1 3F-1 Bookkeeping

- 0 pts Correct

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope

gradescope

Exercise 2. [10 points]. The large-step rules are as follows. For concatenation:

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } s'' \qquad \vdash e_2 \textsf{ matches } s'' \textsf{ leaving } s'}{\vdash e_1\ e_2 \textsf{ matches } s \textsf{ leaving } s'}\ ;$$

for or:

$$\frac{\vdash e_1 \textsf{ matches } s \textsf{ leaving } s'}{\vdash e_1 \mid e_2 \textsf{ matches } s \textsf{ leaving } s'}\ , \qquad \frac{\vdash e_2 \textsf{ matches } s \textsf{ leaving } s'}{\vdash e_1 \mid e_2 \textsf{ matches } s \textsf{ leaving } s'}\ ;$$

for Kleene star:

$$\frac{\vdash \textsf{empty} \mid (e\ e*) \textsf{ matches } s \textsf{ leaving } s'}{\vdash e* \textsf{ matches } s \textsf{ leaving } s'}\ .$$

Exercise 3. [5 points]. The deterministic large-step rules are as follows. For Kleene star:

$$\frac{\vdash e \textsf{ matches } s \textsf{ leaving } S \qquad S = \varnothing, \{s\}}{\vdash e* \textsf{ matches } s \textsf{ leaving } \{s\}}\ ,$$

$$\frac{\begin{cases} \vdash e \textsf{ matches } s \textsf{ leaving } S \qquad S \neq \varnothing, \{s\} \qquad s' = L_s(S) \\ \vdash e* \textsf{ matches } s' \textsf{ leaving } S' \\ \vdash N(e \mathbin{\&} !(\textsf{empty}) \mathbin{\&} !(s - s'))\ e* \textsf{ matches } s \textsf{ leaving } S'' \end{cases}}{\vdash e* \textsf{ matches } s \textsf{ leaving } \{s\} \cup S' \cup S''}\ .$$

Here $L_s(S)$ denotes the longest[1] string in S that is not s itself; $s - s'$ denotes the prefix of s before the suffix s', which is also regarded as the corresponding regular expression that only matches this literal string; ! and & are *complement* and *intersection* of regular expressions (which are valid as regular expressions are closed under taking complement and union and thus intersection); and $N(e)$ denotes "normalizing" regular expression e to remove ! and & used inside (which is also valid as | and Kleene star $*$ are complete for regular expressions). Note that the third rule applies when $S \neq \varnothing, \{s\}$, so $s' = L_s(S)$ always exists and is shorter than s, assuring that the third rule always makes progress when recursing on the match for S'. To clarify, the main idea here is that $e*$ is equivalent to $\textsf{empty} \mid ((s - s')\ e*) \mid ((e \mathbin{\&} !(\textsf{empty}) \mathbin{\&} !(s - s'))\ e*)$ where $s - s'$ is some match of e against s.[2] This decomposition helps handle such matching $s - s'$ one by one, which is somehow forced by the limitation in the given framework that only a finite and fixed set of hypothesis is allowed in each inference rule.

For concatenation $e_1\ e_2$, the main idea of the rules is to reduce concatenation to other

[1] It can actually be arbitrary choice and need not be the longest.

[2] I am not completely sure that the third rule always makes progress when recursing on the match for S'', but intuitively by taking $\& \,!\,(s - s')$, the $e \mathbin{\&} !(\textsf{empty}) \mathbin{\&} !(s - s')$ part matches for at most $S \setminus \{s'\}$, whose size is strictly smaller than S, and finally the recursion should reach the $S = \varnothing$ base case, if not ending at other rules for other grammars.

