COMP4161 T3/2023
Advanced Topics in Software Verification

Assignment 2

This assignment starts on October 12th, and is due on November 2nd 23:59:59. We will accept Isabelle theory (.thy) files only. You are allowed to make late submissions up to five days (120 hours) after the deadline, but at a cost: -5 marks per day.

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 files ...
```

For example:

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

For this assignment, all proof methods and proof automation available in the standard Isabelle distribution is allowed. This includes, but is not limited to, `simp`, `auto`, `blast`, `force`, and `fastforce`.

However, if you're going for full marks, you shouldn't use "proof" methods that bypass the inference kernel, such as `sorry`. We *may* award partial marks for plausible proof sketches where some subgoals or lemmas are sorried.

If you use `sledgehammer`, it's important to understand that the proofs suggested by `sledgehammer` are just suggestions, and aren't guaranteed to work. Make sure that the proof suggested by `sledgehammer` actually terminates on your machine (assuming an average spec machine). If not, you can try to reconstruct the proof yourself based on the output, or apply a few manual steps to make the subgoal smaller before using `sledgehammer`.

Note: this document contains explanations of the problems and your assignment tasks. The full set of definitions can be found in the associated Isabelle theory files.

Hint: there are hints at the end of this document.

1 Lists as multisets (49 marks)

This question explores lists as multisets. A list [*0*, *0*, *0*, *2*, *1*, *3*, *2*] can be seen as a multiset containing 0,1,2,3, with multiplicity 3,1,2,1, respectively.

Isabelle has a built-in function *count-list* that gives multiplicity:

value *count-list* [*0*::*nat*,*0*,*0*,*2*,*1*,*3*,*2*] (*0*::*nat*)

In the following, instead of using this *count-list*, we define multiplicity as a finite map *m-count* and prove some properties about it.

In working on these questions, the *ext* lemma can be useful. Also, it is often useful (and necessary) to fine control the simplifier by using *simp only*:, *simp* (*no-asm-simp*), etc.

1.1 Multiplicity function for lists as multisets

We define *m-count*, as below. It converts a list as a multiset to a function that returns *nat option*: in particular, *m-count ls x* returns *None* if *x* does not appear in *ls* and returns *Some n* if x appears $n + 1$ times in *ls*. The *nat* value is offset by 1 because we start from *Some 0* when x appears once in *ls*.

```
primrec m-count :: 'a list ⇒ ('a, nat) map where
  m-count [] = Map.empty
| m-count (a#as) =
    (case (m-count as) a of
       None ⇒ (m-count as)(a ↦ 0)
     | Some n ⇒ (m-count as)(a ↦ Suc n))
```

(a) State the correspondence between Isabelle's *count-list* and *m-count* as logical equality and prove it. (3 marks)

(b) Prove that *m-count ls x* = *None* is equivalent to x not being a member of *ls*. (3 marks)

1.2 Ordering on multisets

We can define an ordering on multisets: *ls1* $\leq$ *ls2* holds if, for any x, the multiplicity of x is higher in *ls2* than in *ls1*. This ordering is defined in terms of *m-count* as below:

```
definition le :: ('a, nat) map ⇒ ('a, nat) map ⇒ bool where
  le m1 m2 ≡
     ∀x. case (m1 x, m2 x) of
           (Some a, Some b) ⇒ a ≤ b
         | (Some a, None) ⇒ False
         | (None, -) ⇒ True
```

(c) Give two examples of multisets *my-ls1*, *my-ls2* such that *le* (*m-count my-ls1*) (*m-count my-ls2*) holds. Then show that *le* is defined correctly by proving it. (3 marks)

(d) Prove that adding an element to a list by (#) always gives a list greater than the original list. (3 marks)

(e) Prove that *m-count* is preserved by changing the order of the elements in the list. (6 marks)

1.3 Addition of multisets

Next, we consider addition of two multisets. For lists as multisets, this corresponds to (@). We can also consider addition of multiplicity functions, which is defined for *m-count* as below:

