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

WS 2013/14
22. 10. 2013

Semantics of Programming Languages

Exercise Sheet 2

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a syntactic substitution, i.e., *subst x a′ a* shall be the expression a where every occurrence of variable x has been replaced by expression a'.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

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

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 $aval' :: aexp' \Rightarrow state \Rightarrow (val \times state)\ option$, 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* 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*).

Homework 2.1 Constant Multiplication

Submission until Tuesday, October 29, 10:00am.

Write a function *multn* that takes a natural number n::*nat* and an arithmetic expressions a::*aexp*, and returns an arithmetic expression b such that for all states s::*state*:
$aval\ (multn\ n\ a)\ s = int\ n * aval\ a\ s$.
Prove that your function is correct.

fun *multn* :: *"nat ⇒ aexp ⇒ aexp"* **where**
lemma *"aval (multn n a) s = int n * aval a s"*

Homework 2.2 Tail-Recursive Counting

Submission until Tuesday, October 29, 10:00am.

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 some 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 in Isabelle a tail-recursive version of *count* (which counts the number of occurrences of a particular element in a list), using an auxiliary argument. More precisely, you should define a function *count_tr* :: *'a list ⇒ 'a ⇒ nat ⇒ nat* such that $\forall$ *xs y. count_tr xs y 0 = count xs y*. Like *itrev*, your definition should be tail-recursive: in the recursive case the right-hand side should only consist of if-expressions, case-distinctions and recursive applications of *count_tr*.

First you need to define the function:

primrec *count_tr* :: *"'a list ⇒ 'a ⇒ nat ⇒ nat"* **where**

Then you need to prove that *count_tr* is correct. Here, *count* is the function from exercise 1.2, you can copy it from the sample solution.

lemma *"count_tr xs y 0 = count xs y"*

Hint: In order to prove the above lemma, you may first need to prove a more general fact about *count_tr* (employing an arbitrary argument *n* instead of *0*), of which the above lemma is a particular case.

Homework 2.3 Disjunctive Normal Form

Submission until Tuesday, October 29, 10:00am.

***Note**: This is a "bonus" assignment worth five additional points, making the maximum possible score for all homework on this sheet 15 out of 10 points.*

Warning: This assignment is quite hard. Also partial solutions will be graded!

In this assignment, you shall write a function that converts a boolean expression over variables, conjunction, disjunction, and negation to disjunctive normal form, 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 functions that check whether a boolean exression is in DNF or NNF.

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

An approach to convert a boolean expression to DNF is to first convert it to NNF (negation normal form), and then apply the distributivity laws to get the DNF. Thus, start with defining a function that converts any boolean expression to NNF. This can be done by "pushing in" negations, and eliminating double negations.

fun *to_nnf* :: *"bexp ⇒ bexp"* **where**

Prove that your function is correct. Hint: use the induction rule generated by the function package, the syntax is: *induction b rule*: *to_nnf.induct*

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

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

The basic idea of converting an NNF to DNF is to first convert the operands of a conjunction, and then apply the distributivity law to get a disjunction of conjunctions. The function *merge* $(a_1 \vee \ldots \vee a_n)$ $(b_1 \vee \ldots \vee b_m)$ shall return a term of the form $a_1 \wedge b_1 \vee a_1 \wedge b_2 \vee \ldots \; a_n \wedge b_m$ that is equivalent to $(a_1 \vee \ldots \vee a_n) \wedge (b_1 \vee \ldots \vee b_m)$.

fun *merge* :: *"bexp ⇒ bexp ⇒ bexp"* **where**

Show that *merge* preserves the semantics and indeed yields a DNF, if its operands are already in DNF. 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 (And a b) s"*

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

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

fun *nnf_to_dnf* :: *"bexp ⇒ bexp"* **where**

Prove the correctness of your function:

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

lemma [*simp*]: *"is_nnf b ⟹ is_dnf (nnf_to_dnf b)"*

Finally, combine the two functions *to_nnf* and *nnf_to_dnf*, to get a function that converts any boolean expression to DNF:

definition *convert_to_dnf* :: *"bexp ⇒ bexp"*

theorem [*simp*]: *"bval (convert_to_dnf b) s = bval b s"*
theorem [*simp*]: *"is_dnf (convert_to_dnf b)"*