Question assigned to the following page: 2

Exercise 3F-1: Large-Step Operational Semantics for Regular Expressions

We define the judgment:

$$\vdash e \text{ matches } s \text{ leaving } s'$$

meaning that the regular expression e matches some prefix of the string s, leaving the suffix s'.

1. Singleton (Given)

$$\frac{s = "x" :: s'}{\vdash "x" \text{ matches } s \text{ leaving } s'}$$

2. Empty (Given)

$$\frac{}{\vdash \text{empty matches } s \text{ leaving } s}$$

3. Concatenation $(e_1 e_2)$

The concatenation $e_1\, e_2$ matches a string s if e_1 matches a prefix of s, leaving a suffix s', and then e_2 matches a prefix of s', leaving a suffix s''

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

4. Or $(e_1 \mid e_2)$

The or expression $e_1 \mid e_2$ matches a string s if either e_1 matches s leaving s' or e_2 matches s leaving s'.

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s'}{\vdash (e_1 \mid e_2) \text{ matches } s \text{ leaving } s'} \qquad \frac{\vdash e_2 \text{ matches } s \text{ leaving } s'}{\vdash (e_1 \mid e_2) \text{ matches } s \text{ leaving } s'}$$

5. Kleene Star (e^*)

The Kleene star e^* matches a string s if either:

Zero occurrences:

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

One or more occurrences:

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

Question assigned to the following page: 3

Exercise 3F-2: Regular Expression and Sets

Problem: We need to provide deterministic operational semantics rules for the Kleene star (e^*) and concatenation ($e_1 e_2$) that return the set of all possible suffixes S for the judgment:

$$\vdash e \text{ matches } s \text{ leaving } S$$

However, it is **not possible** to correctly define these rules in the given framework without introducing non-finite or recursive structures.

Argument: It Cannot Be Done Correctly in the Given Framework
The challenge arises because the Kleene star (e^*) and concatenation ($e_1 e_2$) inherently involve non-determinism and recursion. Capturing all possible suffixes S requires considering an unbounded number of possibilities, which cannot be expressed with a finite and fixed set of hypotheses in the given framework.

Why It Fails

1. **Kleene Star** (e^*)**:** The Kleene star matches zero or more occurrences of e. Each occurrence of e can split the string s into different prefixes and suffixes, resulting in an exponential number of possible suffix sets. To capture all possible suffixes, we would need to recursively apply the Kleene star rule, which cannot be expressed with a finite set of hypotheses.
2. **Concatenation** ($e_1 e_2$)**:** Concatenation requires matching e_1 against all possible prefixes of s. For each such match, e_2 must match the remaining suffix, leading to a combinatorial explosion of possible suffix sets. This complexity cannot be captured with a finite and fixed set of hypotheses.

Two "Wrong" Attempts

Attempt 1: Kleene Star (e^*)
Incorrect Rule:

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

***Why It Fails:**

- This rule attempts to recursively apply the Kleene star to all suffixes $s' \in S$.
- It is **unsound** because it assumes the set S' can be computed for each s' in a finite way, which is impossible without unbounded recursion.
- It violates the requirement that each inference rule must have a finite and fixed set of hypotheses.

Question assigned to the following page: 3

Attempt 2: Concatenation $(e_1 e_2)$
Incorrect Rule:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } S \quad \forall s' \in S, \vdash e_2 \text{ matches } s' \text{ leaving } S'}{\vdash (e_1 e_2) \text{ matches } s \text{ leaving } \bigcup_{s' \in S} S'}$$

Why It Fails:

- This rule attempts to compute the set of all possible suffixes by iterating over $s' \in S$ and applying e_2 to each s'.
- It is **incomplete** because it does not account for the fact that e_1 and e_2 can match in multiple overlapping ways.
- Like the Kleene star attempt, it violates the finite and fixed hypotheses requirement.

Question assigned to the following page: 4

Exercise 3F-3: Equivalence of Regular Expressions

The equivalence relation $e_1 \sim e_2$ for regular expressions is defined as:

$$e_1 \sim e_2 \iff \forall s \in S. \vdash e_1 \text{ matches } s \text{ leaving } S_1 \wedge \vdash e_2 \text{ matches } s \text{ leaving } S_2 \implies S_1 = S_2$$

This means that two regular expressions e_1 and e_2 are equivalent if, for every string s, the sets of possible suffixes S_1 and S_2 produced by matching e_1 and e_2 against s are identical.

Claim: $e_1 \sim e_2$ is Decidable

Unlike the equivalence of IMP commands, which is undecidable due to its connection to the halting problem, the equivalence of regular expressions is **decidable**. This is because regular expressions can be converted into finite automata (NFAs or DFAs), and the equivalence of finite automata is decidable.

