COMP4161 2013/s2
Advanced Topics in Software Verification

Exam

This take-home exam starts on 14 Nov 2013, 08:00 am and is due on 15 Nov 2013, 07:59 am. We will accept plain text files, PDF files, and Isabelle theory files (.thy); submission instructions are posted on the website

http://www.cse.unsw.edu.au/~cs4161/exam.html

As usual, you may (and should) use helper lemmas to simplify your proofs. You may also use any lemmas in theories imported by Main, Word, Autocorres etc: the *find theorems* command helps find such lemmas. Finally, all work must be your own, the exam policy is more restrictive than for assignments:

You must not discuss the exam with anyone except the lecturers of this course before the exam is due. Do not give or receive assistance.

You are allowed to use all lecture material, slides, and assignment solutions from the web. You are also allowed to use other passive internet resources such as Google, the Isabelle tutorial or Isabelle documentation. You are not allowed to ask for assistance on mailing lists, forums, or anywhere else. You are allowed to clarify questions with the lecturers.

Hints: Each question is divided into a number of sub-questions. Many of these ask you to prove results that can, and should, be used to help prove later lemmas in that question. If you get stuck trying to prove a result, use the **sorry** command and move on to the next part of the question. This way, you can still use an earlier, unproved result to solve a later goal if needed. If you correctly prove a later sub-question using a sorried lemma from a previous question, you will still earn full marks for that later question.

1 Isar and Locales (17 marks)

In this question we are interested in the following statement:

$$[\![0 < a;\ a < b]\!] \Longrightarrow a * a < b * b \tag{1}$$

(a) Prove the statement (1) for `a` and `b` of type `nat` using automated tools. *Hint: use a type-annotation to constrain the types of* `a` *and* `b` *in the lemma statement.* (2 marks)

(b) Prove (1) in Isar style, using only single step rule applications (i.e. using *only* `rule`, `erule` and `assumption` as proof methods[a]) and using no predefined lemmas other than *`less_trans`*, *`mult_less_mono1`* and *`mult_less_mono2`*. (5 marks)

[a] You are *not* allowed to use automated methods here such as `simp`, `auto`, `blast`, `sledgehammer`, etc.

(c) Now we want to prove that the statement for `a` and `b` of any type `'a` that supports:

- a special element, say called *`zero`*, of typ `'a`
- a binary operator $\cdot$ of type `'a ⇒ 'a ⇒ 'a`,
- and a binary relation $\ll$ of type `'a ⇒ 'a ⇒ bool`

such that the 3 properties expressed by *`less_trans`*, *`mult_less_mono1`* and *`mult_less_mono2`* hold for this new type and operators:

- $i \ll j \Longrightarrow zero \ll k \Longrightarrow (i \cdot k) \ll (j \cdot k)$
- $i \ll j \Longrightarrow zero \ll k \Longrightarrow (k \cdot i) \ll (k \cdot j)$
- $x \ll y \Longrightarrow y \ll z \Longrightarrow x \ll z$

Define a type class that enclapsulates these 3 operators and 3 assumptions. (You may use any notation you want for the operators.) (5 marks)

(d) Generalise and prove the statement (1) for `a` and `b` of type `'a` where `'a` is of the class defined above, and using the operators of that class instead of the ones on *`nat`* (5 marks)

2 C code verification (39 marks)

In this question, we ask you to prove the (partial) correctness of a C function *`is_sorted`* that takes a linked list of integers as parameter, and returns 1 if the list is sorted and 0 otherwise (partial correctness means that you are not required to prove termination).

The C function is provided in the `exam.c` file. You may use any verification technique you wish (VCG or AutoCorres), and you are recommended to use the linked lists lemmas from assignment 3 (see solutions on the website).

The precondition should state that the input pointer points to a linked list (using the *linked_list* predicate from assignment 3). The postcondition should state that the returning value of the function is not zero if and only if the list of values from the linked list is sorted (using the predefined *sorted* predicate from Isabelle).

3 Sets as Interval Lists (44 marks)

One common data structure implementation problem is that of implementing sets of numbers. In the following, we will implement sets of natural numbers (*nat set*) as an ordered list of intervals (*nat* $*$ *nat*) *list*.

The set

$$\{2, 3, 5, 7, 8, 9\}$$

for instance should be implemented as

$$[(2, 4), (5, 6), (7, 10)]$$

That means, the intervals are inclusive on the left and exclusive on the right component. We want the implementation to have unique representatives for each set. This implies that intervals should be merged whenever possible, i.e. $[(2, 4), (4, 5)]$ is not a valid implementation and should be written as $[(2, 5)]$. Intervals should be in ascending order from left to right in the list.

(a) **(12 marks)**
Define a type synonym `intervals` for interval lists and two constants `is_int_list` and `to_set`.

The predicate `is_int_list` should decide if a given interval list satisfies the implementation constraints above.

The function `to_set` should take an interval list and return the set of natural numbers this list represents. Use the notation {`a..<b`} for the set of numbers from a to b, excluding b.

(b) **(12 marks)**
Define a function `add` that adds one interval to a given list of intervals. The functions should merge intervals accordingly such that they preserve the implementation invariant `is_int_list`. Write at least three concrete test cases (either using *value* or using lemmas executed with `simp` or `auto`).

(c) **(20 marks)**
Prove

`to_set (add (a,b) xs) = to_set xs` $\cup$ {`a..<b`}

and:

`add` preserves the `is_int_list` invariant.

You may make an additional assumption on a and b and for the `to_set` lemma you may assume `is_int_list xs` if necessary.