André Duarte Teixeira Trindade

Bachelor of Science

A Mechanized Proof of Kleene's Theorem in Why3

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: António Maria Lobo César Alarcão Ravara,
Associate Professor,
NOVA School of Science and Technology

Co-adviser: Mário José Parreira Pereira,
Post-Doctoral Researcher,
NOVA School of Science and Technology

Examination Committee:

Chair:	João Alexandre Carvalho Pinheiro Leite, Associate Professor with Aggregation, NOVA School of Science and Technology
Rapporteur:	Simão Melo Patrício de Sousa, Associate Professor with Aggregation, University of Beira Interior
Adviser:	António Maria Lobo César Alarcão Ravara
Member:	Mário José Parreira Pereira

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

December, 2020

A Mechanized Proof of Kleene's Theorem in Why3

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my mom. For her strength, open mind, caring heart, and unconditional support.

Acknowledgements

The work herein presented would not have been possible without the great deal of support and assistance of my adviser, Professor António Ravara, whom I thank for his invaluable guidance and helpful advice, the opportunity to be part of this project, and, above all, for believing in my potential and helping me become a better researcher.

I would also like to thank my co-adviser, Mário Pereira, for the countless hours he spent helping me overcome the difficulties I encountered along this formidable process, his practical guidance, and sympathetic ear.

Furthermore, I thank my friends, Manuel Sousa Ribeiro, Marta Tenera, and João Sousa, for their encouraging words and entertaining moments. Renata Zevallos and Romea Družetić, for always being present in the most difficult times, even across borders. And Davide Onestini, who has helped me look at life with a different perspective, encouraged me to strive to be a better person, and whose support I am incredibly grateful for.

Finally, my deepest and most sincere gratitude to my family, in particular my parents, Dalila and Duarte, for having supported this journey in every imaginable way, and a special thanks to my brother, Ricardo, and my grandmother, Adriana, whom I deeply care for.

Abstract

In this dissertation we present a mathematically minded development of the correction proof of Kleene's theorem conversion of regular expressions into finite automata, on the basis of equivalent expressive power. We formalise a functional implementation of the algorithm and prove, in full detail, the soundness of its mathematical definition, working within the Why3 framework to develop a mechanically verified implementation of the conversion algorithm. The motivation for this work is to test the feasibility of the deductive approach to the verification of software and pave the way to do similar proofs in the context of a static analysis approach to (object-oriented) programming. In particular, on the subject of behavioural types in typestate settings, whose expressiveness stands between regular and context-free languages and, therefore, can greatly benefit from mechanically certified implementations.

Keywords: Deductive verification; Formal languages;Automata theory; Regular expressions; Kleene's theorem; Why3.

Resumo

Nesta dissertação apresentamos um desenvolvimento matemático da prova de correcção da conversão de expressões regulares em autómatos finitos do teorema de Kleene, com base no seu poder expressivo equivalente. Formalizamos uma implementação funcional do algoritmo e provamos, em detalhe, a correcção da sua definição matemática. Trabalhando no framework Why3 para desenvolver uma implementação mecanicamente certificada do algoritmo de conversão. A motivação para este trabalho é testar a viabilidade da metodologia e preparar o caminho para fazer provas semelhantes no contexto de uma abordagem de análise estática na programação (orientada para objectos). Em particular, no tópico dos tipos comportamentais com *typestates*, cuja expressividade está entre a das linguagens regulares e livres-de-contexto. Podendo, por isso, beneficiar enormemente de implementações mecanicamente certificadas.

Palavras-chave: Verificação dedutiva; Linguagens formais; Teoria de autómatos; Expressões regulares; Teorema de Kleene; Why3.

Contents

CHAPTER 1

Introduction

1.1 Context

As the complexity of software increases, so does the occurrence of bugs and attacks exploiting software vulnerabilities, and as applications begin to deal with more and more sensible data, society becomes less tolerant to faults in these systems. This brings out the necessity for methods that enable the growing community of software developers and engineers to build resilient and robust systems and, specifically, the need for tools able to give guarantees about the correctness of a program.

The rampant production of code often leads to incomplete specifications and the neglect of program correctness. Programmers and companies still heavily rely on testing to raise confidence in a piece of code. Indeed, a carefully designed test suite may detect the presence of many bugs, but it does not guarantee their absence [31]. Another widespread practice is the use of programming languages with type systems. These ensure that programs do not present errors for executing invalid operations, but the type of detected errors is quite limited.

Just this year, on the 16th of February, not long before travel restrictions, travellers in the UK have experienced the consequences of software failure, when Heathrow's airport was hit by a technical problem affecting departure boards and check-in systems. Passengers were left with little to no information about their flights, and electronic tickets could not be checked-in. The issue was fixed a few hours later but, nonetheless, more than 300 flights were cancelled or grounded across Europe, with disruptions extending to the following days[1].

In a highly connected and computationally developed world, detecting software errors and vulnerabilities is becoming increasingly important. Companies are beginning to take notice of the issue and, in the last few years, the push for certified software has never

[1] https://www.theguardian.com/uk-news/2020/feb/17/heathrow-delays-continue-after-technical-glitches

been greater. Not only from industries that have been historically demanding of highly-secure systems, such as the aircraft control industry, but also mainstream companies like Facebook, with their static program analyser tool Infer [15], and the use of TLA+ for formal specification and model checking at Amazon AWS [69].

1.2 Problem

The idea of verifying programs has been around since Turing presented the first known proof of a program in the late 40s [86], and was later popularised by the works of Robert Floyd [40] and Tony Hoare [45], resulting in what is known as Hoare Logic, and Dijkstra's calculus of the weakest precondition [32]. To mathematicians the notion of proof may be ordinary, however, many who took upon the role of programming are not capable of designing a program specification through careful logical arguments.

One of the biggest challenges of program verification is the distance between formal specification and the object's behaviour it means to model and represent [64]. It is very difficult to know to which extent do the models accurately capture all the relevant aspects of the underlying system [9], which makes the process of specification all the more delicate and complex. For this reason, when certifying software, the design phase of development is much more thorough. In turn, this leads to fewer problems when debugging during test and integration phases, potentially saving time, money, and other resources.

In many science and engineering fields, such as Physics or Electronics, the use of mathematical models is very common, and so is true for the foundations of Computer Science [43]. However, it seems like over time, with the popularisation of programming languages and software development, computer scientists have lost their connection with formal approaches. Undoubtedly, theoretical computer science courses taught at universities include the study of many algorithms with a strong mathematical argument. Such is the study of formal languages, automata theory, or computational logic. Nonetheless, most of these mathematical concepts are rather informal, which presents a barrier when reasoning about possible implementations of these techniques and mechanised proofs of their correction.

1.3 State of the Art

There are currently several approaches to the assistance of programmers and engineers in the *verification* and *validation* of software, some more popular and easy to use than others. Concretely, we consider the deductive verification [36], dynamic analysis [6], and static analysis [90] of software. To better understand what differentiates them, it is also important to distinguish between verification and validation. Essentially, the process of verification is concerned with the correct specification of the program, to guarantee that it was built accordingly, consisting of structural and behavioural aspects. Whereas validation evaluates if the program is compliant with what was envisioned and, therefore, is usually a dynamic process that assesses if the software satisfies the expected use cases.

The deductive verification of software is an incredibly powerful method to express the correctness of a program. It allows programmers to design specifications by annotating the source code with a set of logical statements and, with the assistance of automatic or interactive theorem provers, prove that the implementation is correct in regards to the specification. It is, however, a very complex and fastidious process since it demands a great deal of time to ensure that the specification is both complete and sound, and, furthermore, requires programmers to have a profound knowledge of logics.

Dynamic analysis is a process that examines the properties of a procedure during runtime. While it might not be able to establish properties for the program as a whole, it can detect violations and vulnerabilities, and provide a better understanding of the program's behaviour in execution. The dependence on inputs is a double-edged sword: it easily relates changes in input to changes in the program's behaviour and output, but makes dynamic analysis incomplete. Furthermore, dynamic analysis requires building possible execution models and scenarios, which escalate with the procedure's dimension and execution time, frequently rendering the approach unreliable or even unfeasible.

In contrast, static analysis is capable of deriving properties that hold for the program as a whole, examining the source code directly, without actually executing it. By evaluating all possible execution paths and structure values, static analysis is able to avoid faults from simple programming errors or syntax violations, to security vulnerabilities, such as buffer overflows. The automatic quality of static analysis is a great advantage, especially when compared to manual code reviews and, therefore, less prone to human error, increasing the likelihood of finding software vulnerabilities. However, unlike dynamic analysis, static analysis is not able to reason about the program's behaviour when in runtime, i.e., it is not able to predict every context of execution, for example, how possible inputs might affect it. This usually leads to the paring of both techniques in software quality-control environments.

Regardless, static analysis and, although rarely, dynamic analysis may, at times, use approximation models of the properties they intend on verifying. This can lead to the detection of *false positives*, i.e., identifying violations that do not actually exist. Solving false positives requires additional review time and filtering them automatically may lead to *false negatives* (undetected violations), which pose a big concern when developing mission-critical software, often leading programmers to avoid program analysis [50].

An increasingly popular approach to static analysis is the use of *behavioural types* [4] to enrich the expressive quality of types in order to describe dynamic aspects of programs. In particular, behavioural types allow programmers to specify valid sequences of operations, for example, in terms of state-dependent availability of methods or collective communication and interaction protocols for concurrent entities. One great advantage of specifying behavioural models is the ability to guarantee protocol fidelity, i.e., respecting the established constraints on the sequence of operations. In addition, the idea of correct interaction can further guarantee both safety and liveness properties, for example, ensuring the absence of deadlocks and the eventual termination of an interaction.

1.4 Goals and Contributions

This September, we published work [85] on the subject of behavioural types, based on Mungo [28], a tool used for associating behavioural specifications — *typestates* or *usages* — with Java classes, and verifying if objects are used accordingly. These typestate specifications abstract the available operations on an object by defining the sequence of permitted method-calls, which depend on the state of the object [2, 82]. The main contribution of this work was the development of a tool to help Java programmers reason and better visualise the behaviour of objects as typestates, by designing an algorithm that, from a Mungo usage computes an equivalent state machine representation, and *vice versa*. This can be incredibly helpful in the designing phase of a system since it is quite simple to define automata to represent an entity or system. Furthermore, it simplifies the adaptation of legacy code to incorporate behavioural specifications, which is a great step in the popularisation of this approach and in increasing system trustability.

A great part of the work that led to the implementation of this tool was the definition of an automaton model — *deterministic object automaton* (DOA) — that rightfully captures the expressive capacity of Mungo typestates; the definition of a formal grammar for Mungo protocols; and, inspired by the equivalence of regular expressions and finite automata given by Kleene's Theorem [54], the definition of a method for the bi-directional translation between typestates and automata, by way of two algorithms that convert Mungo protocols into DOAs, following the production rules of the grammar, and produce a Mungo protocol by following all possible execution paths of a DOA.

It is only natural, that a tool concerned with rigour and respecting specifications should itself be certified, in order to create trust on all levels of use. Therefore, in a following stage of development, we hope to apply the deductive approach to the certification of such a tool, defining a mathematical proof of the expressive equivalence of Mungo protocols and DOAs, to develop a mechanically certified version of the tool.

In an attempt to test the viability of deductive verification and pave the way towards such a goal, and given the obvious similarities and inspiration on the conversion between regular expressions and finite automata, our efforts with this dissertation concern a constructive formal proof of the expressive equivalence of regular expressions and finite automata, based on the work of Hopcroft et al. [46]. We use it to formalize and specify a mechanisation of the mathematical results in a certified proof of the conversion algorithm within the Why3 framework [89]. Specifically, we will be using the informal definitions of the literature to design more rigorous and complete mathematical definitions of the functions and concepts necessary for the conversion from the regular expression to finite automaton. The equivalence of the two representations is obtained by a language-preserving translation. Therefore, the correctness proof of the algorithm comes down to the proof that the accepted language remains the same upon translation. Additionally, we will be implementing each of the defined mathematical concepts, including the functions and the conversion algorithm preceding the formal proof of equivalence, as

pure functional code, and specify them through careful logical assertions, in order to mechanically-check their correction following the learnings of the constructive proof, where, once again, the proof of expressive equivalence, comes down to the correction of the conversion algorithm.

