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

WS 2014/15
28. 10. 2014

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 Boolean If expressions

We consider an alternative definition of boolean expressions, which feature a conditional construct:

datatype *ifexp* = *Bc′ bool* | *If ifexp ifexp ifexp* | *Less′ aexp aexp*

1. Define a function *ifval* analogous to *bval*, which evaluates *ifexp* expressions.
2. Define a function *translate*, which translates *ifexp*s to *bexp*s. State and prove a lemma showing that the translation is correct.

theory *hw01*
imports *Main*
begin

Homework 3.1 Register Machine from Hell

Submission until Tuesday, November 4, 2014, 10:00am.

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 *state* = *"nat ⇒ int"*

Define a function to execute a single instruction

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

Lift your definition to lists of instructions

fun *execs* :: *"instr list ⇒ state ⇒ state"* **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.

lemma [*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*.

lemma [*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 "*execs* (*cmp e* (*maxvar e* + *1*)) *s 0* = *val e* (*s o Suc*)"

Homework 3.2 a^*b^* language checker

Submission until Tuesday, November 4, 2014, 10:00am. This homework is worth 5 bonus points.
Use the file `tmpl03_ab.thy` for this exercise.

We define an inductive predicate accepting the language $S = a^*b^*$. The homework is to define recursive functions over lists which parse the same language and then to show that each word in S is accepted by *is_ab*.

First we introduce the type of our language tokens containing only a and b

datatype $ab = a \mid b$

Then we define the following language S:

inductive_set S :: *"ab list set"* **where**
  "$w \in S \Longrightarrow [a]$ @ $w \in S$"
| *"$w \in S \Longrightarrow w$ @ $[b] \in S$"*
| *"$[] \in S$"*

Now define a recursive function over *ab list* which checks that the list consists of a's followed by b's. (Hint: define a helper function which only checks that the list only consists of b's.)

fun *is_ab* :: *"ab list $\Rightarrow$ bool"* **where**

Finally show the following theorem:

lemma *"$w \in S \Longrightarrow$ is_ab w"*