Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch, Kevin Kappelmann

WS 23/24
14.11.2023

Semantics of Programming Languages

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 ⇒ bool" where
  ev0: "ev 0" |
  evSS: "ev n ⟹ 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)) ⟹ ev n"
proof -
  assume "ev (Suc (Suc n))" then show "ev n"
  proof (cases)

    ...

  qed
qed
```

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 "¬ ev (Suc (Suc (Suc 0)))"
```

Exercise 4.2 (Deterministic) Labelled Transition Systems (LTS)

A *labelled transition system* is a directed graph with labelled edges. We model such systems as functions:

type_synonym $('q,'l)$ *lts* = *“'q ⇒ 'l ⇒ 'q ⇒ bool”*

For an LTS δ over nodes of type $'q$ and labels of type $'l$, $\delta\ p\ l\ q$ means that there is an edge from p to q labelled with l.

A word from source node u to target node v is the list of edge labels one encounters when going from u to v.
Define an inductive predicate *word*, such that *word* $\delta\ u\ w\ v$ holds iff w is a word from u to v.

inductive *word* :: *“('q,'l) lts ⇒ 'q ⇒ 'l list ⇒ 'q ⇒ bool”* **for** δ

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

definition *“det δ ≡ ∀p l q1 q2. δ p l q1 ∧ δ p l q2 ⟶ q1 = q2”*

Show that for a deterministic LTS, the same word from the same source node leads to at most one target node.

lemma
  assumes *det*: *“det δ”*
 shows *“word δ p ls q ⟹ word δ p ls q' ⟹ q = q'”*

Exercise 4.3 Counting Elements

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

fun *count* :: *“'a ⇒ 'a list ⇒ nat”* **where**
 “count x [] = 0”
| *“count x (y # ys) = (if x=y then Suc (count x ys) else 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
  assumes *“count a xs = Suc n”*
 shows *“∃ps ss. xs = ps @ a # ss ∧ count a ps = 0 ∧ count a ss = n”*

Homework 4.1 Deterministic Automata

Submission until Monday, November 20, 23:59pm.

We are not yet done with our nostalgic trip back to THEO. In this exercise, we explore deterministic automata (DAs), which are just like DFAs but allow for an infinite set of states.
We model DAs with four components, subject to some well-formedness conditions:

1. Q: the set of states
2. δ: the transition function, mapping a (state, letter) pair to a new state,
3. *q0*: the initial state, and
4. F: the set of final states.

The alphabet of a DA is implicitly given in the type of the transition function δ. Here is the datatype holding these four components:

datatype $('q,\ 'a)\ \mathit{DA} = \mathit{DA}\ ('q\ \mathit{set})\ ('q \Rightarrow 'a \Rightarrow 'q)\ 'q\ ('q\ \mathit{set})$

You can use the functions *states*, *transitions*, *initial*, and *finals* to access the components of a DA:

$\mathit{states}\ (\mathit{DA}\ q\ \delta\ q0\ F) = q$

$\mathit{transitions}\ (\mathit{DA}\ q\ \delta\ q0\ F) = \delta$

$\mathit{initial}\ (\mathit{DA}\ q\ \delta\ q0\ F) = q0$

$\mathit{finals}\ (\mathit{DA}\ q\ \delta\ q0\ F) = F$

We now define the expected well-formedness conditions for DAs.

$\mathit{wf\_da}\ M \equiv \mathit{initial}\ M \in \mathit{states}\ M \wedge \mathit{finals}\ M \subseteq \mathit{states}\ M \wedge (\forall\, q\ a.\ q \in \mathit{states}\ M \longrightarrow \mathit{transitions}\ M\ q\ a \in \mathit{states}\ M)$

The definitions of words and languages are the same as in prior week's homework:

type_synonym *'a word = "'a list"*
type_synonym *'a lang = "('a word) set"*

$\varepsilon \equiv []$

We next define the run of DAs inductively. The term $M\ q\ w \rightarrow n\ q'$ means the following: when reading w, the DA M in state q reaches the state q' in n steps.

Note: we allow for arbitrarily many ε-transitions.

