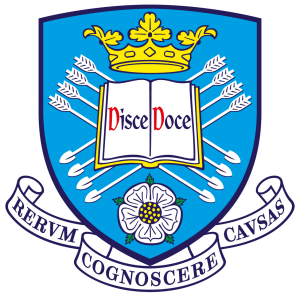


The University Of Sheffield.

# Verification and Correctness of Functional Programs in Isabelle/HOL

Daniel Marshall

Prof. Georg Struth

COM3500 Individual Research Project

1st May 2019

This report is submitted in partial fulfilment of the requirement for the degree of MComp Computer Science with Mathematics by Daniel Marshall.

## Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Daniel Marshall

# Abstract

Formally verifying software by using mathematical methods to prove that they give the correct result is one of the most reliable ways to ensure that a system is safe and dependable, but specifying the behaviour of a program in enough detail for this is often difficult and impractical.

The aim of this project is to use the interactive theorem prover Isabelle/HOL and apply it to verifying the correctness of functional programs, which have a more mathematical structure than imperative programs and so lend themselves well to being analysed in this way.

We begin by proving statements about data structures like natural numbers, lists, and trees, move on to verifying the correctness of various programs such as the Euclidean algorithm and sorting algorithms like insertion sort and merge sort, and finally implement an inductive definition of the shuffle product operation on strings and languages, providing proofs of some known properties.

## Acknowledgements

I would like to express my gratitude to:

- My supervisor, Prof. Georg Struth, for his help and guidance throughout the project, and for inspiring an interest in the field of functional program verification.

- All those who have contributed to the development of Isabelle, as without its powerful tools and thorough documentation this project would not have been possible.

- My family and friends, for their support throughout the time spent working on this project and over the course of studying for this degree.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background

Designing high quality pieces of software that are also reliable is one of the biggest challenges in programming, as the more complex the software becomes, the more difficult it is to be aware of everything that could go wrong. One possible solution to this is formal verification; this is where the algorithms underlying the program are proved to be correct using mathematical methods, so that we can be definitively sure they give the desired result. This is especially useful for systems where safety is critical and any failure of the software could be disastrous. The creators of the formally verified seL4 microkernel, used in fields like aerospace engineering and Internet of Things platforms, state that "it is now feasible and cost-effective to demand strong, code/machine-level guarantees as table stakes for highly critical systems". [1]

However, formal verification of real-world software systems is often very difficult and can be impractical, as the intended behaviour of the program has to be specified in full detail and many hidden assumptions need to be made explicit in order for a proof to be created. This problem could be alleviated via the paradigm of functional programming; as opposed to imperative programs, which come in the form of a set of instructions carried out by the compiler/interpreter, functional programs are made up of functions whose results depend purely upon the arguments that are passed to them, meaning that they have a more mathematical structure that could be

easier to prove things about and there are less potential side effects to consider.

In addition to this, as the software increases in complexity the proof will likewise become more complex, which creates the potential for mistakes to be introduced when the proof is being done by hand that could well invalidate the verification process if they went unnoticed. One way to avoid this could be to use an interactive theorem prover, which is a software tool to help with the creation of formal proofs allowing the user to guide the computer in storing details of the proof and even automatically generating necessary steps. These tools have already seen wide use in more abstract fields like pure mathematics; in 2005, a formal proof of the four colour theorem in graph theory was developed using the proof assistant Coq. [2]

## 1.2 Project

The aim of this project is to combine both of these approaches, using an interactive theorem prover which are most often used for proving theorems in fields like logic and set theory and applying it for the purpose of verifying the correctness of functional programs, which share a lot of their structure with these more abstract mathematical areas.

The proof assistant that will be used is Isabelle/HOL (Higher Order Logic), which is a tool that was developed by researchers at the University of Cambridge and Technische Universität München but now includes contributions from all over the world, and "allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus." [3] Isabelle has been used to formalise many mathematical theorems in a variety of abstract fields, but has also been used for more industrial applications such as formally verifying the operating system of the seL4 microkernel, which comprises "8,700 lines of C code and 600 lines of assembler". [4] As well as this, Isabelle is written in the functional language Standard ML, so its structure makes it more easy to represent data structures common in functional programs which will be useful for this application.

Overall, the project will involve starting by proving basic statements about functional programs such as identities involving natural numbers and lists, and then progressing to more complex

structures like binary trees, strings and languages, allowing gradually more intricate statements to be verified as correct. This involves using the various tools provided by Isabelle in order to aid in creating proofs from basic simplification rules to the command "sledgehammer" which utilises many external automated theorem provers, as well as developing the proofs themselves in a mathematical way using techniques such as proofs by induction and case analysis.

## 1.3 Summary

This report will proceed by first giving a survey of existing literature in this area. As this project is mainly theoretical rather than involving the development of a practical piece of software, this survey will mainly consist of an introduction to functional programming and the field of formal verification, supported by a brief discussion of possible strategies for formally verifying programs by hand or using an interactive theorem prover such as Isabelle.

Following this, the aims and objectives of the project as a whole will be analysed in detail. This section will aim to discuss what will and what will not be covered by the finished project, having a slightly deeper look into some individual aspects of what the project will entail. Some points that need to be considered are what it means to verify the correctness of a program, so that it can be judged whether a proof is successful, and how the process of proving a statement using Isabelle works in practice.

At this stage, the report will go over what has been achieved over the course of the project. This will take the form of a progression through the various data structures that have been covered in increasing order of complexity, and a thorough consideration of what statements have been verified about the behaviour of each one.

Finally, a brief discussion of the main results achieved will be given, along with conclusions about the overall experience of working with Isabelle to verify the correctness of functional programs in order to give a sense of whether this avenue is worth pursuing in the real world. Suggestions for further work will also be provided, including aspects of the project that were too difficult to fully complete and new areas of investigation that came up along the way.

## 1.4 Plan

As has been stated, the project will take the form of a progression through various data structures of increasing complexity, proving statements about each in order to verify their correctness. An outline of the various topics to be covered is as follows.

- Natural numbers. This will start with basic statements about the behaviour of natural numbers including computational statements as well as more mathematical ideas about entities like prime numbers, and progress to the verification of the Euclidean algorithm, a practical tool for finding the greatest common divisor of two natural numbers.

- Lists. This will present the idea that proving statements inductively on natural numbers can be extended to proving statements by structural induction on any algebraic data type, with a view towards verifying the correctness of basic sorting algorithms such as bubble sort and merge sort, showing that they can sort lists into ascending order whilst also preserving the contents of the list.

- Binary trees. These are slightly more complex than the previously considered structures, but the idea of proving things about them is the same; first simple statements about traversals of trees into lists can be proved, moving on to defining the binary search tree property and showing that insertion or deletion of elements into a search tree preserves this property.

- Shuffle algebras. These proofs are the most complex that will be considered, as they involve functions between strings and languages, and Isabelle's automated tools struggle with these statements. However, basic properties of the shuffle product such as commutativity and associativity will be considered, at both the string and language level.

# Chapter 2

# Literature Survey

## 2.1 Functional Programming

As opposed to more traditional imperative programming where writing a program consists of creating a list of instructions that will be executed sequentially by the compiler or interpreter, functional programming involves creating functions that, when given a set of arguments, return a well-defined result. One way to put it is that "the primary role of the programmer is to construct a function to solve a given problem." [5] This means that functional programs have notation and structure very similar to that used for functions throughout pure mathematics, and that similarly we can apply mathematical reasoning and techniques in order to derive information about functional programs.

Functional programming is based on a system of computation known as the lambda calculus or $\lambda$-calculus, which is a "simple but powerful mathematical theory of functions" [6] introduced by mathematician Alonzo Church in the 1930s with which it is possible to simulate any Turing machine. The programming language commonly regarded as the first to make use of the lambda calculus was Lisp, developed by John McCarthy, which is a language that is still used today in some contexts. In his original paper from 1960, McCarthy describes Lisp as "a scheme for representing the partial recursive functions of a certain class of symbolic expressions", [7] but over time the language evolved and is to this day one of the most favoured programming

languages for artificial intelligence research.

In the modern day, many commonly used languages have adopted functional features like recursion and higher order functions, but the majority still primarily encourage programming in a more imperative or object-oriented style. However, there are also many languages that do support functional programming, one of the most prominent of which being Haskell - "a purely functional language that allows programmers to rapidly develop software that is clear, concise and correct." [6] Haskell is widely used in academia, but also has been used for various purposes in industry; for example, it was applied by Facebook for the task of reducing spam, who "found it to be reliable and performant in practice" and that the functional style made their software "easy to test" and helped to "eliminate many bugs before putting policies into production." [8]

## 2.2 Formal Verification

Formal verification of a piece of software involves translating the algorithms into an abstract mathematical model and then proving or disproving its correctness using formal methods. The models used for this process can vary widely depending on the type of software being verified, ranging from automata like finite state machines through to descriptions in logics such as linear temporal logic or computational tree logic.

One of the oldest approaches to formal verification is through the use of Hoare logic, developed by Tony Hoare in 1969 and then built on by other researchers with the goal of creating "an axiomatic basis for computer programming" so that "all the properties of a program and all the consequences of executing it" can be discovered through "purely deductive reasoning." [9] Hoare logic works by using logical triples that describe how the execution of a piece of code changes the state that the computation is in, of the form $\{P\}\ S\ \{Q\}$, where P is the precondition true before the code executes, Q is the postcondition that is true after, and S is the statement being executed. Using standard Hoare logic, only partial correctness can be proven; this means that "we do not require the program to terminate, whereas in total correctness we insist upon its termination." [10]

Another possible and complementary approach is the idea of program derivation, where software is produced from a specification that is known to be correct by a series of steps which preserve correctness. This is particularly appropriate in the context of functional programs - in this case, "it is surprising how often a program can be calculated by simple equational reasoning from a mathematical description of what it is supposed to do." [5] Perhaps the most well-known example of this approach is the Bird-Meertens formalism, a "calculus for program construction" with "ultimate goal the derivation of functional programs." [11] In this formalism, a mathematical specification of the program is created and then transformed by equational reasoning into more and more efficient versions that are still known to be correct, which can then be implemented as a complete functional program.

## 2.3 Real World Applications

These kinds of verification methods have been applied to all kinds of projects in the real world, from industrial software systems to abstract mathematical theorems. One high profile example is the CompCert project, which sought to verify the correctness of a compiler for a significant subset of the C programming language to PowerPC assembly code. The Coq proof assistant, a tool very similar to Isabelle/HOL in many ways, was used for both writing the code for this compiler and also proving that it is correct. Being able to verify the correctness of a compiler is particularly important in the context of being able to write safe software; "the verification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well." [12]

An example of real world software verification that used Isabelle/HOL is the seL4 microkernel which was briefly discussed earlier. The kernel of an operating system could be described as the most critical thing to consider to ensure that the system behaves correctly, as "the security and reliability of a computer system can only be as good as that of the underlying operating system kernel", and "any fault in the kernel's implementation has the potential to undermine the correct operation of the rest of the system." [4] The verification of the seL4 kernel using Isabelle is a proof of concept showing that "full, rigorous, formal verification is

practically achievable for OS microkernels with very reasonable effort compared to traditional development methods." [4]

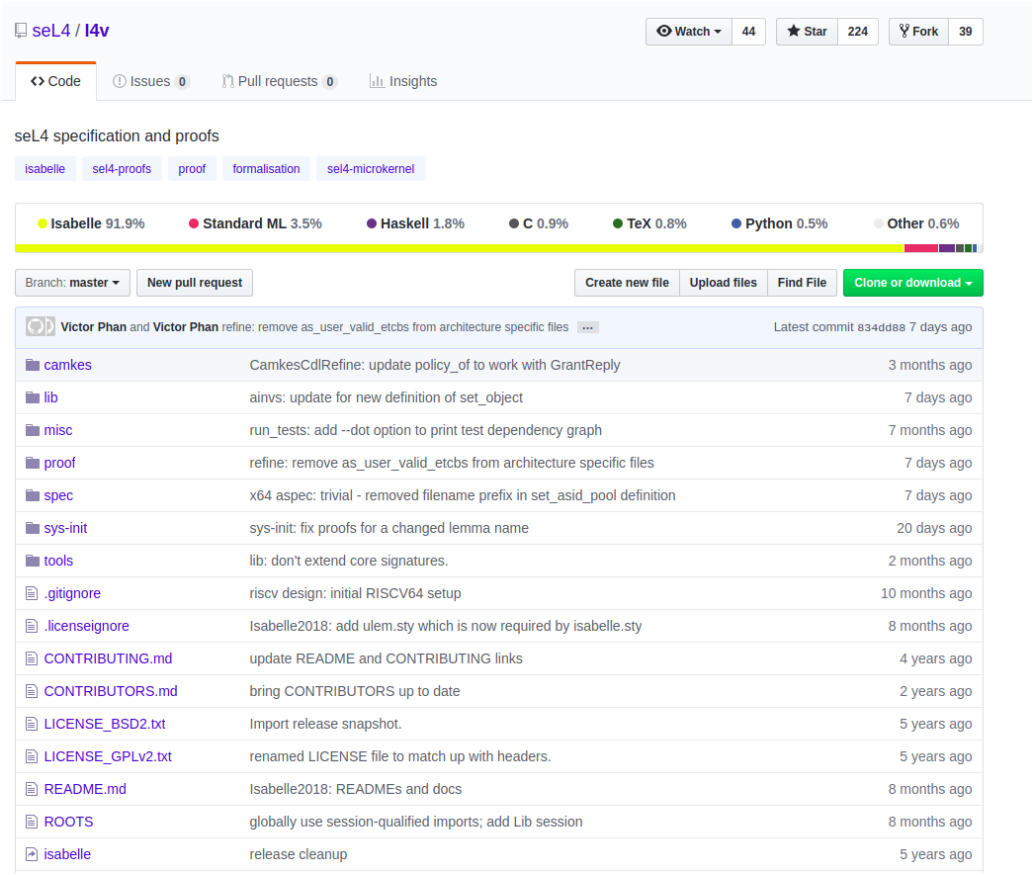


Figure 2.1: A screenshot of the GitHub repository for the seL4 specification and proofs

