COMP4161 S2/2016
Advanced Topics in Software Verification

Assignment 2

This assignment starts on Monday, 2016-09-12 and is due on Thursday, 2016-09-22, 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 were unable to prove, and you'll be awarded part marks for the earlier question in accordance with the progress you made on it.

Introduction

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* (- :== - [60,60] 60)
| *Semi com com* (-;; - [50, 51] 50)
| *Cond bexp com com* (*IF* - *THEN* - *ELSE* - *FI* [0,0,59] 60)
| *While bexp com* (*WHILE* - *DO* - *OD* [0,45] 60)

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 *Semi 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 define the *semantics* of the language, i.e. the *meaning* of a program. This is defined as the output state s' of the program c when executed from an input state s, denoted $(c,s) \Rightarrow s'$.

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

$$\frac{(c_1,\ s) \Rightarrow s'' \qquad (c_2,\ s'') \Rightarrow s'}{(c_1;;\ c_2,\ s) \Rightarrow s'}\ \textsc{eval-Semi}$$

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

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

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

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

We consider the two following example programs:

definition *max* :: *com* **where**
max ≡
(*IF* ($\lambda s.\ s\ ''A'' > s\ ''B''$)
THEN ''R'' :== ($\lambda s.\ s\ ''A''$)
ELSE ''R'' :== ($\lambda s.\ s\ ''B''$)
FI)

definition *factorial* :: *com* **where**
factorial $\equiv$
$''B'' :== (\lambda s.\ 1);;$
$WHILE\ (\lambda s.\ s\ ''A'' \neq 0)\ DO$
$''B'' :== (\lambda s.\ s\ ''B'' * s\ ''A'');;$
$''A'' :== (\lambda s.\ s\ ''A'' - 1)$
OD

1 Variable Assignment (10 marks)

(a) Define a function *var-assig* that computes the set of variables that are assigned to in a command, e.g. *var-assig max* $= \{''R''\}$ and *var-assig factorial* $= \{''A'', ''B''\}$ (7 marks).

(b) Prove that if some variable is not assigned to in a command, then that variable is never modified by the command:

$[\![(c,\ s) \Rightarrow t;\ x \notin \textit{var-assig}\ c]\!] \Longrightarrow s\ x = t\ x$

(3 marks)

2 Equivalent programs (12 marks)

Programs are equivalent if they behave the same:
$c \sim c' \equiv \forall\, s\ t.\ (c,\ s) \Rightarrow t = (c',\ s) \Rightarrow t.$
Note that if two programs are equivalent then either both terminate or both do not terminate.

(a) Prove that *op* $\sim$ is an equivalence relation, i.e. it is reflexive, symmetric, and transitive. (3 marks)

(b) Prove: $[\![c \sim c';\ d \sim d']\!] \Longrightarrow (c;;\ d) \sim (c';;\ d')$ (2 marks)

(c) Prove: $[\![c \sim c';\ d \sim d']\!] \Longrightarrow IF\ b\ THEN\ c\ ELSE\ d\ FI \sim IF\ b\ THEN\ c'\ ELSE\ d'\ FI$ (2 marks)

(d) Prove: $[\![(p,\ s) \Rightarrow t;\ p = WHILE\ b\ DO\ c\ OD;\ c \sim c']\!] \Longrightarrow (WHILE\ b\ DO\ c'\ OD,\ s) \Rightarrow t$ (2 marks)

(e) Prove: $c \sim c' \Longrightarrow WHILE\ b\ DO\ c\ OD \sim WHILE\ b\ DO\ c'\ OD$ (3 marks)

3 The SKIP command (30 marks)

Consider the following recursive function *equivtoskip* that determines if a command behaves the same as *SKIP*:

primrec *equivtoskip* :: *com* ⇒ *bool* **where**
equivtoskip SKIP = *True* |
equivtoskip (*x*:==*a*) = *False* |
equivtoskip (*c1*;;*c2*) = (*equivtoskip c1* ∧ *equivtoskip c2*) |
equivtoskip (*IF b THEN c1 ELSE c2 FI*) = (*equivtoskip c1* ∧ *equivtoskip c2*) |
equivtoskip (*WHILE b DO c OD*) = *False*

