Vrije Universiteit Amsterdam

Bachelor Thesis

Formalization of sorting algorithms in Isabelle/HOL

Author: Marco Pierre Fernandez Burgos (2592652)

1st supervisor: Dr. Jasmin Christian Blanchette
2nd reader: Dr. Femke van Raamsdonk

July 25, 2019

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Software testing can never guarantee that sorting algorithms respond correctly to all kinds of inputs. Instead, formal proofs can be used to assert that an algorithm is consistently correct. Manual formalization might become prone to human error due to a large number of proof steps. Hence, the need for a proof assistant is crucial, which is a software-based tool designed to help with the development of formal proofs in an iterative and automated fashion. By employing Isabelle/HOL, which is one of the major proof assistants, I formalized some sorting algorithms by checking the multiplicity, and that the output is sorted. These algorithms are: tail and non-tail recursive insertion and selection sort, and also tail recursive merge sort. Moreover, I used Isabelle's Isar language to present readable formal proofs as opposed to other formalizations of sorting algorithms in Isabelle/HOL.

Here one can also find the code for the formalization of sorting algorithms in Isabelle/HOL:

https://github.com/marco10507/formalization-of-sorting-algorithms

To Maria Esther Burgos Contreras

For her faith and courage.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Jasmin Christian Blanchette for the continuous support on my thesis, for his patience, and immense knowledge. I could not have imagined having a better supervisor and mentor for my thesis. Besides my supervisor, I would like to thank Dr. Femke van Raamsdonk, for providing the last review of this thesis.

Contents

List of Figures

List of Tables

Glossary

Acronyms

HOL	Higher Order Logic
IH	Induction Hypothesis
$Isar$	Intelligible semiautomated reasoning
$mset$	multiset

Isabelle/HOL symbols

...	Refers to right-hand side of last expression
[...]	Defines a list of elements
#	adding an element at the beginning of list cons, being the same as the operator : in Haskell language
$\bigwedge$	Universal quantifier for binding local variables [1]
$\Longrightarrow$	For separating premises and conclusion of theorems [1]
$\Rightarrow$	Separates types in functions
'a	Type variable where a can be any other letter
{...}	Defines a set of elements
{#...#}	Defines a mset of elements

Logical operators

$\equiv$	Is equivalent to
$\forall$	For all

GLOSSARY

$\in$ Is member of

$\rightarrow$ Implies

$\therefore$ Therefore

Number sets

$\mathbb{N}$ Natural Numbers as $\{0, 1, 2, 3, ...\}$

1

Introduction

1.1 Sorting

An algorithm receives a value or a set of values, also called input, and then follows a set of rules to produce some value, also called output. In particular, sorting algorithms aim to solve the sorting problem by rearranging the records of a collection in a defined order. The sorting problem can be formally defined as follows [2]:

Input: A sequence of n numbers $<r_1, r_2, ..., r_n>$
Output: A permutation (rearrangement) $<r'_1, r'_2, ..., r'_n>$ of input sequence such that $r'_1 \leq r'_2 \leq ... \leq r'_n$

The above formal description shows two main characteristics for the output. The first characteristic is that the output is a permutation of the input. The other characteristic is that the output is in increasing order: this means that any element that belongs to the output is always smaller than or equal to its successor. Next, two examples showing a correct and a wrong output after sorting is found:

Correct output: Input: [4, 3, 2, 1, 0] and Output: [0, 1, 2, 3, 4]
Wrong output: Input: [4, 4, 3, 2, 1, 0, 0] and Output: [0, 1, 2, 3, 4]

On the one hand, the correct output is clearly sorted because (1) the output sequence $0 \leq 1 \leq 2 \leq 3 \leq 4$ holds, and (2) it is a rearrangement of the input. On the other hand, the wrong output is in increasing order. However; it is not a permutation of the input since the output does not include the duplicates of the numbers 4 and 0

1.2 Orders

The binary relation $\leq$ used for sorting elements is required to be a total order. This binary relation meets the requirements for a *partial order* and an extra condition known as the *comparability* condition. A formal definition for the *partial order* relation follows [3]:

A partial order on a set S is a binary relation R such that:

- $\forall a \in S : (a, a) \in R$ (R is reflexive)
- $\forall a, b, c \in S : (a, b) \in R \wedge (b, c) \in R \rightarrow (b, c) \in R$ (R is transitive)
- $\forall a, b, c \in S : (a, b) \in R \wedge (b, c) \in R \rightarrow a = b$ (R is antisymmetric)

Let R be a partial order relation on Set S. Any two elements $y, x \in S$ are comparable if either xRy or yRx. For example, either $x \leq y$ or $y \leq x$. When the previous condition holds, then the relation R is comparable.

1.3 Pure functional programming

Pure functional programming is a programming paradigm where the functions do not have side effects, and the input is not modified. In other words, the functions do not utilize variables but constants. Moreover, the functions cannot access any other data than the one contained in its own arguments or inputs [4].

```
primrec remove1 :: "'a ⇒ 'a list ⇒ 'a list" where
"remove1 x [] = []" |
"remove1 x (y # xs) = (if x = y then xs else y # remove1 x xs)"
```

Figure 1.1: remove1: a predefined Isabelle/HOL function

Isabelle/HOL reassembles a purely functional programming language. Therefore, it includes base types, type constructors, function types, and type variables. It also has terms that can be formed by applying functions to arguments [5]. Figure 1.1 shows the built-in function $remove1$, which receives as input an element x and a list of type a, then removes x at most one time from the list. Finally, it outputs the result in a list of type a.

Purely functional programming code is similar to a recursively defined mathematical function, this can be seen in figure 1.1 and 1.2. To illustrate, the first constructor in function $remove1$, figure 1.1, is similar to equation 1.1 in figure 1.2, and this is also the case for the other constructor and equation.

$$remove1(x, []) = [] \tag{1.1}$$

$$remove1(x, (y\#ys)) = \begin{cases} ys, & \text{if } x = y \\ y\#remove1(x, ys), & \text{otherwise} \end{cases} \tag{1.2}$$

Figure 1.2: remove1: a recursively defined mathematical function

1.4 Structural induction

Structural induction is a mathematical proof technique which is similar to mathematical induction, but the latter can only work on the domain of $\mathbb{N}$, whereas the former can only work on recursively defined data types [6]. For example, lists and trees. Besides, we can state that structural induction accepts $\mathbb{N}$ since these numbers can be defined as a recursive data type. Next, a concrete example of structural induction will be shown by proving a particular property of the recursively defined function $remove1$ in figure 1.2:

Lemma 1.4.1 *Removing an element $y \in set(x\#xs)$ from list $x\#xs$ always yields to a strictly smaller list than the original list. Formally:*

$$\forall y, x, xs(y \in set(x\#xs) \rightarrow length(remove1(y, (x\#xs))) < length(x\#xs))$$

Proof The proof is by structural induction on list xs.

Base case: when $\mathbf{xs} = []$, then show $\mathbf{length}(\mathbf{remove1}(\mathbf{y}, [\mathbf{x}]) < \mathbf{length}([\mathbf{x}])$ holds by assuming that $\mathbf{y} \in \mathbf{set}([\mathbf{x}])$

$$\begin{aligned} & length(remove1(y, (x\#[]))) \\ = {} & length(remove1(x, (x\#[]))) && \text{[By assumption } y \in set([x])\text{]} \\ = {} & length([]) && \text{[By definition of } remove1 \text{ 1.2]} \\ < {} & length([x]) \end{aligned}$$

Induction hypothesis:

$$\forall y, x, xs(y \in set(x\#xs) \rightarrow length(remove1(y, (x\#xs))) < length(x\#xs))$$

Inductive step: assuming that IH holds, then show:

$$\forall y, a, x, xs(y \in set(a\#x\#xs) \rightarrow length(remove1(y, (a\#x\#xs))) < length(a\#x\#xs))$$

Fix: y, a, x, xs
Assume: $y \in set(a\#x\#xs)$

Case 1 $\mathbf{y} = \mathbf{a}$.

$$\begin{aligned}
& length(remove1(y, (a\#x\#xs))) \\
= {} & length(remove1(a, (a\#x\#xs))) && \text{[By } y{=}a\text{]} \\
= {} & length(x\#xs) && \text{[By definition of } remove1 \text{ 1.2]} \\
< {} & length(a\#x\#xs)
\end{aligned}$$

Case 2 $\mathbf{y} \in \mathbf{set}(\mathbf{x\#xs})$.

$$\begin{aligned}
& length(remove1(y, (a\#x\#xs))) \\
= {} & length(a\#remove1(y, (x\#xs))) && \text{[By definition of } remove1 \text{ 1.2]} \\
= {} & length([a]) + length(remove1(y, (x\#xs))) \\
< {} & length([a]) + length(x\#xs) && \text{[By using IH and since } y \in set(x\#xs)\text{]} \\
= {} & length(a\#x\#xs)
\end{aligned}$$

$\therefore$ By the principle of structural induction, the lemma 1.4.1 holds. Q. E. D.

