Exercise 2F-2. Mathematical Induction [5 points]

The flaw in the inductive proof arises in the **induction step** when attempting to prove $P(n+1)$ from $P(n)$. Specifically:

The proof states: *"Pick any arbitrary $x \in Y \cap Y'$."* However, when $n = 1$, $Y = X \setminus \{f\}$ and $Y' = X \setminus \{f'\}$ (where $|X| = 2$), resulting in $Y \cap Y' = \emptyset$. **No such x exists** in this case. The inductive step **fails for** $n = 1$ because the intersection is empty, making it impossible to equate smells(f) and smells(f').

Since the inductive step does not hold for $n = 1$,the proof does not establish the statement for all n. The proof incorrectly assumes $Y \cap Y'$ is non-empty for all n, which is false when transitioning from $n = 1$ to $n + 1 = 2$.

Highlighted Sentences in the Proof

- *"Pick any arbitrary $x \in Y \cap Y'$. Obviously, $x \neq f$ and $x \neq f'$."* **Invalid when** $n = 1$: $Y \cap Y'$ is empty.
- *"Hence smells(f) = smells(f'), which proves the inductive step..."* **Relies on the existence of** x, which fails for $n = 1$.

Question assigned to the following page: 3

Exercise 2F-3. While Induction [10 points]

Statement

For any Boolean expression b and initial state σ where $\sigma(x)$ is even, if $\langle \text{while } b \text{ do } x := x + 2, \sigma \rangle \Downarrow \sigma'$, then $\sigma'(x)$ is even.

Base Case

Case: While-False

If $\langle b, \sigma \rangle \to \text{false}$: then by the operational semantics rule for *while-false* :

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

Since $\sigma(x)$ is even by assumption,

$$\sigma'(x) = \sigma(x), \text{ remains even.}$$

Inductive Step

Case: While-True

If $\langle b, \sigma \rangle \to \text{true}$:

$$\langle x := x + 2, \sigma \rangle \Downarrow \sigma''$$

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

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

To show $\sigma'(x)$ is even.

$$\langle x := x + 2, \sigma \rangle \Downarrow \sigma''.$$

By the assignment rule,

$$\sigma''(x) = \sigma(x) + 2.$$

Since $\sigma(x)$ is even, $\sigma''(x)$ is even (even + even = even).

Question assigned to the following page: 4

Exercise 2F-4. Language Features, Large-Step [12 points]

Here are the six large-step operational semantics rules for the new commands:

1. Throw Command

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

The `throw` command evaluates the expression e to a value n and immediately terminates execution, propagating n as an exception. The state σ remains unchanged, but the result is marked as an exception with value n.

2. Try Command (Normal Termination)

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

If the command c_1 terminates normally (without throwing an exception), the `try` command also terminates normally with the same final state σ'. The catch block c_2 is not executed.

3. Try Command (Exceptional Termination)

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

If the command c_1 throws an exception with value n, the exception value n is assigned to the variable x, and the catch block c_2 is executed. The result of the `try` command is the result of executing c_2, denoted by t.

4. Finally Command (Normal Termination of c_1)

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma_1 \quad \langle c_2, \sigma_1\rangle \Downarrow t}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow t}$$

If the command c_1 terminates normally, the `finally` block c_2 is executed. The result of the `after` command is the result of executing c_2, denoted by t.

5. Finally Command (Exceptional Termination of c_1, Normal Termination of c_2)

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma_1 \text{ exc } e_1 \quad \langle c_2, \sigma_1\rangle \Downarrow \sigma_2}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma_2 \text{ exc } e_1}$$

If the command c_1 throws an exception e_1, but the `finally` block c_2 terminates normally, the exception e_1 is re-thrown after executing c_2. The final state is σ_2, and the result is marked as an exception with value e_1.

Question assigned to the following page: 4

6. Finally Command (Exceptional Termination of Both c_1 and c_2)

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma_1 \text{ exc } e_1 \quad \langle c_2, \sigma_1\rangle \Downarrow \sigma_2 \text{ exc } e_2}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma_2 \text{ exc } e_2}$$

If both c_1 and c_2 throw exceptions e_1 and e_2, respectively, the exception e_2 (from c_2) is propagated. The final state is σ_2, and the result is marked as an exception with value e_2.

Question assigned to the following page: 5

Exercise 2F-5. Language Features, Analysis [6 points]

Large-step semantics summarizes execution by showing only the starting and final states, skipping the intermediate steps. For example, if a loop runs multiple times, large-step semantics would simply state that the loop executes and updates the state, without detailing each iteration. In contrast, small-step semantics breaks execution into smaller steps, showing how each part of a program runs incrementally. If a loop runs five times, small-step semantics would display each individual step rather than jumping straight to the end.

When it comes to exceptions, we know that they disrupt normal execution by immediately halting the program when an error occurs, such as a division by zero. To manage this, the program can either handle the exception using try-catch or perform necessary cleanup with a finally block. Since they interrupt a program at a specific step, it makes more sense to describe them using small-step semantics. This way, we can see when the exception happens, how the program responds, and what happens next in a step-by-step manner. Therefore, I believe that this method is easier and more elegant, since it doesn't need extra components like exception flags or special states, which large-step semantics often require to handle exceptions.