All in all, this thesis proves to be an important exercise for the certification of programs. Concretely, the certification of a program for the conversion of regular expressions into finite automata. Hopefully contributing to a future where deductive verification of software becomes mainstream in the development of our everyday systems.

1.5 Outline

In Chapter 2 we discuss the concept of *formal methods*, in particular, that of deductive program verification, and their relevance in today's necessity for reliable systems, followed by an introduction to Why3's platform, and the exposition of a few relevant examples of certified projects. In Chapter 3 we review and define some of the relevant concepts of formal languages, along with the conversion algorithm for the translation from regular expressions into finite automata, and define a formal correctness proof for such compilation. This lays the groundwork for Chapter 4, where we will be implementing these concepts and algorithm as pure functional code, specifying them, and discussing their mechanically-assisted verification. For these two chapters, we also include a review of related work and discussion regarding our contributions. We conclude this document with Chapter 5, in which we analyse the developments presented in this dissertation and possible future explorations.

CHAPTER 2

Background

The general approach to programming, especially in environments that promote the rapid production of code, is to rely on empirical methods such as testing, code reviews, or informal guidelines. In particular, testing is the most widely adopted approach since it can be easily automated and can, to some extent, help building confidence in a piece of code. However, testing is not a perfect solution, as it does not guarantee the absence of bugs. To do so would require developers to be able to draft a fully-exhaustive test suit — something that no one can be certain of. If we intend on building complex systems that deal with highly sensitive information or where precision is of utmost importance, there is a special need for more thorough verification methods [47].

In this chapter, we take a look at formal methods, which try to mitigate this problem by specifying mathematical models of systems, in order to prove properties about such systems. On this topic, we consider *abstract interpretation*, *model checking*, *type systems*, *behavioural types*, and *deductive program verification*. The latter opens the way to an overview of Why3, a tool that facilitates the specification of a program and its verification, with the help of theorem provers, and the one we have chosen to work with for the developments herein presented. Finally, we review a few projects built on top of deductive verification tools, showing that applying deductive methods in the construction of real-world programs is realistic and rewarding.

2.1 Formal Methods

Formal methods [22, 64] appear as a complement to system testing rather than a solution. The use of formal methods becomes a necessity when one requires a high assurance that the system behaves as intended. The idea is that by formalising a mathematical model of the system — the *specification* — we are then able to verify the system properties and prove whether the program complies with the desired specification, ensuring its correct

behaviour. However, as testing does not guarantee the absence of errors, formal methods do not solve bad assumptions in the system's design or incomplete specifications.

Previously, in Sec. 1.3, we have discussed the differences between dynamic analysis, static analysis and deductive verification, as well as a brief introduction to behavioural types. In this section, we will have a more in-depth look at some of the approaches to program analysis, such as abstract interpretation, model checking, type systems, and behavioural types, as well as a more detailed account of deductive program verification.

2.1.1 Abstract Interpretation

Abstract interpretation [26, 27] establishes an *abstract domain* (an approximation model) in order to gain information about the semantics of a program, without performing every calculation. It maps concrete semantics to abstract semantics, whenever, at some point, concrete semantics pose an undecidable problem. These qualities make compilers one of the main applications of abstract interpretation, for example for *constant propagation* optimisation, where constant variables are replaced with literal values in order to save loads of the variables and remove their uses. However, due to the loss of precision, abstract interpretation models may result in over-approximations of the states in a program, leading to the compromise of *decidability* over *tractability*.

2.1.2 Model Checking

In model checking [9, 20, 47], a program is modelled as a finite state machine and its properties specified with temporal logic. The model checker must consider all possible behaviours of the system, to verify if the model — the system requirements — respects the specification, exploring any reachable states to verify safety and liveness properties. However, trying to predict all possible execution paths may lead to a *state explosion problem*, i.e., the state space can grow exponentially [21]. Therefore, variants of this method include *symbolic model checking* [14], in which states are represented using boolean functions, avoiding the explicit construction of state machine graphs and using binary decision diagrams instead, or *bounded model checking* [9], a technique that limits the search space to some threshold.

2.1.3 Type Systems

As simply put by Cardelli in his seminal work, Type Systems [16], "the fundamental purpose of a *type system* is to prevent the occurrence of execution errors during the running of a program". Type systems can prevent these errors and prove the absence of certain damaging behaviours by computing a *static* approximation of the runtime behaviour of a program [74]. This can be achieved by syntactically classifying program expressions according to the kind of values they can produce [71].

The fact that type systems can perform independently with very few interventions from the programmer, makes them tractable and simple to use, thus their popularity and

presence in most modern programming languages. With type systems, the programmer is able to define abstractions that help reason about concrete intuitions, and structure systems modularly, allowing interfaces to be designed independently from implementation [16, 74]. To be considered *safe*, a language must be able to guarantee that both the language's inherent abstractions and those defined by the programmer are preserved.

There are several approaches that answer to the necessity of more expressive type systems, such as *polymorphism* [63], providing the flexibility to reuse the same code fragment with different argument types, at the cost of increasing time for finding bugs; *dependent types* [61], which allow types to be parameterised by terms of the language [71] and can be used to mechanically encode mathematical proofs through logical prepositions and reason about the functional correctness of a program; or *behavioural types*, which we will be reviewing in the following section.

2.1.4 Behavioural Types

Stateful objects are non-uniform [70], i.e., their methods' availability depends on their internal state. Behavioural types [4, 48] have appeared at the need for tools that allow one to reason about the possible behaviour of an entity, such as an object or a communication channel. They do this by describing those entities in terms of constraints on the sequences of operations that allow for the correct use of involved entities.

There are several approaches to behavioural types applications. In particular, *session types* [48] in association with concurrent or distributed procedures to model the correct interaction between parties and encompass safety and liveness properties; *typestates* in object-oriented programing, to specify state-dependent availability of operations, allowing the easy representation of objects through state machines; or *choreographies*, which specify collective communication behaviour, for example, to describe the topology of communication networks [4]. These methods can prevent errors such as trying to read from a file that is not yet open, or guarantee the absence of deadlocks or that a message will eventually be received.

Tools based on behavioural types allow the programmer to statically check if the code of a program respects the intended behaviour of each entity, by defining usage protocols capturing the availability of operations. And, while the definition of these contracts requires more manual work and reasoning than current type systems, for example, in object-oriented languages such as Java, increasingly more tools are making the use of behavioural types more practical. Our tool [85] is one example, making the iterative process of program development simpler by allowing users to model objects as finite automata and turning them into typestates, which they can associate with their Java code. Scribble [91], a language used to model distributed applications through multiparty protocols, describing communication between participating parties, is able to verify if the produced protocol is safe to be implemented and automatically generate local protocols for each role (participating party), describing how the role interacts with other parties. Another

incredibly useful contribution is StMungo [28], which translates Scribble local protocols into Mungo typestate specifications for each role, based on the protocol's message flow. StMungo is then able to abstract each role as a Java class by following its generated typestate protocol, as well as a Java main class corresponding to that same role, which complies with the generated Mungo protocol.

2.1.5 Deductive Program Verification

Deductive program verification is the process of turning the correctness of a program into a set of mathematical statements [36, 71] — *verification conditions* — and prove them by applying deductive reasoning. One of the most relevant challenges of deductive verification is to understand how does one establish the connection between the program itself and its mathematical specification. In his 1949 paper, "Checking a large routine", Alan Turing presents the rigorous proof of a program that computes the factorial of a number by repeated additions [68, 86]. Turing approaches the previous issue by associating mathematical assertions to each instruction of the program and verifying each assertion individually to guarantee the correctness of the program.

Modern-day program verification, however, is most influenced by the works of Robert Floyd [40] and Tony Hoare [45]. Commonly referred to as *Hoare Logic*, their work presents a formal system with a set of logic rules that help reason about the correction of a program. The idea is to formalise the specification of a program as a logical representation of its behaviour. The specification consists of two logical formulas, a *precondition* and a *postcondition*, leading to the notion of *Hoare triple*:

$$\{P\}\ C\ \{Q\}$$

Here P, the precondition, is a predicate that states that a command C must execute in a state satisfying the conditions of P, and Q, the postcondition, states that the execution of C leads to a state that satisfies Q. Intuitively, we can read the above triple as: for all states satisfying P, the execution of C results in states satisfying Q, if C terminates.

Hoare Logic provides a set of axioms and inference rules that help specify any construct in a simple imperative programming language. Through the composition of these axioms and rules, we can reason about the validity of a triple. From a practical standpoint, to prove the validity of a triple of a program we would have to assert each of its instructions with pre and postconditions. Besides making it hard to scale correctness proofs for larger programs, annotating each and every instruction would be error-prone. Later, however, Dijkstra proposes that the programmer is only required to explicitly provide logical assertions in certain parts of the program, with the remaining intermediate assertions being inferred automatically [32]. This is done by computing the *weakest precondition* of a program C given a postcondition Q — $wp(C, Q)$ — such that the execution of C ends in a state satisfying Q. Consequently, the Hoare triple $\{wp(C, Q)\}\ C\ \{Q\}$ is valid, and, therefore, the validity of a triple of the form $\{P\}\ C\ \{Q\}$ derives from $wp(C, Q) \implies P$.

After formalising the specification of a program, one needs to prove the validity of the mathematical statements entailing that specification, i.e., prove P and Q. One approach is to draft the proof manually, similarly to what Hoare did for his proof of the *FIND* algorithm [44]. However, this would be a complex and fastidious task with a great chance of introducing errors during the proof, and probably the reason why Hoare sent the first version of *FIND*'s proof to referees [51]. Fortunately, modern-day programmers and engineers can count on *theorem provers*, tools that can be used to write and check formal proofs. On this subject, one possibility is to use an interactive proof assistant, such as Coq [8] or Isabelle [88], which requires the user to manually conduct most of the deductive reasoning. Alternatively, *automatic theorem provers* are able to automatically search for the proof of a given mathematical formula. In particular, SMT (*Set Modulo Theory*) provers, such as Alt-Ergo [10], CVC4 [7] or Z3 [29], have been very successful.

Throughout the years, other logic systems based on the foundations of Hoare Logic, such as *separation logic* [76], have been proposed with the goal of verifying a specific class of problems. Dijkstra's contribution has also played a big role in today's deductive program verification medium, with several verification tools based on the *calculus of the weakest precondition*. We can cite Dafny [58], KeY [1], VeriFast [49], and Why3 [38]. In the following section, we will be focusing on the latter, as this is the tool of choice for the procedures we will be presenting with this dissertation.

While the deductive verification of programs can be an incredibly powerful method to assert software correctness, it can also be incredibly challenging and complex. It demands much more work from programmers compared to automatic program analysis methods and, furthermore, requires programmers to have a deep knowledge of logics. At the same time, it is very difficult to ensure that the specification is complete and correct, a process that, along with that of developing the specification itself, escalates with the size of the programs we are building. These issues are some of the biggest obstacles in the mainstream adoption of deductive program verification, but an extremely important piece in the development of safety-critical systems.

2.2 The Why3 framework

Why3 [89] is a tool for the deductive verification of software, allowing the user to implement and specify programs, and consequently prove their correctness. Once verified, Why3 supports the extraction of *correct-by-construction* executable code of those programs [38]. For its correction proofs, Why3 has the advantage of supporting many external automatic theorem provers (such as Alt-Ergo, CVC4 and Z3) and proof assistants (such as Coq and Isabelle).

The Why3 framework provides a specification and implementation language, WhyML, which includes features popular to functional languages, such as polymorphism, algebraic data types or pattern matching, as well as imperative features like records with mutable fields and exceptions. In addition, WhyML programs can be annotated with pre and

postconditions, invariants and termination measures. These annotations are then used to generate verification conditions based on its implementation of a weakest precondition calculus. The user is also allowed to introduce assertions in the code to ease automatic proofs.