1.5 Computational induction

```
fun splice :: "'a list ⇒ 'a list ⇒ 'a list" where
"splice [] ys = ys" |
"splice xs [] = xs" |
"splice (x#xs) (y#ys) = x#y#splice xs ys"
```

Figure 1.3: splice: a predefined Isabelle/HOL function

This kind of induction follows the recursive pattern in a function to form an induction principle which can be used to simplify inductive proofs, *Concrete semantics, page 18* [7]. For example, the induction principle of any property $P\ zs$ for the splice function in figure 1.3 follows:

$$\frac{\bigwedge ys.\ P\ [\,]\ ys \quad \bigwedge x, xs.\ P\ (x\#xs)\ [\,] \quad \bigwedge x, y, xs, ys.\ P\ xs\ ys \Longrightarrow P\ (x\#xs)\ (y\#ys)}{P\ zs}$$

1.6 Isabelle/HOL

Isabelle/HOL is a specialization of the generic proof assistant Isabelle for higher-order logic (HOL). Proofs are carried out by using the Isar language, which allows presenting proofs in a human-readable fashion [8]. Isabelle/HOL also includes productivity tools such as Sledgehammer, which invokes multiple automatic theorem provers to try to solve a particular problem. Figure 1.4 shows a fragment of a proof in Isabelle/HOL. The entire proof is located in Appendix IV.

Isabelle/HOL can form a chain of intermediate results, also known as proof by calculational reasoning, that are composed by basic principles, such as transitivity of $\leq$, $<$ or $=$. Calculations are formed by using the commands **also** and **finally**, and the "..." notation, which holds the value of the most recent right-hand-side expression [9]. This proof pattern is found in figure 1.4, line 5-10.

In figure 1.4, line 2, structural induction is carried out on list xs for any y and x. At this stage, Isabelle/HOL shows two subgoals to prove:

1. ⋀y x. y ∈ set [x] ⟹ length (remove1 y [x]) < length [x]

2. ⋀a xs y x.
   (⋀y x. y ∈ set (x#xs) ⟹ length (remove1 y (x#xs)) < length (x#xs))
   ⟹ y ∈ set (x#a#xs) ⟹ length (remove1 y (x#a#xs)) < length (x#a#xs)

The first subgoal refers to the base case and the second to the inductive step, which specifies two premises and a conclusion. Figure 1.4 does not show the resolution for the base case; however, line 4-8 shows a resolution for case $True$, which represents the case $y \in set(a\#xs)$. This resolution is almost identical to the chain of (in)equations presented in Case 2, inductive step, previous section, the only difference is that case $True$ uses a instead of x.

```
1  lemma "y ∈ set (x#xs) ⟹ length (remove1 y (x#xs)) < length (x#xs)"
2  proof(induct xs arbitrary: y x)
3   ...
4   proof(cases "y ∈ set (a#xs)")
5      case True
6      have "length (remove1 y (x#a#xs)) = length (x#remove1 y (a#xs))" ...
7      also have "... = length [x] + length (remove1 y (a#xs))" ...
8      also have "... < length [x] + length (a#xs)" ...
9      also have "... = length (x#a#xs)" ...
10     finally show "length (remove1 y (x#a#xs)) < length (x#a#xs)" by this
11   next
12     case False
13     ...
14 qed
```

Figure 1.4: Proving lemma 1.4.1 using Isabelle/HOL

2

Informal proofs for sorting

2.1 Predefined functions and data types, and derived lemmas

Table 2.1 lists all the Isabelle/HOL functions used to implement and prove the sorting algorithms. Keep in mind that the notation for multisets in Isabelle/HOL is $\{\#...\#\}$. For example, a multiset of natural number is defined as $\{\#\mathit{1}, \mathit{2}, \mathit{3}\#\}$.

Table 2.1: Predefined Isabelle/HOL functions.

Function	Definition
$Min :: {}'a\ set \Rightarrow {}'a$	returns the smallest element from a set.
$Max :: {}'a\ set \Rightarrow {}'a$	returns the largest element from a set.
$remove1 :: {}'a \Rightarrow {}'a\ list \Rightarrow {}'a\ list$	removes at most one element from a list.
$sorted :: {}'a\ list \Rightarrow bool$	checks whether a list is in total order.
$length :: {}'a\ list \Rightarrow nat$	returns the number of elements in a list.
$mset :: {}'a\ list \Rightarrow {}'a\ multiset$	transforms a list into a multiset.
$+ :: {}'a\ multiset \Rightarrow {}'a\ multiset \Rightarrow {}'a\ multiset$	multiset union.
$set :: {}'a\ list \Rightarrow {}'a\ set$	transforms a list into a set.
$take :: nat \Rightarrow {}'a\ list \Rightarrow {}'a\ list$	takes the n first elements of a list.
$drop :: nat \Rightarrow {}'a\ list \Rightarrow {}'a\ list$	drops the n first elements of a list.

Predefined functions such as take and drop come with predefined lemmas. The automatic theorem provers, such as Sledgehammer, can prove variations of these predefined lemmas. The following lemmas are modifications of predefined lemmas that Isabelle/HOL can automatically prove.

Lemma 2.1.1 (take drop permutation)

$$\forall n, xs(mset(take(n, xs)) + mset(drop(n, xs)) = mset(xs))$$

Lemma 2.1.2 (rest permutation)

$$\forall y, ys(mset(remove1(Min(set(y\#ys)), (y\#ys))) = mset(y\#ys) - \{\#Min(set(y\#ys))\#\})$$

2.2 Merge and insertion sort

Proving an implementation of insertion sort will lead more or less directly to show merge sort. However, only merge sort is presented in this section to prevent from making chapter 2 too long. Nevertheless, for the sake of completeness, the informal proofs for insertion sort are available in appendix I. If one encounters challenging to follow the informal proofs for merge sort, then one should first fully understand the informal proofs for insertion sort, which are based entirely on structural induction. I used as inspiration for the insertion sort correctness the slides of the Computer Science Theory course, University of Minnesota Duluth, page 9 [10].

$$\begin{aligned} merge(xs, []) &= xs \\ merge([], ys) &= ys \\ merge(x\#xs, y\#ys) &= \begin{cases} x\#merge(xs, y\#ys), & \text{if } x \le y \\ y\#merge(x\#xs, ys), & \text{otherwise} \end{cases} \end{aligned}$$

Figure 2.1: merge: a recursively defined mathematical function

$$\begin{aligned} merge\_sort([]) &= [] \\ merge\_sort([x]) &= [x] \\ merge\_sort(xs) &= merge(merge\_sort(left), merge\_sort(right)) \\ &\quad \textbf{where } half = (length(xs)\ div\ 2) \\ &\quad \textbf{, } left = take(half, xs) \\ &\quad \textbf{and } right = drop(half, xs) \end{aligned}$$

Figure 2.2: merge_sort: a recursively defined mathematical function

Lemma 2.2.1 (merge order) *The merge function yields a sorted list (output) if the input list xs and ys are sorted. Formally:*

$$\forall xs, ys(sorted(xs) \land sorted(ys) \rightarrow sorted(merge(xs, ys)))$$

Proof The proof is by computational induction on list *xs* and *ys*.

Base case 1 and 2: In the base case 1, the list *ys* is empty, and in the base case 2, the list *xs* is empty. Then it follows that base case 1 and 2 output, lists *xs* and *ys*, respectively. Therefore, both cases hold, because we assume that *ys* and *xs* are sorted.

Induction hypotheses:

IH.1: $\forall y, xs, ys(sorted(xs) \wedge sorted(y\#ys) \rightarrow sorted(merge(xs, y\#ys)))$

IH.2: $\forall x, xs, ys(sorted(x\#xs) \wedge sorted(ys) \rightarrow sorted(merge(x\#xs, ys)))$

Inductive step: assuming that both, IH.1 and IH.2 hold, then show:

$$\forall x, y, xs, ys(sorted(x\#xs) \wedge sorted(y\#ys) \rightarrow sorted(merge(x\#xs, y\#ys)))$$

Fix: *x, y, xs, ys*
Assume: $sorted(x\#xs) \wedge sorted(y\#ys)$

Case 1 $\mathbf{x \leq y}$. The premise *sorted(x#xs)* implies *sorted(xs)*, because removing the first element of a sorted list leaves the rest of the list sorted. $sorted(xs) \wedge sorted(y\#ys)$ holds, then from IH.1, it follows that *sorted(merge(xs, y#ys))* holds too. Moreover, by definition of merge function *merge (x#xs,y#ys) = x#merge (xs, (y#ys))*. Hence, the expression **sorted(x#merge(xs, (y#ys)))** holds, because $merge(xs, y\#ys)$, $y\#ys$ and $x\#xs$ are sorted, and $x \leq y$.

Case 2 $\mathbf{x > y}$. Similar to case 1, $sorted(x\#xs) \wedge sorted(ys)$ holds, then from IH.2, it follows that *sorted(merge(x#xs, ys))* holds too. Moreover, by definition of merge function *merge (x#xs,y#ys) = y#merge (x#xs, ys)*. Hence, **sorted(y#merge(x#xs, ys))** holds, because $merge(x\#xs, ys)$, $y\#ys$ and $x\#xs$ are sorted, and $x > y$.

∴ By the principle of computational induction, the lemma 2.2.1 holds. Q. E. D.

