Implementation and Verification of Sorting Algorithms with the Interactive Theorem Prover HOL

Sara Quarfot Orrevall
Abstract

Implementation and Verification of Sorting Algorithms with the Interactive Theorem Prover HOL

Sara Quarfo Orrevall

As the world becomes increasingly reliant on technology and the technology becomes increasingly complex, ensuring software correctness is becoming both increasingly important and difficult. Methods like software testing are rarely enough to guarantee that a program will always work as intended. Formal methods offer attractive alternatives. Using formal methods, properties about software can be unambiguously proven for all possible input.

In this project we use the interactive theorem prover HOL to define and formally verify a simplified version of the popular sorting algorithm Timsort. We also formalize the time-complexity property and prove the best-case time-complexity of the simplified algorithm. We intended to use the CakeML compiler to generate verified machine code from the HOL definitions, and thus produce an end-to-end verified executable program. Because of time constraints, we instead generated ML code using EmitML. The resulting ML code is not guaranteed to retain the proven properties during execution. The project demonstrates how sorting algorithms can be formally verified and provides parts that could be re-used to verify the actual Timsort algorithm.
Contents

1 Introduction 3

2 Project 4
  2.1 Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  2.2 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  2.3 Delimitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

3 Background 5
  3.1 Formal verification and theorem proving . . . . . . . . . . . . . 5
    3.1.1 Interactive theorem provers . . . . . . . . . . . . . . . . . 6
    3.1.2 HOL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    3.1.3 HOL Syntax . . . . . . . . . . . . . . . . . . . . . . . . . 7
  3.2 Properties of sorting algorithms . . . . . . . . . . . . . . . . . . 7
  3.3 Timsort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

4 Related work 10

5 Formalizing the specification 10
  5.1 Permutation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  5.2 Sortedness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

6 Verifying stacksort 12
  6.1 Proving properties of get_run . . . . . . . . . . . . . . . . . . . 13
  6.2 Proving properties of decompose . . . . . . . . . . . . . . . . . 16
  6.3 Proving properties of merge-collapse . . . . . . . . . . . . . . . 18
  6.4 Verifying stacksort . . . . . . . . . . . . . . . . . . . . . . . . . . 20

7 Runtime analysis of stacksort 20
  7.1 Formalizing time-complexity . . . . . . . . . . . . . . . . . . . . 21
  7.2 Best-case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
  7.3 Worst-case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

8 Code generation 23

9 Discussion 24
  9.1 Conclusion and future work . . . . . . . . . . . . . . . . . . . . 24
1 Introduction

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger W. Dijkstra[1]

The world is becoming increasingly dependent on technology. Failure or unexpected behavior of said technology can have dire consequences on many people’s lives. Software errors throughout history have, for example, contributed to massive power outages[2] and given lethal overdoses of radiation during radiation treatment[3]. Meanwhile, much of the technology we use is becoming increasingly complex. Accurately predicting all the ways a piece of software will - or can - be used is difficult, if not impossible[4]. Yet these predictions are what software testing relies on. Testing[5] consists of running the software on different input to see if the outcome is the expected one. While the tests are intended to cover all important cases, they are not all-encompassing. This approach alone is often not enough to ensure that a product will always work as intended.

Fortunately, there are other approaches. Formal methods[6], based on mathematics and logic, can be used to prove properties of a function. Rather than executing the function, formal methods analyze its structure and semantics and can prove that a property holds for all input. This provides a reliability that testing simply cannot. We can use these methods to formally verify the function, that is, confirm by mathematical proof that it fulfills some given specification. Depending on what one wants to prove and which tools are used, the process can range anywhere from being completely automated to being done with pen-and-paper proofs. Both ends of the spectrum can be limiting in their use. Automated provers are often unable to prove much more than trivial problems, while pen-and-paper proofs are tedious and prone to human error[7]. Because of this, interactive theorem provers are often used. These combine automated processes and directions from a human user.

Aside from working properly, an important aspect of any algorithm is its time- and space-complexity[8]. These are measurements of expected running time and memory usage - properties that are crucial when determining which algorithm to use. The time-complexity gives us an indication of how useful an algorithm is as the size of the data increases. Two algorithms solving the same problem but with different time-complexities can have a difference of years in execution time[8, p. 18]. Knowing the time-complexity means we can make an informed decision about when to (not) use an algorithm, and can also highlight a need for improvement. As for space-complexity, not knowing how much space an algorithm requires could lead to the program crashing if given too little, or wasting space if given too much. Just like other properties, time- and space-complexity can be formalized and proven with formal methods.
Timsort\[9\] is a sorting algorithm created by Tim Peters that combines parts of merge sort and binary insertion sort. It works especially well on data that contains already sorted segments, which Peters believes real-world data very often does (“real-world data” being that which is encountered outside of testing/benchmarking scenarios). Both Python and Java use Timsort among their default sorting algorithms, Python since 2003 and Java since 2012\[10\]. One might think that any bugs in an algorithm used by such popular programming languages would have been discovered after years of frequent use. In 2015, this was found to not be the case for Timsort. De Gouw et al, in an attempt to formally verify the Java implementation\[11\], discovered a bug that could cause the algorithm to crash. They then, again with formal methods, found a way to solve the issue and proved their solution to be bug-free. The bug was found to be present in the Python version as well. The authors go on to say that while the bug is not actually likely to occur, malicious users could exploit it for denial-of-service attacks. Clearly, formal verification can be worthwhile even for tried-and-tested algorithms.

2 Project

In this section we describe the goals of this project, the methodology used, and the delimitations.

2.1 Goal

The goal of this project is to provide an end-to-end formally verified, executable, and simplified version of the Timsort sorting algorithm, which we call stacksort. We also attempt to prove its best- and worst-case time-complexity. We aim to demonstrate techniques for how sorting algorithms can be formally verified, and to provide parts that can potentially be used to do the same for the actual Timsort algorithm.

2.2 Methodology

The verification is done using formal methods in the interactive theorem prover HOL\[12\]. First, the algorithm is defined in HOL in order to verify it. The definitions are not executable, however - executable code must be generated from them. HOL chosen specifically because of its compatibility with CakeML\[13\], a programming language with verified mechanism for generating executable code from definitions like those in HOL. Using these guarantee that any properties proven of the definitions will be retained in the final executable. Thus it allows us to provide an end-to-end verified and executable implementation. Another tool for code generation is the HOL module EmitML\[14\]. This module can convert HOL definitions to ML\[15\] code, which can then be compiled. This process does not guarantee retention of properties, however.
2.3 Delimitations

We do not attempt to implement and verify the full Timsort algorithm because of time constraints and lack of experience with HOL and theorem proving in general. Even when describing the full Timsort algorithm in this report, a few optimizations are ignored. The optimizations are there to minimize memory usage and maximize cache-performance. They do not change the algorithm’s time-complexity nor its output, and so ignoring them does not affect any of the properties we are interested in.

