1 3F-1 Bookkeeping

- **0 pts** Correct

**Answer 3F-1:**

The basic idea here is to apply the command rules without the guarding boolean expression $b$ and instead non-deterministically allow both evaluation paths:

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

$$\textsc{or-1}\ \frac{e_1 \text{ matches } s \text{ leaving } s'}{\vdash e_1|e_2 \text{ matches } s \text{ leaving } s'} \qquad \textsc{or-2}\ \frac{e_2 \text{ matches } s \text{ leaving } s'}{\vdash e_1|e_2 \text{ matches } s \text{ leaving } s'}$$

$$\textsc{star-0}\ \frac{}{\vdash e* \text{ matches } s \text{ leaving } s} \qquad \textsc{star}\ \frac{\vdash e \text{ matches } s \text{ leaving } s' \qquad \vdash e* \text{ matches } s' \text{ leaving } s''}{\vdash e* \text{ matches } s \text{ leaving } s''}$$

**Exercise 3F-2. 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:

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

And use rules of inference like the following:

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

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

You must do one of the following:

- *either* give operational semantics rules of inference for $e*$ and $e_1 e_2$. You may *not* place a derivation inside a set constructor, as in: $\{x \mid \exists y.\ \vdash e \text{ matches } x \text{ leaving } y\}$. 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".

## 2 3F-2 Regular Expressions, Large Step

- **0 pts** Correct

**Answer 3F-2:**

This is not possible to do. Within this framework, we don't have a way of expressing a well-founded inductive rule for the Kleen-star that captures the idea of "0 or more" times. We could try something trivial like:

$$\text{STAR-SIMPLE}\ \frac{\vdash ((\mathsf{empty}|e)|(ee|eee))\ \mathsf{matches}\ s\ \mathsf{leaving}\ S}{\vdash e*\ \mathsf{matches}\ s\ \mathsf{leaving}\ S}$$

However, this is incomplete in that it only expresses the idea of applying it 0, 1, 2, or 3 times instead of an arbitrary $n$ times. Extending this chain infinitely is not allowed by the instructions. We could attempt to use the usual inductive definition like:

$$\text{STAR-IND}\ \frac{\vdash e\ \mathsf{matches}\ s\ \mathsf{leaving}\ S \qquad s' \in S \qquad \vdash e*\ \mathsf{matches}\ s'\ \mathsf{leaving}\ S'}{\vdash e*\ \mathsf{matches}\ s\ \mathsf{leaving}\ \{s\} \cup S' \cup S}$$

However, this doesn't quite work because 1) unsure if we can use set operators in the hypothesis, and 2) we need a forall quantifier. Since we can't move this to the set constructor, the rule is incomplete.

**Exercise 3F-3. 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.

**Answer 3F-3** :

This is undecidable.

*Proof.* By contradiction. Assume the equivalence of $e$ is decidable, i.e. we have a deterministic algorithm that can answer if $e \sim e$. Let $c \in Comm$ be an Imp command. As given, the equivalence of $c$ can be defined as $c \sim c \Leftrightarrow halt(c)$ and is known to be undecidable. Now consider the following reduction from $c$ to $e$:

First, assume an *encode* function that serializes a give $\sigma$ to a string under an invertible scheme. Let *decode* be the inverse of *encode*. Assume the state $\sigma$ is reduced to the string domain by these functions. By the operational semantics of $Comm$, we know $c$ must be one of the following and each can be reduced to an $e$:

## 3 3F-3 Regular Expressions and Sets

- **0 pts** Correct

**Answer 3F-2:**

This is not possible to do. Within this framework, we don't have a way of expressing a well-founded inductive rule for the Kleen-star that captures the idea of "0 or more" times. We could try something trivial like:

$$\text{STAR-SIMPLE}\ \frac{\vdash ((\mathsf{empty}|e)|(ee|eee))\ \mathsf{matches}\ s\ \mathsf{leaving}\ S}{\vdash e*\ \mathsf{matches}\ s\ \mathsf{leaving}\ S}$$

However, this is incomplete in that it only expresses the idea of applying it 0, 1, 2, or 3 times instead of an arbitrary $n$ times. Extending this chain infinitely is not allowed by the instructions. We could attempt to use the usual inductive definition like:

$$\text{STAR-IND}\ \frac{\vdash e\ \mathsf{matches}\ s\ \mathsf{leaving}\ S \qquad s' \in S \qquad \vdash e*\ \mathsf{matches}\ s'\ \mathsf{leaving}\ S'}{\vdash e*\ \mathsf{matches}\ s\ \mathsf{leaving}\ \{s\} \cup S' \cup S}$$

However, this doesn't quite work because 1) unsure if we can use set operators in the hypothesis, and 2) we need a forall quantifier. Since we can't move this to the set constructor, the rule is incomplete.

**Exercise 3F-3. 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.

**Answer 3F-3** :

This is undecidable.

