ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Dr. A. Lochbihler and Prof. Dr. P. Müller

Formal Methods and Functional Programming

Exercise Sheet 8: Induction

Submission deadline: Monday, April 14, 2014, 11:00 am

Note that the tutors for exercise groups will mostly be different for the second half of the course. Please have a look at the course web page (`http://www.infsec.ethz.ch/education/ss2014/fmfp`) to see who is your tutor. For questions about the new exercise groups, please contact Alex Summers (`alexander.summers@inf.ethz.ch`).

Please submit your solution before **11am** on the submission date specified above. Solutions can be submitted via e-mail to your tutor or by using the boxes in front of **CAB F 51.2**. Make sure that every sheet contains your name, the exercise sheet number and your tutor's name. Don't forget to staple your pages if you submit more than one page.

Assignment 1

The background of this assignment is a simple run-length encoding scheme (`http://en.wikipedia.org/wiki/Run-length_encoding`). In our case, the input data represented by a list of natural numbers is encoded as a list of natural numbers of even length. The encoded representation has the form $n_1 : v_1 : n_2 : v_2 : \ldots : []$, where each pair $n_i : v_i$ denotes, that the input data contained n_i consecutive occurrences of v_i. For example, the input $1 : 1 : 1 : 5 : 5 : 5 : 5 : []$ will be encoded as $3 : 1 : 4 : 5 : []$

`encode` computes the run-length encoding of a given list of natural numbers, represented as a list of natural numbers. It is defined in terms of the auxiliary function `encode'` that performs the actual encoding.

```
encode []     = []                                   -- (E1)
encode (x:ys) = encode' ys 1 x []                    -- (E2)

encode' [] n z cs = cs++[n,z]                        -- (E'1)
encode' (x:ys) n z cs
  | x == z    = encode' ys (n+1) x cs                -- (E'2)
  | otherwise = encode' ys 1 x (cs++[n,z])           -- (E'3)
```

decode decodes run-length encoded data represented as a list of natural numbers. replicate x y creates a list of length x where each element is y.

```
decode []  = []                                      -- (D1)
decode [x] = []                                      -- (D2)
decode (x:y:zs) = (replicate x y) ++ (decode zs)     -- (D3)

replicate 0 y = []                                   -- (R1)
replicate x y = y:(replicate (x-1) y)                -- (R2)
```

srclen computes the length of the source data from the encoded representation.

```
srclen []       = 0                                  -- (S1)
srclen [x]      = 0                                  -- (S2)
srclen (x:y:zs) = x + srclen zs                      -- (S3)
```

Note: The pathological cases $(D2)$ and $(S2)$ are only there to make the two functions total.

Prove the following lemma[1]:

$\forall\, n, z :: \texttt{Nat}, \forall\, xs, cs :: \texttt{[Nat]} \cdot (\texttt{length cs \% 2 = 0} \;\;\Rightarrow$
$\quad \texttt{srclen (encode' (decode } xs\texttt{) } n\ z\ cs\texttt{)} = \texttt{(srclen } xs\texttt{) + } n \texttt{ + (srclen } cs\texttt{)})$

Extra Lemmas: You may use the following lemmas without proving them:

L1: $\forall\, x, y, n :: \texttt{Nat}, \forall\, zs, cs :: \texttt{[Nat]} \cdot$
$\quad \texttt{encode' ((replicate } x\ y\texttt{) ++ } zs\texttt{) } n\ y\ cs = \texttt{encode' } zs\ \texttt{(}x\texttt{+}n\texttt{)}\ y\ cs$

L2: $\forall\, x :: \texttt{Nat}, \forall\, ys, zs :: \texttt{[Nat]} \cdot\ (x\texttt{:}ys\texttt{) ++ } zs \texttt{ = } x\texttt{:(}ys \texttt{ ++ } zs\texttt{)}$

L3: $\forall\, x, y :: \texttt{Nat}, \forall\, zs :: \texttt{[Nat]} \cdot$
$\quad (\texttt{length } zs \texttt{ \% 2 = 0} \;\;\Rightarrow\;\; \texttt{srclen (}zs\texttt{++[}x\texttt{,}y\texttt{]) = srclen } zs \texttt{ + } x)$

L4: $\forall\, x, y :: \texttt{Nat}, \forall\, zs :: \texttt{[Nat]} \cdot \texttt{length } zs \texttt{ \% 2 = 0} \;\;\Rightarrow\;\; \texttt{length (}zs\texttt{++[}x\texttt{,}y\texttt{]) \% 2 = 0}$

Assignment 2

Use the result of assignment 1 to prove, that

$$\forall\, xs :: \texttt{[Nat]} \cdot \texttt{srclen (encode (decode } xs\texttt{))} = \texttt{srclen } xs$$

Hints: You may also use the Extra Lemmas from assignment 1 without proving them.

