Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Peter Lammich, Johannes Hölzl

WS 2013/14
5. 11. 2013

Semantics of Programming Languages

Exercise Sheet 4

Exercise 4.1 Reflexive Transitive Closure

A binary relation is expressed by a predicate of type $R :: 's \Rightarrow 's \Rightarrow bool$. Intuitively, R s t represents a single step from state s to state t.

The reflexive, transitive closure R^* of R is the relation that contains a step R^* s t, iff R can step from s to t in any number of steps (including zero steps).

Formalize the reflexive transitive closure as inductive predicate:

inductive *star* :: "$('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$"

When doing so, you have the choice to append or prepend a step. In any case, the following two lemmas should hold for your definition:

lemma *star_prepend*: "$[\![r\ x\ y;\ star\ r\ y\ z]\!] \Longrightarrow star\ r\ x\ z$"
lemma *star_append*: "$[\![\ star\ r\ x\ y;\ r\ y\ z\ ]\!] \Longrightarrow star\ r\ x\ z$"

Now, formalize the star predicate again, this time the other way round:

inductive *star'* :: "$('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$"

Prove the equivalence of your two formalizations

lemma "$star\ r\ x\ y = star'\ r\ x\ y$"

Hint: The induction method expects the assumption about the inductive predicate to be first.

Exercise 4.2 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
```

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)))"*

Homework 4.1 Elements of a List

Submission until Tuesday, November 12, 10:00am.

Give all your proofs in Isar, not apply style

Define a recursive function *elems* returning the set of elements of a list:

fun *elems* :: *"'a list ⇒ 'a set"*

To test your definition, prove:

lemma *"elems [1,2,3,(4::nat)] = {1,2,3,4}"*

Now prove for each element x in a list xs that we can split xs in a prefix not containing x, x itself and a rest:

lemma *"x ∈ elems xs ⟹ ∃ys zs. xs = ys @ x # zs ∧ x ∉ elems ys"*

Homework 4.2 Paths in Graphs

Submission until Tuesday, November 12, 10:00am.

Give all your proofs in Isar, not apply style

A graph is specified by a set of edges: E :: $('v \times 'v)$ *set*. A path in a graph from u to v is a list of vertices $[u_1,\ldots,u_n]$ such that $u{=}u_1$, $(u_i,u_{i+1}){\in}E$, and $(u_n,v){\in}E$. Moreover, the empty list is a path from any node to itself.

For example, in the graph: $\{(i, i+1) \mid i \in \mathbb{N}\}$, we have that [*3*,*4*,*5*] is a path from *3* to *6*, and [] is a path from *1* to *1*.

Note that not including the last node of the path into the list simplifies the formalization.

Formalize an inductive predicate *is_path*

inductive *is_path* :: "$('v \times 'v)$ *set* ⇒ *'v* ⇒ *'v list* ⇒ *'v* ⇒ *bool*"

Test your formalization for some examples:

lemma "*is_path* {(*i*,*i*+*1*) | *i*::*nat*. *True*} *3* [*3*,*4*,*5*] *6*"
lemma "*is_path* {(*i*,*i*+*1*) | *i*::*nat*. *True*} *1* [] *1*"

Prove the following two lemmas that allow you to glue together and split paths:

lemma *path_appendI*:
  assumes "*is_path E u p1 v*"
  assumes "*is_path E v p2 w*"
  shows "*is_path E u* (*p1*@*p2*) *w*"

Hint: For the next lemma, do an induction over *p1*, and, in the induction step, use rule-inversion on *is_path*.

lemma *path_appendE*:
  assumes "*is_path E u* (*p1*@*p2*) *w*"
  shows "∃ *v*. *is_path E u p1 v* ∧ *is_path E v p2 w*"