Question assigned to the following page: 2

EXERCISE 3F-1: Regular Expression, Large-Step

Regular expressions serve as an abstraction for string matching. This response defines **large-step operational semantics rules of inference** for three primary forms of regular expressions:

- **Concatenation (e1 e2)**
- **Alternation (e1 | e2)**
- **Kleene Star (e*)**

The semantics are represented by the judgment:

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

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

Inference Rules

1. Concatenation (e1 e2)

A concatenation e1 e2 means that e1 must match some prefix of s, leaving s', and then e2 must match s', leaving s'':

$$\frac{\vdash e1 \text{ matches } s \text{ leaving } s' \quad \vdash e2 \text{ matches } s' \text{ leaving } s''}{\vdash e1e2 \text{ matches } s \text{ leaving } s''}$$

where s'' is the suffix left after e2 matches s'. This ensures that e1 matches the beginning portion of s, and e2 continues matching from where e1 left off.

2. Alternation (e1 | e2)

Alternation allows either e1 or e2 to match. This introduces two rules:

$$\frac{\vdash e1 \text{ matches } s \text{ leaving } s'}{\vdash e1|e2 \text{ matches } s \text{ leaving } s'}$$

$$\frac{\vdash e2 \text{ matches } s \text{ leaving } s'}{\vdash e1|e2 \text{ matches } s \text{ leaving } s'}$$

This ensures that **both possibilities are valid**, meaning that e1 | e2 behaves non-deterministically by allowing either e1 or e2 to match.

Question assigned to the following page: 2

3. Kleene Star (e*)

Kleene Star allows zero or more occurrences of e:

- If e* matches without consuming characters:

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

- If e matches some prefix of s, and e* continues matching:

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

where s'' is the suffix left after further matches. This ensures that e* can match multiple times in a row, allowing recursive application of the rule.

Addressing Edge Cases

To ensure full rigor, we must explicitly consider cases where matching **fails** or where empty strings are involved.

Failure Cases

- If e1 does not match s, then e1 e2 **cannot match**.
- If e1 | e2 is given an input that neither e1 nor e2 can match, then it fails to match.

Handling Empty Strings

- **Concatenation with empty:**

$$\vdash emptye \text{ matches } s \text{ leaving } s$$

The empty string empty acts as an identity element in concatenation.

- **Alternation with empty:**

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

This ensures that if empty is an option, it allows s to remain unchanged.

- **Kleene star on empty:**

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

This rule ensures that e* matches the empty case correctly.

Question assigned to the following page: 2

Examples

Example 1: Concatenation (e1 e2)

If e1 = "ab" and e2 = "c", then e1 e2 matches "abc", leaving an empty suffix.

Example 2: Kleene Star (e*)

If e = "a" and s = "aaa", then e* can match the first two a's, leaving "a" as the unmatched suffix.

Example 3: Handling Empty Cases

If e = empty, then empty* matches any input s leaving s unchanged.

This completes the **large-step operational semantics rules** for **Exercise 3F-1B**, now addressing **both formal notation requirements and edge cases**.

Question assigned to the following page: 3

EXERCISE 3F-2: Regular Expression and Sets

In this exercise, we attempt to define deterministic operational semantics for **concatenation (e1 e2)** and **Kleene star (e*)**, ensuring that each rule has a **finite and fixed set of hypotheses**. However, we demonstrate that this is **impossible** within the given framework.

1. Incorrect Rule Attempt for Concatenation (e1 e2)

A deterministic operational semantics must ensure that every inference rule has a **finite and fixed** set of premises. However, concatenation inherently requires checking **all possible suffixes from e1**, leading to an unbounded number of premises. The following rule illustrates this issue:

A natural attempt for defining concatenation would be:

$$\frac{\vdash e1 \text{ matches } s \text{ leaving } S, \forall s' \in S, \vdash e2 \text{ matches } s' \text{ leaving } S'}{\vdash e1e2 \text{ matches } s \text{ leaving } \bigcup_{s' \in S} S'}$$

Why This Rule Fails

- The set S contains **all possible suffixes from e1**, meaning the rule must check **every possible s' for e2**.
- This results in an **unbounded** number of premises ($\forall s' \in S$), violating the **fixed hypothesis constraint**.
- The rule is **incomplete** because it does not guarantee termination for all cases.

2. Incorrect Rule Attempt for Kleene Star (e*)

A common approach for Kleene star is:

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

Why This Rule Fails

- The rule requires e* to **recursively apply** on **all possible suffixes s'**, leading to **infinite expansion**.
- Similar to concatenation, this rule introduces an **unbounded number of premises**, violating the **finite hypothesis constraint**.

Question assigned to the following page: 3

- The framework **explicitly disallows recursion in inference rules**, preventing e* from being expressed deterministically.
- Without recursion, **there is no finite way to define e*** while ensuring that it correctly captures all possible suffixes.

3. Conclusion: The Task is Impossible

- The given operational semantics framework **requires inference rules with a finite and fixed set of premises**.
- Both concatenation (e1 e2) and Kleene star (e*) **introduce an unbounded number of premises**, making it **impossible to capture all possible suffixes deterministically**.
- Because the framework does not allow **recursive or infinite premises**, any attempt to define inference rules for e* and e1 e2 either **fails to capture all cases** (incompleteness) or **violates the fixed hypothesis constraint** (unsoundness).

