1 2F-1 Bookkeeping

- 0 pts Correct

Peer Review ID: 65774275 — enter this when you fill out your peer evaluation via gradescope

Exercise 2F-2. Mathematical Induction [5 points]. Find the flaw in the following inductive proof that "All flowers smell the same". Please indicate exactly which sentences are wrong in the proof via highlighting or underlining.

Proof: Let F be the set of all flowers and let $\mathsf{smells}(f)$ be the smell of the flower $f \in F$. (The range of smells is not so important, but we'll assume that it admits equality.) We'll also assume that F is countable. Let the property $P(n)$ mean that all subsets of F of size at most n contain flowers that smell the same.

$$P(n) \overset{\text{def}}{=} \forall X \in \mathcal{P}(F).\ |X| \le n \implies (\forall f, f' \in X.\ \mathsf{smells}(f) = \mathsf{smells}(f'))$$

(the notation $|X|$ denotes the number of elements of X)

One way to formulate the statement to prove is $\forall n \geq 1.P(n)$. We'll prove this by induction on n, as follows:

Base Case: $n = 1$. Obviously all singleton sets of flowers contain flowers that smell the same (by the definition of $P(n)$).

Induction Step: Let n be arbitrary and assume that all subsets of F of size at most n contain flowers that smell the same. We will prove that the same thing holds for all subsets of size at most $n+1$. Pick an arbitrary set X such that $|X| = n+1$. Pick two distinct flowers $f, f' \in X$ and let's show that $\mathsf{smells}(f) = \mathsf{smells}(f')$. Let $Y = X - \{f\}$ and $Y' = X - \{f'\}$. Obviously Y and Y' are sets of size at most n so the induction hypothesis holds for both of them. Pick any arbitrary $x \in Y \cap Y'$. Obviously, $x \neq f$ and $x \neq f'$. We have that $\mathsf{smells}(f') = \mathsf{smells}(x)$ (from the induction hypothesis on Y) and $\mathsf{smells}(f) = \mathsf{smells}(x)$ (from the induction hypothesis on Y'). Hence $\mathsf{smells}(f) = \mathsf{smells}(f')$, which proves the inductive step, and the theorem.

(One indication that the proof might be wrong is the large number of occurrences of the word "obviously" `:-)`)

Solution:

The sentence "Pick any arbitrary $x \in Y \cap Y'$" is wrong because in the case where $n = 2$, $Y \cap Y' = \emptyset$. That is, there are no elements in $Y \cap Y'$. This is because one element is removed to make Y, the other element is removed to Y', and there are no elements left that could form the intersection $Y \cap Y'$. The smells function cannot be applied to a non-existent element, so the inductive step for $P = 2$ is not logically valid.

Exercise 2F-3. While Induction [10 points]. Prove by induction the following statement about the operational semantics:

For any BExp b and any initial state σ such that $\sigma(x)$ is even, if

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma \rangle \Downarrow \sigma'$$

then $\sigma'(x)$ is even. Make sure you state what you induct on, what the base case is and what the inductive cases are. Show representative cases among the latter. Do not do a proof by mathematical induction!

2 2F-2 Mathematical Induction

- 0 pts Correct

Exercise 2F-2. Mathematical Induction [5 points]. Find the flaw in the following inductive proof that "All flowers smell the same". Please indicate exactly which sentences are wrong in the proof via highlighting or underlining.

Proof: Let F be the set of all flowers and let $\mathsf{smells}(f)$ be the smell of the flower $f \in F$. (The range of smells is not so important, but we'll assume that it admits equality.) We'll also assume that F is countable. Let the property $P(n)$ mean that all subsets of F of size at most n contain flowers that smell the same.

$$P(n) \overset{\text{def}}{=} \forall X \in \mathcal{P}(F).\ |X| \leq n \implies (\forall f, f' \in X.\ \mathsf{smells}(f) = \mathsf{smells}(f'))$$

(the notation $|X|$ denotes the number of elements of X)

One way to formulate the statement to prove is $\forall n \geq 1.P(n)$. We'll prove this by induction on n, as follows:

Base Case: $n = 1$. Obviously all singleton sets of flowers contain flowers that smell the same (by the definition of $P(n)$).

