

Isabelle/HOL Exercises

Gertrud Bauer, Gerwin Klein, Tobias Nipkow,
Michael Wahler, Markus Wenzel

September 3, 2002

This document presents a collection of exercises for getting acquainted with the proof assistant Isabelle/HOL [1]. The exercises come out of an annual Isabelle/HOL course taught at the Technical University of Munich. They are arranged in chronological order, and in each year in ascending order of difficulty.

Contents

1	2000	2
1.1	Lists	2
1.2	Arithmetic	2
1.3	The towers of Hanoi	3
2	2001	4
2.1	Lists	4
2.2	Trees	4
2.3	Tries	5
2.4	Interval lists	7
3	2002	10
3.1	Lists	10
3.2	Sorting	10
3.3	Compilation	12
3.4	Merge sort	13
3.5	The towers of Hanoi	13

1 2000

1.1 Lists

Define a primitive recursive function `snoc` that appends an element at the *right* end of a list. Do not use `@` itself.

consts

```
snoc :: "'a list => 'a => 'a list"
```

Prove the following theorem:

theorem rev_cons: `"rev (x # xs) = snoc (rev xs) x"`

Hint: you need to prove a suitable lemma first.

1.2 Arithmetic

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`.

Summation

Define a (primitive recursive) function `sum ns` that sums a list of natural numbers:
 $sum[n_1, \dots, n_k] = n_1 + \dots + n_k$.

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 $k-1$: $Sum f k = f 0 + \dots + f(k-1)$.

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 + 1) = Sum f k + Sum whatever 1"

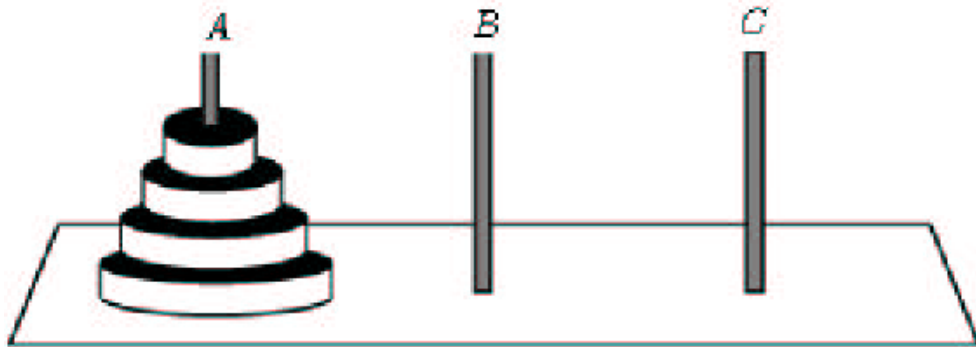
What is the relationship between *sum* and *Sum*? Prove the following equation, suitably instantiated.

theorem "Sum f k = sum whatever"

Hint: familiarize yourself with the predefined functions *map* and *[i..j]* on lists in theory List.

1.3 The towers of Hanoi

We are given 3 pegs *A*, *B* and *C*, and *n* disks with a hole, such that no two disks have the same diameter. Initially all *n* disks rest on peg *A*, ordered according to their size, with the largest one at the bottom. The aim is to transfer all *n* disks from *A* to *C* by a sequence of single-disk moves such that we never place a larger disk on top of a smaller one. Peg *B* may be used for intermediate storage.



The pegs and moves can be modelled as follows:

```
datatype peg = A | B | C
types move = "peg * peg"
```

Define a primitive recursive function

```
consts
  moves :: "nat => peg => peg => peg => move list"
```

such that *moves n a b c* returns a list of (legal) moves that transfer *n* disks from peg *a* via *b* to *c*. Show that this requires $2^n - 1$ moves:

theorem "length (moves n a b c) = $2^n - 1$ "

Hint: You need to strengthen the theorem for the induction to go through. Beware: subtraction on natural numbers behaves oddly: $n - m = 0$ if $n \leq m$.

