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

WS 2015/16
17. 11. 2015

Semantics of Programming Languages

Exercise Sheet 6

Exercise 6.1 A different instruction set architecture

We consider a different instruction set which evaluates boolean expressions on the stack, similar to arithmetic expressions:

- The boolean value *False* is represented by the number *0*, the boolean value *True* is represented by any number not equal to *0*.
- For every boolean operation exists a corresponding instruction which, similar to arithmetic instructions, operates on values on top of the stack.
- The new instruction set introduces a conditional jump which pops the top-most element from the stack and jumps over a given amount of instructions, if the popped value corresponds to *False*, and otherwise goes to the next instruction.

Modify the theory *Compiler* by defining a suitable set of instructions, by adapting the execution model and the compiler and by updating the correctness proof.

Exercise 6.2 While Free Programs

a) Show that while-free programs always terminate, i.e., show that for any while-free command and any state, the big-step semantics yields a result state.

b) Show that non-terminating programs contain a while loop, i.e., show that all commands, for which there is a state such that the big-step semantics yields no result, contain a while loop.

Homework 6.1 Functional Small-Step

Submission until Tuesday, Nov 24, 10:00am.

Specify a functional version of the small-step semantics, and show that it matches the relational version:

fun *fstep* :: *"com ∗ state ⇒ (com ∗ state) option"*
theorem "$(c,s) \to (c',s') \longleftrightarrow$ *fstep* $(c,s) =$ *Some* (c',s')"

Note: A return value of *None* means that there is no step.

Homework 6.2 Left and Right Movers

Submission until Tuesday, Nov 24, 10:00am.

A semaphore is a counter which can be incremented and decremented by parallel processes, however, decrement has to wait until the counter is greater 0. This ensures that the counter is never negative.

Semaphores can be used to synchronize the access of processes to resources.

We model the possible operations on semaphores as follows:

datatype *action* =
  Up vname — Increment
| *Down vname* — Decrement
| *Other* — Unrelated operation

Define the effect of an action on a state. Here, the state holds the values of the semaphores. Assume that other actions do not modify the state.

inductive *exec* :: *"action ⇒ state ⇒ state ⇒ bool"*

Next, we want to develop a scheduler for two processes. The actions of the processes are modeled as lists.

We use a small-step approach, i.e., we define a configuration that contains the remaining actions of the two processes and the current semaphore state:

type_synonym *config* = *"action list × action list × state"*

Then, you have to define a relation *step* such that *step c l c′* means that in c one action is scheduled, and the resulting configuration is c'. The label l indicates the process (1 or 2) and the executed action:

datatype *label* = *P1 action* | *P2 action*

inductive *step* :: *"config ⇒ label ⇒ config ⇒ bool"*

A well-known result on semaphores is that down-operations are right-movers and up-operations are left movers.

Show that down-operations are right-movers, i.e. a down operation on one process can be exchanged with a subsequent operation on the other process. Intuitively, this moves the down-operation to the right in the interleaving sequence.

lemma
  assumes *"step c1 (P1 (Down x)) c2"*
  assumes *"step c2 (P2 a) c3"*
  shows *"∃ ch. step c1 (P2 a) ch ∧ step ch (P1 (Down x)) c3"*

Note: The case where process 2 contains the down-operation is symmetric, and you are not required to prove it.

Show that Up-operations are left-movers, i.e. an up operation on one process can be exchanged with an operation on the other process that comes before it. Intuitively, this moves the up-operation to the left in the interleaving sequence.

lemma
  assumes *"step c1 (P1 a) c2"*
  assumes *"step c2 (P2 (Up x)) c3"*
  shows *"∃ ch. step c1 (P2 (Up x)) ch ∧ step ch (P1 a) c3"*

Hint: With a careful setup, and using *inductive_simps* (look it up in the docs) to generate auxiliary lemmas, there is a one-line proof of these properties. However, an Isar-proof is more structured and may be simpler to develop, so do not invest too much time in finding a short proof.

Homework 6.3 Locking Order

Submission until Tuesday, Nov 24, 10:00am. **5 bonus points, hard!**

Another well-known result is that a locking-order implies deadlock freedom: Assume that there is an ordering on locks, such that a process may only acquire locks which are greater than all locks it has already acquired. Moreover, assume that a process eventually releases all acquired locks. Then, there are no deadlocks.

Note that locks can be simulated by semaphores initialized to 1.

We define well-formed action sequences as follows:

fun *well_formed_aux* :: *"vname set ⇒ action list ⇒ bool"* **where**
  "well_formed_aux A (Down x#l) ⟷ well_formed_aux (insert x A) l ∧ (∀ y∈A. x>y)"
| *"well_formed_aux A (Up x#l) ⟷ well_formed_aux (A−{x}) l ∧ x∈A"*
| *"well_formed_aux A (Other#l) ⟷ well_formed_aux A l"*
| *"well_formed_aux A [] ⟷ A={}"*

abbreviation *"well_formed l ≡ well_formed_aux {} l"*

Note that the additional parameter *A* captures the locks that the process has already acquired. For simplicity, we use the lexicographic ordering on semaphore names as lock ordering. You have to import *~~/src/HOL/Library/List_lexord* and *~~/src/HOL/Library/Char_ord* to get this!

Moreover, we define the initial state, a final state, a deadlocked state, and a step without an explicit label:

abbreviation *init* :: *state* **where** *"init* $\equiv \lambda\_. \ 1$*"* — Initial state
fun *final* **where** *"final* ([],[],_) $\longleftrightarrow$ *True"* | *"final* _ $\longleftrightarrow$ *False"*
definition *"deadlocked c* $\equiv \neg$*final c* $\wedge$ ($\forall$ *c' a.* $\neg$*step c a c'*)*"*
abbreviation *"step' c c'* $\equiv \exists$ *a. step c a c'"*

Your task is to prove that schedules of well-formed action sequences cannot deadlock:

theorem
  assumes *WF*: *"well_formed l1" "well_formed l2"*
  assumes *STEPS*: *"star step'* (*l1*,*l2*,*init*) *c'"*
  shows *"*$\neg$*deadlocked c'"*

Here are some hints on one possible way of proving this: Try to find a suitable invariant on configurations, i.e., a predicate that holds for the initial configuration, and is preserved by a step. Having established such a predicate, you can easily prove that it holds for any reachable configuration:

lemma
  assumes *reachable*: *"star R c0 c'"*
  assumes *initial*: *"I c0"*
  assumes *preserve*: *"*$\bigwedge$*c c'. I c* $\Longrightarrow$ *R c c'* $\Longrightarrow$ *I c'"*
  shows *"I c'"*

The invariant should contain enough information about the configuration and the acquired locks to get through the following (informal) argument:

If a state is stuck, there are two cases: 1) Both processes want to acquire locks (wlog a and b) which are not free. Due to locking order, the locks are held by the respective other process. Again, due to locking order, this implies $a > b$ and $a < b$, which is a contradiction.

2) Another possibility for stuck states is that one process is already finished. However, well-formedness ensures that a finished process has released all its locks.