Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2020/21
9.11.2020

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a syntactic substitution, i.e., *subst x a' a* shall be the expression a where every occurrence of variable x has been replaced by expression a'.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

Exercise 2.4 Variables of Expression (Time Permitting)

Define a function that returns the set of variables occurring in an arithmetic expression.

fun *vars* :: *"aexp ⇒ vname set"* **where**

Show that arithmetic expressions do not depend on variables that they don't contain.

lemma *ndep*: *"x ∉ vars e ⟹ aval e (s(x:=v)) = aval e s"*

Homework 2.1 Tail-Recursive Form of Addition

Submission until Sunday, Nov 15, 23:59.

A function is called tail-recursive if for all function equations, the recursive call is the last computation performed. Tail-recursive functions are often preferred in software as they don't require a new stack frame for each call, making them easier to handle.
In this exercise, define a tail-recursive version of the *add* function.

fun *add* :: *"nat ⇒ nat ⇒ nat"*

Next, prove that *add* is associative. Hint: The proof will require at least one additional lemma. Also remember that some proofs by induction may require generalization with *arbitrary*.

theorem *add_assoc*: *"add (add x y) z = add x (add y z)"*

Finally, you must prove that *add* is commutative. This may require more lemmas in addition to those used for the associativity proof.

theorem *add_commut*: *"add x y = add y x"*

Homework 2.2 Where expressions

Submission until Sunday, Nov 15, 23:59.

Do not forget to hand in your homework in the submission system! Note that the percentage displayed in the system is just a very rough indication of the score you will get and need not correspond to the actual score you will receive during grading.

We modify the *aexp* datatype by adding a syntactic *where* construct to arithmetic expressions:

datatype *wexp = N val | V vname | Plus wexp wexp | Where wexp vname wexp*

The new *Where* constructor acts like in mathematical texts, where variables are defined after they are used. For example, the sentence "compute f(n) where n = g(x)" ultimately means "compute f(g(x))". Applied to our arithmetic expressions, this means evaluating *Where f n g* requires evaluating *g*, then assigning the result to the variable name *n*, and finally evaluating *f* under this new state.

We modify the evaluation function *aval* to accommodate for the new construct *Where*:

fun *wval* :: *"wexp ⇒ state ⇒ val"*

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

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

theorem *val_inline*: *"aval (inline e) s = wval e s"*

Define a function that eliminates occurrences of *Where e1 x e2* that are never used, i.e., where x does not occur free in e_1. An occurrence of a variable in an expression is called free if it is not in the body of a *Where* expression that binds the same variable. For example, the variable x occurs free in *wexp.Plus (wexp.V x) (wexp.V x)*, but not in *Where (wexp.Plus (wexp.V x) (wexp.V x)) x (wexp.N 0)*. Prove the correctness of your transformation.

fun *elim* :: *"wexp ⇒ wexp"*
theorem *wval_elim*: *"wval (elim e) s = wval e s"*

General 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. *wexp.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.