Lemma 2.2.2 (merge permutation) *The merge function output is a permutation of its input. Formally:*

$$\forall xs, ys(mset(merge(xs, ys)) = mset(xs) + mset(ys))$$

Proof The proof is by computational induction on list *xs* and *ys*.

Base case 1: when $xs = []$, then show $mset(merge([], ys)) = mset([]) + mset(ys)$ holds.

$$
\begin{aligned}
& mset(merge([], ys)) \\
&= mset(ys) && \text{[By definition of function } merge\text{]} \\
&= mset([]) + mset(ys)
\end{aligned}
$$

Base case 2: when $ys = []$, then show $mset(merge(xs, [])) = mset(xs) + mset([])$ holds.

$$\begin{aligned} & mset(merge(xs, [])) \\ &= mset(xs) && \text{[By definition of function } merge\text{]} \\ &= mset(xs) + mset([]) \end{aligned}$$

Induction hypotheses:

IH.1: $\forall y, xs, ys(mset(merge(xs, y\#ys)) = mset(xs) + mset(y\#ys))$

IH.2: $\forall x, xs, ys(mset(merge(x\#xs, ys)) = mset(x\#xs) + mset(ys))$

Inductive step: assuming that, both, IH.1 and IH.2 holds, then show:

$$\forall x, y, xs, ys(mset(merge(x\#xs, y\#ys)) = mset(x\#xs) + mset(y\#ys))$$

Fix: *x, y, xs, ys*

Case 1 $\mathbf{x} \leq \mathbf{y}$.

$$\begin{aligned} & mset(merge(x\#xs, y\#ys)) \\ &= mset(x\#merge(xs, y\#ys)) && \text{[By using case 1 and merge function definition]} \\ &= \{\#x\#\} + mset(merge(xs, y\#ys)) && \text{[By mset definition]} \\ &= \{\#x\#\} + mset(xs) + mset(y\#ys) && \text{[By IH.1 and since } x \leq y\text{]} \\ &= mset(x\#xs) + mset(y\#ys) && \text{[By mset definition]} \end{aligned}$$

Case 2 $\mathbf{x} > \mathbf{y}$.

$$\begin{aligned} & mset(merge(x\#xs, y\#ys)) \\ &= mset(y\#merge(x\#xs, ys)) && \text{[By using case 2 and merge function definition]} \\ &= \{\#y\#\} + mset(merge(x\#xs, ys)) && \text{[By mset definition]} \\ &= \{\#y\#\} + mset(x\#xs) + mset(ys) && \text{[By IH.2 and since } x > y\text{]} \\ &= mset(x\#xs) + mset(y\#ys) && \text{[By mset definition]} \end{aligned}$$

$\therefore$ By the principle of computational induction, the lemma 2.2.1 holds. Q. E. D.

Theorem 2.2.1 (merge_sort order) *The merge_ sort function yields a sorted list (output). Formally:*

$$\forall xs(sorted(merge\_sort(xs))$$

PROOF The proof is by computational induction on list *xs*.

Base case 1 and 2: The base case 1 is the empty list, and the base case 2 is $[x]$. Therefore, both cases hold since the empty list and $[x]$ are always sorted.

Induction hypotheses: Let l be $\forall x, xs$(take((length(x#xs) div 2), (x#xs))) and r be $\forall x, xs$(drop((length(x#xs) div 2), (x#xs))).

$$\text{IH.1: } \forall l(sorted(merge\_sort(l)))$$

$$\text{IH.2: } \forall r(sorted(merge\_sort(r)))$$

Inductive step: assuming that, both, IH.1 and IH.2 hold, then show:

$$\forall x, xs(merge\_sort(x\#xs))$$

Fix: *x, xs*
Let *half* be *length(x#xs) div 2*, *left* be *take(half, x#xs)* and right be *drop(half, x#xs)*.

By IH.1 and IH.2 *merge_sort(left)* and *merge_sort(right)* are sorted. Moreover, by definition of merge_sort *merge_sort (x#xs) = merge(merge_sort(left), merge_sort(right))*. Hence, **sorted (merge(merge_sort(left), merge_sort(right)))** holds, because lemma 2.2.1 shows that if the merge function receives as input two sorted lists, then the merge function produces a sorted list.

∴ By the principle of computational induction, the theorem 2.2.1 holds. Q. E. D.

Theorem 2.2.2 (merge_sort permutation) *The merge_sort function output is a permutation of its input. Formally:*

$$\forall xs(mset(merge\_sort(xs)) = mset(xs))$$

PROOF The proof is by computational induction on list xs.

Base case 1 and 2: The base case 1 is the empty list, and the base case 2 is $[x]$. By definition of merge sort, base case 1 and 2 output the empty list and $[x]$, respectively. Therefore, both cases hold since both input and output are the same.

Induction hypotheses: Let l be $\forall x, xs$(take((length(x#xs) div 2), (x#xs))) and r be $\forall x, xs$(drop((length(x#xs) div 2), (x#xs))).

$$\text{IH.1: } \forall l(mset(merge\_sort(l)) = mset(l))$$

$$\text{IH.2: } \forall r(mset(merge\_sort(r)) = mset(r))$$

Inductive step: assuming that, both, IH.1 and IH.2 hold, then show:

$$\forall x, xs(mset(merge\_sort(x\#xs)) = mset(x\#xs))$$

Fix: x, xs
Let *half* be *length(x#xs) div 2*, *left* be *take(half, x#xs)* and right be *drop(half, x#xs)*.

$$\begin{aligned}
& mset(merge_sort(x\#xs)) \\
&= mset(merge(merge_sort(left), merge_sort(right))) && \text{[By merge_sort definition]} \\
&= mset(merge_sort(left)) + mset(merge_sort(right)) && \text{[By using lemma 2.2.2]} \\
&= mset(left) + mset(right) && \text{[By using IH.1 and IH.2]} \\
&= mset(x\#xs) && \text{[By using lemma 2.1.1]}
\end{aligned}$$

∴ By the principle of computational induction, the theorem 2.2.2 holds. Q. E. D.

2.3 Selection sort

$$selection\_sort([]) = [] \tag{2.1}$$

$$selection\_sort(x\#xs) = minimum\#selection\_sort(rest) \tag{2.2}$$

$$\textbf{where } minimum = Min(set(x\#xs)) \tag{2.3}$$

$$\textbf{and } rest = remove1(minimum, (x\#xs))) \tag{2.4}$$

Figure 2.3: selection_sort: a recursively defined mathematical function

Lemma 2.3.1 (selection_sort halts) *For any given list, selection_ sort function always terminates.*

Proof Termination proof for the selection_sort recursive case is not trivial, because the recursive calls use as input the rest of the list $x\#xs$, figure 2.3. The halting of the recursive case can be shown by comparing the length of list $x\#xs$, left-hand side, and the length of the rest of the list, ride-hand side:

$$length(rest) < length(x\#xs)$$

∴ By the lemma 1.4.1, remove member, the lemma 2.3.1 holds, because the $rest$ is smaller than list $x\#xs$ since removing one member of any list, with at least one element, always reduce the length of the original list. Q. E. D.

Theorem 2.3.1 (selection_sort permutation) *The selection_sort function output is a permutation of its input. Formally:*

$$\forall xs(mset(selection\_sort(ys)) = mset(ys))$$

PROOF The proof is by computational induction on list ys.

Base case: We have $mset(selection\_sort[])$ since $ys = []$. Moreover, we can rewrite this expression to $mset([])$ by using the $selection\_sort$ definition. Therefore, the base case holds, because from $mset(selection\_sort[])$ we can get $mset([])$.

Induction hypothesis: Let re be $\forall y, ys(remove1(Min(set(y\#ys)), (y\#ys)))$.

$$\forall re(mset(selection\_sort(re)) = mset(re))$$

Inductive step: assuming that IH holds, then show:

$$\forall y, ys(mset(selection\_sort(y\#ys)) = mset(y\#ys))$$

Fix: *y, ys*
Let *minimum* be *Min (set (y#ys))* and let *rest* be *remove1(minimum, (y#ys))*.

Case 1 **minimum = y**.

$$\begin{aligned}
& mset(selection_sort(y\#ys)) \\
&= mset(minimum\#selection_sort(rest)) && \text{[By selection_sort definition]} \\
&= \{\#minimum\#\} + mset(selection_sort(rest)) && \text{[By definition of mset]} \\
&= \{\#minimum\#\} + mset(rest) && \text{[By using IH]} \\
&= \{\#minimum\#\} + mset(remove1(y, y\#ys)) && \text{[By } minimum = y\text{]} \\
&= \{\#y\#\} + mset(ys) && \text{[By } minimum = y \text{ and remove1 definition]} \\
&= mset(y\#ys) && \text{[By mset definition]}
\end{aligned}$$

Case 2 **minimum ≠ y**.

$$\begin{aligned}
& mset(selection_sort(y\#ys)) \\
&= mset(minimum\#selection_sort(rest)) && \text{[By selection_sort definition]} \\
&= \{\#minimum\#\} + mset(selection_sort(rest)) && \text{[By definition of mset]} \\
&= \{\#minimum\#\} + mset(rest) && \text{[By using IH]} \\
&= \{\#minimum\#\} + (mset(y\#ys) - \{\#minimum\#\}) && \text{[By using lemma 2.1.2]} \\
&= mset(y\#ys) && \text{[By mset definition]}
\end{aligned}$$

∴ By the principle of computational induction, the theorem 2.3.1 holds. Q. E. D.

