Bachelor Thesis

Functional Algorithms in Lean

Author: Mascha van der Marel (2630566)

1st supervisor:	Jasmin Blanchette
daily supervisor:	Jasmin Blanchette
2nd reader:	Anne Baanen

A thesis submitted in fulfilment of the requirements for the VU Bachelor of Science degree in Computer Science

July 13, 2022

Abstract

Lean is an interactive theorem prover which can be used to formalize informal mathematical proofs. This guarantees that the proofs are correct. In this thesis, I port a number of well-known functional algorithms from Isabelle/HOL, another, older theorem prover, to Lean 3. These algorithms are described and proved informally in *Functional Algorithms, Verified!* [1] accompanied by formal proofs in Isabelle/HOL. The following algorithms are ported to Lean 3: (stable) insertion sort, quicksort, top-down merge sort, binary trees, complete binary trees, almost complete binary trees and converting a list to an almost complete tree. I prove the functional correctness and time complexity of the algorithms and the properties of the data structures. One of the challenges concerns the termination of recursive definitions in Lean, which is simpler in Isabelle/HOL. This thesis has made a contribution to potentially updating *Functional Algorithms, Verified!* with formalizations in Lean.

Acknowledgements

I would like to thank Jasmin Blanchette, my supervisor, for his guidance. He gives concrete and clear advices, is easily approachable, responds very quickly and is a sharp reader and editor. I also would like to thank Anne Baanen for being my second reader. Furthermore, I would like to thank my employer, De Nederlandsche Bank, for giving me the opportunity to study Computer Science. Last, I would like to thank my family for putting up with me every time I was stuck again in another Lean proof.

Contents

1 Introduction

Lean is an interactive theorem prover. Theorem provers are used to formalize mathematical proofs, thereby guarantying that the proof is correct. Conversely, informal mathematical proofs, also called pen-and-paper proofs, do not provide such guarantee. The risk with paper proofs is that errors can slip into the proof, going unnoticed at first, but in the end potentially bringing down an entire research paper. This risk is particularly present when mathematicians build on work from other mathematicians, mixing and matching various proofs. Unnoticed errors are impossible in Lean since the program will give an error.

This thesis contributes to the formalization of a number of well-known functional algorithms in Lean. Admittedly, these functional algorithms have already been formalized with other theorem provers. However, to my knowledge most of them have not yet been formalized in Lean, since Lean is a relatively new theorem prover, launched by Leonardo de Moura in 2013. It is becoming increasingly popular and got the attention of mathematicians, such as Thomas Hales and Kevin Buzzard. Hales uses Lean for his project Formal Abstracts, which is about formalizing the main results of informal mathematical results such as research papers. The long-term vision is that these formal abstracts can enable machine learning in math and transform the way in which mathematics is practiced [2]. Buzzard uses Lean for his Xena project in which he aims to rewrite every theorem and proof in the undergraduate math curriculum of the Imperial College London in Lean [3]. He also explained in his talk about the Future of Mathematics in 2019 [4] that Lean was the only existing proof assistant suitable for formalizing *all* of math. Lean has an active community, which has developed and maintains a large library, called mathlib, in which many mathematical building blocks are formalized for reuse in other proofs.

One area where Lean's library is less developed is the formalization of functional algorithms, which is more in the domain of computer science than mathematics. In this area, more work is done using Isabelle/HOL, an older theorem prover, and other theorem provers. In *Functional Algorithms, Verified!* [1] various functional algorithms are proved and formalized with Isabelle/HOL. The book explains data structures and algorithms for functional languages and shows the functional correctness and running time of these algorithms. Its unique feature is the formalization of all proofs in Isabelle/HOL with hyperlinks to the accompanying files in the book. I will port these proofs to Lean 3, so that ultimately *Functional Algorithms, Verified!* could be accompanied by formalized proofs in Lean. The Lean code for the proofs is available on GitHub[1].

In this thesis, I will first give background information about the interactive theorem prover Lean. Then I will discuss the sorting algorithms that I ported to Lean, followed by the binary tree data structures and algorithms. The related work section discusses other approaches. Finally, I will draw the conclusion including ideas for future work.

[1] https://github.com/MaschavanderMarel/bachelor_project

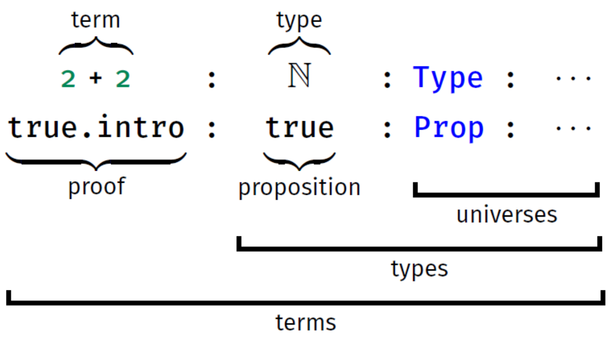
2 Background

In this section, I will explain some high-level key elements of Lean. First, I discuss type theory, then inductive types and last proof tactics.

2.1 Type Theory

There exist many proof assistants that use type theory as their logical foundation. This is also the case for Isabelle/HOL and Lean. Isabelle/HOL is based on simple type theory and Lean is based on dependent type theory with inductive types.

Type theory is a formal system in which every term has a type. *The Hitchhiker's Guide to Logical Verification* [5, p. 47] depicts this for Lean as follows:



This picture illustrates that, for example, in the expression `2+2:ℕ`, `2+2` is a term with type `ℕ`. In the expression `ℕ:Type`, `ℕ` is a term with type `Type`. Types consisting of types, such as `Type` and `Prop` in this picture, are called universes (and by convention start with a capital). The picture also shows that `true.intro` is a proof term of type `true` (a proposition term), which in turn has type `Prop`. By using terms and types to represent proofs and propositions, the question whether `H` is a proof of `P`, can be answered by checking whether the term `H` has type `P`. As a consequence, Lean does not need a proof checker but only contains a type checker.

As said before, Lean's logical foundation is *dependent* type theory. A dependent type is a type that depends on a (non-type) term [5], i.e., a value. As an example, think of a vector of `n` natural numbers. This vector has type `vec n ℕ`. Thus, the type of the vector depends on the length of the vector. Note that the fact that the vector consists of only natural numbers does not make it a dependent type even though a vector consisting of `n` letters is another type of vector. These are different vectors and thus distinct types, but when types depend on types, they are not called dependent types but type constructors. The algorithms ported in this project only use type constructors.