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope

2 3F-2 Regular Expressions, Large Step

- 0 pts Correct

Exercise 2. [10 points]. The large-step rules are as follows. For concatenation:

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

for or:

$$\frac{\vdash e_1 \text{ matches } s \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } s'}\ , \qquad \frac{\vdash e_2 \text{ matches } s \text{ leaving } s'}{\vdash e_1 \mid e_2 \text{ matches } s \text{ leaving } s'}\ ;$$

for Kleene star:

$$\frac{\vdash \text{empty} \mid (e\ e*) \text{ matches } s \text{ leaving } s'}{\vdash e* \text{ matches } s \text{ leaving } s'}\ .$$

Exercise 3. [5 points]. The deterministic large-step rules are as follows. For Kleene star:

$$\frac{\vdash e \text{ matches } s \text{ leaving } S \qquad S = \varnothing, \{s\}}{\vdash e* \text{ matches } s \text{ leaving } \{s\}}\ ,$$

$$\frac{\begin{cases} \vdash e \text{ matches } s \text{ leaving } S \qquad S \neq \varnothing, \{s\} \qquad s' = L_s(S) \\ \vdash e* \text{ matches } s' \text{ leaving } S' \\ \vdash N(e \,\&\, !(\text{empty}) \,\&\, !(s - s'))\ e* \text{ matches } s \text{ leaving } S'' \end{cases}}{\vdash e* \text{ matches } s \text{ leaving } \{s\} \cup S' \cup S''}\ .$$

Here $L_s(S)$ denotes the longest[1] string in S that is not s itself; $s - s'$ denotes the prefix of s before the suffix s', which is also regarded as the corresponding regular expression that only matches this literal string; ! and & are *complement* and *intersection* of regular expressions (which are valid as regular expressions are closed under taking complement and union and thus intersection); and $N(e)$ denotes "normalizing" regular expression e to remove ! and & used inside (which is also valid as | and Kleene star $*$ are complete for regular expressions). Note that the third rule applies when $S \neq \varnothing, \{s\}$, so $s' = L_s(S)$ always exists and is shorter than s, assuring that the third rule always makes progress when recursing on the match for S'. To clarify, the main idea here is that $e*$ is equivalent to empty $\mid ((s - s')\ e*) \mid ((e \,\&\, !(\text{empty}) \,\&\, !(s - s'))\ e*)$ where $s - s'$ is some match of e against s.[2] This decomposition helps handle such matching $s - s'$ one by one, which is somehow forced by the limitation in the given framework that only a finite and fixed set of hypothesis is allowed in each inference rule.

For concatenation $e_1\ e_2$, the main idea of the rules is to reduce concatenation to other

[1] It can actually be arbitrary choice and need not be the longest.

[2] I am not completely sure that the third rule always makes progress when recursing on the match for S'', but intuitively by taking $\&\,!\,(s - s')$, the $e \,\&\, !(\text{empty}) \,\&\, !(s - s')$ part matches for at most $S \setminus \{s'\}$, whose size is strictly smaller than S, and finally the recursion should reach the $S = \varnothing$ base case, if not ending at other rules for other grammars.

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope

grammars based on the structure of e_1:

$$\frac{\vdash \text{"x" matches } s \text{ leaving } \varnothing}{\vdash \text{"x" } e_2 \text{ matches } s \text{ leaving } \varnothing}, \qquad \frac{\vdash \text{"x" matches } s \text{ leaving } \{s'\} \qquad \vdash e_2 \text{ matches } s' \text{ leaving } S}{\vdash \text{"x" } e_2 \text{ matches } s \text{ leaving } S},$$

$$\frac{\vdash e_2 \text{ matches } s \text{ leaving } S}{\vdash \text{empty } e_2 \text{ matches } s \text{ leaving } S},$$

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

and for the remaining case $e*\ e_2$, the rules are similar to those for Kleene star:[3]

$$\frac{\vdash e \text{ matches } s \text{ leaving } S \qquad S = \varnothing, \{s\} \qquad \vdash e_2 \text{ matches } s \text{ leaving } T}{\vdash e*\ e_2 \text{ matches } s \text{ leaving } T},$$

$$\frac{\begin{cases} \vdash e \text{ matches } s \text{ leaving } S \qquad S \neq \varnothing, \{s\} \qquad s' = L_s(S) \\ \vdash e_2 \text{ matches } s \text{ leaving } T \\ \vdash e*\ e_2 \text{ matches } s' \text{ leaving } S' \\ \vdash N(e \,\&\, !(\text{empty}) \,\&\, !(s - s'))\ e*\ e_2 \text{ matches } s \text{ leaving } S'' \end{cases}}{\vdash e*\ e_2 \text{ matches } s \text{ leaving } T \cup S' \cup S''}.$$