2 2001

2.1 Lists

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 counter example) 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 counter example) 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)"
```

2.2 Trees

In the sequel we work with skeletons of binary trees where neither the leaves (“tip”) nor the nodes contain any information:

```
datatype tree = Tp | Nd tree tree
```

Define a function `tips` that counts the tips of a tree, and a function `height` that computes the height of a tree.

Complete binary trees of a given height are generated as follows:

```
consts cbt :: "nat ⇒ tree"
```

```
primrec
```

```
"cbt 0 = Tp"
"cbt(Suc n) = Nd (cbt n) (cbt n)"
```

We will now focus on these complete binary trees.

Instead of generating complete binary trees, we can also *test* if a binary tree is complete. Define a function *iscbt f* (where *f* is a function on trees) that checks for completeness: *Tp* is complete and *Nd l r* is complete iff *l* and *r* are complete and *f l = f r*.

We now have 3 functions on trees, namely *tips*, *height* und *size*. The latter is defined automatically — look it up in the tutorial. Thus we also have 3 kinds of completeness: complete wrt. *tips*, complete wrt. *height* and complete wrt. *size*. Show that

- the 3 notions are the same (e.g. *iscbt tips t = iscbt size t*), and
- the 3 notions describe exactly the trees generated by *cbt*: the result of *cbt* is complete (in the sense of *iscbt*, wrt. any function on trees), and if a tree is complete in the sense of *iscbt*, it is the result of *cbt* (applied to a suitable number — which one?)

Find a function *f* such that *iscbt f* is different from *iscbt size*.

Hints:

- Work out and prove suitable relationships between *tips*, *height* und *size*.
- If you need lemmas dealing only with the basic arithmetic operations (+, *, ^ etc), you can “prove” them with the command *sorry*, if neither *arith* nor you can find a proof. Not *apply sorry*, just *sorry*.
- You do not need to show that every notion is equal to every other notion. It suffices to show that $A = C$ und $B = C \implies A = B$ is a trivial consequence. However, the difficulty of the proof will depend on which of the equivalences you prove.
- There is \wedge and \implies .

2.3 Tries

Section 3.4.4 of [1] is a case study about so-called *tries*, a data structure for fast indexing with strings. Read that section.

The data type of tries over the alphabet type *'a* und the value type *'v* is defined as follows:

```
datatype ('a, 'v) trie = Trie "'v option" "('a * ('a, 'v) trie) list"
```

A trie consists of an optional value and an association list that maps letters of the alphabet to subtrees. Type *'a option* is defined in section 2.5.3 of [1].

There are also two selector functions *value* und *alist*:

```

consts value :: "('a, 'v) trie  $\Rightarrow$  'v option"
primrec "value (Trie ov al) = ov"

```

```

consts alist :: "('a, 'v) trie  $\Rightarrow$  ('a * ('a, 'v) trie) list"
primrec "alist (Trie ov al) = al"

```

Furthermore there is a function `lookup` on tries defined with the help of the generic search function `assoc` on association lists:

```

consts assoc :: "('key * 'val)list  $\Rightarrow$  'key  $\Rightarrow$  'val option"
primrec "assoc [] x = None"
      "assoc (p#ps) x =
        (let (a, b) = p in if a = x then Some b else assoc ps x)"

```

```

consts lookup :: "('a, 'v) trie  $\Rightarrow$  'a list  $\Rightarrow$  'v option"
primrec "lookup t [] = value t"
      "lookup t (a#as) = (case assoc (alist t) a of
                          None  $\Rightarrow$  None
                          | Some at  $\Rightarrow$  lookup at as)"

```

Finally, `update` updates the value associated with some string with a new value, overwriting the old one:

```

consts update :: "('a, 'v) trie  $\Rightarrow$  'a list  $\Rightarrow$  'v  $\Rightarrow$  ('a, 'v) trie"
primrec
  "update t [] v = Trie (Some v) (alist t)"
  "update t (a#as) v =
    (let tt = (case assoc (alist t) a of
              None  $\Rightarrow$  Trie None []
              | Some at  $\Rightarrow$  at)
     in Trie (value t) ((a, update tt as v) # alist t))"

```

The following theorem tells us that `update` behaves as expected:

```

theorem " $\forall$  t v bs. lookup (update t as v) bs =
          (if as = bs then Some v else lookup t bs)"

```

As a warming up exercise, define a function

