Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Hupel

WS 2017/18
07.11.2017

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

Exercise 4.2 Avoiding Stack Underflow

A *stack underflow* occurs when executing an instruction on a stack containing too few values – e.g., executing an *ADD* instruction on an stack of size less than two. A well-formed sequence of instructions (e.g., one generated by *comp*) should never cause a stack underflow.

In this exercise, you will define a semantics for the stack-machine that throws an exception if the program underflows the stack.

Modify the *exec1* and *exec* - functions, such that they return an option value, *None* indicating a stack-underflow.

fun *exec1* :: *"instr ⇒ state ⇒ stack ⇒ stack option"*
fun *exec* :: *"instr list ⇒ state ⇒ stack ⇒ stack option"*

Now adjust the proof of theorem *exec_comp* to show that programs output by the compiler never underflow the stack:

theorem *exec_comp*: *"exec (comp a) s stk = Some (aval a s # stk)"*

Exercise 4.3 A Structured Proof on Relations

We consider two binary predicates T and A and assume that T is total, A is antisymmetric and T is a subset of A. Show with a structured, Isar-style proof that then A is also a subset of T:

assumes *"∀ x y. T x y ∨ T y x"*
  and *"∀ x y. A x y ∧ A y x ⟶ x = y"*
  and *"∀ x y. T x y ⟶ A x y"*
shows *"A x y ⟶ T x y"*

General homework instructions

All proofs in the homework must be carried out in Isar style.

Remember that induction proofs start as follows in Isar:

proof (*induction x arbitrary*: *y*)

You will then see a clickable suggestion in the Output panel that will insert the proof template into the theory.

Similarly, a proof by case distinction starts as follows:

proof (*cases x*)

Again, there will be a clickable suggestion in the Output panel.

Homework 4.1 Relational Addition

Submission until Tuesday, November 21, 10:00am.

Define an inductive predicate *plus* of type *nat ⇒ nat ⇒ nat ⇒ bool* such that the following properties hold.

inductive *plus* :: *"nat ⇒ nat ⇒ nat ⇒ bool"* **where**
lemma *plus_zero*: *"plus 0 m m"*
lemma *zero_plus*: *"plus m 0 m"*
lemma *plus_comm*: *"plus m n k ⟹ plus n m k"*

Also prove that *plus* is equivalent to *op +* on natural numbers. You are not allowed to use the proof below for the properties above.

Hint: Consider proving both directions separately.

lemma *"plus m n k ⟷ m + n = k"*

Homework 4.2 Avoiding Stack Underflow (II)

Submission until Tuesday, November 21, 10:00am.

In the tutorial, we have defined a modified version of the *exec* function that returns *None* if the stack is not large enough. However, this function actually *executes* the instructions. Sometimes, we cannot pay this cost: Here, we want to detect this situation *statically*. Define a function *can_execute* that, given an initial stack size and a list of instructions, returns a *bool* indicating whether a stack underflow will occur.

fun *can_execute* :: *"nat ⇒ instr list ⇒ bool"*

Prove that the function correctly analyzes stack underflow behaviour.

lemma *"exec is s stk = Some stk' ⟹ can_execute (length stk) is"*

(Strictly speaking, this is only one of the two necessary directions for correctness. For the purpose of the homework, the above direction is sufficient. The other direction is more difficult and will earn you three bonus points if solved.)

Homework 4.3 Avoiding Stack Underflow (III)

Submission until Tuesday, November 21, 10:00am.

Define a relational version of *exec1* and *exec*. Leave the cases in which the stack would underflow undefined.

inductive *exec1r* :: *"instr ⇒ state ⇒ stack ⇒ stack ⇒ bool"*
inductive *execr* :: *"instr list ⇒ state ⇒ stack ⇒ stack ⇒ bool"*

Prove equivalence.

lemma *exec1r_equiv1*: *"exec1r i s stk stk' ⟹ exec1 i s stk = Some stk'"*
lemma *exec1_equiv1*: *"execr is s stk stk' ⟹ exec is s stk = Some stk'"*
lemma *exec1r_equiv2*: *"exec1 i s stk = Some stk' ⟹ exec1r i s stk stk'"*
lemma *execr_equiv2*: *"exec is s stk = Some stk' ⟹ execr is s stk stk'"*