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

Lambda Calculus
Winter Term 2023/24
Solutions to Exercise Sheet 3

Exercise 1 (Fixed-point Combinator)

a) In the last tutorial, we came up with an encoding for lists together with the functions nil, cons, null, hd, and tl. Use a fixed-point combinator to compute the length of a list in this encoding.

b) In the last homework, we encoded lists with the fold encoding, i.e. a list $[x, y, z]$ is represented as $\lambda c\, n.\ c\, x\, (c\, y\, (c\, z\, n))$. Define a length function for lists in this encoding.

Solution

a) 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 $\mathsf{y}\ f =_\beta^* f\ (\mathsf{y}\ f)$.

Recall how the Church numerals are implemented:

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

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

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

We obtain the following definition:

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

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

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:

$$d ::= i \in \mathbb{N}_0 \mid d_1\ d_2 \mid \lambda\ d$$

a) Convert the terms $\lambda x\ y.\ x$ and $\lambda x\ y\ z.\ x\ z\ (y\ z)$ into terms according to de Bruijn.

b) Convert the term $\lambda\ ((\lambda\ (1\ (\lambda\ 1)))\ (\lambda\ (2\ 1)))$ into our usual representation.

c) Define substitution and β-reduction on de Bruijn terms.

d) 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

a) $\lambda\ \lambda\ 1$ and $\lambda\ \lambda\ \lambda\ (2\ 0\ (1\ 0))$.

b) This example is taken from the Wikipedia article on de Bruijn indices where 1-based indices are used. For 1-based indices the solution is $\lambda z.\ (\lambda y.\ y\ (\lambda x.\ x))\ (\lambda x.\ z\ x)$. For 0-based indices we have $\lambda z.\ (\lambda y.\ z\ (\lambda x.\ y))\ (\lambda x.\ f\ z)$ where f is some free variable.

c)

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

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

We now define $(\lambda\ d)\ e \to_\beta d[e/0]$. Note that the β-reduction removes the λ surrounding the term d. This means that we need to decrease the indices of all free variables in d by one, which is taken care of by the third case for $i[t/j]$. The other cases for $\to_\beta$ remain the same as before.

d) The following proof is also available as an Isabelle theory.

Similarly to the fourth assertion of Lemma 1.1.5 in 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:

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

Homework 4 (Efficient Substitution on de Bruijn)

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

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

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:

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

You can find a Haskell template for this exercise here.