Interactive Formal Verification (L21)
Exercises and Marking Scheme

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Lent Term, 2017

Interactive Formal Verification consists of twelve lectures and four practical sessions. The handouts for the first two practical sessions will not be assessed. You may find that these handouts contain more work than you can complete in an hour. You are not required to complete these exercises; they are merely intended to be instructive. Many more exercises can be found at http://isabelle.in.tum.de/exercises/. Note that many of these on-line examples are very simple: the assessed exercises are considerably harder. You are strongly encouraged to attempt a variety of exercises, and perhaps to develop your own.

The handouts for the last two practical sessions determine your final mark (50% each). For each assessed exercise, please complete the indicated tasks and write a brief document explaining your work. You may earn additional credit by preparing these documents using Isabelle's theory presentation facility (See section 4.2 of the Isabelle/HOL manual). You can combine the resulting output with a document produced using your favourite word processing package. Please ensure that your specifications are correct (because proofs based on incorrect specifications could be worthless) and that your Isabelle theory actually runs.

Each assessed exercise is worth 100 marks.

- 50 marks are for completing the tasks. Proofs should be competently done and tidily presented. Be sure to delete obsolete material from failed proof attempts. Excessive length (within reason) is not penalised, but slow or redundant proof steps may be.

- 20 marks are for a clear, basic write-up. It can be just a few pages, and probably no longer than 6 pages. It should explain your proofs, preferably displaying these proofs if they are not too long. It could perhaps outline the strategic decisions that affected the shape of your proof and include notes about your experience in completing it.

- The final 30 marks are for exceptional work. To earn some of these marks, you may need to vary your proof style, maybe expanding some

apply-style proofs into structured proofs. The point is not to make your proofs longer (brevity is a virtue) but to demonstrate a variety of Isabelle skills, perhaps even techniques not covered in the course. An exceptional write-up also gains a few marks in this category, while untidy proofs will lose marks. Very few students will gain more than half of these marks, but note that 85% is a very high score.

Isabelle theory files for all four sessions can be downloaded from the course materials website. These files contain necessary Isabelle declarations that you can use as a basis for your own work.

You must work on these assignments as an individual; collaboration is not permitted. Here are the deadline dates. Exercises are due at 12 NOON.

- 1st exercise: Tuesday, 21 February 2017
- 2nd exercise: Thursday, 9 March 2017

Please deliver a printed copy of each completed exercise to student administration, and also send the corresponding theory file to lp15@cam.ac.uk. The latter should be enclosed in a directory bearing your name.

1 Replace, Reverse and Delete

Define a function `replace`, such that `replace x y zs` yields `zs` with every occurrence of `x` replaced by `y`.

```
consts replace :: "'a ⇒ 'a ⇒ 'a list ⇒ 'a list"
```

Prove or disprove (by counterexample) the following theorems. You may have to prove some lemmas first.

```
theorem "rev(replace x y zs) = replace x y (rev zs)"
theorem "replace x y (replace u v zs) = replace u v (replace x y zs)"
theorem "replace y z (replace x y zs) = replace x z zs"
```

Define two functions for removing elements from a list: `del1 x xs` deletes the first occurrence (from the left) of `x` in `xs`, `delall x xs` all of them.

```
consts del1   :: "'a ⇒ 'a list ⇒ 'a list"
       delall :: "'a ⇒ 'a list ⇒ 'a list"
```

Prove or disprove (by counterexample) the following theorems.

```
theorem "del1 x (delall x xs) = delall x xs"
theorem "delall x (delall x xs) = delall x xs"
theorem "delall x (del1 x xs) = delall x xs"
theorem "del1 x (del1 y zs) = del1 y (del1 x zs)"
theorem "delall x (del1 y zs) = del1 y (delall x zs)"
theorem "delall x (delall y zs) = delall y (delall x zs)"
theorem "del1 y (replace x y xs) = del1 x xs"
theorem "delall y (replace x y xs) = delall x xs"
theorem "replace x y (delall x zs) = delall x zs"
theorem "replace x y (delall z zs) = delall z (replace x y zs)"
theorem "rev(del1 x xs) = del1 x (rev xs)"
theorem "rev(delall x xs) = delall x (rev xs)"
```

2 Power, Sum

2.1 Power

Define a primitive recursive function *pow x n* that computes x^n on natural numbers.

```
consts
  pow :: "nat => nat => nat"
```

Prove the well known equation $x^{m \cdot n} = (x^m)^n$:

```
theorem pow_mult: "pow x (m * n) = pow (pow x m) n"
```

Hint: prove a suitable lemma first. If you need to appeal to associativity and commutativity of multiplication: the corresponding simplification rules are named `mult_ac`.

2.2 Summation

Define a (primitive recursive) function *sum ns* that sums a list of natural numbers: $sum[n_1, \ldots, n_k] = n_1 + \cdots + n_k$.

```
consts
  sum :: "nat list => nat"
```

Show that *sum* is compatible with *rev*. You may need a lemma.

```
theorem sum_rev: "sum (rev ns) = sum ns"
```

Define a function *Sum f k* that sums *f* from 0 up to $k-1$: $Sum\ f\ k = f\ 0 + \cdots + f(k-1)$.

```
consts
  Sum :: "(nat => nat) => nat => nat"
```

Show the following equations for the pointwise summation of functions. Determine first what the expression `whatever` should be.

```
theorem "Sum (%i. f i + g i) k = Sum f k + Sum g k"
theorem "Sum f (k + l) = Sum f k + Sum whatever l"
```

What is the relationship between `powSum_ex.sum` and `Sum`? Prove the following equation, suitably instantiated.

```
theorem "Sum f k = sum whatever"
```

Hint: familiarize yourself with the predefined functions `map` and `[i..<j]` on lists in theory List.

