# Interactive Formal Verification (L21)
# Exercises

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Lent Term, 2015

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.

- 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.

- 20 marks are for a clear, basic write-up. It can be just a few pages, and 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.

- 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.

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 deadline dates. Exercises are due at 12 NOON.

- 1st exercise: Tuesday, 17th February 2015
- 2nd exercise: Thursday, 5th March 2015

Please deliver a printed copy of each completed exercise to student administration, 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"`

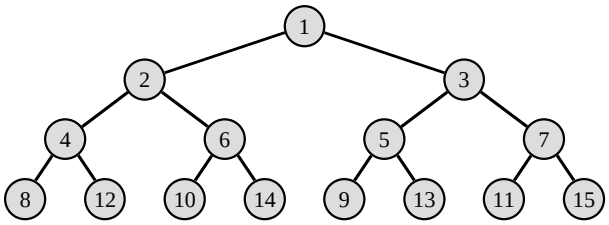
What is the relationship between `sum` and `Sum`? Prove the following equation, suitably instantiated.

**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: Functional Arrays

Braun trees implement functional arrays subscripted by positive integers, based on their representation as binary numbers. Naturally enough, the underlying data structure is the binary tree. A location in the tree is found by starting at the root, testing whether the subscript is even or odd, and descending into the left or right subtree, respectively; this process terminates when 1 is reached. The numbers in the diagram are not the labels of branch nodes, but indicate the subscript positions of array elements where data is stored.




More information can be found in the lecture notes for the Part IA course *Foundations of Computer Science*, starting at page 80. See also *ML for the Working Programmer* [1], page 154.

To start off, we declare the datatype of binary trees and the `lookup` function, which returns the label of a functional array designated by a subscript. Observe the test for an even number and the use of division by two, which in the case of an odd number, discards the remainder.

```
datatype 'a tree = Lf | Br "'a" "'a tree" "'a tree"

fun lookup :: "'a tree ⇒ nat ⇒ 'a option"
  where
   "lookup Lf k = None"
 | "lookup (Br v t1 t2) k =
      (if k=1 then Some v
       else
       if even k then  lookup t1 (k div 2)
                 else  lookup t2 (k div 2))"
```

The `update` operation for a functional array is declared as follows. This function can also extend the array, but only if this can be done without inserting intermediate elements: if it reaches the fringe of the tree then it does nothing unless `k=1`. Provided we extend the tree with consecutive subscripts, there will be no gaps and the tree will be strongly balanced: the size of any right subtree is less than or equal to that of the left and the difference is no greater than one.

```
fun update :: "'a tree ⇒ nat ⇒ 'a ⇒ 'a tree"
```

```
  where
   "update Lf k v = (if k=1 then Br v Lf Lf else Lf)"
 | "update (Br w t1 t2) k v =
     (if k=1 then Br v t1 t2
      else
      if even k then  Br w (update t1 (k div 2) v)  t2
      else  Br w  t1  (update t2 (k div 2) v))"
```

**Task 1** *Define an Isabelle function* `defined` *of type* `'a tree ⇒ nat set`*, to return the set of set of defined subscripts in a binary tree. The outer form is given below, and you only need to define the function* `dpl`*.* *[7 marks]*

```
fun defined :: "'a tree ⇒ nat set"
  where
   "defined Lf = {}"
 | "defined (Br v t1 t2) = dpl (defined t1) (defined t2)"
```

**Task 2** *Prove the following theorems about* `lookup`*.* *[10 marks]*

```
lemma lookup_None_eq: "lookup t k = None ⟷ k ∉ defined t"
lemma lookup_Some_eq: "k ∈ defined t ⟷ (∃v. lookup t k = Some v)"
```

**Task 3** *Prove the following theorems about* `update`*.* *[10 marks]*

```
lemma df_subset_df_update: "defined t ⊆ defined (update t k v)"
lemma df_upd_subs: "defined (update t k v) ⊆ insert k (defined t)"
```

**Task 4** *Prove the following lemmas, which will be useful for the next task.* *[6 marks]*

```
lemma dpl_D1: "{Suc 0..<2*m} ⊆ dpl A1 A2 ⟹ {Suc 0..<m} ⊆ A1"
lemma dpl_D2: "{Suc 0..<Suc(2*m)} ⊆ dpl A1 A2 ⟹ {Suc 0..<m} ⊆ A2"
```

**Task 5** *Prove the following theorems, which show that* `update` *defines a new array element under suitable conditions.* *[17 marks]*

```
lemma df_imp_df_update:
   "⟦0<k; {1..<k} ⊆ defined t⟧ ⟹ k ∈ defined (update t k v)"
lemma df_update:
   "⟦0<k; {1..<k} ⊆ defined t⟧
    ⟹ defined (update t k v) = insert k (defined t)"
```

*Remark*: no proof should require more than a dozen lines. However, the difficulty of a proof depends critically on the overall strategy. Carefully consider which induction rule to use. You may need to prove some lemmas to help simplify expressions involving division by 2 or to reason about even and odd numbers sensibly.

# 4 Assessed Exercise II: 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. The theory `HOL/Number_Theory/Binomial` contains basic definitions and proofs. Information about binomial coefficients. is widely available on the Internet, including the lecture course notes available here:

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

**Task 1** *Prove the following theorem about binomial coefficients. (Note that* `0 - 1 = 0` *on the natural numbers.)* *[5 marks]*

**lemma** `times_binomial:`
```
  "Suc k * (n choose Suc k) = n * ((n - 1) choose k)"
```

**Task 2** *Prove the following theorem, which relates to a weighted sum of a row of Pascal's triangle. (It involves arithmetic on type* `real` *as well as* `nat`*, so the function* `real` *is implicitly inserted at multiple points.)* *[15 marks]*

**lemma** `choose_row_sum:`
```
  "(∑k = 0..m. (r choose k) * (r/2 - k)) =
   (Suc m) /2 * (r choose (Suc m))"
```

**Task 3** *Prove the following lemma, which will be useful below.* *[10 marks]*

**lemma** `setsum_choose_drop_zero:`
```
 "(∑k = 0..Suc n. if k=0 then 0 else (Suc n - k) choose (k - 1)) =
  (∑j = 0..n. (n-j) choose j)"
```

**Task 4** *Prove the following theorem, which relates binomial coefficients with Fibonacci numbers.* *[20 marks]*

**lemma** `ne_diagonal_fib:`
```
   "(∑k = 0..n. (n-k) choose k) = fib (Suc n)"
```

*Remark*: as in Assessed Exercise I, choose your induction rule with care in each proof. The precise statement of the induction formula is also important. Some of the proofs are best done using calculational reasoning.

## References

[1] Lawrence C. Paulson. *ML for the Working Programmer*. 2nd edition, 1996.