1 3F-1 Bookkeeping

- 0 pts Correct

Peer Review ID: 68535107 — enter this when you fill out your peer evaluation via gradescope

Exercise 3F-2. Regular Expression, Large-Step [10 points]. Regular Expressions are commonly used as abstractions for string matching. Here is an abstract grammar for regular expressions:

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

We will call the first five cases the *primary* forms of regular expressions. The last four cases can be defined in terms of the first five. We also give an abstract grammar for strings (modeled as lists of characters):

s	$::=$	nil	empty string
	$\mid$	"x" $::$ s	string with first character $\hat{x}$ and other characters s

We write "bye" as shorthand for "b" :: "y" :: "e" :: nil. This exercise requires you to give large-step operational semantics rules of inference related to regular expressions matching strings. We introduce a judgment:

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

The interpretation of the judgment is that the regular expression e matches some prefix of the string s, leaving the suffix s' unmatched. If $s' = \mathsf{nil}$ then r matched s exactly. Examples:

$$\vdash \text{"h"}(\text{"e"}+)\ \mathsf{matches}\ \text{"hello"}\ \mathsf{leaving}\ \text{"llo"}$$

Note that this operational semantics may be considered *non-deterministic* because we expect to be able to derive all three of the following:

$$\begin{aligned}
&\vdash (\text{"h"} \mid \text{"e"})*\ \mathsf{matches}\ \text{"hello"}\ \mathsf{leaving} && \text{"ello"}\\
&\vdash (\text{"h"} \mid \text{"e"})*\ \mathsf{matches}\ \text{"hello"}\ \mathsf{leaving} && \text{"hello"}\\
&\vdash (\text{"h"} \mid \text{"e"})*\ \mathsf{matches}\ \text{"hello"}\ \mathsf{leaving} && \text{"llo"}
\end{aligned}$$

Here are two rules of inference:

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

Give large-step operational semantics rules of inference for the other three primal regular expressions.

Solution: The rules for e_1e_2, $e_1|e_2$, and $e*$ are below:

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } s_1 \quad \vdash e_2 \textsf{ matches } s_1 \textsf{ leaving } s_2}{\vdash e_1e_2 \textsf{ matches } s \textsf{ leaving } s_2}\ \textsf{Concatenation}$$

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } s_1}{\vdash e_1|e_2 \textsf{ matches } s \textsf{ leaving } s_1}\ \textsf{Or1}$$

$$\frac{\vdash e_2 \textsf{ matches } s \textsf{ leaving } s_2}{\vdash e_1|e_2 \textsf{ matches } s \textsf{ leaving } s_2}\ \textsf{Or2}$$

$$\frac{}{\vdash e* \ \textsf{ matches } s \textsf{ leaving } s}\ \textsf{Kleene1}$$

$$\frac{\vdash e \textsf{ matches } s \textsf{ leaving } s_1 \quad \vdash e* \ \textsf{ matches } s_1 \textsf{ leaving } s_2}{\vdash e* \ \textsf{ matches } s \textsf{ leaving } s_2}\ \textsf{Kleene2}$$

2 3F-2 Regular Expressions, Large Step

- 0 pts Correct

Exercise 3F-3. Regular Expression and Sets [5 points]. We want to update our operational semantics for regular expressions to capture multiple suffices. We want our new operational semantics to be deterministic — it return the set of all possible answers from the single-answer operational semantics above. We introduce a new judgment:

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

And use rules of inference like the following:

$$\frac{}{\vdash \text{"x"} \textsf{ matches } s \textsf{ leaving } \{s' \mid s = \text{"x"} :: s'\}} \qquad \frac{}{\vdash \textsf{empty matches } s \textsf{ leaving } \{s\}}$$

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } S \quad \vdash e_2 \textsf{ matches } s \textsf{ leaving } S'}{\vdash e_1 \mid e_2 \textsf{ matches } s \textsf{ leaving } S \cup S'}$$

You must do one of the following:

- *either* give operational semantics rules of inference for $e*$ and e_1e_2. You may *not* place a derivation inside a set constructor, as in: $\{x \mid \exists y.\ \vdash e \textsf{ matches } x \textsf{ leaving } y\}$. Each inference rules must have a finite and fixed set of hypotheses.
- *or* argue in one or two sentences that it cannot be done correctly in the given framework. Back up your argument by presenting two attempted but "wrong" rules of inference and show that each one is either unsound or incomplete with respect to our intuitive notion of regular expression matching.

Part of doing research is getting stuck. When you get stuck, you must be able to recognize whether "you are just missing something" or "the problem is actually impossible".

Solution:

1-2 Sentence Argument: This **cannot** be done correctly in the given framework because the rules for e_1e_2 and $e*$ must include a variable number of hypotheses, which violates the requirement that the number of hypotheses must be fixed. For example, if $\vdash e_1 \textsf{ matches } s \textsf{ leaving } S$, the we must try matching e_2 to *all* such $s' \in S$; likewise, if $\vdash e \textsf{ matches } s \textsf{ leaving } S$, then we must try matching $e*$ to *all* $s' \in S$.