How to Compute $e_1 \sim e_2$

1. **Convert e_1 and e_2 to NFAs**:
   - Use the standard construction (e.g., Thompson's algorithm) to convert e_1 and e_2 into NFAs N_1 and N_2.
2. **Convert NFAs to DFAs**:
   - Use the subset construction to convert N_1 and N_2 into DFAs D_1 and D_2.
3. **Minimize the DFAs**:
   - Use the Hopcroft minimization algorithm to minimize D_1 and D_2 into their canonical forms D_1' and D_2'.
4. **Check for Isomorphism**:
   - Compare the minimized DFAs D_1' and D_2'. If they are isomorphic (i.e., they have the same structure), then $e_1 \sim e_2$. Otherwise, they are not equivalent.

Why This Works

I would say that, regular expressions, NFAs, and DFAs are all equivalent in expressive power, and equivalence between them is well-defined. The process of converting regular expressions to NFAs, NFAs to DFAs, and minimizing DFAs is algorithmic and guaranteed to terminate, and the isomorphism check between minimized DFAs is also algorithmic and efficient.

Question assigned to the following page: 5

Exercise 3F-4: SAT Solving with DPLL(T)

The last two included tests take significantly longer due to the increased complexity of the constraints and the interactions between the Boolean and theory solvers in the DPLL(T) framework. Several factors contribute to the extended runtime:

1. **Complex Arithmetic Constraints:** Both tests involve chains of inequalities and equalities, requiring the arithmetic solver to reason about variable bounds. In cases where variable bounds are tight and the solution space is narrow, the solver may perform numerous failed attempts before identifying conflicts or valid assignments.
2. **Delayed Conflict Detection:** In Test 35, the solver must deduce that no assignment of x, y, z can simultaneously satisfy $(x > y) \wedge (y > z) \wedge (z = 10) \wedge (x < 12)$. If the solver explores many partial assignments without early detection of the arithmetic conflict, it wastes time on fruitless branches.
3. **Inefficient Theory Propagation:** The interaction between the SAT and arithmetic solvers may be inefficient. If the theory solver does not promptly propagate inequalities (e.g., deducing that $x > 11$ from $(x > y) \wedge (y > 10)$), the SAT solver continues to explore inconsistent assignments.
4. **Lack of Effective Conflict Clause Learning:** Without robust conflict clause learning, the solver repeatedly explores similar conflicts. Efficient clause learning helps prune the search space, but if implemented poorly, significant backtracking overhead occurs.
5. **Suboptimal Variable Selection Heuristics:** The solver may lack advanced heuristics like VSIDS (Variable State Independent Decaying Sum) or phase saving, resulting in inefficient exploration of the search space.

Detailed Analysis of Tests 35 and 36

Test 35:

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

- From $(z = 10)$, we get $z = 10$.
- From $(y > z)$, it follows that $y > 10 \Rightarrow y \geq 11$.
- From $(x > y)$, it follows that $x > 11 \Rightarrow x \geq 12$.
- The constraint $(x < 12)$ conflicts with $x \geq 12$, making the formula unsatisfiable.

Question assigned to the following page: 5

Why it is slow: The solver may explore assignments for x below 12 before realizing no solution exists, especially if conflict detection is delayed.

Test 36:

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

- From $(z = 10)$, we have $z = 10$.
- From $(y > z)$, it follows that $y > 10 \Rightarrow y = 11$.
- From $(x > y)$, it follows that $x > 11 \Rightarrow x \geq 12$.
- The constraint $(x < 13)$ allows $x = 12$, satisfying all constraints.

Why it is slow despite being satisfiable:

- The solver still explores invalid assignments (e.g., $x = 11$) before finding a valid one.
- Without strong propagation, the solver does not immediately deduce that x must be at least 12.
- Backtracking occurs when earlier decisions on y or x fail, adding overhead.

Which Module to Rewrite for Performance Improvement

The most critical module to rewrite is the theory propagation and conflict detection module. This is because, early detection of arithmetic conflicts would prevent exploring invalid branches and improved theory propagation reduces the number of SAT-level decisions and backtracks.

Improvements:

1. **Stronger Bound Propagation:** Implement an arithmetic constraint solver that aggressively propagates variable bounds. For example, from $(y > z) \wedge (z = 10)$, immediately infer $y > 10$, which, combined with $(x > y)$, gives $x > 11$. Detect the conflict with $(x < 12)$ early.
2. **Enhanced Conflict Clause Learning:** Generate minimal conflict clauses that directly block invalid assignments. This helps the solver avoid revisiting the same conflicts.
3. **Variable Selection Heuristics:** Use heuristics like VSIDS or phase saving to prioritize decisions on variables involved in arithmetic constraints.

Code Defect

A particularly egregious defect in the provided code is the failure to propagate arithmetic-derived conflicts back to the SAT solver immediately. Therefore, we must ensure that after every arithmetic assignment, the theory solver updates all related variable bounds and reports conflicts instantly.