Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Andrei Popescu, Peter Lammich

WS 2012/13
23. 10. 2012

Semantics of Programming Languages

Exercise Sheet 2

This exercise sheet depends on definitions from the files *AExp.thy* and *BExp.thy*, which may be obtained from http://www21.in.tum.de/teaching/semantik/WS1213/IMP/. Copy them into the same directory as your *Ex02.thy* file, and add them to your imports as follows:

theory *Ex02*
imports *Main AExp BExp*
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a syntactic substitution, i.e., *subst x a′ a* shall be the expression *a* where every occurrence of variable *x* has been replaced by expression *a′*.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 $aval' :: aexp' \Rightarrow state \Rightarrow (val \times state)\ option$, 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 $aexp' \Rightarrow state \Rightarrow val \times state$. 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:

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

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*).

Homework 2.1 Exclusive Or

Submission until Tuesday, October 30, 10:00am.

Write a function *xor* that takes two boolean expressions a::*bexp* and b::*bexp*, and returns a boolean expression c such that for all states s::*state*: $bval\ c\ s \longleftrightarrow (bval\ a\ s \neq bval\ b\ s)$.
Prove that your function is correct.

definition *xor* :: *"bexp ⇒ bexp ⇒ bexp"* **where**

lemma *"bval (xor a b) s ⟷ bval a s ≠ bval b s"*

Homework 2.2 Tail-Recursive Multiplication

Submission until Tuesday, October 30, 10:00am.

The list-reversing function *itrev* is an example of a *tail-recursive* function: Note that the right-hand side of the second equation for *itrev* is simply an application of *itrev* to different arguments.

fun *itrev* :: *"'a list ⇒ 'a list ⇒ 'a list"* **where**
"itrev [] ys = ys" |
"itrev (x#xs) ys = itrev xs (x#ys)"

In this homework problem you will define in Isabelle a tail-recursive version of multiplication for natural numbers, using an auxiliary argument. More precisely, you should define a function *mult* :: *nat ⇒ nat ⇒ nat ⇒ nat* such that $\forall\ x\ y.\ mult\ x\ y\ 0 = x * y$. Like *itrev*, your definition should be tail-recursive: in the recursive case the right-hand side should be just an application of *mult* to different arguments. The only functions you are allowed to use in the recursive clauses for *mult* are *Suc* and +.

First you need to define the function:

fun *mult* :: *"nat ⇒ nat ⇒ nat ⇒ nat"*

Then you need to prove that *mult* is correct:

lemma *mult_correct*: *"mult x y 0 = x * y"*

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

Homework 2.3 Associativity of Addition

Submission until Tuesday, October 30, 10:00am.

***Note**: This is a "bonus" exercise worth five additional points, making the maximum possible score for all homework on this sheet 15 out of 10 points.*

In this assignment, you shall write an arithmetic expression optimizer that summarizes all constants that occur in an expression to a single constant. For example, *Plus* (*N 1*) (*Plus* (*N 2*) (*V ''x''*)) shall become *Plus* (*N 3*) (*V ''x''*).

Note that there is a function *asimp* in *AExp.thy* that eliminates zeroes and evaluates constant subexpressions, but cannot handle the above case, as it does not know about associativity.

In this exercise, the approach will be a bit different. First, write a function *collect_const* :: *aexp* ⇒ *int* that returns the sum of all constants in an expression. Next, write a function *zero_const* :: *aexp* ⇒ *aexp* that replaces all constants in an expression by zeroes (they will be optimized away later).

fun *collect_const* :: *"aexp ⇒ int"* **where**
fun *zero_const* :: *"aexp ⇒ aexp"* **where**

Next, define a function *move_const* :: *aexp* ⇒ *aexp* that produces an arithmetic expression that adds the results of *collect_const* and *zero_const*. Show that *move_const* preserves the value of an expression.

definition *move_const* :: *"aexp ⇒ aexp"* **where**
lemma *aval_move_const*[*simp*]: *"aval* (*move_const t*) *s* = *aval t s"*

Finally, define a function *full_asimp* :: *aexp* ⇒ *aexp*, that uses *asimp* to eliminate the zeroes left over by *move_const*, and show that it preserves the value of arithmetic expressions.

definition *full_asimp* :: *"aexp ⇒ aexp"* **where**
lemma *aval_full_asimp*[*simp*]: *"aval* (*full_asimp t*) *s* = *aval t s"*