2.2 Inductive Types

Lean supports inductive types. An inductive type is built up from a specified list of constructors [6]. I frequently use a special type of inductive type, namely recursive inductive types, in this project. In a recursive inductive type, a constructor acts on the type being defined by taking it as input. The definition of natural numbers is a canonical example of a recursive inductive type. Lean's core library defines it as follows:

```
inductive nat: Type
| zero : nat
| succ (n : nat) : nat
```

The inductive type natural number has two constructors, namely `zero` and `succ`. The first constructor equals the number zero. The rest of the natural numbers are constructed by taking the successor of zero (which is 1), the successor of the successor of zero (which is 2), and so on.

Another example of a recursive inductive type is `list`, which occurs frequently in this project. Lean's core library defines it as follows:

```
inductive list (T : Type u)
| nil : list
| cons (hd : T) (tl : list) : list
```

The inductive type `list` has two constructors, namely `nil` and `cons`. Since the definition of `list` is polymorphic, the `nil` constructor takes one argument, namely the parameter for the type of list it should produce. For example, `list.nil ℕ` generates an empty list of natural numbers. The `cons` constructor takes two arguments, namely `hd` for the head of the list and `tl` for the tail of the list. The type of list is inferred from the first argument. So, all lists are constructed by starting with an empty list and adding head elements to it one by one.

Recursive inductive types are especially convenient for carrying out proofs by induction, which is used for the majority of proofs in this project. A simple example is given below:

```
theorem zero_add (n : ℕ) : 0 + n = n :=                                    Try it!
begin
  induction n,
  case zero : { refl },
  case succ : n ih
  { rewrite [nat.add_succ],
-- type of nat.add_succ: ∀ (n m : ℕ), n + m.succ = (n + m).succ
    rewrite [ih]}
end
```

This theorem first proves the base case that `0 + 0 = 0` with the reflexivity tactic. Then it proves that `0 + n.succ = n.succ` by rewriting to `(0 + n).succ = n.succ` with `nat.add_succ` and then rewriting to `n.succ = n.succ` with the inductive hypothesis `ih : 0 + n = n`.

Another way to prove this theorem is by induction with pattern matching:

```
theorem zero_add': ∀ n: ℕ,  0 + n = n                                      Try it!
| nat.zero := by refl
| (nat.succ n) := by rw [nat.add_succ, zero_add']
```

The patterns on the left, `nat.zero and nat.succ`, correspond to the constructors of the natural number. Lean introduces the inductive hypothesis as `zero_add' : ∀ (n : ℕ), 0 + n = n`. The main difference with the induction tactic above is that this inductive hypothesis is generalized to all `n`, whereas the above inductive hypothesis is not. The generalization will be needed for some of the proofs that require computation induction.

2.3 Proof Tactics

The inductive proof example above uses proof tactics, which can be recognized by `begin...end` or `by` commands. Tactic-style proofs are backward proofs, where the reasoning starts from the goal to be proved by breaking it down into subgoals. Conversely, structured proofs can be used for forward proofs, where the reasoning starts with the available assumptions or theorems to reach the goal. Often the styles are combined in a single proof by for example starting with backward proof reasoning but applying forward proof reasoning to reach subgoals.

Tactic versus structured proofs each have their own advantages and disadvantages. Tactics tend to be easier to work with but more difficult to read than structured proofs. An important advantage of tactics is that they enable Lean's proof automation since proof automation is a type of tactic. With automation tactics certain proofs can be found or executed by Lean automatically, which spares one the hassle of working out tedious proof steps for in essence simple or trivial proofs. Examples of automation tactics that I applied frequently are `cc`, `ring` and `linarith`. The congruence closure tactic `cc` attempts to solve the goal by chaining equalities from the context and applying congruence (i.e., if `a = b`, then `f a = f b`). `ring` is a tactic for solving equations in the language of commutative (semi)rings. `linarith` tries to find a contradiction between hypotheses that are linear (in)equalities [7].

3 Sorting Algorithms

In this section and the next section on trees, I discuss the functional algorithms that I ported to Lean. I follow the structure of the proofs in *Functional Algorithms, Verified!* [1] (hereafter abbreviated to FAV), so that FAV can be accompanied by formalized proofs in Lean keeping the need to update the book itself as small as possible. I use the lemma numbering from the book in the Lean code files for ease of reference. The names of the lemmas are identical to the names in the accompanying Isabelle/HOL files of FAV. Whenever possible, I reuse definitions and proofs from the Lean math library (mathlib) to build on the extensive work by the Lean community and focus on what is not in there yet instead of redoing existing work. The exception is when mathlib follows a different proof structure than FAV. In that case, I will redo the definitions or proofs consistent with FAV. The differences will be explained in section 5, “Related Work”.

Sorting algorithms involve sortedness. Even though the Lean math library already defines sorted as an inductive predicate, I implement a recursive definition of sorted in Lean to mimic the definition in FAV:

```
variable {α : Type*}
variable r: α → α → Prop
variable x : α
variable xs : list α

def sorted' [is_linear_order α r] : list α → Prop
| [] := true
| (h::t) := (∀ y ∈ t.to_set, r h y ) ∧ sorted' t
```

Please note that the sections below use the above variables as well without repeating them. The function is called `sorted'` to distinguish it from Lean's predefined function `sorted`. `sorted'` is defined on a linear order as in FAV. Any sorting algorithm must obviously return a sorted list corresponding to above definition, but it must also ensure that the list contains the exact same elements as the unsorted list which is ascertained with the preservation of the multisets. I show this functional correctness and the time-complexity of (stable) insertion sort, quicksort, and top-down merge sort in the next sections.

