Interactive Formal Verification (L21)
Exercises

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Lent Term, 2013

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, 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. 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, 19th February 2013
- 2nd exercise: Thursday, 7th March 2013

Please deliver a printed copy of each completed exercise to student administration by 12 noon on the given date, 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 Binary Search Trees

3.1 Preliminaries

We begin by declaring a binary tree with labelled nodes.

```
datatype tree = Lf | Br int tree tree
```

A *binary search tree* is a binary tree with an ordering constraint: at every node, the label is greater than every label in the left subtree, and smaller than every label in the right subtree (both inequalities strict).

The `update` operation for a binary search tree can be declared as follows. Note the use of the ordering constraint to recursively update the correct subtree.

```
fun update :: "tree => int => tree"
  where
   "update Lf v = Br v Lf Lf"
 | "update (Br w t1 t2) v =
      (if v<w then  Br w (update t1 v)  t2
       else
       if w<v then  Br w  t1  (update t2 v)
       else  Br v t1 t2)"
```

Task 1 *Define an Isabelle function* `lookup` *of type* `tree ⇒ int ⇒ bool`*, so as to return* `True` *if a given label occurs in a binary search tree. It should take advantage of the ordering constraint analogously to* `update`*. [4 marks]*

Task 2 *Define an Isabelle function* `labels` *of type* `tree ⇒ int set`*, to return the set of labels in a binary tree. Also define an Isabelle function* `ordered` *of type* `tree ⇒ bool`*, specifying the ordering constraint for binary search trees. [6 marks]*

Task 3 *Prove the following theorems. [15 marks]*

```
lemma labels_update: "labels (update t v) = {v} ∪ labels t"
lemma ordered_update: "ordered (update t v) ⟷ ordered t"
lemma ordered_lookup: "ordered t ⟹ lookup t v ⟷ v ∈ labels t"
lemma lookup_update:
      "ordered t ⟹ lookup (update t v) w = (v=w ∨ lookup t w)"
```

3.2 Additional Functions on Trees

The remainder of this exercise involves the function `gax`:

```
fun gax :: "tree => tree => tree"
  where
   "gax Lf t = t"
 | "gax (Br v t1 t2) t = gax t1 (gax t2 (update t v))"
```

Task 4 *Prove the following theorems, first replacing the function* `f` *in the first one by something more appropriate. Thus explain what* `gax` *actually does.* *[10 marks]*

```
lemma labels_gax: "labels (gax t1 t2) = f (labels t1, labels t2)"
lemma ordered_gax: "ordered (gax t1 t2) = ordered t2"
```

Task 5 *Define an Isabelle function* `delete` *of type* `tree ⇒ int ⇒ tree`*, to delete a label from a binary tree. (*Hint*: you may find the function* `gax` *helpful.) Then, prove the following theorems.* *[15 marks]*

```
lemma labels_delete:
      "ordered t ⟹ labels (delete t v) = labels t - {v}"
lemma ordered_delete: "ordered t ⟹ ordered (delete t v)"
```

Note: If your solution generalises the type of trees to be polymorphic, rather than having integer labels, then you will gain 5–10 marks in the category of "expertise". This generalisation requires the use of an appropriate axiomatic type class.

4 Finite State Machines

A nondeterministic *finite state machine*, as formalised below, has a set of initial states (where the machine may start execution), a set of final states (indicating acceptance of a string) and finally a next state relation.

Type `'a` denotes the set of states, while type `'b` represents the alphabet. The next state relation represents labelled transitions, $s \xrightarrow{c} s'$. It holds whenever it is possible to go from state s to state s' reading the character c.

```
record ('a,'b) fsm = init  :: "'a set"
                     final :: "'a set"
                     nxt :: "'a  => 'b => 'a => bool"
```

The definition above uses Isabelle records, which have not been covered in the course. As used here, they are very simple. A field of a record `r` is designated using the field name as a function, as in `init r`. Records are further described in the Tutorial (section 8.2), but it is permissible to replace the record by a 3-tuple in this exercise.

Task 1 *Define an Isabelle function* `reaches` *of type* `('a, 'b) fsm ⇒ 'a ⇒ 'b list ⇒ 'a ⇒ bool` *generalising the next state relation to the that of* reachability, $s \xrightarrow{l}{}^{*} s'$, *where* l *is a list of characters. Therefore* `reaches` *should satisfy*

- $s \xrightarrow{[]}{}^{*} s$ *(with the empty list, a state* s *is reachable from itself), and*
- $s \xrightarrow{(c\#l)}{}^{*} s'$ *if* $s \xrightarrow{c} s''$ *and* $s'' \xrightarrow{l}{}^{*} s'$ *for som intermediate state* s''.

(Note that a string of characters has type `'b`*.)* *[5 marks]*

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

```
lemma reaches_append_iff:
     "reaches fsm s (xs@ys) s' ⟷
        (∃s''. reaches fsm s xs s'' ∧ reaches fsm s'' ys s')"
```

A finite state machine *accepts* a given string provided some final state is reachable from some initial state via that string.

```
definition accepts :: "('a,'b) fsm => 'b list => bool"  where
  "accepts fsm xs ==
```

```
    ∃s s'. reaches fsm s xs s' ∧ s ∈ init fsm ∧ s' ∈ final fsm"
```

For example, we can define a trivial finite state machine that does not accept any strings. The proof of the theorem is trivial. This example also illustrates the syntax for writing record expressions. Note the special brackets used to enclose the record fields.

```
definition Null where
  "Null = (|init = {}, final = {}, nxt = λst x st'. False|)"

lemma accepts_empty:  "~ accepts Null l"
```

Task 3 *Prove the following theorem, for a suitable definition of the function* `SingStr`*. This should create a machine that accepts only a string consisting of the given single character.* *[10 marks]*

```
lemma accepts_singstr: "accepts (SingStr a) l ⟷ l = [a]"
```

Task 4 *Prove the following theorem, for a suitable definition of the function* `Reverse`*. This should create a machine that reverses the direction of all transitions in a given machine. (You may find it helpful to prove a lemma relating the functions* `reaches` *and* `Reverse`*.)* *[12 marks]*

```
lemma accepts_rev: "accepts fsm l ⟹ accepts (Reverse fsm) (rev l)"
```

Task 5 *Prove the following theorem, for a suitable definition of the function* `Times`*. This should create a machine that runs to other given machines in parallel. Each state of this machine should be an ordered pair. The point of this construction is to define the intersection of two languages. (You will probably need Isabelle's operator for forming the Cartesian product of two sets, written* `A × B`*. .)* *[18 marks]*

```
lemma accepts_Times_iff:
    "accepts (Times fsm1 fsm2) xs ⟷
     accepts fsm1 xs ∧ accepts fsm2 xs"
```

Remark: This material represents part of a development of the theory of regular languages. In fact it needs much work before it can be useful in that guise. A regular language is normally defined to be any language accepted by some finite state machine. Unfortunately, with our definitions, the set of states of a machine affects its type. So the relevant definition cannot be expressed. To develop this material further requires a different treatment of machine states.