Lastly, a perhaps more down-to-earth example of verification using Isabelle can be seen from the CoCon system, which is a conference management system with verified document confidentiality. Such conference systems are widely used within the scientific community and often involve personal information being communicated between users, so being able to verify the security of such a system is a valuable application of a proof assistant and would have many future applications in a variety of fields. The CoCon system exists as "a case study in verified security for realistic systems" that "led to a novel security model and verification method generally applicable to systems describable as input-output automata." [13]

# Chapter 3

# Requirements

## 3.1 Objectives

The overall aim of the project is to start from the basics of becoming familiar with Isabelle and proving simple statements about data types such as booleans, natural numbers and lists, and then to gradually progress to proving more and more involved statements about more complex structures including binary trees and sets of lists, and statements featuring the use of higher-order functions such as maps and filters. Alongside this there will be an ongoing process of using various techniques to prove the necessary statements, combining manual mathematical methods such as proof by case analysis or induction and more automatic techniques such as Isabelle's "sledgehammer", which calls a number of external automatic theorem provers in order to assist the programmer in finding a proof.

Isabelle allows proofs to be written in two different styles; procedural proofs give a series of tactics to apply in order to prove a statement, making the proofs look more like traditional programs, while declarative proofs written in Isabelle's proof language Isar give the actual mathematical operations that should be performed, and so are "readable without being run because you need to state what you are proving at any given point." [14] This project will primarily present proofs in the declarative style using the Isar language, as the procedural style is generally considered to be deprecated and the declarative style is also encouraged by the

Archive of Formal Proofs, which is a collection of proofs checked by Isabelle organised like a scientific journal. [15]

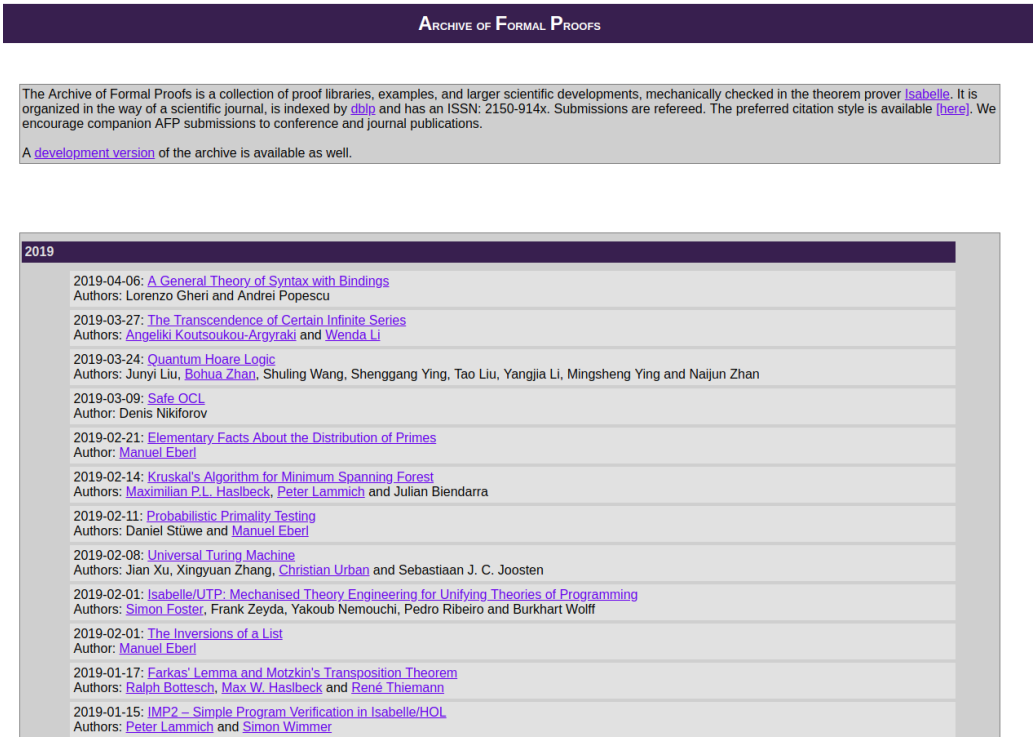


Figure 3.1: A screenshot of the Archive of Formal Proofs

## 3.2 Scope

Of course, there are limits as to what can be achieved with regard to verifying functional programs within the timescale of the project, as well as limits as to what can be achieved using Isabelle. For example, although the aim is to prove statements involving higher order functions and other similar entities, it will not be possible to prove abstract statements about advanced structures from functional programming like applicatives and monads. This is because Isabelle's type class system is "not powerful enough" [16] to specify these general structures; it allows for generalised types using type variables but it does not allow for generalised type constructors which would be necessary for this,although it may be possible to prove statements about specific monads for instance.

Another consideration is whether to attempt to use Isabelle in order to derive programs from

their mathematical specifications, using techniques like the Bird-Meertens formalism which was discussed earlier. This would be an interesting strategy to follow which could give many valuable results, but it would certainly be more complex than simply proving statements about functional programs in a procedural or declarative style. The merits of such an approach will be discussed and the process of deriving a program from its specification will be touched upon in some of the more complex proofs later in the project.

## 3.3 Evaluation

As this is an especially open-ended sort of project, with no real defined end goal beyond getting as far as possible with verifying more and more complex functional programs, it will be quite difficult to evaluate the project in the end and come to a conclusion on whether it was successful or not. However, the intended final aim of the project is to be able to verify functional programs that would be genuinely useful in a practical context rather than only being useful as theoretical examples, and so whether this is achieved will give a sense of whether the project was an overall success.

In order to progress towards this eventual goal, the strategy will be to start from basic examples that demonstrate the variety of functionality offered by Isabelle, and then build these up into more thorough examples making use of many features from both Isabelle and functional programming, with the final result of proving statements about functions that could exist as a vital part of a real piece of software. Some examples of functions like this will be simple sorting algorithms that act recursively over lists.

# Chapter 4

# Analysis

## 4.1 Process

The process of proving a statement's correctness within Isabelle/HOL differs depending on whether the proof is intended to be procedural or declarative. Procedural proofs specify a series of tactics to be applied in order to reduce the statement to a point where its truth value can be determined. This is done using Isabelle's **apply** function, which applies a given method to the statement, ranging all the way from a particular natural deduction rule in logic through to more fully-featured methods for rewriting terms. One particularly ubiquitous proof method is **simp**. Theorems that have already been proven can be declared as simplification rules, and the **simp** method will use these rules to rewrite the given expression, ideally into a simpler form. In cases where **simp** is not enough, more powerful methods such as **auto** can be used. Furthermore, more specialised methods like **blast** exist for particular types of statement; **blast** is particularly powerful for complex logical goals, and "because of its strength in logic and sets and its weakness in equality reasoning, it complements the earlier proof methods." [14] This style of proof is most common among proof assistants.

Declarative proofs operate slightly differently, although they can use many of the same methods. Instead of specifying a series of tactics to be applied, declarative proofs use Isabelle's proof language Isar to write the proof as a structured document, in the same way that one might

expect to see a mathematical proof written out. This is done to "provide a conceptually different view on machine-checked proofs", as "human-readable formal texts gain some value in their own right, independently of the mechanic proof-checking process." [17] This style is less commonly seen in other proof assistants, but is generally accepted as the superior style in modern Isabelle proofs. An example of an Isar proof is given below, one of the examples distributed with the current release of Isabelle. It shows the drinker's principle from classical predicate logic, sometimes stated as "in a room with people there always is at least one person, such that if that person starts to drink, then everybody in the room starts to drink." [18]

```
theorem Drinker's_Principle: "∃x. drunk x ⟶ (∀x. drunk x)"
proof cases
  assume "∀x. drunk x"
  then have "drunk a ⟶ (∀x. drunk x)" for a ..
  then show ?thesis ..
next
  assume "¬ (∀x. drunk x)"
  then have "∃x. ¬ drunk x" by (rule de_Morgan)
  then obtain a where "¬ drunk a" ..
  have "drunk a ⟶ (∀x. drunk x)"
  proof
    assume "drunk a"
    with ‹¬ drunk a› show "∀x. drunk x" by contradiction
  qed
  then show ?thesis ..
qed
```

(This principle is sometimes referred to as a "paradox", but it is a perfectly valid logical theorem; a more natural way of stating the above proof would be to say that either everybody is drinking, in which case the right hand side of the implication is true, or there exists at least one person who is not drinking, in which case for that particular person the left hand side of the implication is false, so the whole implication is again true.) Although this proof does not make use of methods such as **auto** or **blast**, it is possible to do so within Isar proofs, and indeed it will be necessary in more complex scenarios.

## 4.2 Tools

Isabelle also provides a number of automated tools, the use of which can make the process of finding a successful proof of any statement much easier and quicker. The two main examples used to great effect over the course of this project are **nitpick** and **sledgehammer**.

- **nitpick** is "a counterexample generator for Isabelle/HOL that is designed to handle formulas combining (co)inductive datatypes, (co)inductively defined predicates, and quantifiers." [19] Nitpick is very useful for checking that lemmas or theorems have been stated correctly, as if a counterexample exists it is clear the statement is not correct and a lot of time can be saved by avoiding trying to prove something that was faulty to begin with.

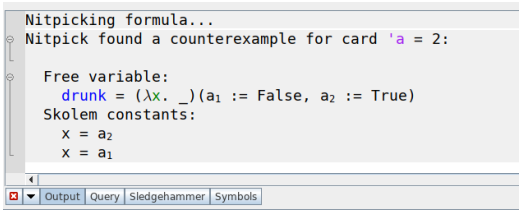


Figure 4.1: An example of output from **nitpick**

- **sledgehammer** is "a tool that applies automatic theorem provers (ATPs) and satisfiability-modulo-theories (SMT) solvers on the current goal." [20] The provers and solvers used for this project are CVC4 and Z3, which are SMT solvers, SPASS and E, which are ATPs run locally, and Vampire, which is an ATP run remotely over the web. These are very useful for automatically finding a proof for complex statements, confirming that it is possible to prove and also giving a good idea of how to go about achieving that goal.