WhyML's specification component grants the possibility of writing *ghost code* [37] — code that does not interfere with program flow — with the sole purpose of aiding verification. Since it has no computational use, it can be safely removed from the code, without compromising the end result of the certified code, at the time of code extraction [71].

Once completed, the program implementation and specification follows the automated proof that the program satisfies its specification. When attempting to prove the validity conditions, generated from the annotations in the program, the user can call the automated provers. However, these provers might not be able to prove a validity condition right away. Fortunately, Why3 allows the user to apply logical transformations, such as splitting conjunctions or unfolding definitions, to simplify the formulae and help the provers. When an automated prover successfully proves a verification condition we may assume its validity, granted that we trust the soundness of the tools we are using. Otherwise, an unsuccessful attempt of proof may indicate that: even though the verification condition is valid, the provers were not able to discharge its correctness and the user might need to apply a few transformations; there is an error in the code, i.e, it does not comply with the intended specification; or the specification might be insufficient.

In our experience of using the Why3 tool, besides the characteristics we have been describing, one of the most helpful features is its ability to show which verification conditions it is trying to prove at a time, and point what are the errors in the proof or what are the verification conditions it was not able to discharge. This is a great improvement, for example, when compared with Dafny or VeriFast.

2.3 Certified software

Let us conclude this chapter by introducing a few of the most prominent and interesting projects proven correct using formal methods. These not only show that it is possible and realistic to include deductive program verification in the development of real-world software but, perhaps most importantly, they open a hopeful perspective to a future where every piece of deployed software can be mechanically verified.

CompCert is an example of certified software taking on the problem of *miscompilation*, by formally verifying a C compiler using the Coq proof assistant [23]. Since compilers are complex software and implement delicate algorithms, the fact that they might contain bugs is a potential way of introducing problems in otherwise proven-correct software, weakening the usefulness of formal verification of source code. The idea of CompCert

is that by formally verifying the compiler, if source-level verification of a program certifies that it respects a certain specification — it respects the defined set of acceptable behaviours — then we are guaranteed that the compiled code also satisfies that specification. Verification of the CompCert compiler was achieved by proving semantic preservation: if a given source program has well-defined semantics, its compiled code is observationally equivalent [23, 59]. However, abstracting from the details of proof, the most interesting takeaway from this project is how the use of an interactive proof assistant can scale to the proof of an industrial-sized system and, especially, how it opens the way for complete toolchains of verified software — from verified source programs to certified low-level representations of those programs.

DeepSpec comprises of multiple projects working towards building a network of specifications that promote the verification of full functional correctness of software and hardware [30]. This, of course, goes in line with the goals of many other efforts in the science of developing certified software. Where DeepSpec stands out, and what is intended by the so-called network of specifications, is the desire for end-to-end correction. That is, besides the more common standard of considering useful specifications those describing behaviour in detail with formal semantics, a maximally useful interface specification should also be *two-sided* — connected to both implementations and clients. This results in the development of specifications at many levels, including application-level and machine-level interfaces. Two of DeepSpec's projects are CertiKOS [17], a formally verified hypervisor kernel, and Kami [53], a Coq library for reasoning over hardware designs. CertiKOS tackles the problem of bugs and vulnerabilities at a machine-level interface, by proposing a certified kernel structured using different abstraction layers, which are formally specified and certified by verifying each kernel module at its proper abstraction level [41, 42]. This not only allows each component to be certified separately — by formally specifying each of the components — but also to minimising the number of unwanted interfaces while maximising modularity and extensibility [17, 42]. Kami, on the other hand, extends beyond software implementation and specification to hardware. It moves away from the typical verification of hardware, generally focused on a limited scope of verifiable components, proven with relatively weak specifications and invariants, to a procedure that goes in line with the methodology used for verification of code, supporting techniques for machine-checked correctness proofs, further allowing extraction to translate the program — in this case, high-level hardware designs — into low-level circuit descriptions [19, 53]. To simplify this task, the hardware design is broken down into different, simpler, components, formalised as *labelled transitions systems*, reasoning about each of them individually, specifying, implementing, and verifying each module separately, entirely within Coq [19, 53, 87].

seL4 , more specifically, the seL4 Microkernel, is a general-purpose operating system kernel with an end-to-end functional correctness proof [75, 84]. Its philosophy goes in

line with CompCert's and CertiKOS': the need for trusting the software where other (possibly verified) programs run on, and, specifically, the requirement of built-in safety and security on low-level software. Given the complexity of verification, seL4's approach was to design an architecture based on components, that simplifies this process. Part of that is done by establishing what pieces of software should be verified. Therefore, keeping in mind the necessity for preserving integrity and confidentiality, the trusted computing base (TCB) was reduced and split into what was considered critical code and non-critical code. In other words, the amount of code with privileged access to the hardware had to be minimised and verified to guarantee that no faults or failures, occurring when executing the non-critical code, affect critical execution. And, while untrusted software can be part of and contribute to the system, it must not have the ability to interfere with critical operations [55, 75, 84]. Furthermore, formal proofs of seL4 extend to prove security and safety, meaning that if its specification enforces integrity, confidentiality and availability, so does the implementation. However, and once again, given the complexity of the system, assumptions had to be made, mainly those related to the model of the system, specifically how faithful is it regarding the system and how much do guarantees over the model translate to the system itself. The latter, for example, can be tackled with the existence of projects such as CompCert, that by formally verifying the compiler and its generated code, reduce the assumptions over the correctness of the model of the hardware. Lastly, the assumption that formally stated properties are indeed good descriptions of the system's behavior [75] — an issue transversal to any system built upon deductive specification. The seL4 Microkernel has already been deployed in several real-world applications [84]. One of the most notorious being DARPA's project [39, 83] related to preventing cyberattacks on autonomous drones, such as the Boeing Unmanned Little bird, an optionally-piloted helicopter [39, 57].

VOCaL is a fairly recent effort towards developing a mechanically verified OCaml library of general-purpose data-structures and algorithms [5, 18]. It is not enough to have a program proven correct if the tools, specifically libraries, offered by the programming languages we use are not correct themselves. One of the steps towards building this verifiable library was to design a specification language for OCaml [18], whose semantics are defined by means of translation into separation logic, allowing the description of the scope and nature of the side effects of the code, and, for each function, the specification of what part of the mutable state it may access, modify, create and destroy. This further allows the specification language to be independent from the verification tool. Programs can be verified either using CFML (for targeting pointer-based data structures), Coq, or Why3. One of the interesting aspects of VOCaL is the goal of producing modular proofs. As many specifications and proofs for data-structures and algorithms can be independent of the programming language, this approach promotes the development of other formally verified libraries, and even the possibility of developing cross-language bases for formally verified code [5].

MetaCoq. Something we can easily understand from the previous projects is that, the deeper we go in program certification — as to say, the lower the level upon which verification is done — the stronger will be the correctness guarantees of (upper level) applications running on top of that system. One of the common aspects between the projects we have seen before is that, to mechanically verify programs, one has to use proof assistants and consequently trust that, in fact, they are well specified, implemented, and work as expected. MetaCoq [62] aims at solving this problem on the scope of the Coq proof assistant, verifying a subset of Coq's kernel using Coq. As has been the pattern of certifying code, one of the necessary steps for building verified programs is to formally specify those programs. MetaCoq provides a formalisation of Coq in Coq by defining the formal semantics of Coq's type theory [80]. The project builds on top of a quoting library for Coq, that is, a translation from Coq to an equivalent simpler language, responsible for the reification of Coq's internal syntax and logical environment, and type-checking algorithms [80]. It defines a correspondence from Coq kernel terms to a representation of their syntax tree as an inductive type. Based on the specification of Coq, and assuming its correction, the correctness proof can be done over this simpler language [81].

CHAPTER 3

Equivalence of Regular Expressions and Finite Automata

Included in the curriculum of any Computer Science undergraduate is the study of abstract state machines, regular expressions, grammars and languages. Some of these concepts, like finite automata and formal grammars, help students and computer scientists alike design and formalise programs, ranging from hardware to software.

In this chapter, we take on the work presented by Hopcroft et. al [46], Kozen [56], and others [60, 77, 78], and complement it by defining a mathematical correctness proof of the classic algorithm of the conversion from regular expressions to finite automata. Other contributions include easy to read mathematical definitions of inductive descriptions of the concepts leading to the definition of the algorithm and its correctness proof. In order to test these developments, we have implemented them in functional OCaml code[1].

At first, we present the reader with a few relevant notions about formal languages for the sections to come, followed by the definition of regular expression and language of a regular expression, as well as the definition and formal notation of finite automata, specifically, nondeterministic finite automata with epsilon-transitions (ε-NFAs). We then propose mathematical definitions for epsilon-closures and the extended transition function for ε-NFAs, based on their informal definitions as described by Hopcroft et al. in [46]. Finally, based on the previous definitions, we present the notion of language of an ε-NFA, leading to the definition of the conversion algorithm from regular expressions to finite automata by way of a function.

Using these concepts, we conclude this chapter with the definition of a formal correctness proof of the equivalence between finite automata and regular expressions, paving the way for the formalisation of a machine-checked proof of the conversion algorithm in the next chapter, and a discussion regarding similar proofs in classic literature and recent works.

[1] https://github.com/draexlar/Correction-of-RegEx-to-FA

3.1 Preliminaries

Let us begin this chapter by reviewing the basic mathematical definitions of formal languages and regular expressions. An *alphabet* Σ is a finite set of *symbols*, while a *word* — represented by the letters w, u and v throughout this chapter — is a sequence of symbols from Σ. A *language* over Σ is a set of words in Σ^*. Note that, according to the *Kleene closure*, for any set A, $A^* = \bigcup_{n \in \mathbb{N}_0} A^n$, therefore Σ^* is an infinite type, representing the language of all words.

3.2 Regular Expressions

Regular expressions are an algebraic description of languages, offering a declarative way to express the strings we want to accept, thus serving as the input language for many systems that process languages [46]. We consider regular expressions as defined by the following grammar:

$$R := \emptyset \mid \varepsilon \mid a \mid R \cdot R \mid R + R \mid R^*$$

where $\emptyset$ is the empty regular expression, ε represents the *empty word*, and the letter a denotes a generic *input symbol* in the alphabet, while $R \cdot R$ (or RR) denotes the concatenation of two regular expressions, $R + R$ the union of two regular expressions, and R^* the Kleene closure.

3.3 Language of a Regular Expression

The language of a regular expression R, written as $L(R)$, is defined inductively as follows:

Definition 1. Language of a regular expression R

$L \subseteq RegEx \to 2^{\Sigma^*}$

$$L(\emptyset) = \emptyset \qquad L(R \cdot E) = L(R) \cdot L(E) = \{v \cdot w \mid v \in L(R) \ \wedge \ w \in L(E)\}$$
$$L(\varepsilon) = \{\varepsilon\} \qquad L(R + E) = L(R) \cup L(E)$$
$$L(a) = \{a\} \qquad L(R^*) = L(R)^* = \{w_1 \cdot \ldots \cdot w_n \mid n \geq 0 \ \wedge \forall i, 0 < i \leq n.\ w_i \in L(R)\ \}$$

Properties of languages. Let L be a language. Since languages are sets, in the following list we omit usual properties, as the ones in regards to the union, such as commutativity and associativity, or the $\emptyset$ being the identity, among others.

- Since the empty word, ε, is an identity with respect to the concatenation of words ($\forall u, v.\ u\varepsilon v = uv$), then $\{\varepsilon\}^* = \{\varepsilon\}$.
- The empty language is the absorbing element of the concatenation: $\emptyset \cdot L = \emptyset = L \cdot \emptyset$.
- The Kleene closure of the empty set (or language) is ε: $\emptyset^* = \{\varepsilon\}$.
- The Kleene closure is idempotent: $(L^*)^* = L^*$.

3.4 Formal Notation of an ε-NFA

We now centre our attention to finite automata that, as opposed to regular expressions, are machine-like descriptions of languages. They allow one to, more easily, reason about languages as a sequence of actions between states. Specifically, we will be looking into nondeterministic finite automata with epsilon-transitions (ε-transitions), as these are the ones for which the algorithm of conversion of regular expressions is defined.

