Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lukas Stevens

Lambda Calculus
Winter Term 2021/22
Solutions to 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.

Solution

- We use the Y-combinator:

  $$\mathsf{y} := \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

  The Y-combinator satisfies the property $\mathsf{y}\ f =_\beta^* f\ (y\ f)$.

  Recall how the Church numerals are implemented:

  $$\mathsf{zero} := \lambda f\ x.\ x \qquad\qquad \mathsf{succ} := \lambda n\ f\ x.\ f\ (n\ x)$$

  In a programming language with recursion, length would be implemented as follows:

  ```
  len x = if null x then 0 else Succ (len (tl x))
  ```

  We obtain the following definition:

  $$\mathsf{length} := \mathsf{y}\ (\lambda l\ x.\ (\mathsf{null}\ x)\ \mathsf{zero}\ (\mathsf{succ}\ (l\ (\mathsf{tl}\ x))))$$

- $\mathsf{length} := \lambda l.\ l\ (\lambda x.\ \mathsf{succ})\ \underline{0}$

Exercise 2 (β-reduction on de Bruijn Preserves Substitution)

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

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

Define substitution and β-reduction on de Bruijn terms.

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

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

Solution

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

$$i[t/j] = \begin{cases} i \text{ if } i < j \\ t \text{ if } i = j \\ i-1 \text{ if } i > j \end{cases}$$
$$(d_1\ d_2)[t/j] = (d_1[t/j])\ (d_2[t/j])$$
$$(\lambda d)[t/j] = \lambda(d[t \uparrow_0 /j+1])$$

We now have $(\lambda d)e \to_\beta d[e/0]$. The other cases for $\to_\beta$ remain the same as before. Similarly to the lecture, we first prove the key property (*)

$$i < j+1 \longrightarrow t[v \uparrow_i /j+1][u[v/j]/i] = t[u/i][v/j]$$

by induction on t. Now

$$s \to_\beta s' \implies s[u/i] \to_\beta s'[u/i]$$

can be proved by induction on $\to_\beta$ for arbitrary u and i.

The base case is the hardest. We need to show

$$((\lambda s)\ t)[u/i] \to_\beta s[t/0][u/i]$$

for arbitrary s, t. Proof:

$$\begin{aligned} & ((\lambda s)\ t)[u/i] \\ = \; & (\lambda s[u \uparrow_0 /i+1])\ t[u/i] && \text{Def. of substitution} \\ \to_\beta \; & s[u \uparrow_0 /i+1][t[u/i]/0] \\ = \; & s[t/0][u/i] && (*) \end{aligned}$$

The other cases follow trivially from the rules of $\to_\beta$ and the definition of substitution.

Homework 3 (Multiplication)

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

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

Homework 4 (Efficient Substitution on de Bruijn)

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

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

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.