Concrete Semantics with Isabelle/HOL

Exercise Sheet 6

**Exercise 6.1** Deskip

Define a recursive function

```isabelle
fun deskip :: "com ⇒ com"
```

that eliminates as many `SKIP`s as possible from a command. For example:

```isabelle
deskip (SKIP;; WHILE b DO (x ::= a;; SKIP)) = WHILE b DO x ::= a
```

Prove its correctness by induction on `c`:

```isabelle
lemma "deskip c ∼ c"
```

Remember lemma `sim_while_cong` for the `WHILE` case.

**Exercise 6.2** Nondeterminism

Go back to Exercise 5.2 to finish it, including point 5: Adapt the small step semantics, and the equivalence proof of big and small step semantics.

**Homework 6.1** A Register Machine (8 points)

Submission until Monday 15.06.2015 10:02 AM. Please send your submissions to jasmin.blanchette@mpi-inf.mpg.de and mathias.fleury@mpi-inf.mpg.de with “[CONCRETE6]” in the subject of your email. If your name is Voornaam van Achternaam (which would be kind of cool), please call your theory file `Van_Achternaam_Voornaam.thy`.

The theory file `~/src/HOL/IMP/Compiler.thy` starts by defining a stack-machine instruction language. In this exercise, we start looking at an alternative model: a register machine, with infinitely many registers. Real register machines have a small finite number of registers, e.g. 16 or 32, and must spill into memory if more intermediate values are required, but we abstract from this aspect here.
The theory file starts with some basic infrastructure, which we also copy without asking ourselves too many questions or emitting grumpy comments:

```plaintext
declare [coercion_enabled]
declare [coercion "int :: nat ⇒ int"]
```

```plaintext
fun inth :: "'a list ⇒ int ⇒ 'a" (infixl "!!") 100 where
  "(x # xs) !! i = (if i = 0 then x else xs !! (i - 1))"
```

```plaintext
lemma inth_append [simp]:
  "0 ≤ i ⇒ (xs @ ys) !! i = (if i < size xs then xs !! i else ys !! (i - size xs))"
by (induction xs arbitrary: i) (auto simp: algebra_simps)
```

```plaintext
abbreviation (output)
  "isize xs == int (length xs)"
```

```plaintext
notation isize ("size")
```

We also specify the following option to disable annoying warnings about our overloading of the symbol ⇒, which is used both for the case expression syntax and for the big-step semantics:

```plaintext
declare [syntax_ambiguity_warning = false]
```

(Nipkow & Klein seem to be perfectly able to live with those warnings, but Blanchette & Fleury, not to mention Wand, are annoyed by them.)

Then comes the specification of our instruction set. The constructor names are as before, but their arguments are different.

```plaintext
type_synonym reg = nat

datatype instr =
  LOADI int reg — puts an integer into a register
| LOAD vname reg — puts the content of a variable into a register
| ADD reg reg reg — adds the contents of two registers and put the result in a 3rd one
| STORE reg vname — puts the content of a register into a variable
| JMP int — jump unconditionally by offset
| JMPLESS reg reg int — jump by offset if the 1st register’s content is less than the 2nd’s
| JMPGE reg reg int — ditto but for “greater than or equal to”
```

Instead of a stack, we have a “register state”, which maps each register to its value:

```plaintext
type_synonym reg_state = "reg ⇒ val"

type_synonym config = "int × state × reg_state"
```

**Task A:** Adapt the specification of the execution functions from Compiler.thy to reflect the new instruction set.

```plaintext
fun iexec :: "instr ⇒ config ⇒ config"
definition
```
exec1 :: “instr list ⇒ config ⇒ config ⇒ bool” ("_/ ⊢ (_. →/ _.)") [59,0,59] 60

abbreviation
exec :: “instr list ⇒ config ⇒ config ⇒ bool” ("_/ ⊢ (_. →*/ _.)") 50

where
"exec P ≡ star (exec1 P)"

Task B: Continue with the basic lemmas about the new instruction set.

lemma exec1I [intro]
declare star.step [intro]

lemmas exec_induct = star_induct [of "exec1 P", split_format(complete)]

lemma iexec_shift [simp]
lemma execI_appendR
lemma execI_appendL
lemma execI_appendL
lemma execI_Cons1 [intro]
lemma execI_appendL.if [intro]
lemma execI_appendL.trans [intro]

Task C: Define compilation of arithmetic and Boolean expressions. You will need to find a way to ensure that evaluating a subexpression does not overwrite intermediate results for another. Various strategies are possible. Try to find a simple strategy to keep eventual proofs simple.