In the subject of finite automata, nondeterminism is seen has the capacity to be in multiple states at once, often perceived as the ability of "guessing" something about the input [46]. Finite automata with ε-transitions — often referred to as ε-NFA — are nothing else than nondeterministic finite automata that allow the execution of a transition through the symbol ε. This is seen as a spontaneous transition and, given that ε represents the empty word, this capability does not alter the class of languages accepted by finite automata.

Formally, we represent an ε-NFA A by a quintuple

$$A = \langle S,\ \Sigma,\ s_0,\ \delta,\ F \rangle,$$

where:

1. S is a finite set of *states* in the automaton.
2. Σ, as we have mentioned before, is the alphabet: a finite set of *input symbols*, recognised by the automaton.
3. s_0 is the *initial state* (where the computation begins), and must be one of the states in S.
4. δ ($\delta \subseteq S \times \Sigma \to S$) is the *transition function*. Particularly, in the instance of ε-NFAs, δ takes as arguments a state in S, and a member of $\Sigma \cup \{\varepsilon\}$, which can either be an *input symbol* or the symbol ε, returning a subset of S.
5. F, a subset of S, is the set of *final* or *accepting states* — the states where the computation can finish.

3.5 Epsilon-Closures

Now that we have defined ε-NFAs, we would like to define a function that allows one to reflect on the behaviour of an automaton given a sequence of symbols, otherwise known as a *word*. However, we must first present the concept of *ε-closure* of a state in the automaton. The idea is to find all states reachable from a given state s along a path with ε-transitions. Meaning, we compute all the states reachable from s by following all ε-transitions out of s, and follow the same procedure for each of the obtained states, and so on.

Below, we give an inductive definition to function `ECLOSE` for computing the *ε-closure* of a given state s, based on the textual description proposed in [46].

Definition 2. Eclose function

$\texttt{ECLOSE} \subseteq S \to 2^S$

$$\texttt{ECLOSE} \triangleq \left\{ (s, Q) \;\middle|\; Q = \{s\} \cup \{\, r \mid \forall q.\ q \in \delta(s, \varepsilon) \ \wedge\ q \neq s \ \wedge\ r \in \texttt{ECLOSE}(q) \,\} \right\}$$

Notice that, included in the ε-*closure* of a state is the own state. This is the base step of execution of ECLOSE. Other than that, if a state q is in $\texttt{ECLOSE}(s)$, and there is an ε-transition from q to a state r, then r is in $\texttt{ECLOSE}(s)$ as well. More precisely, we could say that $\texttt{ECLOSE}(s)$ contains all states in $\delta(q, \varepsilon)$, and even that $\texttt{ECLOSE}(q) \subseteq \texttt{ECLOSE}(s)$. This automatic computation of ε-transitions contributes to the perception that the ε-*closure* describes a "hidden" behavior of the automaton.

3.6 Extended Transition Function

To understand what it means for an ε-NFA to accept a certain input, we turn to the definition of extended transition function. This function, which we represent as δ^*, reflects on the behaviour of the automaton given a word which, in turn, will help us reason about the language accepted by the automaton. Informally, given a state s and a word w, the extended transition function returns the set of states reachable from the given state s, along a path whose labels (input symbols) form the given word, when concatenated. Note that the empty word, ε, does not contribute to the word, i.e., it can be implicit ($\forall u, v.\ u\varepsilon v = uv$) but will nonetheless be consumed thanks to function ECLOSE.

Below, we present the inductive definition for δ^*, which, like in the previous section, we have defined based on the textual description given by Hopcroft et al.in [46].

Definition 3. Extended transition function

$\delta^* \subseteq S \times \Sigma^* \to 2^S$

BASIS: If the label of the path is ε, then we can only follow ε-transitions from the given state. This corresponds to the above definition of ECLOSE. As such we define the base case for δ^* as:

$$\delta^*(s, \varepsilon) = \texttt{ECLOSE}(s)$$

INDUCTION: Considering a word of the form $w = ua$, where a is the last symbol of the w, we define the induction step as:

$$\begin{aligned}\delta^*(s, ua) &= \bigcup_{q \in \bigcup_{r \in \delta^*(s,u)} \delta(r,a)} \texttt{ECLOSE}(q) \\ &= \{\, p \mid \forall r, q.\ r \in \delta^*(s, u) \ \wedge\ q \in \delta(r, a) \ \wedge\ p \in \texttt{ECLOSE}(q) \,\}\end{aligned}$$

Simply put, to compute $\delta^*(s, ua)$, we begin by computing $\delta^*(s, u)$ — all states reachable from s following a path u. Then, for each of state r in the set $\delta^*(s, u)$, we have to compute $\delta(r, a)$. In other words, we have to follow all a-transitions from states

reachable from s along a path u. The resulting states from this computation are, thus, states reachable from s along a path w. Note, however, that we have to account for the possibility that this path may end with multiple ε-transitions, which in turn may have others (recall that ε can be implicit in the word). Therefore, we must perform an additional closure step, and compute the `ECLOSE` for each of those states, in order to follow any subsequent ε-transitions.

At a first glance, our definition of the extended transition function might resemble that of Kozen in [56]. However, key differences such as the use of ε-closures in the definition and the first argument being a state rather than a set of states, simplify the computation of acceptance — performing ε-transitions automatically, rather than having ε explicit in the middle of words — and helps in the formalisation of these concepts, which we will be presenting in Chapter 4.

3.7 Language of an NFA

The language of an NFA is, thus, the set of words accepted by the automaton. Concretely, we define it as the set of words in the closure of the automaton's alphabet — every possible word constructed by the concatenation of symbols accepted by the alphabet or ε — that lead the automaton's initial state to any of its final states.

Formally, the language L of an NFA, as defined in [46], is as follows. Note that the properties previously defined for the language in Sec. 3.3 are also true here.

Definition 4. Language of an NFA A

$L \subseteq NFA_\Sigma \to 2^{\Sigma^*}$

$$L(A) = \{w \in \Sigma^* \mid (\delta^*(s_0, w) \cap F) \neq \emptyset\}$$

3.8 Converting Regular Expressions into Finite Automata

For the translation of regular expressions into finite automata, we will be using an auxiliary function, `Compile`. This function takes a regular expression R, over a certain alphabet Σ, as an argument and returns an ε-NFA, over that same alphabet.

The conversion methodology we follow is actually inspired by the algorithms presented by Hopcroft et al. [46] and Sipser [78], in order to build more compact and easy to understand ε-NFAs. For the definition of the conversion algorithm, we make use of a `Compile` function, which we have adapted from the lecture notes of Theory of Computation by A. Ravara.

The definition of the `Compile` function can be done inductively on R, with the basis being the construction of automata from $R = \emptyset$ (representing the empty regular expression), $R = \varepsilon$, and a single input symbol $R = a$ ($a \in \Sigma$). Afterwards, it is possible to construct automata from the combination of these simpler ones, through concatenation, union and

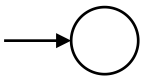
closure of regular expressions. For each of these cases, we present the corresponding computation, as well as a graphic representation of the resulting automaton.

Definition 5. Compile function

$\texttt{Compile} \subseteq \text{RegExp} \to \varepsilon\text{-NFA}_{\Sigma}$

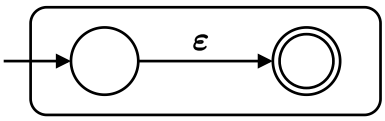
BASIS:

Case $R = \emptyset$. To accept the empty language basically means the automaton stays idle, so it only needs an initial state, no final states and no transitions.



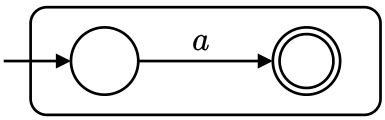
$$\texttt{Compile}(\emptyset) = \langle\ \{s_0\},\ \emptyset,\ s_0,\ \emptyset,\ \emptyset\ \rangle$$

Case $R = \varepsilon$. To accept the empty word we only need one initial state and one final state, with an ε-transition from one to the other.



$$\texttt{Compile}(\varepsilon) = \langle\ \{s_0, s_1\},\ \emptyset,\ s_0,\ \{(s_0, \varepsilon, s_1)\},\ \{s_1\}\ \rangle$$

Case $R = a$. Similarly, to accept the word with length 1 (a symbol a, such that $a \in \Sigma$) we only need one initial state and one final state, and a transition from the initial state to the final state through symbol a.



$$\texttt{Compile}(a) = \langle\ \{s_0, s_1\},\ \{a\},\ s_0,\ \{(s_0, a, s_1)\},\ \{s_1\}\ \rangle$$

INDUCTION:

Let us consider two regular expressions, E and G, each with its own alphabet:

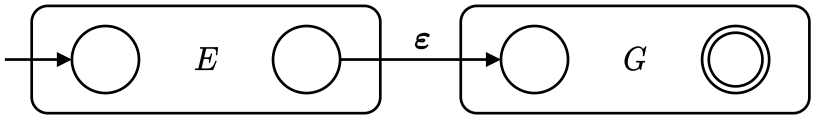
$$E \in \text{RegExp}(\Sigma_E) \ \text{ and } \ G \in \text{RegExp}(\Sigma_G),$$

and let us define their corresponding automata as:

$$\texttt{Compile}(E) = \langle\ S_E,\ \Sigma_E,\ s_{0_E},\ \delta_E,\ F_E\ \rangle \ \text{ and } \ \texttt{Compile}(G) = \langle\ S_G,\ \Sigma_G,\ s_{0_G},\ \delta_G,\ F_G\ \rangle,$$

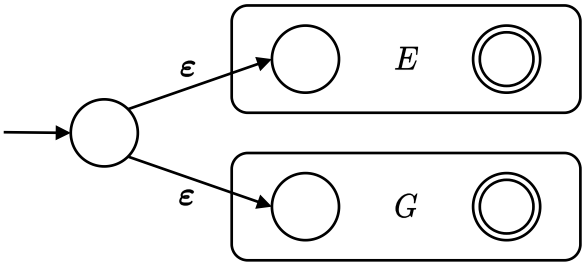
we define the automata resulting from the concatenation, union, and closure, respectively, as follows.

Case $R = EG$. To accept the words resulting from the concatenation of a word from the language of E and a word from the language of G, we need to build an automaton that, similarly, connects the automaton resulting from E and the one resulting from G. We do this by adding ε-transitions that connect the final states of the automaton described by E ($\texttt{Compile}(E)$) to the initial states of the one described by G ($\texttt{Compile}(G)$).



$$\texttt{Compile}(EG) = \langle\ S_E \cup S_G,\ \Sigma_E \cup \Sigma_G,\ s_{0_E},\ \delta_{EG},\ F_G\ \rangle\ \text{, with}$$
$$\delta_{EG} = \delta_E \cup \{(f, \varepsilon, s_{0_G}) \mid f \in F_E\} \cup \delta_G\ \text{, and}$$
$$S_E \cap S_G = \emptyset$$

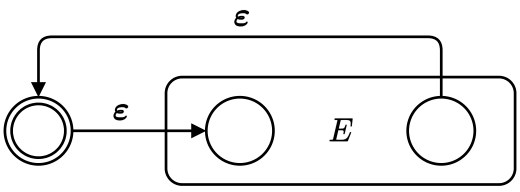
Case $R = E + G$. To accept the word belonging to $L(E + G)$, that is, a word either belonging to $L(E)$ or $L(G)$, we need to decide whether the word should be computed by $\texttt{Compile}(E)$ or $\texttt{Compile}(G)$. We need to start the computation of the word in a new state that *nondeterministically* decides if to go to the initial state of $\texttt{Compile}(E)$ or $\texttt{Compile}(G)$, and consequently reach the accepting state of either one of these automata. Thus, the resulting automaton from $\texttt{Compile}(E+G)$ has a new initial state i (the only one), that transitions to the previous initial states of $\texttt{Compile}(E)$ and $\texttt{Compile}(G)$ through a ε-transition.



