Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch, Kevin Kappelmann

WS 23/24
30.10.2023

Semantics of Programming Languages

Exercise Sheet 2

This exercise sheet depends on definitions from the files *AExp.thy* and *BExp.thy*, which may be imported as follows:

theory *Ex02* **imports** *"HOL−IMP.AExp" "HOL−IMP.BExp"* **begin**

Exercise 2.1 Induction

Define a function *deduplicate* that removes duplicate occurrences of subsequent elements from a list.

fun *deduplicate* :: *"'a list ⇒ 'a list"*

The following should evaluate to *True*, for instance:

value *"deduplicate [1,1,2,3,2,2,1::nat] = [1,2,3,2,1]"*

Prove that a deduplicated list has at most the length of the original list:

lemma *"length (deduplicate xs) ≤ length xs"*

Exercise 2.2 Substitution Lemma

A syntactic substitution replaces a variable by an expression.

Define a function *subst* that performs a syntactic substitution, i.e., *subst x a′ a* shall be the expression a where every occurrence of variable x has been replaced by expression a'.

fun *subst* :: *"vname ⇒ aexp ⇒ aexp ⇒ aexp"*

Instead of syntactically replacing a variable x by an expression a', we can also change the state s by replacing the value of x by the value of a' under s. This is called *semantic substitution*.

The *substitution lemma* states that semantic and syntactic substitution are compatible. Prove the substitution lemma:

lemma *subst_lemma*: *"aval (subst x a′ a) s = aval a (s(x := aval a′ s))"*

Note: The expression $s(x := v)$ updates a function at point x. It is defined as:

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

Compositionality means that one can replace equal expressions by equal expressions. Use the substitution lemma to prove *compositionality* of arithmetic expressions:

lemma *comp*: *"aval a1 s = aval a2 s ⟹ aval (subst x a1 a) s = aval (subst x a2 a) s"*

Exercise 2.3 Arithmetic Expressions With Side-Effects

We want to extend arithmetic expressions by the postfix increment operation $x{+}{+}$, as known from Java or C++.

The increment can only be applied to variables. The problem is, that it changes the state, and the evaluation of the rest of the term depends on the changed state. We assume left to right evaluation order here.

Define the datatype of extended arithmetic expressions. Hint: If you do not want to hide the standard constructor names from IMP, add a tick ($'$) to them, e.g., $V'\ x$.

The semantics of extended arithmetic expressions has the type $\mathit{aval}' :: \mathit{aexp}' \Rightarrow \mathit{state} \Rightarrow \mathit{val} \times \mathit{state}$, i.e., it takes an expression and a state, and returns a value and a new state. Define the function aval'.

Test your function for some terms. Is the output as expected? Note: <> is an abbreviation for the state that assigns every variable to zero:

$<> \equiv \lambda x.\ 0$

value *"<>(x := 0)"*
value *"aval' (Plus' (PI' ''x'') (V' ''x'')) <>"*
value *"aval' (Plus' (Plus' (PI' ''x'') (PI' ''x'')) (PI' ''x'')) <>"*

Is the plus-operation still commutative? Prove or disprove!

Show that the valuation of a variable cannot decrease during evaluation of an expression:

lemma *aval'_inc*:
"aval' a <> = (v, s') ⟹ 0 ≤ s' x"

Hint: If *auto* on its own leaves you with an *if* in the assumptions or with a *case*-statement, you should modify it like this: (*auto split*: *if_splits prod.splits*).

Homework 2.1 Tail-Recursive Counting

Submission until Monday, November 6, 23:59pm.

Define a tail-recursive counting function (which counts the number of occurrences of a particular element in a list), using an auxiliary argument:

fun *count_tr* :: *"'a list ⇒ 'a ⇒ nat ⇒ nat"*

Tail-recursive means that the recursive call must be the outermost function call, i.e., the recursive call must be of form *count_tr* $(x \# xs)\ y\ n =$ *count_tr* $A\ B\ C$ for some (non-recursive) terms A, B, and C.

Then you need to prove that *count_tr* is correct w.r.t. to the count function defined in Exercise 1.1:

lemma *tailrec_count*: *"count_tr xs y 0 = count xs y"*

Hint: In order to prove the above lemma, you may first need to prove a more general fact about *count_tr* (employing an arbitrary argument n instead of *0*), of which the above lemma is a particular case.

Homework 2.2 Let expressions

Submission until Monday, November 6, 23:59pm.

We extend the *aexp* datatype by adding a construct for *let*-expressions:
datatype *aexp'* = *N' int* | *V' (char list)* | *Plus' aexp' aexp'* | *Let' (char list) aexp' aexp'*
An expression *Let x e a* binds the expression e to variable x in a:
aval' (Let' x a b) s = aval' b (s(x := aval' a s))

Define a function that transforms such an expression into an equivalent one that does not contain *Let*. Prove that your transformation is correct. *Hint:* Re-use the imported tutorial material!

fun *inline* :: *"aexp' ⇒ aexp"*
value *"inline (Let' ''x'' (Plus' (N' 1) (N' 1)) (Plus' (V' ''x'') (V' ''x''))) =*
Plus (Plus (N 1) (N 1)) (Plus (N 1) (N 1))"

theorem *aval_inline*: *"aval (inline e) s = aval' e s"*

Define a function that eliminates occurrences of *Let' x e1 e2* that are never used, i.e., where x does not occur free in e_2. 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. For example, the variable x occurs free in *Plus' (V' x) (V' x)*, but not in *Let' x (N' 0) (Plus' (V' x) (V' x))*. Do not eliminate any other *Let*s! Prove the correctness of your transformation.

fun *elim* :: *"aexp' ⇒ aexp'"*

Some examples:

value *"elim (Let' ''x'' (N' 1) (N' 0)) = N' 0"*
value *"elim (Let' ''x'' (N' 1) (Let' ''x'' (N' 2) (V' ''x''))) = Let' ''x'' (N' 2) (V' ''x'')"*

theorem *aval'_elim*: *"aval' (elim e) s = aval' e s"*

Hints:

- Define an auxiliary function for *elim*
- 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.