COMP4161 S2/2018
Advanced Topics in Software Verification

Assignment 2

This assignment starts on Monday, 2018-09-03 and is due on Thu, 2018-09-20, 23:59h. We will accept Isabelle .thy files only. In addition to this pdf document, please refer to the provided Isabelle templates for the definitions and lemma statements.

The assignment is take-home. This does NOT mean you are allowed to 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 (18 marks)

Prove the following statements, using only the proof methods: `rule`, `erule`, `assumption`, `frule`, `drule`, `rule_tac`, `erule_tac`, `frule_tac`, `drule_tac`, `rename_tac`, and `case_tac`; and using only the proof rules: `impI`, `impE`, `conjI`, `conjE`, `disjI1`, `disjI2`, `disjE`, `notI`, `notE`, `iffI`, `iffE`, `iffD1`, `iffD2`, `ccontr`, `classical`, `FalseE`, `TrueI`, `allI`, `allE`, `exI`, and `exE`. 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) `(¬ (∀x. P x)) = (∃x. ¬ P x)` (3 marks)

(b) `(∀x. P ⟶ Q x) = (P ⟶ (∀x. Q x))` (2 marks)

(c) `∀x. ¬ f x ⟶ f (g x) ⟹ ∀x. f x ∨ f (g x)` (3 marks)

(d) `⟦∀x. ¬ f x ⟶ f (g x); ∃x. f x⟧ ⟹ ∃x. f x ∧ f (g (g x))` (4 marks)

(e) `(∀x. Q x = P x) ∧ ((∃x. P x) ⟶ H) ∧ (∃x. Q x) ⟶ H` (3 marks)

(f) `(∀Q. (∀x. P x ⟶ Q) ⟶ Q) ⟶ (∃x. P x)` (3 marks)

2 Strings and Regular Expressions (38 marks)

(a) (4 marks) Use `primrec` to define functions `chop a xs` and `glue a xss` that take a separator `a` and chop a list `xs` into a list of lists `xss` and glue them together again, respectively. As an example let `a` be the newline character, `xs` a text string of multiple lines, and `xss` the list of lines in `xs`.

(b) (4 marks) Use `primrec` to define a function `num_of x xs` (without using `chop`) that counts the occurrences of an element `x` in a list `xs`. Prove the following lemmas (`sum_list` is the sum of the elements of a list):

- `length (glue a xss) = sum_list (map length xss) + length xss - 1`
- `length (chop x xs) = num_of x xs + 1`

(c) (4 marks) Prove the following correctness lemmas for `chop` and `glue`:

- `ls ∈ set (chop a xs) ⟹ a ∉ set ls`
- `glue a (chop a xs) = xs`

(d) (6 marks) Let regular expressions be defined as in the lecture, in the provided file `RegExp.thy`. Define a function `any_of rs` that takes a list of regular expressions `rs` and returns the regular expression that matches any of the expressions in `rs`. Prove:

- `lang (any_of rs) = ⋃ (lang ‘ set rs)`

Further, define a function `repeat A n` that takes a set of strings `A` and an number `n`, and returns the set of strings that is the `n`-time concatenation of the strings in A. Prove:

- `star A = (⋃n repeat A n)`

(e) (10 marks) Using `definition` or `primrec` define the following.

- A function `matches_sub r xs` that returns `True` iff `r` matches any substring of `xs`. You can assume a function `matches r xs` that returns `True` iff the regular expression `r` matches `xs`.
- `string xs` that returns the regular expression that matches string `xs`.
- `is_prefix xs ys` that returns `True` iff list `xs` is a prefix of `ys`.
- `is_substring xs ys` that returns `True` iff list `xs` is a sublist of `ys`.

. Then prove:

- `matches_sub r xs = (∃xs' ys zs. xs = ys @ xs' @ zs ∧ matches r xs')`
- `matches (string xs) ys = (ys = xs)`
- `is_substring xs ys = (∃bs cs. ys = bs @ xs @ cs)`
- `matches_sub (string xs) l = is_substring xs l`

(f) (8 marks) Use `primrec` to define a function `rlen r` that determines if regular expression `r` has matches of constant length `n`, and if so returns `Some n`, otherwise `None`. This function cannot be fully precise with `primrec`, but make it precise enough such that it can return the length of the string `xs` if the regular expression was constructed using `string xs`. For example:

```
rlen (string ''abc'') = Some 3
rlen (Alt (string ''a'') (string ''ab'')) = None
rlen (Alt (string ''a'') (string ''b'')) = Some 1
```

Prove:

- `rlen (string xs) = Some (length xs)`
- `rlen r = Some n ⟹ ∀xs∈lang r. length xs = n`

3 Normal Forms (44 marks)

This question is looking at the normal forms of a particular rewriting rule on strings. The rewriting rule is `[x,y,x]` ⟶ `[x]`, i.e. anywhere in a string, it rewrites the pattern `[x, y, x]` (for any `x` and `y`) into `x`. This system is confluent and terminating, so normal forms exist, and we can compute them by repeated application of the rule.

(a) (5 marks) Define a function `find_pat xs` that searches the list `xs` from the left for the first occurrence of a pattern of the form `[x, y, x]` and returns the list with this occurrence replaced by `x`. If the pattern does not occur in `xs`, return `xs`. Prove the following sanity-test lemmas:

- `hd (find_pat xs) = hd xs`
- `last (find_pat xs) = last xs`
- `(find_pat xs ≠ xs) = (∃as bs x y. xs = as @ x # y # x # bs)`

(b) (13 marks) Prove the following properties of `find_pat`. They encode as much of the confluence of the rewrite system as we will need in the rest of the assignment.

- `find_pat (x # find_pat xs) = find_pat (find_pat (x # xs))`
- `find_pat (find_pat xs @ [x]) = find_pat (find_pat (xs @ [x]))`
- `find_pat xs = xs ⟹ find_pat (rev xs) = rev xs`
- `find_pat xs = xs ⟹ find_pat (rev xs @ [x]) = rev (find_pat (x # xs))`

(c) (8 marks) The assignment template gives the definition of a function that computes the normal form of this rewrite system as:

```
nf xs = (if find_pat xs = xs then xs else nf (find_pat xs))
```

From the failed termination proof in the template, extract a lemma about `find_pat` that lets Isabelle prove termination automatically. Supply the lemma by declaring it as `[simp]`. Then prove the following basic properties about `nf`:

- `nf (find_pat xs) = nf xs`
- `find_pat (nf xs) = nf xs`
- `nf (nf xs) = nf xs`
- `hd (nf xs) = hd xs`
- `last (nf xs) = last xs`

(d) (18 marks) Finally, since the pattern in the rewrite rule is symmetric, the normal form of the reverse of the list should be the reverse of the normal form of the list. Prove this formally by showing the following lemmas.

- `nf (x # nf xs) = nf (x # xs)`
- `nf (x # xs) = find_pat (x # nf xs)`
- `nf (xs @ [x]) = find_pat (nf xs @ [x])`
- `nf (rev xs) = rev (nf xs)`

Hints:

- `nf.simps` is prone to non-termination as a `simp` rule. Use `apply (subst nf.simps)`, potentially with instantiation to specialise the rule, for unfolding `nf` manually.
- For induction on lists involving `rev`, the reverse induction rule `rev_induct` is occasionally useful.
- Remember that `fun` produces custom `induct` and `cases` rules, which can help to reduce the number of manual case distinctions.
- The template states a number of helper lemmas that you can choose to use in your solution, in which case you have to prove them to get full marks. It is Ok to not prove them if you do not use them.
- You are allowed to use sledgehammer for questions 2 and 3 in this assignment.