3 Background

In the following sections, formal methods, HOL, the specification for sorting algorithms, and the Timsort algorithm are described in more detail.

3.1 Formal verification and theorem proving

To verify a system is to confirm that it adheres to its specification. Formal verification is the process of achieving this through formal methods. The idea of formal methods is to use languages based on mathematical theory with strict re-writing rules. The languages allow us to express properties unambiguously and to prove them. These proofs can be formulated to all-encompassing, which means that they can prove something about all possible input. This is a clear advantage over methods like testing or simulations, as these only expose the software to a limited amount of cases/scenarios, and only prove that the software works properly in those exact cases/scenarios. It is still possible for the software to malfunction in others. However, formal methods come with their own set of challenges. Common complaints include the need for mathematical sophistication and a gap between theory and practice - many find it hard to see how to apply the methods to real-world problems.

We will make use of theorem proving specifically. This method expresses functions and properties according to some mathematical logic system. Axioms and inference rules defined by this system can then be used to conduct proofs. A particular proof technique, structural induction, can be used to prove properties of functions with infinite possible input. The steps of weak structural induction are as follows:

1. Prove that the property holds for the base case(s).

2. Assume that the property holds for some input of size $n$ - this is referred to as the induction hypothesis. Use the induction hypothesis to prove that the property holds for input of size $n + 1$.

For functions taking lists as input, $n$ measures the length of these. Strong induction is a variation that differs only in the induction hypothesis. It instead assumes that the property holds for any input of size $n$ or less. Weak and strong
induction are equivalent, but a strong induction hypothesis may be needed to prove properties of more complicated algorithms.

3.1.1 Interactive theorem provers

Interactive theorem provers - sometimes shortened ITPs - are tools to aid people in constructing formal proofs. Fully automated theorem provers are many times not powerful enough to conduct non-trivial proofs\[7\]. A random non-trivial proof comes with too many possibilities for an automated prover to find the appropriate steps to solve it, at least within reasonable time and space constraints. Humans are better at discerning an appropriate approach. Pen-and-paper proofs, on the other hand, are both tedious and prone to human error. ITPs fall somewhere in between. They can proof-check and keep track of different (sub)goals while the user directs the proof. They often also provide some level of automation that allow users to execute less complicated but perhaps still lengthy proofs, or parts of proofs, in fewer steps.

3.1.2 HOL

HOL\[12\] is an interactive theorem prover for higher order logic, based on the functional programming language ML(Meta-Language)\[15\]. Users can construct theorems by putting that which they want to prove on a goalstack and solve it, i.e prove it correct. Proofs are conducted in a top-down manner, by applying so-called tactics. These rewrite, simplify, or strip the goal down to smaller and simpler sub-goals that can eventually be solved with an inference rule or some axiom. For example, consider the following goal:

\[ \forall A B. \; A \land B \iff B \land A \]

To prove this theorem, we can begin by applying the tactic REPEAT GEN_TAC. The tactic removes quantifiers from a goal until there are none left. Thus the goal now looks like

\[ A \land B \iff B \land A \]

To prove the equality we must prove an implication in both directions, as can be seen by the theorem EQ_IMP_THM:

\[ \forall t1 t2. \; (t1 \iff t2) \iff (t1 \implies t2) \land (t2 \implies t1) \]

The tactic EQ_TAC uses EQ_IMP_THM to split the goal into the two following sub-goals

\[ A \land B \implies B \land A \]

\[ B \land A \implies A \land B \]

The bottom goal is dealt with first. The outermost connective of a goal can be removed with STRIP_TAC. We can apply this repeatedly with REPEAT STRIP_TAC to strip the goal both of the implication and the conjunctions to get

6
Above the horizontal line are the assumptions, and below the conclusion. We must prove that the assumptions $A$ and $B$ lead to the conclusion $A$ in the first sub-goal and $B$ in the second. The tactic `ASM_REWRITE_TAC[]` uses the assumptions of a goal to rewrite or prove the conclusion. Additional theorems can be added within the brackets for the tactic to use. Since we have exactly what we want to prove in the assumptions, no additional theorems are needed in this case. Proving both of these sub-goals in turn proves $B \land A \implies A \land B$.

The remaining goal $A \land B \implies B \land A$ can be solved in the same manner, which then proves the original goal, resulting in the theorem

$$\vdash \forall A B. \ A \land B \iff B \land A$$

However, HOL also provides many automated procedures. The original goal could have been solved by simply applying `METIS_TAC[]`, which can reason about first order logic. Like `ASM_REWRITE_TAC[]`, this tactic can take additional theorems, but none are needed in this case.

This project will be using HOL4 specifically, which at the time of writing is the most recent version of HOL.

### 3.1.3 HOL Syntax

Here we describe some HOL syntax that will be used in later parts of the report.

- `[]`: Empty list
- `[x]`: List containing one element, $x$
- `[x;y]`: List containing two elements, $x$ and $y$
- `x::xs`: Append the element $x$ to the list $xs$
- `xs ++ ys`: Concatenate the lists $xs$ and $ys$
- `$\$`: Turn an infix operator into a prefix operator, for example $\$+ a b \iff a + b$

### 3.2 Properties of sorting algorithms

Sorting algorithms are expected to fulfill the following two properties:

1. The output is a permutation of the input
2. The elements of the output appear in non-decreasing order according to some given total order

Non-decreasing order means that no element with a higher index may be lesser than one with a lower index. They may be of equal value. Implicitly, the algorithm is also expected to terminate for all input.
A total order is a binary relation $R$ that fulfill the following properties\[21]:

1. Antisymmetry: $xRy$ and $yRx$ implies $x = y$
2. Transitivity: $xRy$ and $yRz$ implies $xRz$
3. Connexity: either $xRy$ or $yRx$

Note that connexity implies reflexivity, that is, $xRx$. Interestingly, no literature was found describing exactly why the relation is required to be total. In section \[5.2\] we discuss why these properties are important for the concept of sorting, and what role they will play in our proofs (if any).

### 3.3 Timsort

Timsort\[9\], named after its creator Tim Peters, combines parts of binary insertion sort and merge sort. In short, the algorithm divides the list by finding segments of already sorted elements, referred to as “runs”. If a run is under a certain length, it is extended by inserting subsequent elements using binary insertion sort. The runs are then merged in a way similar to the merging part of merge sort. Timsort will now be described in more detail, while still skipping over some details regarding memory and cache optimizations that are not relevant to this report.

