

Prof. Dr. Jürgen Giesl
Peter Schneider-Kamp

Exercises *Functional Programming* – Sheet 9

Solutions will be collected until Wednesday, June 29, 2005 in the exercise course.

Exercises can be solved both in English and in German.

Exercise 1 (2 + 2 + 3 points)

In the lecture, 12 transformation rules were presented which transform complex Haskell-expressions into simple Haskell-expressions. Extend the set of these rules with additional rules which transform further Haskell-expressions to simple Haskell-expressions.

- (a) Add a transformation rule for match-expressions with patterns using `@`. The pattern `var@pat` matches an expression `exp` iff `pat` matches `exp`. Moreover, the variable `var` is instantiated by `exp`.

For example, the pattern `m@(Cons x xs)` matches the expression `Cons 5 Nil` and `x` is instantiated to `5` and `xs` to `Nil`.

- (b) Add a transformation rule for match-expressions with patterns using `x+n` where `x` is a variable and `n` is an integer greater than 0. The pattern `(x+n)` matches an integer `m` iff `x` is at least `n`. Moreover, the variable `x` is instantiated by the value of `m-n`.

For example, the pattern `(x+2)` matches the expression `5` and binds `x` to the value `3`.

- (c) Add a transformation rule for pattern declarations of the following form:

$$\text{constr}_n \underline{\text{pat}}_1 \dots \underline{\text{pat}}_n = \underline{\text{exp}}$$

As an example for such a pattern declaration, consider the following expression which uses a pattern declaration to compute the first element of a list.

```
let (head:_) = [1,2,3] in head
```

Exercise 2 (6 + 5 points)

The data structure `List` is defined by

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

- (a) Transform the following Haskell-expression to a simple Haskell-expression using the transformation rules (1)-(9) from the lecture. Please give all intermediate expressions and indicate in each step which transformation rule was used.

```
let dup Nil = Nil
    dup (Cons x xs) = Cons x (Cons x (dup xs))
in dup (Cons 1 (Cons 2 Nil))
```

- (b) Consider the data structure `Tree`.

```
data Tree a = Node a (List (Tree a))
```

Transform the following Haskell-expression to a Haskell expression of the form `let ts = ...` using the transformation rules (1), (11), and (12) from the lecture such that `ts` is a pair of functions equivalent to `(sizeTree, sizeSet)`.

```
let sizeTree (Node x ts) = 1 + (sizeSet ts)
    sizeSet Nil = 0
    sizeSet (Cons t ts) = sizeTree t + sizeSet ts
in sizeTree (Node 1 (Cons (Node 2 Nil) Nil))
```

Hint:

You can simplify your results during the transformation as in the example in the lecture:

- (i) `if (isan-tuple exp) then exp1 else exp2` can be replaced by `exp1`.
- (ii) `(\var->exp) exp'` can be replaced by `exp`, where all free occurrences of `var` are replaced by `exp'`.