Task D: Answer the following question: Is it possible to implement acomp and bcomp on a register machine with a fixed number of registers, without spilling to memory? Justify your answer (informally).

Task E: Formulate correctness statements about acomp and bcomp. The statements should be strong enough that they capture any invariant on which your recursion relies. In particular, if your version of acomp does not modify certain registers, it might be necessary to capture this information in the correctness condition.

Unusually for this course, you are not asked to prove the lemmas for Task E. Simple sorrys suffice. To compensate for this lack of rigor, give an informal justification of why the lemmas should hold. (The more detailed, the better.)

Homework 6.2  GCL à la Sauce Small-Step (8 points)

Submission until Monday 15.06.2015 10:02 AM. Please send your submissions to jasmin.blanchette@mpi-inf.mpg.de and mathias.fleury@mpi-inf.mpg.de with “[CONCRETE6]” in the subject of your email. If your name is Voornaam van Achternaam (which would be kind of cool), please call your theory file Van_Achternaam_Voornaam.thy.
In last week’s homework, we had our dose of guarded command language (GCL), or so we thought. GCL is on the menu again, this time servi à la sauce aux petits pois (pas)—i.e., small-step-style. Recall the type of guarded commands:

```plaintext
datatype gcom =
  Skip | Ass vname aexp — American English
  Sq gcom gcom
  IfBlock “(bexp × gcom) list”
```

**Step A:** Define a small-step semantics:

```plaintext
inductive small stepg :: “gcom × state ⇒ gcom × state ⇒ bool” (infix “→g” 55)
abbreviation small stepgs :: “gcom × state ⇒ gcom × state ⇒ bool” (infix “→g×” 55)
where “x →g y ⇔ star small stepg x y”
lemmas small stepg induct = small stepg.induct[split_format(complete)]
```

(You can still use the big-step semantics for evaluating expressions.)

**Step B:** Derive the elimination rules corresponding to the gcom constructors. Which attribute can you put on the rule (e.g., [simp])? Justify briefly.

**Step C:** Recall the big-step semantics from last week:

```plaintext
inductive big stepg :: “gcom × state ⇒ state ⇒ bool” (infix “⇒g” 55)
where
  Skip: “(Skip, s) ⇒g s”
  Ass: “(Ass x a, s) ⇒g s(x := aval a s)”
  Sq: “(c, s) ⇒g s’ ⇒ (c’, s’) ⇒g s”′ ⇒ (Sq c c’, s) ⇒g s”
  IfBlock: “(b, c) ∈ set Gs ⇒ bval b s ⇒ (c, s) ⇒g s’ ⇒ (IfBlock Gs, s) ⇒g s’”
```

declare big stepg.intros [intro, simp]
lemmas big stepg induct = big stepg.induct[split_format(complete)]

**Theorem big iff small:** “cs ⇒g t = cs →g× (Skip, t)”
Homework 6.3  A Complete Evaluation Function for GCL? (4 points)

Submission until Monday 15.06.2015 10:02 AM. Please send your submissions to jasmin.blanchette@mpi-inf.mpg.de and mathias.fleury@mpi-inf.mpg.de with “[CONCRETE6]” in the subject of your email. If your name is Voornaam van Achternaam (which would be kind of cool), please call your theory file Van_Achternaam_Voornaam.thy.

Last week, a student suggested the following improved version of big_stepgf, which he called big_stepgfc:

```haskell
fun big_stepgfc :: "gcom ⇒ state ⇒ state option" where
  "big_stepgfc Skip s = Some s"
| "big_stepgfc (Ass x e) s = Some (s(x := aval e s))"
| "big_stepgfc (S zug c1 c2) s =
  (case (big_stepgfc c1 s) of
   None ⇒ None
   | Some s' ⇒ big_stepgfc c2 s')"
| "big_stepgfc (IfBlock []) s = None"
| "big_stepgfc (IfBlock ((e, c) # gr)) s =
  (if bval e s ∧ big_stepgfc c s ≠ None then
   big_stepgfc c s
  else
   big_stepgfc (IfBlock gr) s)"
```

He wrote:

> I postulate that it is complete, but leave the proof as a challenge to the instructors ;-)

The instructors ain’t any dummer than him and leave the proof or disproof as a challenge to all students.