Theorem 2.3.2 (selection_sort order) *The selection_sort function yields a sorted list (output). Formally:*

$$\forall xs(sorted(selection\_sort(xs))$$

PROOF The proof is by computational induction on list *xs*.

Base case: We have $mset(sorted\_sort[])$ since $xs = []$. Moreover, we can rewrite this expression to $sorted([])$ by using the *selection_sort* definition. Therefore, the base case holds, because the empty list is always sorted.

Induction hypothesis: Let *re* be $\forall x, xs(remove1(Min(set(x\#xs)), (x\#xs)))$.

$$\forall re(sorted(selection\_sort(re)))$$

Inductive step: assuming that IH holds, then show:

$$\forall x, xs(sorted(selection\_sort(x\#xs)))$$

Fix: *x, xs*
Let *minimum* be *Min (set (x#xs))* and let *rest* be *remove1(minimum, (x#xs))*.

1. Claim 1: mset(selection_sort(rest)) = mset(x#xs) - {#minimum#}

   PROOF (SUBPROOF)
   $mset(selection\_sort(rest))$
   $= mset(rest)$ [By using theorem 2.3.1]
   $= mset(x\#xs) - \{\#minimum\#\}$ [By using lemma 2.1.2]

   ∴ Claim 1 holds. Q. E. D.

2. Claim 2: $\forall n(n \in set(selection\_sort(rest)) \wedge minimum \leq n)$ holds by using Claim 1.

By IH the expression *selection_sort(rest)* is sorted, but it also has to be a permutation of the original list *x#xs* without the minimum, otherwise, *selection_sort(rest)* might include some number n such that $\exists n(n \leq minimum \wedge n \in set(selection\_sort(rest)))$ holds. Claim 1 shows that indeed the *selection_sort(rest)* is a permutation. Moreover, by *selection_sort* definition $selection\_sort(x\#xs) = minimum\#selection\_sort(rest)$. Hence, $sorted(minimum\#selection\_sort(rest))$ holds, because claim 2 shows that the minimum is always less than or equal to all the elements in $selection\_sort(rest)$, and by

IH $selection\_sort(rest)$ is sorted.

$\therefore$ By the principle of computational induction, the theorem 2.3.2 holds. Q. E. D.

3

Formalization in Isabelle/HOL

3.1 Introduction

This section shows some fragments of the formalized lemmas in Isabelle/HOL. All the lemmas and implementations of the sorting algorithms are available in appendices II, III, and IV. To learn how to formalize in Isabelle/HOL, I extensively used the book *Concrete Semantics with Isabelle/HOL*, part I [7], the manual *The Isabelle/Isar Reference Manual* [11], and the article *Structured Induction Proofs in Isabelle/Isar* [12]. One can use all these references to understand more technical details about the formalizations in the appendices.

3.2 Formalization strategy

In this section, I present the strategy I used to formalize sorting algorithms in Isabelle/HOL. This strategy is essential to avoid pitfalls, such as the belief that Isabelle/HOL proves any lemma directly by writing a few lines of code. From experience, this proof assistant verifies instantly proof steps but not entire lemmas. I suggest the next approach to formalize in Isabelle/HOL:

1. Recursively define the mathematical functions for the algorithms.

2. Write the informal proofs to show particular properties about the algorithms.

3. Implement the mathematical functions in Isabelle/HOL.

4. Formalize proofs in Isabelle/HOL using the Isar language, providing as many proof steps and details as provided in the informal proofs.

5. Use automatic provers such as sledgehammer to verify every proof step.

Indeed, as one gets more experience in Isabelle/HOL and formalization, steps one and two increasingly become redundant. However, for a beginner, it is vital to use the Isar language and present the proof as closely related to the informal proof as possible. In step five, by splitting the lemma into subproblems, the automatic provers are more likely to find a proof, because the search space is much smaller.

3.3 Insertion sort

```
lemma sorted3 : "sorted (y#insert x ys) = (y ≤ x ∧ sorted(insert x ys))"
proof(induction ys arbitrary: y rule: sorted.induct)
  ...
qed
```

Figure 3.1: Auxiliary lemma for Isabelle/HOL insert_order lemma

```
lemma insert_order: "sorted(ys)⟹sorted (insert y ys)"
proof (induct ys rule: insert.induct)
      ...
      show "sorted (y#insert x ys)"
      proof(simp  del:sorted.simps add: False sorted3 "local.2.prems")
        show "y ≤ x ∧ sorted (insert x ys)"
        proof(rule conjI)
          show "y ≤ x" ...
        next
          have "sorted ys" ...
          then show "sorted (insert x ys)" ...
      ...
qed
```

Figure 3.2: Proving lemma I.0.1 (insert order) using Isabelle/HOL

Isabelle/HOL provides many predefined lemmas that come along with the sorted function; the simplifier uses these lemmas to rewrite expressions. Some of these lemmas are:

1. *sorted.simps*(2):

   ```
   sorted (x#ys) = (∀y ∈ set ys. x ≤ y) ∧ sorted ys
   ```

2. *sorted*2_*simps*(2):

   ```
   sorted (x # y # zs) = x ≤ y ∧ sorted (y # zs)
   ```

The lemma sorted.simps(2) is too aggressive to prove that the expression *sorted (y#insert x ys)* holds, line 4, figure 3.2, because it associates each element of *insert x ys* to all its successors. To illustrate, by using lemma sorted.simps(2) the expression *sorted (y#insert x ys)* rewrites to:

```
∀z ∈ set (insert x ys). y ≤ z) ∧ sorted (insert x ys)
```

For $\forall z \in set(insert\ x\ ys).\ y \leq z)$, not even the automatic theorem provers can find an efficient proof. However, the sorted2_simps(2) lemma is less aggressive; it links the insert function directly without the quantifier, but it cannot be applied directly to *sorted (y#insert x ys)* since sorted2_simps(2) requires at least two elements.

Unfortunately, Sledgehammer cannot find any adequate proof for proof step in line 4; the automatic provers were taking more than one second to reconstruct the proof. If an automatic prover runs seemingly forever, that is a sign that the proof is too hard for it [13]. I solved this issue by adding an auxiliary lemma called sorted3 in figure 3.1. This auxiliary lemma uses sorted2_simps(2) to rewrite *sorted (y#insert x ys)* to:

```
y ≤ x ∧ sorted (insert x ys)
```

In line 6, figure 3.2, we can see that Isabelle/HOL can indeed rewrite *sorted (y#insert x ys)* to $y \leq x \wedge sorted(insert\ x\ ys)$ by using lemma sorted3. The automatic provers can find efficient proofs for $y \leq x \wedge sorted(insert\ x\ ys)$ that reconstructs under few milliseconds as opposed to more than one second. This section is quite technical, but the bottom line is: the automatic provers are more efficient when using sorted3 than sorted2_simps(2).

3.4 Merge sort

Some non-terminating tail-recursive functions are allowed in Isabelle/HOL. However, the vast majority of functions must be total; this means that these recursive functions must halt. Isabelle/HOL has an automatic termination prover, which demands that the arguments of recursive calls on the right-hand side need to be strictly smaller than the arguments on the left-hand side.

The default method for termination proofs is the lexicographic_order method. This method search for an adequate lexicographic combination of size measure. Some functions do not have a simple termination argument. In these circumstances, the termination relation has to be set manually [14].

Consider the merge function, figure 3.3, which merges two sorted lists. The lexicographic_order method fails on this function because it is not clear which argument should

```
1 function merge:: "nat list ⇒ nat list ⇒ nat list" where
2 "merge xs [] = xs" |
3 "merge [] ys = ys" |
4 "merge(x#xs)(y#ys)=(if x ≤ y then x#merge xs (y#ys) else y#merge (x#xs) ys)"
5 by pat_completeness auto
6 termination
7 proof (relation "measure (λ(xs,ys). length xs + length ys)")
8 ...
9 assume a1: "x ≤ y"
10   ...
11   show "length xs + length (y#ys) < length (x#xs) + length (y#ys)" ...
12   ...
13 assume a2: "  x ≤ y "
14   ...
15   show "length (x#xs) + length ys < length (x#xs) + length (y#ys)" ...
16 ...
17 qed
```

Figure 3.3: Proving termination for merge function in Isabelle/HOL

be considered for the standard size ordering. To prove termination manually, we must provide a custom well-founded relation.

First, we need to prove the pattern completeness of the datatype constructors. We show this by issuing the command pat_completeness, line 5. Then we need to proof termination by using the measure function ($\lambda(xs, ys).\ length\ xs + length\ ys$), which states that the standard size is the sum of both arguments, line 7. Finally, we show that in the recursive case, the sum of the lists is strictly smaller than the sum of the original inputs, lines 11 and 15.

3.5 Selection sort

Functions defined with the function keyword come with their induction schema, which follows the recursion schema and derives from the termination arrangement. For example, the selection sort function, appendix IV, proves the tailor-made induction rule:

$$\frac{P\ [\,] \quad \bigwedge x, xs.\ P\ xs \Longrightarrow P(x\#xs)}{P\ m}$$

This induction rule simplifies inductive proofs. For example, the induction rule in figure 3.3, line 2, produces two subgoals, lines 4 and 10.

