Interactive Formal Verification (L21)
Exercises

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Lent Term, 2014

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. Of those, 50 marks are for completing the tasks, 25 marks are for the write-up and the final 25 marks are for demonstrating a wide knowledge of Isabelle primitives and techniques. To earn these final 25 marks, you may need to vary your proof style and perhaps to expand some brief **apply**-style proofs into structured proofs; the point is not to make your proofs longer (other things being equal, brevity is a virtue) but to show that you have mastered a variety of Isabelle skills. You could even demonstrate techniques not covered in the course.

To get full credit, your write-up must be clear. It can be brief, 2–4 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.

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 deadlines:

- 1st exercise: Tuesday, 18th February 2014
- 2nd exercise: Thursday, 6th March 2014

Please deliver a printed copy of each completed exercise to student administration **by midday** on the given date, 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: Verifying a Tautology Checker

This exercise reproduces an example first done by Boyer and Moore [1] during the 1970s. It concerns a topology checker for propositional logic with only one connective, namely if-then-else.

$$\varphi \texttt{ ::= FALSE | TRUE | VAR n | IF } \varphi \texttt{ THEN } \varphi\texttt{1 ELSE } \varphi\texttt{2}$$

This one connective can represent all the familiar Boolean operators, such as conjunction. We declare a datatype `ifexpr` of propositional formulas, using natural numbers for the names of propositional variables.

```
datatype ifexpr =
    FALSE
  | TRUE
  | VAR nat
  | IF ifexpr ifexpr ifexpr
```

Task 1 *A propositional formula is evaluated with respect to an environment mapping all propositional variables to either true or false. A set of the "true" variables is one simple representation of an environment. Define a function* `eval :: [nat set, ifexpr] => bool` *to evaluate propositional formulas according to the obvious semantics of the conditional operator.* *[5 marks]*

The tautology checker requires if-expressions to be transformed into normal form, where `IF` does not appear within the condition of an if-expression. Normal form can be reached by repeatedly replacing

`IF (IF p p1 p2) q r` by `IF p (IF p1 q r) (IF p2 q r)`

until this replacement is no longer possible.

Proving termination of a function to normalise an if-expression requires first defining the following function, which provides an upper bound on the recursive calls.

```
fun normrank :: "ifexpr ⇒ nat"
where
  "normrank (IF p q r) =
    normrank p + normrank p * normrank q + normrank p * normrank r"
| "normrank p = 1"
```

Task 2 *Write a function* `norm :: ifexpr ⇒ ifexpr` *to transform if-expressions into normal form. Because the proof of termination involves* `normrank`*, it*

is necessary to use `function` *rather than* `fun`*. See section 4.1 of "Defining Recursive Functions in Isabelle/HOL" for details. The termination proof requires a very simple lemma about* `normrank`*, as well as some algebraic reasoning.* *[7 marks]*

Task 3 *Prove the following theorem, which states that normalisation preserves the meaning of a formula.* *[3 marks]*

lemma `norm_is_sound: "eval env (norm p) = eval env p"`

Task 4 *Define a predicate* `normalised :: ifexpr ⇒ bool` *that returns true if and only if the given formula is in normal form. Then prove the following result, which states that the normalisation function indeed converts formulas into normal form.* *[5 marks]*

lemma `normalised_norm: "normalised (norm p)"`

The tautology checker is defined as follows. The if-expression is assumed to be normalised, so its condition can only be some variable `n`. The sets `ts` and `fs` are assumed to be disjoint.

```
fun (sequential) taut :: "[nat set, nat set, ifexpr] => bool"
where
    "taut ts fs FALSE     = False"
  | "taut ts fs TRUE      = True"
  | "taut ts fs (VAR n)   = (n ∈ ts)"
  | "taut ts fs (IF (VAR n) q r) =
       (if n ∈ ts then taut ts fs q
        else if n ∈ fs then taut ts fs r
        else taut (insert n ts) fs q ∧ taut ts (insert n fs) r)"
  | "taut ts fs p = False"     — default case (overlapping patterns)
```

The main tautology checker begins with empty sets of true and false variables.

```
definition tautology
where "tautology p ≡ taut {} {} (norm p)"
```