*Proof.* By contradiction. Assume the equivalence of $e$ is decidable, i.e. we have a deterministic algorithm that can answer if $e \sim e$. Let $c \in Comm$ be an Imp command. As given, the equivalence of $c$ can be defined as $c \sim c \Leftrightarrow halt(c)$ and is known to be undecidable. Now consider the following reduction from $c$ to $e$:

First, assume an *encode* function that serializes a give $\sigma$ to a string under an invertible scheme. Let *decode* be the inverse of *encode*. Assume the state $\sigma$ is reduced to the string domain by these functions. By the operational semantics of $Comm$, we know $c$ must be one of the following and each can be reduced to an $e$:

1. $c$ is *skip*. Reduce $c$ to *empty*
2. $c$ is a *seq*. Reduce $c$ to *concat*
3. $c$ is an $if$. Reduce $c$ to *or*
4. $c$ is an *assign*. Perform a deterministic and decidable string replacement
5. $c$ is a *while*. Reduce $c$ to *kleene star*

The rules apply inductively on $c$. For all *bexp* and *aexp*, we evaluate using the operational semantics of Imp and update the state as necessary. The resulting set $S$ from executing $e$ on an arbitrary $\sigma$ encoded as a string via $encode(\sigma)$ holds all possible final strings, each equivalent to a state via *decode*. Then it holds that $\exists s \in S.\langle c, \sigma\rangle \downarrow \sigma' \wedge decode(s) = \sigma'$. If we know the final state for $c$ then we know if $halt(c)$ holds, and so the equivalence of $c$ is decidable. This gives us the contradiction. □

**Exercise 3C. SAT Solving.** Download the Homework 3 code pack from the course web page. Update the skeletal SMT solver so that it correctly integrates the given DPLL-style CNF SAT solver with the given theory of bounded arithmetic. In particular, you must update only the Main.solve function. Your updated solver must be correct. This notably implies that it must correctly handle all of the included test cases — we use `diff` for some testing, but if you change only the listed method you should end up with the same answers as the reference.

In addition, create an example "tricky" input that can be parsed by our test harness.

Submit your `.ml` and `.input` files.

**Exercise 3F-4. 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.

**Answer 3F-4:**

Test input 35 and 36 are very similar in structure: they are a chain of arithmetic expressions chained with the logical *and* operator. When DPLL encounters this, it will assign meta variables to each arithmetic clause and search for a solution. Since these will all be unit clauses, DPLL will conclude that they must all be satisfiable. DPLL then queries the theory solver on whether or not all clauses can be satisfied at the same time. The theory solver here is an arithmetic solver that uses bruteforce search in a bounded space. Specifically, the implementation will exhaustively search the space of 3 variables with values between -127 and 128 and check if any assignment can satisfy the constraints. The expressions the variables

## 4 3F-4 Equivalence

- **0 pts** Correct

1. $c$ is *skip*. Reduce $c$ to *empty*
2. $c$ is a *seq*. Reduce $c$ to *concat*
3. $c$ is an $if$. Reduce $c$ to *or*
4. $c$ is an *assign*. Perform a deterministic and decidable string replacement
5. $c$ is a *while*. Reduce $c$ to *kleene star*

The rules apply inductively on $c$. For all *bexp* and *aexp*, we evaluate using the operational semantics of Imp and update the state as necessary. The resulting set $S$ from executing $e$ on an arbitrary $\sigma$ encoded as a string via $encode(\sigma)$ holds all possible final strings, each equivalent to a state via *decode*. Then it holds that $\exists s \in S.\langle c, \sigma\rangle \downarrow \sigma' \wedge decode(s) = \sigma'$. If we know the final state for $c$ then we know if $halt(c)$ holds, and so the equivalence of $c$ is decidable. This gives us the contradiction. □

**Exercise 3C. SAT Solving.** Download the Homework 3 code pack from the course web page. Update the skeletal SMT solver so that it correctly integrates the given DPLL-style CNF SAT solver with the given theory of bounded arithmetic. In particular, you must update only the Main.solve function. Your updated solver must be correct. This notably implies that it must correctly handle all of the included test cases — we use `diff` for some testing, but if you change only the listed method you should end up with the same answers as the reference.

In addition, create an example "tricky" input that can be parsed by our test harness.

Submit your `.ml` and `.input` files.

**Exercise 3F-4. 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.

**Answer 3F-4:**

Test input 35 and 36 are very similar in structure: they are a chain of arithmetic expressions chained with the logical *and* operator. When DPLL encounters this, it will assign meta variables to each arithmetic clause and search for a solution. Since these will all be unit clauses, DPLL will conclude that they must all be satisfiable. DPLL then queries the theory solver on whether or not all clauses can be satisfied at the same time. The theory solver here is an arithmetic solver that uses bruteforce search in a bounded space. Specifically, the implementation will exhaustively search the space of 3 variables with values between -127 and 128 and check if any assignment can satisfy the constraints. The expressions the variables

are associated with are $>$, $<$, and $=$. However, the theory doesn't include a theory of equality or order so it must simply exhaust the search space. To improve the performance, I would rewrite the arithmetic solver module to take into account the constraints of already assigned variables and the relationship between all variables, and follow the rules of arithmetic to properly propagated these constraints similar to the theory of equality. For example, in the case of test 35 and 36, we could narrow down the search space to 10, 11, and 12.

Extra: It only works on a small subset of integers. This could be improved if the solution wasn't a simple bruteforce search of a bounded space.

**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.

5 3F-5 SAT Solving

- **0 pts** Correct