

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 $subst :: vname \Rightarrow aexp \Rightarrow aexp \Rightarrow aexp$ 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a syntactic substitution, i.e., $subst\ x\ a'\ a$ shall be the expression a where every occurrence of variable x has been replaced by expression a' .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:

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

Compositionality means that one can replace equal expressions by equal expressions. Use the substitution lemma to prove *compositionality* of arithmetic expressions:

lemma *comp*: “ $aval\ a1\ s = aval\ a2\ s \implies aval\ (subst\ x\ a1\ a)\ s = aval\ (subst\ x\ a2\ a)\ s$ ”

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: $\langle \rangle$ is an abbreviation for the state that assigns every variable to zero:

$\langle \rangle \equiv \lambda x. 0$

```

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') \implies s x \leq 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 Variables of Expression

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

```

fun vars :: "aexp  $\Rightarrow$  vname set" where

```

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

lemma $ndep$: " $x \notin vars e \implies aval e (s(x:=v)) = aval e s$ "
by (*induction e auto*)

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 integers¹, 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
```

Define a function α that maps a tree to the set of its nodes

```
fun  $\alpha$  :: “bst  $\Rightarrow$  int set” where
```

Define a function that checks whether a tree is a search tree

```
fun invar :: “bst  $\Rightarrow$  bool” where
```

Note: Intuitively, α maps a (concrete) BST to the (abstract) set that it describes, and *invar* is the invariant (or well-formedness condition) on the concrete BST.

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  $\Rightarrow$  bst  $\Rightarrow$  bool” where
```

Show that lookup is correct

```
lemma “invar t  $\Longrightarrow$  lookup x t  $\longleftrightarrow$  x  $\in$   $\alpha$  t”
```

Define a function *ins* 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 ins :: “int  $\Rightarrow$  bst  $\Rightarrow$  bst” where
```

Show that insert is correct

```
lemma ins_correct1: “[ invar t ]  $\Longrightarrow$   $\alpha$  (ins x t) = insert x ( $\alpha$  t)”
```

```
lemma ins_correct2: “invar t  $\Longrightarrow$  invar (ins x t)”
```

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.

Note: A bonus point counts on the side of your personal score, but not for the maximal reachable points by which we will divide your personal score to get your percentage.

```
fun dell :: “int  $\Rightarrow$  bst  $\Rightarrow$  bst”
```

¹In general, any ordered datatype

Show correctness

lemma *dell_correct1*: “*invar t* $\implies \alpha$ (*dell a t*) = {*x* ∈ α *t*. *x* ≥ *a*}”

lemma *dell_correct2*: “*invar t* \implies *invar (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 `tmp102_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* \Rightarrow *state* \Rightarrow *val*” **where**
“*aval (N n) s* = *n*” |
“*aval (V x) s* = *s x*” |
“*aval (Plus a₁ a₂) s* = *aval a₁ s* + *aval a₂ 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* \Rightarrow *bool*” **where**

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

fun *normalize* :: “*aexp* \Rightarrow *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)*”