Interactive Formal Verification (L21)
Exercises and Marking Scheme

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Michaelmas Term, 2018

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/. Many of these on-line examples are easy: the assessed exercises are considerably harder. You are strongly encouraged to attempt a variety of exercises, and perhaps make up your own.

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 proof and include notes about your experience in completing it.

[1]See section 4.2 of the Isabelle/HOL Tutorial, https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2018/doc/tutorial.pdf.

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

- 1st exercise: Wednesday, 21 November 2018
- 2nd exercise: Friday, 30 November 2018

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: Propositional Modal Logic

Modal logic is a formalism where statements can be possibly or necessarily true. Its semantics is specified with respect to a set of *possible worlds* along with an *accessibility relation* between worlds. Background material on modal logic is widely available; see for example my *Logic and Proof* lecture notes, pages 28–29.[2]

Suppose we are given infinitely many propositional letters P_0, P_1, ..., indexed by the natural numbers. Then the formulas of propositional modal logic are built up from the atomic formulas P_n using disjunction ($A \lor B$), negation ($\neg A$) and necessitation ($\Box A$).

Task 1 *Write a datatype declaration of formulas with the four cases above. Then introduce definitions or abbreviations for conjunction as $\neg(\neg A \lor \neg B)$, implication as $\neg A \lor B$ and the diamond operator $\Diamond A$ as $\neg(\Box\neg A)$. [5 marks]*

Next we define the semantics of propositional modal logic. We identify the set of possible worlds with the type of natural numbers.

```
type_synonym world = nat
```

An *interpretation* maps the propositional letter P_n to the set of worlds in which P_n is true. The semantics of a formula is a function of an accessibility relation R, an interpretation I and a world w. The formula $\Box A$ is true in world w if and only if, for all v such that $(w, v) \in R$, formula A is true in world v. Disjunctions and negations are treated using standard Boolean truth tables.

Task 2 *Define the predicate* `isTrue R I w A` *corresponding to the description above. [5 marks]*

The notion of a valid formula takes a parameter R, the accessibility relation. A formula is valid (with respect to R) if it is true for every interpretation and world.

```
definition valid where "valid R A ≡ ∀I w. isTrue R I w A"
```

Task 3 *Prove the following. The first three should be trivial and count as sanity checks for your answers so far. [5 marks]*

```
lemma "valid R (Box (A OR Neg A))"
lemma "valid R (Box (((A IMP B) IMP A) IMP A))"
lemma "valid R (Box (A IMP B) IMP (Box A IMP Box B))"
```

[2] https://www.cl.cam.ac.uk/teaching/1819/LogicProof/logic-notes.pdf

```
lemma "∃R A. ¬ valid R (Box A IMP A)"
lemma "∃R. ∀A. valid R (Box A IMP A)"
```

Deductive systems for modal logics include the standard axioms and rules for propositional logic to which is added the *distribution* axiom

$$\Box(A \to B) \to (\Box A \to \Box B),$$

as in the previous task.[3] A variety of other axioms are sometimes added, singly or in combination. Some of these are defined below.

```
abbreviation axT where "axT A ≡ Box A IMP A"
abbreviation axB where "axB A ≡ A IMP Box(Dia A)"
abbreviation ax4 where "ax4 A ≡ Box A IMP Box(Box A)"
abbreviation ax5 where "ax5 A ≡ Dia A IMP Box(Dia A)"
abbreviation axR where "axR A ≡ Box (Box A) IMP Box A"
```

Each of these axioms turns out to characterise some property of the accessibility relation, such as being reflexive, symmetric, transitive, Euclidean or dense. The latter two properties are defined below:

```
definition Euclidean
  where "Euclidean r ≡ ∀x y z. (x,y) ∈ r ⟶ (x,z) ∈ r ⟶ (y,z) ∈ r"
definition dense
  where "dense r ≡ ∀x y. (x,y) ∈ r ⟶ (∃z. (x,z) ∈ r ∧ (z,y) ∈ r)"
```