3.1 Insertion Sort

Insertion sort is a simple but computationally inefficient algorithm. It works like sorting playing cards in your hands. Cards from the unsorted part are placed at the correct position in the sorted cards. FAV defines insertion sort with the help of an auxiliary function that inserts an element at the right position in an already sorted list. These two functions are already defined in Lean's math library as `ordered_insert` (called `insort` in FAV) and `insertion_sort` (called `isort` in FAV):

```
@[simp] def ordered_insert (a : α) [decidable_rel r] : list α → list α
| []       := [a]
| (b :: l) := if a ≼ b then a :: b :: l else b :: ordered_insert l

@[simp] def insertion_sort [decidable_rel r]: list α → list α
| []       := []
| (b :: l) := ordered_insert r b (insertion_sort l)
```

3.1.1 Functional Correctness

The functional correctness of insertion sort is proved by the following lemmas:

```
lemma mset_insort [decidable_rel r] : (ordered_insert r x xs : multiset α) = {x} + ↑xs

lemma mset_isort [decidable_rel r] : (insertion_sort r xs : multiset α ) = ↑xs

lemma set_insort [decidable_rel r] : (ordered_insert r x xs).to_set  = {x} ∪ xs.to_set

lemma sorted_insort [decidable_rel r] [is_linear_order α r] :
  sorted' r (ordered_insert r x xs) = sorted' r xs

lemma sorted_isort [decidable_rel r] [is_linear_order α r]: sorted' r (insertion_sort r xs)
```

The first two lemmas prove the preservation of the multiset with a structural induction on the list `xs`. The ↑ symbol means that the list is coerced into a multiset. The second lemma uses the first lemma. The definition of `sorted'` uses `set`, so the lemma `set_insort` proves the preservation of the set using the first lemma. In addition, I needed certain functions and lemmas that are available in Isabelle/HOL's library but not in Lean's math library. These are converting a multiset to a set and proving the set preservation if a list is converted to a set directly or via a multiset. Finally, the last two lemmas prove sortedness with a structural induction on the list `xs`, where `sorted_insort` uses `set_insort` and `sorted_isort` uses `sorted_insort`.

All the lemmas contain the instance `[decidable_rel r]` which means that it is decidable whether the relation `r` is true or false. This is required for Lean's predefined functions `insertion_sort` and `ordered_insert`. A special trick was needed for the `induction'` tactic in the `sorted_insort` lemma. The `induction'` tactic generalizes the inductive hypothesis by default. However, this causes a `failed to synthesize type class instance for is_trans α r` error in the application of the `trans` lemma even when the instance `is_trans α r` is explicitly part of the goal state. Modifying the tactic into `induction' fixing *`, resulting in effectively the same behavior as the standard `induction` tactic, fixes all hypotheses and resolves the error. This seems to be an unintended consequence of the `induction'` tactic considering that generalizing the inductive hypothesis should not interfere with instances that are in the goal state. Another way to solve it is by using `@trans` providing all implicit arguments explicitly.

3.1.2 Time Complexity

To analyze the running time of functional algorithms, FAV takes the approach to count the number of all functional calls in the computation of a functional algorithm. For insertion sort, the running time functions are translated to Lean as follows:

```
def T_insort [decidable_rel r] : α → list α → nat
| x [] := 1
| x (y::ys) := if  r x y  then 0 else T_insort x ys + 1

def T_isort [decidable_rel r] : list α → nat
| [] := 1
| (x::xs) := T_isort xs + T_insort r x (insertion_sort r xs) + 1
```

The following lemmas prove the quadratic upper bound for the running time of insertion sort:

```
lemma T_insort_length [decidable_rel r]: T_insort r x xs <= xs.length + 1

lemma length_insort [decidable_rel r] : (ordered_insert r x xs).length = xs.length + 1

lemma length_isort [decidable_rel r] : (insertion_sort r xs).length = xs.length

lemma T_isort_length [decidable_rel r]: T_isort r xs <= (xs.length + 1) ^ 2
```

All lemmas are proved by structural induction on `xs`. The proof of `length_isort` needs lemma `length_insort`. The proof of `T_isort_length` uses lemma `T_insort_length` and lemma `length_isort`.

The inductive step of the last lemma uses the forward proof tactic `calc`, which is a chain of transitive results for equalities or inequalities. The inductive hypothesis `T_isort r xs ≤ (xs.length + 1) ^ 2` cannot be used as a plain substitute in the goal `T_isort r xs + T_insort r hd (insertion_sort r xs) + 1 ≤ (xs.length + 1 + 1) ^ 2`, because the IH contains a less than or equal operator. The `calc` tactic achieves the intermediate step `T_isort r xs + T_insort r hd (insertion_sort r xs) + 1 ≤ (xs.length + 1) ^ 2 + T_insort r hd (insertion_sort r xs) + 1` with the IH. This lemma also uses the tactic `ring`, which solves equations in commutative rings (a ring in which multiplication is commutative). Together with the `simp` tactic, this proves the intermediate step that `(xs.length + 1) ^ 2 + (xs.length + 1) + 1 <= (xs.length + 1 + 1) ^ 2`.

3.2 Stable Insertion Sort Key

A sorting algorithm is stable if the order of equal elements is preserved. For example, consider a list of tuples `[(2, x), (2, y), (1, z)]`, which is sorted by the first item in the tuple, called the key. This returns `[(1, z), (2, x), (2, y)]` if the sorting algorithm is stable, because `(2, x)` is still listed before `(2, y)` as in the original list. Similar to FAV, I define the stable insertion sort functions in Lean as follows:

```
variables {α : Type*} {κ : Type*}
variable r: κ → κ → Prop
variables (x: α) (k: κ ) (xs: list α)
variables (f: α → κ) (P: α → Prop)

def insort_key [decidable_rel r] [is_linear_order κ r] : list α → list α
| []       := [x]
| (y :: ys) := if r (f x) (f y) then x :: y :: ys else y :: insort_key ys

def isort_key [decidable_rel r] [is_linear_order κ r]: list α → list α
```

```
| []        := []
| (x :: xs) := insort_key r x f (isort_key xs)
```

Note that the function `f` maps `α` to `κ` and the sorting is on a key of type `κ`. The proof of the functional correctness is completely analogous to the proof of insertion sort discussed above and is therefore not repeated here.

3.2.1 Stability

The following lemmas prove the stability:

```
lemma insort_is_Cons [decidable_rel r] [is_linear_order κ r]:
  (∀ a ∈ xs.to_set, r (f x) (f a)) → insort_key r x f xs = (x:: xs)

lemma filter_insort_key_neg [decidable_rel r] [is_linear_order κ r] [decidable_pred P]:
  ¬ P x → (insort_key r x f xs).filter P = xs.filter P

lemma filter_insort_key_pos [decidable_rel r] [is_linear_order κ r] [decidable_pred P]:
  sorted' r (xs.map f) ∧ P x → (insort_key r x f xs).filter P = insort_key r x f (xs.filter
P)

lemma sort_key_stable [decidable_rel r] [is_linear_order κ r] [decidable_pred (λ y, f y =
k)]:
  (isort_key r f xs).filter (λ y, f y = k) = xs.filter (λ y, f y = k)
```

The first lemma shows with a case analysis on `xs` that if the key of an element is `r` related to the keys of the other elements, the element is inserted in front of the list. The second lemma proves by induction on `xs` that if a predicate of an element is false, inserting that element into a list and filtering that list on the predicate is equal to filtering the original list. The `decidable_pred` instance, meaning that it is decidable whether a predicate is true or false, is required for the predefined filter function on lists in Lean. The third lemma shows by induction on `xs` and using the first lemma that if a predicate of an element is true, inserting that element into a list and filtering it on the predicate is equal to first filtering the list and then inserting the element. The last lemma is proved by induction on `xs`, where the inductive step (list `xs_hd :: xs_tl`) is shown by a case analysis on `xs_hd`. If `f xs_hd ≠ k` then the claim is proved with the IH and the second lemma. If `f xs_hd = k` then the claim is proved with the IH, and the first and third lemma.