$$\texttt{Compile}(E + G) = \langle\ \{i\} \cup S_E \cup S_G,\ \Sigma_E \cup \Sigma_G,\ i,\ \delta_{E+G},\ F_E \cup F_G\ \rangle\text{, with}$$
$$\delta_{E+G} = \{(i, \varepsilon, s_{0_E}), (i, \varepsilon, s_{0_G})\} \cup \delta_E \cup \delta_G\ \text{, and}$$
$$i \notin (S_E \cup S_G)\ \wedge\ S_E \cap S_G = \emptyset$$

Case $R = E^*$. Recall that, by definition, $L(E^*) = L(E)^*$. This means that $\varepsilon \in L(E^*)$ (by the Kleene closure). Therefore, the automaton we will build *must* accept the empty word ε. On the other hand, we can think of E^* as a successive concatenation of expressions E ($\varepsilon + E + EE + EEE + ...$). Intuitively, it seems that to construct the new automaton we have to use ideas both from the concatenation and union of regular expressions. Hence, the resulting automaton from $\texttt{Compile}(E^*)$ has a new start state i, that will also be the only final state of this NFA (thus accepting ε), and has two new $\varepsilon - transitions$: one exiting i to the

previous initial state of the automaton $\texttt{Compile}(E)$, and another one from the previous final state of that automaton to state i.



$$\texttt{Compile}(E^*) = \langle\ \{i\} \cup S_E,\ \Sigma_E,\ i,\ \delta_{E^*},\ \{i\}\ \rangle, \text{ with}$$
$$\delta_{E^*} = \{(i, \varepsilon, s_{0_E})\} \cup \delta_E \cup \{(f, \varepsilon, i)\ |f \in F_E\}\ , \text{and}$$
$$i \notin S_E.$$

3.9 Proof of Correctness

The algorithm presented in the previous section showed us that regular expressions can be converted in finite automata. Now, we would like to confirm that, in fact, these two representations have equivalent descriptive power. That is, to show that the generated NFA recognises the language described by the original regular expression, regardless of their apparent representational difference.

Concretely, to prove the correctness of the conversion algorithm (or function) is to prove that the given regular expression and the automaton resulting from the conversion of that same regular expression convey the same meaning. Formally, we want to show that, for any regular expression R, the finite automaton obtained from applying the function $\texttt{Compile}$ to R describes a language equal to that of the regular expression.

Theorem 1. $\forall R \in RegExp(\Sigma).\ L(R) = L(\texttt{Compile}(R))$.

The proof herein presented is done by structural induction on a given regular expression. Therefore, similarly to the reasoning followed when defining the $\texttt{Compile}$ function, we consider the possibilities of an empty regular expression, $\emptyset$, the empty word, ε, or a single input symbol a, as well as the concatenation and union.

For the following proof recall the definitions of ε-closure, extended transition function, and language of an ε-NFA A:

$$\texttt{ECLOSE} \triangleq \left\{ (s, Q) \;\middle|\; Q = \{s\} \cup \{\ r \mid \forall q.\ q \in \delta(s, \varepsilon)\ \wedge\ q \neq s\ \wedge\ r \in \texttt{ECLOSE}(q)\ \}\ \right\}$$

$$\delta^*(s, \varepsilon) = \texttt{ECLOSE}(s)$$

$$\delta^*(s, ua) = \{\ p \mid \forall r, q.\ r \in \delta^*(s, u)\ \wedge\ q \in \delta(r, a)\ \wedge\ p \in \texttt{ECLOSE}(q)\ \}$$

$$L(A) = \{\ w \in \Sigma^* \mid (\delta^*(s_0, w)\ \cap\ F) \neq \emptyset\ \}.$$

Proof. The proof is done by structural induction on the regular expression R.

Case $R = \emptyset$.

By definition, $L(\emptyset) = \emptyset$.
Let $A = \texttt{Compile}(\emptyset) = \langle\ \{s_0\},\ \emptyset,\ s_0,\ \emptyset,\ \emptyset\ \rangle$.

We want to show that $L(A) = \emptyset$.

Let $F = \texttt{finalStates}(A) = \emptyset$.
Hence, $L(A) = \{\ w \in \emptyset^* \mid (\delta^*(s_0, w)\ \cap\ \emptyset) \neq \emptyset\ \} = \emptyset$, since the defining property is false.

Case $R = \varepsilon$.

By definition, $L(\varepsilon) = \{\varepsilon\}$.
Let $A = \texttt{Compile}(\varepsilon) = \langle\ \{s_0, s_1\},\ \emptyset,\ s_0,\ \{(s_0, \varepsilon, s_1\},\ \{s_1\}\ \rangle$.

We want to show that $L(A) = \{\varepsilon\}$.

Note that, by definition, $\emptyset^* = \{\varepsilon\}$ and $\{\varepsilon\}^* = \{\varepsilon\}$.
Let $F = \texttt{finalStates}(A) = \{s_1\}$.
We have,

$$
\begin{aligned}
\delta^*(s_0, \varepsilon) &= \textsf{ECLOSE}(s_0) \\
&= \{s_0\} \cup \{\ r \mid \forall q.\ q \in \delta(s_0, \varepsilon)\ \wedge\ q \neq s_0\ \wedge\ r \in \textsf{ECLOSE}(q)\ \} \\
&= \{s_0\} \cup \{\ r \mid \forall q.\ q \in \{s_1\}\ \wedge\ q \neq s_0\ \wedge\ r \in \textsf{ECLOSE}(q)\ \} \\
&= \{s_0\} \cup \textsf{ECLOSE}(s_1) \\
&= \{s_0\} \cup \{s_1\} \cup \{\ r \mid \forall q.\ q \in \delta(s_1, \varepsilon)\ \wedge\ q \neq s_1\ \wedge\ r \in \textsf{ECLOSE}(q)\ \} \\
&= \{s_0\} \cup \{s_1\} \cup \emptyset \\
&= \{s_0, s_1\}
\end{aligned}
$$

Hence,

$$
\begin{aligned}
L(A) &= \{\ w \in \Sigma^* \mid (\delta^*(s_0, w)\ \cap\ F) \neq \emptyset\ \} \\
&= \{\ w \in \{\varepsilon\}^* \mid (\delta^*(s_0, w)\ \cap\ \{s_1\}) \neq \emptyset\ \} \\
&= \{\ w \in \{\varepsilon\} \mid (\delta^*(s_0, \varepsilon)\ \cap\ \{s_1\}) \neq \emptyset\ \} \\
&= \{\ w \in \{\varepsilon\} \mid (\{s_0, s_1\}\ \cap\ \{s_1\}) \neq \emptyset\ \} \\
&= \{\varepsilon\}
\end{aligned}
$$

Case $R = a$.

By definition, $L(a) = \{a\}$.
Let $A = \texttt{Compile}(a) = \langle\ \{s_0, s_1\},\ \{a\},\ s_0,\ \{(s_0, a, s_1\},\ \{s_1\}\ \rangle$.

We want to show that $L(A) = \{a\}$.

Note that, by definition, $\{a\}^* = \{a^n \mid n \in \mathbb{N}_0\}$ and $a^0 = \varepsilon$.
Furthermore, $\forall w \in \Sigma^*.\ w = \varepsilon w = w\varepsilon$.
Let $F = \texttt{finalStates}(A) = \{s_1\}$.

Notice that, by definition of the automaton A, no word of length greater than 1 will lead the initial state s_0 to the only final state s_1. Moreover, given the alphabet of A, the only words possible to form are sequences of a.

Lemma 2. $\forall n \in \mathbb{N}_0,\ n > 1,\ \delta^*(s_0, a^n) \notin F$

We have,

$$\begin{aligned}
\delta^*(s_0, \varepsilon) &= \textsc{eclose}(s_0) \\
&= \{s_0\} \cup \{\ r \mid \forall q.\ q \in \delta(s_0, \varepsilon)\ \wedge\ q \neq s_0\ \wedge\ r \in \textsc{eclose}(q)\ \} \\
&= \{s_0\}
\end{aligned}$$

$$\begin{aligned}
\delta^*(s_0, a) &= \{\ p \mid \forall r, q.\ r \in \delta^*(s_0, \varepsilon)\ \wedge\ q \in \delta(r, a)\ \wedge\ p \in \textsc{eclose}(q)\ \} \\
&= \{\ p \mid \forall r, q.\ r \in \{s_0\}\ \wedge\ q \in \delta(r, a)\ \wedge\ p \in \textsc{eclose}(q)\ \} \\
&= \{\ p \mid \forall r, q.\ r = s_0\ \wedge\ q \in \delta(s_0, a)\ \wedge\ p \in \textsc{eclose}(q)\ \} \\
&= \{\ p \mid \forall r, q.\ r = s_0\ \wedge\ q \in \{s_1\}\ \wedge\ p \in \textsc{eclose}(q)\ \} \\
&= \{\ p \mid \forall r, q.\ r = s_0\ \wedge\ q = s_1\ \wedge\ p \in \textsc{eclose}(s_1)\ \} \\
&= \textsc{eclose}(s_1) \\
&= \{s_1\}
\end{aligned}$$

Hence,

$$\begin{aligned}
L(A) &= \{\ w \in \Sigma^* \mid (\delta^*(s_0, w)\ \cap\ F) \neq \emptyset\ \} \\
&= \{\ w \in \{a\}^* \mid (\delta^*(s_0, w)\ \cap\ \{s_1\}) \neq \emptyset\ \} \\
&\quad \text{by Lemma 2} \\
&= \{\ w \in \{\varepsilon, a\} \mid (\delta^*(s_0, w)\ \cap\ \{s_1\}) \neq \emptyset\ \} \\
&= \{\ w \in \{\varepsilon\} \mid (\delta^*(s_0, \varepsilon)\ \cap\ \{s_1\}) \neq \emptyset\ \} \cup \{\ w \in \{a\} \mid (\delta^*(s_0, a)\ \cap\ \{s_1\}) \neq \emptyset\ \} \\
&= \{a\}
\end{aligned}$$

Case $R = EG$.

Let $E \in \text{RegExp}(\Sigma_E)$ and $G \in \text{RegExp}(\Sigma_G)$
By definition, $L(EG) = L(E) \cdot L(G)$

Let

$$\begin{aligned}
&\texttt{Compile}(E) = \langle\ S_E,\ \Sigma_E,\ s_{0_E},\ \delta_E,\ F_E\ \rangle, \text{ and} \\
&\texttt{Compile}(G) = \langle\ S_G,\ \Sigma_G,\ s_{0_G},\ \delta_G,\ F_G\ \rangle.
\end{aligned}$$

By induction hypothesis,

$$L(\texttt{Compile}(E)) = L(E) \text{ and } L(\texttt{Compile}(G)) = L(G).$$

Finally, let

$$\begin{aligned}
&A = \texttt{Compile}(EG) = \langle\, S_E \cup S_G,\ \Sigma_E \cup \Sigma_G,\ s_{0_E},\ \delta_{EG},\ F_G \,\rangle \text{, with} \\
&\delta_{EG} = \delta_E \cup \{(f, \varepsilon, s_{0_G}) \mid f \in F_E\} \cup \delta_G \text{, and} \\
&S_E \cap S_G = \emptyset
\end{aligned}$$

Remark. $\delta_E \cup \{(f, \varepsilon, s_{0_G}) \mid f \in F_E\} \cup \delta_G = \delta_E \uplus \{(f, \varepsilon, s_{0_G}) \mid f \in F_E\} \uplus \delta_G$.

We want to show that $L(A) = L(EG)$.

Note that, by definition of the automaton A, the words leading from the initial state, s_{0_E}, to the accepting state(s) in G, are those who first go through the automaton for E, and then go through the automaton for G. This is given by the above definition of δ_{EG}, where words starting in one of the final states for the automaton for E lead to the start state of the automaton for G, s_{0_G}, by ε. Therefore, words accepted by A can be split into two words: one accepted by $\texttt{Compile}(E)$ and one accepted by $\texttt{Compile}(G)$.