Induction Step: Let n be arbitrary and assume that all subsets of F of size at most n contain flowers that smell the same. We will prove that the same thing holds for all subsets of size at most $n+1$. Pick an arbitrary set X such that $|X| = n+1$. Pick two distinct flowers $f, f' \in X$ and let's show that $\mathsf{smells}(f) = \mathsf{smells}(f')$. Let $Y = X - \{f\}$ and $Y' = X - \{f'\}$. Obviously Y and Y' are sets of size at most n so the induction hypothesis holds for both of them. Pick any arbitrary $x \in Y \cap Y'$. Obviously, $x \neq f$ and $x \neq f'$. We have that $\mathsf{smells}(f') = \mathsf{smells}(x)$ (from the induction hypothesis on Y) and $\mathsf{smells}(f) = \mathsf{smells}(x)$ (from the induction hypothesis on Y'). Hence $\mathsf{smells}(f) = \mathsf{smells}(f')$, which proves the inductive step, and the theorem.

(One indication that the proof might be wrong is the large number of occurrences of the word "obviously" :-))

Solution:

The sentence "Pick any arbitrary $x \in Y \cap Y'$" is wrong because in the case where $n = 2$, $Y \cap Y' = \emptyset$. That is, there are no elements in $Y \cap Y'$. This is because one element is removed to make Y, the other element is removed to Y', and there are no elements left that could form the intersection $Y \cap Y'$. The smells function cannot be applied to a non-existent element, so the inductive step for $P = 2$ is not logically valid.

Exercise 2F-3. While Induction [10 points]. Prove by induction the following statement about the operational semantics:

For any BExp b and any initial state σ such that $\sigma(x)$ is even, if

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$$

then $\sigma'(x)$ is even. Make sure you state what you induct on, what the base case is and what the inductive cases are. Show representative cases among the latter. Do not do a proof by mathematical induction!

Solution:

Before proceeding with the proof, we introduce the following lemma:

Lemma: *If $n \in \mathbb{Z}$ and n is even, then $n+2$ is also even.*

Proof. Let $n \in \mathbb{Z}$ be an even integer. By definition of an even integer, there must exist some $k \in \mathbb{Z}$ such that

$$n = 2k.$$

Adding 2 to n, we have

$$n + 2 = 2k + 2 = 2(k+1).$$

By definition of an even integer, this means that $n+2$ is also even (since $n+2$ is equal to $2q$ where $q = k+1$). □

We now continue with the original proof. We wish to prove that for any Bexp b and any initial state σ such that $\sigma(x)$ is even, if

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x+2, \sigma\rangle \Downarrow \sigma'$$

then $\sigma'(x)$ is even. The proof proceeds by induction on the structure of the derivation of this `while` command. By inversion, there are only two rules that could have been applied to this `while` command: the rule for `while true` or the rule for `while false`. In the base case, the last rule that was used was the rule for `while false`. Specifically, this means the overall derivation D of the `while` command is of the form

$$\frac{D_1 :: \langle b, \sigma\rangle \Downarrow \texttt{false}}{\langle \texttt{while}\ b\ \texttt{do}\ x := x+2, \sigma\rangle \Downarrow \sigma}$$

where D_1 is the derivation of the evaluation of b in state σ. In this case, the initial state σ and the final state σ' are both σ, so $\sigma = \sigma'$. This implies that $\sigma(x) = \sigma'(x)$, so if x was even in the state before the `while` command, it must still be even in the state after the `while` command.

In the inductive step, we assume that for all sub-derivations of commands of the form $\langle \texttt{while}\ b\ \texttt{do}\ x := x+2, \sigma\rangle \Downarrow \sigma'$, if $\sigma(x)$ is even, then $\sigma'(x)$ is also even. There is only one inductive case to consider: the last rule that was used was the one for `while true` (as we already used the rule for `while false` in the base case, and there are no other rules for `while`). Specifically, this means the overall derivation D of this `while` command is of the form

$$\frac{D_1 :: \langle b, \sigma\rangle \Downarrow \texttt{true} \quad D_2 :: \langle x := x+2, \sigma\rangle \Downarrow \sigma_1 \quad D_3 :: \langle \texttt{while b do}\ x := x+2, \sigma_1\rangle \Downarrow \sigma'}{\langle \texttt{while}\ b\ \texttt{do}\ x := x+2, \sigma\rangle \Downarrow \sigma'}$$

where D_1 is the derivation of the evaluation of b in state σ to produce `true`, D_2 is the derivation of the evaluation of $x := x + 2$ in state σ' to produce the state σ', and D_3 is the derivation of the evaluation of `while b do` $x := x + 2$ in state σ_1 to produce the state σ'. By definition of the rules for arithmetic expression evaluation and assignment, we know that $\sigma_1(x) = \sigma(x) + 2$. By the lemma proved in the beginning of this problem, we know that $\sigma_1(x)$ must be even. Since $\sigma_1(x)$ is even, we apply the inductive hypothesis on D_3 to derive that $\sigma'(x)$ is also even. Thus, in the `while true` case, if $\sigma(x)$ is even, then $\sigma'(x)$ is also even.

