COMP4161 S2/2017
Advanced Topics in Software Verification

Assignment 2

This assignment starts on Monday, 2017-09-4 and is due on Monday, 2017-09-18, 23:59h. We will accept Isabelle .thy files only. In addition to this pdf document, please refer to the provided Isabelle template for the definitions and lemma statements.

The assignment is take-home. This does NOT mean you can work in groups. Each submission is personal. For more information, see the plagiarism policy: https://student.unsw.edu.au/plagiarism

Submit using `give` on a CSE machine: `give cs4161 a2 a2.thy`

For all questions, you may prove your own helper lemmas, and you may use lemmas proved earlier in other questions. If you can't finish an earlier proof, use *sorry* to assume that the result holds so that you can use it if you wish in a later proof. You won't be penalised in the later proof for using an earlier *true* result you are yet to prove, and you'll be awarded part marks for the earlier question in accordance with the progress you made on it.

1 Higher-Order Logic (16 marks)

Prove each of the following statements, using only the proof methods: `rule`, `erule`, `assumption`, `frule`, `drule`, `rule_tac`, `erule_tac`, `frule_tac`, `drule_tac`, `rename_tac`, and `cases_tac`; and using only the proof rules: `impI`, `impE`, `conjI`, `conjE`, `disjI1`, `disjI2`, `disjE`, `notI`, `notE`, `iffI`, `iffE`, `iffD1`, `iffD2`, `ccontr`, `classical`, `FalseE`, `TrueI`, `conjunct1`, `conjunct2`, `allI`, `allE`, `exI`, `exE`, `spec`, and `mp`. You do not need to use all of these methods and rules. You may use rules proved in earlier parts of the question when proving later parts.

(a) $(\neg\ (\forall x.\ P\ x)) = (\exists x.\ \neg\ P\ x)$ (3 marks)

(b) $(\forall x.\ P \longrightarrow Q\ x) = (P \longrightarrow (\forall x.\ Q\ x))$ (3 marks)

(c) $(\forall x.\ P\ x \wedge Q\ x) = ((\forall x.\ P\ x) \wedge (\forall x.\ Q\ x))$ (3 marks)

(d) $\forall x.\ \neg\ R\ x \longrightarrow R\ (M\ x) \Longrightarrow \forall x.\ R\ x \vee R\ (M\ x)$ (3 marks)

(e) $[\![\forall x.\ \neg\ R\ x \longrightarrow R\ (M\ x);\ \exists x.\ R\ x]\!] \Longrightarrow \exists x.\ R\ x \wedge R\ (M\ (M\ x))$
(4 marks)

2 List Datatype (14 marks)

Consider a datatype *'a list2* that is similar to the usual list datatype, *'a list*, except it allows you to add an element not only to the front, but also to the end of a list.

(a) Define the datatype *'a list2*. (2 marks)

(b) Write a function *list-of-list2* that converts an *'a list2* back into an *'a list*. (2 marks)

(c) Define a function *swap-cons* that swaps the two non-nil constructors (the ones that add an element to the front and to the back) in an *'a list2*. (2 marks)

(d) Write a lemma stating that *swap-cons* reverses the list represented by an *'a list2*, and prove it. (3 marks)

(e) Define a function *app2* that appends two *'a list2*s. (2 marks)

(f) Prove the correctness of *app2*. (3 marks)

3 Normal Forms for Propositional Formulae (29 marks)

Consider the following datatype *fml* of formulae of propositional logic:

datatype *fml* =
  Var pvar
| *Neg fml*
| *Conj fml fml*
| *Disj fml fml*

where *pvar* denotes variables ranging over values of type *bool*. The function *eval-fml* computes the value of a formula for a given state (i.e., a valuation of variables).

