**Technische Universität München**
**Institut für Informatik**
Prof. Tobias Nipkow, Ph.D.
Simon Wimmer

**Lambda Calculus**
**Winter Term 2017/18**
Exercise Sheet 3

## Exercise 1 (Fixed-point Combinator)

- Use a fixed-point combinator to compute the length of lists on the encoding given in the last tutorial.
- Find an easier solution for the encoding from the last homework.

## Exercise 2 ($\beta$-reduction on de Bruijn Preserves Substitution)

We consider an alternative representation of $\lambda$-terms that is due to de Bruijn. In this representation, $\lambda$-terms are defined according to the following grammar:

$$d ::= i \in \mathbb{N} \mid d_1\ d_2 \mid \lambda\ d$$

Define substitution and $\beta$-reduction on de Bruijn terms.

Now restate Lemma 1.2.5 for de Bruijn terms and prove it:

$$s \rightarrow_\beta s' \implies s[u/x] \rightarrow_\beta s'[u/x]$$

## Homework 3 (Multiplication)

Define multiplication using $\mathsf{fix}$ and prove its correctness. You can assume that you are given a predecessor function $\mathsf{pred}$ such that:

- $\mathsf{pred}\ \underline{0} \rightarrow_\beta^* \underline{0}$
- $\mathsf{pred}\ (\mathsf{succ}\ n) \rightarrow_\beta^* n$

## Homework 4 (Efficient Substitution on de Bruijn)

We define a new lifting operator $- \uparrow_-^-$:

$$i \uparrow_l^n = \begin{cases} i, & \text{if } i < l \\ i + n, & \text{if } i \geq l \end{cases}$$
$$(d_1\ d_2) \uparrow_l^n = d_1 \uparrow_l^n\ \ d_2 \uparrow_l^n$$
$$(\lambda\ d) \uparrow_l^n = \lambda\ d \uparrow_{l+1}^n$$

Use $- \uparrow_-^-$ to define a more efficient version of substitution for de Bruijn terms that only applies lifting in the case that a variable is actually replaced by a term. Prove that $t[s/0]$ yields the same result for both, your new version and the version from the tutorial. *Hint*: Find a suitable generalization first.

#### Homework 5 (Expanding Lets)

We have a language with `let`-expressions, i.e.:

$$t = v \mid t\ t \mid \texttt{let}\ v = t\ \texttt{in}\ t$$

Write a program which expands all `let`-expressions. The `let`-semantics are:

$$(\texttt{let}\ v = t_1\ \texttt{in}\ t_2) = (\lambda v.\ t_2)\ t_1$$

If you want to use a language different from ML, Ocaml, Haskell, Java, and Python, please talk to the tutor first.