Interactive Formal Verification (L21)
Exercises

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Lent Term, 2016

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 *will be assessed* to 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 prepare these documents using Isabelle's theory presentation facility (See section 4.2 of the Isabelle/HOL manual) but this is not required. 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, 16th February 2016
- 2nd exercise: Thursday, 3rd March 2016

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 `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: The Exponent Function

This Exercise concerns elementary properties of prime numbers, leading up to a function denoting the exponent of a prime `p` in the factorisation of a given number `s`.

3.1 Prime numbers

Task 1 *The theorem* `prime_dvd_mult_eq_nat` *asserts a fundamental property of prime numbers:* `prime p ⟹ (p dvd m * n) = (p dvd m ∨ p dvd n)`. *Prove the following generalisation of that property:* *[5 marks]*

```
lemma prime_dvd_cases:
  assumes pk: "p*k dvd m*n" and p: "prime p"
  shows "(∃x. k dvd x*n ∧ m = p*x) ∨ (∃y. k dvd m*y ∧ n = p*y)"
```

Task 2 *Now prove the following result by a precisely formulated induction:* *[10 marks]*

```
lemma prime_power_dvd_prod:
  assumes pc: "p^c dvd m*n" and p: "prime p"
  shows "∃a b. a+b = c ∧ p^a dvd m ∧ p^b dvd n"
```

Task 3 *Therefore obtain the following corollary:* *[5 marks]*

```
lemma prime_power_dvd_cases:
 "⟦p^c dvd m * n; a + b = Suc c; prime p⟧ ⟹ p^a dvd m ∨ p^b dvd n"
```

3.2 The exponent function

Now we define the exponent of a prime power in the factorisation of `s`.

```
definition
  exponent :: "nat ⇒ nat ⇒ nat" where
  "exponent p s = (if prime p then (GREATEST r. p^r dvd s) else 0)"
```

Task 4 *For this definition to make sense, we need an upper bound on* `n` *such that* $\texttt{p}^{\texttt{n}}$ `dvd a`. *Prove the following. Why is* `0 < a` *necessary?* *[7 marks]*

```
lemma power_dvd_bound: "⟦p ^ n dvd a; Suc 0 < p; 0 < a⟧ ⟹ n < a"
```

Task 5 *Then prove these fundamental properties of* `exponent p s`. *The theorems* `GreatestI` *and* `Greatest_le` *allow reasoning about GREATEST.* *[7 marks]*

```
lemma exponent_eq_0 [simp]: "¬ prime p ⟹ exponent p s = 0"

lemma exponent_ge:
  assumes "p ^ k dvd n" "prime p" "0 < n"
    shows "k ≤ exponent p n"

lemma power_exponent_dvd: "p ^ exponent p s dvd s"

lemma exponent_equalityI:
  "(⋀r. p ^ r dvd a ⟷ p ^ r dvd b) ⟹ exponent p a = exponent p b"
```

Task 6 *Now for our main result: the exponent of* `p` *in the product is the sum of the exponents of* `p` *in the two factors. Prove it.* *[16 marks]*

```
theorem exponent_mult_add:
  assumes "a > 0" "b > 0"
    shows "exponent p (a * b) = (exponent p a) + (exponent p b)"
```

4 Assessed Exercise II: Propositional Logic

In this Exercise, we formalise a fragment of propositional logic along with a proof system for it. We prove soundness and the deduction theorem.

4.1 Propositional formulas

We begin by defining the datatype of *formulas*, comprising atoms (propositional letters), disjunctions and negations. Atoms are designated by natural numbers.

```
datatype fm =
    Atom nat
  | Disj fm fm   (infixr "OR" 130)
  | Neg fm
```

Task 1 *Define a function* `subst` *of type* `fm ⇒ nat ⇒ fm ⇒ fm` *to substitute a given formula* `X` *for a particular atom* `i` *in the formula* `A`*. All occurrences of* `Atom i` *should be replaced by* `X`*, while other Atoms should be preserved; disjunctions and negations should be transformed recursively.*

