Advanced Topics in Formal Methods and Prog. Languages
– Software Verification with Isabelle/HOL –

Assignment 2

ver 1.0

Submission Guidelines

- Due time: Oct 04, 2024, 6pm (Canberra Time)
- Submit via Wattle.
- Accepted formats are plain text (.txt) files, PDF (.pdf) files, and Isabelle theory (.thy) files.
- Isabelle files should be executable (a template is provided on the course webpage).
- Please read and sign the declaration on the last page and attach a copy to your submission.
- **No late submission, deadline is strict.**

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 are going for full marks, you should not 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 is 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. Please **avoid proofs based on** *smt*. It may useful (and necessary) to fine control the simplifier by using *simp only:*, *simp (no-asm-simp)*, etc.

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. Some hints are provided at the end of this document.

Exercise 1 (Binary Trees) (49 Marks)

This exercise explores binary tree. Their definition is standard:

```
datatype ('a :: linorder) tree =
    Leaf
  | Branch "'a tree" 'a "'a tree"
```

An example is giving by the following tree:

```
"Branch (Branch Leaf (3::nat) Leaf) 1 (Branch (Branch Leaf 4 Leaf) 2 Leaf)"
```

The nodes inside the tree have the same arbitrary type `'a`. For the later parts of this exercise, we assume that these elements are taken from a linear order, i.e. we have reflexivity ($x \leq x$), transitivity ($x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$), antisymmetry ($x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$) and linearity/totality ($x \leq y \vee y \leq x$).

Question 1: Height and width of a tree (8 marks)

The height of a tree can be defined by a simple recursive function.

```
primrec tree-height :: "('a::linorder) tree ⇒ nat" where
    "tree-height Leaf = 1"
  | "tree-height (Branch l x r) = 1 + max (tree-height l) (tree-height r)"
```

(a) Define a function `number-leaves` that determines the number of leaves of a given tree.

(b) Prove a (strict) upper bound of tree height, depending on `number-leaves`.

Question 2: Flattening trees (5 marks)

We can create a function that flattens the tree and returns a list of nodes.

```
primrec lnr-list :: "'a :: linorder tree ⇒'a list" where
    "lnr-list Leaf = []"
  | "lnr-list (Branch l x r) = lnr-list l @ x # lnr-list r"
```

Isabelle automatically generates a function `"set-tree :: tree ⇒'a set"`, which transforms a tree into a set, listing all the nodes of a tree. For example,

```
set-tree (Branch (Branch Leaf (3::nat) Leaf) 1 (Branch (Branch Leaf 4 Leaf) 2 Leaf))
= {1,2,3,4}
```

(c) Show that `lnr-list` contains all elements of the tree, ignoring duplicates.

(d) Strengthen the result of Part (c) to take distinctness into account.

Question 3: Sorted trees (22 marks)

We are now making use of the linear order on nodes, and define a sorted tree.

```
primrec sorted-tree where
    "sorted-tree Leaf = True"
  | "sorted-tree (Branch l x r) =
   ((∀y∈set-tree l. y ≤ x) ∧ (∀y∈set-tree r. y ≥ x) ∧ sorted-tree l ∧ sorted-tree r)"
```

(e) Show that this function indeed characterises a sorted tree.

(f) Define `insort-tree x t` which inserts an element `x` into the (sorted) tree `t` such that `t` stays sorted.

(g) Prove the two properties of sorted trees given in the Isabelle file.

Using the insert function we can now build up trees from lists:

```
definition rl-tree where
  "rl-tree ts ≡ foldr insort-tree ts Leaf"
```

Combining this with `rl-tree` we can sort trees:

```
abbreviation sort-tree where
  "sort-tree t ≡ rl-tree (lnr-list t)"
```

(h) Prove that `"lnr-list (sort-tree t) = sort (lnr-list t)"`.

(i) Finally show that `"sort-tree"` generates a sorted tree.

Question 4: Different traversal (X marks)

The last question defines a different traversal of trees; going 'top-down'. The definitions are similar to the one above.

primrec `nlr-list :: "'a :: linorder tree ⇒] 'a list" where`

```
    "nlr-list Leaf = []"
  | "nlr-list (Branch l x r) = x # nlr-list l  nlr-list r"
```

definition `lr-tree where`

```
  "lr-tree init ts ≡ foldl (λ t x. insort-tree x t) init ts"
```

(j) Define a function that checks distinctness of subtrees: for any given node `x`, `x` is neither in the left nor the right subtree of `x`. Some examples for trees featuring this properties are given in the Isabelle file.

(k) Now show that any sorted tree `t` with distinct subtrees satisfies the property `lr-tree Leaf (nlr-list t) = t`.
Please note that this is a hard exercise.

Exercise 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.

Question 5 Datatypes and evaluation7 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.

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.

Question 6: Target machine big-step semantics and compiler correctness (21 marks)

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.

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

(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.

Question 7: Target machine small-step semantics (16 marks)

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:

```
inductive s-sem :: mstate ⇒ mstate ⇒ bool (- ⇝ - 100) where
  (rs, σ, LoadI r n)
| (rs, σ, Load r v)
| (rs, σ,Add r1 r2)
| (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.

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

(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.

(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`.

Question 8: No universal bound on register usage (7 marks)

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`.

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

1 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 an induction over a function `f`. For example, if `f x` returns a

list, you can **apply** *(induct "f x")*. In that case the base case is *[] = f x* and the step is *a # as = f x*.

You may also 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.

Academic Integrity

I declare that this work upholds the principles of academic integrity, as defined in the University Academic Misconduct Rule; is entirely my own work, with only the exceptions listed; is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener; gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used; in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

_____ _____

Date Signature