Again similarly the main idea in the case $e*\ e_2$ is that $e*\ e_2$ is equivalent to $e_2 \mid ((s-s')\ e*\ e_2) \mid ((e \,\&\, !(\text{empty}) \,\&\, !(s-s'))\ e*\ e_2)$ where $s - s'$ is some match of e against s.

Exercise 4. [7 points]. The equivalence of regular expressions is decidable, using the procedure below.

First of all note that it is valid to take the *complement* $!e$ of a regular expression e (e.g. by flipping the accept/reject states of the corresponding *deterministic finite-state automaton* (DFA), as regular expressions are equivalent to DFAs) and thus also the *intersection* $e_1 \,\&\, e_2$ of regular expressions (e.g. by $!(!e_1 \mid !e_2)$). Also observe that $e_1 \sim e_2$ if and only if $e_1 \& !e_2$ and $!e_1 \& e_2$ are both regular expressions matching nothing. Therefore to determine equivalence of regular expressions it suffices to determine *emptiness* of regular expressions. Again thanks to the fact that regular expressions are equivalent to DFAs, the emptiness of a regular expression can be determined by testing the emptiness of its corresponding DFA, which is easily solved by testing the reachability from the initial state to all accept states. Putting it all together, the procedure for determining whether $e_1 \sim e_2$ is as follows:

1. construct the corresponding DFAs a_1 and a_2 of regular expressions $e_1 \,\&\, !e_2$ and $!e_1 \,\&\, e_2$;
2. for each accept state s of a_1,
   if it is reachable from the initial state s_0 of a_1, then return false;
3. repeat the same for a_2: for each accept state s of a_2,
   if it is reachable from the initial state s_0 of a_2, then return false;
4. return true.

[3] One can actually regard the rules for Kleene star as a special case of the rules for $e*\ e_2$ with $e_2 = \text{empty}$ (and hence $T = \{s\}$), which checks with intuition.

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope

3 3F-3 Regular Expressions and Sets

- 0 pts Correct

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope

gradescope

grammars based on the structure of e_1:

$$\frac{\vdash \text{"x" matches } s \text{ leaving } \varnothing}{\vdash \text{"x" } e_2 \text{ matches } s \text{ leaving } \varnothing}, \qquad \frac{\vdash \text{"x" matches } s \text{ leaving } \{s'\} \qquad \vdash e_2 \text{ matches } s' \text{ leaving } S}{\vdash \text{"x" } e_2 \text{ matches } s \text{ leaving } S},$$

$$\frac{\vdash e_2 \text{ matches } s \text{ leaving } S}{\vdash \text{empty } e_2 \text{ matches } s \text{ leaving } S},$$

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

and for the remaining case $e*\ e_2$, the rules are similar to those for Kleene star:[3]

$$\frac{\vdash e \text{ matches } s \text{ leaving } S \qquad S = \varnothing, \{s\} \qquad \vdash e_2 \text{ matches } s \text{ leaving } T}{\vdash e*\ e_2 \text{ matches } s \text{ leaving } T},$$

$$\frac{\begin{cases} \vdash e \text{ matches } s \text{ leaving } S \qquad S \neq \varnothing, \{s\} \qquad s' = L_s(S) \\ \vdash e_2 \text{ matches } s \text{ leaving } T \\ \vdash e*\ e_2 \text{ matches } s' \text{ leaving } S' \\ \vdash N(e \,\&\, !(\text{empty}) \,\&\, !(s - s'))\ e*\ e_2 \text{ matches } s \text{ leaving } S'' \end{cases}}{\vdash e*\ e_2 \text{ matches } s \text{ leaving } T \cup S' \cup S''}.$$

Again similarly the main idea in the case $e*\ e_2$ is that $e*\ e_2$ is equivalent to $e_2 \mid ((s-s')\ e*\ e_2) \mid ((e \,\&\, !(\text{empty}) \,\&\, !(s - s'))\ e*\ e_2)$ where $s - s'$ is some match of e against s.

