Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2020/21
16.11.2020

Semantics of Programming Languages

Exercise Sheet 3

Exercise 3.1 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* :: "$('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$" **for** *r*

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 (append if you prepended the step before or vice versa):

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

Prove the equivalence of your two formalizations:

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

Exercise 3.2 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 3.3 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 (without proof methods more powerful than simp!):

lemma
  assumes *total*: "$\forall\ x\ y.\ T\ x\ y \vee T\ y\ x$"
    and *anti*: "$\forall\ x\ y.\ A\ x\ y \wedge A\ y\ x \longrightarrow x = y$"
    and *subset*: "$\forall\ x\ y.\ T\ x\ y \longrightarrow A\ x\ y$"
  shows "$A\ x\ y \longrightarrow T\ x\ y$"

Homework 3.1 Avoiding Stack Underflow (II)

Submission until Sunday, Nov 22, 23:59.

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.

theorem *can_exec_correct*:
  "can_execute (length stk) ins ⟹ exec ins s stk ≠ None"
theorem *can_exec_complete*:
  "exec ins s stk = Some res ⟹ can_execute (length stk) ins"

Homework 3.2 Avoiding Stack Underflow (III)

Submission until Sunday, Nov 22, 23:59.

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.

theorem *step_equiv*: "*exec1r i s stk stk'* ⟷ *exec1 i s stk* = *Some stk'*"
theorem *exec_equiv*: "*execr ins s stk stk'* ⟷ (*exec ins s stk* = *Some stk'*)"

Homework 3.3 Negation Normal Form

Submission until Sunday, Nov 22, 23:59.

In this assignment, you shall write a function that converts a boolean expression over variables, conjunction, disjunction, and negation to negation normal form (NNF), and prove its correctness.

We start by defining our version of boolean expressions:

datatype *bexp* = *Var vname* | *Not bexp* | *And bexp bexp* | *Or bexp bexp*

type_synonym *state* = "*vname* ⇒ *bool*"

fun *is_var* :: "*bexp* ⇒ *bool*" **where**
  "*is_var* (*Var* _) = *True*"
| "*is_var* _ = *False*"

fun *bval* :: "*bexp* ⇒ *state* ⇒ *bool*"

Next, we define a predicate that checks whether a boolean expression is in NNF. In NNF, only variables may be negated.

inductive *is_nnf* :: "*bexp* ⇒ *bool*"

Now we want to show that the above definition is equivalent to a non-inductive definition of NNF. We define a *sub* function first that extracts all sub-expressions from a *bexp*.

fun *sub* :: "*bexp* ⇒ *bexp set*"
value "*sub* (*And* (*Not* (*Var* ''x'')) (*Var* ''y'')) =
 {
  Var ''x'',
  Var ''y'',
  Not (*Var* (''x'')),
  And (*Not* (*Var* ''x'')) (*Var* ''y'')
}"

theorem *nnf_not*: "*is_nnf b* = (∀ *b'*. *Not b'* ∈ *sub b* ⟶ *is_var b'*)"

Now define a function *nnf* which converts any boolean expression to NNF. This can be achieved by "pushing in" negations and eliminating double negations.

fun *nnf* :: "*bexp* ⇒ *bexp*"

Prove that your function is correct.

theorem *nnf_sound*: "*is_nnf (nnf b)*"
theorem *nnf_compl*: "*bval (nnf b) s = bval b s*"