The algorithm goes over the data looking for any already ordered segments, either in non-decreasing or strictly decreasing order. These are put on a stack, the decreasing ones being reversed first. In order to set up favorable conditions for the merging, the runs should be at least $\text{minrun}$ elements long. The size of $\text{minrun}$ depends on the length of the list. If a run is too short, it is extended with subsequent elements using binary insertion sort.

The four runs at the top of the stack, W, X, Y, and Z, with Z being the most recently added, are merged under certain conditions. The idea is both to keep the merges balanced (i.e. of roughly the same size) and the stack from growing too large. There are four invariants\[22]:

1. $|X| \geq |Z|$
2. $|X| > |Y| + |Z|$
3. $|W| > |X| + |Y|$
4. $|Y| > |Z|$

If invariant 1 is violated, X and Y are merged. If any of the others are violated, Y and Z are merged. Merging continues until all invariants hold. As the stack shrinks by one with each merge, merging cannot go on forever - it will eventually terminate. The merging process is perhaps best described through examples.
There are two modes, regular and galloping.

Say the two lists \( A = [1,2\ldots10] \) and \( B = [1,2\ldots10] \) are to be merged with the relation \( \leq \). First \( A[0] \) is compared to \( B[0] \), and since \( 1 \leq 1 \), \( A[0] \) is moved to the merged area. The merging continues with \( A' = [2,3\ldots10] \) and \( B = [1,2\ldots10] \). This time, \( B[0] \) is the smaller element, so \( B[0] \) is moved to the merged area. Merging continues similarly with \( A' = [2,3\ldots10] \) and \( B' = [2,3\ldots10] \) until all elements are merged. These lists were merged all in regular mode.

Now consider the lists \( A = [1,2\ldots100] \) and \( B = [50,51\ldots150] \). Since the first 50 elements of \( A \) are all lesser or equal than \( B[0] \), elements from \( A \) will be moved to the merged area several times in a row. Timsort keeps track of this number. The idea is that if elements from only one list is moved to the merged area many times in a row, there is a good chance that that pattern will continue for a while, and that we could save in on comparisons by skipping elements. So if the number reaches the \texttt{min_gallop} number, merging will go into galloping mode. This number is usually set to 8 (more on that in a bit). The above lists would therefore enter galloping mode after the first 8 elements of \( A \) have been moved to the merged area, and none from \( B \). Galloping starts out with the lists \( A' = [9,10\ldots100] \) and \( B = [50,51\ldots150] \). Galloping mode consists of comparing \( B[0] \) to \( A'[2^j - 1] \), \( j \) starting on 0 and increasing until the interval in which \( B[0] \) should sit in \( A' \) is found. Binary search uses this interval to find the exact position. All elements before that position in \( A' \) can be moved to the merged area, in this case the elements \( [9,10\ldots50] \). Galloping continues with still the same list \( B \), and \( A'' = [51,52\ldots10] \). The same process is now done for \( A''[0] \) and \( B[2^j - 1] \). We continue until both searches find slices less than \texttt{min_gallop} to move to the merged area. In this case, this would happen after two more searches, since we could only remove one element from \( B \) followed by only one element from \( A'' \). Merging then goes back to regular mode.

The variable \texttt{min_gallop} is set to 8 because it has been found to be the most effective threshold[9, Galloping with a broken leg]. If it is too small, it is less likely that the pattern will continue. In that case it might not be worth going into galloping mode. However if it is too high, we waste time on comparisons that could have been skipped.

Once all runs have been found, the runs remaining on the stack are merged, resulting in the sorted list.
4 Related work

Here we mention a few works related to this project, giving examples of some different tools and approaches to formal verification.

We have already mentioned the work of de Gouw et al. [11], who discovered a bug in the Java Timsort implementation. Before this, Timsort had only three stack invariants. They turned out to not always hold, which meant it was possible for the stack to grow larger than expected. This could cause an out-of-bounds exception. The researchers proposed a fourth invariant (invariant 3 in section 3.3) and verified the solution. Verification was done with the formal verification tool KeY. The Python implementation has since been changed to have the fourth invariant [23]. For Java, the stack-size was increased instead. This is a less efficient solution, as de Gouw notes in a follow-up blog post [24].

Stack-properties of Timsort have also been verified with the interactive theorem prover Isabelle/HOL by Zhang, Zhao, and Sanan [25]. Their goal was to provide a C implementation with verified stack-properties. It was achieved by defining the algorithm in an imperative language, compatible with Isabelle/HOL, called Simpl. Once verified, the Simpl definitions were manually translated to C code.

Auger, Nicaud and Pivoteau have proven several time-complexity and stack-properties of Timsort [22] with pen-and-paper proofs. They have additionally proposed simplified versions of the algorithm that we will make use of in the process of implementing Timsort.

In his bachelor thesis [26], Marco Pierre Fernandez Burgos formalized and verified several simple sorting algorithms with both pen-and-paper proofs and Isabelle/HOL. The thesis is a good introduction to what formal verification of sorting algorithms entail. We have taken much inspiration from it regarding how to display formal proofs in a report.

5 Formalizing the specification

In order to formally verify a function, its specification must be formalized. The specification in this case consists of the two conditions placed on sorting algorithms:

1. The output is a permutation of the input
2. The elements of the output appear in non-decreasing order according to some given total order

We also want to guarantee that the function always terminates, but this is implicit in HOL - we are not allowed to define functions that HOL cannot prove terminates. For simple definitions HOL does this on its own. For others, we
have to provide guidance in a termination proof that it sets up for us.

In the following sections, we formalize the specification in HOL with the help of existing theorems from the libraries listTheory, relationTheory, and sortingTheory. The name `sort_fun` denotes our sorting algorithm. It takes a relation `R` with which to sort the elements, and a list `l` to sort. The binary relation `R` is a prefix operation in HOL, meaning that `R x y` expresses what we have previously expressed with the mathematical infix notation `xRy`.