Also, define a function `vars` *of type* `nat set` *to return the set of all Atoms in a given formula* `A`*. Then prove the three statements below. If your definitions are correct, each should be trivial by induction.* *[7 marks]*

```
lemma vars_subst:
  "vars (subst X i A) =
    (if i ∈ vars A then vars X ∪ (vars A - {i}) else vars A)"

lemma subst_trivial [simp]: "i ∉ vars A ⟹ subst X i A = A"

lemma subst_idem [simp]:
  "i ∉ vars X ⟹ subst X i (subst X i A) = subst X i A"
```

The semantics of propositional logic is simple: a formula is evaluated recursively using standard truth tables. The values of the atoms are given with respect to an *environment*, which maps each atom to true or false. Here is the null environment:

```
definition e0 :: "nat ⇒ bool"  where "e0 ≡ λk. False"
```

Task 2 *Define a function* `eval` *of type* `(nat ⇒ bool) ⇒ fm ⇒ bool` *to evaluate the formula* `A` *with respect to the environment* `e`*. Check your definition by proving the statement below, which again should be a trivial induction.* *[4 marks]*

```
lemma eval_subst: "eval e (subst X i A) = eval (e (i:= eval e X)) A"
```

4.2 Propositional axioms and rules

We use a Hilbert system for propositional logic, involving five axiom schemes:

$$A \to A$$
$$A \to (A \vee B)$$
$$(A \vee A) \to A$$
$$(A \vee (B \vee C)) \to ((A \vee B) \vee C)$$
$$(C \vee A) \to (((\neg C) \vee B) \to (A \vee B))$$

Thus, the axioms include everything of the form $A \to A$ (where A is replaced by any propositional formula), and so forth for the other four schemes. The following abbreviation is helpful for expressing the axioms in Isabelle.

```
abbreviation Imp :: "fm ⇒ fm ⇒ fm"   (infixr "IMP" 125)
  where "Imp A B ≡ Disj (Neg A) B"
```

Task 3 *Define the set of axioms as a constant* `boolax` *of type* `fm set` *in Isabelle. Validate the definition by proving that all axioms evaluate to true, as stated below:* *[7 marks]*

```
lemma boolax_hold: "A ∈ boolax ⟹ eval e A"
```

We could now obtain a classic Hilbert system by adding in the rule of Modus Ponens, but a more flexible approach is to define *deducibility* from a set H of formulas.

$$\frac{A \in H}{H \vdash A} \qquad \frac{A \in \mathit{boolax}}{H \vdash A} \qquad \frac{H \vdash A \to B \quad H' \vdash A}{H \cup H' \vdash B}$$

Task 4 *Define this relation in Isabelle by completing the following inductive definition. Then prove the soundness theorem given below:* *[7 marks]*

```
inductive ded :: "fm set ⇒ fm ⇒ bool" (infixl "⊢" 55)

theorem ded_sound:
  assumes "H ⊢ A"
    shows "(∀B∈H. eval e B) ⟹ eval e A"
```

4.3 The Deduction Theorem

The deduction theorem states that $\{A\} \cup H \vdash B$ implies $H \vdash A \to B$. Although it may look natural, it is not always easy to prove. We begin by proving two lemmas, then set up the induction.

Task 5 *Prove these lemmas.* *[10 marks]*

```
lemma S:
  assumes "H ⊢ A IMP (B IMP C)" "H' ⊢ A IMP B"
    shows "H ∪ H' ⊢ A IMP C"

lemma Imp_triv_I: "H ⊢ B ⟹ H ⊢ A IMP B"
```

Task 6 *Finally, prove the deduction theorem. The first statement is in a form suitable for induction. It easily implies the second statement.*
[15 marks]

```
lemma deduction_Diff: assumes "H ⊢ B" shows "H - {C} ⊢ C IMP B"

theorem deduction: "insert A H ⊢ B ⟹ H ⊢ A IMP B"
```

Remark: no proof should require much more than a dozen lines. Proving suitably chosen lemmas can simplify your work.