```

consts modify :: "('a, 'v) trie  $\Rightarrow$  'a list  $\Rightarrow$  'v option  $\Rightarrow$  ('a, 'v) trie"

```

for inserting as well as deleting elements from a trie. Show that `modify` satisfies a suitably modified version of the correctness theorem for `update`.

The above definition of `update` has the disadvantage that it often creates junk: each association list it passes through is extended at the left end with a new (letter,value) pair without removing any old association of that letter which may already be present. Logically, such cleaning up is unnecessary because `assoc` always searches the list from the

left. However, it constitutes a space leak: the old associations cannot be garbage collected because physically they are still reachable. This problem can be solved by means of a function

```
consts overwrite :: "'a ⇒ 'b ⇒ ('a * 'b) list ⇒ ('a * 'b) list"
```

that does not just add new pairs at the front but replaces old associations by new pairs if possible.

Define *overwrite*, modify *modify* to employ *overwrite*, and show the same relationship between *modify* and *lookup* as before.

Instead of association lists we can also use partial functions that map letters to subtrees. Partiality can be modelled with the help of type *'a option*: if *f* is a function of type *'a ⇒ 'b option*, set *f a = Some b* if *a* should be mapped to some *b* and set *f a = None* otherwise.

```
datatype ('a, 'v) trie = Trie "'v option" "'a ⇒ ('a, 'v) trie option"
```

Modify the definitions of *lookup* and *modify* accordingly and show the same correctness theorem as above.

2.4 Interval lists

Sets of natural numbers can be implemented as lists of intervals, where an interval is simply a pair of numbers. For example the set $\{2, 3, 5, 7, 8, 9\}$ can be represented by the list $[(2, 3), (5, 5), (7, 9)]$. A typical application is the list of free blocks of dynamically allocated memory.

Definitions

We introduce the type

```
types intervals = "(nat*nat) list"
```

This type contains all possible lists of pairs of natural numbers, even those that we may not recognize as meaningful representations of sets. Thus you should introduce an *invariant*

```
consts inv :: "intervals => bool"
```

that characterizes exactly those interval lists representing sets. (The reason why we call this an invariant will become clear below.) For efficiency reasons intervals should be sorted in ascending order, the lower bound of each interval should be less or equal the upper bound, and the intervals should be chosen as large as possible, i.e. no two adjacent intervals should overlap or even touch each other. It turns out to be convenient to define *Aufgabe5.inv* in terms of a more general function

```
consts inv2 :: "nat => intervals => bool"
```

such that the additional argument is a lower bound for the intervals in the list.

To relate intervals back to sets define an *abstraktion funktion*

```
consts set_of :: "intervals => nat set"
```

that yields the set corresponding to an interval list (that satisfies the invariant).

Finally, define two primitive recursive functions

```
consts add :: "(nat*nat) => intervals => intervals"  
      rem :: "(nat*nat) => intervals => intervals"
```

for inserting and deleting an interval from an interval list. The result should again satisfies the invariant. Hence the name: *inv* is invariant under the application of the operations *add* and *rem* — if *inv* holds for the input, it must also hold for the output.

Proving the invariant

```
declare Let_def [simp]  
declare split_split [split]
```

Start off by proving the monotonicity of *inv2*:

```
lemma inv2_monotone: "inv2 m ins  $\implies$  n  $\leq$  m  $\implies$  inv2 n ins"
```

Now show that *add* and *rem* preserve the invariant:

```
theorem inv_add: "[[ i  $\leq$  j; inv ins ]  $\implies$  inv (add (i,j) ins)"  
theorem inv_rem: "[[ i  $\leq$  j; inv ins ]  $\implies$  inv (rem (i,j) ins)"
```

Hint: you should first prove a more general statement (involving *inv2*). This will probably involve some advanced forms of induction discussed in section 9.3.1 of [1].

Proving correctness of the implementation

Show the correctness of *add* and *rem* wrt. their counterparts \cup and $-$ on sets:

