View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Directory of Open Access Journals

LEAN FORMALIZATION OF INSERTION SORT STABILITY AND CORRECTNESS

František SILVÁŠI*, Martin TOMÁŠEK**
*,**Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, E-mail: *frantisek.silvasi@tuke.sk, **martin.tomasek@tuke.sk

ABSTRACT

We present a fully mechanized proof of correctness and stability of the insertion sort algorithm, while handling stability not as an afterthought in its formal specification, but rather as a property removing any unspecified behaviour from the algorithm, by explaining what happens to elements that are considered equivalent. We therefore express the combined notion of being sorted along with stability as a single inductive predicate, allowing us to share uncovered information in proofs, resulting in a more elegant approach to showing correctness and stability of sorting algorithms. Naturally, there are also cases when we can indeed forget about stability. We prove, that under the assumption that the sequence to be sorted contains unique elements only, sorting and stable sorting are equivalent notions. Formalization is conducted in the Lean theorem prover.

Keywords: *Formal specification; Insertion sort; Proof of correctness; Theorem proving*

1. INTRODUCTION

Given the ubiquity of application of sorting algorithms, it should come as no surprise that many attempts at formal software specification with consequent verification revolve around this very topic. In addition to the fact they are commonplace, and as such, having their formal proof of correctness is inherently very useful, they are also reasonably complex algorithms (in terms of invariants they establish) with relatively easy to formulate specification and implementation.

This tradition was started by Hoare and Foley [1] in 1971, when they conducted a proof on paper of the imperative quicksort algorithm. They also suggested the possibility of mechanization of the said proof, but disregarded the idea after short contemplation given the state of available tools then.

Continuing with the tradition, Filliâtre and Magaud [3] used Coq [2] to formalize and prove correct imperative insertion sort, quicksort and heapsort. Selection sort in Java using Krakatoa Modelling Language [5] for specification of behaviour has been formalized by Tushkanova, Giorgetti and Kouchnarenko [4].

With regards to more recent developments, it has been implicitly established that the notion of stability is also an important characterizing factor of sorting algorithms and as such, its presence (or lack thereof) should be captured in formal specifications. We are aware of two developments of certified sorting algorithms that also consider stability. In 2013, Sternagel [6] used a declarative Haskell implementation of merge sort (one supposedly used by GHC, a major compiler provider for the language in question) as basis for an implementation in Isabelle/HOL [7], which he then not only proved correct with regards to the expected sorting behaviour (to be discussed shortly), but also showed that the implementation he is using happens to be stable. Shortly thereafter, Gouw, Boer and Rot [8] formalized a Java implementation of counting sort and radix sort, utilizing KeY [9] to conduct proofs of correctness and stability. It is worth noting that stability of counting sort when used as a subroutine for radix sort is of paramount importance to achieve correct behaviour of radix sorting. In such cases, proof of correctness cannot omit this important property at all.

The main contributions of this paper are as follows. We use the Lean theorem prover [10] to specify stability and correctness of sorting algorithms utilizing a single inductive predicate to express the notion of being sorted along with stability, allowing us to avoid having to recover ordering information in the stability proof separately. We are not aware of this approach being used prior. Demonstration of the practicality thereof is conducted by showing that behaviour of (stable) insertion sort conforms to the specification. To the best of our knowledge, this is the first mechanized study to consider stability of insertion sort. In addition to that, we provide an equivalence proof of the notion of being sorted and being sorted stably, assuming uniqueness of elements. The study also represents an example of utilizing formal verification tools to solve problems related to real world applications.

2. NOTATION, STABLE INSERTION SORT AND ITS USUAL SPECIFICATION

Insertion sort generally consists of two parts. A procedure to insert an element into a correct position in a subsequence formed from the original one and a routine to call the insertion procedure on every element of the sequence. This way, a series of incrementally longer sorted subsequences is built, until the very last element is inserted. The algorithm is described in more detail in Algorithms by Sedgewick [11].

A short digression to notation and nomenclature used henceforth. Sequences of elements will be modelled by lists of form $head :: tail$, with $[\,]$ being an empty list and $[x]$ a list of length one. We will also use the words list and sequence interchangeably.

Our specification slightly differs from what is usually used for a correct (and possibly stable) sorting algorithm.

2.1. Usual correctness

Correctness is generally expressed in terms of two properties (already present in [1]):

- the resulting sequence is sorted

- the resulting sequence is a permutation of the input sequence

This coincides with our intuitive idea of what a sorting algorithm should do, if we for a moment completely disregard stability.