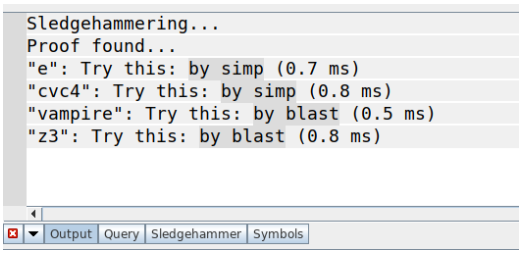


Figure 4.2: An example of output from **sledgehammer**

# Chapter 5

# Implementation

## 5.1 Natural Numbers

The most basic functional data type that we can prove statements about in Isabelle is booleans, a data type defined as having a value of either True or False. However, most easily stated theorems in propositional and predicate logic have already been formally verified in Isabelle and have entries in the Archive of Formal Proofs, since natural deduction proofs lend themselves well to the structure of the Isar proof language. Indeed, an example of this was given earlier in the form of a proof of the drinker's principle. As such, we will begin by discussing what is probably the next most simple data type; the natural numbers. We can define the natural numbers in Isabelle as follows.

```
- ‹Natural numbers, defined as either zero or the successor of another natural
   number.›
datatype mynat = Zero | Suc mynat

- ‹Addition of natural numbers, defined recursively.›
fun add :: "mynat ⇒ mynat ⇒ mynat" where
"add Zero n = n" |
"add (Suc m) n = Suc (add m n)"
```

So far, this definition looks very similar to how we might define a data type for natural numbers

in any other functional language, such as Haskell. The difference in Isabelle (and other proof assistants such as Coq or Agda) is that now that we have defined the type, we can also define lemmas and statements that we expect to hold true for values of that type, and then prove them with Isabelle's assistance. A basic example of this involving the set of even numbers (numbers divisible by two) is given below.

```
- ‹Sets of natural numbers can be defined inductively; a number is even if it is
   either zero or the successor of the successor of another natural number.›
inductive_set even :: "mynat set"
  where
  zero[intro!]: "Zero ∈ even" |
  step[intro!]: "n ∈ even ⟹ Suc (Suc n) ∈ even"

- ‹We prove that if a number n is even, there exists another number that can be
   doubled to give n.›
theorem can_halve_even [intro!]: "n ∈ even ⟹ (∃k. add k k = n)"
- ‹The proof is inductive, using the definition of even numbers to give the
   required cases.›
proof (induct rule: even.induct)
  case zero
- ‹When n is zero, we can double zero to give zero, so the proof is simple.›
  have "add Zero Zero = Zero" by simp
  then show ?case by (rule exI)
next
  case (step n)
- ‹In the step case, we have a number that doubles to give n, so take its
   successor for the required example of a number that doubles to give
   Suc (Suc n)).›
  from this obtain m where "add m m = n" by atomize_elim
  then have "add (Suc m) (Suc m) = (Suc (Suc n))" by simp
  then show ?case by (rule exI)
qed
```

As can be seen from this instance, the best strategy to achieve a proof in Isabelle can often be to follow the structure of the definition. Here we define even numbers as an inductive set of natural numbers, where an even number is either zero or the successor of the successor of another even number, and then the proof by induction follows this structure; we prove the statement for the base case - zero - and then for the step case, showing that if the statement is true for a particular even number it must be true for the successor of the successor of that number.

A further example of a simple proof after this fashion is given in Appendix A, involving divisibility and prime numbers.

We now come to our final goal involving natural numbers, which is to verify the correctness of an implementation of the Euclidean algorithm for finding the greatest common divisor of two natural numbers. The simplest way to calculate the greatest common divisor of two numbers is to test all their divisors and find the largest that is common to each, but this is not the most efficient method. "The French mathematician Gabriel Lamé proved that the number of steps required in the Euclidean algorithm is at most five times the number of digits in the smaller integer" [21]; as such, the complexity of the Euclidean algorithm is at most $O(\log n)$ with $n$ being the smaller integer, which is faster than what could be achieved by testing all its divisors. We can use the Euclidean algorithm by hand to find, for instance, $gcd(12378, 3054)$, as follows:

$$12378 = 4 \cdot 3054 + 162$$

$$3054 = 18 \cdot 162 + 138$$

$$162 = 1 \cdot 138 + 24$$

$$138 = 5 \cdot 24 + 18$$

$$24 = 1 \cdot 18 + 6$$

$$18 = 3 \cdot 6 + 0$$

As 6 is the last non-zero remainder that appears, 6 is the greatest common divisor of 12378 and 3054. We can now proceed to verifying the correctness of the Euclidean algorithm with the assistance of Isabelle.

---

```
- ‹We can define more complex examples of inductive sets. Here we define a set
   of 3-tuples of natural numbers, such that if (n, m, g) is in gcd then g is
   the greatest common divisor of n and m.›
inductive_set gcd :: "(nat × nat × nat) set" where
zero[intro!]: "(n, 0, n) ∈ gcd" |
step[intro!]: "(n - m, m, g) ∈ gcd ⟹ m ≤ n ⟹ (n, m, g) ∈ gcd" |
swap[intro!]: "(m, n, g) ∈ gcd ⟹ n < m ⟹ (n, m, g) ∈ gcd"

- ‹We can then inductively prove statements about the gcd using the structure of
```

```
   the definition. First we prove that the gcd is indeed a divisor of n and m.›
lemma gcd_divides: "(n, m, g) ∈ gcd ⟹ g dvd n ∧ g dvd m"
proof(induction rule: gcd.induct)
    case(zero n)
    then show ?case by simp
next
    case(step n m g)
    then show ?case using dvd_diffD by blast
next
    case(swap n m g)
    then show ?case by simp
qed

- ‹Finally we prove the correctness of the definition of gcd; in other words, if
   d is a divisor of n and m, then it must either be g or smaller than g.›
theorem gcd_greatest [rule_format]:
    "(n, m, g) ∈ gcd ⟹ 0 < n ∨ 0 < m ⟹ (∀ d. d dvd n ⟶ d dvd m ⟶ d ≤ g)"
proof(induction rule: gcd.induct)
    case(zero n)
    then show ?case using dvd_imp_le by simp
next
    case(step n m g)
    then show ?case
- ‹This proof is slightly more complicated in that we need to split the step
   case into two cases; one where n = m, and one where it does not.›
    proof(cases "n = m")
    case True
    then show ?thesis using step by simp
    next
    case False
    then show ?thesis using step by simp
    qed
next
    case(swap n m g)
    then show ?case by simp
qed
```

Our definition of the greatest common divisor follows the structure of the algorithm in that each instance of the step case is equivalent to a step in the algorithm, until one of the remainders is equal to zero at which point the algorithm is complete and we can use the base case. Proofs in Isabelle are often made much easier by using a functional or inductive definition like this, so that there is a natural route for the proof to follow.

## 5.2 Lists

Although these proofs on natural numbers are certainly of interest, and provide a good demonstration of the functionality of Isabelle, apart from the last proof of the Euclidean algorithm they cannot really be said to demonstrate verification of the correctness of a functional program. As such, we move on to what is possibly the most ubiquitous functional data type, the list.

Functional programs generally feature data types which are algebraic; that is, they are types assigned to data that are themselves made up of other types. Natural numbers were an example of this, as they are constructed from other natural numbers. However, the most simple more general example that can be constructed from any other type is a list, which is usually defined as either the empty list or a list formed by concatenating a single arbitrary element of any other type with another list.

We can prove statements inductively over lists in a similar way as with natural numbers; just as we can prove a statement is true for all natural numbers by first showing it is true for zero and then showing that if it is true for $n$ it must be true for $Suc\, n$, we can prove a statement is true for all lists by first showing it is true for the empty list and then showing that if it is true for some list $xs$, then it must also be true when some element $x$ is concatenated with $xs$.

---

```
- ‹Lists, defined as either the empty list or a single element at the head of
   another list.›
datatype 'a mylist = Nil | Cons 'a "'a mylist"

- ‹Some simple list functions; appending two lists and reversing a list.›
fun app :: "'a mylist ⇒ 'a mylist ⇒ 'a mylist" where
"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"

fun myrev :: "'a mylist ⇒ 'a mylist" where
"myrev Nil = Nil" |
"myrev (Cons x xs) = app (myrev xs) (Cons x Nil)"

- ‹Lemmas about these functions can be proved in a similar way to those about
   natural numbers, using induction over the structure of lists.›
```

```
lemma app_nil [simp]: "app xs Nil = xs"
proof (induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    then show ?case by simp
qed

lemma app_assoc [simp]: "app (app xs ys) zs = app xs (app ys zs)"
proof(induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    then show ?case by simp
qed

lemma rev_app [simp]: "myrev (app xs ys) = app (myrev ys) (myrev xs)"
proof(induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    then show ?case by simp
qed

- ‹We prove the reverse function is an involution, which is simple using the
   lemmas that were already proved.›
theorem rev_rev [simp]: "myrev (myrev xs) = xs"
proof(induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    then show ?case by simp
qed
```

This last proof is the most involved thus far, and required a variety of lemmas to be proven in order to prove the main theorem. First the functions were defined; in order to define the reversal function, it was necessary to first define a function for appending one list onto the end of another, and then it became necessary to show that this append function was associative in order to prove that the reverse function was associative, which made the final proof that reversing twice is equivalent to the identity function possible.

Now that we have the reverse function, we might want to consider proving its correctness; we already know that it is its own inverse, so all that is left to show is that if an element is in the original list, it still remains after the list is reversed. We can do that as follows, with some preliminary lemmas about what it means for an element to be contained in a list.

---

```
- ‹Function definitions can also contain conditional statements, like this one
   which returns True if a given element is contained in a given list and False
   otherwise.›
fun contains :: "'a ⇒ 'a mylist ⇒ bool" where
"contains x Nil = False" |
"contains x (Cons y xs) = (if y = x then True else contains x xs)"

- ‹Statements about such functions can be proved in much the same way.›
lemma contains_app [simp]: "contains x xs ⟹ contains x (app xs ys)"
proof (induct xs)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    then show ?case by simp
qed

lemma contains_prep [simp]: "contains x xs ⟹ contains x (app ys xs)"
proof (induct ys)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    then show ?case by simp
qed

- ‹We can now prove the correctness of the reverse function; we already know
   that it is its own inverse, and now we show that it preserves the elements
   of the original list.›
theorem contains_rev [simp]: "contains x xs ⟹ contains x (myrev xs)"
proof (induct xs)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    show ?case
- ‹This proof again requires a case split, with cases for if the required
   element is the head of the list and for if it is somewhere in the tail.›
    proof (cases "y = x")
    case True
```

```
    then show ?thesis by simp
    next
    case False with Cons
    have "contains x (myrev xs)" by simp
    then show ?thesis by simp
    qed
qed
```

The definition of the contains function is fairly simple, stating only that no element is contained in the empty list, and otherwise we must check every element in the list to see whether they are the element we are looking for. Note that proving that appending preserves containment and proving that prepending preserves containment required induction on different variables, as no kind of commutativity was ever proven and so the simplification was too complex the other way around in both cases. The final proof given, of the statement that if an element is in a list, it will still be there if the list is reversed, required the use of both of the previously proven statements about containment, along with the traditional simplification rules.

Now we can begin attempting to verify the correctness of some simple sorting algorithms. We will first prove the correctness of insertion sort, a basic algorithm for sorting, and then move on to merge sort, a more efficient alternative. The way insertion sort works is that we build the sorted array one element at a time, by inserting each element into its correct place. As such, in order to prove its correctness we can first verify that inserting an element into a sorted list preserves its sortedness, at which point proving the overall correctness should be simple.

First, however, we must consider what it means for a sorting algorithm to be correct. It needs to correctly sort the list such that after the algorithm is finished, the list has been sorted into ascending order, and it needs to preserve the elements in the original list, such that the sorted list is a permutation of the elements in the original list. With this in mind, we define the necessary criteria and proceed to verify insertion sort like so.

