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

WS 2011/12
04. 11. 2011

Semantics of Programming Languages

Exercise Sheet 3

Exercise 3.1 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.

Exercise 3.2 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"*

Homework 3.1 Compilation to Stack Machine

Submission until Wednesday, November 16, 12:00 (noon).

On exercise sheet 2, we defined arithmetic expressions with side effects and exceptions. Regard a version with only side effects here:

datatype *aexp′ = N′ int | V′ vname | PI′ vname | Plus′ aexp′ aexp′*

Evaluation can be formulated like this:

fun *aval′ :: "aexp′ ⇒ state ⇒ val × state"* **where**
"aval′ (N′ n) s = (n, s)" |
"aval′ (V′ x) s = (s x, s)" |
"aval′ (PI′ x) s = (s x, s(x:=s x+1))" |
"aval′ (Plus′ a1 a2) s =
(case aval′ a1 s of (v1, s′) ⇒
(case aval′ a2 s′ of (v2, s″) ⇒ (v1+v2, s″)))"

In the lectures, compilation of arithmetic expressions to a stack machine was discussed. In this homework, arithmetic expressions with side effects shall be compiled to an extended stack machine. Extend the instruction datatype and the *exec*-function accordingly, define a compiler from *aexp′* to instructions of the new stack machine, and show that the compiler is correct.

The simplest way is to add a single instruction *PILOAD vname* to the stack machine, that loads a variable onto the stack and then increments its value.

datatype *instr = LOADI val | LOAD vname | PILOAD vname | ADD*

The execution function may be defined to map tuples of states and stacks to tuples of states and stacks:

fun *exec1 :: "instr ⇒ (state × stack) ⇒ (state × stack)"* **where**
fun *exec :: "instr list ⇒ (state × stack) ⇒ (state × stack)"* **where**
fun *comp :: "aexp′ ⇒ instr list"* **where**

If you formalize your stack machine as indicated above, you have to show the following correctness property:

theorem *"exec (comp a) (s,stk) = (case aval′ a s of (v,s′) ⇒ (s′,v#stk))"*

Hint: Because the definition of *aval′* includes case expressions on product types, you may need to use the split rule *prod.split* in your proof.

Homework 3.2 Avoiding Stack Underflow

Submission until Wednesday, November 16, 12:00 (noon).
NOTE: This homework problem builds on the extended instruction type *instr*, execution function *exec*, and expression compiler *comp* from the previous problem.

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.

Define an inductive predicate *can_exec* :: *nat* ⇒ *instr list* ⇒ *nat* ⇒ *bool*, where *can_exec n is n′* means that with any initial stack of length *n*, the instructions *is* can be executed *without underflowing the stack*, resulting in a final stack of length *n′*.

inductive *can_exec* :: *"nat ⇒ instr list ⇒ nat ⇒ bool"* **where**

Using your introduction rules for *can_exec*, prove each of the following instances:

lemma *"can_exec 0 [LOAD x] (Suc 0)"*
lemma *"can_exec 0 [LOAD x, LOADI v, ADD] (Suc 0)"*
lemma *"can_exec (Suc (Suc 0)) [PILOAD x, ADD, ADD, LOAD y] (Suc (Suc 0))"*

This next proposition should NOT be provable!

"can_exec (Suc 0) [PILOAD x, ADD, ADD, LOAD y] (Suc 0)"

Prove that sequences of instructions generated by *comp* never underflow the stack:

theorem *"can_exec n (comp a) (Suc n)"*