \ \&\& \ (y \geq z) \ \&\& \ !(y = z) \ \&\& \ (z = 10) \ \&\& \ (x \leq 12) \ \&\& \ !(x = 12).$$

Each clause involving arithmetic is then assigned a boolean variable, producing the CNF formula

$$T_0 \ \&\& \ !T_1 \ \&\& \ T_2 \ \&\& \ !T_3 \ \&\& \ T_4 \ \&\& \ T_5 \ \&\& \ !T_6.$$

The DPLL algorithm is run on this formula to find a boolean assignment. Since each variable occurs in a unit clause, each is immediately assigned to the make said unit clause true, and the obvious satisfying assignment is quickly produced.

After this, the assignment to the theory variables is passed to the theory solver i,e, the solver in arith.ml. Since the solver simply brute force considers all assignments to x, y, z within the range $[-127, 128]$, a total of 2^{24} possibilities are searched before unsatisfiability is determined. Every other test case contains at most two arithmetic variables, hence the comparative runtime difference.

Since no solution is found by the theory, the negation of all the current assignments of the theory variable is added to the formula, resulting in the new formula

$$(T_3 \ || \ !T_4 \ || \ !T_2 \ || \ !T_5 \ || \ T_1 \ || \ T_6 \ || \ !T_0) \ \&\& \ T_0 \ \&\& \ !T_1 \ \&\& \ T_2 \ \&\& \ !T_3 \ \&\& \ T_4 \ \&\& \ T_5 \ \&\& \ !T_6.$$

DPLL is run agan, and the unit clauses immediately determine assignments once more. For each unit clause, after assignment the literal is removed from any other clause containing it, eventually leaving this first clause empty and causing DPLL to report unsatisfiability overall.

In this run, as well as for test-36, all aspects solely involving DPLL are quick due to the unit clauses, so the only reasonable way to improve the performance here would be to rewrite arith.ml. In these particular cases, all arithmetic terms are linear, so delegating to a solver for linear bounded arithmetic would be beneficial. More specifically, this could be accomplished by first collecting all linear arithmetic clauses, passing these to a solver which uses, say, the Simplex method, then only brute forcing the non-linear clauses after a solution to the linear clauses is found.

4 3F-4 Equivalence

- 0 pts Correct

3F-4. Equivalence of regular expressions is decidable. Given regular expressions r_1, r_2, it suffices to determine whether they describe the same language. Since any regular language is matched by a unique DFA with a minimal number of states (up to isomorphism), we can simply compute these minimal DFAs then check that they are isomorphic by looking at all possible relabellings. Concretely, this could be done applying Thompon's construction to get NFAs, applying the Rabin-Scott powerset construction to produce equivalent DFAs, then using Hopcraft's algorithm to minimizes these DFAs. $\square$

3F-5. Considering the last two test cases, they take comparatively longer to run due to the large number of arithmetic variables causing a time-consuming brute force search in the integer arithmetic solver. More concretely, consider the input

$$(x > y) \ \&\& \ (y > z) \ \&\& \ (z = 10) \ \&\& \ (x < 12)$$

for test-35. This input is expanded to be solely in terms of $\leq$, $\geq$, and $=$, resulting in

$$(x \geq y) \ \&\& \ !(x = y) \ \&\& \ (y \geq z) \ \&\& \ !(y = z) \ \&\& \ (z = 10) \ \&\& \ (x \leq 12) \ \&\& \ !(x = 12).$$

Each clause involving arithmetic is then assigned a boolean variable, producing the CNF formula

$$T_0 \ \&\& \ !T_1 \ \&\& \ T_2 \ \&\& \ !T_3 \ \&\& \ T_4 \ \&\& \ T_5 \ \&\& \ !T_6.$$

The DPLL algorithm is run on this formula to find a boolean assignment. Since each variable occurs in a unit clause, each is immediately assigned to the make said unit clause true, and the obvious satisfying assignment is quickly produced.

After this, the assignment to the theory variables is passed to the theory solver i,e, the solver in arith.ml. Since the solver simply brute force considers all assignments to x, y, z within the range $[-127, 128]$, a total of 2^{24} possibilities are searched before unsatisfiability is determined. Every other test case contains at most two arithmetic variables, hence the comparative runtime difference.

Since no solution is found by the theory, the negation of all the current assignments of the theory variable is added to the formula, resulting in the new formula

$$(T_3 \ || \ !T_4 \ || \ !T_2 \ || \ !T_5 \ || \ T_1 \ || \ T_6 \ || \ !T_0) \ \&\& \ T_0 \ \&\& \ !T_1 \ \&\& \ T_2 \ \&\& \ !T_3 \ \&\& \ T_4 \ \&\& \ T_5 \ \&\& \ !T_6.$$

DPLL is run agan, and the unit clauses immediately determine assignments once more. For each unit clause, after assignment the literal is removed from any other clause containing it, eventually leaving this first clause empty and causing DPLL to report unsatisfiability overall.

In this run, as well as for test-36, all aspects solely involving DPLL are quick due to the unit clauses, so the only reasonable way to improve the performance here would be to rewrite arith.ml. In these particular cases, all arithmetic terms are linear, so delegating to a solver for linear bounded arithmetic would be beneficial. More specifically, this could be accomplished by first collecting all linear arithmetic clauses, passing these to a solver which uses, say, the Simplex method, then only brute forcing the non-linear clauses after a solution to the linear clauses is found.

As for outright defects in the code, the most egregious is the inclusion of pure variable elimination in dpll.ml. In DPLL(T), since theory variables may be logically dependent, it's not valid to assume any pure variable can be set to true. Consider, for example, the formula

$$(x > 10 \mid\mid x < 3) \ \&\& \ (x > 10 \mid\mid x < 9) \ \&\& \ (x < 7).$$

The term $(x > 10)$ is pure, but setting it to true would result in the formula being incorrectly reported as unsatisfiable, since $(x > 10)$ would contradict $(x < 7)$. While the implementation in this homework may still produce the correct answer, as any inconsistencies in the theory result in a total restart, the use is still unsound in general, and in any case, it would lead to a greater number of inconsistent assignments.

5 3F-5 SAT Solving

- 0 pts Correct