Fakultät für Informatik und Mathematik
Prof. Dr. Martin Griebl

Summer semester 2023

Exercises for Lecture: Functional Programming
Exercise Sheet 1 (Reductions, Referential Transparency)

Problem 1 (Java: Reduction)

One important concept in functional programming are higher-order functions, i.e., functions taking other functions as argument. For example, the sum and product of a set $A = \{a_1, \ldots, a_n\}$ of numbers can be expressed as reductions. Reductions generalize step-by-step addition and multiplication with an initial value:

$$\sum_{i=1}^{n} a_i = \Big(\big((0 + a_1) + a_2\big) + \ldots + a_n\Big) = \text{reduce}(+, 0, A)$$

$$\prod_{i=1}^{n} a_i = \Big(\big((1 \cdot a_1) \cdot a_2\big) \cdot \ldots \cdot a_n\Big) = \text{reduce}(\cdot, 1, A)$$

Functions can be passed as arguments to methods in Java by passing an object having a function which calculates the function value. Consider the interface `java.util.function.BinaryOperator`[1]:

```
interface BinaryOperator<A> extends ... {
    A apply(A x, A y);
    ...
}
```

The method `apply` can represent a function such as + or ·. (The interface also contains default and static methods which we do not need in the following.)

(a) Implement a method

```
static <A> A reduce(BinaryOperator<A> f, A init, java.util.Collection<A> coll)
```

which takes a reduction operator (`f`), an initial value for the reduction (`init`) and a collection `coll` (e.g. of type `java.util.LinkedList`) as argument, performs the reduction and returns the result.

(b) Subtask (a) can be implemented in Java most naturally using a loop. Try now to construct a recursive solution

```
static <A> A reduceRec(BinaryOperator<A> f, A init, java.util.List<A> list)
```

[1] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/function/BinaryOperator.html`

which acts recursively on the passed list. Do not modify the list in the course of the recursion but pass an appropriate sublist to the recursive call (using `list.subList(..., ...)`).

Test your methods `reduce` and `reduceRec` using a small list of `Integer`s and $+, 0$ and $\cdot, 1$.

(c) The type of `reduce` (and `reduceRec`) can be generalized, i.e., not every type (of arguments and result of `f`, `init`, list elements) has to be `A`.

Generalize the signature of `reduce`. The new signature shall be compatible with the old one, i.e., code that uses the old version of `reduce` should also work with the generalized version. Use the interface `BiFunction` instead of `BinaryOperator`.

(d) Give an example for a reduction where different types are needed for the inputs of the reduction operator.

Problem 2 (Java: Referential Transparency)

The lecture has introduced the Substitution Principle that equals may be replaced for each other everywhere. Unfortunately, the assignment operation `=` in Java does *not* follow this principle, i.e., one side of the equation cannot be replaced by the other in every case. A similar statement holds for substituting method bodies at method call sites (for non-recursive methods).

(a) Give a (short) piece of Java code violating the Substitution Principle.

(b) Give a (short) piece of Java code *without reassignment or input/output*, for which the Substitution Principle does not hold.

Due date: n/a, discussion on April 26/27, 2023