We have shown that in all possible derivations of the command $\langle$`while` b `do` $x := x + 2, \sigma\rangle \Downarrow \sigma'$, if $\sigma(x)$ is even, then $\sigma'(x)$ is also even. This concludes the proof. □

Exercise 2F-4. Language Features, Large-Step [12 points]. We extend IMP with a notion of integer-valued *exceptions* (or *run-time errors*), as in Java, ML or C#. We introduce a new type T to represent command terminations, which can either be normal or exceptional (with an exception value $n \in \mathbb{Z}$):

$$\begin{array}{rcll} T & ::= & \sigma & \text{"normal termination"} \\ & | & \sigma \ \texttt{exc}\ n & \text{"exceptional termination"} \end{array}$$

We use t to range over possible terminations T. We then redefine our operational semantics judgment:

$$\langle c, \sigma \rangle \Downarrow T$$

The interpretation of

$$\langle c, \sigma \rangle \Downarrow \sigma' \ \texttt{exc}\ n$$

is that command c terminated abruptly by throwing an exception with value $n \in \mathbb{Z}$ at a point in c's execution when the state was σ'. We only model one type of exception, but every exception has an integer "argument" n (or "payload" or "value") that is set when the exception is thrown and available when the exception is caught.

Note that our previous command rules must be updated to account for exceptions, as in:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \ \texttt{exc}\ n}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma' \ \texttt{exc}\ n}\ \textsf{seq1} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle c_1; c_2, \sigma\rangle \Downarrow t}\ \textsf{seq2}$$

We also introduce three additional commands:

$$\begin{array}{l} \textsf{throw}\ e \\ \textsf{try}\ c_1\ \textsf{catch}\ x\ c_2 \\ \textsf{after}\ c_1\ \textsf{finally}\ c_2 \end{array}$$

- The throw e command raises an exception with argument e.
- The try command executes c_1. If c_1 terminates normally (i.e., without an uncaught exception), the try command also terminates normally. If c_1 raises an exception with value e, the variable $x \in L$ is assigned the value e and then c_2 is executed.

3 2F-3 While Induction

- 0 pts Correct

where D_1 is the derivation of the evaluation of b in state σ to produce `true`, D_2 is the derivation of the evaluation of $x := x + 2$ in state σ' to produce the state σ', and D_3 is the derivation of the evaluation of `while b do` $x := x + 2$ in state σ_1 to produce the state σ'. By definition of the rules for arithmetic expression evaluation and assignment, we know that $\sigma_1(x) = \sigma(x) + 2$. By the lemma proved in the beginning of this problem, we know that $\sigma_1(x)$ must be even. Since $\sigma_1(x)$ is even, we apply the inductive hypothesis on D_3 to derive that $\sigma'(x)$ is also even. Thus, in the `while true` case, if $\sigma(x)$ is even, then $\sigma'(x)$ is also even.

We have shown that in all possible derivations of the command $\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$, if $\sigma(x)$ is even, then $\sigma'(x)$ is also even. This concludes the proof. □

Exercise 2F-4. Language Features, Large-Step [12 points]. We extend IMP with a notion of integer-valued *exceptions* (or *run-time errors*), as in Java, ML or C#. We introduce a new type T to represent command terminations, which can either be normal or exceptional (with an exception value $n \in \mathbb{Z}$):

$$\begin{array}{rcll} T & ::= & \sigma & \text{“normal termination”} \\ & | & \sigma\ \texttt{exc}\ n & \text{“exceptional termination”} \end{array}$$

We use t to range over possible terminations T. We then redefine our operational semantics judgment:

$$\langle c, \sigma\rangle \Downarrow T$$

The interpretation of

$$\langle c, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n$$

is that command c terminated abruptly by throwing an exception with value $n \in \mathbb{Z}$ at a point in c's execution when the state was σ'. We only model one type of exception, but every exception has an integer “argument” n (or “payload” or “value”) that is set when the exception is thrown and available when the exception is caught.

