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

WS 2013/14
29. 10. 2013

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.

Homework 3.1 Let expressions (I)

Submission until Tuesday, November 5, 2013, 10:00am.

Please include the string ,,[Semantics]" into the subject-line of your submissions!

The following type adds a *Let* construct to arithmetic expressions:

datatype *lexp = N val | V vname | Plus lexp lexp | Let vname lexp lexp*

The new *Let* constructor acts like a local variable binding: When evaluating *Let x e1 e2*, we first evaluate *e1*, bind the resulting value to the variable *x* and then evaluate *e2* in the new state.

Define a function *lval*, which evaluates *lexp* expressions. Note that you can use the notation $f(x := v)$ to express function update. It is defined as follows:

$f(a := b) = (\lambda x.\ \mathit{if}\ x = a\ \mathit{then}\ b\ \mathit{else}\ f\ x)$

fun *lval* :: *"lexp ⇒ state ⇒ val"*

Define a function that transforms such an expression into an equivalent one that does not contain *Let*. Prove that your transformation is correct. Note: Do the transformation by inlining the bound variables.

fun *inline* :: *"lexp ⇒ aexp"*
value *"inline (Let ''x'' (Plus (N 1) (N 1)) (Plus (V ''x'') (V ''x'')))"*
— Should return: *aexp.Plus (aexp.Plus (aexp.N 1) (aexp.N 1)) (aexp.Plus (aexp.N 1) (aexp.N 1))*

lemma *val_inline*: *"aval (inline e) st = lval e st"*

Define a function that eliminates occurrences of *Let x e1 e2* that are never used, i.e., where *x* does not occur free in *e2*. An occurrence of a variable in an expression is called free, if it is not in the body of a *Let* expression that binds the same variable. E.g., the variable x occurs free in *Plus* (*V* x) (*V* x), but not in *Let* x (*N* *0*) (*Plus* (*V* x) (*V* x)). Prove the correctness of your transformation.

fun *elim* :: *"lexp ⇒ lexp"*
lemma *"lval (elim e) st = lval e st"*

Some Hints:

- When different datatypes have a constructor with the same name, they can unambiguously be referred to using their qualified name, e.g., *aexp.Plus* vs. *lexp.Plus*.
- When you feel that the proof should be trivial to finish, you can also try the *sledgehammer* command. It invokes an extensive proof search that includes more library lemmas.

Homework 3.2 Let expressions (II)

Submission until Tuesday, November 5, 2013, 10:00am. This homework is worth 5 bonus points.

When inlining let-expressions, the inlined expression may be exponentially larger than the original expression. Show that, for all n, there is an expression e of size at least n, such that its inlined version is exponentially larger.

Hints Define a function *gen_exp* :: *nat* ⇒ *lexp* that constructs a suitable expression for any n.

The *size*-function gives you the size of any datatype, including *aexp* and *lexp*. Note that it is defined to be zero for non-recursive constructors. Other useful functions may be integer division (a *div* b) and exponentiation $a\hat{}b$.

Part of this homework's challenge is to come up with the correct theorems yourself. So make sure that the theorems you prove really state the intended proposition.