### 5.1 Permutation

The permutation condition is straightforward - the output should consist of the same elements as the input. There is a definition of this property in the library sortingTheory, under the name `PERM_DEF`:

\[
\forall L1 \, L2. \text{PERM } l1 \, l2 \iff \forall x. \text{FILTER } (\neq x) \, L1 = \text{FILTER } (\neq x) \, L2
\]

This means that for two lists to be permutations of each other, filtering them for any element should yield the same result. For example, the lists `[1,2,1]` and `[1,1,2]` both return `[1,1]` when filtering for `1`, `[2]` when filtering for `2`, and `[]` when filtering for any other element. Thus they are permutations of each other. The permutation property can therefore be expressed as

\[
\forall R \, l. \text{PERM } l \, (\text{sort}_R \, l)
\]

### 5.2 Sortedness

For the second condition, we must formally describe a total order. Fortunately HOL has all its properties defined in the relationTheory library (although the connex property is somewhat confusingly referred to as “total”).

\[
\forall R. \text{antisymmetric } R \iff \forall x \, y. \ R x \land R y \implies x = y
\]

\[
\forall R. \text{transitive } R \iff \forall x \, y \, z. \ R x \land R y \land R z \implies R x \land R z
\]

\[
\forall R. \text{total } R \iff \forall x \, y. \ R x \lor R y
\]

With this in mind, we can use another definition in the sortingTheory library, `SORTED_DEF`:

\[
(\forall R. \text{SORTED } R \, [] \iff T) \land
(\forall x. \text{SORTED } R \, [x] \iff T) \land
\forall y \, x \, rst. \text{SORTED } R \, (x::y::rst) \iff R x \land \text{SORTED } (y::rst)
\]

This only guarantees that each element is greater than the one before it, not that the whole list appears in non-decreasing order. However, if the relation is transitive it follows automatically, as can be seen by the theorem `SORTED_EL_LESS`.
∀R. transitive R \iff ∀ls. \text{SORTED } R \ l s \\
∀m n. \ m < n \land n < \text{LENGTH } l s \Rightarrow R (\text{EL} \ m \ l s) (\text{EL} \ n \ l s)

where \text{EL} \ m \ l s \ finds \ the \ element \ at \ index \ m \ in \ l s. \ The \ bottom \ line \ expresses \ that \ a \ list \ is \ in \ non-decreasing \ order. \ For \ any \ two \ elements, \ the \ one \ with \ the \ lower \ index \ is \ lesser \ than \ the \ other. \ Given \ that \ the \ relation \ R \ is \ transitive, \ this \ is \ equivalent \ to \ being \ sorted \ according \ to \ \text{SORTED} \_\text{DEF}. \ Therefore \ assuming \ transitivity \ and \ proving \ sortedness \ according \ to \ \text{SORTED} \_\text{DEF} \ proves \ that \ the \ list \ is \ in \ non-decreasing \ order.

We also have to assume that the relation is connex. Otherwise there would be cases where there was no way of sorting the list. Consider for example the non-connex relation < and the list [1,1]. There is no way of sorting this list without removing elements, which would of course violate the permutation condition.

Antisymmetry does not have to be assumed, however. Leaving it out has no effect on our ability to place elements in non-decreasing order. So why is it a requirement? It seems to be in order to guarantee that there will be one sorting of any given list. Say we have a relation that puts strings in alphabetical order, but only looks at the first letter. Given two strings x and y beginning with the same letter, both R x y and R y x hold, but we cannot conclude that x = y, meaning that relation is not antisymmetric. Using the relation to sort a list of strings, strings beginning with the same letter could be placed in any order. While this is a valid sorting, it seems to go against one of the main purposes of sorting - to facilitate searching. Imagine having to find a specific string in a large list sorted in this way. The chunk of strings beginning with the same letter would have to be searched through linearly. Still, it is not needed for the formal proof. The theorem we must prove is thus

∀R l. \ transitive \ R \land \text{total} \ R \Rightarrow \text{SORTED} \ (\text{sort} \_\text{fun} \ R \ l)

6 Verifying stacksort

The algorithm we call stacksort is a simplified version of Timsort found in the report by Auger et al as Algorithm 5. In this algorithm, there is no minimum length for runs, and thus binary insertion sort is not needed to extend them. The same invariants dictating when to merge runs apply, but they are merged through regular merging (i.e there is no galloping mode).

The process of verifying stacksort involves defining help-functions and proving properties about these that will be helpful for the final proofs. In the following sections, a select few definitions and proofs will be shown. With these we intend to give a comprehensive overview of the work that has been done. The first
section looks at the function get\_run and how properties of it can be proven with simple induction. These proofs are shown in detail. The second section looks at decompose, a function that requires proof of termination and more complicated induction. In the third we show how we struggled with HOL when working with the function merge\_collapse. We will make use of several already existing functions such as LENGTH and DROP. Some of them are obvious in what the do, others will be given short descriptions.

At the end the final function and its proofs will be shown and briefly explained.

### 6.1 Proving properties of get\_run

We acquire the first non-decreasing run of a list with the following definition

**Definition get\_run\_def:**

\[
\text{get}\_\text{run} \ R \ [\ ] = [\ ] \land \\
\text{get}\_\text{run} \ R \ [x] = [x] \land \\
\text{get}\_\text{run} \ R \ (x::y::xs) = \\
\quad \text{if } R \ x \ y \ \text{then } (x::(\text{get}\_\text{run} \ R \ (y::xs))) \ \text{else } [x]
\]

End

In the recursive call the length of the list shrinks by exactly one. Continuous recursion will therefore eventually lead to a length of 1, at which point the function terminates by the second base-case. Thus the function always terminates, and this is simple enough for HOL to prove automatically. Now we want to prove that the run found is indeed sorted, namely

\[\forall R \ l. \ \text{SORTED} \ (\text{get}\_\text{run} \ R \ l)\]

We prove this by induction on \(l\). This process will be shown in detail - aside from some very basic arithmetic - even though HOL provides tools to shorten the process somewhat.

**Base-case 1:** \(l = [\ ]\)

\[
\text{SORTED} \ R \ (\text{get}\_\text{run} \ R \ [\ ]) \quad \text{rewrite with } \text{get}\_\text{run}\_\text{def} \\
= \text{SORTED} \ R \ [\ ] \quad \text{true by } \text{SORTED}_\text{DEF}
\]

**Base-case 2:** \(l = [x]\)

\[
\text{SORTED} \ R \ (\text{get}\_\text{run} \ R \ [x]) \quad \text{rewrite with } \text{get}\_\text{run}\_\text{def} \\
= \text{SORTED} \ R \ [x] \quad \text{true by } \text{SORTED}_\text{DEF}
\]

**Inductive step:** We assume the induction hypothesis

\[
\text{SORTED} \ R \ (\text{get}\_\text{run} \ R \ (y::xs)) \ \text{and want to prove} \\
\text{SORTED} \ R \ (\text{get}\_\text{run} \ R \ (x::y::xs))
\]

We begin by case-splitting on \(xs\).
\textbf{Case} $\text{xs} = []$

\[
\text{SORTED } R \ (\text{get}\_\text{run} \ R \ (x::y::[])) \quad \text{rewrite with } \text{get}\_\text{run}\_\text{def}
\]

\[
= \text{SORTED } R \ (\text{if } R \ x \ y \ \text{then } [x;y] \ \text{else } [x])
\]

And now case-split on $R \ x \ y$

\textbf{Case} $R \ x \ y$

\[
\text{SORTED } R \ (\text{if } R \ x \ y \ \text{then } [x;y] \ \text{else } [x]) \quad \text{rewrite with } R \ x \ y
\]

\[
= \text{SORTED } R \ [x;y] \quad \text{true by } R \ x \ y \ \text{and } \text{SORTED}\_\text{DEF}
\]

\textbf{Case} $\neg R \ x \ y$

\[
\text{SORTED } R \ (\text{if } R \ x \ y \ \text{then } [x;y] \ \text{else } [x]) \quad \text{rewrite with } \neg R \ x \ y
\]

\[
= \text{SORTED } R \ [x] \quad \text{true by } \text{SORTED}\_\text{DEF}
\]

Thus the case of $\text{xs} = []$ is proven. On to the other.

