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:

\[ y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

The Y-combinator satisfies the property \( y f \rightarrow^* f (y f) \).

Recall how the Church numerals are implemented:

\[ \text{zero} := \lambda f. x. x \quad \text{succ} := \lambda n. f. x. f (n x) \]

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

\[ \text{len } x = \text{if null } x \text{ then } 0 \text{ else Succ } (\text{len } (tl x)) \]

We obtain the following definition:

\[ \text{length} := y (\lambda l. x. (\text{null } x) \text{ zero } \text{ succ } (l (tl x))) \]

• \( \text{length} := \lambda l. l \text{ add } 0 \)

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:

\[ d ::= i \in \mathbb{N} | d_1 \; d_2 | \lambda \; d \]

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

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

\[ s \rightarrow^*_\beta s' \implies s[u/x] \rightarrow^*_\beta s'[u/x] \]
Solution

\[ i \uparrow_{t} = \begin{cases} 
  i, & \text{if } i < l \\
  i + 1, & \text{if } i \geq l 
\end{cases} \]

\[(d_1 \ d_2) \uparrow_{t} = d_1 \uparrow_{i} \ d_2 \uparrow_{t} \]

\[(\lambda \ d) \uparrow_{t} = \lambda \ d \uparrow_{t+1} \]

\[ i[t/j] = \begin{cases} 
  i & \text{if } i < j \\
  t & \text{if } i = j \\
  i - 1 & \text{if } i > j 
\end{cases} \]

\[(d_1 \ d_2)[t/j] = (d_1[t/j]) \ (d_2[t/j]) \]

\[(\lambda d)[t/j] = \lambda (d[t \uparrow_{0}/j + 1]) \]

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

\[ i < j + 1 \rightarrow t[v \uparrow_{i}/j + 1][u[v/j]/i] = t[u/i][v/j] \]

by induction on \(t\). Now

\[ s \rightarrow_{\beta} s' \implies s[u/i] \rightarrow_{\beta} s'[u/i] \]

can be proved by induction on \(\rightarrow_{\beta}\) for arbitrary \(u\) and \(i\).

The base case is the hardest. We need to show

\[ ((\lambda s) \ t)[u/i] \rightarrow_{\beta} s[t/0][u/i] \]

for arbitrary \(s, t\). Proof:

\[ ((\lambda s) \ t)[u/i] = (\lambda s[u \uparrow_{0}/i + 1]) \ t[u/i] \quad \text{Def. of substitution} \]

\[ \rightarrow_{\beta} s[u \uparrow_{0}/i + 1][t[u/i]/0] \]

\[ = s[t/0][u/i] \quad (*) \]

The other cases follow trivially from the rules of \(\rightarrow_{\beta}\) and the definition of substitution.

Homework 3 (Multiplication)

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

- \(\text{pred } 0 \rightarrow_{\beta} 0\)
- \(\text{pred } (\text{succ } n) \rightarrow_{\beta} n\)
Homework 4 (Efficient Substitution on de Bruijn)

We define a new lifting operator $\uparrow^n$:

\[
i \uparrow^n_l = \begin{cases} 
  i, & \text{if } i < l \\
  i + n, & \text{if } i \geq l
\end{cases}
\]

\[
(d_1 \ d_2) \uparrow^n_l = d_1 \uparrow^n_l \ d_2 \uparrow^n_l \\
(\lambda d) \uparrow^n_l = \lambda d \uparrow^n_{l+1}
\]

Use $\uparrow^n$ 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. \textit{Hint}: Find a suitable generalization first.

Homework 5 (Expanding Lets)

We have a language with \texttt{let}-expressions, i.e.:

\[
t = v \mid t \mid t \mid \texttt{let } v = t \texttt{ in } t
\]

Write a program which expands all \texttt{let}-expressions. The \texttt{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.