

Prof. Dr. Jürgen Giesl  
René Thiemann

## Exercises *Functional Programming* – Sheet 13

Solutions will be collected until Thursday, Jan 30, 2003 in the exercise course.

### Exercise 1 ( (2+2+1) + (2+5\*) + 2 points)

The representation of natural numbers and boolean values in the pure lambda calculus were given in the lecture.

- Give translations of the constants `plus`, `times`, and `not` into pure lambda terms such that  $\forall n, m \in \mathbb{N} : \overline{\text{plus } n \ m} \rightarrow_{\beta}^* \overline{n + m}$ ,  $\overline{\text{times } n \ m} \rightarrow_{\beta}^* \overline{n * m}$  and  $\overline{\text{not True}} \rightarrow_{\beta}^* \overline{\text{False}}$ ,  $\overline{\text{not False}} \rightarrow_{\beta}^* \overline{\text{True}}$ .
- Give translations for the constants `equals_0`, `bigger_0`, and `Pred` that correspond to the following Haskell-functions.

```

equals_0 0      = True
equals_0 (n+1) = False

```

```

bigger_0 0      = False
bigger_0 (n+1) = True

```

```

pred (n+1) = n
pred 0     = 0

```

*Hint:* In the translation of `Pred` it is useful to switch to another representation of natural numbers which is based on pairs.

- Please implement a function `to_pure :: Term -> Term` that translates a lambda-term into a pure one. `to_pure` should be able to deal with every translation presented in the lecture and the translations from this exercise for all  $n, m, \dots$ . Test your transformations of the above constants. You can use the framework given on the website.

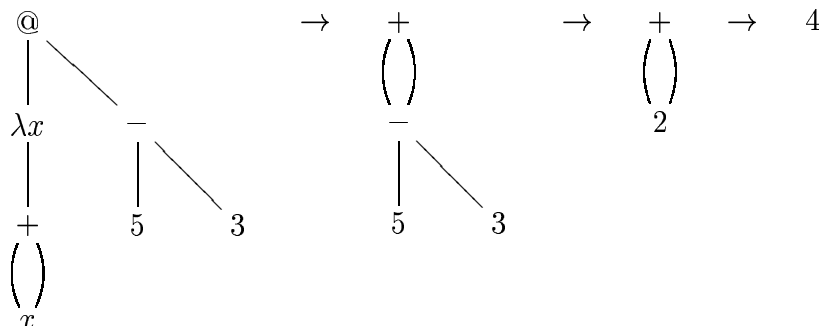
### Exercise 2 ( 6 points)

If we want to implement  $\rightarrow_{\beta\delta}$  in a more efficient way we should not use leftmost-outermost-evaluation, but lazy-evaluation. The difference to lo-evaluation is, that duplicated expressions

---

\*5 bonus points for  $\overline{\text{Pred}}$

are evaluated only once. So if we want to evaluate  $(\lambda x.x + x)(5 - 3)$ , we only want to perform the steps  $(\lambda x.x + x)(5 - 3) \rightarrow (5 - 3) + (5 - 3) \rightarrow 2 + 2 \rightarrow 4$ . So, the subterm  $5 - 3$  is evaluated only once. Using a graphical representation, the evaluation should have the following form (here, @ denotes application).



So in the first term, the variable  $x$  is shared and in the second term, the subterm  $5 - 3$  is shared.

Develop a data structure that corresponds to this graphical representation of terms. Describe in pseudo-code a valid transformation `to_graph :: Term -> GraphTerm` from an ordinary lambda-term to a term represented as a graph. This transformation should at least combine variables whenever possible. Here, you should also consider problems of the following type: Can we share a free variable „ $x$ “ with a bound variable „ $x$ “?

Describe in pseudo-code an algorithm for substitution `subst :: (Variable, GraphTerm) -> GraphTerm -> GraphTerm` for terms in graphical representation.

Your transformation and substitution should work in a way such that

- `subst (x, to_graph t) (to_graph t') ≡ to_graph ( t' [x/t] )`
- The number of nodes needed to represent `t' [x/t]` is less than the sum of the nodes needed to represent `t` and `t'`.

*Hint:* You may use global variables (a counter might be useful) and you are not restricted to use Haskell by the design of your pseudo-code algorithms.

### Exercise 3 (1 + 2 points)

In the lecture, `fix` was translated into the lambda-term  $\overline{\text{fix}} = (\lambda xy.y(xy))(\lambda xy.y(xy))$ . Then  $\overline{\text{fix}} z \rightarrow_{\beta}^* z(\overline{\text{fix}} z)$  holds.

There exists another fix-point-combinator called  $Y$  with  $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ . Please show:

- $Y z \not\rightarrow_{\beta}^* z(Y z)$
- $Y z \leftrightarrow_{\beta}^* z(Y z)$