3 Assessed Exercise I: Euler's totient function

Euler's totient function, written $\phi(n)$, denotes the number of integers 1, 2, ..., n that are coprime to the positive integer n. So for example $\phi(1) = \phi(2) = 1$ and $\phi(3) = \phi(4) = 2$. The totient function is fundamental to number theory and this exercise establishes some of its elementary properties. See Baker, *A Concise Introduction to the Theory of Numbers* (Cambridge University Press, 1984), page 9. Other books on elementary number theory will also cover this function.

Task 1 *Define the totient function φ of type* `nat ⇒ nat` *as described above. Note that the cardinality of a finite set can be expressed using the built-in function* `card`*, and the two-argument predicate* `coprime` *is also available. Greek letters can be inserted using the Symbols palette, but you may give the function the name* `phi` *if you prefer.*

Then prove the following two facts. *[5 marks]*

```
lemma phi_1: "φ 1 = 1"
lemma phi_2: "φ 2 = 1"
```

Task 2 *The following exercise establishes an alternative characterisation of the totient function.* *[10 marks]*

```
lemma phi_altdef: "φ(n) = card {m. coprime m n ∧ m < n}"
```

Task 3 *Among the other straightforward properties of the totient function is that $\phi(p) = p - 1$ if p is prime.* *[10 marks]*

```
lemma phi_prime[simp]:
  assumes "prime p" shows "φ p = (p-1)"
```

Task 4 *The result above can be generalised to $\phi(p^j) = p^j - p^{j-1}$, where p is prime and $j > 0$.* *[25 marks]*

```
lemma phi_prime_power[simp]:
  assumes "prime p" "j > 0" shows "φ (p ^ j) = p ^ j - p ^ (j-1)"
```

Hint: none of these proofs require induction. Typically they involve manipulations of sets of positive integers, perhaps using equational reasoning.

4 Assessed Exercise II: The Binary Euclidean Algorithm

The greatest common divisor of two natural numbers can be computed efficiently using a binary version of Euclid's algorithm. It eliminates common factors of two (which in hardware can be done efficiently by shifting), and given two odd numbers it subtracts them, producing another even number.

- The GCD of x and 0 is x.
- If the GCD of x and y is z, then the GCD of $2x$ and $2y$ is $2z$.
- The GCD of $2x$ and y is the same as that of x and y if y is odd.
- The GCD of x and y is the same as that of $x - y$ and y if $y \leq x$.
- The GCD of x and y is the same as the GCD of y and x.

This algorithm is actually nondeterministic, in that the steps can be applied in any order. However the result is unique because a pair of positive integers has exactly one greatest common divisor.

Task 1 *Inductively define the set* `BinaryGCD` *such that* `(x, y, g) ∈ BinaryGCD` *means* `g` *is computed from* `x` *and* `y` *as specified by the description above.* *[5 marks]*

```
consts  BinaryGCD :: "(nat × nat × nat) set"
```

Task 2 *Show that the* `BinaryGCD` *of* `x` *and* `y` *is really the greatest common divisor of both numbers, with respect to the divides relation. Hint: it may help to consider whether* `d` *is even or odd. Be careful to choose the right form of induction, and justify your choice in your write-up.* *[15 marks]*

```
lemma GCD_greatest_divisor:
  "(x,y,g) ∈ BinaryGCD ⟹ d dvd x ⟹ d dvd y ⟹ d dvd g"
```

Task 3 *Prove the following statement. In the form given (assuming* `n` *to be odd), it can be proved directly by induction.* *[10 marks]*

```
lemma GCD_mult:
  "(x,y,g) ∈ BinaryGCD ⟹ odd n ⟹ (n*x,n*y,n*g) ∈ BinaryGCD"
```

Remark: the theorem above actually holds for all `n`, but the simplest way of proving it is probably to prove that `BinaryGCD` corresponds exactly to the true `gcd` function and then to use properties of the latter.

Task 4 *How do we know that* `BinaryGCD` *can compute a result for all values of* `a` *and* `b`*? To prove it requires a carefully formulated induction, as shown in the theorem statement below. We need course-of-values induction (expressed by the theorem* `less_induct`*), which allows us to assume the induction formula for everything smaller than* `n`*. (Why doesn't standard induction work here?)*

Hint: the algorithm is complete even if the steps `GCDEven` *and* `GCDOdd` *are deleted. They merely improve performance, so your proof can ignore them. You will still need to consider various cases corresponding to the remaining steps of the algorithm.* *[20 marks]*

lemma `GCD_defined_aux: "a+b ≤ n ⟹ ∃g. (a, b, g) ∈ BinaryGCD"`

Armed with this lemma, the completeness statement is trivial.

theorem `GCD_defined: "∃g. (a, b, g) ∈ BinaryGCD"`