Question assigned to the following page: 2

Exercise 3F-1. Regular Expression, Large-Step [10 points].

Concatenation:

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

OR:

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

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

Kleene Star:

$$\vdash e^* \text{ matches } s \text{ leaving } s$$

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

Peer Review ID: 311013226 — enter this when you fill out your peer evaluation via gradescope

Question assigned to the following page: 3

Exercise 3F-2. Regular Expression and Sets [5 points].

It does not appear to be possible to define rules of inference for $e*$ or $e1e2$ to deterministically capture all suffices. This is because the Kleene star and concatenation patterns either need to use a set constructor to quantify their premises to capture all possibilities. The attempts below show failed attempts for the Kleene star, but a concatenation attempt also fails for a similar reason because we either need to use an infinite set of hypothesis to capture all of the leftovers, or we need to use a set-comprehension in the conclusion.

Attempt 1 (Kleene star) Base case to capture 0 occurrences of e:

$$\frac{}{\vdash e^* \text{ matches } s \text{ leaving } \{s\}}$$

The recursive case is intended to capture cases where e matches at least once, leaving a set of possible suffices S'. The second premise attempts to apply $e*$ to the string again to capture repeated matches of e. Then the conclusion takes the union of these sets.

$$\frac{\vdash e \text{ matches } s \text{ leaving } S' \quad \vdash e^* \text{ matches } s \text{ leaving } S''}{\vdash e^* \text{ matches } s \text{ leaving } S' \cup S''}$$

By the rule defined for us, the set of S' will represent all the single matches of e in s. However, in the second predicate, S'' will use the base rule to simply leave a set consisting of s. This does not capture all of the multiple matches for multiple instances of the pattern e so it is incomplete because it does not use the leftovers, (the set S').

Attempt 2 (Kleene star) We use the same base case as defined above in attempt 1, but adjust our recursive case to attempt to use the leftovers:

$$\frac{\vdash e \text{ matches } s \text{ leaving } S' \quad \vdash e^* \text{ matches } s' \text{ leaving } S''}{\vdash e^* \text{ matches } s \text{ leaving } S' \cup S''}$$

However, in this case our second predicate only matches a single suffix s', where $s' \in S'$. This means that again we are missing some of the leftover elements in S', and this is incomplete. We would like to add an additional premise that captures the idea of using every leftover in S', but that would either require infinitely many premises (one for each s', recursively matching the 0 instance) or a set-comprehension in the conclusion which is prohibited.

Peer Review ID: 311013226 — enter this when you fill out your peer evaluation via gradescope

Question assigned to the following page: 4

Exercise 3F-3. Equivalence [7 points].

Equivalence for regular expressions is decidable. Since every regular expression corresponds to a finite automaton, we can construct a finite state machine that tracks the set of possible leftover suffixes for each input string. Comparing two regular expressions then reduces to checking whether their corresponding automata induce the same function from inputs to suffix sets. Since this can be done algorithmically in finite time, equivalence is decidable.

Peer Review ID: 311013226 — enter this when you fill out your peer evaluation via gradescope

Question assigned to the following page: 5

Exercise 3F-4. SAT Solving [6 points].

The last two tests are:

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

and

$$(x > y) \ \& \ (y > z) \ \& \ (z = 10) \ \& \ (x < 13)$$

These test cases each have three arithmetic variables and nested constraints, where the value of x depends on the value of y and the value of y depends on the value of z. DPLL(T) solvers extend the DPLL algorithm, which handles pure boolean logic, to also handle integer arithmetic. DPLL uses optimizations such as simplifying unit clauses that contain only a single literal and identifying pure variables that are always or never true to simplify boolean problems. DPLL(T) converts integer and mixed constraints to boolean constraints, but does not use pure variable elimination and alters unit propagation. This means that the DPLL(T) solver can fix $z = 10$, removing one degree of freedom, but still $y > 10$ and $x > y > 10$. Unlike pure boolean constraints, y and x still have multiple choices which must be tested explicitly. Ideally, the solver would use $z = 10$ to filter impossible values for x and y, but then it would still need to check for a quadratic number of possibilities by testing all values of $y > 10$ and then all values $x > y$. This search process increases the time to solve these constraints compared to pure boolean formulas or formulas without nested constraints.

The module that is key for implementing the solver is in `arith.ml`. In its current implementation, it uses a brute force approach to try all possible values in a naive manner. One optimization is constraint propagation where we filter out impossible values before looping through assignments. In the case of $x > y > 10$, we would not bother trying values of $y < 10$ or $x < 11$. This reduces the search space for possible satisfiable values.

Another implementation we could implement is choosing the variable with the smallest feasible range first. In the first test case shown, x has no feasible values, so we should immediately return unsatisfiable. In the second, x's only feasible values is 12, which can drastically reduce the search space.

Other optimizations could also be implemented, but these two should dramatically improve the performance compared to the current brute force approach.

Peer Review ID: 311013226 — enter this when you fill out your peer evaluation via gradescope