Programming Paradigms - Problem Set 6

Jaffar Totanji
j.totanji@innopolis.university

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)

Solution:

1. First, we present the polymorphic types of dup, dip and twice:

dup :: (a -> a -> b) -> a -> b
dip :: (a -> a -> a) -> a -> (a -> a)
twice :: (a -> a) -> a -> a

a) dip (+) 1 2

- We have the following typings:

  dip :: (a1 -> a1 -> a1) -> a1 -> (a1 -> a1)
  (+) :: Int -> Int -> Int
  1 :: Int
  2 :: Int

- Let's now match the actual type of each argument of dip with the corresponding expected type:

  (a1 -> a1 -> a1) = (Int -> Int -> Int) => a1 = Int (1st argument of dip "+")
  a1 = Int (2nd argument of dip "1")
  a1 = Int (2nd argument of dip (+) 1 "2", its 1st argument is the result of dip (+) 1)

- All of these constraints are resolved with a1 = Int:

dip (+) 1 2 :: Int

b) dup (dip (+)) 1

- We have the following typings:

  dup :: (a1 -> a1 -> b1) -> a1 -> b1
  dip :: (a2 -> a2 -> a2) -> a2 -> (a2 -> a2)
  (+) :: Int -> Int -> Int
  1 :: Int

- Let's now match the actual type of each argument of dup with the corresponding expected type starting with the arguments of dup themselves:

  dip :: (a2 -> a2 -> a2) -> a2 -> (a2 -> a2)

  and

  dip (+):
  (a2 -> a2 -> a2) = Int -> Int -> Int (1st argument of dip “+”)
  => a2 = Int
  => dip (+) :: a2 -> a2 -> a2
  where a2 = Int

  => dip (+) :: Int -> Int -> Int

  and now dup (dip (+)) 1:
  (a1 -> a1 -> b1) = Int -> Int -> Int (1st argument of dup “dip (+)” from prev. step)
  => a1 = Int
  => b1 = Int

  a1 = Int (2nd argument of dup “1”)

- All of these constraints are resolved with a1 = Int, b1= Int, a2 = Int:
  dup (dip (+)) 1 :: Int

  (a1 -> a1 -> b1) = (a2 -> a2) (from the 1st argument of dup “dip (+)”)
  a1 = Int (from the 2nd argument of dup “1”)

c) twice dip

- We have the following typings:

  twice :: (a1 -> a1) -> a1 -> a1
  dip :: (a2 -> a2 -> a2) -> a2 -> (a2 -> a2)

- Let's now match the actual type of each argument of twice with the corresponding expected type:

  (a1 -> a1) = (a2 -> a2 -> a2) -> a2 -> (a2 -> a2) (1st argument of twice “dip”)

  => a1 = a2 -> a2 -> a2
  and
  => a1 = a2 -> (a2 -> a2) which is the same as a2 -> a2 -> a2

  then a1 = a2 -> a2 -> a2

- All of these constraints are resolved with a1 = a2 -> a2 -> a2:

  twice dip :: (a2 -> a2 -> a2) -> (a2 -> a2 -> a2)

d) dip dip

- We have the following typings:

  dip :: (a1 -> a1 -> a1) -> a1 -> (a1 -> a1) (outermost)
  dip :: (a2 -> a2 -> a2) -> a2 -> (a2 -> a2) (innermost)

- Let's now match the actual type of each argument of dip with the corresponding expected type:

  (a1 -> a1 -> a1) = (a2 -> a2 -> a2) -> a2 -> (a2 -> a2) (1st argument of dip “dip”)

  => a1 = (a2 -> a2 -> a2)

  and

  => a1 = a2

and
=> a1 = (a2 -> a2)

- This results in a1 being an infinite type, which in turn results in a Type Error.

e) twice twice twice

- We have the following typings:

  twice :: (a1 -> a1) -> a1 -> a1 (1st)
  twice :: (a2 -> a2) -> a2 -> a2 (2nd)
  twice :: (a3 -> a3) -> a3 -> a3 (3rd)

- Let's now match the actual type of each argument of twice with the corresponding expected type:

  (a1 -> a1) = (a2 -> a2) -> a2 -> a2 (1st argument of 1st twice "2nd twice")

  => a1 = a2 -> a2

  a1 = (a3 -> a3) -> a3 -> a3 (2nd argument of 1st twice "3rd twice")

  => a1 = (a3 -> a3) -> a3 -> a3

  => a2 = (a3 -> a3)

- All of these constraints are resolved with a1 = a2 -> a2 and a2 = a3 -> a3:

  twice twice twice :: a2 -> a2
  where a2 = a3 -> a3

f) dup twice

- We have the following typings:

  twice :: (a1 -> a1) -> a1 -> a1
  dup :: (a2 -> a2 -> b2) -> a2 -> b2

- Let's now match the actual type of each argument of dup with the corresponding expected type:

  (a2 -> a2 -> b2) = (a1 -> a1) -> a1 -> a1

  => a2 = a1 -> a1

and

=> a2 = a1

and

=> b2 = a1

- This results in a2 being an infinite type, which results in a Type Error.