With regards to their formal phrasing, we could utilize inductive predicates, such as a set of common ones for the sorted property (named *empty*, *one* and *more* respectively):

$$\frac{}{sorted\ [\,]} \qquad \frac{}{sorted\ [x]} \qquad \frac{x \leq y \quad sorted\ (y :: t)}{sorted\ (x :: y :: t)}$$

Where the $\leq$ operator determines decidable linear order, a requirement on types of elements we can regard as sorted.

It is also common to use non–inductive definitions should one find them more convenient, such as the following one expressing that a sequence is a permutation of an another one:

$$permutation\ l_1\ l_2 \stackrel{\text{def}}{=} \forall x, count\ x\ l_1 = count\ x\ l_2$$

Where *count x l* represents the number of occurrences of x in l.

2.2. Usual stability

Stability is preservation of relative order of equivalent elements by a sorting procedure. In general, the property is of interest in the context of incremental sorting (sort people by their last name, then by their first name, preserving the order of the initial sort). There is however an interesting computational factor to consider. Regarding stability and the disadvantages of an algorithm not exhibiting it, Sternagel [6] notes that "swapping elements may cause memory updates on physical media". The statement is however completely void of meaning with regards to extensional specification of the algorithm behaviour, which only constraints outputs of the pertinent function, not the way it is produced. As such, it could very easily be the case that a stable sorting algorithm swaps equivalent elements more times than an unstable one, and only in the end permutes them into an order that is identical to the original one.

The point we are trying to make, is that implementation details, while definitely not negligible (swapping big records comes to mind naturally, but one can imagine a scenario where different magnetic tapes need to be unwound for the swap to happen), cannot be described by extensional way of behaviour specification. This observation leads to interesting discussion. For instance, one could perhaps consider a stable sorting algorithm idempotent. Certainly, once something is sorted, re–applying the function again should have no effect. However, this approach does not work intentionally. Not only is a stable algorithm allowed to swap equal elements, it can also arbitrarily swap unequal elements (as long as everything is properly reshuffled at the end). Therefore, the observation about idempotence makes sense from the extensional point of view only in the case where a distinguishing factor (such as a sort by selected key) is present, such that when two parts of a whole compare equal, does not mean that the entire record being swapped (or moved) is also equal.

Which brings us to a minor wrinkle for a sorting routine with an expected type signature of $sort : list\ \alpha \to list\ \alpha$. There is no way to even represent the notion of position. Indeed, consider the list $[0,0,0]$ and its sorted version $[0,0,0]$. How does one tell whether the sorting algorithm used for the abovementioned transformation was stable or unstable?

As such, the property of sort stability needs to be considered with some way of denoting the concept of position. Gouw, Boer and Rot [8] used an auxiliary indexing variable while Sternagel [6] utilized a very similar idea of first associating indices with each sequence to be sorted and then introduced a key function to separate the indices from elements. Considering we are concerned with a declarative version of insertion sort, it is more convenient for us to use associated indices.

The specification is then generally enriched by adding a third property of interest, which represents stability:

$$stable\ sort\ key \stackrel{\text{def}}{=} \forall x\ l, filter\ (\lambda y, key\ x = key\ y)\ (sort\ key\ l) = filter\ (\lambda y, key\ x = key\ y)\ l$$

Here $filter\ P\ l$ denotes a function that only keeps elements of l that satisfy the predicate P and *key* gives us the element portion of element–index association. The idea behind this formulation is that we take a look at the sequence containing a particular element post sorting (the left hand side of the above equality) and pre sorting (the right hand side). Then, if the said subsequences are equivalent for each of the elements, we can conclude that *sort* is stable with regards to the *key* function.

3. LEAN DEFINITIONS, BASIC PROPERTIES AND STABLE INSERTION SORT SPECIFICATION

Lean is a theorem prover based on dependent type theory. Its focus is on integration of interactive and automated theorem proving in a common framework. Let us examine a Lean functional implementation of the insertion sort along with auxiliary definitions we shall be using for the purposes of subsequent verification. It is worth noting right away, however, that given our interest in stability of sorting, we shall be considering indexing right away, in the form of ordered pairs; as such, we will be using $key = fst$ to avoid having to generalize the element–index association and passing a key function around; more on the topic later.