[1]Hint: we recommend using strong structural induction (as explained in the exercise session) on one of the list-typed variables. Alternatively, you could use induction on the *length* of one of the lists.

Assignment 3

This assignment introduces the principle of *induction on the shape of derivation trees* in the context of the Lambda Calculus[2]. Induction on the shape of derivation trees will be used extensively throughout the second half of the course, although on other derivation systems that will be introduced in the upcoming lectures.

We recall the relevant definitions, here:

Types:

$$\begin{array}{rl} \tau ::= & \tau \to \tau \\ & a \end{array}$$

Where $a \in \mathcal{V}_{\mathcal{T}}$ is a type variable.

Typing rules: (we use underlined symbols for the "meta-variables" in the rule definitions, to differentiate from the particular variables/terms etc. used in instantiations of the rules).

$$\frac{}{\underline{\Gamma}, \underline{x} : \underline{\tau} \vdash \underline{x} :: \underline{\tau}}\ (\textsc{Var}) \quad \frac{\underline{\Gamma}, \underline{x} : \underline{\tau_1} \vdash \underline{t} :: \underline{\tau_2}}{\underline{\Gamma} \vdash (\lambda \underline{x}.\underline{t}) :: \underline{\tau_1} \to \underline{\tau_2}}\ (\textsc{Abs}) \quad \frac{\underline{\Gamma} \vdash \underline{t_0} :: \underline{\tau_1} \to \underline{\tau_2} \quad \underline{\Gamma} \vdash \underline{t_1} :: \underline{\tau_1}}{\underline{\Gamma} \vdash (\underline{t_0}\ \underline{t_1}) :: \underline{\tau_2}}\ (\textsc{App})$$

Type Substitution (new): We define a *type substitution* operation $\tau_1[a \mapsto \tau_2]$ (which can be read as "τ_1 with every occurrence of a replaced by τ_2"), where τ_1 and τ_2 are types and a is a type variable, by the following cases:

$$\begin{array}{rcll} a[a \mapsto \tau_2] & = & \tau_2 & \\ b[a \mapsto \tau_2] & = & b & \text{(if } a \not\equiv b\text{)} \\ (\tau_3 \to \tau_4)[a \mapsto \tau_2] & = & (\tau_3[a \mapsto \tau_2]) \to (\tau_4[a \mapsto \tau_2]) & \end{array}$$

We also extend this substitution to apply to *typing contexts* (by applying the substitution to every type in the context):

$$\Gamma[a \mapsto \tau_2] = \{x{:}(\tau[a \mapsto \tau_2]) \mid x{:}\tau \in \Gamma\}$$

In particular, we have the following (useful, for this exercise) property of non-empty contexts: $(\Gamma, x{:}\tau)[a \mapsto \tau_2] = (\Gamma[a \mapsto \tau_2], x{:}(\tau[a \mapsto \tau_2]))$

Exercise: Prove the following *substitution lemma* for Lambda Calculus type derivations (we use $D, D_1, D_2, \ldots$ as variables for derivation trees):

$\forall\, \Gamma, a, t, \tau_1, \tau_2, D_1 \cdot$
$\ \mathtt{root}(D_1) = \Gamma \vdash t :: \tau_1$
$\quad \Rightarrow$
$\ \exists D_2 \cdot \mathtt{root}(D_2) = \Gamma[a \mapsto \tau_2] \vdash t_1 :: \tau_1[a \mapsto \tau_2])$

(see next page for hints)

[2]Recall slides 33 onwards of the FP lecture "Higher-order Programming and Types"

Hints / Reminders (see also exercise session)

Given a derivation tree D, we denote by $\texttt{root}(D)$ the judgement at the root of the tree, i.e., the conclusion of the last derivation rule applied in D. Thus, informally, the lemma to prove says that for any derivation of a typing judgement, there is another derivation of the corresponding typing judgement where a type substitution has been applied, throughout. More information about induction on the shape of derivation trees will be given to you in the further course of the lecture.

In order to simplify the writing of the proof and avoid duplication, it is suggested that you define:

$$
\begin{aligned}
&P(D_1) \triangleq \\
&\forall\, \Gamma, a, t, \tau_1, \tau_2 \cdot \\
&\qquad \texttt{root}(D_1) = (\Gamma \vdash t :: \tau_1) \\
&\qquad\qquad \Rightarrow \\
&\qquad \exists\, D_2 \cdot \texttt{root}(D_2) = (\Gamma[a \mapsto \tau_2] \vdash t_1 :: \tau_1[a \mapsto \tau_2]).
\end{aligned}
$$

and prove $\forall\, D_1 \cdot P(D_1)$ by induction on the shape of derivation tree D_1.

Thus, you would need to prove $P(D_1)$ for some arbitrary derivation tree D_1, assuming as induction hypothesis that $\forall D' \sqsubset D_1 \cdot P(D')$ holds (where $\sqsubset$ is the "is a subderivation of" relation, as discussed in the exercise session).