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

WS 2011/12
25. 11. 2011

Semantics of Programming Languages

Exercise Sheet 6

Exercise 6.1 Small step equivalence

We define an equivalence relation $\approx$ on programs that uses the small-step semantics. Unlike with $\sim$, we also demand that the programs take the same number of steps.

The following relation is the n-steps reduction relation:

inductive
  n_steps :: "*com* $*$ *state* $\Rightarrow$ *nat* $\Rightarrow$ *com* $*$ *state* $\Rightarrow$ *bool*"
  ("_ $\rightarrow$ ^_ _" [60,1000,60]999)
where
  zero_steps: "*cs* $\rightarrow$ ^0 *cs*" |
  one_step: "$cs \rightarrow cs' \Longrightarrow cs' \rightarrow$ ^n $cs'' \Longrightarrow cs \rightarrow$ ^$(Suc\ n)$ cs''"

Prove the following lemmas:

lemma *small_steps_n*: "$cs \rightarrow* cs' \Longrightarrow (\exists n.\ cs \rightarrow$ ^n $cs')$"
lemma *n_small_steps*: "$cs \rightarrow$ ^n $cs' \Longrightarrow cs \rightarrow* cs'$"

The equivalence relation is defined as follows:

definition
  small_step_equiv :: "*com* $\Rightarrow$ *com* $\Rightarrow$ *bool*" (**infix** "$\approx$" 50) **where**
  "$c \approx c'$ == $(\forall s\ t\ n.\ (c,s) \rightarrow$ ^n $(SKIP,\ t) = (c',\ s) \rightarrow$ ^n $(SKIP,\ t))$"

Prove the following lemma:

lemma *small_eqv_implies_big_eqv*: "$c \approx c' \Longrightarrow c \sim c'$"

How about the reverse implication?

Exercise 6.2 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.

Homework 6 Micro-Step Semantics

Submission until Wednesday, December 7, 2011, 12:00 (noon).

In the lectures you have seen big-step and small-step semantics for the IMP language, and how to prove that they are equivalent. In this homework you will formalize a new *micro-step* semantics, and show that it is also equivalent to big-step.

The micro-step semantics relates pairs consisting of a list of commands together with a state. A single micro-step consists of executing the command at the head of the list, just like the small-step semantics—unless that command is a sequence (c_1; c_2). In that case a micro-step consists of putting c_1 and c_2 back onto the list separately, without executing either one.

inductive *micro_step* :: *"com list × state ⇒ com list × state ⇒ bool"* **where**
  ms_skip: *"micro_step (SKIP # l, s) (l, s)"* |
  ms_assign: *"micro_step ((x ::= a) # l, s) (SKIP # l, s(x := aval a s))"* |
  ms_semi: *"micro_step ((*c_1*;* c_2*) # l, s) (*c_1 *#* c_2 *# l, s)"* |
  ms_ift: *"bval b s ⟹ micro_step ((IF b THEN* c_1 *ELSE* c_2*) # l, s) (*c_1 *# l, s)"* |
  ms_iff: *"¬ bval b s ⟹ micro_step ((IF b THEN* c_1 *ELSE* c_2*) # l, s) (*c_2 *# l, s)"* |
  ms_while: *"micro_step ((WHILE b DO c) # l, s)*
    ((IF b THEN c; WHILE b DO c ELSE SKIP) # l, s)"

We define *micro_steps* as an abbreviation for the reflexive, transitive closure of *micro_step*. (Recall that *star* is defined in *Star.thy*.)

abbreviation *"micro_steps ≡ star micro_step"*

Because these are relations on pairs, we will need to generate new induction rules for them using the *split_format* attribute.

lemmas *micro_step_induct =*
  micro_step.induct[split_format(complete)]
lemmas *micro_steps_induct =*
  star.induct [**where** *r=micro_step, split_format(complete)*]

Your assignment is to prove that the micro-step semantics is equivalent to the big-step semantics:

theorem *"micro_steps ([c], s) ([], s′) ⟷ (c, s) ⇒ s′"*

You should prove implications in each direction separately. The following lemma states the right-to-left direction:

lemma *big_step_imp_micro_steps*: "$(c,\ s) \Rightarrow s' \Longrightarrow$ *micro_steps* $([c],\ s)\ ([],\ s')$"

Hint: Proving *big_step_imp_micro_steps* may require additional lemmas; alternatively, you may find that it is easier to prove a generalization of this lemma instead. Also, note that you will probably need to use the rule *star_trans* (from *Star.thy*) in your proof.

To help with the left-to-right direction, we recommend defining a function *seq* that combines a list of commands into a single command. Then you can prove a lemma like *micro_steps_imp_big_step_seq* below:

fun *seq* :: "*com list* $\Rightarrow$ *com*" **where**
"*seq* $[] =$ *SKIP*" |
"*seq* $(c \# l) = (c;$ *seq* $l)$"

lemma *micro_steps_imp_big_step_seq*:
"*micro_steps* $(cs,\ s)\ (cs',\ s') \Longrightarrow \forall t.\ ($*seq* $cs,\ s) \Rightarrow t \longleftrightarrow ($*seq* $cs',\ s') \Rightarrow t$"

Together with *big_step_imp_micro_steps*, you should then be able to prove the final theorem:

theorem "*micro_steps* $([c],\ s)\ ([],\ s') \longleftrightarrow (c,\ s) \Rightarrow s'$"