EXERCISE 2F-2: Mathematical Induction

Problem Statement

The given proof attempts to show that "All flowers smell the same" using mathematical induction. The goal is to identify and explain the flaw in the proof.

Restatement of the Proof

1. **Definition and Inductive Property:**
   - Let $\boldsymbol{F}$ be the set of all flowers.
   - Let $\boldsymbol{smells(f)}$ denote the smell of a flower $\boldsymbol{f \in F}$.
   - The property to be proved by induction:

$$\boldsymbol{P(n): \forall X \subseteq F, |X| \leq n \Rightarrow \forall f, f' \in X, smells(f) = smells(f')}$$

2. **Base Case ($\boldsymbol{n = 1}$):**
   a. Any singleton set of flowers has all flowers smelling the same by definition.

3. **Inductive Step:**
   a. Assume $\boldsymbol{P(n)}$ is true: all subsets of size at most $\boldsymbol{n}$ contain flowers that smell the same.

   b. Consider an arbitrary set $\boldsymbol{X}$ such that $\boldsymbol{|X| = n + 1}$. Pick two distinct flowers $\boldsymbol{f, f' \in X}$ and let's show that $\boldsymbol{smells(f) = smells(f')}$.

   c. Define two subsets:
      i. $\boldsymbol{Y = X - f}$, a subset of size at most $\boldsymbol{n}$.
      ii. $\boldsymbol{Y' = X - f'}$, also a subset of size at most $\boldsymbol{n}$.

   d. **Obviously, $\boldsymbol{Y}$ and $\boldsymbol{Y'}$ are sets of size at most $\boldsymbol{n}$ so the induction hypothesis holds for both of them.**

   e. **Pick any arbitrary $\boldsymbol{x \in Y \cap Y'}$. Obviously, $\boldsymbol{x \neq f}$ and $\boldsymbol{x \neq f'}$.**

   f. We have that:

   $smells(f') = smells(x)$ (from the induction hypothesis on **Y**)

   $smells(f) = smells(x)$ (from the induction hypothesis on **Y'**)

Question assigned to the following page: 2

$\Rightarrow smells(f) = smells(f')$ completing the proof.

Identifying the Flaw

- The flaw in the proof lies in the assumption that **Y** and **Y'** always have a common element **x**, which is not necessarily true. Consider a scenario where **X** consists of flowers labelled {A, B, C}. Removing **A** gives subset **Y = {B, C}**, while removing **C** gives subset **Y' = {A, B}**. If the proof relied on a common element between **Y** and **Y'**, but in cases where **X** contains distinct groups of flowers, no such common element may exist, breaking the logical progression of the argument.
- The induction hypothesis only guarantees that subsets of size **n** have uniform smell, but it does not ensure that **Y** and **Y'** must always overlap.
- If **Y** and **Y'** are disjoint (which can occur when **X** has no common element after removing **f** and **f'**), then the chain of equalities used to deduce $smells(f) = smells(f')$ breaks down.
- This invalidates the proof as it does not hold in all cases. A correct proof would need to establish a stronger induction hypothesis that accounts for cases where subsets do not share common elements. One possible approach could be demonstrating an invariant property that extends to all subsets, or redefining the problem to incorporate additional structural constraints ensuring connectivity between subsets.

Conclusion

The given proof incorrectly applies the induction hypothesis by assuming the existence of a common element between **Y** and **Y'**, which deviates from proper inductive reasoning. Induction requires that the assumption applies to all subsets of size **n**, but it does not imply that an arbitrary subset of size **n+1** will necessarily retain structural properties ensuring common elements between all possible pairwise selections. This oversight weakens the logical foundation of the step from **n** to **n+1**, making the proof invalid. Since this is not guaranteed, the inductive step does not necessarily follow, rendering the proof incorrect. A correct proof, if it exists, would need to address this gap explicitly.

Question assigned to the following page: 3

EXERCISE 2F-3: While Induction

Statement to Prove

For any boolean expression b and initial state σ such that $\sigma(x)$ is even, if $\langle \textbf{while } \boldsymbol{b} \textbf{ do } \boldsymbol{x} \coloneqq \boldsymbol{x} + \mathbf{2}, \boldsymbol{\sigma} \rangle \Downarrow \boldsymbol{\sigma'}$, then $\sigma'(x)$ is also even.

Proof by Induction on the Operational Semantics

By applying structural induction on the big-step evaluation, we analyze how the loop interacts with state transformations. This method ensures that our proof aligns with the rules governing iterative execution.

Base Case: The Loop Does Not Execute

If b evaluates to false in the initial state , the loop does not execute at all. The big-step operational semantics for while specifies:

$$\langle \textbf{while } \boldsymbol{b} \textbf{ do } \boldsymbol{c}, \boldsymbol{\sigma} \rangle \Downarrow \boldsymbol{\sigma} \textbf{if } \langle \boldsymbol{b}, \boldsymbol{\sigma} \rangle \Downarrow \textbf{false}$$

Since **x** remains unchanged in this case, the rules governing while evaluation confirm that $\sigma(x)$ retains its even property. No modifications affect its parity, ensuring correctness.

Inductive Case: The Loop Executes At least Once

If **b** evaluates to true, the loop executes **at least once**, meaning:

$\langle \boldsymbol{x} := \boldsymbol{x} + \mathbf{2}, \boldsymbol{\sigma} \rangle \Downarrow \boldsymbol{\sigma_1}$, and then recursively:

$\langle \textbf{while } \boldsymbol{b} \textbf{ do } \boldsymbol{x} := \boldsymbol{x} + \mathbf{2}, \boldsymbol{\sigma_1} \rangle \Downarrow \boldsymbol{\sigma'}$;

Where $\boldsymbol{\sigma_1}$ is the updated state after executing $\boldsymbol{x} := \boldsymbol{x} + \mathbf{2}$.

