Interactive Formal Verification (L21)
Exercises and Marking Scheme

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Michaelmas Term, 2020

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/, but they tend to be easy. The assessed exercises are considerably harder, as you can see by looking at those of previous years.

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. Avoid inserting **apply** commands before the **proof** keyword.

- 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

[1] See section 4.2 of the Isabelle/HOL Tutorial, https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle2019/doc/tutorial.pdf.

perhaps outline the strategic decisions that affected the shape of your proof and include notes about your experience in completing it. Please don't copy the text of the exercises into your own write-up.

- 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. Taking the effort to make your proofs more readable can help. Even better, strive for proofs that are direct and insightful; untidy or circuitous proofs and needless complexity can lose marks.

  An exceptional write-up also gains a few marks in this category. 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, 4 November 2020
- 2nd exercise: Wednesday, 18 November 2020

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: Semantics of a Functional Language

This exercise concerns a tiny functional programming language with a conditional expression and an equality test. The language admits nonsensical terms such as `Succ T`, but they do no harm. We prove some elementary semantics theorems:

- each evaluation step preserves the type and value of an expression
- a Church–Rosser property, that expression evaluation always leads to a unique final result

We define the language's abstract syntax as a datatype.

```
datatype exp = T | F | Zero | Succ exp | IF exp exp exp | EQ exp exp
```

We define a one-step semantics inductively. Note the three evaluation rules for `IF`. For `Succ x` the only possibility is to reduce `x`, but for `EQ x y` there are six possibilities. No order is imposed on which argument to reduce first or even to ensure that one evaluation finishes before the other starts.

```
inductive Eval :: "exp ⇒ exp ⇒ bool" (infix "⇛" 50) where
    IF_T:    "IF T x y ⇛ x"
  | IF_F:    "IF F x y ⇛ y"
  | IF_Eval: "p ⇛ q ⟹ IF p x y ⇛ IF q x y"
  | Succ_Eval: "x ⇛ y ⟹ Succ x ⇛ Succ y"
  | EQ_same: "EQ x x ⇛ T"
  | EQ_S0:   "EQ (Succ x) Zero ⇛ F"
  | EQ_0S:   "EQ Zero (Succ y) ⇛ F"
  | EQ_SS:   "EQ (Succ x) (Succ y) ⇛ EQ x y"
  | EQ_Eval1: "x ⇛ z ⟹ EQ x y ⇛ EQ z y"
  | EQ_Eval2: "y ⇛ z ⟹ EQ x y ⇛ EQ x z"
```

The language has two types, for the booleans and the natural numbers.

```
datatype tp = bool | num
```

The typing relation is defined inductively. A conditional expression can return a result of either type.

```
inductive TP :: "exp ⇒ tp ⇒ bool" where
  T:    "TP T bool"
| F:    "TP F bool"
| Zero: "TP Zero num"
| IF:   "⟦TP p bool; TP x t; TP y t⟧ ⟹ TP (IF p x y) t"
| Succ: "TP x num ⟹ TP (Succ x) num"
| EQ:   "⟦TP x t; TP y t⟧ ⟹ TP (EQ x y) bool"
```

Task 1 *Define an evaluation function* `evl :: "exp ⇒ nat"` *mapping* `T` *to one,* `F` *to zero, and is otherwise consistent with the evaluation relation* `(⇛)`*. Prove that it is well-defined as expressed by the following proposition.* *[5 marks]*

proposition `value_preservation:`
assumes `"x ⇛ y" "TP x t"` **shows** `"evl x = evl y"`

Task 2 *Prove the following two results. The first concerns the value of a boolean (as an integer), while* type preservation *is a key property expected of any typed programming language.* *[10 marks]*

lemma
assumes `"TP x t" "t = bool"` **shows** `"evl x < 2"`

proposition `type_preservation:`
assumes `"x ⇛ y" "TP x t"` **shows** `"TP y t"`

Task 3 *The following claim is a form of Church–Rosser property. Find a counterexample and briefly comment on the underlying cause.* *[5 marks]*

lemma
assumes `"x ⇛ y" "x ⇛ z"` **shows** `"∃u. y ⇛ u ∧ z ⇛ u"`

Task 4 *Define mult-step evaluation, either inductively or otherwise, such that* `x ⇛* y` *holds provided there is a chain of zero or more* `⇛`*-steps from* `x` *to* `y`*. Then prove the following two results:* *[10 marks]*

proposition `type_preservation_Star:`
assumes `"x ⇛* y" "TP x t"` **shows** `"TP y t"`

lemma `Succ_EvalStar:`
assumes `"x ⇛* y"` **shows** `"Succ x ⇛* Succ y"`

Task 5 *Prove the following Church–Rosser property.* *[20 marks]*

proposition `diamond:`
assumes `"x ⇛ y" "x ⇛ z"` **shows** `"∃u. y ⇛* u ∧ z ⇛* u"`

No proof should require more than 25 lines. It may help to use **inductive_simps** and to prove some lemmas by induction.

4 Assessed Exercise II: Sums of Divisors

This exercise concerns some simple properties of the set of divisors of a natural number, as defined below. Note that an **abbreviation** is mathematically equivalent to a **definition** but always expands mathematically.

```
definition divisors :: "nat ⇒ nat set" where
    "divisors m ≡ {n. n dvd m}"

abbreviation sigma :: "nat ⇒ nat" where
    "sigma m ≡ ∑ (divisors(m))"
```

Task 1 *As a warmup, prove the following simple consequences of the definitions.* *[2 marks]*

```
lemma finite_divisors: "n>0 ⟹ finite (divisors n)"

lemma prime_divisors: "prime p ⟷ divisors p = {1,p} ∧ p>1"
```

Task 2 *For a prime number we have* `divisors p = {1,p}`*. Prove the following lemma, concerning the situation when there exists a third divisor.* *[5 marks]*

```
lemma sigma_third_divisor:
  assumes "1 < a" "a < n" "a dvd n"
  shows "1+a+n ≤ sigma(n)"
```

Task 3 *The property* `sigma n = Suc n` *turns out to characterise the prime numbers. Prove it.* *[15 marks]*

```
proposition prime_iff_sigma: "prime n ⟷ sigma(n) = Suc n"
```

Task 4 *The divisors of p^n have the form p^k for $k \leq n$. Prove it.* *[3 marks]*

```
lemma dvd_prime_power_iff:
  fixes p::nat
  assumes prime: "prime p"
  shows "{d. d dvd p^n} = (λk. p^k) ‘ {0..n}"
```

Task 5 *Prove the following lemma, which will be useful below.* *[10 marks]*

```
lemma prodsums_eq_sumprods:
  fixes p :: nat and m :: nat
  assumes "coprime p m"
  shows "∑((λk. p^k) ` {0..n}) * sigma m
        = ∑{p^k * b |k b. k ≤ n ∧ b dvd m}"
```

Task 6 *Conclude by proving this interesting distributive law for* `sigma`. *[15 marks]*

```
theorem
  assumes "prime p" "coprime p m"
  shows "sigma (p^n) * sigma m = sigma (p^n * m)"
```