Note that our previous command rules must be updated to account for exceptions, as in:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n}\ \textsf{seq1} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle c_1; c_2, \sigma\rangle \Downarrow t}\ \textsf{seq2}$$

We also introduce three additional commands:

$$\begin{array}{l} \textsf{throw}\ e \\ \textsf{try}\ c_1\ \textsf{catch}\ x\ c_2 \\ \textsf{after}\ c_1\ \textsf{finally}\ c_2 \end{array}$$

- The throw e command raises an exception with argument e.
- The try command executes c_1. If c_1 terminates normally (i.e., without an uncaught exception), the try command also terminates normally. If c_1 raises an exception with value e, the variable $x \in L$ is assigned the value e and then c_2 is executed.

- The finally command executes c_1. If c_1 terminates normally, the finally command terminates by executing c_2. If instead c_1 raises an exception with value e_1, then c_2 is executed:
  - If c_2 terminates normally, the finally command terminates by throwing an exception with value e_1. (That is, the original exception e_1 is re-thrown at the end of the finally block, as in Java.)
  - If c_2 throws an exception with value e_2, the finally command terminates by throwing an exception with value e_2. (That is, the new exception e_2 overrides the original exception e_1, also as in Java.)

These constructs are intended to have the standard exception semantics from languages like Java, C# or OCaml — except that the catch block merely assigns to x, it does not bind it to a local scope. So unlike Java, our catch does not behave like a let. We thus expect:

```
x := 0 ;
{ try
    if x <= 5 then throw 33 else throw 55
  catch x
    print x } ;
while true do {
  x := x - 15 ;
  print x ;
  if x <= 0 then throw (x*2) else skip
}
```

to output "`33 18 3 -12`" and then terminate with an uncaught exception with value `-24`.

Give the large-step operational semantics inference rules (using our new judgment) for the three new commands presented here. You should present six (6) new rules total.

Solution:

The 6 new rules are below:

$$\frac{\langle e, \sigma \rangle \Downarrow n}{\langle \mathsf{throw}\ e, \sigma \rangle \Downarrow \sigma\ \mathsf{exc}\ n}$$

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2, \sigma \rangle \Downarrow \sigma'}$$

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma'\ \mathsf{exc}\ n \quad \langle x := n, \sigma' \rangle \Downarrow \sigma'[x := n] \quad \langle c_2, \sigma'[x := n] \rangle \Downarrow t}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2, \sigma \rangle \Downarrow t}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle \textsf{after } c_1 \textsf{ finally } c_2, \sigma\rangle \Downarrow t}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \textsf{ exc } n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle \textsf{after } c_1 \textsf{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \textsf{ exc } n_1}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \textsf{ exc } n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma'' \textsf{ exc } n_2}{\langle \textsf{after } c_1 \textsf{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \textsf{ exc } n_2}$$

Exercise 2F-4. Language Features, Analysis [6 points]. Argue for or against the claim that it would be more natural to describe "IMP with exceptions" using small-step contextual semantics. You may use "simpler" or "more elegant" instead of "more natural" if you prefer. Do not exceed two paragraphs (one should be sufficient). Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter.

Solution:

It would **not** be more natural to describe "IMP with exceptions" using small-step semantics. The main reason for this is that the big-step semantics rules offer clearer descriptions of how try catch and after finally commands work. For example, in the case of after finally, we could write the following local reduction rules (assuming we define the redexes and contexts appropriately): $\langle \textsf{throw } n; c, \sigma\rangle \to \langle \textsf{throw } n, \sigma\rangle$, $\langle \textsf{after skip finally } c, \sigma\rangle \to \langle c, \sigma\rangle$, and $\langle \textsf{after throw } n \textsf{ finally } c\rangle \to \langle c; \textsf{throw } n, \sigma\rangle$. Here we define throw to be a terminating command like skip, but $\langle \textsf{throw } n, \sigma\rangle$ results in an exceptional termination whereas $\langle \textsf{skip}, \sigma\rangle$ results in a normal termination. The second rule in the list states that if we reached a normal termination in the first sub-command, then we simply execute the command c after the finally keyword. The third rule states that if we reached an exceptional termination in the first sub-command, then we execute c and afterward terminate with the same value as the first sub-command (as long as c evaluates normally; otherwise, c's termination will be the final termination). The meanings of these after finally rules may not be immediately clear from their definitions; one has to reason about how sequences of commands reduce down to skip or throw n to understand what these rules mean. In contrast, terminations of entire sub-commands for after finally are explicitly labeled in the rules, allowing one to more clearly understand what after finally commands truly do. For instance, in the big-step rule for after c_1 finally c_2 where both c_1 and c_2 produce exceptional terminations, one can see the evaluation of each *entire* sub-command in the premises and which termination is "selected" as the *final* termination. The ability to *see* these terminations directly in the rule itself permits a more natural interpretation that more accurately represents the English definition of after finally.

