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

WS 2012/13
20. 11. 2012

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
nsteps :: "*com* ∗ *state* ⇒ *nat* ⇒ *com* ∗ *state* ⇒ *bool*"
("_ →^_ _" [*60*,*1000*,*60*]*999*)
where
zero_steps: "*cs* →^*0* *cs*" |
one_step: "*cs* → *cs*′ ⟹ *cs*′ →^*n* *cs*′′ ⟹ *cs* →^(*Suc n*) *cs*′′"

Prove the following lemmas:

lemma *small_steps_n*: "*cs* →∗ *cs*′ ⟹ (∃ *n*. *cs* →^*n* *cs*′)"
lemma *n_small_steps*: "*cs* →^*n* *cs*′ ⟹ *cs* →∗ *cs*′"
lemma *nsteps_trans*: "*cs* →^*n1* *cs*′ ⟹ *cs*′ →^*n2* *cs*′′ ⟹ *cs* →^(*n1*+*n2*) *cs*′′"

The equivalence relation is defined as follows:

definition
small_step_equiv :: "*com* ⇒ *com* ⇒ *bool*" (**infix** "≈" *50*) **where**
"*c* ≈ *c*′ == (∀ *s t n*. (*c*,*s*) →^*n* (*SKIP*, *t*) = (*c*′, *s*) →^*n* (*SKIP*, *t*))"

Prove the following lemma:

lemma *small_eqv_implies_big_eqv*: "*c* ≈ *c*′ ⟹ *c* ∼ *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 there exists a corresponding instruction which, similarly 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.

end

Homework 6.1 Algebra of Commands

Submission until Tuesday, November 27, 10:00am.

We define an extension of the language with nondeterministic choice (OR) and parallel composition ($\|$), for which we consider the small-step equivalence relation $\approx$ defined in Exercise 6.1. For your convenience, all the necessary notions are (re)defined below. A template file will also be provided for you.

Your task will be to prove various algebraic laws for the small-step equivalence. The most helpful methods will be number induction and/or pair-based rule induction over the *nsteps* relation, using *nsteps_induct* (provided below).

```
datatype
  com =
— sequential part as before —
     | Or com com            (infix "OR" 59)
     | Par com com           (infix "‖" 59)

inductive
  small_step :: "com * state ⇒ com * state ⇒ bool" (infix "→" 55)
where
— sequential part as before —
OrL: "(c1 OR c2,s) → (c1,s)" |
OrR: "(c1 OR c2,s) → (c2,s)" |
ParL: "(c1,s) → (c1',s') ⟹ (c1 ‖ c2,s) → (c1' ‖ c2,s')" |
ParLSkip: "(SKIP ‖ c,s) → (c,s)" |
ParR: "(c2,s) → (c2',s') ⟹ (c1 ‖ c2,s) → (c1 ‖ c2',s')" |
ParRSkip: "(c ‖ SKIP,s) → (c,s)"

inductive
  nsteps :: "com * state ⇒ nat ⇒ com * state ⇒ bool"
  ("_ →^_ _" [60,1000,60]999)
where
  zero_steps[simp,intro]: "cs →^0 cs" |
  one_step[intro]: "cs → cs' ⟹ cs' →^n cs'' ⟹ cs →^(Suc n) cs''"

lemmas nsteps_induct = nsteps.induct[split_format(complete)]


definition
  small_step_equiv :: "com ⇒ com ⇒ bool" (infix "≈" 50) where
  "c ≈ c' ≡ (∀s t n. (c,s) →^n (SKIP, t)  ⟷  (c', s) →^n (SKIP, t))"
```

As a demo, we prove that OR is commutative (w.r.t. $\approx$). The proof here goes in two steps: first lemma *Or_commute_n*, then the desired fact *Or_commute* by simply unfolding the definition.

```
lemma Or_commute_n: "(c OR d, s) →^n (SKIP, t) ⟹ (d OR c, s) →^n (SKIP, t)"
by (induct n arbitrary: c d) (fastforce intro: one_step OrL OrR)+
```

lemma *Or_commute*: "c *OR* $d \approx d$ *OR* c"
unfolding *small_step_equiv_def* **using** *Or_commute_n* **by** *blast*

Now it's your turn to prove commutativity and associativity of $\|$. You are free to do either automatic or Isar proofs.

lemma *Par_commute*: "$c \parallel d \approx d \parallel c$"

lemma *Par_assoc*: "$(c \parallel d) \parallel e \approx c \parallel (d \parallel e)$"

The last task of this exercise is to prove distributivity of ; over *Or*, namely, lemma *Seq_Or_distrib* below. This will be harder then the other proofs, and therefore we provide some guidelines.

First, you should prove the following inversion rules for *Or* and ; w.r.t. *nsteps*. (Most likely you will need an Isar proof for the second.)

lemma *Or_nsteps_invert*:
assumes "$(c$ *OR* $d,\ s) \rightarrow\hat{}n\ (SKIP,\ t)$"
shows "$\exists\ n1.\ n = Suc\ n1 \wedge ((c,s) \rightarrow\hat{}n1\ (SKIP,t) \vee (d,\ s) \rightarrow\hat{}n1\ (SKIP,\ t))$"

lemma *Seq_nsteps_invert*:
assumes "$(c\ ;\ d,\ s) \rightarrow\hat{}n\ (SKIP,\ t)$"
shows "$\exists\ n1\ n2\ s1.\ n = Suc\ (n1 + n2) \wedge (c,s) \rightarrow\hat{}n1\ (SKIP,s1) \wedge (d,\ s1) \rightarrow\hat{}n2\ (SKIP,\ t)$"

