

Semantics of Programming Lectures

Exercise Sheet 4

From this sheet onward, you should write all your (non-trivial) proofs in Isar!

Exercise 4.1 Rule Inversion

Recall the evenness predicate ev from the lecture:

```
inductive  $ev$  :: " $nat \Rightarrow bool$ " where  
   $ev0$ : " $ev\ 0$ " |  
   $evSS$ : " $ev\ n \Longrightarrow ev\ (Suc\ (Suc\ n))$ "
```

Prove the converse of rule $evSS$ using rule inversion. Hint: There are two ways to proceed. First, you can write a structured Isar-style proof using the *cases* method:

```
lemma " $ev\ (Suc\ (Suc\ n)) \Longrightarrow ev\ n$ "  
proof –  
  assume " $ev\ (Suc\ (Suc\ n))$ " then show " $ev\ n$ "  
  proof (cases)  
    ...  
  qed  
qed
```

Optional: Alternatively, you can write a more automated proof by using the **inductive_cases** command to generate elimination rules. These rules can then be used with "*auto elim:*". (If given the [*elim*] attribute, *auto* will use them by default.)

```
inductive_cases  $evSS\_elim$ : " $ev\ (Suc\ (Suc\ n))$ "
```

Next, prove that the natural number three ($Suc\ (Suc\ (Suc\ 0))$) is not even. Hint: You may proceed either with a structured proof, or with an automatic one. An automatic proof may require additional elimination rules from **inductive_cases**.

```
lemma " $\neg ev\ (Suc\ (Suc\ (Suc\ 0)))$ "
```

Exercise 4.2 (Deterministic) labeled transition systems

A *labeled transition system* is a directed graph with edge labels. We represent it by a predicate that holds for the edges.

type_synonym $(\text{'}q, \text{'}l)$ $lts = \text{"'}q \Rightarrow \text{'}l \Rightarrow \text{'}q \Rightarrow \text{bool}"$

I.e., for an LTS δ over nodes of type $\text{'}q$ and labels of type $\text{'}l$, $\delta p l q$ means that there is an edge from p to q labeled with l .

A word from source node u to target node v is the sequence of edge labels one encounters when going from u to v .

Define a predicate *word*, such that $\text{word } \delta u w v$ holds iff w is a word from u to v .

inductive $\text{word} :: \text{"('}q, \text{'}l) lts \Rightarrow \text{'}q \Rightarrow \text{'}l \text{ list} \Rightarrow \text{'}q \Rightarrow \text{bool}"$ **for** δ

A *deterministic* LTS has at most one transition for each node and label

definition $\text{"det } \delta \equiv \forall p l q1 q2. \delta p l q1 \wedge \delta p l q2 \longrightarrow q1 = q2"$

Show: For a deterministic LTS, the same word from the same source node leads to at most one target node, i.e., the target node is determined by the source node and the path

lemma

assumes $\text{det: "det } \delta"$

shows $\text{"word } \delta p ls q \Longrightarrow \text{word } \delta p ls q' \Longrightarrow q = q'"$

Exercise 4.3 Counting Elements

Recall the count function, that counts how often a specified element occurs in a list:

fun $\text{count} :: \text{"'}a \Rightarrow \text{'}a \text{ list} \Rightarrow \text{nat}"$ **where**

$\text{"count } x [] = 0"$

$| \text{"count } x (y \# ys) = (\text{if } x=y \text{ then } \text{Suc } (\text{count } x ys) \text{ else } \text{count } x ys)"$

Show that, if an element occurs in the list (its count is positive), the list can be split into a prefix not containing the element, the element itself, and a suffix containing the element one times less

lemma $\text{"count } a xs = \text{Suc } n \Longrightarrow \exists ps ss. xs = ps @ a \# ss \wedge \text{count } a ps = 0 \wedge \text{count } a ss = n"$

Homework 4.1 Product Construction for LTS

Submission until Sunday, November 21, 23:59pm.

The product construction is a standard construction for the intersection of two *lts*. Define the transition relation *prod* of the product of two given transition systems.

inductive $\text{prod} :: \text{"('}q_1, \text{'}l) lts \Rightarrow (\text{'}q_2, \text{'}l) lts \Rightarrow \text{'}q_1 \times \text{'}q_2 \Rightarrow \text{'}l \Rightarrow \text{'}q_1 \times \text{'}q_2 \Rightarrow \text{bool}"$ **for** $\delta_1 \delta_2$

Show that your product only contains those words.

Hint: Make sure to set up the induction properly. When you explicitly state the arguments in a computation or rule induction, it might be necessary to then declare some of the variables in the arguments as arbitrary.

Also make sure to chain in the proper facts into your induction with *using*

theorem *prod_sound*:

assumes “*word* (*prod* δ_1 δ_2) (p_1, p_2) *ls* (q_1, q_2)”
shows “*word* δ_1 p_1 *ls* q_1 \wedge *word* δ_2 p_2 *ls* q_2 ”

Now prove that your product accepts all words that occur in both *lts*.

Hint: You will need rule induction and rule inversion.

lemma *prod_complete*:

assumes “*word* δ_1 p_1 *ls* q_1 ”
and “*word* δ_2 p_2 *ls* q_2 ”
shows “*word* (*prod* δ_1 δ_2) (p_1, p_2) *ls* (q_1, q_2)”

Finally, the single correctness statement follows:

corollary “ $\{w. \text{word } (\text{prod } \delta_1 \delta_2) (p_1, p_2) w (q_1, q_2)\} = \{w. \text{word } \delta_1 p_1 w q_1\} \cap \{w. \text{word } \delta_2 p_2 w q_2\}$ ”

using *prod_sound prod_complete* **by** *fast*

Homework 4.2 Minimal Compiler

Submission until Sunday, November 21, 23:59pm.

Recall the *aexp* compiler with exceptions from the last tutorial.

We defined correctness of the compiled code as follows:

correct a ins $\equiv \forall s \text{ stk}. \text{exec ins } s \text{ stk} = \text{Some } (aval a s \# \text{stk})$

We are also interested in optimal code. Show that any correct list of instructions needs to contain a *LOAD* instruction for every variable of an *aexp*.

Hint: Do a proof by contradiction, without induction. You will need auxiliary lemmas about *aval* and *exec* with changed state (those need induction).

theorem *vars_in_ins*:

assumes “ $x \in \text{vars } a$ ”
shows “*correct a ins* $\implies \text{LOAD } x \in \text{set ins}$ ”