3.3 Quicksort

Quicksort is a divide-and-conquer algorithm. It selects a pivot element from the list. The other elements of the list are partitioned into two sub-lists depending on whether they are less than or greater than the pivot. The sub-lists are sorted recursively. Quicksort is predefined in Lean as `qsort`, but it has a different design from the definition in FAV. Therefore, mimicking FAV, I define quicksort in Lean as follows:

```
lemma well_founded_qs [decidable_rel r]: (list.filter (λ (y), r y x) xs).sizeof < 1 +
xs.sizeof

lemma well_founded_qs' [decidable_rel r]: (list.filter (λ (y),¬ r y x) xs).sizeof < 1 +
xs.sizeof

def quicksort [decidable_rel r]: list α → list α
| [] := []
| (x::xs) :=
  have (list.filter (λ y, r y x) xs).sizeof < 1 + xs.sizeof, by apply well_founded_qs,
  have (list.filter (λ y, ¬r y x) xs).sizeof < 1 + xs.sizeof, by apply well_founded_qs',
  quicksort (xs.filter (λ y, r y x)) ++ [x] ++ quicksort (xs.filter (λ y, ¬ r y x))
```

The definition of `quicksort` uses the same variables as the definition of `sorted'` in section 3. Unlike the recursive definition for insertion sort, Lean cannot determine the termination of the quicksort algorithm automatically. Lean does not know that the size of a filtered list is less than or equal to the unfiltered list. In other words, Lean needs to be informed that the recursive application is a decreasing well-founded relation. The two `have` commands and well-founded lemmas solve this. Conversely, in the recursive definition of insertion sort (see section 3.1), Lean can determine that `ordered_insert l` is smaller than `ordered_insert (b::l)`, because the list `l` is structurally smaller than the list `(b::l)`. For the proof of the well-founded lemmas, I use the `linarith` tactic, a tactic for solving goals with linear arithmetic, to accomplish part of the proof automatically.

3.3.1 Functional Correctness

The following lemmas show the functional correctness of quicksort:

```
lemma mset_quicksort [decidable_rel r]: ∀ xs: list α, (↑(quicksort r xs): multiset α) = ↑xs

lemma set_quicksort [decidable_rel r]: (quicksort r xs).to_set = xs.to_set :=

lemma sorted'_append [is_linear_order α r] :
  sorted' r (xs ++ ys) ↔ sorted' r xs ∧ sorted' r ys ∧ (∀ (x ∈ xs), ∀ (y ∈ ys), r x y) :=

lemma sorted_quicksort [decidable_rel r] [is_linear_order α r] : ∀ xs, sorted' r (quicksort
r xs)
```

The first lemma is proved by computation induction, which requires the pattern matching technique in Lean. This technique generates a more generalized IH `∀ (xs : list α), ↑(quicksort r xs) = ↑xs` instead of the less general IH `↑(quicksort r xs) = ↑xs` that is generated by the `induction` tactic used so far. We need the generalized version to call `mset_quicksort` recursively on a *filtered* list `xs` due to the definition of quicksort instead of on the list `xs`. The second lemma follows easily from the first. The third lemma is proved by structural induction on `xs`. The last lemma is shown by computation induction again, using the second and third lemma.

There is a peculiarity in the Lean proof of the lemmas `mset_quicksort` and `sorted_quicksort`. Like with the definition of quicksort, Lean needs to be told that the recursive application of the lemmas is a decreasing well-founded relation. However, if the same `have` commands with the well-founded lemmas are included in the proofs, Lean does not see them even though they are in the goal's context. The use of tactic mode (`begin...end`) in the proofs causes this. The solution is to first introduce the decreasing well-founded lemmas as a conjunction and then separating them with the `cases` tactic:

```
have h1: (list.filter (λ y, r y x) xs).sizeof < 1 + xs.sizeof ∧ (list.filter (λ y, ¬ r y x)
xs).sizeof < 1 + xs.sizeof, from begin
  apply and.intro,
  { apply well_founded_qs},
  apply well_founded_qs',
end,
cases h1,
```

This trick seems to be undocumented and was inspired by analyzing the definition of `merge_sort` in Lean's math library.

3.4 Top-down Merge Sort

Merge sort is another divide-and-conquer algorithm. FAV describes three variants, of which the top-down variant is the simplest. It splits the list into two halves, sorts each halve and merges the results. Two functions define top-down merge sort, namely the auxiliary function `merge` and the function `msort`. The auxiliary function `merge` is already in Lean's math library and is reused. Next to that, mathlib also contains the function `merge_sort`, with the same effect as the function `msort` from FAV, although with a different implementation. Therefore, I redefine it analogue to the structure in FAV using the length of the list and drop and take functions to follow FAV's proof structure. The definitions for top-down merge sort are:

```
def merge : list α → list α → list α
| []       l'        := l'
| l        []        := l
| (a :: l) (b :: l') := if a ≼ b then a :: merge l (b :: l') else b :: merge (a :: l) l'

def msort [decidable_rel r]: list α → list α
| xs := begin
  let n := xs.length,
  apply if h: 0 < n/2 then
  have (take (n / 2) xs).length < n, from
    begin
      simp,
      calc
        n/2 < n /2 + n/ 2  : nat.lt_add_of_pos_left h
        ... =  (n / 2) * 2 : by ring
        ... <=  n : by apply nat.div_mul_le_self,
    end,
  have (drop (n / 2) xs).length < n, from
    begin
      simp,
      have h1: 0 < n, from calc
        0 < n/2 : h
        ... <= n : nat.div_le_self' n 2,
      exact nat.sub_lt h1 h
    end,
  merge r (msort (take (n/2) xs)) (msort (drop (n/2) xs))
  else xs
end
using_well_founded {
  rel_tac := λ_ _, `[exact ⟨_, inv_image.wf length nat.lt_wf⟩],
  dec_tac := tactic.assumption }
```

Like with quicksort, the definition needs proof of the decreasing relation for the recursive application of `msort` which is given by the two `have` commands. However, that is not sufficient since Lean cannot automatically determine that the relation is well-founded. The definition also needs the `using_well_founded` syntax. This provides a proof that for any relation generated from the function `length: list α → ℕ`, the relation `λ x y, length x < length y` is well-founded [8].