Lemma 3.

$$\begin{aligned}
&\forall w \in \Sigma^*.\ (\delta^*_{EG}(s_{0_E}, w) \cap F_G) \neq \emptyset \\
&\quad \implies \exists u \in \Sigma^*_E,\ \exists v \in \Sigma^*_G.\ w = uv \ \wedge\ (\delta^*_E(s_{0_E}, u) \cap F_E) \neq \emptyset \ \wedge\ (\delta^*_G(s_{0_G}, v) \cap F_G) \neq \emptyset
\end{aligned}$$

Once again, we point to the fact that the ε symbol is the *empty word*, thus not contributing to the overall structure of the word: $\forall u, v \in \Sigma^*.\ uv = u\varepsilon v$. Therefore, we know that, when we compute the acceptance of a word, by automaton A ($\delta^*_E G$), whenever we reach the (what used to be) the final state(s) of $\texttt{Compile}(E)$, the transition to (what used to be) the initial state of $\texttt{Compile}(G)$ is performed automatically, due to ECLOSE. Specifically, we have,

$$\begin{aligned}
\forall f \in F_E,\ \texttt{ECLOSE}(f) &= \{f\} \cup \{r \mid \forall q.\ q \in \delta_{EG}(f, \varepsilon) \ \wedge\ q \neq f \ \wedge\ r \in \texttt{ECLOSE}(q)\} \\
&= \{f\} \cup \texttt{ECLOSE}(s_{0_G}) \\
&= \{f, s_{0_G}\} \cup \{r \mid \forall q.\ q \in \delta_{EG}(s_{0_G}, \varepsilon) \ \wedge\ q \neq s_{0_G} \ \wedge\ r \in \texttt{ECLOSE}(q)\}
\end{aligned}$$

Hence,

$$\begin{aligned}
L(A) &= \{\, w \in \Sigma^* \mid (\delta^*_{EG}(s_{0_E}, w) \cap F_G) \neq \emptyset \,\} \\
&= \{\, w \in (\Sigma_E \cup \Sigma_G)^* \mid (\delta^*_{EG}(s_{0_E}, w) \cap F_G) \neq \emptyset \,\} \\
&\quad \text{by Lemma 3} \\
&= \{\, uv \in (\Sigma_E \cup \Sigma_G)^* \mid (\delta^*_E(s_{0_E}, u) \cap F_E) \neq \emptyset \wedge (\delta^*_G(s_{0_G}, v) \cap F_G) \neq \emptyset \,\} \\
&= \{\, u \in \Sigma^*_E \mid (\delta^*_E(s_{0_E}, u) \cap F_E) \neq \emptyset \,\} \cdot \{\, v \in \Sigma^*_G \mid (\delta^*_G(s_{0_G}, v) \cap F_G) \neq \emptyset \,\} \\
&= L(E) \cdot L(G) \\
&\quad \text{by induction hypothesis} \\
&= L(EG)
\end{aligned}$$

Case $R = E + G$.

Let $E \in \mathrm{RegExp}(\Sigma_E)$ and $G \in \mathrm{RegExp}(\Sigma_G)$
By definition, $L(E + G) = L(E) \cup L(G)$

Let

$$\begin{aligned}
\texttt{Compile}(E) &= \langle\, S_E,\ \Sigma_E,\ s_{0_E},\ \delta_E,\ F_E \,\rangle, \text{ and} \\
\texttt{Compile}(G) &= \langle\, S_G,\ \Sigma_G,\ s_{0_G},\ \delta_G,\ F_G \,\rangle.
\end{aligned}$$

By induction hypothesis,

$$L(\texttt{Compile}(E)) = L(E) \ \text{ and } \ L(\texttt{Compile}(G)) = L(G).$$

Finally, let

$$\begin{aligned}
&A = \texttt{Compile}(E + G) = \langle\, \{i\} \cup S_E \cup S_G,\ \Sigma_E \cup \Sigma_G,\ i,\ \delta_{E+G},\ F_{E+G} \,\rangle, \text{ with} \\
&i \notin (S_E \cup S_G) \ \wedge \ S_E \cap S_G = \emptyset, \text{ and} \\
&\delta_{E+G} = \{(i, \varepsilon, s_{0_E}), (i, \varepsilon, s_{0_G})\} \cup \delta_E \cup \delta_G \\
&F_{E+G} = F_E \cup F_G
\end{aligned}$$

We want to show that $L(A) = L(E + G)$.

Remark. $\forall w \in \Sigma^*.\ w = \varepsilon w$.

We have,

$$\begin{aligned}
\texttt{ECLOSE}(i) &= \{i\} \cup \{\, r \mid \forall q.\ q \in \delta_{E+G}(i, \varepsilon) \ \wedge \ q \neq i \ \wedge \ r \in \texttt{ECLOSE}(q) \,\} \\
&= \{i\} \cup \{\, r \mid \forall q.\ q \in \{s_{0_E}, s_{0_G}\} \ \wedge \ q \neq i \ \wedge \ r \in \texttt{ECLOSE}(q) \,\} \\
&= \{i\} \cup \texttt{ECLOSE}(s_{0_E}) \cup \texttt{ECLOSE}(s_{0_G}) \\
&= \{i, s_{0_E}, s_{0_G}\} \\
&\qquad \cup \{\, r \mid \forall q.\ q \in \delta_{E+G}(s_{0_E}, \varepsilon) \ \wedge \ q \neq s_{0_E} \ \wedge \ r \in \texttt{ECLOSE}(q) \,\} \\
&\qquad\quad \cup \{\, r \mid \forall q.\ q \in \delta_{E+G}(s_{0_G}, \varepsilon) \ \wedge \ q \neq s_{0_G} \ \wedge \ r \in \texttt{ECLOSE}(q) \,\}
\end{aligned}$$

Note that, by definition of automaton A, starting at the new initial state, i, we can go to the start state of either the automaton for E or the automaton for G. This is given by the above definition of the transition function δ_{E+G}, where the start state, i, leads to either s_{0_E} or s_{0_G}, the initial states for the automata for E and G, respectively, by ε. This behaviour is, furthermore, shown in the above computation of $\texttt{ECLOSE}(f)$. We can, thus, reach one of the accepting states of automaton A — which, by definition, is in $F_E \cup F_G$ — through a word w that, if part of Σ_E^* leads to an acceptance state of the automaton for E, otherwise ($w \in \Sigma_G^*$), leads to an acceptance state of the automaton for G. This is also given by the definition of δ_{E+G} since, besides the transitions we mentioned before, it comprises of the union of δ_E and δ_G.

Lemma 4.

$$\begin{aligned}&\forall w \in \Sigma^*.\ (\delta^*_{E+G}(i,w) \cap F_{E+G}) \neq \emptyset \\ &\quad \implies \left(w \in \Sigma_E^* \wedge (\delta^*_E(s_{0_E},w) \cap F_E) \neq \emptyset\right) \vee \left(w \in \Sigma_G^* \wedge (\delta^*_G(s_{0_G},w) \cap F_G) \neq \emptyset\right)\end{aligned}$$

Hence,

$$\begin{aligned}L(A) &= \{\ w \in \Sigma^* \mid (\delta^*_{E+G}(i,w)\ \cap\ F_{E+G}) \neq \emptyset\ \} \\ &= \{\ w \in (\Sigma_E \cup \Sigma_G)^* \mid (\delta^*_{E+G}(s_{0_E},w)\ \cap\ (F_E \cup F_G)) \neq \emptyset\ \} \\ &\quad \text{by Lemma 4} \\ &= \{\ w \in (\Sigma_E \cup \Sigma_G)^* \mid (\delta^*_E(s_{0_E},w)\ \cap\ F_E) \neq \emptyset \vee (\delta^*_G(s_{0_G},w)\ \cap\ F_G) \neq \emptyset\ \} \\ &= \{\ w \in \Sigma_E^* \mid (\delta^*_E(s_{0_E},w)\ \cap\ F_E) \neq \emptyset\ \} \cup \{\ w \in \Sigma_G^* \mid (\delta^*_G(s_{0_G},w)\ \cap\ F_G) \neq \emptyset\ \} \\ &= L(E) \cup L(G) \\ &\quad \text{by induction hypothesis} \\ &= L(E+G)\end{aligned}$$

■

With the successful conclusion of the correction proof of the `Compile` function, we can now be confident that the conversion from regular expressions to finite automata is language-preserving, i.e., both representations describe the same language.

3.10 Discussion and Related Work

In this chapter, we have contributed to the classic study of automata and regular expressions with formal mathematical definitions of classical concepts such as ε-closure (Sec. 3.5) and the extended transition function (Sec. 3.6), as well as a proof of correction for the conversion of regular expressions into finite automata, by proving that the language described by the given regular expression is the same as the language of the resulting automaton. While we have not yet defined the correctness proof for the third

inductive case of regular expressions, the closure ($R = E^*$) — given the increased complexity compared to the other inductive cases, due to the infinite nature of the definition of closure — and left some lemmas to our basis of trust, the work herein presented is already of great value.

Existing proofs in classic literature tend to be more focused on proving that it is possible to construct an automaton from a regular expression [46, 56, 78] rather than actually proving the equality between their languages, even if that is what they propose to do, ending up defining the conversion algorithm as proof. And while there is work that, just likes us, successfully proves this equality — the most interesting being given by Michael Sipser in a lecture for his course on Theory of Computation [79] — even though quite intuitive, it is nonetheless informal, proving the equivalence textually rather than through mathematical statements. Kaiser [52], on the other hand, besides being more focused on mechanisation, presents a variation on the more common translation between regular expressions and finite automata that uses deterministic finite automata (DFAs) instead of NFAs. Subsequent more formal works [33, 34], with also mechanically-checked proofs (ours will be discussed in the next chapter), can be found as well. However, they lack a direct explanation of the proof of equivalence between regular expressions and NFAs, for example by only proving that a language is regular (a language is accepted by an NFA) leaving the proof of language equality implicit. Furthermore, missing a clear and easy to understand exposition of the proof, pointing in the direction of the proof but, for instance, only explaining what the result should be (prove the equality of the languages), with far less specificity and details compared to the one we present.

Our main goal is an elegant presentation of the results that are well known in the literature for automata theory. By giving formal definitions to classical concepts, like the ones mentioned at the beginning of this section, and designing a formal proof for the conversion from regular expressions to finite automata, through clear mathematical statements, detailing every step of the proof, we not only strengthen the credibility of the algorithm and help reason about the relation between regular expressions, automata and languages, but also help guide towards the formalisation of a mechanically-checked proof of the algorithm, which we will be discussing in the next chapter.

CHAPTER 4

Mechanical Verification

The Why3 framework allows users to implement and specify programs, and prove that the implemented code is compliant with the specification. This requires the formalization of the underlying algebraic theory and proof that, indeed, the behaviour of the implemented program is correct and respects the contracts of the specification.

In this chapter, we present a mechanical proof of the conversion from regular expressions to finite automata. We do so by implementing a functional WhyML program, following the concepts we have introduced in the previous chapter and, for each of them, formalise logical assertions based on their mathematical definitions and the correctness proof proposed in the previous chapter.

The work we have presented in Chapter 3 has, thus, proved to be very helpful in structuring the mechanical proof and guiding us towards specifying the right pre and postconditions, and defining base properties and lemmas. However, we have come to understand that many concepts that may be perceived as obvious (to us humans, at least) and, therefore, implicit in the constructive mathematical proof, may raise several issues and must be further clarified and specified in order to help the machine verify certain statements.

Throughout the next sections, we will guide the reader on understanding our thought process for implementing and formalizing a specification for the concepts we defined in the previous chapter, detailing each assertion that went into their specification, as well as implementation and specification problems. We will also discuss the issues we faced while verifying the specification in regards to the proposed implementation, and possible alternatives and solutions to specific problems and limitations. We conclude the chapter with a discussion about related work concerning the formalization of formal languages and Kleene algebra.

4.1 Implementation and Specification

For every proof, especially when talking about deductive program verification, there is a certain degree of assumptions that can be and are made. For the present issue, the execution of the algorithm only depends on function `Compile`, therefore, only that and other auxiliary functions it might depend upon have to be implemented. Simply put, what this means is that all other functions are only necessary for the specification of the program. As to say, functions only used in pre and postconditions or lemmas, do not require implementation, and thus need not be proven, as the only requirement in the context of proof is their specification.