Task 4 *Prove the following results. (The proofs are similar.)* *[10 marks]*

```
lemma refl_iff_T: "refl R ⟷ (∀A. valid R (axT A))"
lemma trans_iff_4: "trans R ⟷ (∀A. valid R (ax4 A))"
```

Task 5 *Prove the following two results.* *[10 marks]*

```
lemma sym_iff_B: "sym R ⟷ (∀A. valid R (axB A))"
lemma Euclidean_iff_5: "Euclidean R ⟷ (∀A. valid R (ax5 A))"
```

Task 6 *Prove the following result.* *[15 marks]*

```
lemma dense_iff_R: "dense R ⟷ (∀A. valid R (axR A))"
```

Hint: no proof should require more than 20 lines. When proving properties of relations, it's enough to consider atomic formulas and a specific, simple interpretation.

[3] There is also a rule to infer $\Box A$ from A.

4 Assessed Exercise II: Partitions of a Set

A *partition* of a set A is a pairwise disjoint family $\mathcal{B}$ of nonempty sets whose union equals A, i.e. $\bigcup \mathcal{B} = A$.

Task 1 *Define the predicate* `partitions A 𝓑` *to hold precisely when* $\mathcal{B}$ *is a partition of* A. *[5 marks]*

Hint: the built-in constants `pairwise` and `disjnt` can be used to express your definition more succinctly and may facilitate reasoning about it. You can use the query panel to locate facts about these primitives. Script characters like $\mathcal{B}$ can be found in the Symbols panel, Letter tab, if you wish to use them. It would be wise to test your definition by executing examples such as those below (provable by `eval` alone).

```
lemma "partitions {0,1,2,3} {{0,2},{1},{3::int}}"
lemma "¬ partitions {0,1,2,3} {{0,2,3},{1},{3::int}}"
lemma "¬ partitions {0,1,2,3} {{0},{1},{3::int}}"
lemma "¬ partitions {0,1,2,3} {{0,2},{},{1,3::int}}"
```

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

```
lemma partitions_empty1 [simp]: "partitions {} 𝓑 ⟷ 𝓑 = {}"
lemma partitions_empty2 [simp]: "partitions A {} ⟷ A = {}"
lemma partitions_insert:
  assumes "partitions A 𝓑" "a ∉ A"
  shows "partitions (insert a A) (insert {a} 𝓑)"
lemma partitions_insert2:
  assumes "partitions A 𝓑" "B ∈ 𝓑" "a ∉ A"
  shows "partitions (insert a A) (insert (insert a B) (𝓑 - {B}))"
```

As we see from the results above, there are two ways to obtain a partition of the set `insert a A` (where `a ∉ A`) from a partition $\mathcal{B}$ of `A`: we can insert the singleton set `{a}`, or else we can insert `a` into some member `B` of the partition. Now we work towards proving that these are the only two ways by reversing those constructions.

Task 3 *Prove the following:* *[20 marks]*

```
lemma Union_diff_sing:
  assumes "A ∈ 𝓐" "pairwise disjnt 𝓐"
  shows " ⋃(𝓐 - {A}) = Union 𝓐 - A"
lemma partitions_diff_sing:
  assumes "partitions (insert a A) 𝓑" "{a} ∈ 𝓑" "a ∉ A"
  shows "partitions A (𝓑 - {{a}})"
```

```
lemma partitions_diff_insert:
  assumes "partitions (insert a A) 𝓑" "insert a C ∈ 𝓑"
          "C ≠ {}" "a ∉ C" "a ∉ A"
  shows "partitions A (insert C (𝓑 - {insert a C}))"
```

Task 4 *Finally, prove this result, giving necessary and sufficient conditions for* $\mathcal{B}$ *to a partition of* `insert a A`*.* *[20 marks]*

```
lemma partitions_insert_iff:
  assumes "a ∉ A"
  shows "partitions (insert a A) 𝓑
     ⟷ (∃𝓒. partitions A 𝓒 ∧ (𝓑 = insert {a} 𝓒
       ∨ (∃C∈𝓒. 𝓑 = insert (insert a C) (𝓒-{C}))))"
```