3.4.1 Functional Correctness

The following lemmas show functional correctness of top-down merge sort:

```
lemma mset_merge [d: decidable_rel r] : (↑ (merge r xs ys): multiset α)  = ↑ xs + ↑ ys

lemma set_merge [decidable_rel r] :(merge r xs ys).to_set = xs.to_set ∪ ys.to_set

lemma mset_msort [decidable_rel r] : ∀ xs: list α, (↑ (msort r xs):multiset α) = ↑ xs

lemma sorted_merge [decidable_rel r] [is_linear_order α r]:
  sorted' r (merge r xs ys) ↔ sorted' r xs ∧ sorted' r ys

lemma sorted_msort [decidable_rel r] [is_linear_order α r]:
  ∀ xs: list α, sorted' r (msort r xs)
```

The first lemma is proved by structural induction on `xs` and on `ys` inside the inductive step of `xs`. The second lemma follows easily. The third lemma is show by induction on the computation of `msort` whilst providing proof of the decreasing well-founded relation for the recursive application of `msort`. The fourth lemma is proved with the same mechanism as the first lemma. The last lemma is shown by induction on the computation of `sorted_msort` with the decreasing well-founded relation proof.

3.4.2 Time Complexity

The functions for the time complexity of top-down merge sort count the number of comparisons:

```
def C_merge [decidable_rel r] : list α → list α → nat
| [] _ := 0
| _ [] := 0
| (x::xs) (y::ys) := 1 + (if r x y then C_merge xs (y::ys) else C_merge (x::xs) ys)

def C_msort [decidable_rel r]: list α → ℕ
| xs := begin
  apply if h: 0 < xs.length / 2 then
  have (take (xs.length / 2) xs).length < xs.length, from       ...
  have (drop (xs.length /2) xs).length < xs.length, from        ...
  C_msort (take (xs.length / 2) xs) + C_msort (drop (xs.length /2) xs) + C_merge r (msort r
(take (xs.length / 2) xs)) (msort r (drop (xs.length /2) xs))
  else 0
end
using_well_founded {
  rel_tac := λ_ _, `[exact ⟨_, inv_image.wf length nat.lt_wf⟩],
  dec_tac := tactic.assumption }
```

Since the proofs for the decreasing property of the recursion are identical to the proofs in the definition of `msort` in section 3.4, the code snippet above collapses them.

The following lemmas prove the time complexity of top-down merge sort:

```
lemma length_merge [decidable_rel r] : (merge r xs ys).length = xs.length + ys.length

lemma length_msort [decidable_rel r] : ∀ xs, (msort r xs).length = xs.length

lemma C_merge_ub [d: decidable_rel r] : C_merge r xs ys <= xs.length + ys.length

lemma take_drop_eq_length (n: ℕ ) : (take n xs).length + (drop n xs).length = xs.length

lemma C_msort_le [decidable_rel r] (k: ℕ ) : xs.length = 2^k → C_msort r xs <= k * 2^k
```

Unlike FAV, I use an additional lemma `take_drop_eq_length` which proves that the length of taking `n` elements from a list plus the length of dropping the same number of elements from that list equals the length of the original list. Surprisingly, this lemma is not available in Lean's math library.

4 Trees

In this section, I discuss the functional algorithms of binary trees, complete binary trees, almost complete binary trees and the conversion of lists to almost complete binary trees in Lean.

4.1 Binary Trees

A binary tree is a tree data structure in which each node has at most two children. Like lists, binary trees are defined as recursive inductive types. Lean's math library already contains binary trees. There is a small difference with FAV, namely the constructor of a `leaf` is called `nil` and the order of the input for the node constructor is `α, tree, tree` instead of `tree, α, tree`. Since this does not change the structure of the proofs, Lean's definition is reused:

```
inductive {u} tree (α : Type u) : Type u
| nil : tree
| node : α → tree → tree → tree
```

Lean's library does not contain any basic binary tree functions, so I define the basic functions that are needed later:

```
variable { α : Type}
variable t : tree α

def size: tree α → ℕ
| nil := 0
| (node a l r) := size l + size r + 1

def size1: tree α → ℕ
| nil := 1
| (node a l r) := size1 l + size1 r

def height : tree α → ℕ
| nil := 0
| (node a l r) := max (height l) (height r) + 1

def min_height: tree α → ℕ
| nil := 0
| (node a l r) := min (min_height l) (min_height r) + 1

def inorder: tree α → list α
| nil := []
| (node a l r) := inorder l ++ [a] ++ inorder r
```

The function `size` counts the number of nodes in a tree. The function `size1` counts the number of nodes plus one. The `height` of a tree is the longest path from the root to a leave. Note that according to FAV's definition of height, a tree with a single node has a height of one. It is also common in other literature to define the height of a single node tree as zero. The `min_height` of a tree is the shortest path from the root node to a leave. In addition to the above definitions, I define `set_tree: tree α → set α`, `preorder': tree α → list α`, `postorder: tree α → list α`, `subtrees: tree α → set (tree α)` in line with FAV and the Isabelle/HOL files. The code snippet above does not include these definitions, because they are not essential for the rest of this thesis.

4.1.1 Properties

Binary trees have the following obvious properties:

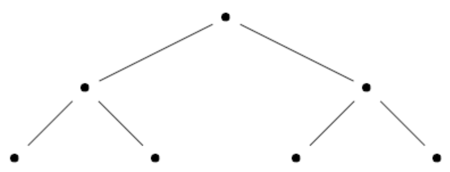
```
lemma size1_size: size1 t = size t + 1

lemma height_le_size_tree: height t <= size t

lemma min_height_le_height : min_height t <= height t

lemma min_height_size1: 2 ^ min_height t <= size1 t
lemma size1_height: size1 t <= 2 ^ height t
```

These lemmas are all proved by structural induction on the tree `t`. Because the node constructor takes two trees as input, the `induction` tactic generates two inductive hypotheses: one for the left subtree and one for the right subtree. Other than that, the induction works the same as for lists. Furthermore, I use tactics such as `calc`, `ring`, `linarith` and Lean's built-in lemmas on power operations on natural numbers.

