Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Sascha Böhme, Alexander Krauss

WS 2010/11
8. 11. 2010

Semantics of Programming Languages

Exercise Sheet 3

Note: The following exercises build on the theories about arithmetic and boolean expressions, which you can download from the website. You need the theories `AExp.thy` and `BExp.thy`. Import theory `BExp` in your theory header. `AExp` is then pulled in automatically.

Exercise 3.1 Boolean If expressions

We consider an alternative definition of boolean expressions, which feature a conditional construct:

datatype *ifexp* = *B bool* | *If ifexp ifexp ifexp* | *Less aexp aexp*

(a) Define a function *ifval* analogous to *bval*, which evaluates *ifexp* expressions.

(b) Define a function *translate*, which translates *ifexp*s to *bexp*s. State and prove a lemma showing that the translation is correct.

Exercise 3.2 More Arithmetic Constructs

Extend *aexp* with further arithmetic operators of your choice. Extend *aval* and *asimp* accordingly.

Homework 3 Let expressions

Submission until Wednesday, November 17, 2010, 12:00 (noon).

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

datatype *lexp* = *N nat* | *V name* | *Plus lexp lexp* | *Let name 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.

(a) 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:

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

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

(c) 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.

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 alse try the *sledgehammer* command (from the Isabelle→Commands menu). It invokes an extensive proof search that includes more library lemmas. Since we haven't yet learned the syntax to do manual proofs, this may be useful for glueing facts together automatically. However, it will not help when you need a non-trivial lemma first.