(a) Prove the correctness of *equivtoskip*: *equivtoskip c* ⟹ *c* ∼ *SKIP*. (5 marks)

(b) Find some program that behaves like *SKIP* but for which *equivtoskip* returns *False*, i.e. prove: ∃ *c*. *c* ∼ *SKIP* ∧ ¬ *equivtoskip c*. (5 marks)

(c) Define a recursive function *less-skip* that eliminates as many *SKIP*s as possible from a command. For example:

less-skip (*SKIP*;; *SKIP*;; *WHILE b DO x* :== *a* ;; *SKIP OD*) =
WHILE b DO x :== *a OD*

less-skip (*IF b THEN SKIP*;; *SKIP ELSE* (*x* :== *a* ;; *SKIP*) *FI*) =
IF b THEN SKIP ELSE x :== *a FI*

less-skip (*IF b THEN SKIP*;; *SKIP ELSE SKIP FI*) = *SKIP*

less-skip (*WHILE b DO SKIP*;; *SKIP OD*) = *WHILE b DO SKIP OD*

(5 marks)

(d) Prove, by induction on *c*, that *less-skip* preserves the semantics of a program:
less-skip c ∼ *c* (15 marks)

4 Sequencing associativity (36 marks)

In this question we want to transform a program so that all sequence commands are associated to the right, e.g., *c1*;; (*c2*;; *c3*);; (*c4*;; *c5*) becomes *c1*;; (*c2*;; (*c3*;; (*c4*;; *c5*))). This transformation should also happen inside other commands like if-then-else, e.g., *c1*;; *IF b THEN c2*;; *c3*;; *c4 ELSE c5 FI*;; *c6* becomes *c1*;; (*IF b THEN c2*;; (*c3*;; *c4*) *ELSE c5 FI*;; *c6*).

Since this transformation is concerned primarily with sequencing, we translate IMP programs into a tree structure that hides the other IMP commands behind a single constructor, *Node*:

datatype *tag* = *S* | *A vname aexp* | *C bexp* | *W bexp*
datatype *tree* = *Node tag tree list* | *Branch tree tree*

fun *com-to-tree* **where**
  com-to-tree SKIP = *Node S* []
| *com-to-tree* (*Assign v a*) = *Node* (*A v a*) []
| *com-to-tree* (*Semi c1 c2*) = *Branch* (*com-to-tree c1*) (*com-to-tree c2*)
| *com-to-tree* (*Cond b c1 c2*) = *Node* (*C b*) [*com-to-tree c1*, *com-to-tree c2*]
| *com-to-tree* (*While b c*) = *Node* (*W b*) [*com-to-tree c*]

(a) Define an inverse function *tree-to-com* that transforms a tree back into a program. Note that not all trees represent an IMP program. Given a malformed tree you may return any program. (3 marks)

(b) Prove that *tree-to-com* (*com-to-tree c*) = *c*. (2 marks)

(c) Prove that *wf-tree t* $\Longrightarrow$ *com-to-tree* (*tree-to-com t*) = *t*. You will first need to define the predicate *wf-tree* that indicates if a tree is *well formed* in the sense that it represents an IMP program (e.g. an Assign node must have an empty list of subtrees). (6 marks)

(d) We now want to define a function *Branch-assoc* that transforms the tree to have sequences associated to the right:

Branch-assoc (*Branch* (*Branch t1 t2*) *t3*) =
  Branch-assoc (*Branch t1* (*Branch t2 t3*))
Branch-assoc (*Branch* (*Node t ls*) *t3*) =
  Branch (*Node t* (*map Branch-assoc ls*)) (*Branch-assoc t3*)
Branch-assoc (*Node t ls*) =
  Node t (*map Branch-assoc ls*)