We show some important basic proof patterns for structural induction and calculational reasoning, in figure 3.3. The induction proof automatically creates the subgoals and also the fix-assume steps, which are abbreviated using the case idiom, lines 3 and 6. For example, the case "case (2 x xs)" is an abbreviation for:

```
fix : x xs
assume hyps: "mset set (selection_sort ?xb) = mset ?xb"
```

We can also reduce the complexity of our proof by entering new assumptions. By using the keyword cases, line 11, we can obtain two additional assumptions, lines 12 and 22. Finally, we show a proof structure for calculational reasoning, lines 13-20, and 23-28, where we produce the proof with the "glue statements" **also** and **finally**.

```
1 theorem selection_sort_permutation: "mset (selection_sort(xs)) = mset xs"
2 proof(induct xs rule: selection_sort.induct)
3   case 1
4   then show "mset (selection_sort []) = mset []" by simp
5 next
6   case (2 x xs)
7   let ?minimum = "Min (set (x # xs))"
8   let ?rest = "remove1 ?minimum (x # xs)"
9   have IH: "mset (selection_sort ?rest) = mset ?rest" using "2.hyps" ...
10  then show "mset (selection_sort (x # xs)) = mset (x # xs)"
11  proof(cases "?minimum = x")
12    case True
13    have "mset (selection_sort (x # xs)) = mset(?minimum#selection_sort(?
      rest))" ...
14    also have "... = {#?minimum#} + mset(selection_sort(?rest))" ...
15    also have "... = {#?minimum#} + mset(?rest)" ...
16    also have "... = {#?minimum#} + mset(remove1 x (x # xs))" ...
17    also have "... = {#?minimum#} + mset(xs)" ...
18    also have "... = {#x#} + mset(xs)" ...
19    also have "... = mset (x#xs)" ...
20    finally show "mset (selection_sort (x # xs)) = mset (x # xs)" ...
21  next
22    case False
23    have c1: "mset (selection_sort (x # xs)) = mset(?minimum#selection_sort
      (?rest))" ...
24    also have c2:"... = {#?minimum#} + mset(selection_sort(?rest))" ...
25    also have c3:"... = {#?minimum#} + mset(?rest)" ...
26    also have c4:"... = {#?minimum#} + mset(x # xs) - {#?minimum#}" ...
27    also have c5:"... = mset (x # xs)" by simp
28    finally show "mset (selection_sort (x # xs)) = mset (x # xs)" ...
29  qed
30 qed
```

Figure 3.4: Formalizing lemma 2.3.1 (selection_sort permutation) in Isabelle/HOL

4

Conclusion

This thesis aimed to prove the correctness of some sorting algorithms using the proof assistant Isabelle/HOL. Based on informal proofs, I verified the total correctness of these algorithms by showing that (1) they sort according to the sorting problem, and (2) they eventually terminate. I used the Isar language to present the lemmas in sub-proofs; making the code modular, and maybe easier to maintain.

Surprisingly, I found more challenging verifying selection sort than merge and insertion sort. By assuming that the merge and insertion sort input lists are sorted, one can show that these two algorithms sort correctly. However, to prove that the selection sort output is sorted, I had to verify that the result of its recursive calls does not include new elements.

Even though the automatic theorem provers are excellent for proof search, sometimes they cannot find any proof, because the search space is massive or the current goal is not in first-order logic. Therefore, some level of fundamental mathematics and proof techniques, such as structural or computational induction, is required to break down a problem into subproblems, and because the automatic provers do not attempt to do induction since they are for first-order logic.

References

[1] TOBIAS NIPKOW. **Session 2: Isabelle's meta-logic**, 2010. [online] Available: `https://isabelle.in.tum.de/doc/prog-prove.pdf`. ix

[2] THOMAS H CORMEN, CHARLES E LEISERSON, RONALD L RIVEST, AND CLIFFORD STEIN. *Introduction to algorithms*, chapter 1, pages 5–5. MIT press, 3rd edition, 2009. 1

[3] ALFRED V. AHO AND JOHN E. HOPCROFT. *The Design and Analysis of Computer Algorithms*, chapter 3, pages 76–76. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1974. 2

[4] AMR SABRY. **What is a purely functional language?** *Journal of Functional Programming*, **8**(1):1–22, 1998. publisher: Cambridge University Press. 2

[5] TOBIAS NIPKOW. *Programming and proving in Isabelle/HOL*, chapter 2, pages 5–5. 2019. [online] Available: `https://isabelle.in.tum.de/doc/prog-prove.pdf`. 2

[6] ROD M BURSTALL. **Proving properties of programs by structural induction**. *The Computer Journal*, **12**(1):41–48, 1969. publisher: The British Computer Society. 3

[7] TOBIAS NIPKOW AND GERWIN KLEIN. **Concrete Semantics**. *A Proof Assistant Approach*, 2014. 4, 17

[8] TOBIAS NIPKOW, LAWRENCE C PAULSON, AND MARKUS WENZEL. *Isabelle/HOL: a proof assistant for higher-order logic*, **2283**, chapter 1, pages 3–3. Springer Science & Business Media, 2002. 5

[9] GERTRUD BAUER AND MARKUS WENZEL. **Calculational Reasoning Revisited (An Isabelle/Isar Experience)**. In *Theorem Proving in Higher Order Logics: 14th*

International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001. Proceedings, **2152**, pages 75–90. Springer, 2003. 5

[10] Doug Dunham. **Structural induction**, 2 2010. [online] Available: `https://www.d.umn.edu/~ddunham/cs3512s10/notes/l10.pdf`. 8

[11] Makarius Wenzel et al. **The Isabelle/Isar reference manual**, 2004. [online] Available: `https://isabelle.in.tum.de/doc/isar-ref.pdf`. 17

[12] Makarius Wenzel. **Structured induction proofs in Isabelle/Isar**. In *International Conference on Mathematical Knowledge Management*, pages 17–30. Springer, 2006. 17

[13] Jasmin Christian Blanchette and Lawrence C. Paulson. **A User's Guide to Sledgehammer for Isabelle/HOL**, 6 2019. [online] Available: `https://isabelle.in.tum.de/dist/doc/sledgehammer.pdf`. 19

[14] Alexander Krauss. **Defining recursive functions in Isabelle/HOL**, 2008. [online] Available: `https://isabelle.in.tum.de/doc/functions.pdf`. 19

Appendix I

Appendix: Insertion sort informal proofs

$$insert(x, []) = [] \tag{I.1}$$

$$insert(x, (y\#ys)) = \begin{cases} x\#y\#ys, & \text{if } x < y \\ y\#insert(x, ys), & \text{otherwise} \end{cases} \tag{I.2}$$

Figure I.1: insert: a recursively defined mathematical function

$$insertion\_sort([]) = [] \tag{I.3}$$

$$insertion\_sort(x\#xs) = insert(x, insertion\_sort(xs)) \tag{I.4}$$

Figure I.2: insertion_sort: a recursively defined mathematical function

Lemma I.0.1 (insert order) *The insert function yields to sorted list (output) if the element is inserted to a sorted list. Formally:*

$$\forall y, ys(sorted(ys) \rightarrow sorted(insert(y, ys)))$$

Proof The proof is by structural induction on list ys.

Base case: when $ys = []$, then show that $sorted(insert(y, []))$ holds by assuming that $sorted([])$ holds.

$insert(y, [])$

$= []$ [By definition of *insert* function]

Hence, $\mathbf{sorted}(\mathbf{insert}(\mathbf{y}, []))$ holds because $\mathbf{sorted}([])$ is sorted.

I. APPENDIX: INSERTION SORT INFORMAL PROOFS

Induction hypothesis:

$$\forall y, ys(sorted(ys) \rightarrow sorted(insert(y, ys)))$$

Inductive step: assuming that IH holds, then show:

$$\forall a, y, ys(sorted(a\#ys) \rightarrow sorted(insert(y, a\#ys)))$$

Fix: a, y, ys
Assume: $sorted(a\#ys)$

Case 1 $\mathbf{y} < \mathbf{a}$.

$sorted(insert(y, a\#ys))$
$= sorted(y\#a\#ys)$ [By definition of *insert* function and since $y < a$]

Hence, $\mathbf{sorted(y\#a\#ys)}$ holds, because $sorted(a\#ys)$ holds and since $y < a$.

Case 2 $\mathbf{y} \geq \mathbf{a}$. The premise $sorted(a\#ys)$ implies $sorted(ys)$, because removing the first element of a sorted list leaves the rest of the list sorted. $sorted(insert(y, ys))$ holds by using IH and $sorted(ys)$. Moreover, $insert(y, a\#ys)) = sorted(a\#insert(y, ys)$ by definition of the *insert* and since $y \geq a$. Hence, $\mathbf{sorted(a\#insert(y, ys))}$ holds, because $sorted(insert(y, ys))$ and $sorted(a\#ys)$ holds, and since $\mathbf{y} \geq \mathbf{a}$.

∴ By the principle of structural induction, the lemma I.0.1 holds. Q. E. D.