\textbf{Case} $\text{xs} = h::t$

\[
\text{SORTED } R \ (\text{get}\_\text{run} \ R \ (x::y::h::t)) \quad \text{rewrite with } \text{get}\_\text{run}\_\text{def}
\]

\[
= \text{SORTED } R \ (\text{if } R \ x \ y \ \text{then } x::\text{get}\_\text{run} \ R \ y::h::l \ \text{else } [x])
\]

This too must be split on $R \ x \ y$.

\textbf{Case} $R \ x \ y$

\[
\text{SORTED } R \ (\text{if } R \ x \ y \ \text{then } (x::\text{get}\_\text{run} \ R \ y::h::l) \ \text{else } [x])
\]

\[
= \text{SORTED } (R \ (x::\text{get}\_\text{run} \ y::h::t)) \quad \text{rewrite with } \text{SORTED}\_\text{DEF}
\]

\[
= \text{SORTED } R \ x \ y \ \land \ \text{SORTED } R \ (\text{get}\_\text{run} \ R \ y::h::l) \quad \text{true by } R \ x \ y \ \text{and the induction hypothesis}
\]

\textbf{Case} $\neg R \ x \ y$

\[
\text{SORTED } R \ (\text{if } R \ x \ y \ \text{then } (x::\text{get}\_\text{run} \ R \ y::h::l) \ \text{else } [x])
\]

\[
= \text{SORTED } R \ [x] \quad \text{rewrite by } \neg R \ x \ y \quad \text{true by } \text{SORTED}\_\text{DEF}
\]

Thus the case of $\text{xs} = h::t$ is proven as well, proving the entire goal.

Now we want to prove some property that will help with permutation proofs for subsequent definitions. By looking at the structure of the definition that will use \text{get}\_\text{run}, this property was determined to be

\[
\forall R \ l. \ \text{get}\_\text{run} \ R \ l \ ++ \ (\text{DROP} \ (\text{LENGTH} \ (\text{get}\_\text{run} \ R \ l)) \ l) \ = \ l
\]

Meaning that the run found by \text{get}\_\text{run} appended to the rest of the list equals the original list. Again induction on $l$ is used. The base cases are trivial enough to be omitted this time. We only work the left-hand side of the equality.
**Inductive step:** We assume the induction hypothesis
\[
\text{run}_R \cdot (y::xs) ++
(DROP \cdot (\text{LENGTH} \cdot (\text{run}_R \cdot (y::xs))) \cdot (y::xs))
= (y::xs)
\]

And want to prove
\[
\text{run}_R \cdot (x::y::xs) ++
(DROP \cdot (\text{LENGTH} \cdot (\text{run}_R \cdot (x::y::xs))) \cdot (x::y::xs))
= (x::y::xs)
\]

Again we want to transform the left-hand-side to correspond with the right-hand-side. We begin by case splitting on \( R \cdot x \cdot y \).

**Case** \( R \cdot x \cdot y \)
\[
\text{run}_R \cdot (x::y::xs) ++
(DROP \cdot (\text{LENGTH} \cdot (\text{run}_R \cdot (x::y::xs))) \cdot (x::y::xs))
\]
rewrite with \( \text{run}_R \) def and \( R \cdot x \cdot y \)
\[
= x::\text{run}_R \cdot (y::xs) ++
(DROP \cdot (\text{LENGTH} \cdot (\text{run}_R \cdot (y::xs))) \cdot (x::y::xs))
\]
rewrite with \( \text{LENGTH} \)
\[
= x::\text{run}_R \cdot (y::xs) ++
(DROP \cdot (1 + (\text{LENGTH} \cdot (\text{run}_R \cdot (y::xs)))) \cdot (x::y::xs))
\]
rewrite with \( \text{DROP} \) def
\[
= x::\text{run}_R \cdot (y::xs) ++
(DROP \cdot (\text{LENGTH} \cdot (\text{run}_R \cdot (y::xs))) \cdot (y::xs))
\]
rewrite with induction hypothesis
\[
= (x::y::ys)
\]
equal to 1

**Case** \( \neg R \cdot x \cdot y \)
\[
\text{run}_R \cdot (x::y::xs) ++
(DROP \cdot (\text{LENGTH} \cdot (\text{run}_R \cdot (x::y::xs))) \cdot (x::y::xs))
\]
rewrite with \( \text{run}_R \) def and \( \neg R \cdot x \cdot y \)
\[
= [x] ++ \text{DROP} \cdot (\text{LENGTH} \cdot [x]) \cdot (x::y::xs)
\]
rewrite with \( \text{LENGTH} \)
\[
= [x] ++ \text{DROP} \cdot 1 \cdot (x::y::xs)
\]
rewrite with \( \text{DROP} \) def
\[
= [x] ++ (y::xs) = (x::y::xs)
\]
equals 1

Thus the goal is proved.
6.2 Proving properties of decompose

Now we look at the function that breaks down a list into all its runs.