```
theorem set_of_add:  
  "[[ i  $\leq$  j; inv ins ]  $\implies$  set_of (add (i,j) ins) = set_of [(i,j)]  $\cup$  set_of ins"  
theorem set_of_rem:  
  "[[ i  $\leq$  j; inv ins ]  $\implies$  set_of (rem (i,j) ins) = set_of ins - set_of [(i,j)]"
```

Hints: in addition to the hints above, you may also find it useful to prove some relationship between *inv2* and *set_of* as a lemma.

General hints

- You should be familiar both with simplification and predicate calculus reasoning. Automatic tactics like *blast* and *force* can simplify the proof.

- Equality of two sets can often be proved via the rule *set_ext*:

$$(\wedge x. (x \in A) = (x \in B)) \implies A = B$$

- Potentially useful theorems for the simplification of sets include

$$Un_Diff: A \cup B - C = A - C \cup (B - C)$$

$$Diff_triv: A \cap B = \{\} \implies A - B = A.$$

- Theorems can be instantiated and simplified via *of* and *[simplified]* (see [1]).

3 2002

3.1 Lists

Define a universal and an existential quantifier on lists. Expression `alls P xs` should be true iff $P\ x$ holds for every element x of xs , and `exs P xs` should be true iff $P\ x$ holds for some element x of xs .

consts

```
alls :: "('a ⇒ bool) ⇒ 'a list ⇒ bool"
```

```
exs  :: "('a ⇒ bool) ⇒ 'a list ⇒ bool"
```

Prove or disprove (by counter example) the following theorems. You may have to prove some lemmas first.

Use the `[simp]`-attribute only if the equation is truly a simplification and is necessary for some later proof.

```
lemma "alls (λx. P x ∧ Q x) xs = (alls P xs ∧ alls Q xs)"
```

```
lemma "alls P (rev xs) = alls P xs"
```

```
lemma "exs (λx. P x ∧ Q x) xs = (exs P xs ∧ exs Q xs)"
```

```
lemma "exs P (map f xs) = exs (P o f) xs"
```

```
lemma "exs P (rev xs) = exs P xs"
```

Find a term Z such that the following equation holds:

```
lemma "exs (λx. P x ∨ Q x) xs = Z"
```

Express the existential via the universal quantifier — `exs` should not occur on the right-hand side:

```
lemma "exs P xs = Z"
```

Define a function `is_in x xs` that checks if x occurs in xs vorkommt. Now express `is_in` via `exs`:

```
lemma "is_in a xs = Z"
```

Define a function `nodups xs` that is true iff xs does not contain duplicates, and a function `deldups xs` that removes all duplicates. Note that `deldups [x, y, x]` (where x and y are distinct) can be either `[x, y]` or `[y, x]`.

Prove or disprove (by counter example) the following theorems.

```
lemma "length (deldups xs) <= length xs"
```

```
lemma "nodups (deldups xs)"
```

```
lemma "deldups (rev xs) = rev (deldups xs)"
```

3.2 Sorting

For simplicity we sort natural numbers.

Sorting with lists

The task is to define insertion sort and prove its correctness. The following functions are required:

consts

```
insert :: "nat  $\Rightarrow$  nat list  $\Rightarrow$  nat list"  
sort   :: "nat list  $\Rightarrow$  nat list"  
le    :: "nat  $\Rightarrow$  nat list  $\Rightarrow$  bool"  
sorted :: "nat list  $\Rightarrow$  bool"
```

In your definition, *insert* *x xs* should insert a number *x* into an already sorted list *xs*, and *sort* *ys* should build on *insert* to produce the sorted version of *ys*.

To show that the resulting list is indeed sorted we need a predicate *sorted* that checks if each element in the list is less or equal to the following ones; *le n xs* should be true iff *n* is less or equal to all elements of *xs*.

Start out by showing a monotonicity property of *le*. For technical reasons the lemma should be phrased as follows:

```
lemma [simp]: " $x \leq y \implies le\ y\ xs \longrightarrow le\ x\ xs$ "
```

Now show the following correctness theorem:

```
theorem "sorted (sort xs)"
```

This theorem alone is too weak. It does not guarantee that the sorted list contains the same elements as the input. In the worst case, *sort* might always return *[]* — surely an undesirable implementation of sorting.

Define a function *count xs x* that counts how often *x* occurs in *xs*. Show that

```
theorem "count (sort xs) x = count xs x"
```

Sorting with trees

Our second sorting algorithm uses trees. Thus you should first define a data type *bintree* of binary trees that are either empty or consist of a node carrying a natural number and two subtrees.

Define a function *tsorted* that checks if a binary tree is sorted. It is convenient to employ two auxiliary functions *tge/tle* that test whether a number is greater-or-equal/less-or-equal to all elements of a tree.

Finally define a function *tree_of* that turns a list into a sorted tree. It is helpful to base *tree_of* on a function *ins n b* that inserts a number *n* into a sorted tree *b*.

Show