Theorem I.0.1 (insetion_sort order) *The insertion_sort function yields a sorted list (output). Formally:*

$$\forall ys(sorted(insertion\_sort(ys)))$$

PROOF The proof is by structural induction on list ys.

Base case: when $ys = []$, then show that $sorted(insertion\_sort([]))$ holds by assuming that $sorted([])$ holds.

$insertion\_sort([])$
$= []$ [By definition of function $insertion\_sort$]

Hence, $\mathbf{sorted(insert\_sort([]))}$ holds, because $sorted([])$ is sorted.

Induction hypothesis:

$$\forall ys(sorted(insert\_sort(ys)))$$

Inductive step: assuming that IH holds, then show:

$$\forall y, ys(sorted(insert\_sort(y\#ys)))$$

Fix: y, ys

$sorted(insertion\_sort(y\#ys))$
$= sorted(insert(y, insertion\_sort(ys)))$ [By definition of function $insertion\_sort$]

Hence, $\mathbf{sorted(insert(y, insert\_sort(ys)))}$ holds, because (1) $insertion\_sort(ys)$ is sorted by using IH, and (2) the lemma I.0.1 states that when the insert function adds any element to a sorted list, then its final output is sorted.

∴ By the principle of structural induction, the theorem I.0.1 holds. Q. E. D.

Lemma I.0.2 (insert permutation) *The insert function output is a permutation of its own input. Formally:*

$$\forall y, ys(mset(insert(y, ys)) = mset(y\#ys))$$

PROOF The proof is by structural induction on list ys.

Base case: when $ys = []$, then show $mset(insert(y, [])) = mset([y])$ holds.

$mset(insert(y, []))$
$= mset([y])$ [By definition of function $insert$]

Induction hypothesis:

$$\forall y, ys(mset(insert(y, ys)) = mset(y\#ys))$$

Inductive step: assuming that IH holds, then show:

$$\forall a, y, ys(mset(insert(y, a\#ys)) = mset(y\#a\#ys))$$

Fix: y, a, ys

Case 1 $\mathbf{y < a}$.

$mset(insert(y, a\#ys))$
$= mset(y\#a\#ys)$ [By using $insert$ function definition and since $y < a$]

Case 2 $\mathbf{y} \geq \mathbf{a}$.

$$
\begin{aligned}
& mset(insert(y, a\#ys)) \\
&= mset(y\#insert(a, ys)) && \text{[By using } insert \text{ function definition and since } y \geq a] \\
&= \{\#y\#\} + mset(insert(a, ys)) && \text{[By mset definition]} \\
&= \{\#y\#\} + mset(a\#ys) && \text{[By using IH]} \\
&= mset(y\#a\#ys) && \text{[By mset definition]}
\end{aligned}
$$

∴ By the principle of structural induction, the lemma I.0.2 holds. Q. E. D.

Theorem I.0.2 (insertion_sort permutation) *The insertion_sort function output is a permutation of its own input. Formally speaking:*

$$\forall ys(mset(insert\_sort(ys)) = mset(ys))$$

Proof The proof is by structural induction on list *ys*.

Base case: when $ys = []$, then show $mset(insertion\_sort([])) = mset([])$ holds.

$$
\begin{aligned}
& mset(insertion\_sort([])) \\
&= mset([]) && \text{[By definition of function } insertion\_sort]
\end{aligned}
$$

Induction hypothesis:

$$\forall ys(mset(insert\_sort(ys) = mset(ys)))$$

Inductive step: assuming that IH holds, then show:

$$\forall y, ys(mset(insert\_sort(y\#ys) = mset(y\#ys))$$

Fix: y, ys

$$
\begin{aligned}
& mset(insert\_sort(y\#ys) \\
&= mset(insert(y, (insert\_sort(ys)))) && \text{[By } insertion\_sort \text{ function definition]} \\
&= mset(y\#(insert\_sort(ys))) && \text{[By using lemma I.0.2]} \\
&= \{\#y\#\} + mset(insert\_sort(ys)) && \text{[By using mset definition]} \\
&= \{\#y\#\} + mset(ys) && \text{[By usin IH]} \\
&= mset(y\#ys) && \text{[By using mset definition]}
\end{aligned}
$$

∴ By the principle of structural induction, the theorem I.0.2 holds. Q. E. D.

Appendix II

Appendix: Insertion sort code

```
theory "insertion-sort"
  imports Main "HOL-Library.Multiset"
begin

declare[[names_short]]

text \<open>non-tail recursive\<close>

primrec insert:: "nat ⇒ nat list ⇒ nat list" where
insert_Nil: "insert x [] = [x]" |
insert_Cons: "insert x (y#ys) = (if x < y then (x#y#ys) else y#insert x ys)"

value "insert 1 [2,4,10]"

primrec insertion_sort:: "nat list ⇒ nat list" where
insertion_sort_Nil : "insertion_sort [] = []" |
insertion_sort_Cons: "insertion_sort (x#xs) = insert x (insertion_sort(xs))
    "

value "insert_sort [2,4,10,0,3]"

lemma sorted3 : "⟦sorted(y#ys);   x < y⟧⟹sorted (y#insert x ys) = (y ≤ x ∧
     sorted(insert x ys))"
proof(induction ys  rule: sorted.induct)
  case 1
  then show "sorted (y # insert x []) = (y ≤ x ∧ sorted (insert x []))" by
    auto
next
  case (2 x ys)
  then show ?case by (simp del:List.linorder_class.sorted.simps add:
    sorted2_simps)
qed
```

II. APPENDIX: INSERTION SORT CODE

```
lemma insert_order: "sorted(ys)⟹sorted (insert x ys)"
proof (induct ys arbitrary: x)
  case Nil
  then show "sorted (insert x [])" by simp
next
  case (Cons y ys)
  then show "sorted (insert x (y # ys))"
  proof (cases "x < y")
    case True
    then show "sorted (insert x (y # ys))"
    proof (simp only: True insert_Cons if_True)
      show "sorted (x # y # ys)"
      proof(simp)
        show "x ≤ y ∧ Ball (set ys) ((≤) x) ∧ Ball (set ys) ((≤) y) ∧ sorted
   ys"
        proof(intro conjI)
          show "x ≤ y"  by (simp add: Orderings.order_class.order.
  strict_implies_order True)
        next
          show "Ball (set ys) ((≤) x)" using True local.Cons.prems by auto
        next
          show "Ball (set ys) ((≤) y)"  using List.linorder_class.sorted.
  simps(2) local.Cons.prems by simp
        next
          show "sorted ys" using List.linorder_class.sorted.simps(2) local.
  Cons.prems by simp
        qed
      qed
    qed
  next
    case False
    then show "sorted (insert x (y # ys))"
    proof(simp only:False insert_Cons if_False)
      show "sorted (y # insert x ys)"
      proof(simp  del:List.linorder_class.sorted.simps add: False sorted3 "
  local.Cons.prems")
        show "y ≤ x ∧ sorted (insert x ys)"
        proof(rule conjI)
          show "y ≤ x"  by (simp add: False leI)
        next
          have "sorted ys" using "local.Cons.prems" List.linorder_class.
  sorted.simps(2) by blast
          then show "sorted (insert x ys)"  by (simp add: local.Cons.hyps)
        qed
      qed
    qed
```

```
  qed
qed

theorem insertion_sort_order : "sorted(insertion_sort(ys))"
proof (induct ys)
  case Nil
  then show "sorted (insertion_sort [])" by simp
next
  case (Cons y ys)
  show "sorted (insertion_sort (y # ys))"
  proof (simp only: insertion_sort_Cons)
    show "sorted (insert y (insertion_sort ys))" by (simp only: "local.Cons.
    hyps" insert_order)
  qed
qed

lemma insert_permutation: "mset (insert x ys) = mset (x#ys)"
proof(induct ys arbitrary: x)
  case Nil
  then show "mset (insert x []) = mset [x]" by simp
next
  case (Cons y ys)
  then show "mset (insert x (y # ys)) = mset (x # y # ys)"
  proof (cases "x < y")
    case True
    then show "mset (insert x (y # ys)) = mset (x # y # ys)" by simp
  next
    case False
    have "mset (insert x (y # ys)) = mset (y#insert x ys)" using False by
    simp
    also have "... = {#y#} + mset(insert x ys)" by simp
    also have "... = {#y#} + mset (x # ys)" using "local.Cons.hyps" False by
     simp
    also have "... = mset (x # y # ys)"  by simp
    finally show "mset (insert x (y # ys)) = mset (x # y # ys)" by this
  qed
qed

theorem insertion_sort_permutation: "mset (insertion_sort ys) = mset ys"
proof(induct ys)
  case Nil
  then show "mset (insertion_sort []) = mset []" by simp
next
  case (Cons x xs)
  have "mset (insertion_sort (x # xs)) = mset (insert x (insertion_sort(xs))
    )" by simp
  also have "... =  mset(x#(insertion_sort(xs)))" using  insert_permutation
```

```
    by simp
  also have "... = {#x#} + mset(insertion_sort(xs))" by simp
  also have "... = {#x#} + mset xs" using "local.Cons.hyps" by simp
  also have "... = mset (x # xs)" using "local.Cons.hyps" by simp
  finally show "mset (insertion_sort (x # xs)) = mset (x # xs)" by this
qed


text \<open>tail recursive\<close>

fun insertion_sort_tail:: "nat list ⇒ nat list ⇒ nat list" where
insertion_sort_tail_Nil : "insertion_sort_tail [] accum = accum" |
insertion_sort_tail_Cons: "insertion_sort_tail (x#xs) accum =
    insertion_sort_tail (xs) (insert x accum)"

value "insert_sort_tail ([2,4,10]) ([])"

theorem insert_sort_tail_order: "sorted(ACCUM)⟹sorted(insertion_sort_tail
    xs ACCUM)"
proof(induct xs arbitrary:ACCUM)
  case Nil
  then show "sorted (insertion_sort_tail [] ACCUM)" by simp
next
  case (Cons a xs)
  then show "sorted (insertion_sort_tail (a # xs) ACCUM)" by (simp add:
    insert_order)
qed

theorem insertion_sort_tail_permutation: "mset (insertion_sort_tail xs ACCUM
    ) = mset (xs@ACCUM)"
proof(induct xs arbitrary:ACCUM)
  case Nil
  then show "mset (insertion_sort_tail [] ACCUM) = mset ([] @ ACCUM)" by
    simp
next
  case (Cons a xs)
  then show ?case by (simp add: insert_permutation)
qed
```

