Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Dr. Peter Lammich

WS 2015/16
27. 10. 2015

Semantics of Programming Languages

Exercise Sheet 3

Exercise 3.1 Relational *aval*

Theory *AExp* defines an evaluation function *aval* :: *aexp* ⇒ *state* ⇒ *val* for arithmetic expressions. Define a corresponding evaluation relation *is_aval* :: *aexp* ⇒ *state* ⇒ *val* ⇒ *bool* as an inductive predicate:

inductive *is_aval* :: *"aexp ⇒ state ⇒ val ⇒ bool"*

Use the introduction rules *is_aval.intros* to prove this example lemma.

lemma *"is_aval (Plus (N 2) (Plus (V x) (N 3))) s (2 + (s x + 3))"*

Prove that the evaluation relation *is_aval* agrees with the evaluation function *aval*. Show implications in both directions, and then prove the if-and-only-if form.

lemma *aval1*: *"is_aval a s v ⟹ aval a s = v"*
lemma *aval2*: *"aval a s = v ⟹ is_aval a s v"*
theorem *"is_aval a s v ⟷ aval a s = v"*

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 Avoiding Stack Overflow

Now, modify the semantics such that *None* is returned if the stack gets longer than a fixed size *maxsize*.

Define a function that, for a given expression, returns a suitable stack size, and show that the stack does not overflow.

context
  fixes *maxsize* :: *nat*
begin

The *context* construct allows you to locally fix some value. Once you close the context, this value becomes a parameter of everything defined inside the context.

Work with the following operation, which ensures that the stack does not overflow:

definition *push* :: *"val ⇒ stack ⇒ stack option"* **where**
  "push i stk ≡ if length stk < maxsize then Some (i#stk) else None"

fun *exec′1* :: *"instr ⇒ state ⇒ stack ⇒ stack option"*

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

end — End of context

Function to return the minimum required stack size for a given expression

fun *stacksize* :: *"aexp ⇒ nat"*

Prove the correctness lemma: Hint: For the induction to go through, you need to generalize the lemma over the stack!

theorem *exec_comp′*: *"stacksize a ≤ maxsize*
  ⟹ exec′ maxsize (comp a) s [] = Some ([aval a s])"

Homework 3.1 Compilation to Register Machine

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

In this exercise, you will define a compilation function from expressions to register machines and prove that the compilation is correct. Recall the arithmetic expressions with side effects from Tutorial 2:

type_synonym *vname = string*
type_synonym *val = int*
type_synonym *state = "vname ⇒ val"*

datatype *aexp = N int | V vname | Plus aexp aexp | Incr vname*

fun *aval* :: *"aexp ⇒ state ⇒ val × state"* **where**

"*aval* $(N\ n)\ s = (n{,}s)$"
| "*aval* $(V\ x)\ s = (s\ x{,}s)$"
| "*aval* $(Plus\ a_1\ a_2)\ s = (let$
  $(v_1{,}s) = aval\ a_1\ s;$
  $(v_2{,}s) = aval\ a_2\ s$
 $in\ (v_1{+}v_2{,}s))$"
| "*aval* $(Incr\ x)\ s = (s\ x,\ s(x{:=}s\ x + 1))$"

The registers in our simple register machines are natural numbers:

type_synonym *reg* = *nat*

The instructions are:

datatype *instr* =
 LDI int reg — Load an integer constant in a register (Load Immediate).
| *LD vname reg* — Load a variable value in a register.
| *ST reg vname* — Store a register's content to a variable.
| *ADD reg reg* — Add the contents of two registers, replacing the first one with the result.

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 $\sigma\ (Var\ x)$, to access a register, we can write $\sigma\ (Reg\ x)$.
To extract the variable state from a machine state σ, we can use $\sigma \circ Var$, where *op* $\circ$ is function composition.

Complete the following definition of the function for executing an instruction on a machine state σ. You need to add the cases of the instruction being "load immediate", "load", and "store".

fun *exec* :: "*instr* ⇒ *mstate* ⇒ *mstate*" **where**
— Add your cases here
"*exec* $(ADD\ r1\ r2)\ \sigma = \sigma\ (Reg\ r1 := \sigma\ (Reg\ r1) + \sigma\ (Reg\ r2))$"

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* [] $\sigma = \sigma$" |
— 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 list of register-machine instructions whose execution in any machine state should lead to a machine state having in r the value of evaluating a, and the variable state modified according to the side-effects in a.

Here is the intended behavior of *cmp*:

- *cmp* (*N n*) *r* loads immediate *n* into *r*
- *cmp* (*V x*) *r* loads *x* into *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*).
- *cmp* (*Incr x*) *r* load *x* into *r*, increment *r*, store *r* to *x*, decrement *r*. Note: Figure out how to encode increment and decrement with the available instructions. If you need a free register, use *r+1*.

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

Finally, you need to prove the following correctness lemma, which states that our register-machine compiler is correct, in that executing the compiled instructions of an arithmetic expression yields (in the indicated register) the same result as evaluating the expression, and the variables are modified correctly.

Hint: For proving correctness, you will need auxiliary lemmas stating that exec commutes with list concatenation and 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 [*simp*]: *"s (Reg r := x) o Var = s o Var"* **by** *auto*

lemma *cmpCorrect*: *"*
execs (*cmp a r*) σ (*Reg r*) = *fst* (*aval a* (σ *o Var*))
∧ *execs* (*cmp a r*) σ *o Var* = *snd* (*aval a* (σ *o Var*))*"*
— The first conjunct states that the register contains the correct value, the second conjunct states that the variable state is correct. Note that *fst* and *snd* select the first and second element of a pair.