$q \in \mathit{states}\ M \Longrightarrow M\ q\ \varepsilon \rightarrow n\ q$

$q \in \mathit{states}\ M \Longrightarrow M\ q\ w \rightarrow 0\ q$

$[\![q \in \mathit{states}\ M;\ M\ \mathit{transitions}\ M\ q\ a\ \mathit{as} \rightarrow n\ q']\!] \Longrightarrow M\ q\ a\ \#\ \mathit{as} \rightarrow \mathit{Suc}\ n\ q'$

We next define $M\ q\ w \rightarrow c\ q'$, meaning that automaton M reaches q' from state q when reading the word w completely.

$\mathit{run\_complete} \equiv \lambda M\ q\ w.\ \mathit{run}\ M\ q\ w\ (\mathit{length}\ w)$

The language of a DA is the set of all completely accepted words.

$\mathit{lang}\ M \equiv \{w.\ \exists\, \mathit{qf}.\ \mathit{qf} \in \mathit{finals}\ M \wedge M\ \mathit{initial}\ M\ w \rightarrow c\ \mathit{qf}\}$

Prove that DAs are indeed deterministic.

Hint: use **inductive_cases** to set up appropriate *elim* rules (see tutorial). The next few proofs should then be one-liners.

lemma *run_determ*: "$M\ q\ w \rightarrow n\ q' \Longrightarrow M\ q\ w \rightarrow n\ q'' \Longrightarrow q' = q''$"

Prove that DAs only visit valid states.

lemma *visit_valid_start*: *“M q w′ →n q′′ ⟹ q ∈ states M”*
lemma *visit_valid_end*: *“M q w →n q′ ⟹ q′ ∈ states M”*

Prove that runs of DAs can be merged and split.

lemma *run_append*: *“M q w →c q′ ⟹ M q′ w′ →c q′′ ⟹ M q (w @ w′) →c q′′”*
lemma *run_split*: *“M q (w @ w′) →c q′′ ⟹ ∃ q′. M q w →c q′ ∧ M q′ w′ →c q′′”*

Now define a recursive function *run_fun* such that *run_fun M q w n* computes the state reached by *M* in *n* steps when reading *w* from states *q*. Note: if there are no more letters to read, the DA should stay at its current state.

fun *run_fun* :: *“(′q, ′a) DA ⇒ ′q ⇒ ′a word ⇒ nat ⇒ ′q”*

Prove you that your function is correct.

lemma *run_complete_to_run_fun* :
  “M q w →c q′ ⟹ run_fun M q w (length w) = q′”
lemma *run_run_fun*:
  assumes *“wf_da M”*
    and *“q ∈ states M”*
  shows *“M q w →n (run_fun M q w n)”*

Let us now consider a simple example DA.

datatype *sigma = A | B*

testδ q A = 0

testδ q B = 1

testM ≡ DA {0, 1} testδ 0 {1}

Here are some example runs of the DA:

value *“testM (initial testM) [A,B,B,B] →c 1”*
value *“testM (initial testM) [A,B,A,A] →c 0”*

Partial execution:

value *“testM (initial testM) [A,B,B,B] →3 1”*

Out of bounds execution stays in same state:

value *“testM (initial testM) [A,B,B,B] →6 1”*

Automaton stays at current state in 0 steps:

value *“testM (initial testM) [A,B,B,B] →0 (initial testM)”*

Now prove that *testM* only accepts words ending with *B*.
Hints:

- the first direction does not need an induction, but the second one might.
- Write an Isar proof for both! Use the properties shown previously.

lemma *lang_testM_subseteq*: *"lang testM* $\subseteq \{w.\ \exists\, w'.\ w = w'\ @\ [B]\}$*"*

Bonus (2 pts): now prove the converse.

Bonus points are added to your final homework score. However, they do not raise the maximum number of achievable homework points.

lemma *subseteq_lang_testM*: *"*$\{w.\ \exists\, w'.\ w = w'\ @\ [B]\} \subseteq$ *lang testM"*

Finally:

corollary *lang_testM_eq*: *"lang testM* $= \{w.\ \exists\, w'.\ w = w'\ @\ [B]\}$*"*