```
definition m-add :: ('a, nat) map ⇒ ('a, nat) map ⇒ ('a,nat) map where
  m-add m1 m2 ≡
     λx. case (m1 x, m2 x) of
          (Some a, Some b) ⇒ Some (a + b + 1)
        | (Some a, None) ⇒ m1 x
        | (None, -) ⇒ m2 x
```

(f) Prove that the empty map is an identity for addition of maps (both for left and right). (2 marks)

(g) Prove that *m-add* correctly returns the multiplicity of added multisets, i.e., two lists appended by Isabelle (@). (6 marks)

1.4 Sorting of multisets

Consider the simple sorting function defined as below:

primrec *insort'* :: *'a::ord ⇒ 'a list ⇒ 'a list* **where**
  insort' x [] = [x] |
  insort' x (y#ys) =
  (if x ≤ y then (x#y#ys) else y#(insort' x ys))

definition *sort'* :: *('a::ord) list ⇒ 'a list* **where**
  sort' xs = foldr insort' xs []

(h) With respect to the above sorting function, prove that the sorting of a list does not change the multiset it represents; in other words, that *sort'* preserves multiplicity. (7 marks)

1.5 Union of multisets

We now consider a union of two multisets, whose multiplicity can be given by the following function:

definition *m-union* :: *('a, nat) map ⇒ ('a, nat) map ⇒ ('a,nat) map* **where**
  m-union m1 m2 ≡
    λx. case (m1 x, m2 x) of
      (Some a, Some b) ⇒ Some (max a b)
     | (Some a, None) ⇒ m1 x
     | (None, -) ⇒ m2 x

This means that, for each element x appearing in either of the two lists, the multiplicity of that element in the union of these two list is equal to the higher of the multiplicities of the element in the two lists. Such a union *m-Un* of lists as multisets can be defined as follows:

primrec *m-Un'* :: *'a list ⇒ 'a list ⇒ 'a list ⇒ 'a list* **where**
  m-Un' [] ys ac = ys @ ac
| m-Un' (x#xs) ys ac =
   (if x ∈ set ys then m-Un' xs (remove1 x ys) (x#ac)
    else m-Un' xs ys (x#ac))

definition *m-Un* **where** *m-Un l1 l2 = m-Un' l1 l2 []*

(i) Prove the following lemma about *m-Un'* and (@). (4 marks)

   m-Un' l1 l2 (ac @ ls) = (m-Un' l1 l2 ac) @ ls

(j) Prove the simplification lemma *m-count-remove1* about *remove1* and *m-count*. (4 marks)

(k) Prove the correctness of *m-Un* with respect to *m-union*. (8 marks)

   m-count (m-Un l1 l2) = m-union (m-count l1) (m-count l2)

2 Compiler for arithmetic expressions (51 marks)

In this question, we define a small language of arithmetic expressions and a compiler from those expressions to a simple target language for a machine that operates on registers.

The language of arithmetic expressions is defined as type *aexp* as follows, where the type *vname* is the type of variable names:

datatype *aexp = N int | V vname | Plus aexp aexp*

We give a straightforward semantics to this language as function *eval* which, given an expression and a variable state (a function from variable names to value, i.e., int), computes the value of the expression:

```
type-synonym val = int
type-synonym vstate = vname ⇒ val

primrec eval :: aexp ⇒ vstate ⇒ val where
  eval (N n) s = n
| eval (V x) s = s x
| eval (Plus e1 e2) s = eval e1 s + eval e2 s
```

(a) Give an example of an arithmetic expression which evaluates to 7 and uses *Plus* twice or more. Prove that your example actually evaluates to 7. (3 marks)

The compiler then compiles the terms of type *aexp* to programs for a simple target machine that operates on registers. These programs are terms of the following datatype, where registers are identified by natural numbers:

```
type-synonym reg = nat

datatype prog =
    LoadI reg val
  | Load reg vname
  | Add reg reg
  | Seq prog prog (- ;; - 100)
  | Skip
```

Here, *Seq p q* first runs the program p, followed by q. *Seq p q* can also be written *p ;; q*. *Skip* can be used to mark the end of a program.

We define our compiler as a function that, given an expression, yields a program of the target machine that corresponds to that expression. It also takes as a second argument a register identifier, and may use registers equal to or above that identifier for the program to be compiled. As a second return value, it also returns the next register identifier not so far used.

