Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Simon Wimmer

WS 2019/20
04.11.2019

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

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 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:

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"*

Homework 3.1 A Simple Grammar

Submission until Monday, November 11, 2019, 11:11am.
You are given the following grammar:

$$S \to \varepsilon \mid aSb$$

Your first task is to formalize this grammar as an inductive definition in Isabelle:

abbreviation *a* **where** *"a ≡ CHR ''a''"*
abbreviation *b* **where** *"b ≡ CHR ''b''"*

inductive_set *G* :: *"string set"*

Our goal is to show that G produces the following language:

definition
  "L = {w. ∃ n. w = replicate n a @ replicate n b}"

First prove this direction:

theorem *G_is_replicate*:
  assumes *"w ∈ G"*
  shows *"∃ n. w = replicate n a @ replicate n b"*

And now the converse:

theorem *replicate_G*:
  assumes *"w = replicate n a @ replicate n b"*
  shows *"w ∈ G"*

Finally, we can prove that G indeed produces L:

corollary *L_eq_G*:
"L = G"
unfolding *L_def* **using** *G_is_replicate replicate_G* **by** *auto*

Homework 3.2 Register Machine from Hell

Submission until Monday, November 11, 2019, 11:11am.

Processors from Hell has released its next-generation RISC processor. It features an infinite bank of registers R_0, R_1, etc, holding unbounded integers. Register R_0 plays the role of the accumulator and is the implicit source or destination register of all instructions. Any other register involved in an instruction must be distinct from R_0. To enforce this requirement the processor implicitly increments the index of the other register. There are 4 instructions:

`LDI` i has the effect $R_0 := i$

`LD` n has the effect $R_0 := R_{n+1}$

`ST` n has the effect $R_{n+1} := R_0$

`ADD` n has the effect $R_0 := R_0 + R_{n+1}$

where i is an integer and n a natural number.
The instructions are specified by:

datatype *instr = LDI int | LD nat | ST nat | ADD nat*

The state of the machine is just a function from register numbers to values

type_synonym *rstate = "nat ⇒ int"*

Define a function to execute a single instruction

fun *exec* :: *"instr ⇒ rstate ⇒ rstate"* **where**

Lift your definition to lists of instructions

fun *execs* :: *"instr list ⇒ rstate ⇒ rstate"* **where**

Show that *execs* commutes with *op* @. Hint: The [*simp*] - attribute declares this as a default simplifier rule, such that simp and auto will rewrite with this rule by default.

theorem *execs_append*[*simp*]: *"⋀s. execs (xs @ ys) s = execs ys (execs xs s)"*

Next, we want to write a compiler for arithmetic expressions. To simplify the mapping from variables to registers, we define variable names to be natural numbers.

datatype *expr = C int | V nat | A expr expr*

fun *val* :: *"expr ⇒ (nat ⇒ int) ⇒ int"* **where**

"val(C i) s = i" |
"val(V n) s = s n" |
"val(A e1 e2) s = val e1 s + val e2 s"

You have been recruited to write a compiler from *expr* to *instr list*. You remember your compiler course and decide to emulate a stack machine using free registers, i.e. registers not used by the expression you are compiling. The type of your compiler is

fun *cmp* :: *"expr ⇒ nat ⇒ instr list"* **where**

where the second argument is the index of the first free register and can be used to store intermediate results. The result of an expression should be returned in R_0. Because R_0 is the accumulator, you decide on the following compilation scheme: Variable i will be held in R_{i+1}.

To actually compile an expression, you need to find an initial value for the free register index. Define a function that returns the maximum variable used in an arithmetic expression.

fun *maxvar* :: *"expr ⇒ nat"* **where**

Show that the value of expressions does not depend on variables greater than *maxvar*.

theorem *val_maxvar_same*[*simp*]:
"ALL n <= maxvar e. s n = s' n ⟹ val e s = val e s'"

Finally, prove that your compiler is correct. You will need to generalize the lemma to any free register index > *maxvar e*.

Moreover, an auxiliary lemma may be useful, which states that a compiled program does not change registers less than the index of the first free register.

Hint: Beware of off-by-one errors introduced by the implicit increment of the register index. The register indexes in the state are shifted by one wrt. the registers in the instructions!

theorem *compiler_correct*: *"execs (cmp e (maxvar e + 1)) s 0 = val e (s o Suc)"*