3.1. Basic definitions

Definition 3.1.
A function to insert an element of sequence into its correct sorted position.

```
def insert_le_key {α : Type}
[decidable_linear_order α] :
(α × ℕ) → list (α × ℕ) → list (α × ℕ)
  | a []       := [a]
  | a (h :: t) := if fst a ≤ fst h
                  then a :: h :: t
                  else h :: insert_le_key a t
```

As an aside, let us use the very first definition to familiarize a reader with basics of Lean that are perhaps not as common in other similar tools or differ in one way or another from common mathematical notation. For a more thorough explanation of the tool, one may refer to the Introduction to Lean [12]. Curly braces are used to introduce implicit arguments that are to be automatically inferred from following arguments. For the purposes of the paper, one can also consider square brackets as delimiters for implicit arguments; those will in general contain restrictions on types, similar to typeclasses in Haskell. As such, this construct defines a function called *insert_le_key* that operates on any type α for which there exists *decidable_linear_order*. It is a binary function from an ordered pair $(\alpha \times \mathbb{N})$ and a list of ordered pairs (list $(\alpha \times \mathbb{N})$) to a list of ordered pairs.

Remark 3.1. *From now on, all definitions as well as theorem statements shall be written in Lean; those are important to obtain understanding about the topic in question. Proofs however, will remain informal and brief, often simply describing proof structure and pertinent lemmas; this is because they have been fully mechanized. Omission of a definition or a theorem statement means that authors found it either common, trivial, unimportant or uninteresting. Some formulations may be slightly altered for aesthetics on paper; this is generally limited to omitting explicit type annotation. An interested reader is encouraged to inspect the entire development containing all of the definitions, theorems and proofs at the following url:* `https://github.com/ClanokAEI/LeanStableInsert/blob/master/stable_sort.lean`. *We can establish that all the proofs are correct under the assumption that we trust the Lean proof–checking core. For the purposes of the study, Lean 3.2.0 is used; as it is a tool in active development, older or newer versions may be incompatible.*

Remark 3.2. *For brevity, unless α is explicitly introduced, we will consider all of the upcoming definitions and theorems containing α as universally quantified over a type variable α, along with assumed implicit availability of decidable linear order for it. Referring to 3.1, we shall therefore be omitting the $\{\alpha$: Type$\}$ followed by [decidable_linear_order α] in statement formulations. Also, the product $\alpha \times \mathbb{N}$ representing an element to be sorted and an index respectively shall be abbreviated as γ.*

Definition 3.2.
A function that does the sorting.

```
def insert_sort (l : list α) : list γ :=
foldr insert_le_key [] (add_key l)
```

We shall be building sorted subsequences from right, as indicated by $foldr(ight)$, which denotes an operation of reduction. The *add_key* function will be defined shortly and is used to simply attach a set of unique indices to a sequence. For our intents and purposes, the indexed version of sort shall be used throughout the study.

Definition 3.3.
A projection to extract elements from their indexed representations.

```
def π₁ {α : Type} : list γ → list α := map fst
```

Where *map f l* builds a new sequence from *l* containing elements transformed by f, also preserving their order.

Definition 3.4.
A projection to extract indices from element–index pairs.

```
def π₂ {α : Type} : list γ → list ℕ := map snd
```

Definition 3.5.
A sorting function with the expected signature.

```
def sort : list α → list α := π₁ ∘ insert_sort
```

Now we can relate the *insert_sort* we shall be using throughout the study with *sort*. Thereafter, we shall no longer return to this technicality.

Theorem 3.1.
A standard sorting function sort can be obtained from insert_sort by composing it with π_1.

```
theorem insert_sort_well_behaved : ∀l : list α,
π₁ (insert_sort l) = sort l
```

Proof. Follows from the definition of function composition and 3.5. □

The already mentioned *add_key* function transforms a list of αs to a list of γs, with the first component being the element itself and the second component representing its index in the sequence.

Definition 3.6.
Assign a sequence of unique strictly increasing indices to a list of elements.

```
def add_key {α : Type} (l : list α) : list γ
:= add_key_from l 0
```

The *add_key_from* function is defined as follows:

Definition 3.7.
Sequences are assigned as second components of ordered pairs, while elements themselves are first components.

```
def add_key_from {α : Type} (l : list α)
: list γ := zip l (iota_asc_from n (length l))
```

Zip forms a list of ordered pairs from corresponding elements of its arguments and *iota_asc_from* is:

Definition 3.8.
Iota_asc_from m n is used to generate n positive integers starting with m, increasing by one.

```
def iota_asc_from : ℕ → ℕ → list ℕ
| m 0        := []
| m (succ k) := m :: iota_asc_from (succ m) k
```

3.2. Specification

As we have already stated, our approach is a little bit different from what is usually the case. With stability being an afterthought, it is added as a third property of the specification. As such, when trying to prove that any given sorting algorithm is stable, one needs to consider the way elements are being shuffled (whether it is directly with comparative sorts or indirectly with, well, non–comparative sorts), and possibly show that inversions are being undone in a way that preserves (or does not preserve) the original relative order of equivalent elements. This process is completely disjoint from the notion of being sorted and all the information available from this property is thrown away, having to be recovered in the stability proof separately. We therefore enhance the notion of what it means to be sorted, expressing stability along with the property of being sorted. It can be viewed as an extension of what Gouw, Boer and Rot [8] used, utilizing an index once equality is established.