fun *eval-fml* :: *fml* ⇒ *valuation* ⇒ *bool* **where**
  $\textit{eval-fml}\ (\textit{Var}\ v)\ s = s\ v$
| $\textit{eval-fml}\ (\textit{Neg}\ p)\ s = (\neg(\textit{eval-fml}\ p\ s))$
| $\textit{eval-fml}\ (\textit{Conj}\ p\ q)\ s = (\textit{eval-fml}\ p\ s \wedge \textit{eval-fml}\ q\ s)$
| $\textit{eval-fml}\ (\textit{Disj}\ p\ q)\ s = (\textit{eval-fml}\ p\ s \vee \textit{eval-fml}\ q\ s)$

A formula is in negation normal form (NNF) when negation is only applied to variables. We now want to define a function *nnf* that converts a formula into NNF.

function *nnf* :: *fml* ⇒ *fml* **where**
nnf (*Var x*) = *Var x* |
nnf (*Neg* (*Neg p*)) = *nnf*(*p*) |
nnf (*Neg* (*Var x*)) = *Neg* (*Var x*) |
nnf (*Neg* (*Conj p q*)) = *nnf* (*Disj* (*Neg p*) (*Neg q*)) |
nnf (*Neg* (*Disj p q*)) = *nnf* (*Conj* (*Neg p*) (*Neg q*)) |
nnf (*Conj p q*) = *Conj* (*nnf p*) (*nnf q*) |
nnf (*Disj p q*) = *Disj* (*nnf p*) (*nnf q*)

Here we need to use Isabelle's **function** command and manually prove that the computation of *nnf* does terminate. To prove termination, we need to define a measure on *fml* that decreases during the computation of *nnf*.

termination *nnf*
  apply (*relation inv-image* (*less-than*) (λ*t*. (*neg-depth t*)))

(a) Define a predicate *is-nnf* that tests if a formula is in NNF. (2 marks)

(b) Complete the definition of the function *nnf* by proving its termination. (6 marks)

(c) Prove that *nnf* is correct, i.e., it returns a formula in NNF and it preserves the value of a formula. (6 marks)

A formula is in conjunctive normal form (CNF) if it is in NNF and is a series of conjunctions of subformlae that have no conjunctions within them.

(d) Define a function *is-cnf* that tests if a formula is in CNF. (3 marks)

(e) Define a function *nnf-to-cnf* that converts a formula in NNF into CNF. (4 marks)

(f) Prove that *nnf-to-cnf* correctly transforms a formula in NNF into CNF and that it preserves its value. (8 marks)

4 Sublists (16 marks)

A sublist of a list *ls* is a list containing only the elements of *ls* in the same order as in *ls*. For example, for a list *ls*=[*1*,*2*,*3*], the sublists of *ls* are the following lists: [], [*1*], [*2*], [*3*], [*1*,*2*], [*1*,*3*], [*2*,*3*], [*1*,*2*,*3*].

(a) Define an inductive predicate *is-sublist ls xs* which holds when *xs* is a sublist of *ls*. Demonstrate the correctness of your definition using examples. (4 marks)

(b) Define a function *sublist-fun* which returns a list of all the sublists of a list. (2 marks)

(c) Prove that *is-sublist ls xs* if and only if $xs \in set\ (sublist\text{-}fun\ ls)$. (10 marks)

5 Operational Semantics of IMP (25 marks)

Consider the following simple imperative language IMP, with a skip statement, assignment of variables to arithmetic expressions, sequencing, conditionals ("if-then-else") and while loops. Variables can only be of type *nat*; their names are just strings. The state of a program is a valuation of all the variables (i.e., a mapping from variable names to their current value). The syntax of arithmetic expressions and Boolean expressions is left unspecified: they are represented by functions that take a state as parameter, to get the values of the variables, and return the result of the expression.

```
type-synonym vname = string
type-synonym state = vname ⇒ nat
type-synonym aexp  = state ⇒ nat
type-synonym bexp  = state ⇒ bool

datatype
  com = SKIP
    | Assign vname aexp      (- ::= - [1000,61] 61)
    | Seq   com com          (-;; - [60, 61] 50)
    | If    bexp com com     (IF - THEN - ELSE - FI [0,0,61] 61)
    | While bexp com         (WHILE - DO - OD [0,45] 61)

type-synonym config = com × state
```

In order to make the programs more readable, we introduce some syntax:

- the term *Assign x a* can be written as *x ::= a*,
- the term *Seq c1 c2* as *c1;; c2*,
- the term *If b c1 c2* as *IF b THEN c1 ELSE c2 FI*, and
- the while loop *While b c* as *WHILE b DO c OD*.