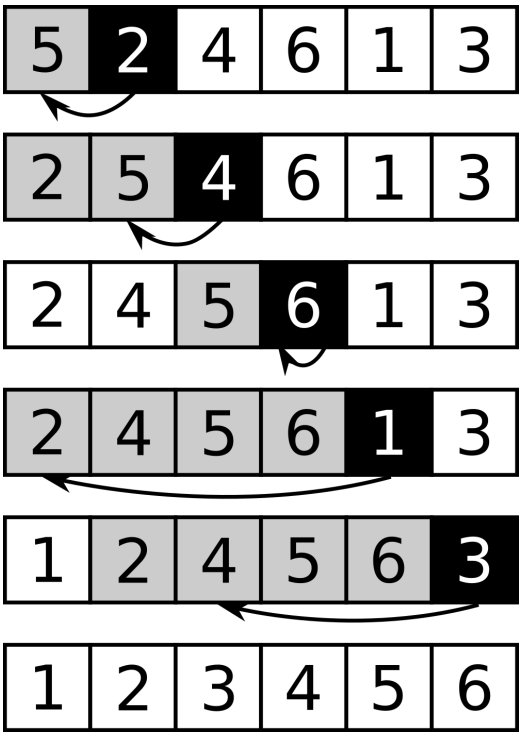


Figure 5.1: An illustration of the insertion sort algorithm

```
- ‹Here we define some auxiliary functions that will be used to test whether a
   sorting algorithm is correct or not; for this, we need to know that the new
   list is a permutation of the original list, and also that the elements of the
   new list will be in sorted order.›
fun inlist :: "'a ⇒ 'a list ⇒ bool" where
"inlist x [] = False" |
"inlist x (y#xs) = (if x = y then True else inlist x xs)"

fun permutation :: "'a list ⇒ 'a list ⇒ bool" where
"permutation x y = (∀n. count n x = count n y)"

fun sorted :: "nat list ⇒ bool" where
"sorted [] = True" |
"sorted (n#[]) = True" |
"sorted (n#(m#ns)) = (if n ≤ m then sorted (m#ns) else False)"

- ‹Now we proceed to proving the correctness of insertion sort. We define a
   functional version of the insert function and then the overall insertion sort
   algorithm as follows.›
fun insert :: "nat ⇒ nat list ⇒ nat list" where
"insert n [] = (n#[])" |
"insert n (m#ns) = (if n ≤ m then n#m#ns else m#insert n ns)"

fun insertionsort :: "nat list ⇒ nat list" where
```

```
"insertionsort [] = []" |
"insertionsort (x#xs) = insert x (insertionsort xs)"

- ‹First we must prove that inserting an element into a sorted list preserves
   sortedness.›
lemma insertsorted: "sorted xs ⟹ sorted (insert x xs)"
proof (induction xs)
    case Nil
    thus ?case by simp
next
    case (Cons y xs)
- ‹Not only does this proof require a case split, it is also apparent that some
   statements were difficult to prove by hand and required the use of
   sledgehammer, giving a proof using metis which is one of Isabelle's automatic
   theorem provers. The use of such tools will become more frequent as we
   proceed.›
    hence "sorted xs" by (metis sorted.elims(3) sorted.simps(3))
    hence ih: "sorted (insert x xs)" using Cons by simp
    thus ?case
    proof (cases "x ≤ y")
    case True
    hence "sorted (x#y#xs) = sorted (y#xs)" by simp
    also have "insert x (y#xs) = x#y#xs" using True by simp
    ultimately show ?thesis using Cons by simp
    next
    case False
    hence "y ≤ x" by simp
    also have "insert x (y#xs) = y#insert x xs" using False by simp
    ultimately show ?thesis using Cons
      by (metis ih sorted.simps(3) insert.elims)
    qed
qed

- ‹This allows us to prove that insertion sort gives a list that is sorted with
   relative ease.›
lemma insertionsort_sorted: "sorted (insertionsort xs)"
proof(induction xs)
    case Nil
    thus ?case by simp
next
    case (Cons y xs)
    thus ?case using insertsorted by simp
qed

- ‹We also need to prove that inserting an element into a list gives a
   permutation of the original list with the addition of the new element. This
   proof uses a similar case split to the earlier proof.›
lemma insertperm: "permutation (x#xs) (insert x xs)"
```

```
proof(induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    hence all: "∀ n. count n (x#xs) = count n (insert x xs)" by simp
    thus ?case
    proof (cases "x ≤ y")
    case True
    then have "insert x (y#xs) = x#y#xs" by simp
    thus ?thesis by simp
    next
    case False
    hence "y ≤ x" by simp
    then have "permutation (x#y#xs) (y#x#xs)" by simp
    also have "insert x (y#xs) = y#insert x xs" using False by simp
    ultimately show ?thesis using Cons by simp
    qed
qed

- ‹Again, this allows us to prove that insertion sort gives a permutation of the
   original list.›
lemma insertionsort_perm: "permutation xs (insertionsort xs)"
proof (induction xs)
    case Nil
    thus ?case by simp
next
    case (Cons y xs)
    hence "insertionsort (y#xs) = insert y (insertionsort xs)" by simp
    also have "permutation (y#xs) (y#insertionsort xs)" using Cons
      by simp
    thus ?case using insertperm by simp
qed

- ‹This gives us the correctness of insertion sort, as a trivial consequence of
   the lemmas we have now proved.›
theorem insertionsort_correct:
  "sorted (insertionsort xs) ∧ permutation xs (insertionsort xs)"
    using insertionsort_sorted insertionsort_perm by simp
```

This proof was certainly more complex than those that came before it, and required not only the proof of several lemmas but also the use of **metis**, one of Isabelle's automated theorem proving tools based on first order logic. Tools like this will be seen more often as we progress to more complex statements, as they allow a lot of the menial work involved in writing a proof

to be cut down to a more manageable level.

Merge sort is a significantly more efficient algorithm, using a divide and conquer strategy whereby we divide the unsorted list into sublists until each contains only one element, and then merge the sublists into new sorted sublists until there is only one remaining; this remaining list will be the sorted list.

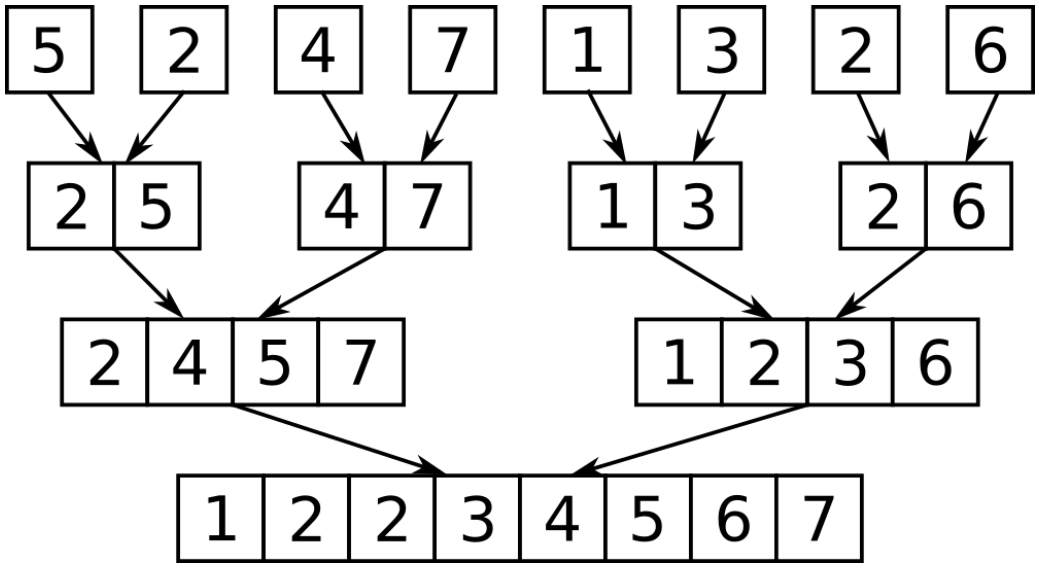


Figure 5.2: An illustration of the merge sort algorithm

Merge sort has an average and worst-case performance of $O(n \log n)$, as compared to insertion sort's $O(n^2)$; in fact, this is the best possible runtime for a comparison-based sorting algorithm, which has no additional information about the contents of the list. As such, merge sort is in most cases superior to insertion sort; "the advantage of merge sort is even more pronounced when we sort 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours." [22]

```
- ‹As merge sort is a more complicated algorithm (though it is more efficient),
   it will take more of Isabelle's machinery to prove, so it gives a good
   demonstration of the power that is provided.›
fun leq :: "nat ⇒ nat list ⇒ bool" where
"leq _ [] = True" |
"leq x (y#xs) = (x ≤ y ∧ leq x xs)"
```

```
fun merge :: "nat list ⇒ nat list ⇒ nat list" where
"merge xs [] = xs" |
"merge [] ys = ys" |
"merge (x#xs) (y#ys) =
  (if x ≤ y then x#merge xs (y#ys) else y#merge (x#xs) ys)"

fun mergesort :: "nat list ⇒ nat list" where
"mergesort [] = []" |
"mergesort [x] = [x]" |
"mergesort xs =
    merge (mergesort (take (length xs div 2) xs))
      (mergesort (drop (length xs div 2) xs))"

- ‹We give a definition of sortedness in terms of the leq function, that will
   make it easier to prove some statements about merge sort.›
lemma sorted_leq: "sorted (x#xs) = (leq x xs ∧ sorted xs)"
proof(induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons y ys)
    then show ?case
      by (metis Lists.sorted.simps(3) insertionsort.cases le_trans leq.elims(3)
        leq.simps(2) list.inject neq_Nil_conv)
qed

- ‹We then proceed in a similar manner to insertion sort; first we need to prove
   that merge sort gives a permutation of the original list.›
lemma merge_count: "count x (merge xs ys) = count x xs + count x ys"
proof(induction xs ys rule: merge.induct)
    case(1 xs)
    then show ?case by simp
next
    case(2 y ys)
    then show ?case by simp
next
    case(3 x xs y ys)
    then show ?case by auto
qed

lemma count_append: "count x (xs@ys) = count x xs + count x ys"
proof(induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons z zs)
    then show ?case by auto
qed
```

```
lemma take_drop_count: "count x (take n xs) + count x (drop n xs) = count x xs"
proof(induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons y xs)
    then show ?case by (metis append_take_drop_id count_append)
qed

lemma mergesort_count: "count x xs = count x (mergesort xs)"
proof(induction xs rule: mergesort.induct)
    case 1
    then show ?case by simp
next
    case (2 x)
    then show ?case by simp
next
    case(3 x y ys)
    then show ?case by (metis merge_count mergesort.simps(3) take_drop_count)
qed

- ‹We continue to prove that merge sort results in a sorted list.›
lemma merge_leq: "leq x (merge xs ys) = (leq x xs ∧ leq x ys)"
proof(induction xs ys rule: merge.induct)
    case (1 xs)
    then show ?case by simp
next
    case(2 y ys)
    then show ?case by simp
next
    case(3 x xs y ys)
    then show ?case by auto
qed

lemma merge_sorted: "sorted (merge xs ys) = (sorted xs ∧ sorted ys)"
proof(induction xs ys rule: merge.induct)
    case (1 xs)
    then show ?case by simp
next
    case(2 y ys)
    then show ?case by simp
next
    case (3 x xs y ys)
    then show ?case
      by (metis insert.simps(2) insertsorted linear merge.simps(3) merge_leq
        sorted_leq)
qed
```

```
lemma mergesort_sorted: "sorted (mergesort xs)"
proof(induction xs rule: mergesort.induct)
    case 1
    then show ?case by simp
next
    case (2 x)
    then show ?case by simp
next
    case (3 x y ys)
    then show ?case by (simp add: merge_sorted)
qed

- ‹Finally, we have that merge sort is also a correct sorting algorithm.›
theorem mergesort_correct:
  "sorted (mergesort xs) ∧ permutation xs (mergesort xs)"
    using mergesort_count mergesort_sorted by simp
```

As merge sort is a more complex algorithm than insertion sort, the proofs require a little more usage of heavy duty tools like **metis** rather than being able to rely on the simplifier, but they are still not overly difficult to achieve.

## 5.3 Binary Trees

We now move on to a slightly more complicated recursive data structure that nonetheless appears very frequently in functional programs, the binary tree. Binary trees can be defined in a few different ways, but the definition we will use is that a tree can either be empty, or it can be a node which contains a single element and branches into two additional binary trees.

```
- ‹Trees, defined as either the empty tree or a single element attached to two
   subtrees.›
datatype 'a tree = Empty | Node 'a "'a tree" "'a tree"

- ‹Again, we can define simple functions on trees and prove statements about
   them inductively.›
fun mirror :: "'a tree ⇒ 'a tree" where
"mirror Empty = Empty" |
"mirror (Node x l r) = Node x (mirror r) (mirror l)"

lemma mirrorinv: "mirror(mirror t) = t"
```

```
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node x l r)
    then show ?case by simp
qed
```

Some of the simplest statements about trees we can make involve traversals of trees; a traversal reduces a tree into a list, which can be done in several different ways. Here we consider preorder and postorder traversals, which traverse a tree by always taking the leftmost and rightmost branch first respectively.

```
- ‹We define various ways of traversing a tree to give a list; preorder and
   postorder traversals are shown, but an inorder traversal is also possible. We
   also define functions that reduce lists and trees down to the sum of their
   elements.›
fun preorder :: "'a tree ⇒ 'a list" where
"preorder Empty = []" |
"preorder (Node x l r) = x#((preorder l)@(preorder r))"

fun postorder :: "'a tree ⇒ 'a list" where
"postorder Empty = []" |
"postorder (Node x l r) = (postorder l)@(postorder r)@(x#[])"

fun sum_list :: "nat list ⇒ nat" where
"sum_list [] = 0" |
"sum_list (x#xs) = x + sum_list xs"

fun sum_tree :: "nat tree ⇒ nat" where
"sum_tree Empty = 0" |
"sum_tree (Node x l r) = x + sum_tree l + sum_tree r"

- ‹We can then prove that the traversals are correct, in that the sum of the
   elements in the list after traversing the tree is equivalent to the sum of
   elements in the original tree. Only one such example is given, as others
   would be very similar.›
lemma sumlistapp [simp]: "sum_list (xs@ys) = sum_list xs + sum_list ys"
proof(induction xs)
    case Nil
    then show ?case by simp
next
    case (Cons n xs)
    then show ?case by simp
```

```
qed

lemma sumtreelist: "sum_tree t = sum_list (preorder t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node x l r)
    then show ?case by simp
qed

- ‹We can also prove a statement about the relation between the two traversals
   we defined; if you mirror a tree and then take its preorder traversal, this
   is equivalent to taking the postorder traversal and then reversing the
   resulting list.›
lemma prepost: "preorder (mirror t) = rev (postorder t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node x l r)
    then show ?case by simp
qed
```

A perhaps more practical application of binary trees is that they are often used as binary search trees, which are simply binary trees that obey a certain property; for any node in the tree, containing a value of $n$, say, all values in the left subtree of that node must be less than $n$, and all values in the right subtree must be greater than $n$. This allows us to convert the tree into a list of elements in sorted order by simply taking its inorder traversal. We can insert an element into such a tree by traversing the tree and placing the element into its proper place, similarly to inserting an element into a sorted list.