Exercise 4. [7 points]. The equivalence of regular expressions is decidable, using the procedure below.

First of all note that it is valid to take the *complement* $!e$ of a regular expression e (e.g. by flipping the accept/reject states of the corresponding *deterministic finite-state automaton* (DFA), as regular expressions are equivalent to DFAs) and thus also the *intersection* $e_1 \,\&\, e_2$ of regular expressions (e.g. by $!(!e_1 \mid !e_2)$). Also observe that $e_1 \sim e_2$ if and only if $e_1 \& !e_2$ and $!e_1 \& e_2$ are both regular expressions matching nothing. Therefore to determine equivalence of regular expressions it suffices to determine *emptiness* of regular expressions. Again thanks to the fact that regular expressions are equivalent to DFAs, the emptiness of a regular expression can be determined by testing the emptiness of its corresponding DFA, which is easily solved by testing the reachability from the initial state to all accept states. Putting it all together, the procedure for determining whether $e_1 \sim e_2$ is as follows:

1. construct the corresponding DFAs a_1 and a_2 of regular expressions $e_1 \,\&\, !e_2$ and $!e_1 \,\&\, e_2$;
2. for each accept state s of a_1,
   if it is reachable from the initial state s_0 of a_1, then return false;
3. repeat the same for a_2: for each accept state s of a_2,
   if it is reachable from the initial state s_0 of a_2, then return false;
4. return true.

[3] One can actually regard the rules for Kleene star as a special case of the rules for $e*\ e_2$ with $e_2 = \text{empty}$ (and hence $T = \{s\}$), which checks with intuition.

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope

4 3F-4 Equivalence

- 0 pts Correct

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope

gradescope

Exercise 5. [6 points]. By injecting print commands (with immediate flush) at proper places, it can be observed that the (only) time-consuming module is `Arith`. There is hardly any iteration between the DPLL solver and the theory solver for the last two included tests, and most of the running time is consumed in a single call to the theory solver in `Arith`. The poor performance is probably due to the fact that the implementation of `Arith` checks satisfiability of formulae by brute-forcing all integers within some bounded range. As an evidence, the simple satisfiable formula `x = 0 && y = 0 && z = 0` also takes roughly as long a time as the last two tests. Therefore, it is natural to choose to rewrite the `Arith` module first.

One simple heuristic for improving the performance of `Arith` is to change the order in which the integers are brute-forced. The order is somehow crucial as e.g. if the example formula is modified to be `x = -127 && y = -127 && z = -127` (where -127 is the first value brute-forced in `Arith`), then the program indeed terminates instantly. Following an intuition that a "natural" satisfiable formula tends to have solutions consisting of "small" integers, one could consider an alternative order $0, \pm 1, \pm 2, \ldots$ for brute-forcing the integers, which will yield "small" solutions faster. To remark this order is just intuitive and cannot help with the worst-case performance. Furthermore one could also consider using some convex/nonconvex optimization technique as a heuristic in the search for solutions, but anyway this integer (even nonlinear!) programming problem is likely to be hard in general.

As is already mentioned, the `Arith` module checks satisfiability by brute-forcing all integers within some bounded range (in particular $[-127, 128]$), which is incomplete and is definitely a defect. For instance, it turns out that `x = -128 && y = -128 && z = -128` is considered unsatisfiable by `Arith`, which is obviously not the case.

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope

5 3F-5 SAT Solving

- 0 pts Correct

Peer Review ID: 67538562 — enter this when you fill out your peer evaluation via gradescope