1 2F-1 Bookkeeping

- **0 pts** Correct

2 Exercise 2F-2. Mathematical Induction

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.

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

(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.

Just to be clear about my answer, the critical flaw is in "Pick any arbitrary $x \in Y \cap Y'$" - specifically, because there is no guarantee that $Y \cap Y'$ is non-empty. If we can't choose an x, then the final two sentences in the inductive step also do not hold.

3 Exercise 2F-3. While Induction

Proof: We will prove this by inducting on the structure of derivation.

Base case: $D :: \dfrac{}{\langle \texttt{skip}, \sigma \rangle \Downarrow \sigma}$. We are given that $\sigma(x)$ is even initially, and in the case of `skip` our resulting state is the same as the input state, so $\sigma(x)$ remains even.

Case: the last rule in D was `while-false`. $D :: \dfrac{D_1 :: \langle b, \sigma \rangle \Downarrow false}{\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2 \rangle \Downarrow \sigma}$. For any initial state σ, our final state is also σ. Therefore, this case is equivalent to our base case `skip`, as for all initial states the same resulting state is reached.

Case: the last rule in D was `while-true`.

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

By inversion, D_2 follows the rule for assignment: $\dfrac{\langle e, \sigma \rangle \Downarrow n}{\langle x := e, \sigma \rangle \Downarrow \sigma[x := n]}$. By substitution, e is $x + 2$. By the induction hypothesis, $\sigma(x)$ is even. n is therefore two plus an even number, and is itself even. So, $\sigma_1(x)$ is even.

By inversion, and the fact that b must either be true or false, D_3 must follow either the `while-true` or `while-false` forms. By D_2, we know that $\sigma_1(x)$ is even. And by the induction hypothesis on D_3, within the rule $\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma \rangle \Downarrow \sigma'$, if $\sigma(x)$ is even $\sigma'(x)$ is even. Since we know $\sigma_1(x)$ is even, $\sigma'(x)$ must be even in this case.

So, for all initial states σ such that x is even, if $\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma \rangle \Downarrow \sigma'$, then $\sigma'(x)$ is even.

2 2F-2 Mathematical Induction

- **0 pts** Correct

2 Exercise 2F-2. Mathematical Induction

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.

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

(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.

Just to be clear about my answer, the critical flaw is in "Pick any arbitrary $x \in Y \cap Y'$" - specifically, because there is no guarantee that $Y \cap Y'$ is non-empty. If we can't choose an x, then the final two sentences in the inductive step also do not hold.

3 Exercise 2F-3. While Induction

Proof: We will prove this by inducting on the structure of derivation.

Base case: $D :: \dfrac{}{\langle \texttt{skip}, \sigma \rangle \Downarrow \sigma}$. We are given that $\sigma(x)$ is even initially, and in the case of `skip` our resulting state is the same as the input state, so $\sigma(x)$ remains even.

Case: the last rule in D was `while-false`. $D :: \dfrac{D_1 :: \langle b, \sigma \rangle \Downarrow false}{\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2 \rangle \Downarrow \sigma}$. For any initial state σ, our final state is also σ. Therefore, this case is equivalent to our base case `skip`, as for all initial states the same resulting state is reached.

Case: the last rule in D was `while-true`.

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

By inversion, D_2 follows the rule for assignment: $\dfrac{\langle e, \sigma \rangle \Downarrow n}{\langle x := e, \sigma \rangle \Downarrow \sigma[x := n]}$. By substitution, e is $x + 2$. By the induction hypothesis, $\sigma(x)$ is even. n is therefore two plus an even number, and is itself even. So, $\sigma_1(x)$ is even.

By inversion, and the fact that b must either be true or false, D_3 must follow either the `while-true` or `while-false` forms. By D_2, we know that $\sigma_1(x)$ is even. And by the induction hypothesis on D_3, within the rule $\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma \rangle \Downarrow \sigma'$, if $\sigma(x)$ is even $\sigma'(x)$ is even. Since we know $\sigma_1(x)$ is even, $\sigma'(x)$ must be even in this case.

So, for all initial states σ such that x is even, if $\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma \rangle \Downarrow \sigma'$, then $\sigma'(x)$ is even.

3 2F-3 While Induction

- **0 pts** Correct

4 Exercise 2F-4. Language Features, Large-Step

Our rule for `throw` is relatively simple: given a starting state σ, our resulting state is the same, with an added exception:

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

`try` is somewhat more complex, with two cases, depending on whether or not the first command c_1 terminates without exception:

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

`finally` is the most complex. We will need several cases depending on how both c_1 and c_2 terminate:

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

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma\ \texttt{exc}\ n_1 \qquad \langle c_2, \sigma\rangle \Downarrow \sigma\ \texttt{exc}\ n_2}{\langle \texttt{after}\ c_1\ \texttt{finally}\ c_2, \sigma\rangle \Downarrow \sigma\ \texttt{exc}\ n_2}$$

5 Exercise 2F-5. Language Features, Analysis

In my view, it would not be significantly simpler or more elegant to represent IMP exceptions using small-step contextual semantics. Small-step semantics are useful when a construct may be naturally defined in terms of existing constructs, for example a `while` loop may be rewritten as an `if` statement, with the `while` duplicated in the if-true branch. Exceptions, however, are fairly specialized constructs. A `try` for example can't be rewritten using existing branch constructs, despite the fact that it conditionally executes one of its commands, because the factor it branches on (the presence of an exception) is strictly part of the state - not a boolean expression. Were we to attempt to describe IMP exceptions with small-step semantics, we would need to define separate reduction rules for each of the different stateful cases, such as whether c_1 is exceptional or not in a `try`. This is essentially what we did in large-step semantics, sans the redexes and contexts needed for small-step. So, ultimately, I think it is simpler and clearer to use large-step semantics to describe this construct.

4 2F-4 Language Features, Large Step

- **0 pts** Correct

4 Exercise 2F-4. Language Features, Large-Step

Our rule for `throw` is relatively simple: given a starting state σ, our resulting state is the same, with an added exception:

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

`try` is somewhat more complex, with two cases, depending on whether or not the first command c_1 terminates without exception:

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

`finally` is the most complex. We will need several cases depending on how both c_1 and c_2 terminate:

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

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma\ \texttt{exc}\ n_1 \qquad \langle c_2, \sigma\rangle \Downarrow \sigma\ \texttt{exc}\ n_2}{\langle \texttt{after}\ c_1\ \texttt{finally}\ c_2, \sigma\rangle \Downarrow \sigma\ \texttt{exc}\ n_2}$$

5 Exercise 2F-5. Language Features, Analysis

In my view, it would not be significantly simpler or more elegant to represent IMP exceptions using small-step contextual semantics. Small-step semantics are useful when a construct may be naturally defined in terms of existing constructs, for example a `while` loop may be rewritten as an `if` statement, with the `while` duplicated in the if-true branch. Exceptions, however, are fairly specialized constructs. A `try` for example can't be rewritten using existing branch constructs, despite the fact that it conditionally executes one of its commands, because the factor it branches on (the presence of an exception) is strictly part of the state - not a boolean expression. Were we to attempt to describe IMP exceptions with small-step semantics, we would need to define separate reduction rules for each of the different stateful cases, such as whether c_1 is exceptional or not in a `try`. This is essentially what we did in large-step semantics, sans the redexes and contexts needed for small-step. So, ultimately, I think it is simpler and clearer to use large-step semantics to describe this construct.

5 2F-5 Language Features, Analysis

- **0 pts** Correct