Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Simon Wimmer

WS 2016/17
08. 11. 2016

Semantics of Programming Languages

Exercise Sheet 3

This exercise sheet depends on definitions from the file *AExp.thy*, which may be imported as follows:

theory *Ex03*
imports *"~~/src/HOL/IMP/AExp"*
begin

Exercise 3.1 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a syntactic substitution, i.e., *subst* $x\ a'\ a$ shall be the expression a where every occurrence of variable x has been replaced by expression a'.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:

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

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

lemma *comp*: "$aval\ a1\ s = aval\ a2\ s \Longrightarrow aval\ (subst\ x\ a1\ a)\ s = aval\ (subst\ x\ a2\ a)\ s$"

Exercise 3.2 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:

<> ≡ λ*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*).

Exercise 3.3 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 3.1 Let expressions

Submission until Tuesday, November 15, 2016, 10:00am.

The following type adds a *Let* construct to arithmetic expressions:

datatype *lexp* = *N val* | *V vname* | *Plus lexp lexp* | *Let vname lexp lexp*

The new *Let* constructor acts like a local variable binding: When evaluating *Let x e1 e2*, we first evaluate *e1*, bind the resulting value to the variable x and then evaluate *e2* in the new state.

Define a function *lval*, which evaluates *lexp* expressions. Note that you can use the notation $f(x := v)$ to express function update. It is defined as follows:

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

fun *lval* :: *"lexp ⇒ state ⇒ val"*

Define a function that transforms such an expression into an equivalent one that does not contain *Let*. Prove that your transformation is correct. Note: Do the transformation by inlining the bound variables.

fun *inline* :: *"lexp ⇒ aexp"*
value *"inline (Let ''x'' (Plus (N 1) (N 1)) (Plus (V ''x'') (V ''x'')))"*
— Should return: *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 = lval e st"*

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

fun *elim* :: *"lexp ⇒ lexp"*
lemma *"lval (elim e) st = lval e st"*

Some 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. *lexp.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.

Homework 3.2 Negation Normal Form

Submission until Tuesday, November 15, 2016, 10:00am.

In this assignment, you shall write a function that converts a boolean expression over variables, conjunction, disjunction, and negation to negation normal form (NNF), and prove its correctness. A template for this homework is available on the lecture's homepage.

We start by defining our version of boolean expressions:

datatype *bexp = Not bexp | And bexp bexp | Or bexp bexp | Var vname*
fun *bval* :: *"bexp ⇒ state ⇒ bool"* — Definition in template

Next, we define a function that check whether a boolean expression is in NNF.

fun *is_nnf* :: *"bexp ⇒ bool"* — Definition in template

Define a function *nnf* which converts a boolean expression to NNF. This can be achieved by "pushing in" negations and eliminating double negations.

fun *nnf* :: *"bexp ⇒ bexp"*

Prove that your function is correct. Hint: in case you are struggling to finish the proof with structural induction, it may be helpful to consider a different induction principle.

lemma [*simp*]: *"is_nnf (nnf b)"*

lemma [*simp*]: *"bval (nnf b) s = bval b s"*

Homework 3.3 Conjunctive Normal Form

Submission until Tuesday, November 15, 2016, 10:00am.

***Note**: This is a "bonus" assignment worth three additional points, making the maximum possible score for all homework on this sheet 13 out of 10 points.*

Warning: This assignment is quite hard. Partial solutions will also be graded!

In this assignment your task is to extend the previous exercise to allow conversion to CNF. The approach we will take is to first convert expressions to NNF (negation normal form), and then apply the distributivity laws to get the CNF. We again start by defining a function to check whether a expression is in CNF:

fun *is_cnf* :: *"bexp ⇒ bool"* — Definition in template

The basic idea of converting an NNF to CNF is to first convert the operands of a disjunction, and then apply the distributivity law to get a conjunction of disjunctions.

The function *merge* $(a_1 \wedge \ldots \wedge a_n)$ $(b_1 \wedge \ldots \wedge b_m)$ shall return a term of the form $(a_1 \vee b_1)$ $\wedge$ $(a_1 \vee b_2)$ $\wedge$ ... $(a_n \vee b_m)$ that is equivalent to $(a_1 \wedge \ldots \wedge a_n)$ $\vee$ $(b_1 \wedge \ldots \wedge b_m)$.

fun *merge* :: *"bexp ⇒ bexp ⇒ bexp"*

Show that *merge* preserves the semantics and indeed yields a CNF if its operands are already in CNF. Hint: For functions over multiple arguments, the syntax for induction is *induction a b rule*: *merge.induct*

lemma [*simp*]: *"bval (merge a b) s ⟷ bval (Or a b) s"*

lemma [*simp*]: *"is_cnf a ⟹ is_cnf b ⟹ is_cnf (merge a b)"*

Next, use *merge* to write a function that converts an NNF to a CNF. The idea is to first convert the operands of a compound expression, and then compute the overall CNF (using *merge* in the *Or*-case)

fun *nnf_to_cnf* :: *"bexp ⇒ bexp"*

Prove the correctness of your function:

lemma [*simp*]: *"bval (nnf_to_cnf b) s = bval b s"*

lemma [*simp*]: *"is_nnf b ⟹ is_cnf (nnf_to_cnf b)"*

Finally, combine the two functions *nnf* and *nnf_to_cnf*, to get a function that converts any boolean expression to CNF:

definition *cnf* :: *"bexp ⇒ bexp"*

theorem [*simp*]: *"bval (cnf b) s = bval b s"*
theorem [*simp*]: *"is_cnf (cnf b)"*