COMP4161 S2/2015
Advanced Topics in Software Verification

Assignment 2

This assignment starts on Mon, 2015-09-07 and is due on Sun, 2015-09-20, 23:59h.
We will accept Isabelle .thy files only.
Submit using `give` on a CSE machine:

```
give cs4161 a2 files ...
```

For example:

```
give cs4161 a2 a2.thy
```

Hint: *the questions in this assignment are phrased so that things that you prove in earlier sub-questions may often be useful to you in later sub-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.*

In this assignment, we define a small language of arithmetic expressions involving natural numbers. We then define a tiny compiler for this language to a simple stack machine, and prove the compiler correct. We also use the semantics of the stack machine to investigate stack usage of compiled programs.

1 Compiling Arithmetic Expressions (90 marks)

The following **datatype** abstractly describes the syntax of the language. It abstracts over binary and unary operators on natural numbers, which are represented as functions of type $nat \Rightarrow nat \Rightarrow nat$ and $nat \Rightarrow nat$ respectively.

datatype *expr* = *BinOp* (*nat* ⇒ *nat* ⇒ *nat*) *expr expr* |
UnOp (*nat* ⇒ *nat*) *expr* |
Const nat

For example, we can define the plus operator as follows:

definition
plus :: *nat* ⇒ *nat* ⇒ *nat*
where
plus ≡ (λ*a b*. *a* + *b*)

Then the expression "2 + 3" would be represented in this language by the term: *BinOp plus* (*Const 2*) (*Const 3*).

Note that Isabelle already has functions for addition, subtraction etc. The function for addition is written: *op* +; subtraction is *op* − etc. So for instance:

lemma *plus* = (*op* +)
by(*simp add*: *plus-def*)

We give a straightforward semantics to this language by means of an *evaluation function* that, given an expression, computes the result of evaluating that expression.

primrec
eval :: *expr* ⇒ *nat*

where
eval (*Const n*) = *n* |
eval (*BinOp f a b*) = *f* (*eval a*) (*eval b*) |
eval (*UnOp f a*) = *f* (*eval a*)

lemma *eval* (*BinOp* (*op +*) (*Const 2*) (*Const 3*)) = *5*
by *simp*

We will write a trivial compiler for compiling expressions of the language above to a simple stack machine. Programs for the stack machine are terms of the following **datatype**.

datatype *stackp* = *Push nat* |
DoUnOp nat ⇒ *nat* |
DoBinOp nat ⇒ *nat* ⇒ *nat* |
Seq stackp stackp (- ;; -) |
Skip

- The commmand *Push n* pushes the number *n* onto the stack;
- *DoUnOp f* pops the number at the top of the stack, applies *f* to it, and pushes the result back onto the stack;
- *DoBinOp f* pops the top two elements of the stack, applies *f* (a binary operator) to them, and pushes the result back onto the stack;
- *a* ;; *b* first runs the command *a*, followed by *b*;
- *Skip* can be used to mark the end of a command.

For example, the program that adds 2 and 3 could be written as: *Push 2* ;; *Push 3* ;; *DoBinOp op +*.

We will give the semantics for the stack machine as an inductive predicate that, given an initial stack and a program, can be used to calculate the final stack obtained when the program successfully executes. We model a stack of natural numbers as a list, where we will use the convention that the top of the stack is at the head of the list.

type-synonym *stack* = *nat list*

We give a *big step* operational semantics to programs for the stack machine. We write $\langle s,\ p\rangle \Downarrow s'$ to denote that given an initial stack s and program p, the program after execution will terminate leaving the final stack as s'. This predicate is defined inductively via the following rules.

$$\frac{}{\langle s,\ \mathit{Push}\ n\rangle \Downarrow n\ \#\ s}\text{SEM-PUSH} \qquad \frac{\langle s,\ a\rangle \Downarrow t \qquad \langle t,\ b\rangle \Downarrow u}{\langle s,\ a\ ;;\ b\rangle \Downarrow u}\text{SEM-SEQ}$$

$$\frac{}{\langle a\ \#\ s,\ \mathit{DoUnOp}\ f\rangle \Downarrow f\ a\ \#\ s}\text{SEM-DOUNOP} \qquad \frac{}{\langle a\ \#\ b\ \#\ s,\ \mathit{DoBinOp}\ f\rangle \Downarrow f\ a\ b\ \#\ s}\text{SEM-DOBINOP}$$