Notwithstanding, having an implementation for even those functions that are only required in the logical context not only can be helpful when specifying contracts for those functions, avoiding under-specifying, but can also help in verifying other functions that might depend on them. This is why we have decided to implement every function. In the end result — the possible extraction of the code — this will not have an *apparent* implication. However, it will lighten our basis of trust because, on one hand, and as we have mentioned before, this may contribute to a stronger specification, and on the other, if proven correct it means that we do not have to trust it to be true: we know for a fact. Consequently, this might influence the proof of the core code that is extractable and meant to be run.

We begin this section by presenting the reader with a few basic type definitions, as well as the basic concepts we have defined in Chapter 3, such as regular expression and nondeterministic finite automaton. From Sec. 4.1.2 forward, we take a top-down approach and, opposed to the previous chapter, start with the implementation of the conversion algorithm, presenting other important concepts and functions as they become relevant.

In order to keep this section as brief as possible, we have omitted the code for some auxiliary functions, lemmas, and axioms, for which we will give simple and concise explanations of their purpose, whenever necessary, or present them in Sec. 4.2 if relevant. For implementation details regarding this procedure, we recommend the reader to refer to our online repository[1], where we make the code available.

4.1.1 Basic Types

The first step to our implementation is defining the types for regular expression and ε-NFA. Note that, for this implementation, we are not concerned with the parsing of strings, as this is not relevant for the definition of the conversion algorithm.

We start by presenting the basic types for our program in Listing 4.1. These include: a constant representing the empty word ε, denoted `eps`; the notion of symbol and alphabet, which is a set of symbols; words, strings obtained through the concatenation of symbols,

[1] https://github.com/draexlar/Correction-of-RegEx-to-FA

which we represent by a list of symbols; the notion of language, defined as a set of words; the idea of state, which can be defined by an integer, and set of states; the notion of transition, represented as a tuple with an initial state, a symbol and a final state, as well as transition set. Notice that we omit the actual value of eps since, for the correctness proof, this representation is sufficient at a logical level and, at the same time, makes the proof more modular.

```
val constant eps : char (* used for representing the empty transitions *)

(* defining input symbols and alphabet *)
type symbol = char
clone import set.SetApp as SA with type elt = symbol

(* defining word and language as a set *)
type word = list symbol
clone import set.SetApp as SL with type elt = word

(* defining a state and a set of states *)
type state = int
clone import set.SetApp as SS with type elt = state

(* defining a transition and a set of transitions *)
type transition = (state , symbol , state)
clone import set.SetApp as ST with type elt = transition
```

Listing 4.1: Basic types necessary for definition of symbols, words, and sets.

The definition of a type for regular expressions is quite simple. Recall that, alike the mathematical proof defined in the previous chapter, we are not considering the Kleene closure of a regular expression (R^*). As such, we consider: the case of it being empty (represented by Empty); the empty word ε (represented by Eps); a single input symbol a (represented by Symb); the concatenation of two regular expressions (represented by Seq); and the union of two regular expressions (represented by Plus). Additionally, in listing 4.2, we include the definition of a function that, given a regular expression, returns the language of the regular expression, according to the definition presented in Sec. 3.3.

```
(* Type representing a Regular Expression *)
type regex =
  Empty
  | Eps
  | Symb symbol
  | Seq regex regex
  | Plus regex regex

(* Given a Regular Expression, returns its language *)
function regex_lang (r: regex) : fset word =
  match r with
```

```
12    | Empty → empty
13    | Eps → add (Cons eps Nil) empty
14    | Symb a → add (Cons a Nil) empty
15    | Seq e f → let l = regex_lang e in
16                let m = regex_lang f in
17                    concat l m
18    | Plus e f → union (regex_lang f) (regex_lang e)
19    end
```

Listing 4.2: Type for regular expression and language of a regular expression.

Function concat — used in the definition of language for regular expressions and implemented in the following listing — is an auxiliary function that, as the name implies, performs the concatenation of two given languages. Due to the complexity of implementing such function and the fact that it is only needed in the logic context of the proof, we have decided to axiomatise it, rather than implement it, based on the axiomatization of other functions in the Set library of Why3[2], such as `filter` or map. In Listing 4.3, we also define a few important properties about the concatenation of languages that will be relevant for the proof of the conversion algorithm. Namely: $\emptyset$ is the absorbing element of the concatenation; if we concatenate two *non-empty* languages, then the resulting language will also be *non-empty*; if the concatenation of two languages is empty, at least one of those languages is also empty; for any word u belonging to a language L_1 and any word v belonging to a language L_2, the word resulting from their concatenation $(u \cdot v)$ is accepted by a language $L = L_1 \cdot L_2$. These lemmas are easily discharged.

```
1  (* Concatenation of languages *)
2  function concat (a: fset word) (b: fset word) : fset word
3  axiom concat_def:
4    forall l1 l2: fset word, w: word.
5      mem w (concat a b) ↔ exists u v. mem u l1 ∧ mem v l2 ∧ w = (u ++ v)
6
7  (* The empty language is absorbing element of the concatenation. *)
8  lemma neutral_left_concat:
9    forall l1 l2: fset word. is_empty l1 → is_empty (concat l1 l2)
10
11 lemma neutral_rigth_concat:
12   forall l1 l2: fset word. is_empty l2 → is_empty (concat l1 l2)
13
14 (* If none of the languages to be concatenated is empty, then the resulting
       language is not empty. *)
15 lemma not_empty_concat:
16   forall l1 l2: fset word.
17     not is_empty l1 ∧ not is_empty l2 → not is_empty (concat l1 l2)
18
```

[2] http://why3.lri.fr/stdlib/set.html

```
(* If the language resulting from the concatenation of two other languages is
    empty, then, at least, one of those languages is empty. *)
lemma empty_concat:
  forall l1 l2: fset word.
    is_empty (concat l1 l2) → ( is_empty l1  ∨  is_empty l2 )

(* For any two words belonging to two languages, respectively, the concatenation
     of those two words is accepted by the language resulting from the
    concatenation of two previous languages. *)
lemma composition_concat:
  forall u v: word, l1 l2: fset word.
      mem u l1 ∧ mem v l2 → mem (u ++ v) (concat l1 l2)
```

Listing 4.3: Concatenation of two languages.

Notice, in Listing 4.1, that constant `eps` and type `symbol` are of type `char`. This can bring ambiguity, especially when we enter the case where the regular expression is `Symb`. We do not want to have the case where the symbol in `Symb` would be `eps`. It may not seem to be a big problem, but the reason will become clear further down this section. For now, suffices to say that we do this for the sake of coherence. So, to solve this problem, we have defined a predicate `regex_wf` (in Listing 4.4) that checks if a given regular expression is well-formed, specifically, an atómic regular expression cannot be ε.

```
(* RegEx is well formed *)
predicate regex_wf (r: regex)
=
  match r with
  | Symb a → a ≠ eps
  | Seq e f → regex_wf e ∧ regex_wf f
  | Plus e f → regex_wf e ∧ regex_wf f
  | _ → true
  end
```

Listing 4.4: Well-formed regular expression.

The type of automaton, presented in the listing below, is straightforward and is taken directly from the definition of ε-NFA in Sec. 3.4. The only observable difference is the use of a set for transitions, instead of a transition function δ, which does not translate in a big difference in the structure of the automaton, but will simplify the formalization for the rest of the program. Nonetheless, we have implemented a function `delta` (in Listing 4.6) which, just like the definition of Sec. 3.4, returns the set of states reachable from a given state through a transition labelled by a given symbol.

```
type automaton = {
  states: SS.set;
  alphabet: SA.set;
  start: state;
  transitions: ST.set;
```

```
  final_states: SS.set;
} invariant { mem start states }
  invariant { not (is_empty states) }
  invariant { not mem eps alphabet }
  invariant { forall x. mem x alphabet → exists a, c. mem (a,x,c) transitions }
  invariant { subset final_states states }
  invariant { forall t: transition. mem t transitions →
                let (a,b,c) = t in
                mem a states ∧ mem b (add eps alphabet) ∧ mem c states }
```

Listing 4.5: Type for a finite automaton.

Notice we have included invariants in the definition of `automaton`. These respectively guarantee that: any start state must be one of the defined states in the automaton; the automaton cannot have an empty set of states; by definition, ε is not part of the alphabet; if a symbol is part of the automaton's alphabet, then there must be a transition performed by that symbol; all acceptance states must be states defined in the automaton; and, finally, any transition in the automaton must be a tuple, where the first and last elements are states in the automaton and the second element must be ε or a symbol accepted by the alphabet.

In the implementation of function `delta` below, the assertions guarantee that any reachable states must be in the automaton and that for every state in the result there has to be a transition in the automaton that reaches that same state through the given state `s` and symbol `symb`, and vice versa. The function `filter_trans` is responsible for obtaining every transition in the automaton where the first element matches the given state `s`, and the second element matches the symbol `symb`. Because `filter_trans` is a recursive function, we define a variant on the cardinality of the received set of transitions — that decreases in every recursive call — to prove the termination of the function. Lastly, the auxiliary function `delta_get_3rd`, whose implementation we omit, returns the set of the third element of every transition in a given set of transitions.

```
(* Returns states reachable from s through sym *)
let ghost function delta (s: state) (sym: symbol) (a: automaton) : fset state
  ensures  { subset result a.states }
  ensures  { forall c. mem (s, sym, c) a.transitions ↔ mem c result }
  ensures  { (not exists c. mem (s, sym, c) a.transitions) → is_empty result }
=
  let trans = filter_trans s sym a.transitions in
    delta_get_3rd trans

(* Returns a set of transitions with initial state s, performed by symbol sym *)
let rec ghost function filter_trans (s: state) (sym: symbol) (t:fset transition)
    : fset transition
  variant { cardinal t }
  ensures { subset result t }
```

```
14   ensures { forall a, b, c. mem (a, b, c) result → b = sym ∧ a = s }
15   ensures { forall c. mem (s, sym, c) t → mem (s, sym, c) result }
16   ensures { (not exists c. mem (s, sym, c) t) → is_empty result }
17  =
18   if is_empty t then empty
19   else let elem = pick t in
20        let (a,b,_) = elem in
21          if s = a && eq b sym then
22             add elem (filter_trans s sym (remove elem t))
23          else filter_trans s sym (remove elem t)
```

Listing 4.6: Implementation of the transition function (δ).

4.1.2 Compile

The implementation of the `Compile` (Listing 4.7) function is straightforward, based on the formal definition we have presented in Sec. 3.9. Therefore, following the definition of regular expression in Listing 4.2, we implement a function `compile` that, given a regular expression, returns the corresponding finite automaton, considering four cases: the empty regular expression ($R = \emptyset$), returning an automaton with a single (non-accepting) state; the empty word ($R = \varepsilon$), returning an automaton with a single transition through ε, and therefore with only one start state and one final state; the singular input symbol ($R = a$), returning an automaton with one initial state, one final state, one transition performed by a, and the alphabet containing only symbol a; the concatenation of two regular expressions ($R = EF$), returning an automaton resulting from joining the automata obtained from converting each of the regular expressions with an ε-transition; and the union of regular expressions ($R = E + F$), returning an automaton with a new initial states that transitions to either one of the automata, obtained from converting each of the regular expressions E and F, through an ε-transition.

According to what we have seen in Sec. 4.1.1, we begin by defining a precondition guaranteeing that any regular expression must be well-formed to perform the conversion (line 3). The postcondition in line 4 ensures that any acceptance state in the automaton must be reachable from another state. Ultimately, however, what we want to prove is the equivalence between the given regular expression and the resulting automaton. For that, and following the reasoning conducted in the previous chapter, we specify a postcondition stating that the language of the automaton, generated from the regular expression, is equal to the language of that same regular expression (line 6).