Definition decompose_def:
  decompose R [] = [] ∧
  decompose R list =
  let run = (get_run_s R list) in
  (run::(decompose R (DROP (LENGTH run) list)))

Termination
  WF_REL_TAC ‘measure (\(R, l). LENGTH l)’ >>
  rewrite_tac[listTheory.LENGTH_DROP] >>
  simp[RUN_S_LENGTH]

End

Let us first describe some details about get_run_s. The function returns the first run of a list. It assumes that the relation provided is connex - if R x y is not true, it assumes that R y x is. Because of that, connexity is a prerequisite to prove that the run found is sorted. Properties we have proven about get_run_s that we will subsequently use are

RUN_S_LENGTH: ⊢ ∀ R l. l ≠ [] ⇒ 0 < LENGTH (get_run_s R l)

RUN_S_SORTED: ⊢ ∀ R l. total R ⇒ SORTED (get_run_s R l)

Back to decompose. For this definition it was necessary to provide a proof of termination. In the first line of the termination proof, we specify that it is the length of the second argument that will shrink with each recursive call. We must then prove that is really does shrink - that the list in the recursive call is strictly smaller than the original. Otherwise, the definition could loop indefinitely. So, the goal is

∀v3 v2 R. LENGTH (DROP (LENGTH (get_run_s R (v2::v3))) (v2::v3)) < LENGTH (v2::v3)

Rewriting this with an existing theorem called LENGTH_DROP, we get

∀v3 v2 R. LENGTH (v2::v3) - LENGTH (get_run_s R (v2::v3)) < LENGTH (v2::v3)

This can be solved with the knowledge that LENGTH (get_run_s R (v2::v3)) will be greater than 0, and some arithmetic. Giving our theorem RUN_S_LENGTH to the powerful simplification tool simp[] does this for us. Thus decompose is proven to terminate for all input.
Moving on, we would like to prove that each list in the list that `decompose` returns is sorted. Since we intend on using the theorem `RUN_S_SORTED` to help with this, we have to assume that `R` is connex.

\[
\forall R \; \text{total } R \implies \text{EVERY (SORTED } R) (\text{decompose } R \; l)
\]

Now, because `l` is the shrinking element in the recursive call, one might feel compelled to prove this by induction on `l`. The base case would be true by definition of `decompose`, `SORTED`, and `EVERY`. Then we would assume the induction hypothesis

\[
\forall R \; l. \; \text{total } R \implies \text{EVERY (SORTED } R) (\text{decompose } R \; l)
\]

and attempt to prove

\[
\forall h \; R \; l. \; \text{total } R \implies \text{EVERY (SORTED } R) (\text{decompose } R \; (h::l))
\]

But because the list does not shrink one element at a time, but rather an unknown length of whatever run is found, we get stuck here. Some other strategy must be used. One alternative is to induct on `decompose R l` itself. Again, the base case is simple enough to be omitted. The induction hypothesis in this case would be

\[
\forall R \; l. \; v = \text{decompose } R \; l \implies \text{total } R \implies \text{EVERY (SORTED } R) (\text{decompose } R \; l)
\]

and we would want to prove

\[
\forall h \; R \; l. \; h::v = \text{decompose } R \; l \implies \text{total } R \implies \text{EVERY (SORTED } R) (\text{decompose } R \; l)
\]

This can now be solved with a case analysis on `l`. If `l` is an empty list, then `decompose R l` could by definition not result in `(h::v)`, rendering the whole implication true. If `l` is non-empty, we can use the definition of `decompose` to see that

\[
h = \text{get_run_s } R \; l
\]

\[
v = \text{decompose } R \; (\text{DROP (LENGTH (get_run_s } R \; l)) \; l)
\]

According to our theorem `RUN_S_SORTED`, `h` is sorted. According to the induction hypothesis, every list in `v` is sorted. Using the definition of `EVERY`

\[
\forall P \; h \; t. \; \text{EVERY } P \; (h::t) \iff P \; h \land \text{EVERY } P \; t
\]

this proves that every list in `(h::v)` is sorted.

However, there is an even simpler way to prove this. For functions with non-trivial termination HOL creates and saves an induction theorem for it, under the name `functionname_ind`. This has been done for `decompose`, namely
\[ \forall P . \\
(\forall R. P \ R \ []). \land \\
(\forall R \ v2 \ v3. \\
\forall run.
\text{run} = \text{get-run-s} \ R \ (v2::v3) \Rightarrow \\
P \ R \ (\text{DROP} \ (\text{LENGTH} \ \text{run}) \ (v2::v3))) \Rightarrow \\
P \ R \ (v2::v3)) \Rightarrow \\
\forall v \ v1. \ P \ v \ v1 \]

Invoking this theorem is similar to inducting on \text{decompose} \ R \ 1, but allows us to skip the case analysis.

### 6.3 Proving properties of merge\_collapse

The function \text{merge\_collapse} takes a stack and merges the runs on it until the invariants hold. The function \text{merge} from the mergsortTheory library is used for merging. \text{merge\_collapse} has five pattern-matched cases, from an empty list up to a list of at least four element. Only the last case will be shown here. Similar or simpler logic is found in the other cases.

**Definition merge\_collapse_def:**

\[
\ldots \\
\text{merge\_collapse} \ R \ (z::y::x::w::rst) = \\
(\text{let}
\ p1 = (\text{LENGTH} \ x < \text{LENGTH} \ z);
\ p2 = (\text{LENGTH} \ x \leq (\text{LENGTH} \ y + \text{LENGTH} \ z));
\ p3 = (\text{LENGTH} \ w \leq (\text{LENGTH} \ x + \text{LENGTH} \ y));
\ p4 = (\text{LENGTH} \ y \leq \text{LENGTH} \ z)
\ \text{in}
\ (\text{if} \ p1 \ \text{then} \ \text{merge\_collapse} \ R \ (z::(\text{merge} \ R \ y \ x)::w::rst)
\ \text{else} \ \text{if} \ (p2 \ \lor \ p3 \ \lor \ p4) \ \text{then}
\ \text{merge\_collapse} \ R \ ((\text{merge} \ R \ z \ y)::x::w::rst)
\ \text{else} \ (z::y::x::w::rst))
\]

**Termination**

\text{WF\_REL\_TAC} ‘\text{measure (LENGTH o SND)}’ >> \\
\text{srw_tac[]}[]

**End**

This definition required a termination proof as well. It is enough to specify that the length of the second argument - the stack - shrinks with each recursive call. The stack in either of the recursive cases is clearly one element shorter than the original input.

The particularly challenging part about \text{merge\_collapse} is the permutation proof. We want to prove that the elements on the stack stays the same after having been collapsed. \text{FLAT} is a definition that concatenates all lists in a list of
lists. For example, $\text{FLAT } [[[1,2,3],[4,5,6]]] = [1,2,3,4,5,6]$. We use this for our proof

$$\forall R \ l. \ \text{PERM} (\text{FLAT } l) (\text{FLAT} (\text{merge\_collapse } R \ l))$$

Now, the first problem is that $\text{merge\_collapse}$ has both many cases and several variables - $p_1, p_2, p_3, p_4$. Powerful rewriting tools - that have been helpful for earlier proofs - would break the goal down into every possible combination of these cases and variables. They could also try to go several recursions deep. If not careful, this could result in over 60 different sub-goals, which would be tedious and unnecessary to deal with. We are not actually interested in inspecting all these different cases individually. Whether $p_1$ is true or false does not matter - all we want is to divide the goal into the different if-else-clauses. Achieving this required us to find more gentle and specific tactics. Once that is done, we run into the second problem.

