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

WS 2015/16
20. 10. 2015

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a syntactic substitution, i.e., *subst* x a' a shall be the expression a where every occurrence of variable x has been replaced by expression a'.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

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

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 (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:

<> ≡ *λx. 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 Variables of Expression

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

fun *vars* :: *“aexp ⇒ vname set”* **where**

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

lemma *ndep*: *“x∉vars e ⟹ aval e (s(x:=v)) = aval e s”*
  by (*induction e*) *auto*

Homework 2.1 Conditionals

Submission until Tuesday, 27. Oct. 2015, 10:00am.

We define a representation of Boolean expressions that only use conditionals as connective.

datatype *cexp* = *Cond cexp cexp cexp* | *Bc′ bool* | *Less′ aexp aexp*

The semantics of *Cond b t e* is if b holds, then evaluate t, else evaluate e. Define the semantics:

fun *cval* :: *"cexp ⇒ state ⇒ bool"*

Define conversions from *bexp* to *cexp* and back. Show that your conversions preserve the semantics:

fun *b2c* :: *"bexp ⇒ cexp"*

lemma *"cval (b2c b) σ = bval b σ"*

fun *c2b* :: *"cexp ⇒ bexp"*

lemma *"bval (c2b c) σ = cval c σ"*

Homework 2.2 Heaps

Submission until Tuesday, 27. Oct. 2015, 10:00am.

A (min) heap is a binary tree with node labels, such that every node is less than or equal to its successors.

We model heaps by the following datatype:

datatype *heap* = *Leaf* | *Node nat heap heap*

Define a function to check the heap property. Hint: The following function may save you some case distinctions:

```
fun le :: "nat ⇒ heap ⇒ bool" where
  "le n Leaf ⟷ True"
| "le n (Node m _ _) ⟷ n≤m"
```

fun *heap_invar* :: *"heap ⇒ bool"* **where**

Define a function to return the minimal value of a non-empty heap. Set the case for an empty heap to undefined.

```
fun get_min :: "heap ⇒ nat" where
  "get_min Leaf = undefined"
```

The following function maps a heap to the set of its elements.

fun *heap_set* **where**
  "heap_set Leaf = {}"
| *"heap_set (Node a h1 h2) = insert a (heap_set h1 ∪ heap_set h2)"*

Note that *insert x s* inserts element *x* to set *s*. To get the ∪ symbol, type \union, or use the symbols panel!

Show that *get_min* actually returns an element from the heap

lemma *get_min_correct1*: *"h≠Leaf ⟹ get_min h ∈heap_set h"*

Show that *get_min* returns an element smaller or equal to the elements of the heap

lemma *get_min_correct2*: *"h≠Leaf ⟹ heap_invar h ⟹ b∈heap_set h ⟹ get_min h≤b"*

Hint: You may need an auxiliary lemma about *le*.

As a **bonus exercise** for 5 bonus points, implement a function that merges two heaps, and show that it preserves the heap-property and that the set of elements on the new heap is the union of elements in the old heaps. You need not consider balancedness of heaps!

Note: Bonus points count on your side, but not on the total number of reachable points, when we compute the ratio of points that you scored in the homework, which will be 40

It may happen that the function package cannot prove termination of your function by default. In this case, use the following template, which makes use of the size-change termination prover, which should be able to prove termination.

function *merge* :: *"heap ⇒ heap ⇒ heap"*
where
  — Function equations as usual
by *pat_completeness auto*
termination by *size_change*

lemma *merge_correct1*: *"⟦heap_invar h1; heap_invar h2⟧ ⟹ heap_invar (merge h1 h2)"*
lemma *merge_correct2*: *"heap_set (merge h1 h2) = heap_set h1 ∪ heap_set h2"*