Definition 3.9.
An inductive predicate describing a sequence that is stably sorted with regards to key.

```
inductive stable_sorted {α : Type} {β : Type}
[strict_order α] [decidable_linear_order β]
(key : α → β) : list α → Prop
| stable_empty    : stable_sorted []
| stable_one      : ∀x, stable_sorted [x]
| stable_more_eq  : ∀(x y : α) (t : list α),
      key x = key y → x < y →
      stable_sorted (y :: t) →
      stable_sorted (x :: y :: t)
| stable_more_neq : ∀(x y : α) (t : list α),
      key x < key y →
      stable_sorted (y :: t) →
      stable_sorted (x :: y :: t)
```

Remark 3.3. *A very important thing to note here is that throughout the study, we will be using the definition under assignment $\alpha \leftarrow \gamma$, for which the strict comparison operation with $<$ is defined to compare second components; those represent indices. As such, $x < y$ for $(a, m) < (c, n)$ is, by definition, $m < n$.*

Formulating stability along with sortedness reveals a very interesting idea. What we are technically doing is converting a set of linearly ordered βs to a set of strictly ordered αs, which we can then sort as if under the assumption that no elements are equivalent in the first place (considering elements cannot overlap, they must have unique indices representing their position). Colloquially speaking, we can very explicitly make the distinction between strictly less than and equal in the sense that $\leq$ is split into $<$ and $=$.

For completeness sake, we shall reiterate the property of permutations, this time in Lean. It remains unchanged.

Definition 3.10.
A definition of permutation in terms of equivalent number of occurrences of each element.

```
def permutation_dec {α : Type} [decidable_eq α]
(l₁ l₂ : list α) : Prop :=
∀x, count_dec x l₁ = count_dec x l₂
```

Definition 3.11.
A function count_dec x l counts occurrences of x in l.

```
def count_dec {α : Type} [decidable_eq α] :
  α → list α → ℕ
  | _ []       := 0
  | x (h :: t) := if x = h
                  then succ (count_dec x t)
                  else count_dec x t
```

All of the components are in place now to state that *insert_sort* complies with the specification:

Theorem 3.2.
Insert_sort creates a permutation of its input list that is stably sorted.

```
theorem sort_correct_and_stable :
∀(l₁ : list α) (l₂ : list γ),
  l₂ = insert_sort l₁ →
  stable_sorted fst l₂ ∧
  permutation_dec (π₁ l₂) l₁
```

We are using permutation of projection of elements over π_1 rather than leaving them with their indices. The reason for this is twofold, but purely practical. First, we avoid having to define decidable equality on γ (even though it is induced by decidable equality on its components). Second, it gives us (admittedly somewhat pointless) freedom with regards to indices we use when dealing with expressing the permutation property.

4. PROOF OF PERMUTATION

The entire proof will be split into two parts, corresponding with each of the conjuncts. Let us first prove correctness with regards to permutation, for which we shall need a few facts about permutations and the sorting routine.

Lemma 4.1.
Sorting an empty list yields an empty list.

```
lemma sort_empty : insert_sort [] = []
```

Proof. Follows from definitions 3.2, 3.6, *zip*, *zip_with* and *foldr*. □

Lemma 4.2.
Inserting an element into an empty list results in a list containing only that element.

```
lemma insert_into_empty :
  ∀h : γ, insert_le_key h nil = [h]
```

Proof. Follows directly from 3.1. □

Lemma 4.3.
Inserting an element increases its count in a sequence.

```
lemma count_over_projection :
∀x xₖ (l : list γ),
count_dec x (π₁ (insert_le_key (x, xₖ) l)) =
succ (count_dec x (π₁ l))
```

Proof. By induction on structure of l with $h :: t$. For an empty list, the property follows from 4.2. Otherwise, assume $x \leq key\ h$, in which case it holds from 3.11. For $\neg(x \leq key\ h)$, we also have $x \neq key\ h$, then from 3.11 and from the inductive hypothesis the property holds. □

Lemma 4.4.
The number of occurrences of x in a sequence l does not change if we insert an element y distinct from x into l.

```
lemma count_over_projection_neq :
∀x y xₖ (l : list γ), x ≠ y →
count_dec x (π₁ (insert_le_key (y, xₖ) l)) =
count_dec x (π₁ l)
```

Proof. Shares structure with the proof of 4.3. In the inductive step, finish by case analysis on $x = key\ h$; both parts then follow from the inductive hypothesis. □