"Wrong" Rules of Inference: Below are two attempted but "wrong" rules of inference that are both incomplete with respect to our intuitive notion of regular expression matching. This rules avoid using a variable number of hyptheses by only matching e_2 and $e*$ to a single element of S.

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } S \quad \vdash e_2 \textsf{ matches } s_1 \in S \textsf{ leaving } S'}{\vdash e_1e_2 \textsf{ matches } s \textsf{ leaving } S'} \textsf{ Concatenation}$$

$$\frac{\vdash e \textsf{ matches } s \textsf{ leaving } S \quad \vdash e* \textsf{ matches } s_1 \in S \textsf{ leaving } S'}{\vdash e* \textsf{ matches } s \textsf{ leaving } S' \cup \{s\}} \textsf{ Kleene}$$

(**Note:** In the above rules, we define s_1 to be least element of S in a lexicographical ordering of S. Lexicographic ordering is defined to be the standard dictionary ordering of strings, specifically:

$$\text{"a"}, \text{"b"}, ..., \text{"z"}, \text{"aa"}, \text{"ab"}, ... \text{"az"}, \text{"bb"}, ..., \text{"zz"}, \text{"aaa"}, ...$$

In any case, S must be a finite set, as s has finite length and there cannot be an infinite number of suffices of a finite-length string. All finite sets can be well-ordered such that there exists a least element [1]. The lexicographic ordering presented above is just one such ordering.)

We see that the Concatenation rule is incomplete as follows. Consider the string $s =$ "hello", and suppose we want to match the regular expression (("h"|"e")+)("e"|"l") to s. ("h"|"e")+ matches s leaving $S = \{$"ello","llo"$\}$. Because we use lexicographic ordering to select s_1, we use "llo" as s_1 and match the regular expression ("e"|"l") to s_1, leaving $S' = \{$"lo"$\}$. This is not complete, however, because (("h"|"e")+)("e"|"l") also matches s leaving "llo", meaning that S' should be the set $\{$"lo","llo"$\}$, not just $\{$"lo"$\}$.

We see that the Kleene rule is incomplete as follows. Consider the string $s =$ "aab", and suppose we want to match the regular expression ("a"|"aa")$*$ to s. ("a"|"aa") matches s leaving $S = \{$"ab","b"$\}$. Because we use lexicographic ordering to select s_1, we use "b" as s_1 and match the regular expression ("a"|"aa")* to s_1. In cases like this where e does *not* match a string s, we assume that we have a base case rule that matches $e*$ to s leaving $\{s\}$. This means that the set S' will be $\{$"b"$\}$, and the conclusion of the Kleene rule will match ("a"|"aa")$*$ to $s =$ "aab" leaving $\{$"aab","b"$\}$. This is not complete however, because ("a"|"aa")$*$ also matches $s =$ "aab" leaving "ab", meaning that S' should be the set $\{$"aab","ab","b"$\}$, not just $\{$"aab","b"$\}$.

References

[1] E. S. Keeping, “A note on the well-ordering of sets,” *Mathematics Magazine*, vol. 33, no. 1, pp. 43–45, 1959. [Online]. Available: http://www.jstor.org/stable/3029469

[2] M. Almeida, N. Moreira, and R. Reis, “Testing the equivalence of regular expressions.”

[3] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Dpll (t): Fast decision procedures,” in *International Conference on Computer Aided Verification.* Springer, 2004, pp. 175–188.

3 3F-3 Regular Expressions and Sets

- 0 pts Correct

Peer Review ID: 68535107 — enter this when you fill out your peer evaluation via gradescope

Exercise 3F-4. Equivalence [7 points]. In the class notes (usually marked as "optional material" for the lecture component of the class but relevant for this question) we defined an equivalence relation $c_1 \sim c_2$ for IMP commands. Computing equivalence turned out to be undecideable: $c \sim c$ iff c halts. We can define a similar equivalence relation for regular expressions: $e_1 \sim e_2$ iff $\forall s \in S.\ \vdash e_1$ $\mathsf{matches}$ s $\mathsf{leaving}$ $S_1 \ \wedge\ \vdash e_2$ $\mathsf{matches}$ s $\mathsf{leaving}$ $S_2 \implies S_1 = S_2$ (note that we are using an "updated" operational semantics that returns the set of all possible matched suffices, as in the previous problem).

You must *either* claim that $e_1 \sim e_2$ is undecideable by reducing it to the halting problem *or* explain in two or three sentences how to compute it. You may assume that I the reader is familiar with the relevant literature.

Solution: The equivalence of e_1 and e_2 can be computed by first constructing two non-deterministic finite automata (NFA's) N_1 and N_2, which accept the same strings matched by e_1 and e_2, respectively. Then, we convert N_1 and N_2 to two *deterministic* finite automata (DFA's) labeled D_1 and D_2, respectively, such that N_1 is equivalent to D_1 and N_2 is equivalent to D_2. We then minimize both D_1 and D_2 (convert them to forms with minimum numbers of states), and because equivalent minimized DFA's are unique up to isomorphism, the minimized versions of D_1 and D_2 can be compared using a standard representation to determine if $S_1 = S_2$ [2].

