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

WS 2014/15
22. 10. 2013

Semantics of Programming Languages

Exercise Sheet 2

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

theory *Ex02*
imports *"~~/src/HOL/IMP/AExp"*
begin

Exercise 2.1 Substitution Lemma

A syntactic substitution replaces a variable by an expression.

Define a function *subst* :: *vname* ⇒ *aexp* ⇒ *aexp* ⇒ *aexp* 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a syntactic substitution, i.e., *subst x a′ a* shall be the expression *a* where every occurrence of variable *x* has been replaced by expression *a′*.

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 2.2 Arithmetic Expressions With Side-Effects and Exceptions

We want to extend arithmetic expressions by the division operation and by the postfix increment operation *x*++, as known from Java or C++.

The problem with the division operation is that division by zero is not defined. In this case, the arithmetic expression should evaluate to a special value indicating an exception.

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 *aval′* :: *aexp′* ⇒ *state* ⇒ (*val*×*state*) *option*, i.e., it takes an expression and a state, and returns a value and a new state, or an error value. Define the function *aval′*.

(Hint: To make things easier, we recommend an incremental approach to this exercise: First define arithmetic expressions with incrementing, but without division. The function *aval′* for this intermediate language should have type *aexp′* ⇒ *state* ⇒ *val*×*state*. After completing the entire exercise with this version, then modify your definitions to add division and exceptions.)

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

<> ≡ λ*x*. *0*

value *"aval′ (Div′ (V′ ''x'') (V′ ''x'')) <>"*
value *"aval′ (Div′ (PI′ ''x'') (V′ ''x'')) <''x'':=1>"*
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 s = Some (v,s′) ⟹ s x ≤ 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*: *split_if_asm option.splits*).

Exercise 2.3 Let expressions

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 2.1 Binary Search Trees

Submission until Tuesday, Oct 28, 10:00am. **Please include the string „[Semantics]" into the subject-line of your submissions!**

A binary search tree (BST) is a binary tree where Nodes are labeled by integers[1], that has the *search tree property*, i.e., for each Node labeled with *x*, all Nodes in the left subtree have labels less than *x*, and all Nodes in the right subtree have labels greater than *x*.

BSTs are described by the following datatype:

datatype *bst = Leaf | Node int bst bst*

[1]In general, any ordered datatype

Define a function *lookup* that checks whether an element is contained in a BST. For a query on a tree with depth d, your function must not visit more than d Nodes!

fun *lookup* :: *"int ⇒ bst ⇒ bool"* **where**

Define a function *insert* that inserts an element into a BST. Your function must preserve the search tree property, and, for an insertion of an element into a tree with depth d, it must not visit more than d Nodes.

fun *insert* :: *"int ⇒ bst ⇒ bst"* **where**

As a warmup, show that *insert* and *lookup* are related as expected:

lemma *"lookup i (insert j t) ⟷ (if i=j then True else lookup i t)"*

Next, define a function *bst* that checks whether a tree has the search tree property. Use two auxiliary functions to check whether all elements of a tree are less than/greater than a given value:

fun *Nodes_less* :: *"int ⇒ bst ⇒ bool"* **where**
fun *Nodes_greater* :: *"int ⇒ bst ⇒ bool"* **where**
fun *stree* :: *"bst ⇒ bool"* **where**

This time, don't worry about the efficiency of *stree.* It's not intended to be executed, but only to define the property of being a serach tree.
Now show that insertion preserves the search tree property:

lemma *"stree t ⟹ stree (insert x t)"*

Hint: Induction on t. You will need auxiliary lemmas.

Finally, for 5 bonus points, define a function that deletes all elements less than a given value from a tree, and show that it is correct.
Again, for a tree of depth d, your function should visit at most d Nodes.

fun *dell* :: *"int ⇒ bst ⇒ bst"*

Correctness:

lemma *"lookup x (dell a t) = (lookup x t ∧ x≥a)"*
lemma *"stree t ⟹ stree (dell a t)"*

Homework 2.2 Normalizing Expressions

Submission until Tuesday, Oct 28, 10:00am. In this exercise we add constant multiplication to arithmetic expressions. A *normalized* arithmetic expression is an arithmetic expression where *only* variables are multiplied. 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)*.
Use the file `tmpl02_aexp.thy` for this exercise.

For the following exercise we need to add the *algebra_simps* theorem collection to the simplifier:

declare *algebra_simps*[*simp*]

We modify the *aexp* datatype by adding a syntax element for constant multiplication *Mult*:

datatype *aexp* = *N int* | *V vname* | *Plus aexp aexp* | *Mult int aexp*

We modify the *aval* evaluation function to add constant multiplication *Mult*:

fun *aval* :: *"aexp ⇒ state ⇒ val"* **where**
"aval (N n) s = n" |
"aval (V x) s = s x" |
"*aval* (*Plus* a_1 a_2) *s* = *aval* a_1 *s* + *aval* a_2 *s*" |
"aval (Mult i a) s = i ∗ aval a s"

We modify the *aval* evaluation function to add constant multiplication *Mult*:

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

fun *normal* :: *"aexp ⇒ bool"* **where**

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

fun *normalize* :: *"aexp ⇒ aexp"* **where**

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

lemma *"aval (normalize a) s = aval a s"*

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

lemma *"normal (normalize a)"*