Next, we put the above rules in a nicer elimination format:

lemma *Or_nsteps_elim*[*elim*]:
assumes "$(c$ *OR* $d,\ s) \rightarrow\hat{}n\ (SKIP,\ t)$"
and "$\bigwedge\ n1.\ [\![n = Suc\ n1;\ (c,s) \rightarrow\hat{}n1\ (SKIP,t)]\!] \Longrightarrow P$"
and "$\bigwedge\ n1.\ [\![n = Suc\ n1;\ (d,s) \rightarrow\hat{}n1\ (SKIP,t)]\!] \Longrightarrow P$"
shows P
using *assms Or_nsteps_invert* **by** *blast*

lemma *Seq_nsteps_elim*[*elim*]:
assumes "$(c\ ;\ d,\ s) \rightarrow\hat{}n\ (SKIP,\ t)$" **and**
"$\bigwedge\ n1\ n2\ s1.\ [\![n = Suc\ (n1 + n2);\ (c,s) \rightarrow\hat{}n1\ (SKIP,s1);\ (d,s1) \rightarrow\hat{}n2\ (SKIP,\ t)]\!] \Longrightarrow P$"
shows P
using *assms Seq_nsteps_invert* **by** *blast*

Now, you should prove introduction rules for *Or* and ; w.r.t. *nsteps*:

lemma *Or_nsteps_introL*[*intro*]:
assumes "$(c,s) \rightarrow\hat{}n\ (SKIP,t)$" **shows** "$(c$ *OR* $d,\ s) \rightarrow\hat{}(Suc\ n)\ (SKIP,t)$"

lemma *Or_nsteps_introR*[*intro*]:
assumes "$(d,s) \rightarrow\hat{}n\ (SKIP,t)$" **shows** "$(c$ *OR* $d,\ s) \rightarrow\hat{}(Suc\ n)\ (SKIP,t)$"

lemma *Seq_nsteps_intro*[*intro*]:
assumes *1*: "$(c,s) \rightarrow\hat{}n1\ (SKIP,s1)$" **and** *2*: "$(d,s1) \rightarrow\hat{}n2\ (SKIP,\ t)$"
shows "$(c\ ;\ d,\ s) \rightarrow\hat{}(Suc\ (n1 + n2))\ (SKIP,\ t)$"

Hint for the proof of *Seq_nsteps_intro*: Follow a similar route to the proof of the corresponding fact about →∗ from theory *Small_Step*, namely, *seq_comp*. Lemma *nsteps_trans* from Exercise 6.1 is also needed.

Finally, you can prove the desired distributivity law. Hint: If a fully automatic proof does not work, try an Isar proof of the two implications emerging from ⟷ by applying the correct introduction/elimination rules by hand.

lemma *Seq_Or_distrib_n*:
"(c ; (d OR e), s) → ^n (SKIP, t) ⟷ ((c ; d) OR (c ; e), s) → ^n (SKIP, t)"

lemma *Seq_Or_distrib*: *"c ; (d OR e) ≈ (c ; d) OR (c ; e)"*

Homework 6.2 Powerset Construction

Submission until Tuesday, November 27, 10:00am.

Note: *This is a "bonus" exercise worth 5+3 additional points, making the maximum possible score for all homework on this sheet 18 out of 10 points. You'll get 5 points for proving the lemmas, and additional 3 points for aesthetics of your proof, i.e., a confusing apply-style script that somehow manages to prove the theorems is worth 5 points, while a nice Isar-proof that makes clear the structure of the proof is worth 8 points.*

Reconsider the finite state machines (FSMs) from Homework 4.

type_synonym $('Q,'\Sigma)$ *LTS* = "$('Q\times'\Sigma\times'Q)$ *set*"
inductive *accept* :: "$'Q$ *set* $\Rightarrow$ $('Q,'\Sigma)$ *LTS* $\Rightarrow$ $'Q$ $\Rightarrow$ $'\Sigma$ *list* $\Rightarrow$ *bool*"
  for F δ **where**
  base: "$q\in F \Longrightarrow$ *accept* F δ q []"
| *step*[*trans*]: "$[\![\ (q,a,q')\in\delta;$ *accept* F δ q' w $]\!] \Longrightarrow$ *accept* F δ q $(a\#w)$"

In this exercise, you shall define the well-known powerset construction, that converts any finite state machine to a deterministic one.

First define the transition relation and the set of accepting states of the powerset-FSM:

definition *pow_δ* :: "$('Q,'\Sigma)$ *LTS* $\Rightarrow$ $('Q$ *set*$,'\Sigma)$ *LTS*"
definition *pow_F* :: "$'Q$ *set* $\Rightarrow$ $'Q$ *set set*"

Then prove that the transition relation of the powerset-FSM is deterministic. (Note: If you got your definitions right, this proof is a one-liner, and requires no elaborate Isar-proof!)

lemma *pow_δ_det*: "$[\![(q,a,q')\in$*pow_δ* δ; $(q,a,q'')\in$*pow_δ* $\delta]\!] \Longrightarrow q'=q''$"

Finally prove that the powerset construction does not change the words accepted by a state. (Note: It's best (really!) to elaborate this proof on paper first, and then convert it into an Isar-proof. You should prove both directions separately, and you will need to generalize the statement in order to get the induction through.)

theorem *pow_correct*:
  "*accept* F δ q w $\longleftrightarrow$ *accept* (*pow_F* F) (*pow_δ* δ) $\{q\}$ w"