```
theorem [simp]: "tsorted (tree_of xs)"
```

Again we have to show that no elements are lost (or added). As for lists, define a function *tcount x b* that counts the number of occurrences of the number *x* in the tree *b*. Show

theorem "tcount (tree_of xs) x = count xs x"

Now we are ready to sort lists. We know how to produce an ordered tree from a list. Thus we merely need a function *list_of* that turns an (ordered) tree into an (ordered) list. Define this function and prove

theorem "sorted (list_of (tree_of xs))"

theorem "count (list_of (tree_of xs)) n = count xs n"

Hints:

- Try to formulate all your lemmas as equations rather than implications because that often simplifies their proof. Make sure that the right-hand side is (in some sense) simpler than the left-hand side.
- Eventually you need to relate *sorted* and *tsorted*. This is facilitated by a function *ge* on lists (analogously to *tge* on trees) and the following lemma (that you will need to prove):

$$\text{sorted } (a @ x \# b) = (\text{sorted } a \wedge \text{sorted } b \wedge \text{ge } x \ a \wedge \text{le } x \ b)$$

3.3 Compilation

This exercise deals with the compiler example in section 3.3 of [1]. The simple side effect free expressions are extended with side effects.

1. Read sections 3.3 and 8.2 of [1]. Study the section about *fun_upd* in theory *Fun* of HOL: *fun_upd f x y*, written $f(x:=y)$, is *f* updated at *x* with new value *y*.
2. Extend data type $(\text{'a}, \text{'v}) \text{ expr}$ with a new alternative *Assign x e* that shall represent an assignment $x = e$ of the value of the expression *e* to the variable *x*. The value of an assignment shall be the value of *e*.
3. Modify the evaluation function *value* such that it can deal with assignments. Note that since the evaluation of an expression may now change the environment, it no longer suffices to return only the value from the evaluation of an expression.

Define a function *se_free* :: *expr* ⇒ *bool* that identifies side effect free expressions. Show that *se_free e* implies that evaluation of *e* does not change the environment.

4. Extend data type $(\text{'a}, \text{'v}) \text{ instr}$ with a new instruction *Store x* that stores the topmost element on the stack in address/variable *x*, without removing it from the stack. Update the machine semantics *exec* accordingly. You will face the same problem as in the extension of *value*.
5. Modify the compiler *comp* and its correctness proof to accommodate the above changes.

3.4 Merge sort

Implement *merge sort*: a list is sorted by splitting it into two lists, sorting them separately, and merging the results.

With the help of *recdef* define two functions

```
consts merge :: "nat list × nat list ⇒ nat list"
        msort :: "nat list ⇒ nat list"
```

and show

```
theorem "sorted (msort xs)"
```

```
theorem "count (msort xs) x = count xs x"
```

where *sorted* and *count* are defined as in section 3.2.

Hints:

- For *recdef* see section 3.5 of [1].
- To split a list into two halves of almost equal length you can use the functions *n div 2*, *take* und *drop*, where *take n xs* returns the first *n* elements of *xs* and *drop n xs* the remainder.
- Here are some potentially useful lemmas:
linorder_not_le: $(\neg x \leq y) = (y < x)$
order_less_le: $(x < y) = (x \leq y \wedge x \neq y)$
min_def: $\text{min } a \ b \equiv \text{if } a \leq b \text{ then } a \text{ else } b$

3.5 The towers of Hanoi

In section 1.3 we introduced the towers of Hanoi and defined a function *moves* to generate the moves to solve the puzzle. Now it is time to show that *moves* is correct. This means that

- when executing the list of moves, the result is indeed the intended one, i.e. all disks are moved from one peg to another, and
- all of the moves are legal, i.e. never place a larger disk on top of a smaller one.

Hint: this is a nontrivial undertaking. The complexity of your proofs will depend crucially on your choice of model and you may have to revise your model as you proceed with the proof.

References

- [1] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, 2002.
<http://www4.in.tum.de/~nipkow/LNCS2283/>.