```
primrec compile :: aexp ⇒ reg ⇒ prog × reg
where
  compile (N n) r = (LoadI r n, r + 1) |
  compile (V x) r = (Load r x, r + 1) |
  compile (Plus e1 e2) r1 =
    (let (p1, r2) = compile e1 r1;
         (p2, r3) = compile e2 r2
     in ((Seq p1 (Seq p2 (Add r1 r2))), r3))
```

(b) Prove that the compiler always returns a register identifier strictly higher than the one given to it. (4 marks)

2.1 Target machine big-step semantics and compiler correctness

The target machine for the compiler operates on machine states *mstate*, which are a tuple of a register state *rstate* (a function from register identifiers to values), a *vstate* and the program *prog* left to be executed:

type-synonym *rstate* = *reg* ⇒ *val*
type-synonym *mstate* = *rstate* × *vstate* × *prog*

We define a *big-step* operational semantics for our machine and write *ms* ⇓ *rs* to denote that, given an initial machine state *ms*, the program after execution will terminate with a register state *rs*.

inductive *sem* :: *mstate* ⇒ *rstate* ⇒ *bool*
  (- ⇓ - [*0*,*60*] *61*) **where**
  sem-LoadI: (*rs*, σ, *LoadI r n*) ⇓ *rs*(*r* := *n*)
| *sem-Load*: (*rs*, σ, *Load r v*) ⇓ *rs*(*r* := σ *v*)
| *sem-Add*: (*rs*, σ, *Add r1 r2*) ⇓ *rs*(*r1* := *rs r1* + *rs r2*)
| *sem-Seq*: ⟦(*rs*, σ, *p*) ⇓ *rs*′; (*rs*′, σ, *q*) ⇓ *rs*″⟧ ⟹ (*rs*, σ, *p* ;; *q*) ⇓ *rs*″

(c) Prove that the target machine's semantics is deterministic. (5 marks)

(d) Prove that the compiler produces programs that do not modify any registers of identifier lower than the register identifier given to it. (8 marks)

(e) Prove that the compiler produces programs that, when executed, yield the value of the expression in the register *provided* as the argument to *compile* in the final *rstate* according to the program's big-step semantics. (8 marks)

2.2 Target machine small-step semantics

We now give a *small-step* semantics to the programs of our target machine. A small-step semantics defines the state of the program after each execution step, rather than only at completion of the execution. We write *ms* ⇝ *ms*′ to denote that given a machine state *ms*, the execution of one step of the machine results in machine state *ms*′. This predicate is defined inductively as follows:

inductive *s-sem* :: *mstate* ⇒ *mstate* ⇒ *bool* (- ⇝ - *100*)
  where
  (*rs*, σ, *LoadI r n*) ⇝ (*rs*(*r* := *n*), σ, *Skip*)
| (*rs*, σ, *Load r v*) ⇝ (*rs*(*r* := σ *v*), σ, *Skip*)
| (*rs*, σ, *Add r1 r2*) ⇝ (*rs*(*r1* := *rs r1* + *rs r2*), σ, *Skip*)
| (*rs*, σ, *p*) ⇝ (*rs*′, σ, *p*′) ⟹ (*rs*, σ, *p* ;; *q*) ⇝ (*rs*′, σ, *p*′ ;; *q*)
| (*rs*, σ, *Skip* ;; *p*) ⇝ (*rs*, σ, *p*)

(f) Define a function *s-sem-n* that executes n steps of the small-step semantics. (3 marks)

(g) Prove that two executions of resp. *n* and *m* steps according to *s-sem-n* compose into a single execution of *n* + *m* steps if their resp. final and initial machine state match. (4 marks)

(h) Prove that if *p* executes to *p*′ in *n* steps according to *s-sem-n*, then *p* ;; *q* will execute to *p*′ ;; *q* in *n* steps with all other parts of the machine state being the same as in the original execution. (4 marks)

(i) Prove that if a program executes in the big-step semantics to a resulting *rstate*, then it executes in the small-step semantics to a machine state with the same resulting *rstate* and the resulting program *Skip* with no changes to the *vstate*. (5 marks)

2.3 No universal bound on register usage

Finally we will prove that there is no number of registers that is large enough to bound universally the number of registers used by *any* possible compiled program.

Let *term-with-n-Suc* be a function that, given a bound *h*, generates an expression that will use (at least) *Suc h* extra registers during evaluation.

primrec *term-with-n-Suc* :: *nat* ⇒ *aexp* **where**
  term-with-n-Suc 0 = *N 0*
| *term-with-n-Suc* (*Suc n*) = (*Plus* (*term-with-n-Suc n*) (*N 0*))

(j) Prove that compiling *term-with-n-Suc h* will use a number of registers that is indeed strictly lower bounded by *h*. (4 marks)

(k) Using this fact, prove that there is no universal bound on the number of registers used for any compiled program. (3 marks)

3 Hints

- Many proofs will require induction of one kind or the other. Other than inducting on datatypes directly, you may find it useful to do induction on inductively defined relations such as *sem* and *s-sem*. The induction rules for these are automatically generated by Isabelle.

  You can apply these induction rules as elimination rules, e.g. `apply(erule sem.induct)`, but a more convenient and flexible alternative is

  ```
  apply(induct rule: sem.induct)
  ```

  which allows you to specify which variables should not be all-eliminated using e.g.

  ```
  apply(induct arbitrary: x y rule: sem.induct)
  ```

- Not everything needs an induction.
- The equivalent of `spec` for the meta-logic universal quantifier, if you need it, is called `meta_spec`.
- For some exercises, you will likely need additional lemmas to make the proof go through. Part of the assignment is figuring out which lemmas are needed.
- Make use of the `find_theorems` command to find library theorems. You are allowed to use all theorems proved in the Isabelle distribution.