So $\boldsymbol{\sigma_1}(\boldsymbol{x}) = \boldsymbol{\sigma}(\boldsymbol{x}) + \mathbf{2}$

Applying the Inductive Hypothesis:

- Since $\boldsymbol{\sigma}(\boldsymbol{x})$ is even, adding **2** maintains evenness.
- The new state $\boldsymbol{\sigma_1}$ satisfies the inductive hypothesis.
- Given that while is applied to $\boldsymbol{\sigma_1}$, the induction assumption ensures that $\boldsymbol{\sigma'}(\boldsymbol{x})$ remains even.

Thus, $\boldsymbol{\sigma'}(\boldsymbol{x})$ remains even, completing the proof.

Question assigned to the following page: 3

Conclusion

Using structural induction on the big-step semantics, our proof aligns with the logical structure of iterative execution, ensuring correctness. We have demonstrated that if **x** starts as an even number, it remains even after any number of iterations of while b do **x := x + 2**. Consequently, *the final state* σ' satisfies the property that $\sigma'(x)$ is even.

Question assigned to the following page: 4

EXERCISE 2F-4: Language Features, Large-Step

Background

We extend the IMP programming language by introducing integer-valued exceptions, similar to exception handling in Java, ML, or C#. This enhancement requires defining a new type T to represent command terminations, revising the operational semantics, and introducing new constructs for exception handling.

Type System for Termination

A command execution in IMP can now terminate in two ways:

- **Normal Termination:** $T ::= \sigma$ where the command terminates normally, leaving the state unchanged.
- **Exceptional Termination:** $T ::= \sigma \text{exc } n$ where $n \in \mathbb{Z}$ where the command terminates abruptly due to an exception carrying the integer **n**, while leaving the state as σ.

Updated Operational Semantics Judgment

The large-step operational semantics for IMP now follows the format:

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

Where the result T is either σ (normal termination) or $\sigma \text{exc } n$ (exceptional termination).

Revised Rules for Sequential Composition

To accommodate exceptions, the semantics of sequence commands must be modified:

1. **Exception Propagation in Sequence:**

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \text{exc } n}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma' \text{exc } n}$$

   If c_1 throws an exception, the sequence also terminates exceptionally, without executing c_2.

2. **Normal Execution of Sequence:**

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \langle c_2, \sigma' \rangle \Downarrow T}{\langle c_1; c_2, \sigma \rangle \Downarrow T}$$

   If c_1 completes normally, c_2 executes, determining the final outcome.

Question assigned to the following page: 4

Large-Step Semantics for New Constructs

We define the large-step operational semantics for the new exception-handling constructs.

1. **Throw Statement**

   The **throw e** command evaluates **e** and immediately terminates with an exception:

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

2. **Try-Catch Handling**

   The try c_1 catch **x** c_2 construct executes c_1. If c_1 terminates normally, the result propagates. Otherwise, the exception value is assigned to **x** before executing c_2.

   a. **Try-Catch Without Exception:**

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

   If c_1 terminates normally, the try-catch construct terminates normally.

   b. **Try-Catch With Exception:**

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

   If c_1 throws an exception, x is assigned n before executing c_2 .

3. **Finally Handling**

   The `after` c_1 `finally` c_2 construct ensures c_2 executes regardless of whether c_1 succeeds or fails.

   a. Finally Without Exception:

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

   If c_1 terminates normally, c_2 executes, determining the final state.

Question assigned to the following page: 4

b. Finally with Initial Exception:

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

If c_1 throws an exception but c_2 completes normally, the exception is re-thrown.

c. Finally Overriding Exception:

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

If both c_1 and c_2 throw exceptions, the second exception **m overrides** the first.

Conclusion

These six rules define the large-step operational semantics for handling exceptions in IMP. The throw, try-catch, and finally constructs introduce structured exception handling, allowing robust error propagation and recovery, consistent with languages like Java and ML.

Additionally, this submission now includes refinements ensuring clarity in **exception propagation**, **final exception overriding behaviour**, and **scoping of x in try-catch**, aligning fully with the requirements set forth in the problem statement.

Question assigned to the following page: 5

EXERCISE 2F-5: Language Features, Analysis

Small-step contextual semantics, which defines program execution as a sequence of reduction steps, is a more natural choice for describing "IMP with exceptions" as it provides a detailed, step-by-step execution model suited for exception handling. Exceptions disrupt normal control flow, making stepwise execution crucial for accurately modelling their behaviour. Small-step semantics allows capturing these intermediate states explicitly, making it easier to describe when and how an exception is raised, propagated, and handled. Additionally, contextual semantics enables reductions within expressions, meaning that subexpressions are evaluated in a stepwise manner before being integrated into the overall execution process. This ensures accurate modelling of exception-related computations, such as evaluating a ***throw*** argument. This approach provides a structured way to express how execution is interrupted and resumed, reflecting the real-world implementation of exception handling in languages like Java and C#.

By contrast, large-step semantics maps an initial state to a final state without detailing intermediate transitions. This makes capturing exception flow difficult, especially with nested exceptions or finally blocks, as it lacks explicit step-by-step propagation. Large-step rules focus on final results, obscuring transient exception states. While large-step semantics might be simpler for basic imperative execution, small-step semantics is more suitable for modelling the fine details of exception propagation, making it the more precise and elegant choice for describing "IMP with exceptions."