Lemma 4.5.
After projection of keys, an index being inserted is irrelevant.

```
lemma insert_second_irrelevant :
  ∀a b c (l : list γ),
    π₁ (insert_le_key (a, b) l) =
    π₁ (insert_le_key (a, c) l)
```

Proof. By induction on shape of l, then by cases on $a \leq key\ h$ in the $h :: t$ case. □

Lemma 4.6.
Inserting the same element into two equivalent lists preserves the equality.

```
lemma insert_le_x_right_neutral :
  ∀x (l₁ l₂ : list γ),
    π₁ l₁ = π₁ l₂ →
    π₁ (insert_le_key x l₁) =
    π₁ (insert_le_key x l_2)
```

Proof. By induction on l_1, then by cases on l_2. □

Lemma 4.7.
Insertion ignores indices.

```
lemma projection_over_foldr :
∀(l : list α) (l₁ l₂ : list ℕ),
  length l₁ = length l₂ →
  π₁ (foldr insert_le_key nil (zip l l₁)) =
  π₁ (foldr insert_le_key nil (zip l l₂))
```

Proof. By induction on structure of l with $h :: t$, followed by shape analysis of l_1 and l_2 with $h_1 :: t_1$ and $h_2 :: t_2$ respectively; when both empty, the property holds trivially. Mismatching lengths lead to contradiction. In the last case, in $\pi_1\ (insert\_le\_key\ (h, h_1)\ (foldr\ insert\_le\_key\ nil\ (zip\ t\ t_1)))$ $= \pi_1\ (insert\_le\_key\ (h,h_2)\ (foldr\ insert \ldots\ nil\ (zip\ t\ t_2)))$ we can unify $h_1 = h_2$ from 4.5, then we can remove *insert_le_key* because of 4.6. The rest follows from the inductive hypothesis. □

Lemma 4.8.
Sequences generated by iota_asc_from m n are of length n.

```
lemma iota_asc_from_len : ∀ m n,
  length (iota_asc_from m n) = n
```

Proof. By straightforward induction on n. □

Theorem 4.1.
The output of sort is a permutation of its input.

```
theorem sort_permutes :
  ∀l₁ (l₂ : list γ), l₂ = insert_sort l₁ →
  permutation_dec (π₁ l₂) l₁
```

Proof. By induction on structure of l_1, there are two cases to consider. When l_1 is empty, l_2 is also empty from 4.1, the rest follows trivially. For $h :: t$, we assume an arbitrary h_e from 3.10, then proceed by cases on $h = h_e$. Under assumed equality, first use 4.3 to get to $count\_dec\ h\ (\pi_1\ (foldr\ insert \ldots\ nil\ (add\_key\_from\ t\ 1)))$ $= count\_dec\ h\ t$, then utilize 4.7 $iota\_asc\_from\ 0\ (length\ t)$ for indices to match the shape of the inductive hypothesis. Their length is short enough, which can be seen from 4.8. The $h \neq h_e$ case is analogical to the previous one, but with 4.4 instead of 4.3, as the element we are trying to count and the current head mismatch. □

5. PROOF OF STABLE SORTEDNESS

Now on to showing conformance with regards to the notion of being sorted stably, beginning by proving various auxiliary facts. Do note that we do not need to deal with *filter* and how undoing inversions by the sorting procedure affects permutations being generated with regards to it. Useful facts first.

Lemma 5.1.
Iota_asc_from m n can be coerced into shape h :: t assuming $1 \leq n$.

```
lemma iota_split : ∀h t m n,
  1 ≤ n →
  h :: t = iota_asc_from m n →
  h = m ∧ t = iota_asc_from (succ m) (pred n)
```

Proof. Both conjuncts hold from 3.8. □

Definition 5.1.
For convenience, we define a weaker notion of sorted; one that does not allow for variation in ordering criteria nor type of elements to sort and simply represents strictly increasing integer sequences:

```
inductive strictly_increasing : list ℕ → Prop
  | si_empty : strictly_increasing []
  | si_one   : ∀x, strictly_increasing [x]
  | si_more  : ∀h₁ h₂ (t : list ℕ),
          h₁ < h₂ →
          strictly_increasing (h₂ :: t) →
          strictly_increasing (h₁ :: h₂ :: t)
```

Lemma 5.2.
Iota_asc_from m n generates strictly increasing sequences.

```
lemma iota_from_strictly : ∀ m n l,
l = iota_asc_from m n → strictly_increasing l
```