Manipulating permutations with several components is difficult. The proof from mergesortTheory that merging two lists results in a permutation of those lists, $\text{MERGE\_PERM}$, looks like

$$\forall R \ l_1 \ l_2. \ \text{PERM} (l_1 ++ l_2) (\text{merge } R \ l_1 \ l_2)$$

To a person, it then seems obvious that something like

$$\forall R \ w \ x \ y \ z. \ \text{PERM} (w ++ x ++ y ++ z) (w ++ (\text{merge } R \ x \ y) ++ z)$$

would be true as well. Explaining it to a computer is a different story. The expression must be rewritten to match some preexisting theorem regarding permutations that can then solve it. But rewriting when dealing with several components is a challenge in itself. Accurately specifying which part of the expression to rewrite, in what way, and in a way that does not affect some other part of the expression that we want to remain the same is tedious and results in messy proofs\footnote{Likely there are elegant ways to do it, but these are beyond the author’s capabilities.}. A simpler method is to identify the theorems that would be able to solve the goal and pass these to METIS_TAC\footnote{Likely there are elegant ways to do it, but these are beyond the author’s capabilities.}. This rewrites the goal in every way possible according to the provided theorems until a solution is found (or no solution is found, in which case it fails). The search space grows with each added theorem. To keep it as small as possible, we can separately prove several different small lemmas that require fewer theorems. These can then in turn be used in the final proof.
6.4 Verifying stacksort

Finally, we can define the main function

Definition stacksort_def:
  stacksort R l = force_feed (feed_collapse R [] (decompose R l))
End

feed_collapse takes the runs found by decompose and “feeds” them to the stack, initially empty, one at a time. After each added list, it runs the stack through merge_collapse. force_feed then merges what is left on the stack, resulting in the final list. Having proven appropriate sorting and permutation properties of the functions on the right-hand side, proving sorting and permutation for stacksort is simply a matter of rewriting. The permutation proof is facilitated with METIS_TAC[], much like the permutation proof for merge_collapse was. The final proofs are as follows:

\[ \forall R. \text{PERM } l \rightarrow (\text{stacksort } R \ l) \]

\[ \forall R. \text{transitive } R \land \text{total } R \implies \text{SORTED } R \rightarrow (\text{stacksort } R \ l) \]

This completes the verification of stacksort.

7 Runtime analysis of stacksort

Let us now turn to the time-complexity of stacksort. Since the algorithm spends most of its time comparing elements, the number of comparisons is what will be measured. We would like to prove how this number grows as the length of the list increases in a best- and worst-case scenario. The big-O notation will be used to express this. To acquire the necessary data to work with, we define new functions that count the number of comparisons being made. For every function defined for stacksort, a “counting” version is defined. For example, here is the get_run equivalent:

Definition get_run_t_def:
  get_run_t n R [] = (n, []) \land
  get_run_t n R [x] = (n, [x]) \land
  get_run_t n R (x::y::xs) =
  if R x y then
  let
    (num, run) = get_run_t (n + 1) R (y::xs)
  in
    (num, x::run)
  else (n + 1, [x])
End
They are all named as `original_name_t`. Aside from the additional data, these functions should be equivalent to the originals. This is confirmed by proving

\[ \forall n \ R \ l. \ original \ R \ l = \text{SND} \ (\text{counting\_equivalent} \ n \ R \ l) \]

for all of them. `SND` extracts the second element of a tuple. These proofs are mostly a matter of induction and rewriting with the respective function definitions and `SND`.

### 7.1 Formalizing time-complexity

Next, we must formalize the big-\(O\) notation. The definition of big-\(O\) is that "for some natural number \(n_0\) and some strictly positive real number \(c\), \(g(n) \leq cf(n)\) whenever \(n \geq n_0\)[8, p. 58], where \(g\) is our function and \(n\) is the size of the input. We can define this formally as

\[ \exists n_0 \ c. \forall n. \ n_0 \leq n \iff g(n) \leq c \cdot (f(n)) \]

In the case of `stacksort`, \(n\) is the length of the list.

### 7.2 Best-case

Given an already sorted list, `stacksort` requires exactly \(\text{LENGTH} \ l - 1\) comparisons to finish. These are the comparisons required for `get_run` to find the entire list as a run. Note that this is the case even for empty lists, as subtraction in HOL is only defined on natural numbers and \(\text{LENGTH} \ [] - 1 = 0\). As this results in one or zero lists on the stack, nothing needs to be merged, and thus no other comparisons are done before returning. This means that the algorithm runs in \(O(n)\) time. Since it holds for all list lengths, the \(n \geq n_0\) requirement can be left out entirely. Therefore we want to prove

\[ \exists c. \forall R \ l. \ \text{SORTED} \ R \ l \implies \text{FST} \ (\text{stacksort} \ 0 \ R \ l) \leq c \cdot \text{LENGTH} \ l \]

where `FST` extracts the first element of a tuple. To prove this, we can begin by proving the relevant properties about `get_run_t` and propagate these properties to `decompose_t`. Although the functions are intended to be used with a starting value of \(n = 0\), the properties are proven for any \(n\). This is to enable using the induction hypotheses created by HOL. Invoking these would often instantiate the different cases automatically, and could prove trivial base-cases directly, resulting in shorter and cleaner proofs. The properties proven of `decompose_t` are `DECOMPOSE_BEST_n`:

\[ \forall n \ R \ l. \ \text{SORTED} \ R \ l \implies \text{FST} \ (\text{decompose} \ t \ n \ R \ l) = n + (\text{LENGTH} \ l - 1) \]

`DECOMPOSE_BEST_LIST`:

\[ \forall n \ R \ l. \ l \neq [] \land \text{SORTED} \ R \ l \implies \text{SND} \ (\text{decompose} \ t \ n \ R \ l) = [1] \]

`decompose_t` returning an empty stack when given an empty list is simple enough to be handled in the main proof.
With the theorems just mentioned, the main proof is mostly a matter of rewriting and simplifying. We can specify $c$ as 1 and then split the goal into different cases. For the base case of $l = \[]$, we have to prove that at most $1 \times \text{LENGTH} \[]$ (that is, 0) comparisons are made. This is done by simply rewriting with the function definitions until we have proven that no comparisons are made at all. The case when $l$ is not empty is similar, but also uses $\text{DECOMPOSE\_BEST\_n}$ and $\text{DECOMPOSE\_BEST\_LIST}$. Finally we arrive at

$$\forall R. \text{SORTED} (h::t) \implies \text{LENGTH} t \leq \text{SUC} (\text{LENGTH} t)$$

Some further arithmetic simplification proves this. Thus the best-case runtime of $\text{stacksort}$ is proven to be in $\mathcal{O}(n)$ time.

