Exercise 4.1 Reflexive Transitive Closure

Theory Star (available on the course website) defines a binary relation star r, which is the reflexive, transitive closure of the binary relation r. It is defined inductively with the rules “star r x x” and “[ r x y; star r y z ] ⇒ star r x z”.

We also could have defined star the other way round, i.e., by appending steps rather than prepending steps:

\[
\text{inductive star'} :: \text{"(} a \Rightarrow a \Rightarrow \text{bool} \Rightarrow a \Rightarrow a \Rightarrow \text{bool} \text{\" for } r \text{ where}
\]
\[
\text{"star'} r x x \ \text{\" |}
\]
\[
\text{"[star'} r x y; r y z \text{\"] ⇒ star'} r x z \text{\"}
\]

Prove the following lemma. Hint: You will need an additional lemma for the induction.

\[
\text{lemma \"star r x y ⇒ star'} r x y \text{"}
\]

Exercise 4.2 Proving That Numbers Are Not Even

Recall the evenness predicate ev from the lecture:

\[
\text{inductive ev :: \"nat ⇒ bool\" where}
\]
\[
\text{ev0: \"ev 0\" |}
\]
\[
\text{evSS: \"ev n ⇒ ev (Suc (Suc n))\"}
\]

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:

\[
\text{lemma \"ev (Suc (Suc n)) ⇒ ev n\"}
\]
\[
\text{proof =}
\]
\[
\text{assume \"ev (Suc (Suc n))\" then show \"ev n\"}
\]
\[
\text{proof (cases)}
\]
\[
\text{...}
\]
\[
\text{qed}
\]
\[
\text{qed}
\]
Alternatively, you can write a more automated proof by using the \texttt{inductive\_cases} command to generate elimination rules. These rules can then be used with “\texttt{auto elim}”: (If given the \texttt{[elim]} attribute, \texttt{auto} will use them by default.)

\texttt{inductive\_cases evSS}\_elim: “\texttt{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 \texttt{inductive\_cases}.

\texttt{lemma “\neg ev (Suc (Suc (Suc 0)))”}

\textbf{Exercise 4.3 Binary Trees with the Same Shape}

Consider this datatype of binary trees:

\texttt{datatype tree = Leaf int | Node tree tree}

Define an inductive binary predicate \texttt{sameshape :: tree \Rightarrow tree \Rightarrow bool}, where \texttt{sameshape t\_1 t\_2} means that \texttt{t\_1} and \texttt{t\_2} have exactly the same overall size and shape. (The elements in the corresponding leaves may be different.)

\texttt{inductive sameshape :: “tree \Rightarrow tree \Rightarrow bool” where}

Now prove that the \texttt{sameshape} relation is transitive.

\texttt{theorem “[sameshape t\_1 t\_2; sameshape t\_2 t\_3] \Rightarrow sameshape t\_1 t\_3”}

Hint: For this proof, we recommend doing an induction over \texttt{t\_1} and \texttt{t\_2} using rule \texttt{same-\_shape.induct}. You will also need some elimination rules from \texttt{inductive\_cases}. (Look at the subgoals after induction to see which patterns to use.) Finally, note that “\texttt{auto elim}” applies rules tentatively with a limited search depth, and may not find a proof even if you have all the rules you need. You can either try the variant “\texttt{auto elim}!”: which applies rules more eagerly, or try another method like \texttt{blast} or \texttt{force}.

\textbf{Homework 4 IMP with Exceptions}

\textit{Submission until Wednesday, November 23, 12:00 (noon).} In this exercise, you shall add exceptions to the IMP-language. Hint: A good approach is to start by copying the definitions from the original theories, and then modify them. (Please include comments that make it clear exactly which parts you have changed.)

First, extend the command \texttt{datatype} with try-catch blocks and a throw command. There is only one exception type, i.e., the throw command has no further parameters.

\texttt{datatype com}
\begin{verbatim}
  = SKIP
  | Assign vname aexp (\texttt{“\_:= \_” [1000, 61] 61})
  | Semi com com (\texttt{“\_;/ \_” [60, 61] 60})
\end{verbatim}
Define a big-step semantics for this extended language. The proposition \((c, s) \Rightarrow r\)
means that in initial state \(s\), program \(c\) evaluates to the final result \(r\). Due to the
presence of exceptions, the result \(r\) cannot simply have type \(\text{state}\); instead we must use
this extended result type:

\[
\text{datatype result} = \text{Normal state} \mid \text{Exception state}
\]

\[
\text{inductive big_step :: } \text{com} 	imes \text{state} \Rightarrow \text{result} \Rightarrow \text{bool} \quad \text{(infix “⇒” 55)}
\]

where

Next, define a predicate \(\text{nothrow} :: \text{com} \Rightarrow \text{bool}\), where \(\text{nothrow} c\) means that \(c\) contains
no \(\text{THROW}\) statements that are not surrounded by an enclosing \(\text{TRY}\). You may define
it using either \text{fun} or \text{inductive}, as you wish. (Note that your choice may have a big
effect on later proofs!)

Finally, show that a program that does not contain throw-statements outside try-catch
blocks will never return an exception state.

\[
\text{fun is_normal :: } \text{result} \Rightarrow \text{bool} \quad \text{where}
\]

\[
\text{“is_normal} (\text{Normal } s) \iff \text{True} \mid \text{is_normal} (\text{Exception } s) \iff \text{False} \]

\[
\text{theorem “[nothrow } c; (c, s) \Rightarrow r] \Rightarrow \text{is_normal } r”}
\]

\textbf{Note 1}: When doing induction over an inductive predicate, the assumption containing
that predicate must appear \textit{first} in the list of assumptions. If you need to re-order the
assumptions, you can either re-state the theorem, or else use one of these patterns:

\[
\text{theorem “[nothrow } c; (c, s) \Rightarrow r] \Rightarrow \text{is_normal } r”}
\]

\textbf{proof –}

\textbf{assume “(c, s) ⇒ r” and “nothrow c” then show “is_normal r”}

\textbf{apply (induction)}

\textbf{theorem assumes 1: “nothrow c” and 2: “(c, s) ⇒ r” shows “is_normal r”}

\textbf{using 2 I apply (induction)}

\textbf{Note 2}: The default induction rule for \text{big_step} only allows induction over two variables,
using an assumption of the form \(x \Rightarrow r\). To get an induction rule that works with the
three-variable form \(c, s \Rightarrow r\), use the following command:

\[
\text{lemmas big_step_induct = big_step.induct[split_format(complete)]}
\]