The Isabelle/HOL files contain many more lemmas on tree `size`, `height`, `set_tree`, `subtrees,` list of entries and mapping. I have ported these lemmas to Lean as well even though FAV does not explicitly refer to them.

4.2 Complete Trees

Complete binary trees have all leaves on the same level as depicted below:



The definition is:

```
def complete : tree α → Prop
| nil := true
| (node a l r ) := height l = height r ∧ complete l ∧ complete r
```

So, a tree is complete if its left and right subtree have equal height and if both subtrees are complete.

4.2.1 Properties

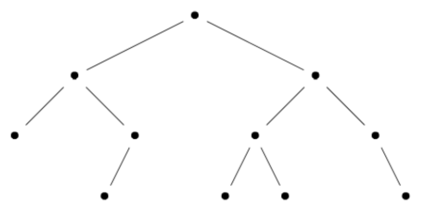
Complete trees have the following properties:

```
lemma complete_iff_height: complete t ↔ min_height t = height t

lemma size1_if_complete : complete t → size1 t = 2 ^ height t

lemma size1_height_if_incomplete: ¬ complete t → size1 t < 2 ^ height t

lemma min_height_size1_if_incomplete: ¬ complete t → 2 ^ min_height t < size1 t

lemma complete_if_size1_height: size1 t = 2 ^ height t → complete t

lemma complete_iff_size1: complete t ↔ size1 t = 2 ^ height t
```

The first four lemmas are proved by structural induction on the tree `t`. The lemmas on incomplete trees are proved by four cases. If the height of the left subtree equals the height of the right subtree, then either the left subtree or the right subtree must be incomplete. If the height of the left subtree is not equal to the height of the right subtree, then the height of the left subtree is smaller or larger than the height of the right subtree. The fifth lemma use the contrapositive of the third lemma. The last lemma follows directly from the second and the fifth lemma.

4.3 Almost Complete Trees

Almost complete trees are trees where the leaves can occur one level above the lowest level as depicted below:



The definition is:

```
def acomplete : tree α → Prop
| t := height t - min_height t <= 1
```

So, the difference between the height and the `min_height` is at most one. Complete trees are a subset of almost complete trees.

4.3.1 Properties

Almost complete binary trees have the following properties:

```
lemma acomplete_optimal : acomplete s ∧ size s <= size t → height s <= height t

lemma acomplete_height : acomplete t → height t = nat.clog 2 (size1 t)

lemma acomplete_min_height: acomplete t → min_height t = nat.log 2 (size1 t)
```

The first property shows that almost complete trees have the smallest height compared to trees with the same number of nodes. The proof is by cases on the completeness of tree `s` and the help of the properties of complete trees given in above section 4.2. The second and third lemma give the exact height and minimum height of almost complete trees respectively. `nat.clog` stands for the ceiling logarithm and `nat.log` for the floor logarithm of natural numbers. The proofs of the tree lemmas are by cases on the completeness of the tree and the help of the properties of complete trees.

Note that the formalization of the last two lemmas is missing in the Isabelle/HOL files, but they are formalized in Lean. For this, I needed two lemmas on logarithms on natural numbers, which are not present in Lean's mathlib:

```
lemma lt_clog_of_pow_lt {m n : ℕ}: 2 ^ m < n → m < nat.clog 2 n

lemma gt_log_of_lt_pow {m n: ℕ} (h: 1 <= n) : n < 2 ^ m → m > nat.log 2 n
```

Both lemmas are proved by induction on `m`.

4.4 Converting a List to an Almost Complete Tree

Contrary to FAV's claim that all proofs are machine-checked, the section on converting a list to an almost complete tree has not been formalized in Isabelle/HOL at all. This section shows how to convert a list `xs` to an almost complete tree `t` such that `inorder t = xs`. If the list is sorted, the almost complete tree is a binary search tree. The definition uses an auxiliary parameter `ℕ` to determine how much of the list is converted to a tree, while the rest of the list is returned with the tree:

```
variable {α : Type}
variables {n: ℕ}
variables xs zs: list α
variable a: α
variables t: tree α

def bal [inhabited α]: ℕ → list α → tree α × list α
| n xs :=
  if h: n ≠ 0 then let
    m := n /2,
    (l, ys) :=
    have m < n, begin
      have h1: 0 < n, from nat.pos_of_ne_zero h,
      have h2: 1 < 2, by simp,
      exact nat.div_lt_self h1 h2,
    end,
    bal m xs,
    (r, zs) :=
    have n - 1 - m < n, begin
      have h1: 0 < (1 + m), by simp [nat.add_one_ne_zero m, nat.add_comm,
nat.pos_of_ne_zero],
      have: m < n, by simp [nat.div_lt_self, nat.pos_of_ne_zero, *],
      have h2: 1 + m <= n, by linarith,
      rw nat.sub_sub n 1 m,
      exact nat.sub_lt_of_pos_le (1 + m) n h1 h2,
    end,
    bal (n - 1 - m) (ys.tail) in
    ((node ys.head l r), zs)
  else (nil, xs)
```

The proofs for `have m < n` and `have n - 1 - m < n` are needed to show that the recursive application of `bal` is a decreasing well-founded relation. The `inhabited α` instance is required for the `ys.head` function since the head function returns a default value in case of an empty list. For example, the function `(@list.nil ℕ).head` returns `0`.

Above definition uses nested `let` commands, for example the definition of `(l, ys)` in the definition of `(r, zs)`, which is problematic in proofs using this definition. Therefore, I made an alternative but similar definition `bal'` that does not use `let` commands and hence works in the proofs. Because the alternative definition is more difficult to read, it is not included here, but the interested reader can look it up in the Lean file on GitHub.

The following definitions balance a prefix of a list or tree, or the full list or tree:

```
def bal_list [inhabited α]: ℕ → list α → tree α
| n xs := (bal' n xs).fst

def balance_list [inhabited α]: list α → tree α
| xs := bal_list xs.length xs

def bal_tree [inhabited α]: ℕ → tree α → tree α
| n t := bal_list n (inorder t)

def balance_tree [inhabited α]: tree α → tree α
| t := bal_tree (size t) t
```

4.4.1 Functional Correctness

The lemmas below prove the functional correctness of the algorithm for converting a list to an almost complete tree, the first of which is:

```
lemma bal_prefix_suffix [inhabited α] :
  n <= xs.length ∧ bal' n xs = (t, zs) → xs = inorder t ++ zs ∧ size t = n
```

This lemma expresses that converting the prefix of a list to a tree, and then converting it back to a list with the `inorder` function, and appending it to the remaining suffix, returns the original list. The proof is by complete induction on `n`. The principle of complete induction is [9]: "Let *P* be any property that satisfies the following: for any natural number *n*, whenever *P* holds of every number less than *n*, it also holds of *n*. Then *P* holds of every natural number." So, one needs to show that `P` holds of any `n`, assuming it holds of every smaller number. To apply this in Lean, I use a modification of the standard Lean `induction` tactic, namely `induction n using nat.strong_induction_on with n ih generalizing t zs xs`, where `strong_induction` is another name for complete induction. The generalization of `t zs xs` is required to apply the IH to the left and right subtrees and the prefix and suffix of the list according to the definition of `bal'`.