Of course, whatever definition we give of such an insertion function, we will need to prove its correctness; that it preserves the search tree property, that any element previously in the tree remains in the tree after insertion, and that the new element has indeed been added. We do this below.

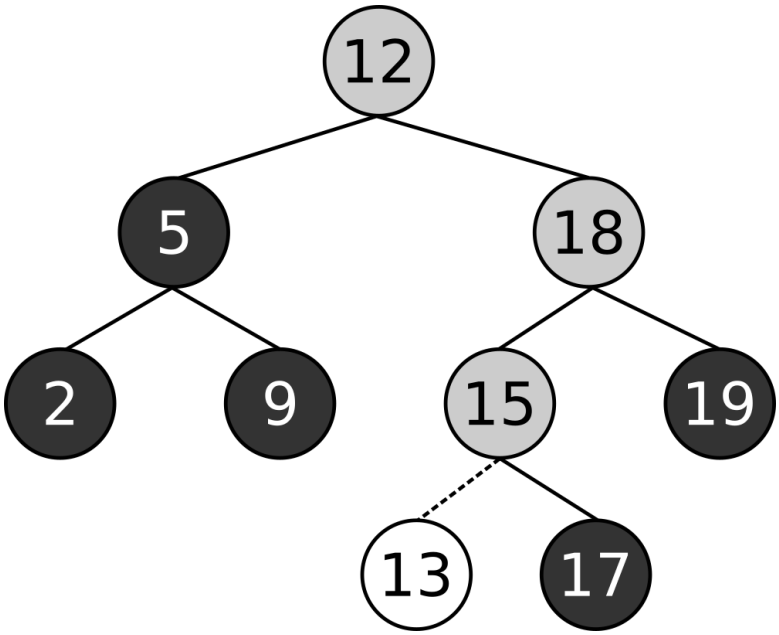


Figure 5.3: An illustration of inserting an element into a binary search tree

```
- ‹Now we define what it means for a tree to be a binary search tree; in the
   non-trivial case, the two subtrees must both themselves be search trees, and
   the root must be greater than all elements on its left and smaller than all
   elements on its right.›
fun greater :: "nat ⇒ nat tree ⇒ bool" where
"greater x Empty = True" |
"greater x (Node y l r) = (x ≥ y ∧ greater x l ∧ greater x r)"

fun smaller :: "nat ⇒ nat tree ⇒ bool" where
"smaller x Empty = True" |
"smaller x (Node y l r) = (x ≤ y ∧ smaller x l ∧ smaller x r)"

fun search :: "nat tree ⇒ bool" where
"search Empty = True" |
"search (Node x l r) = (greater x l ∧ smaller x r ∧ search l ∧ search r)"

- ‹Then we define a function that allows us to insert an element into a binary
   search tree, defining another function to tell us whether an element is
   already present to make the definition tidier.›
fun elem :: "nat ⇒ nat tree ⇒ bool" where
"elem x Empty = False" |
"elem x (Node y l r) =
  (if x = y then True else (if x ≤ y then elem x l else elem x r))"

fun insert :: "nat ⇒ nat tree ⇒ nat tree" where
"insert x Empty = Node x Empty Empty" |
"insert x (Node y l r) = (if elem x (Node y l r) then (Node y l r) else
```

```
    (if x ≤ y then (Node y (insert x l) r) else (Node y l (insert x r))))"

- ‹We now proceed to prove the correctness of the insert function. For this we
   need to prove three things; the insert function must preserve the binary
   search tree property, it must preserve the elements of the original tree, and
   of course it must actually insert the required element.›
lemma insert_greater: "greater x t ⟹ x ≥ y ⟹ greater x (insert y t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node x l r)
    then show ?case by simp
qed

lemma insert_smaller: "smaller x t ⟹ x ≤ y ⟹ smaller x (insert y t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node x l r)
    then show ?case by simp
qed

- ‹We first prove that insert preserves the search tree property, which will be
   the most complicated of the three necessary proofs.›
lemma insert_search: "search t ⟹ search (insert x t)"
proof(induction t rule: tree.induct)
    case Empty
    thus ?case by simp
next
    case (Node y l r)
    thus ?case
    proof(cases "elem x (Node y l r)")
    case True
    thus ?thesis using Node by simp
    next
    case False
    thus ?thesis
    proof(cases "x ≤ y")
- ‹The step case where the element to be inserted is not already in the tree is
   the most complex. Here we split into two cases depending on whether the new
   element is greater or smaller than the root, and then prove all the necessary
   conditions for the new tree to be a binary search tree.›
        case True
        hence insertleft: "insert x (Node y l r) = Node y (insert x l) r"
          using False by simp
        then have "smaller y r" using Node by simp
```

```
        also have "greater y l" using Node by simp
        then have gr: "greater y (insert x l)" using True
          by (simp add: insert_greater)
        from Node have "search l ∧ search r" by simp
        ultimately show ?thesis using Node insertleft gr by simp
    next
        case False
        then have "x ≥ y" by simp
        hence insertright: "insert x (Node y l r) = Node y l (insert x r)"
          using False
        by (metis insert.simps(2) elem.elims(2) elem.simps(2))
        then have "greater y l" using Node by simp
        also have "smaller y r" using Node by simp
        then have sm: "smaller y (insert x r)" using False
          by (simp add: insert_smaller)
        from Node have "search l ∧ search r" by simp
        ultimately show ?thesis using Node insertright sm by simp
    qed
    qed
qed

- ‹The other two proofs are much simpler, allowing us to quickly finish the
   overall proof that the insert function is correct.›
lemma insert_elem: "elem y t ⟹ elem y (insert x t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node z l r)
    then show ?case by simp
qed

lemma insert_inserts: "elem x (insert x t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node z l r)
    then show ?case by simp
qed

theorem insert_correct: "search t ∧ elem y t
  ⟹ search (insert x t) ∧ elem y (insert x t) ∧ elem x (insert x t)"
    using insert_search insert_elem insert_inserts by simp
```

We can also write a similar definition and resulting proof for deletion of an element for a

binary search tree. However, this is significantly more difficult than insertion, as deletion of an element that has two subtrees requires the search tree to be rearranged to preserve its structure; "modifying the tree to insert a new element is relatively straightforward, but handling deletion is somewhat more intricate." As such, the proof is longer and more involved, and the full details of implementing deletion of an element from a binary search tree in Isabelle can be found in Appendix B.

## 5.4 Shuffle Algebras

In our final section, we come on to proving statements about shuffle algebras. These are a way of formalising the notion of shuffling two strings together, in a similar way to interleaving packs of cards, which can also be extended to sets of strings (often called languages).

The shuffle product of two strings is the set of all strings that can be formed by interleaving them, as shown in the following examples.

$$ab \,||\, xy = \{axby,\ xaby,\ axyb,\ xayb,\ xyab\}$$
$$aaa \,||\, aa = \{aaaaa\}$$

It can be defined inductively in this way:

$$u \,||\, \epsilon = \epsilon \,||\, u = u$$
$$ua \,||\, vb = (u \,||\, vb)a + (ua \,||\, v)b$$

where $\epsilon$ is the empty word, $a$ and $b$ are single elements, and $u$ and $v$ are arbitrary words. [23]

We can formalise this definition and its extension to languages in Isabelle as follows, interpreting strings as lists since a functional string is simply a list of characters.

```
- ‹Definitions of strings and languages in terms of basic Isabelle types.›
type_synonym 'a string = "'a list"
```

```
type_synonym 'a language = "'a string set"

- ‹Extension of the basic string append function to languages.›
fun product :: "'a language ⇒ 'a language ⇒ 'a language"
(infixl "·" 60) where
"X · Y = {x@y | x y. x ∈ X ∧ y ∈ Y}"

- ‹The shuffle product defined on strings, and extended to languages.›
fun shuffle :: "'a string ⇒ 'a string ⇒ 'a language"
(infixl "‖" 70) where
"[] ‖ x = {x}" |
"x ‖ [] = {x}" |
"(a#x) ‖ (b#y) = ({[a]} · (x ‖ (b#y))) ∪ ({[b]} · ((a#x) ‖ y))"

fun shuffleproduct :: "'a language ⇒ 'a language ⇒ 'a language"
(infixl "⧢" 80) where
"X ⧢ Y = ⋃ {x ‖ y | x y. x ∈ X ∧ y ∈ Y}"
```

Translating these definitions into Isabelle could be considered an exercise in constructing a program from its specification, as much of the material available on shuffle algebras is purely mathematical. Indeed, proving statements about these operations will be difficult, as the way the definitions mix both strings and languages together makes them difficult to prove things about with automated tools. However, it is well known that the shuffle product has certain basic properties; "the shuffle product is a commutative and associative operation." [23] With a view towards verifying these properties hold for our definition, we begin by gathering some basic statements we will need from Isabelle's standard libraries.

```
- ‹Some basic lemmas about strings that are already provided by Isabelle.›
lemma string_assoc: "x @ (y @ z) = (x @ y) @ z"
    by simp

lemma string_iden: "x @ [] = x ∧ x = [] @ x"
    by simp

- ‹Similar lemmas for the set union function.›
lemma union_assoc: "X ∪ (Y ∪ Z) = (X ∪ Y) ∪ Z"
    by (simp add: sup_commute sup_left_commute)

lemma union_commu: "X ∪ Y = Y ∪ X"
```

```
    by (simp add: sup_commute)

lemma union_refle: "X ∪ X = X"
    by simp

lemma union_iden: "X ∪ {} = X"
    by simp

- ‹Some more complex facts about the language product, including associativity.
   Associativity is proved directly using the definitions.›
lemma product_assoc: "X · (Y · Z) = (X · Y) · Z"
proof -
have yz: "(Y · Z) = {y@z | y z. y ∈ Y ∧ z ∈ Z}" by simp
then have "X · (Y · Z) = {x@n | x n. x ∈ X ∧ n ∈ (Y · Z)}" by simp
then have "... = {x@y@z | x y z. x ∈ X ∧ y ∈ Y ∧ z ∈ Z}" using yz by auto
also have xy: "(X · Y) = {x@y | x y. x ∈ X ∧ y ∈ Y}" by simp
then have "(X · Y) · Z = {n@z | n z. n ∈ (X · Y) ∧ z ∈ Z}" by simp
then have "... = {x@y@z | x y z. x ∈ X ∧ y ∈ Y ∧ z ∈ Z}" using xy
    by (smt Collect_cong mem_Collect_eq string_assoc)
finally show ?thesis by simp
qed

lemma product_iden: "X · {[]} = X ∧ X = {[]} · X"
    by simp

lemma product_union_l: "X · ⋃𝒴 = ⋃{(X · Y) | Y. Y ∈ 𝒴}"
    by auto

lemma product_union_r: "⋃𝒳 · Y = ⋃{(X · Y) | X. X ∈ 𝒳}"
    by auto
```

---

We now have everything we will need in order to prove commutativity of the shuffle product.

We proceed as follows.

---

```
- ‹Commutativity of the shuffle product, for strings and then languages. We
   prove it by induction, with not too much difficulty.›
lemma shuffle_commu: "x ⧢ y = y ⧢ x"
proof(induction x arbitrary: y)
    case Nil
    then show ?case by simp
next
    case (Cons a x)
```

```
    then show ?case
    proof(induction y)
    case Nil
    then show ?case by simp
    next
    case (Cons b y)
    then have "(a#x) ⧢ (b#y) = ({[a]} · (x ⧢ (b#y))) ∪ ({[b]} · ((a#x) ⧢ y))"
      by simp
    then have ex: "... = ({[a]} · ((b#y) ⧢ x)) ∪ ({[b]} · (y ⧢ (a#x)))" using Cons
      by simp
    then have "... = (b#y) ⧢ (a#x)" using union_commu by simp
    then show ?case using ex by simp
    qed
qed

lemma shuffle_lang_commu: "X ⧢⧢ Y = Y ⧢⧢ X"
proof -
    have "X ⧢⧢ Y = ⋃{x ⧢ y | x y. x ∈ X ∧ y ∈ Y}" by simp
    then have "∀ x y. x ⧢ y = y ⧢ x" by (simp add: shuffle_commu)
    then show ?thesis by auto
qed
```

The proof of associativity is very similar in structure, although it requires three nested inductions rather than two. However, it is significantly lengthier and more complex, so it can be found in Appendix C.

