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:

\[
\text{inductive} \quad \text{nsteps} :: \text{"com * state } \Rightarrow \text{nat } \Rightarrow \text{com * state } \Rightarrow \text{bool"}
\]

where

\[
\text{zero steps: } \text{"cs } \rightarrow \hat{0} \text{ cs" |}
\]

\[
\text{one step: } \text{"cs } \rightarrow \text{cs}' \Rightarrow \text{cs}' \rightarrow \hat{n} \text{ cs'' } \Rightarrow \text{cs } \rightarrow \hat{(Suc n)} \text{ cs''"}
\]

Prove the following lemmas:

\[
\text{lemma small_steps_n: } \text{"cs } \rightarrow^{*} \text{ cs' } \Rightarrow (\exists n. \text{ cs } \rightarrow \hat{n} \text{ cs'"})
\]

\[
\text{lemma n_small_steps: } \text{"cs } \rightarrow \hat{n} \text{ cs' } \Rightarrow \text{cs } \rightarrow^{*} \text{ cs'"}
\]

\[
\text{lemma nsteps_trans: } \text{"cs } \rightarrow \hat{n_1} \text{ cs' } \Rightarrow \text{cs' } \rightarrow \hat{n_2} \text{ cs'' } \Rightarrow \text{cs } \rightarrow \hat{(n_1+n_2)} \text{ cs''"}
\]

The equivalence relation is defined as follows:

\[
\text{definition} \quad \text{small_step_equiv} :: \text{"com } \Rightarrow \text{com } \Rightarrow \text{bool" (infix "\approx" 50)} \text{ where}
\]

\[
\text{"c } \approx \text{c'} \equiv (\forall s \ T \ n. \ (c,s) \rightarrow \hat{n} \ (SKIP, t) = (c', s) \rightarrow \hat{n} \ (SKIP, t))"
\]

Prove the following lemma:

\[
\text{lemma small_equiv_implies_big_equiv: } \text{"c } \approx \text{c'} \Rightarrow c \sim \text{c'}"\]

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 ≈ 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”
        (“→” ˆ “” [60,1000,60]999)
where
  zero_steps[simp,intro]: “cs → ˆ0 cs” |
  one_step[intro]: “cs → cs’ ⇒ cs’ → ˆn cs” ⇒ “cs → ˆn (Suc n) cs’”

lemmas
  nsteps_induct = nsteps_induct[split_format(complete)]

definition
  small_step_eqiv :: “com ⇒ com ⇒ bool” (infix “≈” 50) where
  “c ≈ c’ ≡ (∀ s t n. (c,s) → ˆn (SKIP, t) ↔ (c’, s) → ˆn (SKIP, t))”

As a demo, we prove that OR is commutative (w.r.t. ≈). 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) → ˆn (SKIP, t) ⇒ (d OR c, s) → ˆn (SKIP, t)”
by (induct n arbitrary: c d) (fastforce intro: one_step OrL OrR)
lemma Or_commute: “c OR d ≡ 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 || d ≡ d || c”
lemma Par_assoc: “(c || d) || e ≡ c || (d || 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) → (∀ n. SKIP, t)”
shows “∃ n1. n = Suc n1 ∧ ((c,s) → n1 (SKIP,t)) ∨ (d,s) → n1 (SKIP, t)”

lemma Seq_nsteps_invert:
assumes “(c ; d, s) → (∀ n. SKIP, t)”
shows “∃ n1 n2 s1. n = Suc (n1 + n2) ∧ (c,s) → n1 (SKIP,s1) ∧ (d, s1) → n2 (SKIP, t)”

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

lemma Or_nsteps_elim[elim]:
assumes “(c OR d, s) → (∀ n. SKIP, t)”
and “(n1, (n = Suc n1; (c,s) → n1 (SKIP,t)))”
and “(n2, (d,s) → n2 (SKIP, t))”

lemma Seq_nsteps_elim[elim]:
assumes “(c ; d, s) → (∀ n. SKIP, t)” and
“(n1 n2 s1. n = Suc (n1 + n2); (c,s) → n1 (SKIP,s1); (d,s1) → n2 (SKIP, t))”

uses P
show P
using assms Seq_nsteps_invert by blast

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

lemma Or_nsteps_introI[intro]:
assumes “(c,s) → (∀ n. SKIP, t)” shows “(c OR d, s) → (∀ n. SKIP, t)”

lemma Or_nsteps_introR[intro]:
assumes “(d,s) → (∀ n. SKIP, t)” shows “(c OR d, s) → (∀ n. SKIP, t)”

lemma Seq_nsteps_intro[intro]:
assumes 1: “(c,s) → (∀ n. SKIP,s1)” and 2: “(d,s1) → (∀ n2 (SKIP, t))”
shows “(c ; d, s) → (∀ n1 + n2) (SKIP, t)”
Hint for the proof of `Seq.nsteps_intro`: Follow a similar route to the proof of the corresponding fact about $\rightarrow^*$ 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 $\leftrightarrow$ by applying the correct introduction/elimination rules by hand.

**lemma Seq.Or_distrib_n:**

```
"(c ; (d OR e), s) \rightarrow^* (SKIP, t) \leftrightarrow ((c ; d) OR (c ; e), s) \rightarrow^* (SKIP, t)"
```

**lemma Seq.Or_distrib:**

```
"c ; (d OR e) \approx (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, 'Σ) LTS = "('Q × 'Σ × 'Q) set"
inductive accept :: "'Q set ⇒ ('Q, 'Σ) LTS ⇒ 'Q ⇒ 'Σ list ⇒ bool"
  for F δ where
  base: "q ∈ F ⇒ accept F δ q []"
  | step[trans]: "[(q, a, q′) ∈ δ; accept F δ q′ w] ⇒ 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, 'Σ) LTS ⇒ ('Q set, 'Σ) LTS"
definition pow_F :: "'Q set ⇒ '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′) ∈ pow_δ δ; (q, a, q′′) ∈ pow_δ δ] ⇒ 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 ↔ accept (pow_F F) (pow_δ δ) {q} w"