Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Brian Huffman, Peter Lammich

WS 2011/12
11. 11. 2011

Semantics of Programming Languages

Exercise Sheet 4

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 "$[\![r\ x\ y;\ star\ r\ y\ z]\!] \Longrightarrow star\ r\ x\ z$".

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

inductive *star'* :: "$('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$" **for** *r* **where**
  "*star' r x x*" |
  "$[\![star'\ r\ x\ y;\ r\ y\ z]\!] \Longrightarrow star'\ r\ x\ z$"

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

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

Exercise 4.2 Proving That Numbers Are Not Even

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)))"*

Exercise 4.3 Binary Trees with the Same Shape

Consider this datatype of binary trees:

datatype *tree = Leaf int | Node tree tree*

Define an inductive binary predicate *sameshape* :: *tree* ⇒ *tree* ⇒ *bool*, where *sameshape* t_1 t_2 means that t_1 and t_2 have exactly the same overall size and shape. (The elements in the corresponding leaves may be different.)

inductive *sameshape* :: *"tree ⇒ tree ⇒ bool"* **where**

Now prove that the *sameshape* relation is transitive.

theorem "$[\![$*sameshape* t_1 t_2; *sameshape* t_2 $t_3]\!] \Longrightarrow$ *sameshape* t_1 t_3"

Hint: For this proof, we recommend doing an induction over t_1 and t_2 using rule *sameshape.induct*. You will also need some elimination rules from **inductive_cases**. (Look at the subgoals after induction to see which patterns to use.) Finally, note that "*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 "*auto elim*!:", which applies rules more eagerly, or try another method like *blast* or *force*.

Homework 4 IMP with Exceptions

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

```
datatype com
  = SKIP
  | Assign vname aexp      ("_ ::= _" [1000, 61] 61)
  | Semi com  com          ("_;/ _"  [60, 61] 60)
  | If bexp com com        (“(IF _/ THEN _/ ELSE _)”  [0, 0, 61] 61)
  | While bexp com         (“(WHILE _/ DO _)”  [0, 61] 61)
  | TryCatch com com         (“(TRY _/ CATCH _)” [0,61] 61)
  | Throw                  (“THROW”)
```

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 *state*; instead we must use this extended result type:

```
datatype result = Normal state | Exception state

inductive big_step :: “com × state ⇒ result ⇒ bool” (infix “⇒” 55)
where
```

Next, define a predicate *nothrow* :: *com* ⇒ *bool*, where *nothrow c* means that *c* contains no *THROW* statements that are not surrounded by an enclosing *TRY*. You may define it using either **fun** or **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.

```
fun is_normal :: “result ⇒ bool” where
  “is_normal (Normal s) ⟷ True” |
  “is_normal (Exception s) ⟷ False”

theorem “⟦nothrow c; (c, s) ⇒ r⟧ ⟹ is_normal r”
```

Note 1: When doing induction over an inductive predicate, the assumption containing that predicate must appear *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:

```
theorem “⟦nothrow c; (c, s) ⇒ r⟧ ⟹ is_normal r”
proof –
  assume “(c, s) ⇒ r” and “nothrow c” then show “is_normal r”
  apply (induction

theorem assumes 1: “nothrow c” and 2: “(c, s) ⇒ r” shows “is_normal r”
using 2 1 apply (induction
```

Note 2: The default induction rule for *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:

```
lemmas big_step_induct = big_step.induct[split_format(complete)]
```