Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Hupel

WS 2017/18
24.10.2017

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a syntactic substitution, i.e., *subst* x a' a shall be the expression a where every occurrence of variable x has been replaced by expression a'.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

Homework 2.1 Tree Locations

Submission until Thursday, November 2, 10:00am.

We define binary trees as follows:

datatype *'a tree = Node "'a tree" 'a "'a tree" | Leaf*

In this exercise, we want to write a function that updates a sub-tree inside a larger tree. For that, we first have to define what a "location" inside a tree means. In Isabelle, we can use the **type_synonym** to define shorthands for types.

type_synonym *loc = "bool list"*

A location is a list of *bool*s that are either *True* (go left) or *False* (go right). Define a *lookup* function that takes a tree and a location and returns the sub-tree at that position. If the location is too long, just return *Leaf*. Here are some examples:

fun *lookup :: "'a tree ⇒ loc ⇒ 'a tree"*
value *"lookup (Leaf::nat tree) [] = Leaf"*
value *"lookup (Node Leaf (3::nat) (Node Leaf 2 Leaf)) [False] = Node Leaf 2 Leaf"*
value *"lookup (Node Leaf (3::nat) (Node Leaf 2 Leaf)) [False, True] = Leaf"*
value *"lookup (Node Leaf (3::nat) (Node Leaf 2 Leaf)) [False, True, False] = Leaf"*

Now, define a function *contained* that returns *True* or *False* depending on whether the location exists in the tree.

fun *contained :: "'a tree ⇒ loc ⇒ bool"*
value *"contained (Leaf::nat tree) []"*
value *"contained (Node Leaf (3::nat) (Node Leaf 2 Leaf)) [False]"*
value *"contained (Node Leaf (3::nat) (Node Leaf 2 Leaf)) [False, True]"*
value *"¬ contained (Node Leaf (3::nat) (Node Leaf 2 Leaf)) [False, True, False]"*

Finally, a function *update* that replaces the sub-tree at a given location by a new sub-tree. If the location does not exist, return the original tree unchanged.

fun *update :: "'a tree ⇒ loc ⇒ 'a tree ⇒ 'a tree"*

Prove the following lemmas. Hints:

- Use computation induction.
- You might need a lemma about *lookup*.

lemma *"¬ contained t loc ⟹ update t loc t' = t"*
lemma *"contained t loc ⟹ lookup (update t loc t') loc = t'"*

Homework 2.2 Where expressions

Submission until Thursday, November 2, 10:00am.

The following adds a *where* construct to arithmetic expressions:

datatype *wexp* = *N val* | *V vname* | *Plus wexp wexp* | *Where wexp vname wexp*

The new *Where* constructor acts like in mathematical texts, where variables are defined after they are used. For example, the sentence "compute f(n) where n = g(x)" ultimately means "compute f(g(x))". Applied to our arithmetic expressions, this means evaluating *Where t x e* requires evaluating *e*, then adding the result to the state using the variable name *x* and finally evaluating *t*.

Define a function *wval* that evaluates *wexp* expressions.

fun *wval* :: *"wexp ⇒ state ⇒ val"*

Define a function that transforms such an expression into an equivalent one that does not contain *Where*. Prove that your transformation is correct.

fun *inline* :: *"wexp ⇒ aexp"*
value
  "inline (Where (Plus (V ''x'') (V ''x'')) ''x'' (Plus (N 1) (N 1))) =
    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 = wval e st"*

Define a function that eliminates occurrences of *Where e1 x e2* that are never used, i.e., where *x* does not occur free in *e1*. An occurrence of a variable in an expression is called free if it is not in the body of a *Where* expression that binds the same variable. For example, the variable *x* occurs free in *wexp.Plus (wexp.V x) (wexp.V x)*, but not in *Where (wexp.Plus (wexp.V x) (wexp.V x)) x (wexp.N 0)*. Prove the correctness of your transformation.

fun *elim* :: *"wexp ⇒ wexp"*
lemma *"wval (elim e) st = wval e st"*

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. *wexp.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.