Interactive Formal Verification (L21) Exercises and Marking Scheme

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Michaelmas Term, 2019

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, but you are not required to complete them: they are merely intended to be instructive. Many more exercises can be found at http://isabelle.in.tum.de/exercises/, but they tend to be easy. The assessed exercises are considerably harder, as you can see by looking at those of previous years.

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 this document using Isabelle's theory presentation facility.[1] Alternatively, write the document 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. Sledgehammer may be used, but multi-line sledgehammer proofs can be unreadable and should not be presented in their raw form.

- 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

[1] See section 4.2 of the Isabelle/HOL Tutorial, https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2019/doc/tutorial.pdf.

proof and include notes about your experience in completing it. Please don't copy the text of the exercises into your own write-up.

- 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. Taking the effort to make your proofs more readable can help.

  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 forbidden. Copying material found elsewhere counts as plagiarism. Here are the deadline dates. Exercises are due at 12 NOON.

- 1st exercise: Wednesday, 6 November 2019
- 2nd exercise: Wednesday, 20 November 2019

For each exercise, submit both the Isabelle theory file and the accompanying write-up by the deadline, using Moodle.

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: Basic Number Theory

This exercise concerns prime powers that divide a natural number. It imports the theory of prime numbers, `HOL-Computational_Algebra.Primes`.

Task 1 *As a warmup, prove the two lemmas below.* *[5 marks]*

```
lemma dvd_power_iff_le:
  fixes k::nat
  shows "2 ≤ k ⟹ k^m dvd k^n ⟷ m ≤ n"

lemma primepow_divides_prod:
  fixes r::nat
  assumes "prime r" "(r^k) dvd (m * n)"
  shows "∃i j. (r^i) dvd m ∧ (r^j) dvd n ∧ k = i + j"
```

Task 2 *Prove the following lemma.* *[5 marks]*

```
lemma le_power2:
  assumes "k ≥ 2" shows "m ≤ k^m"
```

Task 3 *Prove the following lemma. The constant* `Max` *should be useful here.* *[5 marks]*

```
lemma maximum_index:
  fixes n::nat
  assumes "n > 0" "2 ≤ r"
  shows "∃k. r^k dvd n ∧ (∀l>k. ¬ r^l dvd n)"
```

Now we define the index of a divisor to a natural number. Degenerate cases map to 0, whilst otherwise it returns the maximum j such that r^j divides n.

```
definition index where
   "index r n
    ≡ if r ≤ Suc 0 ∨ n = 0 then 0 else card {j. 1 ≤ j ∧ r^j dvd n}"
```

Task 4 *Prove the following proposition and its two corollaries.* *[10 marks]*

```
proposition pow_divides_index:
  "r^k dvd n ⟷ n = 0 ∨ r = Suc 0 ∨ k ≤ index r n"

corollary le_index:
  "k ≤ index r n ⟷ (n = 0 ∨ r = 1 ⟶ k = 0) ∧ r^k dvd n"

corollary exp_index_divides: "r^(index r n) dvd n"
```

Task 5 *Prove the following lemma.* *[10 marks]*

```
lemma index_trivial_bound: "index r n ≤ n"
```

Task 6 *Prove the following proposition and its corollary.* *[15 marks]*

```
proposition index_mult:
  assumes "prime p" "m > 0" "n > 0"
  shows "index p (m * n) = index p m + index p n"

corollary index_exp:
  assumes "prime p"
  shows "index p (n^k) = k * index p n"
```

No proof should require more than 25 lines.

4 Assessed Exercise II: Basic Combinatorics

This exercise concerns a function $f(n,k)$: the number of ways of selecting k objects, no two consecutive, from n objects arranged in a row. It makes sense to assume that the "objects" are simply the natural numbers below n. Then we can express "no two consecutive" as `Suc j ∉ K` if `j ∈ K`.

Task 1 *Define the function* `mf_family`*, denoting the actual set of ways of selecting k out of n objects, as described above. Then define* `mf` *as its cardinality and test your definitions by proving the following elementary properties.* *[10 marks]*

```
lemma mf_n_0 [simp]: "mf n 0 = 1"

lemma mf_n_1 [simp]: "mf n (Suc 0) = n"

lemma mf_0 [simp]: "mf 0 k = (if k=0 then 1 else 0)"

lemma mf_1 [simp]: "mf (Suc 0) k = (if k < 2 then Suc 0 else 0)"

lemma mf_eq_zero:
  assumes "n > Suc 0" and "k ≥ n"
  shows "mf n k = 0"
```

Task 2 *The following proposition reduces the case $f(n,k)$ to two smaller cases. The reasoning is that if we reject the last object, namely $n-1$, then we still have k elements to choose from the remaining $n-1$ objects. If instead we select the last object, then because of "no two consecutive", we must reject its neighbour, namely $n-2$. Working through this simple argument requires a fair bit of reasoning.* *[25 marks]*

```
proposition mf:
  assumes "Suc 0 < k" "k < n"
  shows "mf n k = mf (n-1) k + mf (n-2) (k-1)"
```

Task 3 *The following proposition expresses our function using binomial coefficients, using the result of the previous task. It requires complete (aka course-of-values) induction on $n+k$, using the Isabelle theorem* `less_induct`*. You'll need to eliminate the degenerate cases first.* *[15 marks]*

```
proposition mf_closed_form: "mf n k = (Suc n - k) choose k"
```