Technische Universität München
Fakultät für Informatik
Dr. Peter Lammich
Simon Wimmer

WS 2018/19
30.10.2018

Semantics of Programming Languages

Exercise Sheet 3

Exercise 3.1 Reflexive Transitive Closure

A binary relation is expressed by a predicate of type $R :: {'s} \Rightarrow {'s} \Rightarrow \mathit{bool}$. 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* :: "$({'a} \Rightarrow {'a} \Rightarrow \mathit{bool}) \Rightarrow {'a} \Rightarrow {'a} \Rightarrow \mathit{bool}$"

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*: "$[\![r\ x\ y;\ \mathit{star}\ r\ y\ z]\!] \Longrightarrow \mathit{star}\ r\ x\ z$"
lemma *star_append*: "$[\![\ \mathit{star}\ r\ x\ y;\ r\ y\ z\ ]\!] \Longrightarrow \mathit{star}\ r\ x\ z$"

Now, formalize the star predicate again, this time the other way round:

inductive *star'* :: "$({'a} \Rightarrow {'a} \Rightarrow \mathit{bool}) \Rightarrow {'a} \Rightarrow {'a} \Rightarrow \mathit{bool}$"

Prove the equivalence of your two formalizations:

lemma "$\mathit{star}\ r\ x\ y = \mathit{star}'\ r\ x\ y$"

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)"*

Homework 3.1 Grammars for Parenthesis Languages

Submission until Tuesday, November 3, 10:00am.

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:

inductive *S* **where**
S_empty: *"S []" |*
S_append: *"S xs ⟹ S ys ⟹ S (xs @ ys)" |*
S_paren: *"S xs ⟹ S (Open # xs @ [Close])"*

Consider the language that is defined by the following variation of the grammar:

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

- Define T as a inductive predicate in Isabelle
- Show that the language produced by T is at least as large as the one produced by S:

lemma *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:

lemma *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 Tuesday, November 3, 10:00am.

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:

type_synonym *reg* = *nat*

These are the available instructions:

datatype *op* = *REG reg* | *VAL int* — An operand is either a register or a constant.

datatype *instr* =
  LD reg vname — Load a variable value in a register.
| *ADD reg op op* — Add the contents of the two operands, placing the result in the register.

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

datatype *v_or_reg* = *Var vname* | *Reg reg*
type_synonym *mstate* = *"v_or_reg ⇒ int"*

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 an instruction on a machine state σ.

fun *op_val* :: *"op ⇒ mstate ⇒ int"* **where**
  "op_val (REG r) σ = σ (Reg r)" |
  "op_val (VAL n) _ = n"

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

Next define the function executing a sequence of register-machine instructions, one at a time. We have already defined for you the case of empty list of instructions. You need to add the recursive case.

fun *execs* :: *"instr list ⇒ mstate ⇒ mstate"* **where**
  "execs [] σ = σ" |
— Add recursive case here

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 pair of:

- an operand representing the value of evaluating a either as a constant or as a value in a register,

- and a list of register-machine instructions leading to this value.

Your program should need no more *ADD* instructions than there are *Plus* operations in the program.
Here is the intended behavior of *cmp*:

- *cmp* $(N\ n)\ r$ simply represents the computation result as a constant
- *cmp* $(V\ x)\ r$ loads x into r with the computation result in r
- *cmp* $(Plus\ a\ a1)\ r$ first compiles a placing the result in r, then compiles $a1$ placing the result in $r + 1$, and finally adds the content of $r + 1$ to that of r (storing the result in r).

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

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 stating

- that *execs* commutes with list concatenation,
- that the instructions produced by *cmp a r* do not alter registers below r,
- and that it if *cmp a r* places the result in a register, then this register is r.

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

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

theorem *cmp_correct*: *"cmp a r = (x, prog) ⟹*
  op_val x (execs prog σ) = aval a (σ o Var) ∧ execs prog σ o Var = σ o Var"
— The first conjunct states that the resulting operand contains the correct value, and the second conjunct states that the variable state is unchanged.

Homework 3.3 Splitting Lists

Submission until Tuesday, November 6, 10:00am.

In this exercise we consider the task of determining whether a list can be split in two parts such that the split fulfills a given property P:

definition
"$\mathit{ex\_split}\ P\ xs \longleftrightarrow (\exists\, ys\ zs.\ xs = ys\ @\ zs \wedge P\ ys\ zs)$"

Define a function *has_split* such that the following holds:

lemma *ex_split_has_split*[*code*]:
"$\mathit{ex\_split}\ P\ xs \longleftrightarrow \mathit{has\_split}\ P\ []\ xs$"

The function *has_split* should give us an executable version of *ex_split*. This means it should be defined like a regular functional program with recursion over lists. In particular no quantifiers should appear in its definition. Prove *ex_split_has_split*!

We now want to apply this function to determine whether a given list of integers can be split into two non-trivial parts that have the same sum:

definition
"$\mathit{ex\_balanced\_sum}\ xs = (\exists\, ys\ zs.\ \mathit{sum\_list}\ ys = \mathit{sum\_list}\ zs \wedge xs = ys\ @\ zs \wedge ys \neq [] \wedge zs \neq [])$"

Characterize *ex_balanced_sum* with *ex_split* for some suitable P:
$\mathit{ex\_balanced\_sum}\ xs \longleftrightarrow \mathit{ex\_split}\ P\ xs$

If you mark this theorem with [*code*] attribute, you should be able to execute *ex_balanced_sum*:

value "$\mathit{ex\_balanced\_sum}\ [1,2,3,3,2,1::nat]$"

Homework 3.4 (Bonus) Efficient List Split

Submission until Tuesday, November 6, 10:00am.

Note: This is a bonus exercise.

We again consider the task from the last exercise of determining whether a list of integers can split in two parts that have the same sum. The solution from the last exercise does this rather inefficiently. This time we want to do it in $\mathcal{O}(\mathit{length}\ xs)$.

Define a function *linear_split* with the required runtime complexity and prove:

lemma *linear_correct*:
"$\mathit{linear\_split}\ xs \longleftrightarrow \mathit{ex\_balanced\_sum}\ xs$"