Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Kevin Kappelmann, Lukas Stevens

WS 24/25
31.10.2024

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 relations T and A and assume that T is total, A is antisymmetric and T is finer than A, i.e., $T\ x\ y$ implies $A\ x\ y$ for all x, y. Show with a structured, Isar-style proof that then A finer than T (without proof methods more powerful than simp!):

lemma
  assumes *total*: "∀ *x y*. *T x y* ∨ *T y x*"
    and *anti*: "∀ *x y*. *A x y* ∧ *A y x* ⟶ *x* = *y*"
    and *subset*: "∀ *x y*. *T x y* ⟶ *A x y*"
  shows "*A x y* ⟶ *T x y*"

Homework 3.1 Grammars for Parenthesis Languages

Submission until Wednesday, November 6, 23:59pm.

In this homework, we will use inductive predicates to specify grammars for languages consisting of words of opening and closing parentheses. We model parentheses as follows:

datatype *paren* = *Open* | *Close*

We define the language of words with balanced parentheses:

$$S \longrightarrow \varepsilon \mid SS \mid (S)$$

as an inductive predicate with the following cases:

S []
⟦*S xs*; *S ys*⟧ ⟹ *S* (*xs* @ *ys*)
S xs ⟹ *S* (*Open* # *xs* @ [*Close*])

Show that words of the language contain the same amount of opening and closing parentheses:

theorem *S_count*: *"S xs ⟹ count xs Open = count xs Close"*

Now consider the language that is defined by the following variation of the grammar:

$$T \longrightarrow \varepsilon \mid TT \mid (T) \mid (T$$

inductive T :: *"paren list ⇒ bool"*

- Define T as a inductive predicate in Isabelle (the example should be easily provable by your introduction rules)
- Show that the language produced by T is at least as large as the one produced by S:

lemma *example*: *"T [Open, Open]"*

theorem *S_T*: *"S xs ⟹ T xs"*

Show that the converse also holds under the condition that the word contains the same amount of opening and closing parentheses:

theorem *T_S*: *"T xs ⟹ count xs Open = count xs Close ⟹ S xs"*

This reuses the *count* function known from sheet 1. *Hint:* You will need a lemma connecting the number of opening and closing parentheses in words produced by T.

Homework 3.2 Compilation to Register Machine

Submission until Wednesday, November 6, 23:59pm.

In this exercise, you will define a compilation function from arithmetic expressions to register machines and prove that the compilation is correct.

The registers in our simple register machines are natural numbers. These are the available instructions:

datatype *instr = LD reg vname | ADD reg op op*

LD loads a variable value in a register. *ADD* adds the contents of the two operands, placing the result in the register.

An operand is either a register or a constant:

datatype *op = REG reg | VAL val*

Recall that a variable state is a function from variable names to integers. Our machine state *mstate* contains both, variables and registers. For technical reasons, we encode it into a single function *v_or_reg ⇒ int*:

datatype *v_or_reg = Var vname | Reg reg*

Note: To access a variable value, we can write σ (*Var x*), to access a register, we can write σ (*Reg x*).

To extract the variable state from a machine state σ, we can use $\sigma \circ$ *Var*, where *o* is function composition.

Complete the following definition of the function for executing instructions on a machine state σ.

fun *op_val* :: *"op ⇒ mstate ⇒ int"*
fun *exec1* :: *"instr ⇒ mstate ⇒ mstate"*
fun *exec* :: *"instr list ⇒ mstate ⇒ mstate"*

We are finally ready for the compilation function. Your task is to define a function *cmp* that takes an arithmetic expression *a* and a register *r* and produces a list of register-machine instructions leading to this value.

fun *cmp* :: *"aexp ⇒ reg ⇒ instr list"*

Your program should need no more *ADD* instructions than there are *Plus* operations in the program, except if the expression is a single *N*.

Prove that property!

theorem *cmp_len*: *"¬is_N a ⟹ num_add (cmp a r) ≤ num_plus a"*

Finally, you need to prove the following correctness theorem, which states that our register-machine compiler is correct, in that executing the compiled instructions of an arithmetic expression yields (as the operand) the same result as evaluating the expression.

Hint: For proving correctness, you will need auxiliary lemmas, including that the instructions produced by *cmp a r* do not alter registers below *r*.

Moreover, the following lemma, which states that updating a register does not affect the variables, may be useful:

lemma *reg_var*[*simp*]: *"s (Reg r := x) o Var = s o Var"*
  by *auto*

theorem *cmp_correct*: *"exec (cmp a r) σ (Reg r) = aval a (σ o Var)"*