We can go on to prove statements about the interchange between the shuffle product and the language product, showing how they distribute over one another. A couple of examples of this are given below.

```
- ‹Proof that concatenation is included in the set of shuffles.›
lemma product_shuffle_str: "x@y ∈ x ⧢ y"
proof(induction x arbitrary: y)
    case Nil
    then show ?case by simp
next
    case (Cons a x)
    then show ?case
    proof(induction y)
    case Nil
```

```
    then show ?case by simp
    next
    case (Cons b y)
    then have "(a # x) ⧢ (b # y) = ({[a]} · (x ⧢ (b#y)))
      ∪ ({[b]} · ((a#x) ⧢ y))" by simp
    then have "({[a]} · (x ⧢ (b#y))) ⊆ (a # x) ⧢ (b # y)" by simp
    then have "{[a]@p | p. p ∈ (x ⧢ (b#y))} ⊆ (a # x) ⧢ (b # y)" by simp
    then have "{[a]@(x@(b#y))} ⊆ (a # x) ⧢ (b # y)" using Cons by simp
    then show ?case by simp
    qed
qed

- ‹A more complex proof, showing that shuffles starting with the first element
   are included in the set of all shuffles.›
lemma prod_shuf_l_str: "{x} · (y ⧢ z) ⊆ (x @ y) ⧢ z"
proof(induction x arbitrary: y z)
    case Nil
    then show ?case by simp
next
    case ax: (Cons a x)
    then show ?case
    proof(induction z arbitrary: y)
    case Nil
    then show ?case by auto
    next
    case (Cons c z)
    then show ?case
    proof(induction y)
        case Nil
        then have "((a#x) @ []) ⧢ (c#z) = (a#x) ⧢ (c#z)" by simp
        also have "{a#x} · ([] ⧢ (c#z)) = {a#x} · {c#z}" by simp
        then have "... = {a#x@c#z}" by simp
        then show ?case using product_shuffle_str by auto
    next
        case (Cons b y)
        then have "((a#x) @ (b#y)) ⧢ (c#z) =
        ({[a]} · ((x@(b#y)) ⧢ (c#z))) ∪ ({[c]} · ((a#x@b#y) ⧢ z))" by simp
        then have "({[a]} · ((x@(b#y)) ⧢ (c#z))) ⊆ ((a#x) @ (b#y)) ⧢ (c#z)"
          by simp
        then have "({[a]} · ((x@(b#y)) ⧢ (c#z))) =
          {a#p | p. p ∈ ((x@(b#y)) ⧢ (c#z))}" by simp
        moreover have "{x} · ((b#y) ⧢ (c#z)) ⊆ ((x@(b#y)) ⧢ (c#z))"
          using ax by blast
        then have "{a#p | p. p ∈ ({x} · ((b#y) ⧢ (c#z)))} ⊆
          {a#p | p. p ∈ ((x@(b#y)) ⧢ (c#z))}"
```

```
            by blast
        ultimately have "{a#p | p. p ∈ ({x} · ((b#y) ⧢ (c#z)))} ⊆
          ((a#x) @ (b#y)) ⧢ (c#z)"
            by (simp add: order_trans)
        also have "{a#p | p. p ∈ ({x} · ((b#y) ⧢ (c#z)))} =
          {a#p | p. p ∈ {x@q | q. q ∈ ((b#y) ⧢ (c#z))}}" by simp
        then have "{a#p | p. p ∈ ({x} · ((b#y) ⧢ (c#z)))} =
          {a#(x@q) | q. q ∈ ((b#y) ⧢ (c#z))}" by blast
        ultimately have "{a#(x@q) | q. q ∈ ((b#y) ⧢ (c#z))} ⊆
          ((a#x) @ (b#y)) ⧢ (c#z)" by simp
        then show ?case by simp
    qed
  qed
qed
```

All of these proofs that apply to strings can also be generalised so that they apply to languages. An example of this was given earlier where we extended the proof of associativity to the more general case. Another example follows, although not every proof previously given will be extended in this way as they are all fairly similar.

```
lemma product_shuffle_lang: "X · Y ⊆ X ⧢⧢ Y"
proof -
    have "X ⧢⧢ Y = ⋃ {x ⧢ y | x y. x ∈ X ∧ y ∈ Y}" by simp
    also have "∀ x y. x @ y ∈ x ⧢ y" by (simp add: product_shuffle_str)
    ultimately have "{x @ y | x y. x ∈ X ∧ y ∈ Y} ⊆
      ⋃ {x ⧢ y | x y. x ∈ X ∧ y ∈ Y}" by blast
    then show ?thesis by simp
qed
```

# Chapter 6

# Discussion

## 6.1 Achievements

Overall, the project has been very successful, as it achieved many of the objectives that were set out originally. Functional data structures were defined within Isabelle from the most basic possible entities like natural numbers on to complex mathematical structures like shuffle algebras, and statements about them were verified starting with basic properties to show the correctness of the definitions and in some cases moving on to verifying the correctness of algorithms involving the specified structures. Although the programs that were proved correct could not really be considered as real world software systems with industrial applications, all functional programs are built up using simple structures like this, so the work done thus far provides a rich groundwork that could be built upon in the future.

A lot of time over the course of the project was spent becoming familiar with the way Isabelle works, so it got off to a slow start; as such, the proofs towards the beginning were quite basic and mostly done to get used to the process of using a proof assistant. However, some of the later proofs were fairly complex; some of the results relating to shuffle algebras were very challenging to achieve, and took a great deal of thought and experimentation. Although proving some of these statements was a difficult task, this undertaking gave a thorough and realistic experience of constructing a functional definition from a given specification and using Isabelle to begin to

verify its properties.

Verifying the correctness of functional programs was definitely an easier task than attempting to verify the correctness of imperative programs would have been. The data structures being defined in a functional way allowed the vast majority of the proofs to be carried out using structural induction and basic case analysis, whereas proofs involving imperative programs tend to be much more complex and require the careful application of loop invariants and Hoare logic. Also, the use of declarative Isar proofs rather than the deprecated procedural style had many benefits; it took some time to get used to the new way of writing, but the proofs are not particularly more complicated than procedural proofs and are much easier to comprehend after the fact, so developing the proofs further at a later date or fixing mistakes is less of an issue.

## 6.2 Further Work

There are many opportunities for further work presented by the results of this project. Perhaps one of the most salient is the idea of program derivation; in this project, the approach taken was to write programs and then develop proofs to ensure that they conform to the necessary specifications, but this can result in proofs being more difficult to achieve than necessary and if the program is incorrect in some way it is possible to spend a long time trying to finish a proof that is in fact impossible. Being able to directly derive a program from its specification using mathematical techniques would largely solve these problems; one possible technique for doing so is the previously mentioned Bird-Meertens formalism.

Another route for further work would of course be extending the work done towards proving statements about basic functional data structures, and proving more complex theorems about them. There are many paths that could be examined here; one example could be proving facts about infinite lists and trees of the sort that can be easily defined in Haskell due to its lazy evaluation. As Isabelle natively requires all functions to be total in order to make program verification viable, defining these infinite structures requires the use of coinductive data types, and indeed the Archive of Formal Proofs includes an article on "general-purpose coinductive data types and sets" with one instance being "coinductive lists, i.e. lazy lists or streams." [24]

# Chapter 7

# Conclusions

Isabelle has been used in the past to formalise a significant number of theorems from fields in both pure mathematics and theoretical computer science; its applications range all the way from verifying the consistency of major results like Gödel's completeness theorem and the prime number theorem to more practical results like proving the correctness of security protocols or properties of the semantics of particular programming languages. The possible value of formal methods for being able to specify, develop and verify both software and hardware systems cannot be overstated, and the continued growth of functional programming as a paradigm coupled with languages with strong type systems and proof assistant features means that formal verification is easier and more practical than ever.

Over the course of this project, progress was made starting from a point of having no experience with Isabelle or indeed proof assistants in general, moving on to proving basic statements about functional data structures like natural numbers, lists, and trees, and culminating in defining data types for mathematical structures like shuffle algebras based on their specification and then verifying known theorems about them. As has been stated there is great potential for future work to build upon these achievements and extend them, working towards being able to verify the correctness of practical real world functional software systems and even to derive them directly from specifications and thus be certain of their properties from the outset.

# References


[1] D. Potts, R. Bourquin, L. Andresen, J. Andronick, G. Klein, and G. Heiser, "Mathematically verified software kernels: Raising the bar for high assurance implementations," tech. rep., General Dynamics C4 Systems, NICTA, 2014.

[2] G. Gonthier, "Formal proof–the four-color theorem," *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008.

[3] Isabelle, "Overview." `https://isabelle.in.tum.de/overview.html`, Last accessed 01/05/2019.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 207–220, ACM, 2009.

[5] R. Bird and P. Wadler, *An Introduction to Functional Programming*. Prentice Hall International Series in Computer Science, Prentice Hall, 1988.

[6] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2016.

[7] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[8] S. Marlow, "Fighting spam with Haskell." `https://code.fb.com/security/fighting-spam-with-haskell/`, Last accessed 01/05/2019.

[9] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[10] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[11] R. Backhouse, "An exploration of the Bird-Meertens formalism," 1988.

[12] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[13] S. Kanav, P. Lammich, and A. Popescu, "A conference management system with verified document confidentiality," in *International Conference on Computer Aided Verification*, pp. 167–183, Springer, 2014.

[14] T. Nipkow, "Programming and proving in Isabelle/HOL," 2018.

[15] Archive of Formal Proofs, "Submission guidelines." `https://www.isa-afp.org/submitting.html`, Last accessed 01/05/2019.

[16] B. Huffman, J. Matthews, and P. White, "Axiomatic constructor classes in Isabelle/HOLCF," in *International Conference on Theorem Proving in Higher Order Logics*, pp. 147–162, Springer, 2005.

[17] M. Wenzel *et al.*, "The Isabelle/Isar reference manual," 2018.

[18] H. Barendregt, "The quest for correctness," *Images of SMC Research*, pp. 39–58, 1996.

[19] J. C. Blanchette, "Picking nits," 2018.

[20] J. C. Blanchette and L. C. Paulson, "Hammering away," 2018.

[21] D. Burton, *Elementary number theory*. Allyn & Bacon, Incorporated, 1976.

[22] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[23] M. Lothaire, *Combinatorics on Words*. Cambridge Mathematical Library, Cambridge University Press, 1997.

[24] A. Lochbihler, "Coinductive," *Archive of Formal Proofs*, vol. 2010, 2010.

[25] F. Haftmann and L. Bulwahn, "Code generation from Isabelle/HOL theories," 2018.

# Appendix A

# Prime Numbers and Divisibility

For some statements, it is perhaps surprisingly true that it is much easier to prove them in Isabelle than it would be to prove them by hand. This is demonstrated quite well in the following case.

```
- ‹As a brief aside, we can also prove statements about mathematical objects
   such as prime numbers. Here we define divisibility and what it means for a
   number to be prime.›
fun modulo :: "mynat ⇒ mynat ⇒ mynat" where
"modulo Zero m = Zero" |
"modulo (Suc n) m =
  (if Suc (modulo n m) = m then Zero else Suc (modulo n m))"

fun divides :: "mynat ⇒ mynat ⇒ bool" where
"divides n m = (modulo m n = Zero)"

fun leq :: "mynat ⇒ mynat ⇒ bool" where
"leq Zero m = True" |
"leq n Zero = False" |
"leq (Suc n) (Suc m) = leq n m"

fun prime :: "mynat ⇒ bool" where
"prime (Suc Zero) = False" |
"prime p = (∀k. (divides k p ⟶ (k = (Suc Zero) ∨ k = p)))"

- ‹Isabelle can then prove that a given number is prime easily, as it is just a
   case of simplification using the provided definitions.›
theorem five_prime: "prime (Suc (Suc (Suc (Suc (Suc Zero)))))"
    by simp
```

Evidently it took some work to give the definitions and state what it means for a number to be prime formally, but after this work was done, proving that a particular number is prime was just a case of applying the simplifier. In general, proof assistants can prove statements about particular values like this very easily in comparison to humans, as the method of going about such a proof is a simple combinatorial process involving building and checking a large number of derivation trees which can be carried out computationally at high speeds on a modern system.

