Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Brian Huffman, Peter Lammich

WS 2011/12
28. 10. 2010

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://www4.in.tum.de/lehre/vorlesungen/semantik/WS1112/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 *syn_subst* :: *name* ⇒ *aexp* ⇒ *aexp* ⇒ *aexp* that performs a syntactic substitution, i.e., *syn_subst x a′ a* shall be the expression *a* where every occurrence of variable *x* has been replaced by expression *a′*.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 (syn_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:

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

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 (syn_subst x a1 a) s = aval (syn_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 leaves you with some if or case statements in the assumptions, you may use the *split*: *split_if_asm option.split_asm* attribute.

Homework 2.1 Less Than or Equal

Submission until Wednesday, November 9, 12:00 (noon).

In our boolean expressions, there is no $\leq$ operator. Write a function that takes two arithmetic expressions *a1* and *a2*, and returns a boolean expression *b*, such that for all *s*: *bval b s* $\longleftrightarrow$ *aval a1 s* $\leq$ *aval a2 s*. Prove that your function is correct.

definition *ble* :: *"aexp ⇒ aexp ⇒ bexp"* **where**

lemma *"bval (ble a1 a2) s ⟷ aval a1 s ≤ aval a2 s"*

Homework 2.2 Tail-Recursive Form of Addition

Submission until Wednesday, November 9, 12:00 (noon).

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 a tail-recursive version of addition for natural numbers, and prove that it is associative and commutative.

First, define a function *add* :: *nat ⇒ nat ⇒ nat* in Isabelle that calculates the sum of its arguments. Like *itrev*, your definition should be tail-recursive: That is, in the recursive case the right-hand side should be just an application of *add* to different arguments.

fun *add* :: *"nat ⇒ nat ⇒ nat"*

Next, you must 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 (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 x y = add y x"*

Homework 2.3 Completeness of Constant Folding

Submission until Wednesday, November 9, 12:00 (noon).

***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.*

The *AExp* theory includes a function *asimp_const* :: *aexp* ⇒ *aexp* that performs *constant-folding* optimizations. It replaces occurrences of the pattern *Plus* (*N x*) (*N y*) with *N* (*x* + *y*), resulting in an equivalent but smaller expression.

The *asimp_const* function is supposed to perform as many constant-folding optimizations as possible, yielding an expression that is somehow "optimal".

Your first task is to specify exactly what "optimal" means. Define a predicate *optimal* that evaluates to *True* if and only if no further constant-folding optimizations of the form *Plus* (*N x*) (*N y*) ↦ *N* (*x* + *y*) are possible.

fun *optimal* :: *"aexp ⇒ bool"*

Test your definition of *optimal* on the following expressions:

value *"optimal* (*Plus* (*N 1*) (*N 1*))*"*
value *"optimal* (*Plus* (*V x*) (*Plus* (*N 1*) (*N 1*)))*"*
value *"optimal* (*Plus* (*N 1*) (*Plus* (*N 1*) (*V x*)))*"*

Which of these are optimal, and which are not?

Finally, prove that *asimp_const* always yields an optimal expression:

theorem *"optimal* (*asimp_const a*)*"*

Hint: Applying *auto* may leave you with subgoals containing case expressions like "*case x of ... ⇒ ...*". You can make progress by using *case_tac*:

apply (*case_tac x*)

Alternatively, you can use the *split* option to have *auto* perform case splitting on type *aexp* automatically:

apply (*auto split*: *aexp.split*)

Beware that automatic case splitting may cause *auto* to loop in some circumstances, depending on how your definitions are formulated.