

Prof. Dr. Jürgen Giesl  
René Thiemann

## Exercises *Functional Programming* – Sheet 9

Solutions will be collected until Thursday, Dec 19, 2002 in the exercise course.

### Exercise 1 ( 2 + ... + 2 = 12 points)

For the following Haskell declarations

```
data List a = Empty | Cons a (List a)

exp1 := let not' = \x -> if x == True then False else True
        in not' y

exp2 := let even' = \x -> if x == 0 then True
                        else not (even' (x - 1))
        in even' 4

exp3 := let even'' = \x -> if x == 0 then True
                          else (if x == 1 then False
                                else even'' (x - 2))
        in even'' 5

exp4 := let double = \x -> if x == 0 then 0
                          else (2 + double (x - 1))
        in double 2

exp5 := let xy = (not (snd xy), True)
        in fst xy

exp6 := let ones = Cons 1 ones
        in ones
```

give the value of  $\mathcal{V}al[\![exp_i]\!] \rho$ , where  $i \in \{1 \dots 6\}$ ,  $\rho = \omega + \rho'$  and  $\rho'(y) = \text{True}$ ,  $\rho'(x) = 3$ .

Describe your computation in detail and for each higher-order function  $f : \text{Dom} \rightarrow \text{Dom}$  that occurs in the calculation, determine what the function  $f^i(\perp)$ ,  $i \in \mathbb{N}$ , computes.

Remark: `exp5` is equivalent to the non-simple Haskell-expression

```
exp5' := let (x,y) = (not y, True) in x
```

This construct was used in Sheet 4, Exercise 3(d) (`minTree`).

## Exercise 2 ( 2 + 2 + 2 points)

We define the following algebraic data type and the following functions.

```
data Nats = Zero | Succ Nats
data List a = Empty | Cons a (List a)
```

```
pred :: Nats -> Nats
pred (Succ x) = x
```

```
isZero :: Nats -> Bool
isZero Zero = True
isZero _     = False
```

```
head :: List a -> a
head (Cons x _) = x
```

```
tail :: List a -> List a
tail (Cons _ xs) = xs
```

```
isEmpty :: List a -> Bool
isEmpty Empty = True
isEmpty _     = False
```

```
fst :: (a,b) -> a
fst (x,y) = x
```

```
snd :: (a,b) -> b
snd (x,y) = y
```

Give simple Haskell-expressions that are equivalent to the following Haskell-expressions. Your solutions may make use of some of the functions defined above. You don't have to use the transformation rules which will be presented next Wednesday in the lecture.

```
(a) let length = \ys -> case ys of
                        Empty      -> 0
                        (Cons x xs) -> 1 + length xs
    in length someList
```

```
(b) plus numberOne numberTwo
    where plus Zero      y = y
          plus (Succ x) y = Succ (plus x y)
```

```
(c) plus (numberOne,NumberTwo)
    where plus (Zero, y) = y
          plus (Succ x, y) = Succ (plus(x,y))
```