A simple compiler can be represented as a function that given an *expr* yields a corresponding *stackp*. The job of the resulting stack program is to evaluate the input expression, leaving the result on the top of the stack.

primrec
compile :: *expr* ⇒ *stackp*
where
compile (*Const n*) = *Push n* |
compile (*BinOp f a b*) = *Seq* (*compile b*) (*Seq* (*compile a*) (*DoBinOp f*)) |
compile (*UnOp f a*) = *Seq* (*compile a*) (*DoUnOp f*)

Question 1 (20 Marks)

(a) Prove that the stack machine semantics is *deterministic*, i.e. that:

$[\![\langle s,\ e\rangle \Downarrow t;\ \langle s,\ e\rangle \Downarrow u]\!] \Longrightarrow u = t$

(5 marks)

(b) Prove that the compiler is correct, i.e. that a compiled program executes to produce the same result as evaluating the input expression:

$\langle s,\ compile\ e\rangle \Downarrow eval\ e\ \#\ s$

(5 marks)

(c) Prove that whether an expression *can* evaluate or not depends only on the *size* of the initial stack from which it is evaluated, i.e. that:

$\langle s,\ p\rangle \Downarrow t \Longrightarrow \forall\, s'.\ length\ s' = length\ s \longrightarrow (\exists\, t'.\ \langle s',\ p\rangle \Downarrow t')$

Hint: you may need to strengthen the lemma statement to get a suitable induction hypothesis. (10 marks)

Question 2: Required Initial Stack (45 marks)

In this question, we will investigate how the size of the initial stack, from which a stack machine program executes, relates to the program's execution. We already know that whether a program *can* execute depends only on the size of the initial stack. In this question we will prove that, for a given program, we can calculate a size h such that if the initial stack has size h then the program will be able to execute successfully (if it can execute at all).

We begin by defining a predicate that captures whether an initial stack size h is sufficient to allow a program p to execute. Note that some programs (e.g. *Skip*, or *Skip* ;; *Skip*) can never execute for any initial stack. For this reason, we require h to allow the program to execute only if it can execute at all, i.e. if there exists some initial stack that allows it to execute.

definition
reqd-init-stack :: *stackp* ⇒ *nat* ⇒ *bool*
where
reqd-init-stack p h ≡ (∃ *s t*. ⟨*s*,*p*⟩ ⇓ *t*) ⟶ (∀ *s*. *length s* ≥ *h* ⟶ (∃ *t*. ⟨*s*,*p*⟩ ⇓ *t*))

(a) Prove that compiled expressions require no initial stack, i.e. that:

reqd-init-stack (*compile e*) *0*

(5 marks)

(b) For each of the *atomic* programs (i.e. *Skip*, *Push n*, *DoBinOp f* and *DoUnOp f*) decide on what you think the minimal initial stack length they each need and prove that it is sufficient, in terms of *reqd-init-stack*. For example, if you think that *Push n* requires an initial stack of length 1, you would prove *reqd-init-stack* (*Push n*) (*Suc 0*). Choose an appropriate length for each of the above atomic programs and prove it sufficient. (10 marks)

(c) Given a program *a* ;; *b*, suppose we know the required initial stack lengths for *a* and *b* are *n* and *m* respectively. Then what would be a suitable initial stack length *h* for *a* ;; *b*? Prove it, i.e. prove that: [\![*reqd-init-stack a n*; *reqd-init-stack b m*]\!] ⟹ *reqd-init-stack a* ;; *b h* where *h* is replaced by your answer.

