Technische Universität München
Fakultät für Informatik
Dr. Peter Lammich
Simon Wimmer

WS 2018/19
23.10.2018

Semantics of Programming Languages

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

Recall the definition of *itrev* from the lecture:

```
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
  "itrev [] ys = ys"
| "itrev (x # xs) ys = itrev xs (x # ys)"
```

We already proved the following lemma connecting *itrev* and *rev*:

```
lemma itrev_rev:
  "itrev xs ys = rev xs @ ys"
```

In the last homework, you already defined *fold_right* on lists. Here is a left version of it:

```
fun fold_left :: "('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ 'b" where
  "fold_left _ [] a = a"
| "fold_left f (x # xs) a = fold_left f xs (f x a)"
```

We want to prove that *rev* can also be expressed in terms of *fold_left*:

```
lemma fold_left_rev:
  "fold_left (#) xs [] = rev xs"
```

Find and prove an appropriate theorem that connects *itrev* and *fold_left*, and then use it to prove *fold_left_rev*.

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

```
fun deduplicate :: "'a list ⇒ 'a list" where
```

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* :: *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a syntactic substitution, i.e., *subst x a′ a* shall be the expression *a* where every occurrence of variable *x* has been replaced by expression *a′*.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 2.3 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"* **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 Delta Encoding

Submission until Tuesday, October 30, 10:00am.

We want to encode a list of integers as follows: The first element is unchanged, and every next element only indicates the difference to its predecessor.
For example: (Hint: Use this as test cases for your spec!)

- *enc* [*1*,*2*,*4*,*8*] = [*1*,*1*,*2*,*4*]
- *enc* [*3*,*4*,*5*] = [*3*,*1*,*1*]
- *enc* [*5*] = [*5*]
- *enc* [] = []

Background: This algorithm may be used in lossless data compression, when the difference between two adjacent values is expected to be small, as e.g. in audio data, image data, or sensor data.
It typically requires much less space to store the small deltas than the absolute values.
Disadvantage: If the stream gets corrupted, recovery is only possible when the next absolute value is transmitted. For this reason, in practice, one will submit the current absolute value from time to time. (This is not modeled in this exercise!)

Specify a function to encode a list with delta-encoding. The first argument is used to represent the previous value, and can be initialized to 0.

fun *denc* :: *"int ⇒ int list ⇒ int list"* **where**

Specify the decoder. Again, the first argument represents the previous decoded value, and can be initialized to 0.

fun *ddec* :: *"int ⇒ int list ⇒ int list"* **where**

Show that encoding and then decoding yields the same list. *Hint:* The lemma will need generalization.

lemma *"ddec 0 (denc 0 l) = l"*

Homework 2.2 Boolean Expressions With Equality

Submission until Tuesday, October 30, 10:00am.

Our current version of Boolean expressions is missing an equality operator on arithmetic expressions. In this exercise your task is to define a function *beq* that, given two arithmetic expressions a_1 and a_2, constructs a Boolean expression *beq* a_1 a_2 such that:

lemma
  "bval (beq a1 a2) s ⟷ aval a1 s = aval a2 s"

Prove this property!

Homework 2.3 Models for Boolean Formulas

Submission until Tuesday, October 30, 10:00am.

Consider the following datatype modeling Boolean formulas:

datatype *'a bexp' = V 'a | And "'a bexp'" "'a bexp'" | Not "'a bexp'" | TT | FF*

Define a function *sat* that decides whether a given assignment satisfies a formula:

fun *sat* :: *"'a bexp' ⇒ ('a ⇒ bool) ⇒ bool"*

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

fun *models* :: *"'a bexp' ⇒ ('a ⇒ bool) set"*
where
  "models (V x) = {σ. σ x}"
| *"models TT = UNIV"*
| *"models FF = {}"*

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* φ:

```
lemma
  "sat φ σ ⟷ σ ∈ models φ"
```

Homework 2.4 Simplifying Boolean Formulas (Bonus)

Submission until Tuesday, October 30, 10:00am.

Note: This is a bonus exercise worth three additional points. In the end, the total number of achievable points will be the sum of all the points you can get for all regular exercises. When we calculate the percentage of the total points you reached, we will just add the bonus points on top of the points you got in the regular exercises.

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.

```
fun has_const where
  "has_const TT = True"
| "has_const FF = True"
| "has_const (Not a) = has_const a"
| "has_const (And a b) ⟷ has_const a ∨ has_const b"
| "has_const _ = False"
```

```
definition "simplified φ ⟷ φ = TT ∨ φ = FF ∨ ¬ has_const φ"
```

Define a function *simplify* that simplifies Boolean formulas and prove that it produces only simplified formulas:

```
lemma "simplified (simplify φ)"
```

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

```
lemma "models (simplify φ) = models φ"
```