Thus, **deterministic operational semantics for e* and e1 e2 cannot be formulated correctly within this framework** without fundamentally changing its structure.

Question assigned to the following page: 4

EXERCISE 3F-3: Equivalence

The equivalence relation e1 ∼ e2 for regular expressions is defined as:

e1∼e2 ⟺ ∀s∈S,⊢e1 matches s leaving S1∧⊢e2 matches s leaving S2 ⟹ S1=S2

This problem asks whether e1 ∼ e2 is **decidable or undecidable**.

Equivalence is Decidable

Unlike the equivalence relation for **IMP commands** (which is undecidable because it reduces to the halting problem), **regular expression equivalence is decidable**. This follows from automata theory:

1. **Regular expressions can be converted into finite automata**: Every regular expression corresponds to a **deterministic finite automaton (DFA)** or **nondeterministic finite automaton (NFA)**.
2. **Automata equivalence is decidable**: Given two DFAs, we can check whether they **accept the same language** using **state minimization and difference computation**.
3. **Equivalence Checking Algorithm**:
    - Convert e1 and e2 into **minimal DFAs**.
    - Compare their state transition structures.
    - If the resulting DFAs are identical, then e1 ∼ e2.
    - If they differ, then e1 and e2 match different sets of suffices, proving e1 ≠ e2.

Thus, **e1 ∼ e2 is computable** in polynomial time using standard DFA minimization techniques, making **regular expression equivalence decidable**.

Question assigned to the following page: 5

EXERCISE 3F-4: SAT Solving

The last two included tests take significantly longer to complete due to inefficiencies in the **DPLL(T)** (Davis-Putnam-Logemann-Loveland with Theories) solver, specifically in the **integration between the SAT solver and the arithmetic theory solver**. Below is a detailed analysis of the root cause and necessary optimizations.

1. Why Do the Last Two Tests Take Longer?

1.1 Causes of Slow Performance

1. **Bounded Integer Arithmetic in arith.ml is Inefficient**
    - The arithmetic solver **exhaustively searches** all possible integer values for each variable between -127 and 128.
    - Instead of using **constraint propagation**, it iterates through **every possible assignment**, leading to **exponential complexity**.
2. **DPLL Algorithm (dpll.ml) Lacks Efficient Conflict Learning**
    - While **unit propagation and pure literal elimination are implemented**, the solver lacks **conflict-driven clause learning (CDCL)**.
    - **Excessive backtracking** occurs because the solver does not effectively prune invalid branches early.
3. **Theory Solver Integration (main.ml) is Incomplete**
    - The **SAT solver finds a Boolean model first**, then **checks arithmetic constraints separately**, causing redundant evaluations.
    - The placeholder FIXME in solver.ml suggests that the **SAT and arithmetic solvers are not properly synchronized**, worsening performance.
4. **Test Cases Likely Contain Heavy Arithmetic Constraints**
    - The slow tests likely contain **many arithmetic constraints** that require checking multiple integer values.
    - This triggers **repeated calls to the inefficient arithmetic solver**, compounding the slowdown.

Question assigned to the following page: 5

2. Proposed Performance Optimization

To improve solver performance, the **first module to rewrite** is **arith.ml (Arithmetic Theory Solver).**

2.1 How to Improve Performance

1. **Replace Exhaustive Search with Constraint Propagation**
    - Instead of iterating through all values from -127 to 128, use **interval propagation** to eliminate invalid assignments early.
    - Implement **bound tightening** to **shrink search space dynamically** based on known constraints.
2. **Modify DPLL to Integrate Arithmetic Reasoning Earlier**
    - Modify main.ml so that **arithmetic constraints are considered during SAT solving**, reducing unnecessary evaluations.
    - Use **incremental theory solving** to avoid rechecking the same constraints.
3. **Enable Early Conflict Detection in DPLL**
    - Implement **non-chronological backtracking (backjumping)** to **skip over already-explored invalid states**.
    - Improve **variable selection heuristics (VSIDS)** to prioritize decision variables that lead to faster conflict detection.

3. Identified Code Defect (Potential Bonus Point)

Defect: Arithmetic Solver is Unnecessarily Exhaustive

Location: arith.ml

- **Issue:** The solver iterates **through all possible values** for each variable, even when an early termination condition is met.
- **Impact:** Causes **exponential slowdowns** for larger test cases with multiple constraints.
- **Fix:**
    - Implement **constraint propagation** to prune invalid variable assignments before they are tested.

Question assigned to the following page: 5

- Use **graph-based constraint solving techniques** instead of brute-force enumeration.

4. Conclusion

- **The last two tests are slow due to inefficient arithmetic constraint solving (arith.ml).**
- **The egregious defect is the brute-force arithmetic solver**, which should be replaced with **constraint propagation and interval analysis**.
- **Rewriting the theory solver interface** and **improving integration with DPLL** will significantly enhance efficiency.

These changes will reduce redundant evaluations and improve solver performance on large test cases.