Hint: h will need to mention n and m. You may also wish to investigate and make use of the list functions take and drop of Isabelle/HOL, and theorems proved about them and lists in general via **find_theorems**. (15 marks)

(d) Using the **primrec** command, define a function *sufficient-init-stack* that given a program p calculates an appropriate stack length h such that *reqd-init-stack* p h holds. (5 marks)

(e) Prove your function correct, i.e. that:

reqd-init-stack p (*sufficient-init-stack* p)

(10 marks)

Question 3: Runtime Stack Growth (25 marks)

We now give a *small-step* semantics to stack programs, to track the maximum stack-height during evaluation. A small-step semantics defines the state of the program after each execution step, rather than only at completion of the execution. We write $\langle s,\ p\rangle \longrightarrow \langle s',\ p'\rangle$ to denote that given a stack s and program p, the execution of one step of p results in program p' and stack s'. This predicate is defined inductively via the following rules.

$$\frac{}{\langle s,\ \mathit{Push}\ n\rangle \longrightarrow \langle n\ \#\ s,\ \mathit{Skip}\rangle}\text{SEM-PUSH}$$

$$\frac{\langle s,\ a\rangle \longrightarrow \langle s',\ a'\rangle}{\langle s,\ a\ ;;\ b\rangle \longrightarrow \langle s',\ a'\ ;;\ b\rangle}\text{SEM-SEQ} \qquad \frac{}{\langle s,\ \mathit{Skip}\ ;;\ b\rangle \longrightarrow \langle s,\ b\rangle}\text{SEM-SEQSKIP}$$

$$\frac{}{\langle a\ \#\ s,\ \mathit{DoUnOp}\ f\rangle \longrightarrow \langle f\ a\ \#\ s,\ \mathit{Skip}\rangle}\text{SEM-DOUNOP}$$

$$\frac{}{\langle a\ \#\ b\ \#\ s,\ \mathit{DoBinOp}\ f\rangle \longrightarrow \langle f\ a\ b\ \#\ s,\ \mathit{Skip}\rangle}\text{SEM-DOBINOP}$$

Let *prog-using-Suc* be a function that, given a bound h, computes a program that will use (at least) *Suc* h extra stack space during evaluation.

```
primrec
  prog-using-Suc :: nat ⇒ expr
where
  prog-using-Suc 0 = Const 0 |
  prog-using-Suc (Suc n) = (BinOp (op +) (prog-using-Suc n) (Const 0))
```

(a) Define a function *semsn* that executes n steps of the small-step semantics (5 marks)

(b) Prove that if a program a executes in the big-step semantics to a resulting stack t from an initial stack s, then it executes in the small-step semantics to the same resulting stack and the resulting program *Skip*.

$\langle s,\ a\rangle \Downarrow t \Longrightarrow \exists\, n.\ \mathit{semsn}\ n\ s\ a\ t\ \mathit{Skip}$

(10 marks)

(c) Let *stack-bound* be a predicate stating that stack size h is a stack bound for program p:

$\mathit{stack\text{-}bound}\ p\ h \equiv \forall\, s\ n\ s'\ p'.\ \mathit{semsn}\ n\ s\ p\ s'\ p' \longrightarrow \mathit{length}\ s' - \mathit{length}\ s \leq h$

Prove that there is no universal stack bound for any compiled program:

$\nexists\, h.\ \forall\, p.\ \mathit{stack\text{-}bound}\ (\mathit{compile}\ p)\ h$

Hint: use (i.e. prove and use) the fact that the specific program *prog-using-Suc* does not have any stack bound (10 marks)

2 Rewriting rules for groups (10 marks)

Assume a binary operator $\star$, an inverse operator i and a neutral element e. Write a confluent and terminating set of rules stating that e is a left-neutral and right-neutral element for the $\star$ operator, and that i is a left-inverse and right-inverse for the $\star$ operator. Justify why your set of rules is confluent and terminating.

In other words, replace *A*-*H* below by appropriate expressions so that it is safe to add the lemmas into the simp set.

axiomatization
  e:: $'a$ **and**
  op:: $'a \Rightarrow 'a \Rightarrow 'a$ (**infix** $\star$ *70*) **and**
  i:: $'a \Rightarrow 'a$
where
  neutral-left[*simp*]: $A = B$ **and**
  neutral-right[*simp*]: $C = D$ **and**
  inverse-left[*simp*]: $E = F$ **and**
  inverse-right[*simp*]: $G = H$