Task 5 *Prove the following theorem, which states that any formula* `p` *accepted by the tautology checker evaluates to true. Hint: this will require a lemma to be proved by induction, describing the behaviour of* `taut ts fs p`*, assuming* `normalised p`*; you will need to find the right relationships among* `env`*,* `ts` *and* `fs`*.* *[13 marks]*

theorem `tautology_sound: "tautology p ⟹ eval env p"`

Task 6 *Prove the following completeness theorem: any normalised formula* `p` *such that* `eval env p` *evaluates to true for all* `env` *will be accepted by the tautology checker. Hint: as in the previous task.* *[17 marks]*

theorem `tautology_complete: "(⋀env. eval env p) ⟹ tautology p"`

4 Assessed Exercise II: Properties of Binomial Coefficients

The binomial coefficients arise in the binomial theorem. They are the elements of Pascal's triangle and satisfy a great many mathematical identities. Below, we examine some of these. The theory `HOL/Library/Binomial` contains basic definitions and proofs, including the binomial theorem itself:

```
(a + b)^n = (∑k = 0..n. (n choose k) * a^k * b^n - k)
```

Many textbooks on discrete mathematics cover binomial coefficients. Information about them is widely available on the Internet, including Wikipedia and the lecture course notes available here:

`http://www.cs.columbia.edu/~cs4205/files/CM4.pdf`

Task 1 *Prove the following theorem. Hint: it follows from the binomial theorem, which is available under the name* `binomial`. *[4 marks]*

```
lemma choose_row_sum: "(∑k=0..n. n choose k) = 2^n"
```

Task 2 *Prove the following two theorems, which concern sums of binomial coefficients.* *[4 marks]*

```
lemma sum_choose_lower:
    "(∑k=0..n. (r+k) choose k) = Suc (r+n) choose n"
lemma sum_choose_upper:
    "(∑k=0..n. k choose m) = Suc n choose Suc m"
```

The built-in theorem `setsum_reindex_cong` allows a summation to be reindexed by any injective function. (The notion of set image is also used.)

```
⟦inj_on f A; B = f ‘ A; ⋀a. a ∈ A ⟹ g a = h (f a)⟧
⟹ setsum h B = setsum g A
```

Task 3 *Prove the following corollary of* `setsum_reindex_cong`*, which allows the indexes of a summation to be reversed.* *[5 marks]*

```
corollary natsum_reverse_index:
  fixes m::nat
  assumes "⋀k. m ≤ k ⟹ k ≤ n ⟹ g k = f (m + n - k)"
  shows "(∑k=m..n. f k) = (∑k=m..n. g k)"
```

Task 4 *Prove the following identity. In addition to the re-indexing result just proved, you will need another of the identities proved above. [10 marks]*

```
lemma sum_choose_diagonal:
  assumes "m≤n"
    shows "(∑k=0..m. (n-k) choose (m-k)) = Suc n choose m"
```

The identity $\binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}$ is straightforward. But the need to show the necessary divisibility properties complicates the Isabelle proof. The key result we need is already available as `binomial_fact_lemma`:

```
k ≤ n ⟹ fact k * fact (n - k) * (n choose k) = fact n
```

Task 5 *Using it, prove the following divisibility property.* *[5 marks]*

```
lemma fact_fact_dvd_fact:
    fixes k::nat
    shows "fact k * fact n dvd fact (n + k)"
```

Now the identity $\binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}$ can be proved by expressing binomial coefficients in terms of factorials:

```
k ≤ n ⟹ n choose k = fact n div (fact k * fact (n - k))
```

Common factors are then cancelled using the result just proved along with the lemma `div_mult_div_if_dvd`:

```
⟦y dvd x; z dvd w⟧ ⟹ x div y * (w div z) = x * w div (y * z)
```

The equational calculation is still short, but each step requires some justification.

Task 6 *Prove the identity, expressed as shown below.* *[18 marks]*

```
lemma choose_mult_p1:
     "((m+r+k) choose (m+k)) * ((m+k) choose k) =
      ((m+r+k) choose k) * ((m+r) choose m)"
```

Task 7 *The formulation given above uses addition instead of subtraction, which tends to complicate proofs about the natural numbers. Now that the hard work has been done, establish the identity in its conventional form.* *[4 marks]*

```
lemma choose_mult:
  assumes "k≤m" "m≤n"
    shows "(n choose m) * (m choose k) =
           (n choose k) * ((n-k) choose (m-k))"
```

References

[1] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.