Proof. By straightforward induction on n with k, there are two cases to consider. For $n = 0$, the sequence is empty and therefore strictly increasing. Otherwise we proceed by cases on shape of derivation of the sequence $iota\_asc\_from\ (succ\ m)\ k$, spawning two additional cases. When the list is empty, a sequence of length one is strictly increasing. The final case follows trivially from the inductive hypothesis. □

Lemma 5.3.
Tail of a strictly increasing sequence is too strictly increasing.

```
lemma strictly_increasing_over_head :
  ∀h t, strictly_increasing (h :: t) →
  strictly_increasing t
```

Proof. On shape of derivation of $strictly\_increasing\ (h :: t)$. □

Lemma 5.4.
Head of a strictly increasing sequence is smaller than any other element in the tail.

```
lemma strictly_split : ∀h t,
  strictly_increasing (h :: t) →
  (∀x, x ∈ t → h < x)
```

Proof. By induction on t. The empty case is contradictory. When $t = h_1 :: t_1$, then either $x = h_1$ or $x \in t_1$. The former is trivial, the latter follows from the inductive hypothesis and from the fact that $strictly\_increasing\ (h :: h_1 :: t_1) \rightarrow strictly\_increasing\ (h :: t_1)$, which can be shown from 5.3 and 5.1. □

Definition 5.2.
A predicate representing presence of an element in a list.

```
def in_comp {α : Type} : α → list α → Prop
  | _ []       := false
  | x (h :: t) := x = h ∨ in_comp x t
```

We shall be using ∈ as notation for in_comp henceforth.

Now a lemma that allows us to skip the entirety of stability proof. Under normal circumstances, one assumes that inserting into a sorted sequence yields a sorted sequence. This is however untrue in our setting, as we may be inadvertently inserting a duplicate element with a greater index. Therefore, we formulate the theorem as follows:

Lemma 5.5.
Inserting an element into a sorted sequence yields a sorted sequence assuming the index of the element being inserted is strictly smaller than any other index in the sequence.

```
lemma insert_pres_sorted : ∀l (e : γ) elem key,
  e = (elem, key) →
  (∀x, x ∈ π₂ l → key < x) →
  stable_sorted fst l →
  stable_sorted fst (insert_le_key e l)
```

Remark 5.1. *The assumption is a bit stronger than it necessarily needs to be, as we only need to assure that the index being inserted is strictly smaller than indices corresponding with equivalent elements. However, this formulation is more convenient, given the expansion of indices at reduction. If we consider insert_le_key = insert for just a moment, then from foldr insert [ ] [(_,0),(_,1),(_,2)], we get insert (_,0) (insert (_,1) (insert (_,2) [ ])). As an effect of folding from right, we always have an index that is strictly less than every other one in the sorted subsequence.*

Proof. By induction on l, we have two cases to consider. For an empty sequence, the theorem follows from 4.2. For the list of shape $h :: t$, cases when $elem = key\ h$ or $elem < key\ h$ hold trivially. For $elem > key$, proceed by shape of derivation of $stable\_sorted\ (h :: t)$. The case where the derivation is empty is contradictory. The $stable\_sorted\ [h]$ case is trivial. In the $stable\_sorted\ [h :: y :: t_2]$ case for equal head elements, note that $elem > key\ y$, as such we eventually need to show $stable\_sorted\ fst\ (y :: insert\_le\_key\ (elem, key)\ t_2)$, using $stable\_more\_eq$ from 3.9. Since $elem > key\ y$, we get to $insert\_le\_key\ (elem,\ key)\ (y :: t_2)$, which then follows from the inductive hypothesis. In the remaining case where $stable\_sorted\ [h :: y :: t_2]$ and head elements are unequal, there are three options. The $elem < key\ y$ and $elem = key\ y$ are similar and follow from $stable\_more\_eq$ and $stable\_more\_neq$ of 3.9. The remaining case when $elem > key\ y$ is analogical to the corresponding case of the derivation from $stable\_more\_eq$. □

Lemma 5.6.
If an index comes from a list that is created by inserting an element into its subsequence, then it must have come from the subsequence or it is the element being inserted.

```
lemma over_second_split :
  ∀x elem idx (l : list γ),
    x ∈ π₂ (insert_le_key (elem, idx) l) →
    x = idx ∨ x ∈ π₂ l
```

Proof. By straightforward induction on l. Note that the case when l is empty is contradictory. □

Lemma 5.7.
If an index is somewhere in a sorted list, then it must have come from the initial list of indices.

```
lemma over_second_projection :
∀x (l₁ : list α) (l₂ : list ℕ),
  length l₁ = length l₂ →
  x ∈ π₂ (foldr insert_le_key nil (zip l₁ l₂))
  → x ∈ l₂
```

