# COMP4161 S2/2014
# Advanced Topics in Software Verification

## Exam

This take-home exam starts on Monday 10 Nov 2014, 08:00 am and is due on Tuesday 11 Nov 2014, 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

Many of the questions require you to write Isabelle proofs. For these questions you should use the template Isabelle theory (.thy) file provided. This file, and these questions, are designed to be completed using Isabelle 2013-2 and AutoCorres 0.98.

As usual, you may (and should) use helper lemmas to simplify your proofs. You may also use any lemmas in theories imported by AutoCorres: 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 Lambda Calculus (15 marks)

Consider the term `λa b. b (λx. x a)`.

(a) What is its $\beta\eta$-normal form? (2 marks)

(b) What is its type? (2 marks)

(c) Provide a step-by-step derivation of its type, i.e. give a pen-and-paper proof of your answer to part (b) (10 marks)

## 2 Induction (45 marks)

(a) A different induction principle for lists (21 marks)

```
⟦P []; ⋀x. P [x]; ⋀a b xs. P xs ⟹ P (a # xs @ [b])⟧ ⟹ P list
```

In this question we will be deriving step-by-step a different induction principle for lists that says a property holds for all lists if it holds for: (1) the empty list, (2) all lists with a single element, and (3) all lists formed by adding new elements to the front and back of an existing list for which the property holds.

To guide your proof, the development is split into the following sub parts.

(i) Using the **inductive_set** command, define the set `pal_lists` that contains: the empty list, all lists with a single element, and all lists formed by adding new elements to the front and back of an existing list in `pal_lists`. (3 marks)

(ii) Prove that if a list `xs` is in `pal_lists`, then `x # xs ∈ pal_lists`, i.e. prove `xs ∈ pal_lists ⟹ x # xs ∈ pal_lists`. (10 marks)

(iii) Prove that `pal_lists` contains all lists, i.e. prove `xs ∈ pal_lists`. (4 marks)

(iv) Using this result, or otherwise, prove the induction principle above. (4 marks)

(b) The function `palindrome` mechanically decides if a list is a palindrome, making use of the in-built function `rev` that reverses a list.

```
palindrome xs ≡ rev xs = xs
```

Implement `palindrome` using the **definition** command. (1 mark)

(c) Prove that if `a # xs @ [b]` is a palindrome, then `a = b`, i.e.

```
palindrome (a # xs @ [b]) ⟹ a = b
```

(4 marks)

(d) Inductively define the set `palindromes` that contains all palindromes and nothing else, using the **inductive_set** command. (5 marks)

(e) Prove that `palindrome xs` if and only if `xs ∈ palindromes`, i.e.

```
palindrome xs = (xs ∈ palindromes)
```

(14 marks)

## 3 C Verification (40 marks)

We will now verify some simple C functions.

```
unsigned divmod(unsigned n, unsigned m, unsigned domod) {
  unsigned d = 0;
  while(n >= m) {
    n -= m;
    d++;
  }
  if (domod) {
    return n;
  } else {
    return d;
  }
}

unsigned even(unsigned n){
  return (divmod(n,2,1) == 0);
}
```

(a) After being processed by AutoCorres with the `unsigned_word_abs` enabled for both functions, what are the names of the functions produced by AutoCorres that represent the semantics of the `divmod()` and `even()` functions, and what are their types? Which monad has AutoCorres used to represent both of them? (5 marks)

(b) Is `divmod()` guaranteed to always terminate? Explain and justify your answer. (5 marks)

(c) We can give an abstract specification for the behaviour of `divmod()` via the following definition.

```
divmod_spec n m domod ≡ if domod ≠ 0 then n mod m else n div m
```

Prove that `divmod()` yields correct answers, i.e. prove that

```
ovalid (λ_. True) (divmod' n m domod)
 (λr s. r = divmod_spec n m domod)
```

(8 marks)

(d) Write and prove correct a weakest precondition rule for `divmod'`, i.e. replace the term `?PRE` in the following statement with the weakest precondition that guarantees the postcondition `Q`, and prove the statement correct: `ovalid ?PRE (divmod' n m domod) Q` (12 marks)

(e) Prove that `even()` returns correct answers, i.e. prove

```
ovalid (λ_. True) (even' n) (λr s. r = (if even n then 1 else 0))
```

(5 marks)

(f) Is `even()` guaranteed to always terminate? Explain and justify your answer. (5 marks)

**end**