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

WS 2022/23
1.11.2022

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a syntactic substitution, i.e., *subst x a′ a* shall be the expression a where every occurrence of variable x has been replaced by expression a'.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

Submission until Monday, Nov 7, 23:59pm.

We want to encode a list of integers as follows: All consecutive repetitions of an element are replaced by a pair that has 1) the element and 2) the number of repetitions.
For example:

enc [*1*,*3*,*3*,*8*] = [(*1*,*1*),(*3*,*2*),(*8*,*1*)]

enc [*3*,*4*,*5*] = [(*3*,*1*),(*4*,*1*),(*5*,*1*)]

Background: This algorithm may be used in lossless data compression, when it is expected that data was created by modifying a fixed background, e.g. in palette-based computer images.
Define a function to encode a list with run-length encoding.

fun *rlenc* :: *"'a ⇒ nat ⇒ 'a list ⇒ ('a × nat) list"*

The first argument is meant to keep track of the value that was last seen, and the second argument is meant to specify the number of times the last element was seen. *Hint:* in Isabelle/HOL there is the function *replicate* that takes a natural number n and an object a, and returns a list of length n, whose members are all a.

value *"replicate (3::nat) (1::nat) = [1,1,1]"*

Test cases:

value *"rlenc 0 0 ([1,3,3,8] :: int list) = [(0,0),(1,1),(3,2),(8,1)]"*
value *"rlenc 1 0 ([3,4,5] :: int list) = [(1,0),(3,1),(4,1),(5,1)]"*

Define the decoder. It takes a list that is encoded by *rlenc*.

fun *rldec* :: *"('a × nat) list ⇒ 'a list"*

Show that encoding and then decoding yields the same list. *Hint:* You will need a lemma which needs generalization. Moreover, you will need to use the lemma *replicate_append_same*, if you used *replicate* in defining *rlenc*.

theorem *enc_dec*: *"rldec (rlenc a 0 l) = l"*

Homework 2.2 Multiplication & Distributivity

Submission until Monday, Nov 7, 23:59pm.

In this exercise we add our language of arithmetic expressions with multiplication of constants and expressions.

We say that an arithmetic expression is *normalized* (with respect to distributivity) if it is an arithmetic expression where constants are *only* multiplied to variables. For example: *Mult 3* (*V ''x''*) is normalized. The following examples are *not* normalized: *Mult 5* (*N 6*), *Mult 5* (*Mult 6 a*), or *Mult 5* (*Plus a b*).

We modify the *aexp* datatype by adding a syntactic construct *Mult* for multiplication with constants:
datatype *aexp* = *N int* | *V* (*char list*) | *Plus aexp aexp* | *Mult int aexp*

We modify the evaluation function *aval* to accommodate for the new construct *Mult*:
aval (*Mult i a*) *s* = *i* ∗ *aval a s*

Step A Implement the function *normal* which returns *True* only when the arithmetic expression is normalized.

fun *normal* :: *"aexp ⇒ bool"*

Step B Implement the function *normalize* which translates an arbitrary arithmetic expression intro a normalized arithmetic expression.

fun *normalize* :: *"aexp ⇒ aexp"*

Step C Prove that *normalize* does not change the result of the arithmetic expression.

theorem *semantics_unchanged*: *"aval* (*normalize a*) *s* = *aval a s"*

Hint: It can be helpful to add the following modifiers to *auto* and friends: *split*: *aexp.split*.

Step D Prove that *normalize* does indeed return a normalized arithmetic expression.

theorem *normalize_normalizes*: *"normal* (*normalize a*)*"*