# Appendix B

# Deletion from Binary Search Trees

```
- ‹Now we define a function allowing us to delete an element from a binary
   search tree. This is more difficult than insertion, as in the case where we
   need to delete the root element the tree must be rearranged to preserve the
   search tree property.›
fun getmax :: "nat tree ⇒ nat" where
"getmax Empty = undefined" |
"getmax (Node x l Empty) = x" |
"getmax (Node x l r) = getmax r"

fun deletemax :: "nat tree ⇒ nat tree" where
"deletemax Empty = undefined" |
"deletemax (Node x l Empty) = l" |
"deletemax (Node x l r) = Node x l (deletemax r)"

fun delete :: "nat ⇒ nat tree ⇒ nat tree" where
"delete _ Empty = Empty" |
"delete x (Node y l Empty) = (if x = y then l else delete x l)" |
"delete x (Node y Empty r) = (if x = y then r else delete x r)" |
"delete x (Node y l r) = (if x < y then (Node y (delete x l) r)
                    else (if x > y then (Node y l (delete x r))
                    else (Node (getmax l) (deletemax l) r)))"

- ‹To prove correctness we must prove the same three properties as before, but
   as the function was more difficult to define the proofs will be
   correspondingly more difficult to achieve.›
lemma getmax_greater: "t ≠ Empty ⟹ greater x t ⟹ x ≥ getmax t"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node y l r)
    then show ?case
    by (metis getmax.simps(2) getmax.simps(3) greater.simps(2) tree.exhaust)
qed

lemma deletemax_greater: "t ≠ Empty ⟹ greater x t ⟹
  greater x (deletemax t)"
proof(induction t)
    case Empty
    then show ?case by simp
```

```
next
    case (Node y l r)
    then show ?case
    by (metis deletemax.simps(2) deletemax.simps(3) greater.elims(2)
      greater.simps(2))
qed

- ‹Here we prove a necessary lemma, stating that if an element x is greater than
   all elements in a tree, then it will still be greater than all elements after
   something is deleted from the tree.›
lemma delete_greater: "greater x t ⟹ greater x (delete y t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node z l r)
    then show ?case
    proof(cases "y = z")
- ‹This proof requires an in-depth case analysis. We need to cover cases where
   the element is or is not the root of the tree, cases where the right subtree
   is or is not empty, and cases where the left subtree is or is not empty. Some
   of these cases require the use of SMT (satisfiability modulo theories)
   solvers.›
    case eq: True
    then show ?thesis
    proof(cases "r = Empty")
        case True
        then show ?thesis using Node by simp
    next
        case rn: False
        then show ?thesis
        proof(cases "l = Empty")
        case True
        then show ?thesis
            by (metis rn Node.IH(2) Node.prems delete.simps(3) greater.simps(2)
              sum_tree.cases)
        next
        case ln: False
        then have "delete y (Node z l r) = Node (getmax l) (deletemax l) r"
          using eq rn
            by (smt Node.prems delete.simps(4) greater.elims(2) greater.simps(2)
              less_not_refl)
        then show ?thesis
          using Node.prems deletemax_greater getmax_greater ln by auto
        qed
    qed
    next
    case neq: False
    then show ?thesis
    proof(cases "r = Empty")
        case True
        then show ?thesis using Node by simp
    next
        case rn: False
        then show ?thesis
        proof(cases "l = Empty")
        case True
        then show ?thesis using rn
```

```
            by (metis Node.IH(2) Node.prems delete.simps(3) greater.elims(2)
              greater.simps(2))
        next
        case False
        then show ?thesis using rn
            by (smt Node.IH(1) Node.IH(2) Node.prems delete.simps(4)
              deletemax_greater getmax_greater greater.elims(2)
                greater.simps(2))
        qed
    qed
    qed
qed

lemma getmax_smaller: "t ≠ Empty ⟹ smaller x t ⟹ x ≤ getmax t"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node y l r)
    then show ?case
    by (metis getmax.simps(2) getmax.simps(3) smaller.simps(2) tree.exhaust)
qed

lemma deletemax_smaller: "t ≠ Empty ⟹ smaller x t ⟹ smaller x (deletemax t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node y l r)
    then show ?case
    by (metis deletemax.simps(2) deletemax.simps(3) smaller.elims(2)
      smaller.simps(2))
qed

- ‹The symmetrical case where if an element is smaller than all elements in a
   given tree it remains smaller after an element is deleted is given as
   follows, in much the same way.›
lemma delete_smaller: "smaller x t ⟹ smaller x (delete y t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node z l r)
    then show ?case
    proof(cases "y = z")
    case eq: True
    then show ?thesis
    proof(cases "r = Empty")
        case True
        then show ?thesis using Node by simp
    next
        case rn: False
        then show ?thesis
        proof(cases "l = Empty")
        case True
        then show ?thesis
            by (metis rn Node.IH(2) Node.prems delete.simps(3) smaller.simps(2)
              sum_tree.cases)
```

```
        next
        case ln: False
        then have "delete y (Node z l r) = Node (getmax l) (deletemax l) r"
          using eq rn by (smt Node.prems delete.simps(4) smaller.elims(2)
            smaller.simps(2) less_not_refl)
        then show ?thesis
          using Node.prems deletemax_smaller getmax_smaller ln by auto
        qed
    qed
    next
    case neq: False
    then show ?thesis
    proof(cases "r = Empty")
        case True
        then show ?thesis using Node by simp
    next
        case rn: False
        then show ?thesis
        proof(cases "l = Empty")
        case True
        then show ?thesis using rn
            by (metis Node.IH(2) Node.prems delete.simps(3) smaller.elims(2)
              smaller.simps(2))
        next
        case False
        then show ?thesis using rn
            by (smt Node.IH(1) Node.IH(2) Node.prems delete.simps(4)
              deletemax_smaller getmax_smaller smaller.elims(2)
                smaller.simps(2))
        qed
    qed
    qed
qed

lemma greater_trans: "greater x t ⟹ y ≥ x ⟹ greater y t"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node z l r)
    then show ?case by simp
qed

lemma getmax_correct: "search t ⟹ greater (getmax t) t"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node x l r)
    then show ?case
    proof(cases "r = Empty")
    case True
    then have "getmax (Node x l r) = x" by simp
    then show ?thesis using Node True by simp
    next
    case False
    then have r: "getmax (Node x l r) = getmax r"
      by (metis getmax.simps(3) tree.exhaust)
```

```
    then have "greater (getmax r) r" using Node.IH(2) Node.prems search.simps(2)
      by blast
    also have "getmax r ≥ x ∧ greater x l"
      using False Node.prems getmax_smaller search.simps(2) by blast
    ultimately show ?thesis using greater_trans r by auto
    qed
qed

lemma greater_deletemax: "t ≠ Empty ⟹ search t ⟹
  greater (getmax t) (deletemax t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node x l r)
    then show ?case using deletemax_greater getmax_correct by blast
qed

lemma deletemax_search: "t ≠ Empty ⟹ search t ⟹ search (deletemax t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node y l r)
    then show ?case
      by (metis deletemax.simps(2) deletemax.simps(3) deletemax_smaller
        search.simps(2) smaller.elims(2))
qed

lemma smaller_getmax_left: "l ≠ Empty ⟹ search (Node x l r) ⟹
  smaller (getmax l) r"
proof(induction r)
    case Empty
    then show ?case by simp
next
    case (Node y n m)
    then show ?case using getmax_greater greater_trans by auto
qed

- ‹With some additional lemmas this gives us what we need to prove the
   correctness of the deletion function, although this proof is still quite
   difficult and requires a thorough case analysis and the use of a variety of
   Isabelle's automated tools.›
theorem delete_search: "search t ⟹ search (delete x t)"
proof(induction t)
    case Empty
    then show ?case by simp
next
    case (Node y l r)
    then show ?case
    proof(cases "x = y")
    case eq: True
    then show ?thesis
    proof(cases "r = Empty")
        case re: True
        then have "delete x (Node y l r) = l" using eq by simp
        then show ?thesis using Node by simp
    next
```

```
        case rn: False
        then show ?thesis
        proof(cases "l = Empty")
        case True
        then have "delete x (Node y l r) = r"
          by (metis delete.simps(3) eq rn sum_tree.cases)
        then show ?thesis using Node by simp
        next
        case ln: False
        then have ra: "delete x (Node y l r) =
          (Node (getmax l) (deletemax l) r)" using eq rn by (smt Node.prems
            delete.simps(4) greater.elims(2) less_not_refl search.simps(2)
              smaller.elims(2))
        then have "greater (getmax l) (deletemax l)"
            using Node.prems greater_deletemax ln search.simps(2) by blast
        also have "smaller (getmax l) r" using Node.prems ln smaller_getmax_left
          by blast
        ultimately show ?thesis using Node.prems deletemax_search ln ra by simp
        qed
    qed
    next
    case neq: False
    then show ?thesis
    proof(cases "r = Empty")
        case True
        then have "delete x (Node y l r) = delete x l" using neq by simp
        then show ?thesis using Node by simp
    next
        case rn: False
        then show ?thesis
        proof(cases "l = Empty")
        case True
        then have "delete x (Node y l r) = delete x r"
            by (metis delete.simps(3) neq rn sum_tree.cases)
        then show ?thesis using Node by simp
        next
        case ln: False
        then show ?thesis
        proof(cases "x < y")
            case True
            then have "delete x (Node y l r) = Node y (delete x l) r"
              using ln rn by (smt Node.prems delete.simps(4) greater.elims(2)
                  search.simps(2) smaller.elims(2))
            then show ?thesis using Node.IH(1) Node.prems delete_greater by auto
        next
            case gr: False
            then have "x > y" using nat_neq_iff neq by blast
            then have "delete x (Node y l r) = Node y l (delete x r)"
              using ln rn gr by (smt Node.prems delete.simps(4) greater.elims(2)
                  search.simps(2) smaller.elims(2))
            then show ?thesis using Node.IH(2) Node.prems delete_smaller by auto
        qed
      qed
    qed
  qed
qed
```