Appendix III

Appendix: Merge sort code

```
theory "merge-sort"
  imports Main "HOL-Library.Multiset"
begin

declare[[names_short]]

text \<open>tail recursive\<close>

function merge:: "nat list ⇒ nat list ⇒ nat list" where
"merge xs [] = xs" |
"merge [] ys = ys" |
"merge (x#xs) (y#ys) = (if x ≤ y then x#merge xs (y#ys) else y#merge (x#xs)
    ys)"
by pat_completeness auto
termination
proof (relation "measure (λ(xs,ys). length xs + length ys)")
  show "wf (measure (λ(xs, ys). length xs + length ys))" by simp
next
  fix xs ys::"nat list"
  fix x y :: nat
  assume a1: "x ≤ y"
  show "((xs, y#ys), x#xs, y#ys) ∈ measure (λ(xs, ys). length xs + length ys
    )"
  proof (simp only: in_measure)
    show "(case (xs, y#ys) of (xs, ys) ⇒ length xs + length ys) < (case (x#
    xs, y#ys) of (xs, ys) ⇒ length xs + length ys)"
     proof(simp only: prod.case)
       show "length xs + length (y#ys) < length (x#xs) + length (y#ys)" by
    simp
     qed
  qed
next
```

```
  fix xs ys::"nat list"
  fix x y :: nat
  assume a2: "  x ≤ y "
  show "((x#xs, ys), x#xs, y#ys) ∈ measure (λ(xs, ys). length xs + length ys
    )"
  proof (simp only: in_measure)
    show "(case (x# xs, ys) of (xs, ys) ⇒ length xs + length ys) < (case (x#
    xs, y#ys) of (xs, ys) ⇒ length xs + length ys)"
     proof(simp only: prod.case)
       show "length (x#xs) + length ys < length (x#xs) + length (y#ys)" by
    simp
     qed
  qed
qed

value "merge ([1,2,3]) ([1,2,3,10])"

lemma sorted4 :"⟦sorted (y#ys);sorted (x#xs);sorted (merge (xs) (y#ys)); x ≤
     y⟧⟹sorted(x#merge (xs) (y#ys))"
proof(induction xs rule: sorted.induct)
  case 1
  then show ?case by auto
next
  case (2 x ys)
  then show ?case by (metis merge.simps(3) sorted2)
qed

lemma sorted5 :"⟦sorted (y#ys);sorted (x#xs);sorted (merge (x#xs) (ys)); y ≤
     x⟧⟹sorted (y#merge (x#xs) (ys))"
proof(induction ys  rule: sorted.induct)
  case 1
  then show ?case  by auto
next
  case (2 x ys)
  then show ?case by (metis merge.simps(3) sorted2)
qed

lemma merge_order: "⟦sorted (xs);sorted(ys)⟧⟹sorted(merge xs ys)"
proof(induct xs ys rule: merge.induct)
  case (1 xs)
  then show "sorted (merge xs [])" by simp
next
  case (2 ys)
  then show "sorted (merge [] ys)" by simp
next
  case (3 x xs y ys)
  then show "sorted (merge (x # xs) (y # ys))"
```

```
  proof(cases "x ≤ y")
    case True
    then show "sorted (merge (x # xs) (y # ys))"
    proof (simp only: merge.simps True if_True)
      have "sorted (merge xs (y # ys))" using "3.hyps"(1) "3.prems"(1) "3.
    prems"(2) True sorted.simps(2) by simp
      then show "sorted (x # merge xs (y # ys))"  by (simp only: "3.prems"
    (1) "3.prems"(2) True sorted4)
    qed
  next
    case False
    then show "sorted (merge (x # xs) (y # ys))"
    proof (simp only: merge.simps False if_False)
      have "sorted(merge (x # xs) ys)" using "3.hyps"(2) "3.prems"(1) "3.
    prems"(2) False sorted.simps(2) by simp
      moreover have "y ≤ x" using False nat_le_linear by simp
      ultimately show "sorted (y # merge (x # xs) ys)" by (simp only: "3.
    prems"(1) "3.prems"(2) False sorted5)
    qed
  qed
qed

lemma merge_permutation: "mset (merge xs ys) = mset xs + mset ys"
proof(induct xs ys rule: merge.induct)
  case (1 ys)
  have "mset (merge ys []) = mset (ys)" by simp
  also have "... =  mset ys + mset []" by simp
  finally show "mset (merge ys []) = mset ys + mset []" by this
next
  case (2 xs)
  have "mset (merge [] xs) = mset (xs)" by simp
  also have "... =  mset xs + mset []" by simp
  then show "mset (merge [] xs) = mset [] + mset xs" by simp
next
  case (3 x xs y ys)
  then show ?case
  proof(cases "x ≤ y")
    case True
    have "mset (merge (x # xs) (y # ys)) = mset (x#merge xs (y # ys))" using
     True by simp
    also have "... =  {#x#} +  mset (merge xs (y # ys))" by simp
    also have "... =  {#x#} +  mset xs + mset (y # ys)" using "3.hyps"(1)
    True  by (simp)
    also have "... =   mset (x # xs) + mset (y # ys)" by (simp add: "3.hyps"
    (1) True)
    finally show "mset (merge (x # xs) (y # ys)) = mset (x # xs) + mset (y #
     ys)" by this
```

```
  next
    case False
    have "mset (merge (x # xs) (y # ys)) = mset (y#merge (x#xs) ys)" using
    False by simp
    also have "... = {#y#} + mset(merge (x#xs) ys)" by simp
    also have "... = {#y#} + mset (x # xs) + mset ys" by (simp add: "3.
    hyps"(2) False)
    also have "... = mset (x # xs) + mset (y # ys)" by simp
    finally show "mset (merge (x # xs) (y # ys)) = mset (x # xs) + mset (y #
     ys)" by this
  qed
qed

value "merge [1,2,3] [1,4,5,6]"

fun merge_sort:: "nat list ⇒ nat list" where
"merge_sort [] = []" |
"merge_sort [x] = [x]" |
"merge_sort (x#xs) = ( let  half = ((length (x#xs)) div 2); left = take half
     (x#xs); right = drop half (x#xs) in  merge (merge_sort (left)) (
    merge_sort (right)))"

value "msort [9,8,7,6,5,4]"

theorem merge_sort_order: "sorted(merge_sort xs)"
proof(induct xs rule:merge_sort.induct)
  case 1
  then show ?case by simp
next
  case (2 x)
  then show ?case by simp
next
  case (3 v vb vc)
  thm  "3.hyps"
  let ?half = "length (v # vb # vc) div 2"
  let ?left = "take ?half (v # vb # vc)"
  let ?right = "drop ?half (v # vb # vc)"
  show "sorted (merge_sort (v # vb # vc))"
  proof (simp only: merge_sort.simps Let_def)
    have "sorted ((merge_sort (?left)))" using "3.hyps"(1) by simp
    moreover have "sorted ((merge_sort (?right)))" using "3.hyps"(2) by simp
    ultimately show "sorted (merge (merge_sort (?left)) (merge_sort (?right)
    ))" by (simp only:merge_order)
  qed
qed

theorem merge_sort_permutation: "mset (merge_sort xs) = mset xs"
```

```
proof(induct xs rule:merge_sort.induct)
  case 1
  then show "mset (merge_sort []) = mset []" by simp
next
  case (2 x)
  then show "mset (merge_sort [x]) = mset [x]" by simp
next
  case (3 v vb vc)
  let ?half = "length (v # vb # vc) div 2"
  let ?left = "take ?half (v # vb # vc)"
  let ?right = "drop ?half (v # vb # vc)"
  have "mset (merge_sort (v # vb # vc)) = mset(merge (merge_sort ?left) (
    merge_sort ?right))" by simp
  also have "... = mset(merge_sort ?left) + mset(merge_sort ?right)" using
    merge_permutation by simp
  also have "... = mset(?left) + mset(?right)"  by (simp add: "3.hyps"(1) "
    3.hyps"(2))
  also have "... = mset (v # vb # vc)"  by (metis append_take_drop_id
    mset_append)
  finally show "mset (merge_sort (v # vb # vc)) = mset (v # vb # vc)" by
    this
qed
```