To improve the readability of the proof of the lemma, I use the following `let` tactics. This functions in the proof but not in the definition of `bal` as explained before:

```
let l := (bal' (n/2) xs).fst,
let ys := (bal' (n/2) xs).snd,
let r := (bal' (n - 1 - n/2) ys.tail).fst,
let as := (bal' (n - 1 - n/2) ys.tail).snd,
```

The above lemma `bal_prefix_suffix` easily proves the following properties of the derived definitions:

```
lemma inorder_bal_list_eq_take [inhabited α ] :
  n <= xs.length → inorder (bal_list n xs) = list.take n xs

lemma inorder_balance_list_eq_list [inhabited α] : inorder (balance_list xs) = xs

lemma inorder_bal_tree_eq_take_inorder [inhabited α]:
  n <= size t → inorder (bal_tree n t) = list.take n (inorder t)

lemma inorder_balance_tree_eq_inorder [inhabited α]: inorder (balance_tree t) = inorder t
```

The following lemmas prove that `bal'` returns an almost complete tree:

```
lemma bal_height [inhabited α] :
  n <= xs.length ∧ bal' n xs = (t, zs) → height t = nat.clog 2 (n + 1)

lemma bal_min_height [inhabited α] :
  n <= xs.length ∧ bal' n xs = (t, zs) → min_height t = nat.log 2 (n + 1)

lemma clog_sub_log_le_one {n: ℕ } : nat.clog 2 n - nat.log 2 n <= 1

lemma bal_acomplete [inhabited α ]: n <= xs.length ∧ bal' n xs = (t, zs) → acomplete t
```

First, the height and the minimum height of the returned tree is determined. Then, the definition of an almost complete tree and the fact that the ceiling logarithm minus the floor logarithm is less than or equal to one prove the last lemma. Surprisingly, Lean's math library does not yet contain a proof for the relation between ceiling logarithm and floor logarithm.

5 Related Work

In this section, I touch on related work by comparing the recursive definition of sorted to inductive predicates, comparing the specification of functional correctness via permutation or multisets, and briefly relating Lean to other theorem provers.

5.1 Inductive Predicates

For the definition of `sorted'` (see section 3), I followed the recursive definition from FAV, repeated here for ease of reference:

```
def sorted' [is_linear_order α r] : list α → Prop
| [] := true
| (h::t) := (∀ y ∈ t.to_set, r h y ) ∧ sorted' t
```

However, it is also possible to define `sorted` as an inductive predicate. The type of an inductive predicate, also known as inductively defined proposition, is a function returning a proposition. Thus, inductive predicates are like inductive types, except that the return is of type `Prop` instead of type `Type`.

The following is an example of an inductive predicate for sorted from *The Hitchhiker's Guide to Logical Verification* [5, p. 88]:

```
inductive sorted : list ℕ → Prop
| nil : sorted []
| single {x : ℕ} : sorted [x]
| two_or_more {x y : ℕ} {zs : list ℕ} (hle : x ≤ y)
    (hsorted : sorted (y :: zs)) :
  sorted (x :: y :: zs)
```

This definition has three constructors: `nil` and `single` are sorted by definition; `two_or_more` is sorted if the first element is less than or equal to the first element of a sorted list.

In fact, Lean's math library uses an inductive predicate, although a different version with only two constructors:

```
def sorted := @pairwise

inductive pairwise : list α → Prop
| nil : pairwise []
| cons : ∀ {a : α} {l : list α}, (∀ a' ∈ l, R a a') → pairwise l → pairwise (a::l)
```

`pairwise R l` means that all the elements with earlier indexes are `R`-related to all the elements with later indexes [10].

Recursive definitions versus inductive predicates each have their own advantages and disadvantages. Recursive definitions need to terminate. Luckily, the recursive definition of `sorted'` requires no special arrangements because Lean can determine its termination automatically. An advantage is that recursive definitions are suitable for the frequently used `refl` and `simp` tactics. Note that `pairwise`, although defined as an inductive predicate, can be used in `simp` tactics as well because of the following predefined lemma in Lean's library:

```
variables {R}
@[simp] theorem pairwise_cons {a : α} {l : list α} :
  pairwise R (a::l) ↔ (∀ a' ∈ l, R a a') ∧ pairwise R l
```

Often, inductive predicates are more elegant and abstract than recursive definitions, because they can do with less constructors, for example when defining even numbers [5]. For `sorted'` that is not the case though, because the recursive definition also only needs two constructors. Last, inductive predicates cannot be unfolded like definitions to construct proofs, but they can be applied by using introduction rules, elimination and induction principles.

5.2 Specification of Functional Correctness

As mentioned in section 3.1, Lean’s math library already defines algorithms for insertion sort, and I reused those definitions. However, the specification of the functional correctness of insertion sort in mathlib follows a different logic from FAV. The specification in Lean’s library is based on permutations whereas the specification in FAV is based on multisets. So, for a sorting algorithm to be correct, it must either return a sorted list that is a permutation of the input list (mathlib), or that has the same multiset of values as the input list (FAV). It makes sense that FAV uses the multiset variant since the FAV definition of `sorted` uses sets and the preservation of multisets yields the preservation of sets. *Software Foundations Volume 3: Verified Functional Algorithms* [11] shows that the two specifications are in fact equivalent by proving that the multisets of two lists are equal if and only if one list is a permutation of the other list.