# Appendix C

# Associativity of the Shuffle Product

```
- ‹Associativity for the shuffle product. This proof is much more complicated,
   and as yet is still unfinished.›
lemma shuffle_assoc: "{x} ||| (y || z) = (x || y) ||| {z}"
proof(induction x arbitrary: y z)
    case Nil
    then show ?case by auto
next
    case (Cons a x)
    then show ?case
    proof(induction y arbitrary: z)
    case Nil
    then show ?case by simp
    next
    case (Cons b y)
    then show ?case
    proof(induction z)
        case Nil
        then show ?case by auto
    next
        case (Cons c z)
        have "(a#x) || (b#y) = ({[a]} · (x || (b#y))) ∪ ({[b]} · ((a#x) || y))"
          by simp
        then have "((a#x) || (b#y)) ||| {c#z} =
        (({[a]} · (x || (b#y))) ∪ ({[b]} · ((a#x) || y))) ||| {c#z}" by simp
        then have "... = (({[a]} · (x || (b#y))) ||| {c#z}) ∪
          (({[b]} · ((a#x) || y))) ||| {c#z}" using lang_shuffle_union_2 by blast
        then have split_r: "((a#x) || (b#y)) ||| {c#z} =
          ({a#p | p. p ∈ (x || (b#y))} ||| {c#z}
         ∪ {b#p | p. p ∈ ((a#x) || y)} ||| {c#z})" by simp
        then have "{a#p | p. p ∈ (x || (b#y))} ||| {c#z} =
          ⋃{q || (c#z) | q. q ∈ {a#p | p. p ∈ (x || (b#y))}}" by auto
        then have "{a#p | p. p ∈ (x || (b#y))} ||| {c#z} =
          ⋃{(a#p) || (c#z) | p. p ∈ (x || (b#y))}" by blast
```

```
  then have "{a#p | p. p ∈ (x ⧢ (b#y))} ⧢⧢ {c#z} =
    ⋃{({[a]} · (p ⧢ (c#z))) ∪ ({[c]} · ((a#p) ⧢ z)) | p. p ∈ (x ⧢ (b#y))}"
      by simp
  then have "{a#p | p. p ∈ (x ⧢ (b#y))} ⧢⧢ {c#z} =
    ⋃{({[a]} · (p ⧢ (c#z))) | p. p ∈ (x ⧢ (b#y))}
  ∪ ⋃{({[c]} · ((a#p) ⧢ z)) | p. p ∈ (x ⧢ (b#y))}"
      by (simp add: Setcompr_eq_image UN_Un_distrib)
  then have "{a#p | p. p ∈ (x ⧢ (b#y))} ⧢⧢ {c#z} =
    ⋃{{a#q | q. q ∈ (p ⧢ (c#z))} | p. p ∈ (x ⧢ (b#y))}
  ∪ ⋃{{c#q | q. q ∈ ((a#p) ⧢ z)} | p. p ∈ (x ⧢ (b#y))}" by simp
  also have "{b#p | p. p ∈ ((a#x) ⧢ y)} ⧢⧢ {c#z} =
    ⋃{q ⧢ (c#z) | q. q ∈ {b#p | p. p ∈ ((a#x) ⧢ y)}}" by auto
  then have "{b#p | p. p ∈ ((a#x) ⧢ y)} ⧢⧢ {c#z} =
    ⋃{(b#p) ⧢ (c#z) | p. p ∈ ((a#x) ⧢ y)}" by blast
  then have "{b#p | p. p ∈ ((a#x) ⧢ y)} ⧢⧢ {c#z} =
    ⋃{({[b]} · (p ⧢ (c#z)))
    ∪ ({[c]} · ((b#p) ⧢ z)) | p. p ∈ ((a#x) ⧢ y)}" by simp
  then have "{b#p | p. p ∈ ((a#x) ⧢ y)} ⧢⧢ {c#z} =
    ⋃{({[b]} · (p ⧢ (c#z))) | p. p ∈ ((a#x) ⧢ y)}
  ∪ ⋃{({[c]} · ((b#p) ⧢ z)) | p. p ∈ ((a#x) ⧢ y)}"
      by (simp add: Setcompr_eq_image UN_Un_distrib)
  then have "{b#p | p. p ∈ ((a#x) ⧢ y)} ⧢⧢ {c#z} =
    ⋃{{b#q | q. q ∈ (p ⧢ (c#z))} | p. p ∈ ((a#x) ⧢ y)}
    ∪ ⋃{{c#q | q. q ∈ ((b#p) ⧢ z)} | p. p ∈ ((a#x) ⧢ y)}" by simp
  ultimately have rhs: "((a#x) ⧢ (b#y)) ⧢⧢ {c#z} =
    ⋃{{a#q | q. q ∈ (p ⧢ (c#z))} | p. p ∈ (x ⧢ (b#y))}
  ∪ ⋃{{c#q | q. q ∈ ((a#p) ⧢ z)} | p. p ∈ (x ⧢ (b#y))}
  ∪ ⋃{{b#q | q. q ∈ (p ⧢ (c#z))} | p. p ∈ ((a#x) ⧢ y)}
  ∪ ⋃{{c#q | q. q ∈ ((b#p) ⧢ z)} | p. p ∈ ((a#x) ⧢ y)}" using split_r
    by auto
  also have "(b#y) ⧢ (c#z) = ({[b]} · (y ⧢ (c#z))) ∪ ({[c]} · ((b#y) ⧢ z))"
    by simp
  then have "{a#x} ⧢⧢ ((b#y) ⧢ (c#z)) =
    {a#x} ⧢⧢ (({[b]} · (y ⧢ (c#z))) ∪ ({[c]} · ((b#y) ⧢ z)))" by simp
  then have "{a#x} ⧢⧢ ((b#y) ⧢ (c#z)) =
    ({a#x} ⧢⧢ ({[b]} · (y ⧢ (c#z)))) ∪ ({a#x} ⧢⧢ ({[c]} · ((b#y) ⧢ z)))"
  by (metis lang_shuffle_union_2 shuffle_lang_commu)
  then have split_l: "{a#x} ⧢⧢ ((b#y) ⧢ (c#z)) =
    ({a#x} ⧢⧢ {b#p | p. p ∈ (y ⧢ (c#z))})
  ∪ ({a#x} ⧢⧢ {c#p | p. p ∈ ((b#y) ⧢ z)})" by simp
  then have "{a#x} ⧢⧢ {b#p | p. p ∈ (y ⧢ (c#z))} =
    ⋃{(a#x) ⧢ q | q. q ∈ {b#p | p. p ∈ (y ⧢ (c#z))}}" by auto
  then have "{a#x} ⧢⧢ {b#p | p. p ∈ (y ⧢ (c#z))} =
    ⋃{(a#x) ⧢ (b#p) | p. p ∈ (y ⧢ (c#z))}" by blast
  then have "{a#x} ⧢⧢ {b#p | p. p ∈ (y ⧢ (c#z))} =
    ⋃{({[a]} · (x ⧢ (b#p))) ∪ ({[b]} · ((a#x) ⧢ p)) | p. p ∈ (y ⧢ (c#z))}"
      by simp
  then have "{a#x} ⧢⧢ {b#p | p. p ∈ (y ⧢ (c#z))} =
```

```
          ⋃{{[a]} · (x ‖ (b#p)) | p. p ∈ (y ‖ (c#z))}
        ∪ ⋃{{[b]} · ((a#x) ‖ p) | p. p ∈ (y ‖ (c#z))}"
            by (simp add: Setcompr_eq_image UN_Un_distrib)
        then have "{a#x} ||| {b#p | p. p ∈ (y ‖ (c#z))} =
          ⋃{{a#q | q. q ∈ (x ‖ (b#p))} | p. p ∈ (y ‖ (c#z))}
        ∪ ⋃{{b#q | q. q ∈ ((a#x) ‖ p)} | p. p ∈ (y ‖ (c#z))}" by simp
        moreover have "{a#x} ||| {c#p | p. p ∈ ((b#y) ‖ z)} =
          ⋃{(a#x) ‖ q | q. q ∈ {c#p | p. p ∈ ((b#y) ‖ z)}}" by auto
        then have "{a#x} ||| {c#p | p. p ∈ ((b#y) ‖ z)} =
          ⋃{(a#x) ‖ (c#p) | p. p ∈ ((b#y) ‖ z)}" by blast
        then have "{a#x} ||| {c#p | p. p ∈ ((b#y) ‖ z)} =
          ⋃{({[a]} · (x ‖ (c#p))) ∪ ({[c]} · ((a#x) ‖ p)) | p. p ∈ ((b#y) ‖ z)}"
            by simp
        then have "{a#x} ||| {c#p | p. p ∈ ((b#y) ‖ z)} =
          ⋃{{[a]} · (x ‖ (c#p)) | p. p ∈ ((b#y) ‖ z)}
        ∪ ⋃{{[c]} · ((a#x) ‖ p) | p. p ∈ ((b#y) ‖ z)}"
            by (simp add: Setcompr_eq_image UN_Un_distrib)
        then have "{a#x} ||| {c#p | p. p ∈ ((b#y) ‖ z)} =
          ⋃{{a#q | q. q ∈ (x ‖ (c#p))} | p. p ∈ ((b#y) ‖ z)}
        ∪ ⋃{{c#q | q. q ∈ ((a#x) ‖ p)} | p. p ∈ ((b#y) ‖ z)}" by simp
        ultimately have lhs: "{a#x} ||| ((b#y) ‖ (c#z)) =
          ⋃{{a#q | q. q ∈ (x ‖ (b#p))} | p. p ∈ (y ‖ (c#z))}
        ∪ ⋃{{b#q | q. q ∈ ((a#x) ‖ p)} | p. p ∈ (y ‖ (c#z))}
        ∪ ⋃{{a#q | q. q ∈ (x ‖ (c#p))} | p. p ∈ ((b#y) ‖ z)}
        ∪ ⋃{{c#q | q. q ∈ ((a#x) ‖ p)} | p. p ∈ ((b#y) ‖ z)}"
            using split_l by auto
        then show ?case sorry
    qed
  qed
qed

- ‹Extension of associativity to languages, also unfinished.›
lemma lang_shuffle_assoc: "X ||| (Y ||| Z) = (X ||| Y) ||| Z"
proof -
  have "X ||| (Y ||| Z) = ⋃{{x} ||| (Y ||| Z) | x. x ∈ X}"
    by (metis lang_shuffle_set_l)
  then have "... = ⋃{{x} ||| ⋃{y ‖ z | y z. y ∈ Y ∧ z ∈ Z} | x. x ∈ X}" by simp
  then have "X ||| (Y ||| Z) = ⋃{{x} ||| (y ‖ z) | x y z. x ∈ X ∧ y ∈ Y ∧ z ∈ Z}"
    sorry
  also have "(X ||| Y) ||| Z = ⋃{x ‖ y | x y. x ∈ X ∧ y ∈ Y} ||| Z" by simp
  then have "... = ⋃{Z ||| P | P. P ∈ {x ‖ y | x y. x ∈ X ∧ y ∈ Y}}"
    by (metis lang_shuffle_union shuffle_lang_commu)
  then have "(X ||| Y) ||| Z =  ⋃{{z} ||| (x ‖ y) | x y z. x ∈ X ∧ y ∈ Y ∧ z ∈ Z}"
    sorry
  ultimately show ?thesis by (smt shuffle_lang_commu Collect_cong shuffle_assoc)
qed
```

# Appendix D

# Shuffle Algebras in Haskell

One possible concern could be that transferring data structures and functions developed in Isabelle into another functional language, whilst being less difficult than rewriting them in an imperative style, would still take some time and effort, and presents the risk that errors and inaccuracies could be introduced along the way, making all the work of verification ultimately irrelevant. However, Isabelle offers powerful code generation tools, allowing for conversion from "HOL specifications into corresponding executable code in the programming languages SML, OCaml, Haskell and Scala." [25] An example of this is given below, showing the result of generating Haskell code from the shuffle algebras theory file.

---

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module Shuffle(Set, product, shuffle, shuffleproduct) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/), (**),
    (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++), (!!), Eq,
    error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
    zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
    String, Bool(True, False), Maybe(Nothing, Just));
import qualified Prelude;

data Set a = Set [a] | Coset [a];

fold :: forall a b. (a -> b -> b) -> [a] -> b -> b;
fold f (x : xs) s = fold f xs (f x s);
fold f [] s = s;

image :: forall a b. (a -> b) -> Set a -> Set b;
image f (Set xs) = Set (map f xs);

removeAll :: forall a. (Eq a) => a -> [a] -> [a];
removeAll x [] = [];
removeAll x (y : xs) = (if x == y then removeAll x xs else y : removeAll x xs);

membera :: forall a. (Eq a) => [a] -> a -> Bool;
membera [] y = False;
membera (x : xs) y = x == y || membera xs y;
```

```
inserta :: forall a. (Eq a) => a -> [a] -> [a];
inserta x xs = (if membera xs x then xs else x : xs);

insert :: forall a. (Eq a) => a -> Set a -> Set a;
insert x (Coset xs) = Coset (removeAll x xs);
insert x (Set xs) = Set (inserta x xs);

member :: forall a. (Eq a) => a -> Set a -> Bool;
member x (Coset xs) = not (membera xs x);
member x (Set xs) = membera xs x;

remove :: forall a. (Eq a) => a -> Set a -> Set a;
remove x (Coset xs) = Coset (inserta x xs);
remove x (Set xs) = Set (removeAll x xs);

top_set :: forall a. Set a;
top_set = Coset [];

inf_set :: forall a. (Eq a) => Set a -> Set a -> Set a;
inf_set a (Coset xs) = fold remove xs a;
inf_set a (Set xs) = Set (filter (\ x -> member x a) xs);

producta :: forall a b. Set a -> Set b -> Set (a, b);
producta (Set xs) (Set ys) = Set (concatMap (\ x -> map (\ a -> (x, a)) ys) xs);

product :: forall a. (Eq a) => Set [a] -> Set [a] -> Set [a];
product x y =
    image (\ (a, b) -> a ++ b)
    (producta (inf_set (image (\ xa -> xa) x) top_set)
        (inf_set top_set (image (\ ya -> ya) y)));

sup_set :: forall a. (Eq a) => Set a -> Set a -> Set a;
sup_set (Coset xs) a = Coset (filter (\ x -> not (member x a)) xs);
sup_set (Set xs) a = fold insert xs a;

bot_set :: forall a. Set a;
bot_set = Set [];

shuffle :: forall a. (Eq a) => [a] -> [a] -> Set [a];
shuffle [] x = insert x bot_set;
shuffle (v : va) [] = insert (v : va) bot_set;
shuffle (a : x) (b : y) =
    sup_set (product (insert [a] bot_set) (shuffle x (b : y)))
    (product (insert [b] bot_set) (shuffle (a : x) y));

sup_seta :: forall a. (Eq a) => Set (Set a) -> Set a;
sup_seta (Set xs) = fold sup_set xs bot_set;

shuffleproduct :: forall a. (Eq a) => Set [a] -> Set [a] -> Set [a];
shuffleproduct x y =
    sup_seta
    (image (\ (a, b) -> shuffle a b)
        (producta (inf_set (image (\ xa -> xa) x) top_set)
        (inf_set top_set (image (\ ya -> ya) y))));

}
```