Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Andrei Popescu, Peter Lammich

WS 2012/13
30. 10. 2012

Semantics of Programming Languages

Exercise Sheet 3

Exercise 3.1 Boolean If expressions

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

datatype $ifexp = Bc'\ bool \mid If\ ifexp\ ifexp\ ifexp \mid Less'\ aexp\ aexp$

1. Define a function *ifval* analogous to *bval*, which evaluates *ifexp* expressions.
2. 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 Relational *aval*

Theory *AExp* defines an evaluation function $aval :: aexp \Rightarrow state \Rightarrow val$ for arithmetic expressions. Define a corresponding evaluation relation $is\_aval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$ as an inductive predicate:

inductive is_aval :: "$aexp \Rightarrow state \Rightarrow val \Rightarrow bool$"

Use the introduction rules *is_aval.intros* to prove this example lemma.

lemma "$is\_aval\ (Plus\ (N\ 2)\ (Plus\ (V\ x)\ (N\ 3)))\ s\ (2 + (s\ x + 3))$"

Prove that the evaluation relation *is_aval* agrees with the evaluation function *aval*. Show implications in both directions, and then prove the if-and-only-if form.

lemma *aval1*: "$is\_aval\ a\ s\ v \Longrightarrow aval\ a\ s = v$"
lemma *aval2*: "$aval\ a\ s = v \Longrightarrow is\_aval\ a\ s\ v$"
theorem "$is\_aval\ a\ s\ v \longleftrightarrow aval\ a\ s = v$"

Homework 3.1 Compilation to Register Machine

Submission until Tuesday, November 6, 10:00am.

In this exercise, you will define a compilation function from expressions to register machines and prove that the compilation is correct.

The registers in our simple register machines are natural numbers:

type_synonym *reg* = *nat*

The instructions are:

- "load immediate" an integer value in a register
- load the value of a variable (from the memory state) in a register
- add to a register the value of another register

datatype *instr* = *LDI int reg* | *LD vname reg* | *ADD reg reg*

Recall that a memory state is a function from variable names to integers. A register state will be a function from registers to integers.

Complete the following definition of the function for executing an instruction given a memory state s and a register state σ, the result being a register state. You need to add the cases of the instruction being "load immediate" and "load".

fun *exec* :: "$instr \Rightarrow (vname \Rightarrow int) \Rightarrow (reg \Rightarrow int) \Rightarrow (reg \Rightarrow int)$" **where**
"$exec\ (ADD\ r1\ r2)\ s\ \sigma = \sigma\ (r1 := \sigma\ r1 + \sigma\ r2)$"

Next define the function executing a sequence of register-machine instructions, one at a time. We have already defined for you the case of empty list of instructions. You need to add the recursive case.

fun *execs* :: "$instr\ list \Rightarrow (string \Rightarrow int) \Rightarrow (reg \Rightarrow int) \Rightarrow (reg \Rightarrow int)$" **where**

"$execs\ []\ s\ \sigma = \sigma$" |

We are finally ready for the compilation function. Your task is to define a function *cmp* that takes an arithmetic expression a and a register r and produces a list of register-machine instructions whose execution in any memory state and register state should lead to a register state having in r the value of evaluating a in that memory state.

Here is the intended behavior of *cmp*:

- $cmp\ (N\ n)\ r$ loads immediate n into r
- $cmp\ (V\ x)\ r$ loads x into r
- $cmp\ (Plus\ a\ a1)\ r$ first compiles a placing the result in r, then compiles $a1$ placing the result in $r + 1$, and finally adds the content of $r + 1$ to that of r (storing the result in r).

fun *cmp* :: "$aexp \Rightarrow reg \Rightarrow instr\ list$"

Finally, you need to prove the following correctness lemma, which states that our register-machine compiler is correct, in that executing the compiled instructions of an arithmetic expression yields (in the indicated register) the same result as evaluating the expression.

Hint: For proving correctness, you will need auxiliary lemmas stating that exec commutes with list concatenation and that the instructions produced by $cmp\ a\ r$ do not alter registers below r.

lemma *cmpCorrect*: "$execs\ (cmp\ a\ r)\ s\ \sigma\ r = aval\ a\ s$"

Homework 3.2 No Uninitialized Registers

Submission until Tuesday, November 6, 10:00am.

In this exercise you will prove that the result of compiling an expression is initialization-safe, in that no *ADD* operation is applied to registers that have not been previously initialized by a "load" or "load immediate" instruction.

First we consider the following function *init* that takes a list of register-machine instructions and returns the set of registers that have been initialized in it.

fun *init* :: *"instr list ⇒ reg set"* **where**
"init [] = {}" |
"init (LDI i r # inss) = {r} ∪ init inss" |
"init (LD x r # inss) = {r} ∪ init inss" |
"init (ADD r1 r2 # inss) = init inss"

Notice that the above recursive definition uses nested patterns. Every "fun" definition comes with a customized induction rule that observes its pattern structure: here, the induction rule is called *init.induct*. Use this rule to prove that *init* commutes with list concatenation. Hint: indicate the desired rule to the *induct* method, using *rule*: *init.induct*.

lemma *init_append[simp]*: *"init (inss1 @ inss2) = init inss1 ∪ init inss2"*

Define recursively the predicate *safe* with the following behavior: *safe inss R* holds true iff all the registers that participate in an *ADD* instruction in *inss* either belong to *R* or are previously initialized in *inss*.
Hint: Use a recursive definition on the first argument with the same pattern structure as for the previous function *init*.

fun *safe* :: *"instr list ⇒ reg set ⇒ bool"*

Prove the following commutation lemma. Hint: As before for *init*, use the induction rule customized to the definition of the function.

lemma *safe_append[simp]*:
"safe (inss1 @ inss2) R ⟷ safe inss1 R ∧ safe inss2 (R ∪ init inss1)"

Prove the following initialization-safety property, stating that in a list of instructions resulted from compiling an expression all the added but not previously initialized registers are in the empty set–i.e., there are no such registers.

lemma *initSafe*: *"safe (cmp a r) {}"*

Proof hint: You need to make a more general statement, replacing the empty set with an arbitrary set of registers. You may also need an intermediate lemma about *init* and *cmp*.