Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2021/22
2.11.2021

Semantics of Programming Lectures

Exercise Sheet 2

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

theory *ex02* **imports** *"HOL−IMP.AExp" "HOL−IMP.BExp"* **begin**

Exercise 2.1 Induction

Define a function *deduplicate* that removes duplicate occurrences of subsequent elements from a list.

fun *deduplicate* :: *"'a list ⇒ 'a list"*

The following should evaluate to *True*, for instance:

value *"deduplicate [1,1,2,3,2,2,1::nat] = [1,2,3,2,1]"*

Prove that a deduplicated list has at most the length of the original list:

lemma *"length (deduplicate xs) ≤ length xs"*

Exercise 2.2 Substitution Lemma

A syntactic substitution replaces a variable by an expression.

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a syntactic substitution, i.e., $subst\ x\ a'\ a$ shall be the expression a where every occurrence of variable x has been replaced by expression a'.

fun *subst* :: *"vname ⇒ aexp ⇒ aexp ⇒ aexp"*

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 ⟹ aval (subst x a1 a) s = aval (subst x a2 a) s"*

Exercise 2.3 Arithmetic Expressions With Side-Effects

We want to extend arithmetic expressions by the postfix increment operation *x*++, as known from Java or C++.

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*, i.e., it takes an expression and a state, and returns a value and a new state. Define the function *aval′*.

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 *"<>(x := 0)"*
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 <> = (v, s′) ⟹ 0 ≤ 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 prod.splits*).

Exercise 2.4 Variables of Expression (Time Permitting)

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

fun *vars* :: *"aexp ⇒ vname set"*

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 Models for Boolean Formulas

Submission until Sunday, November 7, 23:59pm.

Consider the following datatype modeling Boolean formulas:

datatype $bexp' = V\ (char\ list)\ |\ And\ bexp'\ bexp'\ |\ Not\ bexp'\ |\ TT\ |\ FF$

Define a function *sat* that decides whether a given assignment (represented as $vname \Rightarrow bool$) satisfies a formula:

fun *sat* :: "$bexp' \Rightarrow assignment \Rightarrow bool$"

Define a function *models* that computes the set of satisfying assignments for a given Boolean formula:

fun *models* :: "$bexp' \Rightarrow assignment\ set$" **where**
"$models\ (V\ x) = \{\sigma.\ \sigma\ x\}$"
| "$models\ TT = UNIV$"

Here $UNIV = \{x.\ True\}$. Fill in the remaining cases! *Hint:* You can use the set operators $-$, $\cap$, $\cup$ for complement/difference, intersection, and union of sets.

Finally prove that a formula is a satisfying assignment for a formula φ iff it is contained in $models\ \varphi$:

theorem *sat_iff_model*: "$sat\ \varphi\ \sigma \longleftrightarrow \sigma \in models\ \varphi$"

Homework 2.2 Simplifying Boolean Formulas

Submission until Sunday, November 7, 23:59pm.

In this exercise, we want to simplify the Boolean formulas defined in the previous exercise by removing the constants *FF* and *TT* from them where possible. We will say that a formula is *simplified* if does not contain a constant or if it is *FF* or *TT* itself:

$simplified\ \varphi = (\varphi = TT \vee \varphi = FF \vee \neg\ has\_const\ \varphi)$

where

$has\_const\ TT = True$

$has\_const\ FF = True$

$has\_const\ (Not\ a) = has\_const\ a$

$has\_const\ (And\ a\ b) = (has\_const\ a \vee has\_const\ b)$

$has\_const\ (V\ v) = False$

Define a function that simplifies Boolean formulas.

fun *simplify* :: "$bexp' \Rightarrow bexp'$"

Example:

value "$simplify\ (And\ (Not\ FF)\ (V\ ''x'')) = V\ ''x''$"

Prove that it produces only simplified formulas.

theorem *simplify_simplified*: *"simplified (simplify φ)"*

Even more importantly, you need to prove that *simplify* does not alter the semantics of the formula:

theorem *simplify_models*: *"models (simplify φ) = models φ"*

Hints: Define non-recursive auxiliary functions to perform the actual simplification! You will need auxiliary lemmas about them.

Note that you can use the induction scheme of a non-recursive *fun* – it will not have an inductive case but still be a case distinction for the function patterns.

Also keep in mind to unfold a definition only when needed. Otherwise it will complicate your proofs.