We now consider the *semantics* of this language, i.e. the *meaning* of a program. *Relational* big-step semantics of a language can be defined as a ternary relation between the program *c*, the initial state *s*, and the final state s' obtained by executing c from s. We define our big-step semantics as an inductive relation *big-step*, denoted by $(c,s) \Rightarrow s'$, on configurations and final states, where a configuration is a pair of a program and an initial state.

$$\frac{}{(SKIP,\ s) \Rightarrow s}\ \texttt{Skip} \qquad \frac{}{(x ::= a,\ s) \Rightarrow s(x := a\ s)}\ \texttt{Assign}$$

$$\frac{(c_1,\ s) \Rightarrow s'' \qquad (c_2,\ s'') \Rightarrow s'}{(c_1;;\ c_2,\ s) \Rightarrow s'}\ \texttt{Seq}$$

$$\frac{b\ s \qquad (c_1,\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2\ FI,\ s) \Rightarrow s'}\ \texttt{IfTrue}$$

$$\frac{\neg\ b\ s \qquad (c_2,\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2\ FI,\ s) \Rightarrow s'}\ \texttt{IfFalse}$$

$$\frac{\neg\ b\ s}{(WHILE\ b\ DO\ c\ OD,\ s) \Rightarrow s}\ \texttt{WhileFalse}$$

$$\frac{b\ s \qquad (c,\ s) \Rightarrow s'' \qquad (WHILE\ b\ DO\ c\ OD,\ s'') \Rightarrow s'}{(WHILE\ b\ DO\ c\ OD,\ s) \Rightarrow s'}\ \texttt{WhileTrue}$$

Next we consider defining an *interpreter* function *cval* that takes an IMP program c and an initial state s, as well as a clock t, and returns either *None*, if the program runs out of time, or *Some* $(s',\ t')$, if program execution terminates in a final state s' with remaining clock t'. (We need the clock to ensure the interpreter terminates, since all HOL definitions must be total.)

```
fun cval :: com ⇒ state ⇒ nat ⇒ (state × nat) option
where
  cval SKIP s t = Some (s,t)
| cval (x ::= a) s t = Some (s (x := a s),t)
| cval (c1 ;; c2) s t =
    (case (cval c1 s t) of
        None          ⇒ None
      | Some (s2,t2) ⇒ (cval c2 s2 (if t < t2 then t else t2)))
| cval (IF b THEN c1 ELSE c2 FI) s t =
     cval (if b s then c1 else c2) s t
| cval (WHILE b DO c OD) s t =
    (if b s
     then
       (if t = 0 then None else cval (Seq c (WHILE b DO c OD)) s (t − 1))
     else Some (s,t))
```

The interpreter *cval* can be said to define a *functional* big-step semantics for IMP. Now let us prove equivalence between the relational big-step semantics and the functional big-step semantics.

(a) Prove that *big-step* is deterministic (3 marks).

(b) Prove that increasing the clock (allowing more time) for a terminating execution will only increase the final clock by the same amount. (3 marks)

(c) Prove that execution of a terminating program always decreases the clock. (3 marks)

(d) Prove that the relational semantics implies the functional semantics, i.e., executions in the relational semantics can be simulated by the functional semantics. (6 marks)

(e) Prove that the functional semantics is contained in the relational semantics. Prove the equivalence of the two semantics as a corollary. (10 marks)