As described in section 3.4, the algorithm top-down merge sort is also already predefined in Lean’s mathlib as `merge_sort`:

```
include r
def merge_sort : list α → list α
| []        := []
| [a]       := [a]
| (a::b::l) := begin
  cases e : split (a::b::l) with l₁ l₂,
  cases length_split_lt e with h₁ h₂,
  exact merge r (merge_sort l₁) (merge_sort l₂)
end
using_well_founded
{ rel_tac := λ_ _, `[exact ⟨_, inv_image.wf length nat.lt_wf⟩],
  dec_tac := tactic.assumption }
```

To stay close to FAV, I did not reuse Lean’s `merge_sort`, but translated FAV’s definition `msort` into Lean (see section 3.4). However, `merge_sort` and `msort` are not actually structured that differently. Instead of using `take` and `drop` functions like `msort`, `merge_sort` uses the function `split` to divide the list into halves. The function `length_split_lt` takes care of the proof for the decreasing well-founded relation in the recursive application. In fact, the specification of `merge_sort` in Lean's library is similar to the specification in *Software Foundations Volume 3: Verified Functional Algorithms* [11], which also uses a split function. The functional correctness is specified with permutation just like in Lean's library.

5.3 Other Theorem Provers

There is overlap between FAV and *Verified Functional Algorithms* [11] in the sense that both verify various functional algorithms, but FAV uses Isabelle/HOL, while *Verified Functional Algorithms* uses Coq. Lean is more similar to Coq than to Isabelle/HOL. Both Lean and Coq are based on dependent type theory, while Isabelle/HOL is based on simple type theory. Furthermore, the syntax of Lean is quite similar to Coq. At first sight, these two analogies could render porting proofs from Coq to Lean relatively straightforward. However, there are also important differences, such as the preservation of good type theoretic properties [12], and there is quite a debate going on in the communities about which theorem prover is better or more suitable for certain purposes.

Comparing the proofs in Isabelle to my proofs in Lean, it is clearly visible that the Isabelle proofs are shorter. There can be several reasons for this. First, there does not seem to be a Lean equivalent of the `auto` tactic in Isabelle. Isabelle's auto tactic can prove simple inductions automatically, whereas in Lean both the base case and inductive step need to be worked out. Another reason is that with recursive definitions terminations must be proved. The terminations in this thesis are not difficult to see and Isabelle can prove them automatically, but in Lean this requires more work. Last, Lean provides for many ways to write proofs in a highly compact form, which I have not fully used.

6 Conclusion

In this thesis I have explained the relevance of theorem provers for the formalization of informal paper-and-pencil proofs. I have explained in more detail the theorem prover Lean, which is a relatively new theorem prover. Next, I described how I formalized a number of functional algorithms from FAV in Lean 3. FAV is currently accompanied by formal proofs in Isabelle/HOL. The ultimate objective is to port all the algorithms from FAV to Lean to update the book with formal verifications in Lean, a more modern theorem prover. The selection of functional algorithms ported are insertion sort, stable insertion sort, quicksort, top-down merge sort, binary trees, complete binary trees, almost complete binary trees, and the conversion of lists to almost complete trees. The proven lemmas are related to the functional correctness of these algorithms, the time complexity and the properties of the data structures. Last, I touched on related work by comparing the recursive definition of sorted to inductive predicates, comparing the specification of the functional correctness via permutation to the specification via multisets, and relating Lean to other theorem provers.

In this project, I encountered some challenges. The main one was the application of decreasing well-founded relations for the recursive application of definitions of certain algorithms. Next to that, Lean has some unexpected or not well documented peculiarities which were difficult to solve. Also, the Lean math library is extensive, which complicated finding the right theorems for trivial proofs that would require quite an amount of manual work. Especially arithmetic with natural numbers was sometimes unexpectedly cumbersome, but automation tactics such as `linarith`, `cc`, and `ring` could often solve this. Furthermore, the performance of Lean in Visual Studio Code was rather poor, resulting in long waits for the type checking to finish, which is obviously not helpful with the trial-and-error nature of finding proofs. Occasionally, the orange bar of hell would appear indicating that no progress at all is being made.

Future work could be aimed at porting the other algorithms from FAV to Lean. I have ported a selection of algorithms, but FAV contains many more. Next to that, converting this work from Lean 3 to Lean 4 could be part of future work. Lean 4 is the latest version of Lean that contains many new features and addresses shortcomings of Lean 3. Lean 4 also improves the performance of Lean. Unfortunately, Lean 4 is not backwards compatible with Lean 3. Hence, the entire mathematical library from Lean also needs to be ported to Lean 4 which is an interesting future project for the Lean community to undertake.

References

[1] T. Nipkow, J. Blanchette, M. Eberl, A. Gómez-Londoño, P. Lammich, C. Sternagel, S. Wimmer and B. Zhan, "Functional Algorithms, Verified!," 2021. [Online]. Available: https://functional-algorithms-verified.org/functional_algorithms_verified.pdf.

[2] "Formal Abstracts," [Online]. Available: https://formalabstracts.github.io/. [Accessed 13 April 2022].

[3] K. Buzzard, "Xena," [Online]. Available: https://xenaproject.wordpress.com/what-is-the-xena-project/. [Accessed 13 April 2022].

[4] K. Buzzard, "The future of mathematics," 2019. [Online]. Available: https://www.youtube.com/watch?v=Dp-mQ3HxgDE.

[5] A. Baanen, A. Bentkamp, J. Blanchette, J. Hölzl and J. Limperg, "The Hitckhiker's Guide to Logical Verification, 2021 Standard Edition," 25 October 2021. [Online]. Available: https://github.com/blanchette/logical_verification_2021/raw/main/hitchhikers_guide.pdf.

[6] J. Avigad, L. de Moura and S. Kong, "Theorem Proving in Lean, release 3.23.0," 2021. [Online]. Available: https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

[7] "Mathlib tactics," [Online]. Available: https://leanprover-community.github.io/mathlib_docs/tactics.html. [Accessed 15 5 2022].

[8] Lean Community, "The equation compiler and using_well_founded," [Online]. Available: https://leanprover-community.github.io/extras/well_founded_recursion.html. [Accessed 24 April 2022].

[9] J. Avigad, R. Y. Lewis and F. van Doorn, "Logic and Proof, Release 3.18.4," 4 December 2021. [Online]. Available: https://leanprover.github.io/logic_and_proof/logic_and_proof.pdf.

[10] Lean Community, "Mathlib documentation," [Online]. Available: https://leanprover-community.github.io/mathlib_docs/data/list/defs.html#list.pairwise. [Accessed 20 5 2022].

[11] A. W. Appel, *Verified Functional Algorithms, Software Foundations,* vol. 3, B. C. Pierce, Ed., Electronic textbook, 2021.

[12] "Lean Versus Coq: The Cultural Chasm," [Online]. Available: https://artagnon.com/articles/leancoq. [Accessed 20 5 2022].