Proof. By induction on l_1. The case where the list is empty is trivial. Now consider $h :: t$, then by shape of l_2 we get mismatching lengths for empty l_2 and as such a contradiction. Otherwise, from 5.6 we can assume $x = h_2 \vee x \in \pi_2\ (foldr\ insert\_le\_key\ nil\ (zip\ t\ t_2))$. The former case holds trivially, the latter follows from the inductive hypothesis. □

Now we would like to show the left conjunct of 3.2. However, the formulation in this form is not very permissive in terms of indices used, as *insert_sort* uses *add_key*. As such, let us first show the following:

Theorem 5.1.

```
theorem sort_sorts_stably_aux :
∀(l : list γ) (l₁ : list α) (l₂ : list ℕ) m n,
  length l₁ = length l₂
  l₂ = iota_asc_from m n
  l = foldr insert_le_key [] (zip l₁ l₂)
  stable_sorted fst l
```

Proof. By induction on l_1, then by cases on l_2. The case where both lists are empty is trivial. The cases where lengths mismatch are contradictory. We are left with $l_1 = h :: t$ and $l_2 = h_2 :: t_2$. Note that from 5.1 we get $h_2 = m$ and $t_2 = iota\_asc\_from\ (succ\ m)\ (pred\ n)$. Also note that from 5.2 and 5.3, t_2 is strictly increasing. Therefore, it is sufficient to show $stable\_sorted\ fst\ (foldr\ insert\_le\_key\ nil\ (zip\ t\ t_2))$ from 5.5. This is valid because we can prove for an arbitrary x that $h_2 < x$ from 5.4 and 5.7, which is the stability preserving condition. The rest follows from the inductive hypothesis. □

Finally we can formulate the desired property as a simple corollary of 5.1.

Corollary 5.1.
Sort creates stably sorted sequences.

```
theorem sort_sorts_stably :
∀(l₁ : list α) (l₂ : list γ),
  l₂ = insert_sort l₁ → stable_sorted fst l₂
```

Proof. As the indexing sequence l_2, choose $iota\_asc\_from\ 0\ (length\ l_1)$ and use 4.8 to show that lengths of the lists being zipped coincide. The rest of the proof is trivial. □

From 5.1 and 4.1, we can easily show 3.2, which is what we had set out to do.

6. PROOF OF IRRELEVANCE OF STABILITY FOR NON–REPEATING SEQUENCES

Intuitively, stability is a property that does not require consideration if the sequence to be sorted contains no duplicates. Let us formalize this specification in Lean and prove equivalence of unstable and stable sorting under the said assumption.

Definition 6.1.
A predicate expressing sequences containing distinct elements.

```
inductive no_dup {α : Type} : list α → Prop
  | no_dup_empty : no_dup []
  | no_dup_one   : ∀x, no_dup [x]
  | no_dup_more  : ∀x (l : list α),
                     no_dup l →
                     ¬(x ∈ l) →
                     no_dup (x :: l)
```

Lemma 6.1.
Tail of a list containing no duplicates has distinct elements only.

```
lemma no_dup_tail {α : Type} : ∀h (t : list α),
no_dup (h :: t) → no_dup t
```

Proof. By cases on $no\_dup\ (h :: t)$. □

Lemma 6.2.
If an element is not in a list, it must be distinct from its head.

```
lemma not_in_means_neq {α : Type} :
∀(a b : α) (t : list α),
  ¬a ∈ b :: t → a ≠ b
```

Proof. Follows definitionally from 5.2. □

Lemma 6.3.
Heading elements of sequences with distinct elements only are unequal.

```
lemma no_dup_are_unequal {α : Type} :
∀(a b : α) (t : list α),
  no_dup (a :: b :: t) → a ≠ b
```

Proof. By cases on $no\_dup\ (h :: t)$ we only have one option, namely *no_dup_more* of 6.1, which follows from 6.3. □

Theorem 6.1.
A sequence containing distinct elements is sorted if and only if it is stable sorted.

```
theorem distinct_always_stable
{α : Type}
[x : decidable_linear_order α]
[y : strict_order γ] :
∀(l : list γ),
  no_dup (π₁ l) → (
    stable_sorted fst l ⟺
    sorted (π₁ l)
  )
```

Remark 6.1. *We assumed strict order on γ for convenience of formulation. It is worth explicitly noting that with this particular way of stating the theorem, we can immediately see that indices become irrelevant (as they are projected away by π_1) even in the direction sorted → stable_sorted. Also, we are using sorted definition from 2.1.*