References

[1] E. S. Keeping, "A note on the well-ordering of sets," *Mathematics Magazine*, vol. 33, no. 1, pp. 43–45, 1959. [Online]. Available: http://www.jstor.org/stable/3029469

[2] M. Almeida, N. Moreira, and R. Reis, "Testing the equivalence of regular expressions."

[3] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Dpll (t): Fast decision procedures," in *International Conference on Computer Aided Verification.* Springer, 2004, pp. 175–188.

4 3F-4 Equivalence

- 0 pts Correct

Peer Review ID: 68535107 — enter this when you fill out your peer evaluation via gradescope

Exercise 3F-5. SAT Solving [6 points]. Why do the last two included tests take such a comparatively long time? Impress me with your knowledge of DPLL(T) — feel free to use information from the assigned reading or related papers, not just from the lecture slides. I am looking for a reasonably detailed answer. Include a discussion of which single module you would rewrite first to improve performance, as well as how you would change that module.

Potential bonus point: The provided code contains at least one fairly egregious defect. Comment.

Solution: The last two included tests take a comparatively long time because unit propagation in the code does not utilize the theory of arithmetic to eliminate theory clauses and literals. In normal DPLL, when a unit clause is encountered, the single literal in the clause is assigned to true. Then, every clause containing that literal is removed, and the complement of the literal is removed from every clause in which this complement appears. In DPLL(T) with the theory of arithmetic, however, unit propagation can continue even further. For example, in test case 35, we have $(x > y)$ && $(y > z)$ && $(z = 10)$ && $(x < 12)$. By assertions to the bounded arithmetic theory, the DPLL(T) algorithm can transitively infer that $(x >= 12)$ because it must be that $(y >= 11)$ (since $(y > z)$), $(x > y)$, and all arithmetic literals must be integers. This means that, if the theory clause $(x < 12)$ were labeled $T4$, DPLL(T) would be able to conclude $!T4$ and immediately see that the formula is unsatisfiable. This is exactly the kind of enhancement to unit propagation mentioned on page 6 of "DPLL(T): Fast Decision Procedures", where the authors state, "In DPLL(T), ... additional literals can be set to false as a consequence of the assertions made to the theory solver" [3]. The functionality to make such conclusions using the arithmetic theory does not exist in the `Dpll` and `Arith` modules. Without them, the bounded arithmetic theory solver will need to try a much larger space of values for x and y, whereas it could have concluded that the formula was unsatisfiable by making assertions to the theory of arithmetic during unit propagation.

Likewise, for test case 36, we have $(x > y)$ && $(y > z)$ && $(z = 10)$ && $(x < 13)$. Just as in test case 35, DPLL(T) should be able to transitively infer that $(x >= 12)$ by assertions to the theory of arithmetic, allowing it to remove the clauses $(y > z)$ and $(z = 10)$ as well as change $(x > y)$ to $(x >= 12)$. This eliminates the need to search for values of y; now the theory of arithmetic must only search for values of x that satisfy $(x >= 12)$ && $(x < 13)$, which has the solution $x = 12$.

To solve this issue, I would modify the `Dpll` module, specifically in the `handle_unit_clauses` function. In this function, I would add an additional call to the arithmetic theory when a unit clause is propagated to determine what new clauses have been made true or false by the addition of the unit clause. I would then add these new clauses to the list of clauses and remove any unnecessary clauses before re-running the `handle_unit_clauses` function. This would also require the addition of a function to perform inference on the current list of clauses in the `Arith` module to discover what new clauses have become true or false when propagating a unit clause.

Bonus Point: The defect in the code mentioned in the problem statement is that the `Dpll` module eliminates pure variables when it should *not* do so. This is because theory terms may be dependent, unlike propositional logic terms which are independent. We saw this in lecture through the example query $(x > 10 \mid\mid x < 3)$ && $(x > 10 \mid\mid x < 9)$ && $(x < 7)$. Although $x > 10$ only appears positively in the query, we cannot mark it as true in the model and remove all clauses containing it because $x > 10$ and $x < 7$ are dependent terms; they cannot be true at the same time. This will lead the code to conclude that the query is unsatisfiable, when in reality we could satisfy it with the model $x = 2$.

Submission. Turn in the formal component of the assignment as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment via the `autograder.io` website.

References

[1] E. S. Keeping, “A note on the well-ordering of sets,” *Mathematics Magazine*, vol. 33, no. 1, pp. 43–45, 1959. [Online]. Available: http://www.jstor.org/stable/3029469

[2] M. Almeida, N. Moreira, and R. Reis, “Testing the equivalence of regular expressions.”

[3] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “Dpll (t): Fast decision procedures,” in *International Conference on Computer Aided Verification.* Springer, 2004, pp. 175–188.

5 3F-5 SAT Solving

- 0 pts Correct

Peer Review ID: 68535107 — enter this when you fill out your peer evaluation via gradescope