Interactive Formal Verification
Course Handouts

Lawrence C Paulson

Interactive Formal Verification consists of 12 lectures and 4 practical sessions. The handouts for the first two practical sessions will not be assessed. Both handouts contain much more work than can be completed in an hour. You are not required to do all (indeed any) of the problems on these handouts, but please do as many of them as you find beneficial for learning. Many more exercises can be found on the Internet, at http://isabelle.in.tum.de/exercises/. You may use the terminals in SW02 whenever the room has not been booked for another course.

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, but this is not required. A very simple way to print a theory file legibly is to use the Proof General command `Isabelle > Commands > Display draft`. You can combine the resulting output with a document produced using your favourite word processing package. A clear write-up describing elegant, clearly structured proofs of all tasks will receive maximum credit. The document can be quite brief, around two pages, and should explain strategic decisions that affected the shape of your proof, including some 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.

The first assessed exercise will be due on Friday, 25 May 2012 and the second assessed exercise will be due on Friday, 15 June 2012, both at 12 noon. You must work on these assignments as an individual: collaboration is not permitted.

Please deliver a printed copy of each completed exercise to student administration by that deadline, and also send the corresponding theory file to lp15@cam.ac.uk.

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 CNF Conversion Function

Many theorem-proving algorithms are based on conjunctive normal form (CNF). The objective of this exercise is to formalise propositional formulas and their semantics, and then to prove that translating a propositional formula into CNF preserves its semantics while yielding a CNF formula.

More details on these concepts are presented in my Logic and Proof lecture notes, sections 2.5–2.6 (http://www.cl.cam.ac.uk/teaching/current/LogicProof/logic-notes.pdf). They are also discussed in *ML for the Working Programmer*, and ML code relevant to the solution of this exercise is online (http://www.cl.cam.ac.uk/~lp15/MLbook/programs/sample4.sml.gz).

Note: This exercise does not require long, complicated proofs. It does require definitions to be made carefully. Some of the recursive functions are a little tricky.

Task 1 *Declare a datatype* `pfm` *of propositional formulas, including a symbol for false, propositional variables and the connectives of implication, disjunction, conjunction and negation:*

$$\varphi \texttt{ ::= FALSE | VAR n | } \varphi \to \varphi' \texttt{ | } \varphi \lor \varphi' \texttt{ | } \varphi \land \varphi' \texttt{ | } \neg\varphi$$

The type of natural numbers can be used to represent the names of propositional variables.

Task 2 *A propositional formula is evaluated with respect to a truth table (or interpretation) mapping propositional variables to true or false. Define a function* `eval` *to evaluate propositional formulas according to the obvious semantics of the logical connectives.*

```
consts  eval :: "[nat⇒bool, pfm] => bool"
```

Task 3 *Write a function to convert a propositional formula into negation normal form (NNF). Prove the following theorem, which states that it preserves the meaning of a formula.*

```
consts  nnf :: "pfm ⇒ pfm"
lemma eval_nnf: "eval ttab (nnf p) = eval ttab p"
```

Task 4 *Write a function to convert a formula (assumed already to be in NNF) into CNF. Then, prove that it preserves the meaning of a formula.*

```
consts  nnf2cnf :: "pfm ⇒ pfm"
lemma eval_nnf2cnf: "eval ttab (nnf2cnf p) = eval ttab p"
```

The following steps are then trivial.

```
definition cnf :: "pfm => pfm"
  where "cnf p = nnf2cnf (nnf p)"

theorem eval_cnf: "eval ttab (cnf p) = eval ttab p"
```

Task 5 *Define Isabelle/HOL predicates for the concepts of atoms, literals, clauses (disjunctions of literals) and CNF formulas. You could use either recursion or inductive definitions.*

```
consts
  atom    :: "pfm ⇒ bool"
  literal :: "pfm ⇒ bool"
  clause  :: "pfm ⇒ bool"
  cnfprop :: "pfm ⇒ bool"
```

Task 6 *Prove that your CNF conversion function actually delivers a CNF formula.*

```
theorem cnfprop_cnf: "cnfprop (cnf p)"
```

4 Assessed Exercise II: Hereditarily Finite Sets

The *hereditarily finite sets* are built up recursively, starting with the *empty set*, using an operator which *inserts* an element into a set. Note that sets and elements have the same type!

```
definition hempty  :: "hf"
definition hinsert :: "hf ⇒ hf ⇒ hf"
```

There is also a *membership relation* on these strange sets.

```
definition hmem :: "hf ⇒ hf ⇒ bool"
```

The membership relation satisfies the following natural properties.

```
lemma hmem_hempty: "¬ hmem a hempty"
lemma hmem_hinsert: "hmem a (hinsert b c) ⟷ a = b ∨ hmem a c"
```

Two sets are *equal* if and only if they have the same elements.

```
lemma hf_ext: "a = b ⟷ (∀x. hmem x a ⟷ hmem x b)"
```

Using that crucial property (which is called extensionality), one can easily prove that the primitive set operators can be characterised by their elements.

```
lemma hempty_iff:
    "z = hempty ⟷ (∀x. ¬ hmem x z)"
  by (simp add: hf_ext)

lemma hinsert_iff:
    "z = hinsert x y ⟷ (∀u. hmem u z ⟷ hmem u y | u=x)"
  by (auto simp add: hf_ext)
```

Many properties of the hereditarily finite sets can be proved using the following *induction rule*. (This rule also tells us that every such set is a finite construction.)

```
lemma hf_induct [induct type: hf, case_names hempty hinsert]:
  assumes [simp]: "P hempty"
                  "⋀x y. ⟦P x; P y; ¬ hmem x y⟧ ⟹ P (hinsert x y)"
  shows "P z"
```

You are likely to need this induction rule to prove the existence results below.

Task 1 *Prove that insertion operations can be exchanged, as shown below.*

```
lemma hinsert_commute:
    "hinsert x (hinsert y z) = hinsert y (hinsert x z)"
```

Ordered pairs can be defined as shown, where ⦃a, b⦄ abbreviates `hinsert a (hinsert b hempty)`.

```
definition hpair :: "hf ⇒ hf ⇒ hf"
  where "hpair a b = ⦃⦃a,a⦄,⦃a,b⦄⦄"
```

Task 2 *Prove that ordered pairing is injective.*

```
lemma "hpair a b = hpair a' b' ⟷ a=a' & b=b'"
```

Task 3 *Prove that it is always possible to delete a given element* `x` *from a given set* `z`*. (That is, there exists a set* `u` *standing for the result of this deletion.)*

```
lemma delete1:
   "hmem x z ==> ∃u. hinsert x u = z & ¬ hmem x u"
```

Task 4 *Prove the existence of the union of two given sets* `x` *and* `y`*.*

```
lemma binary_union: "∃z. ∀u. hmem u z ⟷ hmem u x | hmem u y"
```

Task 5 *Prove the existence of the union of given a set* `X` *of sets.*

```
lemma union_of_set: "∃z. ∀u. hmem u z ⟷ (∃y. hmem y X & hmem u y)"
```

Task 6 *Define the subset relation as shown, and prove the following theorem. (It is useful for proving the existence of power sets.)*

```
definition hsubset :: "hf ⇒ hf ⇒ bool"
  where "hsubset a b ⟷ (∀x. hmem x a ⟶ hmem x b)"

lemma hsubset_insert2_iff:
  "hsubset z (hinsert x y) ⟷
   hsubset z y ∨ (∃u. hinsert x u = z ∧ ¬ hmem x u ∧ hsubset u y)"
```