2 Mathematical Induction

The main flaw of this proof is that it has an assumption saying, for an arbitrary n, $Y \cap Y'$ must be a non-empty set. BIn other words, by saying "Pick any arbitrary $x \in Y \cap Y'$" it is assumed that there is always some flower x in both Y and Y'. However, if X has only two elements, i.e. $n = 1$, such that $X = \{f, f'\}$, then Y, Y' and $Y \cap Y'$ will be empty. Thus, there is no guaranteed element of x to serve as a common reference for transitivity of the smell equality. The induction step fails for $n = 1$, which prevents it from extending to higher n.

Also, in the base case, while the claim that all singleton sets contain flowers that smell the same is true, a base case should not be justified solely by referring to the definition of the property itself. Instead, it can be more rigorously stated as:

$$(\forall X \in \mathcal{P}(F),\ |X| \leq 1 \implies (\forall f, f' \in X, \quad f = f')) \wedge (\forall f, f' \in X, \quad f = f' \implies smells(f) = smells(f'))$$

And thus the property holds for the base case. Here is the proof provided with the underlined false claims:

Proof: Let F be the set of all flowers and let 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.

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

(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 smells(f) = smells(f').

Let $Y = X \setminus \{f\}$ and $Y' = X \setminus \{f'\}$. 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 smells(f) = smells(x) (from the induction hypothesis on Y) and smells(f') = smells(x) (from the induction hypothesis on Y'). Hence smells(f) = smells(f'), which proves the inductive step, and the theorem.

3 While Induction

Lemma 3.1. *For any BExp b and any initial state σ such that $\sigma(x)$ is even, if, $\langle$while b do $x := x+2, \sigma\rangle \Downarrow \sigma'$ then $\sigma'(x)$ is even.*

Proof. Below is a structural proof for the above lemma.

Base Case If $\langle b, \sigma\rangle \Downarrow false$ then the statement $x := x + 2$ will never be evaluated. And from while-false inference rule

$$\frac{\langle b, \sigma\rangle \Downarrow false}{\langle \text{while } b \text{ do } c, \sigma\rangle \Downarrow \sigma}$$

we know that the next state σ' will be equal to σ. Therefore, if $\sigma(x)$ is even, then $\sigma'(x)$ which is equal to $\sigma(x)$ is also even.

Induction Hypothesis If $\langle b, \sigma\rangle \Downarrow true$, we will evaluate the expression $x := x + 2$ indefinitely in this case. For an arbitrary intermediate state σ_1 of the while evaluation let's assume that if $\sigma(x)$ is even and $\langle$while $true$ do $x := x + 2, \sigma\rangle \to \cdots \to \langle (x := x + 2;)^k$ while $true$ do $x := x + 2, \sigma_1\rangle$ then $\sigma_1(x)$ is even. Here $(x := x + 2;)^k$ means that $x := x + 2$ is evaluated k times.

Inductive Step From the above arbitrary state σ_1 to the next state σ_2 we will evaluate $x := x + 2$ one more time and we will try to prove that x is again even in the next state. From the inference rule of assignment

$$\frac{\langle e, \sigma\rangle \Downarrow n}{\langle x := e, \sigma\rangle \Downarrow \sigma[x := n]}$$

Peer Review ID: 306441531 — enter this when you fill out your peer evaluation via gradescope

Questions assigned to the following page: 3, 4, and 5

we can say that $\sigma_2(x) = \sigma_1(x) + 2$ and from the inductive hypothesis we know that $\sigma_1(x)$ is even. Since, 2 is also even and the summation of two even numbers is also even, we can conclude that $\sigma_2(x)$ is also even.

Thus, we proved the above lemma by structural induction. □

4 Language Features, Large-Step

Let's define the rules for newly added commands,

(throw e) To handle the throw case we just need a simple rule that ends with a exceptional termination.

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

(try c_1 catch x c_2) In this command we should think about two different cases in which the c_1 terminates normally and raises an exception. We can add the following rule for the first case,

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

Here if c_1 terminates normally the try command also terminates normally as defined. And for the second case we can add the following rule,

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \text{ exc } n \quad \langle c_2, \sigma[x := n]\rangle \Downarrow \sigma''}{\langle \text{try } c_1 \text{ catch } x \ c_2, \sigma\rangle \Downarrow \sigma''}$$

Here, if c_1 raises an exception with value n, the variable $x \in L$ is assigned the value n and then c_2 is executed.

(after c_1 finally c_2) In this command we can have three different cases. The first one is that if c_1 terminates normally then finally command terminates by executing c_2. And we can add the following rule for this case,

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

The second case is when c_1 raises an exception with value n_1 and then c_2 ix executed and it terminates normally. We can add the following rule for this case,

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

Here it will terminate by throwing an exception with the value n_1 and since c_2 gets executed it will update the state as well.

The last option is that both c_1 and c_2 throwing errors valued n_1 and n_2 respectively. We can add the following rule for this case,

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

5 Language Features, Analysis

I will argue against the claim that it is more natural to describe "IMP with exceptions" using small-step contextual semantics. In small-step contextual semantics, for each state of the program, the next state can be deterministically decided. And even when we have if or while commands we can first evaluate the boolean expression inside the conditional and then decide what to do next based on that information and we have reduction rules such as <if true then c_1 else c_2, σ> → <c_1, σ>. However the "nature" of the introduced exceptions to this language, especially try-catch and after-finally, is non-deterministic. Unlike an if or while command, where we evaluate a boolean expression to decide the next step, there is no direct way to replicate

Question assigned to the following page: 5

the nondeterministic paths exceptions can take using the same small-step approach. Consequently, it is difficult to capture this nondeterminism with small-step semantics, suggesting that it may not be the most natural framework for describing "IMP with exceptions.

Large-step semantics can be simpler to implement because we can deduce the prior step's outcome from the final termination result. For instance, if an "after c_1 finally c_2" command ultimately terminates exceptionally, we know that c_1 ended exceptionally or both c_1 and c_2 did. Furthermore, the final exception value tells us whether c_2 terminated normally or also threw an exception, making it straightforward to reconstruct the overall control flow.