1 2F-1 Bookkeeping

- 0 pts Correct

Peer Review ID: 65784694 — 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.

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

(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" `:-)`)

The highlighted sentences are incorrect because we haven't shown that x exists. If n+1=2, then Y and Y' are both of size 1 with f being in Y and f' being in Y' so the intersection of Y and Y' is empty which means no such x exists, violating the proof.

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!

We do this using induction on the derivation. Our base case is

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

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" `:-)`)

The highlighted sentences are incorrect because we haven't shown that x exists. If n+1=2, then Y and Y' are both of size 1 with f being in Y and f' being in Y' so the intersection of Y and Y' is empty which means no such x exists, violating the proof.

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!

We do this using induction on the derivation. Our base case is

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

In which case the state does not change so if $\sigma(x)$ was originally even, it remains even.

By inversion, these are the only two cases we have to consider, as it does not matter if b is an equality check, less than or equal to, conjuction, disjunction, etc. All that matters is the resulting evaluation of b to either True or False. If b is False then we have reached the base case and if B is True then we have the following induction step:

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

By induction hypothesis on D_3 we know that in state σ, x is given to be even. Adding 2 to an even number in D_2 means the resulting number is even, thus $\sigma'(x)$ is also even. This means that in every execution of the while loop, if x starts off as an even number, it will end as an even number. Similarly, if we reach the base case, and x is an even number, it will remain an even number. If b never evaluates to False and we don't reach the base case, then the program will continue to run and we will never reach state σ' in the original expression (the while loop runs forever). This means it won't make sense to evaluate if x is even or odd in σ' unless we reach the base call of b=False allowing the program to terminate. Thus, if x is an even number prior to running the while loop, it will remain an even number. $\square$

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:

$\textsf{throw}\ e$
$\textsf{try}\ c_1\ \textsf{catch}\ x\ c_2$
$\textsf{after}\ c_1\ \textsf{finally}\ c_2$

3 2F-3 While Induction

- 0 pts Correct

- 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.

The 6 rules needs to make IMP with exceptions work are the following:

$$\frac{\langle e, \sigma\rangle \Downarrow n}{\langle \text{throw e}, \sigma\rangle \Downarrow \sigma \ \texttt{exc}\ n}\ \textsf{throw}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \text{try } c_1 \text{ catch } x\ c_2, \sigma\rangle \Downarrow \sigma'}\ \textsf{try1} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \ \texttt{exc}\ n \quad \langle c_2, \sigma'[x := n]\rangle \Downarrow t}{\langle \text{try } c_1 \text{ catch } x\ c_2, \sigma\rangle \Downarrow t}\ \textsf{try2}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow t}\ \textsf{after1} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \ \texttt{exc}\ n \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \ \texttt{exc}\ n}\ \textsf{after2}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \ \texttt{exc}\ n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma'' \ \texttt{exc}\ n_2}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \ \texttt{exc}\ n_2}\ \textsf{after3}$$

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.

It would not be more natural to describe IMP with exceptions in small-step semantics. Small-step semantics are well suited for tasks that are very procedural as a sequence of operations such as while loops. With exceptions, we often have to handle the execution of the program based on various conditions. Take for example the after finally command. There are three distinct paths that we can take as seen in the large-step operations presented above. To do this in small-step semantics would require us to execute an operation such as c1, analyze the termination state of c1, if its an exception then run c2, analyze the termination state of c2, and then return an exception from either c1 or c2 or the state after c1 executed normally. These are easy to convey in large-step semantics since we can easily depict the various ways each of these commands return but in small-step semantics, we would need many reduction rules would to represent all of the different ways that each of the commands can return to allow for redexes to be replaced in a single step.

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).

Submitted to Autograder :)

4 2F-4 Language Features, Large Step

- 0 pts Correct

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \texttt{ exc } n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma'' \texttt{ exc } n_2}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \texttt{ exc } n_2} \text{after3}$$

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.

It would not be more natural to describe IMP with exceptions in small-step semantics. Small-step semantics are well suited for tasks that are very procedural as a sequence of operations such as while loops. With exceptions, we often have to handle the execution of the program based on various conditions. Take for example the after finally command. There are three distinct paths that we can take as seen in the large-step operations presented above. To do this in small-step semantics would require us to execute an operation such as c1, analyze the termination state of c1, if its an exception then run c2, analyze the termination state of c2, and then return an exception from either c1 or c2 or the state after c1 executed normally. These are easy to convey in large-step semantics since we can easily depict the various ways each of these commands return but in small-step semantics, we would need many reduction rules would to represent all of the different ways that each of the commands can return to allow for redexes to be replaced in a single step.

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).

Submitted to Autograder :)

5 2F-5 Language Features, Analysis

- **0 pts** Correct