Programming Paradigms Fall 2022 — Problem Sets

by Nikolai Kudasov

October 6, 2022

1 Problem set №6

Consider the following declarations:

```
type Name = String
data Grade = A | B | C | D
data Student = Student Name Grade

data Result a
    = Success a
    | Failure String

dup f x = f x x
dip f x = f (f x x)
twice f x = f (f x)
```

1. Specify the (most generic) types of `dup`, `dip`, and `twice`. Infer the type for each of the following expressions or specify a type error. Justify your answer by providing a step-by-step reasoning process. Assume that the type of integer literals is `Int`:

   (a) `dip (+) 1 2`

   (b) `dup (dip (+)) 1`

   (c) `twice dip`

   (d) `dip dip`

   (e) `twice twice twice`

   (f) `dup twice`

2. Using **explicit recursion**, implement function `studentsWithA :: [Student] -> [Name]` that returns a list of names of students with `A` grade:

   ```
   studensWithA [Student "Jack" D, Student "Jane" A]
   -- ["Jane"]
   ```

   Note: you **cannot use** `(==)` to compare grades for equality.

3. (a) Implement a polymorphic higher-order function

   ```
   whileSuccess :: (a -> Result a) -> a -> a
   ```

   that applies a function repeatedly, as long as the result is a `Success`, otherwise, it returns the last value.

   ```
   f n | n > 100   = Failure "input is too large"
       | otherwise = Success (2 * n)

   example1 = whileSuccess f 1  -- 64
   ```

(b) Implement a polymorphic higher-order function

```
applyResult :: Result (a -> b) -> Result a -> Result b
```

that applies a given function to a given argument when both are available, and returns the first (leftmost) error message otherwise.

```
applyResult (Success length) (Success [1, 2, 3])        -- 3
applyResult (Failure "no function") (Failure "no arg")  -- Failure "no function"
```

(c) Implement a polymorphic higher-order function

```
fromResult :: (a -> b) -> (String -> b) -> Result a -> b
```

that processes any given result:

```
fromResult (+1) length (Success 3)              -- 4
fromResult (+1) length (Failure "not a number") -- 12
```

(d) Implement a polymorphic higher-order function

```
combineResultsWith :: (a -> b -> c) -> Result a -> Result b -> Result c
```

that applies a function to two results, if both are present, and returns the first (leftmost) error message otherwise.

```
combineResultsWith (+) (Result 2) (Result 3)
-- Success 5

combineResultsWith (+) (Failure "x is undefined") (Failure "crash")
-- Failure "x is undefined"
```