Define such a function (using Isabelle's *function* command). To prove termination, you will need to define a measure on the tree that gets smaller with the transformation. It may be useful to use a *lexicographic combination* of size measures: in some recursive calls one measure decreases whereas in others the first measure stays the same but another decreases. The start of the termination proof below uses $size_1$ and $size_2$ as placeholders for two measures in lexicographic combination. You may use the Isabelle provided function *size* that computes the number of constructors used in a value of any datatype (including, in particular, trees). (10 marks)

termination *Branch-assoc*
apply (*relation inv-image* (*less-than* <∗*lex*∗> *less-than*) ($\lambda t.$ ($size_1$ *t*, $size_2$ *t*)))

(e) To prove that the transformation preserves the semantics, we first define a function that takes a function *f* and an IMP program, and applies *f* to all sequencing pairs in the program:

fun *map-Semi* **where**
map-Semi f (*Semi c1 c2*) = *f* (*map-Semi f c1*) (*map-Semi f c2*)
| *map-Semi f* (*Cond b c1 c2*) = *Cond b* (*map-Semi f c1*) (*map-Semi f c2*)
| *map-Semi f* (*While b c*) = *While b* (*map-Semi f c*)
| *map-Semi f x* = *x*

Now we can rewrite *tree-to-com* (*Branch-assoc t*), for any well-formed program *t*, as a *map-Semi* application:

wf-tree t $\Longrightarrow$
tree-to-com (*Branch-assoc t*) = *map-Semi Semi-assoc1* (*tree-to-com t*)

for a suitable *Semi-assoc1* function. Define such a *Semi-assoc1* function and then prove the theorem above. (10 marks)

(f) Now prove that the transformation preserves the semantics:
$(c, s) \Rightarrow s' \Longrightarrow$ (*tree-to-com* (*Branch-assoc* (*com-to-tree c*)), *s*) $\Rightarrow s'$
(5 marks)

5 Compilation (12 marks)

Consider a lower-level label-based language LAB, which only supports assignments and *goto* commands to explicit *labels* in the program. Labels are identified by natural numbers.

type-synonym *label* = *nat*

datatype *lcom* =

$$
\begin{aligned}
& \quad LAssign\ vname\ aexp \\
& \mid Label \quad label \\
& \mid LCGoto \quad bexp\ label \\
& \mid LGoto \quad label
\end{aligned}
$$

type-synonym $lprog = \ lcom\ list$

In this language, our 2 example programs *max* and *factorial* would be represented as:

definition
$lmax :: lprog$ **where**

$$
\begin{aligned}
lmax \equiv \ [\\
& LCGoto\ (\lambda s.\ s\ ''A'' \leq s\ ''B'')\ 0, \\
& LAssign\ ''R''\ (\lambda s.\ s\ ''A''), \\
& LGoto\ 1, \\
& Label\ 0, \\
& LAssign\ ''R''\ (\lambda s.\ s\ ''B''), \\
& Label\ 1]
\end{aligned}
$$

definition
$lfactorial :: lprog$ **where**

$$
\begin{aligned}
lfactorial \equiv [\ & LAssign\ ''B''\ (\lambda s.\ 1), \\
& Label\ 1, \\
& LCGoto\ (\lambda s.\ s\ ''A'' = 0)\ 0, \\
& LAssign\ ''B''\ (\lambda s.\ s\ ''B'' * s\ ''A''), \\
& LAssign\ ''A''\ (\lambda s.\ s\ ''A'' - 1), \\
& LGoto\ 1, \\
& Label\ 0]
\end{aligned}
$$

Define a function *compile* that takes a program in the IMP language and compiles it into the LAB language, such that the following two lemmas hold:

$compile\ max = lmax$

$compile\ factorial = lfactorial.$

You will need to define a helper function that takes an IMP program *and the "current available label"* and returns both the LAB program and a new label.