We can illustrate this conclusion more generally by also reasoning about try catch local reduction rules as well. We could define local reduction rules for try skip and try throw n

4 2F-4 Language Features, Large Step

- **0 pts** Correct

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow t}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow \sigma''\ \mathsf{exc}\ n_1}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \mathsf{exc}\ n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''\ \mathsf{exc}\ n_2}{\langle \mathsf{after}\ c_1\ \mathsf{finally}\ c_2, \sigma\rangle \Downarrow \sigma''\ \mathsf{exc}\ n_2}$$

Exercise 2F-4. Language Features, Analysis [6 points]. Argue for or against the claim that it would be more natural to describe "IMP with exceptions" using small-step contextual semantics. You may use "simpler" or "more elegant" instead of "more natural" if you prefer. Do not exceed two paragraphs (one should be sufficient). Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter.

Solution:

It would **not** be more natural to describe "IMP with exceptions" using small-step semantics. The main reason for this is that the big-step semantics rules offer clearer descriptions of how try catch and after finally commands work. For example, in the case of after finally, we could write the following local reduction rules (assuming we define the redexes and contexts appropriately): $\langle \mathsf{throw}\ n; c, \sigma\rangle \to \langle \mathsf{throw}\ n, \sigma\rangle$, $\langle \mathsf{after\ skip\ finally}\ c, \sigma\rangle \to \langle c, \sigma\rangle$, and $\langle \mathsf{after\ throw}\ n\ \mathsf{finally}\ c\rangle \to \langle c; \mathsf{throw}\ n, \sigma\rangle$. Here we define throw to be a terminating command like skip, but $\langle \mathsf{throw}\ n, \sigma\rangle$ results in an exceptional termination whereas $\langle \mathsf{skip}, \sigma\rangle$ results in a normal termination. The second rule in the list states that if we reached a normal termination in the first sub-command, then we simply execute the command c after the finally keyword. The third rule states that if we reached an exceptional termination in the first sub-command, then we execute c and afterward terminate with the same value as the first sub-command (as long as c evaluates normally; otherwise, c's termination will be the final termination). The meanings of these after finally rules may not be immediately clear from their definitions; one has to reason about how sequences of commands reduce down to skip or throw n to understand what these rules mean. In contrast, terminations of entire sub-commands for after finally are explicitly labeled in the rules, allowing one to more clearly understand what after finally commands truly do. For instance, in the big-step rule for after c_1 finally c_2 where both c_1 and c_2 produce exceptional terminations, one can see the evaluation of each *entire* sub-command in the premises and which termination is "selected" as the *final* termination. The ability to *see* these terminations directly in the rule itself permits a more natural interpretation that more accurately represents the English definition of after finally.

We can illustrate this conclusion more generally by also reasoning about try catch local reduction rules as well. We could define local reduction rules for try skip and try throw n

similar to the ones above for after skip and after throw n, but these rules would fail to illustrate the "bigger picture" of the core idea that the termination of try catch depends on the termination of the *entire* first sub-command. Again, one would need to reason about the fact that the first sub-command evaluates to either skip or throw n to understand what these local reduction rules mean. With big-step semantics, the evaluation of the entire first sub-command is built into the premises of the try catch rules. For this reason, big-step semantics more accurately represents the true meaning of such commands and is hence more natural to use for defining exceptions than small-step semantics.

Exercise 2C. Language Features, Coding. Download the Homework 2 code pack from the course web page. Modify `hw2.ml` so that it implements a complete interpreter for "IMP with exceptions (and `print`)". You may build on your code from Homework 1 (although the `let` command is not part of this assignment). Using OCaml's exception mechanism to implement IMP exceptions is actually slightly harder than doing it "naturally", so I recommend that you just implement the operational semantics rules. The `Makefile` includes a "`make test`" target that you should use (at least) to test your work.

Hint: to check if a termination `term` is an exception, use syntax like

```
begin match term with
| Normal -> do_something
| Exceptional(n) -> do_something_else using n
end
```

Modify the file `example-imp-command` so that it contains a "tricky" terminating IMP command (presumably involving exceptions) that can be parsed by our IMP test harness (e.g., "`imp < example-imp-command`" should not yield a parse error).

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 2F-5 Language Features, Analysis

- 0 pts Correct