### 7.3 Worst-case

Decomposing a list into runs requires $\text{LENGTH} n - 1$ comparisons regardless of what the list looks like, as each comparison “picks” one element off. If it is decomposed into several runs though, they will have to be merged at some point. The worst-case scenario for merging two lists with $\text{merge}$ is when the last elements of both lists have to be compared. As a counter-example, take the lists $A = [1,2]$ and $B = [3..100]$. Once all of $A$ has been compared to the first element in $B$, the merging is done. This took two comparisons. Now we change $A$ to $[1,101]$. After moving 1 to the merged area, 101 must be compared to all elements in $B$. This is a total of 99 comparisons. More abstractly, it is $\text{LENGTH} A + \text{LENGTH} B - 1$ comparisons. It is still in $\mathcal{O}(n)$ time.

The number of runs on the stack at any given time is upper-bounded by $\log n$\textsuperscript{22}. This is a result of the stack invariants. So, if merging is done in $\mathcal{O}(n)$ time, and the number of merges is upper-bounded by $\log n$, then collapsing or merging the entire stack is done in $\mathcal{O}(n \log n)$ time. Proving the worst-case time-complexity of $\text{stacksort}$ would therefore be to prove

$$\exists n \ c. \forall l. \ n \leq \text{LENGTH} l \implies \text{SND} (\text{stacksort}_t \ R \ l) \leq c \times (\text{LENGTH} l) \times \log (\text{LENGTH} l)$$

where $\log$ is shorthand for the actual way of expressing logarithms in HOL. However, despite having a detailed pen-and-paper proof of a similar case provided by Auger et al, we were unsuccessful in proving this. The difficulty lied in formulating a proof regarding the stack size. How do we formulate a proof that expresses what the arguments look like during execution, and not just what the result is? More specifically during all steps of the execution? It requires a different approach compared to our earlier proofs. Things were further complicated by how logarithms are defined in HOL. They do not evaluate like how for example $1 + 1$ evaluates to 2. Instead, $\text{LOG} 1 1$ simply evaluates to $\text{LOG} 1 1$. This made it more difficult to experiment when trying to formulate a proof.
8 Code generation

In order to use our now verified sorting algorithm, code must be generated from the definitions that can then be compiled. This process can cause problems. How do we know that the properties proven will be retained in the executable machine code? If they are not, then the executable is not verified. The solution to this problem is to use code generation and compilation that have themselves been verified.

As previously mentioned, HOL was chosen specifically because of its compatibility with CakeML. CakeML is an ML system developed and verified with HOL4. It takes what it calls a “binary extraction approach” when it comes to generating executable code. The idea is to stay inside the prover while generating binary code for execution. This is done using the CakeML compiler, defined and verified with HOL4. The compiler can be evaluated inside the prover with the definition to be compiled as input. The resulting binary code is then written to somewhere outside of the prover, and can then be used for execution. In other words, CakeML offers an end-to-end verified compilation method. However, the mechanisms for this are not fully automatic. Some work is required to fit our stacksort definition with the CakeML framework. Because of time constraints, we opted for a different code generation tool.

EmitML is a HOL module that translates HOL definitions to ML code. All definitions that stacksort uses, as well as stacksort itself, must be translated in this way. The translation does not guarantee retention of properties. Compiling the code requires non-verified compilers like MoscowML, and thus errors could potentially be introduced in the compilation process. Even though EmitML is an obviously less ideal tool in these respects, it was chosen because it did not require us to learn a new system. Learning how to use EmitML did take some time though - roughly a full day of work - since documentation for the module is sparse. Helpful examples can be found in the examples HOL directory. The resulting ML code is a direct translation to ML syntax and looks much like what a human could have written. For example, here is the resulting decompose function. The HOL equivalent is found in section 6.2.

```ml
fun decompose R [] = []
  | decompose R (v2::v3) = 
    let val run = get
    in
      run::decompose R (rich_listML.DROP (LENGTH run) (v2::v3))
    end
```

A final issue arose when trying to call stacksort because of differing type signatures. The relation that stacksort takes has the signature `a -> a -> bool`. The relation `<=` in ML has the signature `(int * int) -> bool`. To use
the ML relation, we can simply re-define it as

\[
\text{fun new_leq a b = a <= b}
\]

9 Discussion

How does \texttt{stacksort} compare to Timsort? They have the same best- and worst-case time-complexities, and they behave identically in some cases. Lists already sorted or in strictly decreasing order are obvious examples. Cases where all runs are longer than \texttt{min_run} and where galloping mode is never entered during merging will be identical as well.

I went into this project with no experience with HOL or other ITPs, and very limited with Standard ML. A significant portion of this project was spent learning how to work with HOL. While I would not say that it is a difficult tool in itself, the sparse documentation and small community sometimes made it very difficult to solve even trivial problems. I am still very much a novice - only a handful of the hundreds of tactics that HOL provides have been used in this project. I can see why testing is still the preferred method when it comes to checking software for quality, given the comparatively vast amount of documentation and tutorials you then often have access too. It seems to me like part of the reason why formal verification is still a niche community is \textit{because} it is a niche community.

On the other hand, HOL has undeniable strengths. We could see in this report that proving properties of even simple definitions can result in several case splits, all with different assumptions to keep track of. With HOL, this is a non-issue. For more complicated proofs this quality could be invaluable. The induction hypotheses and various rewriting tools were also very helpful. Especially when doing more complicated proofs, it could feel like HOL was the one guiding me, by repeatedly giving me manageable sub-goals to prove until the original proof was done.

9.1 Conclusion and future work

In this project, we have shown how the properties required of sorting algorithms can be formalized, and demonstrated techniques for how to prove these for a simplified version of Timsort, \texttt{stacksort}. It behaves like Timsort in some cases, and has the same best- and worst-case time complexities. We have also formalized the big-O notation and shown how one can go about proving such properties. The best-case time-complexity for the \texttt{stacksort} was proven. Due to lack of time and experience with HOL, we did not succeed in verifying the more complicated worst-case. We have demonstrated how to generate executable code from the verified definitions with the EmitML module in HOL, although the generated code is not guaranteed to retain the proven properties.
Further work would be to continue with the goals of this project by proving the worst-case time-complexity and generating code from the definitions with CakeML. Other additions could be to look at average-case time-complexity, which is perhaps the most interesting case, and to prove space-complexity properties. Proving space-complexity would require figuring out how to measure the space used by the algorithm. It could potentially be done in a similar way to how we measured time in this report.

Parts of this project could also be used to verify the actual Timsort. The properties to prove would of course be the same, and parts of the definitions used to define stacksort could be re-used for Timsort.

References