III. APPENDIX: MERGE SORT CODE

Appendix IV

Appendix: Selection sort code

```
theory "selection-sort"
  imports Main "HOL-Library.Multiset"
begin

text \<open>no tail-recursive\<close>

lemma remove_member: "y ∈ set (x#xs)⟹length (remove1 y (x#xs)) < length (x
    #xs)"
proof(induct xs arbitrary: y x)
  case Nil
  have "length (remove1 y [x]) = length (remove1 x [x])" using Nil.prems by
    simp
  also have "length (remove1 x [x]) = length []" by simp
  also have "length [] < length [x]" by simp
  finally show "length (remove1 y [x]) < length [x]" by this
next
  case (Cons a xs)
  then show "length (remove1 y (x # a # xs)) < length (x # a # xs)"
  proof(cases "y ∈ set (a # xs)")
    case True
    have "length (remove1 y (x # a # xs)) = length (x#remove1 y (a # xs))"
    using One_nat_def Suc_pred True length_Cons length_pos_if_in_set
    length_remove1 remove1.simps(2)  by metis
    also have "... = length [x] + length (remove1 y (a # xs))"  by simp
    also have "... < length [x] + length (a # xs)" using Cons.hyps True by
    simp
    also have "... = length (x # a # xs)" by simp
    finally show "length (remove1 y (x # a # xs)) < length (x # a # xs)" by
    this
  next
    case False
    have "length (remove1 y (x # a # xs)) = length (remove1 x (x # a # xs))"
```

```
      using Cons.prems False by simp
      also have "... = length (a # xs)" by simp
      also have "... < length (x # a # xs)" by simp
      finally show "length (remove1 y (x # a # xs)) < length (x # a # xs)" by
     this
  qed
qed

function selection_sort:: "nat list ⇒ nat list" where
selection_sort_Null:  "selection_sort [] = []" |
selection_sort_Cons: "selection_sort (x#xs) = (let minimum = Min (set(x#xs))
    ; rest = remove1 minimum (x#xs) in minimum#selection_sort(rest))"
by pat_completeness auto
termination
proof (relation "measure (λ(xs). length xs)")
  show "wf (measure length)"  by simp
next
  fix minimum x ::nat
  fix rest xs:: "nat list"
  assume a1: "minimum =  Min (set (x # xs))"
  assume a2: "rest = remove1 minimum (x # xs)"
  show "(rest, x # xs) ∈ measure length"
  proof (simp only: in_measure)
    have p1: "minimum ∈ set (x#xs)" using a1 eq_Min_iff by blast
    show "length rest < length (x # xs)" using a2 p1 by (simp only:
    remove_member)
  qed
qed

value "selection_sort [2,4,10,0,0]"

theorem selection_sort_permutation: "mset (selection_sort(xs)) = mset xs"
proof(induct xs rule: selection_sort.induct)
  case 1
  then show "mset (selection_sort []) = mset []" by simp
next
  case (2 x xs)
  let ?minimum = "Min (set (x # xs))"
  let ?rest = "remove1 ?minimum (x # xs)"
  have IH: "mset (selection_sort ?rest) = mset ?rest" using "2.hyps" by simp
  then show "mset (selection_sort (x # xs)) = mset (x # xs)"
  proof(cases "?minimum = x")
    case True
    have "mset (selection_sort (x # xs)) = mset(?minimum#selection_sort(?
    rest))" using True by simp
    also have "... = {#?minimum#} + mset(selection_sort(?rest))" by simp
    also have "... = {#?minimum#} + mset(?rest)" using IH by simp
```

```
    also have "... = {#?minimum#} + mset(remove1 x (x # xs))" using True by
   simp
    also have "... = {#?minimum#} + mset(xs)" by simp
    also have "... = {#x#} + mset(xs)" using True by simp
    also have "... = mset (x#xs)" by simp
    finally show "mset (selection_sort (x # xs)) = mset (x # xs)" by this
  next
    case False
    have c1: "mset (selection_sort (x # xs)) = mset(?minimum#selection_sort
   (?rest))" by (metis "selection-sort.selection_sort_Cons")
    also have c2:"... = {#?minimum#} + mset(selection_sort(?rest))" by simp
    also have c3:"... = {#?minimum#} + mset(?rest)" using IH by simp
    also have c4:"... = {#?minimum#} + mset(x # xs) - {#?minimum#}" by (
   metis List.finite_set Min_in diff_union_single_conv list.distinct(1)
   mset_remove1 set_empty set_mset_mset)
    also have c5:"... = mset (x # xs)" by simp
    finally show "mset (selection_sort (x # xs)) = mset (x # xs)" by this
  qed
qed

theorem selection_sort_order: "sorted (selection_sort(xs))"
proof(induct xs rule:selection_sort.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs)
  let ?minimum = "Min (set (x # xs))"
  let ?rest = "remove1 ?minimum (x # xs)"
  show "sorted (selection_sort (x # xs))"
  proof(simp only:selection_sort_Cons Let_def)
    show "sorted (?minimum # selection_sort (?rest))"
    proof (simp only:Let_def sorted.simps)
      show "Ball (set (selection_sort (?rest))) ((≤) (?minimum)) ∧ sorted (
   selection_sort (?rest))"
      proof (rule conjI)
        have p1: "mset(selection_sort(?rest)) = mset(x # xs) - {#?minimum#}"
        proof -
          have c1:"mset(selection_sort(?rest)) = mset(?rest)"  using
   selection_sort_permutation by blast
          also have c2:"... = mset(x # xs) - {#?minimum#}"  using c1 by simp
          finally show "mset(selection_sort(?rest)) = mset(x # xs) - {#?
   minimum#}" by this
        qed
        show "Ball (set (selection_sort (?rest))) ((≤) (?minimum))"  by (
   metis List.finite_set Min_le in_diffD p1 set_mset_mset)
      next
        have "sorted (selection_sort (?rest))" using "2.hyps" by simp
```

```
        then show "sorted (selection_sort (?rest))" by assumption
      qed
    qed
  qed
qed

text \<open>tail-recursive\<close>

lemma max_membership: "m = Max(set (x#xs))⟹m ∈ set (x#xs)"
proof(induct xs arbitrary: x m)
  case Nil
  have "m = Max (set [x])" using Nil.prems by simp
  also have "... ∈ set [x]" by simp
  finally show "m ∈ set [x]" by this
next
  case (Cons a xs)
   have "m = Max (set (x # a # xs))" using Cons.prems by simp
   also have "... ∈ set (x # a # xs)"  using Max_in by blast
   finally show "m ∈ set (x # a # xs)" by this
qed

function tr_selection_sort:: "nat list ⇒ nat list ⇒ nat list" where
"tr_selection_sort [] accum = accum" |
"tr_selection_sort (x#xs) accum = (let max = Max (set(x#xs)); rest =remove1
    max (x#xs) in tr_selection_sort(rest) (max#accum))"
by pat_completeness auto
termination
proof(relation "measure (λ(xs, accum). size xs)")
  show "wf (measure (λ(xs, accum). length xs))"  by simp
next
  fix maximum x ::nat
  fix rest xs accum:: "nat list"
  assume a1: "maximum =  Max (set (x # xs))"
  assume a2: "rest = remove1 maximum (x # xs)"
  show "((rest, maximum # accum), x # xs, accum) ∈ measure (λ(xs, accum).
    length xs)"
  proof (simp only: in_measure)
    show "(case (rest, maximum # accum) of (xs, accum) ⇒ length xs) < (case
    (x # xs, accum) of (xs, accum) ⇒ length xs)"
     proof(simp only: prod.case)
       have p1: "maximum ∈ set (x#xs)" using a1 by (simp only: max_membership
    )
       show "length rest < length (x # xs)" using a2 p1 by (simp only:
    remove_member)
     qed
  qed
qed
```

```
value "tr_selection_sort [2,4,10,0,0] []"

theorem tr_selection_sort_output_sorted: "⟦sorted (ACCUM); ∀A e. A ∈ (set
    ACCUM) ∧ e ∈ set xs ∧ e ≤ A⟧⟹sorted (tr_selection_sort xs ACCUM)"
proof(induct xs arbitrary: ACCUM rule:tr_selection_sort.induct)
  case (1 zs)
  then show ?case  by (simp add: sorted01)
next
  case (2 v va zs)
  then show "sorted (tr_selection_sort zs ACCUM)"  by (simp add: sorted01)
qed

theorem tr_selection_sort_is_permutation_of_input: "⟦sorted (ACCUM); ∀A e. A
     ∈ (set ACCUM) ∧ e ∈ set xs ∧ e ≤ A⟧ ⟹mset (tr_selection_sort xs ACCUM)
     = mset xs + mset ACCUM"
proof(induct xs arbitrary: ACCUM)
  case Nil
  show ?case by simp
next
  case (Cons a xs)
  show "mset (tr_selection_sort (a # xs) ACCUM) = mset (a # xs) + mset ACCUM
    "  using Cons.prems(2) by blast
qed
```