Proof. For the direction *stable_sorted* → *sorted*, by straightforward induction on the shape of derivation of *stable_ sorted l* using 6.1. For *sorted* → *stable_sorted*, begin by induction on $sorted\ (\pi_1\ l)$. The cases *empty* and *one* are trivial. For *more*, let the first and the second element be h_1, h_2. Note that $h_1 \leq h_2$. In the case where $h_1 < h_2$, use *stable_more_neq* of 3.9, the rest follows from the inductive hypothesis. The case where $h_1 = h_2$ is contradictory because of 6.3. □

We could use this result to optimize a call to a stable sorting algorithm by using an unstable one (under the assumption that the latter is faster tan the former), if we could prove that elements to be sorted are distinct. More importantly, it demonstrates a scenario where stability is an observable effect, and as such should be a consideration in specifications of sorting functions. Algorithms such as radix sort also show that stability is relevant even in pure environments with memory being completely transparent; the notion of relative position simply does not go away, even despite the fact that we had to explicitly introduce indices to model it.

7. CONCLUSION

The entire theme of the study revolves around using Lean to *fully* formalize a declarative version of the insertion sort algorithm and subsequently proving its correctness along with stability, which can be formulated conveniently, without much extra effort, directly within the notion of being sorted. While this does slightly complicate proofs showing that a sequence is sorted (stably), it allows us to skip $filter$ing (or processes similar to it) of sequences entirely. Therefore, with regards to complexity of proofs, we trade the entire notion of separate stability for arguably trivial proofs dealing with projections of elements as opposed to elements themselves, and for a slightly stronger relationship between sorted sequences and their properties when an element is being inserted into a correct position. It is also worth noting that when we interpret permutations as multisets containing elements of equal counts, indices can be completely ignored. This condition is present in the specification to simply relate input sequences to output sequences. Furthermore, we have shown that we can equate sorting and stable sorting if the sequence in question contains no duplicates – thereby demonstrating that stability cannot be forgotten about in all cases, as it is necessary for the proof. While it is true that indices are artificially added in this scenario, we argue that the notion of relative position does not disappear simply because in some cases, memory (and memory locations) are transparent.

8. ACKNOWLEDGEMENT

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-15-0055. The paper was supported by project KEGA no. 079TUKE-4/2017.

REFERENCES

[1] FOLEY, M. – HOARE, C.A.R.: Proof of a recursive program: Quicksort, *The Computer Journal, Volume 14, Issue 4* pp. 391–395, 1971.

[2] The Coq development team: The Coq proof assistant reference manual, *LogiCal Project*, 2004. `http://coq.inria.fr`

[3] FILLIÂTRE, J.C. – MAGAUD, N.: Certification of sorting algorithms in the Coq system, 1999.

[4] TUSHKANOVA, E. – GIORGETTI, A. – KOUCHNARENKO, O.: Specifying and Proving a Sorting Algorithm, *Rapport de recherche LIFC*, 2009.

[5] MARCHÉ, C.: The Krakatoa tool for Deductive Verification of Java Programs, *Winter School on Object-Oriented Verification, Viinistu, Estonia*, 2009.

[6] STERNAGEL, C.: Proof Pearl – A Mechanized Proof of GHC's Mergesort, *Journal of Automated Reasoning, Volume 51, Issue 4* pp. 357–370, 2013.

[7] NIPKOW, T. – WENZEL, M. – PAULSON, L.C.: Isabelle/Hol: A Proof Assistant for Higher-order Logic, *Springer-Verlag*, 2002.

[8] GOUW, S.d. – BOER, F.d. – ROT, J.: Proof Pearl: The KeY to Correct and Stable Sorting, *Journal of Automated Reasoning, Volume 53, Issue 2* pp. 129–139, 2014.

[9] BECKERT, B. – HÄHNLE, R. – SCHMITT, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach, *Lecture Notes in Computer Science, Series Volume 4334, Springer*, 2007.

[10] MOURA, L.d. – KONG, S. – AVIGAD, J. – DOORN, F.v. – RAUMER, J.v.: The Lean Theorem Prover, *25th International Conference on Automated Deduction (CADE-25), Berlin, Germany*, 2015.

[11] SEDGEWICK, R.: Algorithms, *Addison-Wesley*, 1988.

[12] AVIGAD, J. – MOURA, L.d. – EBNER, G. – ULLRICH, S.: An Introduction to Lean, 2017. `https://leanprover.github.io/introduction_to_lean/introduction_to_lean.pdf`

Received March 20, 2018, accepted May 3, 2018

BIOGRAPHIES

František Silváši is a Ph.D. student at Technical University of Košice working on formal software verification utilizing automated reasoning.

Martin Tomášek received Ph.D. degree in Software and Information Systems in 2005 and currently works as an associate professor at Technical University of Košice. His research interests include concurrency theory, distributed systems, and cloud computing.