In the implementation of `compile`, you may notice the use of function `add_eps_trans` (line 34), whose implementation we have omitted. This function is responsible for joining automata `a` and `b`, by adding ε-transitions from each of the final states in `a` to the initial state of `b`, following the definition of `Compile` in Sec. 3.8.

```
1  (* Translates a given regular expression into an eps-NFA *)
2  let compile (r: regex) : automaton
```

```
  requires { regex_wf r }
  ensures  { forall f. mem f result.final_states
               → exists q, w. mem (q, w, f) result.transitions }
  ensures  { regexLang r = automatonLang result }
=
  match r with
  | Empty → let i = next_val c in
      { states = SS.singleton i;
        alphabet = SA.empty ();
        start = i;
        transitions = ST.empty ();
        final_states = SS.empty (); }
  | Eps → let start = next_val c in
        let final = next_val c in
        let states = SS.singleton start in
        let states = SS.add final states in
      { states = states;
        alphabet = SA.empty ();
        start = start;
        transitions = ST.singleton (start, eps, final);
        final_states = SS.singleton final }
  | Symb a → let start = next_val c in
        let final = next_val c in
        let states = SS.singleton start in
        let states = SS.add final states in
      { states = states;
        alphabet = SA.singleton a;
        start = start;
        transitions = ST.singleton (start, a, final);
        final_states = SS.singleton final }
  | Seq e f → let a = compile e in let b = compile f in
        let a_to_b = add_eps_trans a.final_states a.final_states b.start (ST.
    empty ()) in
      { states = SS.union a.states b.states;
        alphabet = SA.union a.alphabet b.alphabet;
        start = a.start;
        transitions = ST.union a_to_b (ST.union a.transitions b.transitions);
        final_states = b.final_states }
  | Plus e f → let i = next_val c in
        let a = compile e in let b = compile f in
        let trans = ST.add (i, eps, b.start) (ST.singleton (i, eps, a.start)) in
        let trans = ST.union trans a.transitions in
        let trans = ST.union trans b.transitions in
      { states = SS.add i (SS.union a.states b.states);
        alphabet = SA.union a.alphabet b.alphabet;
        start = i;
```

```
48          transitions = trans;
49          final_states = SS.union a.final_states b.final_states }
50    end
```

Listing 4.7: Implementation of the `Compile` function.

There are also a few other properties that can be ensured for the automata resulting from $R = \varepsilon$ or $R = a$. For example, we know that it is sure that, in these cases, there is only *one* accepting state:

```
(r = Eps  ∨  forall a. r = Symb a) → cardinal result.final_states = 1
```

We also know that, in these cases, every state of the resulting automaton has either an outward transition to another state or an inward transition from another state:

```
(r = Eps  ∨  forall a. r = Symb a)
  → forall s. mem s result.states → exists q, w. mem (s, w, q) result.
    transitions  ∨  mem (q, w, s) result.transitions
```

Following this reasoning, a more interesting idea is that all states in automata, resulting from these cases, must be *useful* (formalised in Sec. 4.1.3). A state is useful if it is *reachable* — there is a path from the initial state of the automaton to the said state — and productive — if that state can reach one of the automaton's final states.

```
(r = Eps  ∨  forall a. r = Symb a)
  → forall s. mem s result.states → useful s result
```

It is quite simple to understand why these assertions would not work for the cases where the regular expression is $\emptyset$ or one of the inductive cases (the concatenation and union of regular expressions). The empty case is elementary, given that there are no final states and no transitions. For the inductive cases, we have to think about the possibility of one of the regular expressions being $\emptyset$. In the case of the concatenation, creating an automaton that is disconnected (if $\emptyset$ is the first regular expression) or does not have final states ($\emptyset$ is the second regular expression). Whereas, in the case of the union, an automaton where one of the branches does not have final states. Knowing this, we can rewrite the first condition of each of the previous ensures as:

```
( r = Eps  ∨  forall a. r = Symb a  ∨
  (forall e f. e ≠ Empty ∧ f ≠ Empty ∧ (r = Seq e f  ∨  r = Plus e f)) )
```

4.1.3 Path

In the previous chapter, mainly during the definition of ε-closures (Sec. 3.5) and the extended transition function (Sec. 3.6), and just in the previous section on the subject of useful states, we have referred to the concept of path. Even though we have not given it a formal definition, having this notion will be helpful for the specification of the program.

The idea of path is to check if a given state s can arrive in a given state f, by computing a given word w, i.e., if state f can be reached from state s along a path w.

We formalise a predicate `path` (Listing 4.8) that takes two states, a word and the automaton on which we will check the path, using structural recursion over the given word — a list of symbols in our implementation. In WhyML, pattern matching considers three types of lists (or words, in our case): `Nil` — the empty list — representing the implicit empty word (equivalent to an explicit ε); the list with only one element, either a symbol of the alphabet or an explicit ε; and the list with more than one element — the word composed by multiple input symbols.

```
(* True if there is a path from state s to f, through word w. *)
predicate path (s: state) (w: word) (f: state) (a: automaton)
=
  match w with
  | Nil → path_eps s f a
  | Cons x Nil → mem x (add eps a.alphabet) ∧
              if x = eps then path_eps s f a
              else mem (s, x, f) a.transitions  ∨
                (exists q. (path_eps s q a) ∧ mem (q, x, f) a.transitions)  ∨
                (exists q, r. (path_eps s q a) ∧ mem (q, x, r) a.transitions ∧
                path_eps r f a)
  | Cons x xs → xs ≠ Nil ∧ mem x (add eps a.alphabet) ∧
                 exists r. ( mem (s, x, r) a.transitions  ∨
                 (exists q. (path_eps s q a) ∧ mem (q, x, r) a.transitions) ) ∧
                 path r xs f a
  end
```

Listing 4.8: Predicate path.

Remember that, after each state, there may be multiple ε-transitions. Following the reasoning used in the cases where the word is ε and that of using `ECLOSE` in the definition of the extended transition function δ^*, we use a predicate `path_eps` to account for the possibility of ε-transitions in-between states, which we implement in the listing below.

```
(* True if there is a path through successive eps-transitions from one state to
    the other. *)
inductive path_eps state state automaton
=
  | path_base: forall s a. mem s a.states → path_eps s s a
  | path_ind: forall s q r a. mem (s, eps, q) a.transitions → path_eps q r a →
     path_eps s r a
```

Listing 4.9: Predicate that checks a path through successive ε-transitions, between two given states.

Notice the condition in the base case for `s` to be a defined state in the automaton. This is necessary because we can only say there is a path for a state in the automaton if it is part of the automaton. Since in the inductive step we need to check if a transition is in the automaton and the recursive call might hit the base case, it is not necessary to check

if any of the states is in the automaton. The same is also true for the definition of `path` in Listing 4.8.

We can now formalise the concept of useful state by means of a predicate path (Listing 4.10). Recall that a state is useful if it is *reachable* — there is a path from the initial state of the automaton to the said state — and productive — if that state can reach one of the automaton's final states.

```
(* True if there is path from the start state of the automaton a to the given
    state s. *)
predicate reachable (s: state) (a: automaton)
=
  exists w. path a.start w s a

(* True if a given state s can reach a final state of automaton a. *)
predicate productive (s: state) (a: automaton)
=
  exists w, q. mem q a.final_states ∧ path s w q a

(* True if a state is both reachable and productive. *)
predicate useful (s: state) (a: automaton)
=
  reachable s a ∧ productive s a
```

Listing 4.10: Predicates for reachable, productive, and useful state.

4.1.4 Language of an NFA

Let us now rewind to the last postcondition of `compile` (Listing 4.7), stating that the language of the given regular expression must be equal to that of the resulting automaton.

Recall the definition of language of an NFA, we have presented in Sec. 3.7:

$$L(A) = \{w \in \Sigma^* \mid (\delta^*(s_0, w) \cap F) \neq \emptyset\}$$

We can immediately notice a problem with this definition: Σ^* is *not* a finite set, something that is hard to reason about and, furthermore, the definition by comprehension can be somewhat complicated. A first approach would be to just think about the three base cases and implement a constructive function, as in the listing below, where the definition of language is based on the cardinality of the set of final states or the set of transitions. However, not only would we have to think about a possible implementation for the inductive cases, but we would also have to formalise assertions making sure the constructive implementation is equivalent to the mathematical definition.

```
function automatonLangConstr (a: automaton) : fset word
=
  if is_empty a.final_states then empty
  else
```

```
    let tail =
      if mem a.start a.final_states then singleton (Cons eps Nil)
      else empty
    in
      if is_empty a.transitions then tail
      else
        if cardinal a.transitions = 1 then
          let (s, x, q) = pick a.transitions in
            if s = a.start && mem q a.final_states then add (Cons x Nil) tail
            else tail
        else tail (* inductive steps *)
```

Listing 4.11: Constructive implementation of the automaton language.

Fortunately, WhyML's Set library[3] makes available a function `filter` (in Listing 4.12) that allows us to define a set by comprehension, making the implementation of the function automatonLang quite straightforward, regarding the mathematical definition.

```
(* { x | x in s ∧ p x } *)
function filter (s: fset 'a) (p: 'a → bool) : fset 'a
axiom filter_def:
  forall s: fset 'a, p: 'a → bool, x: 'a. mem x (filter s p) ↔ mem x s ∧ p x
```

Listing 4.12: Function filter.

You can see that there is a clear resemblance to the formal definition of language, where set s would be Σ^* and predicate p would be $(\delta^*(s_0, w) \cap F) \neq \emptyset$. However, the problem of how we can represent Σ^* in WhyML remains an issue. We have first to understand what it really means for a word to belong to this non-finite set. Being defined over the alphabet Σ, its closure is, thus, the set of *all* words that can be formed with symbols in the alphabet and, while it is hard to reason about the actual content of an infinite set — the actual words that are part of it — we can reason about their properties or, perhaps better put, their structure. For example, we know that all words in Σ^* are *exclusively* made up of symbols in Σ. Therefore, we can write a function `sigma_ext` (naturally receiving an alphabet as argument) that, although cannot have an implementation (which is not a problem, given that it is only used in the logical context), we can define its properties through pre and postconditions, as in the listing below.

```
val function sigma_ext (a: fset symbol) : fset word
  ensures {forall w. mem w result → w ≠ Nil}
  ensures {forall w. mem w result → forall x. mem x w → mem x (add eps a)}
  ensures {forall x w. mem x w ∧ mem x (add eps a) ∧ w ≠ Nil → mem w result}
```

Listing 4.13: Formalizations of Σ^*.

The definition of a predicate `accepted_words` (Listing 4.14) that, as suggested by its signature, checks if a word is accepted by a given automaton, is quite simple, with an

[3] http://why3.lri.fr/stdlib/set.html

implementation that follows very closely the definition in language $((\delta^*(s_0, w) \cap F) \neq \emptyset)$. Once again, you can see the use of predicate `path` in its postcondition. Indeed, we have to be sure that if the result is true, i.e., if the word w is accepted, then there is a final state that can be reached from the initial state of the automaton along a path w, and *vice versa*.

```
(* True if word w leads the start state of the automaton a to a final state *)
let ghost predicate accepted_words (a: automaton) (w: word)
  ensures { result ↔ exists f. mem f a.final_states ∧ path a.start w f a }
=
  not ( is_empty (inter (delta_ext a.start w a) a.final_states) )
```

Listing 4.14: Predicate `accepted_words`.

With function `sigma_ext` and predicate `accepted_words` now defined, we can make use of function `filter` (presented in Listing 4.12), to formalise the language of the automaton in a manner resembling the mathematical definition of Sec. 3.7, as shown in the listing below.

```
(* Returns the language of an automaton a *)
let ghost function automatonLang (a: automaton) : fset word
=
  filter (sigma_ext a.alphabet) (accepted_words a)
```

Listing 4.15: Formalization of the automaton's language.

The specification of `automatonLang` is quite delicate since it will be the main responsible for the result we will be proving on `compile`. We begin by making sure that, in the automaton received as an argument, all final states can be reached by another state. Note that, this is quite different from saying these states are reachable, which would mean there is a path from the start state of the automaton to the final state. This is certainly not true in a case where, for example, the automaton received was generated by a regular expression $R = \emptyset \cdot E$, where E could be any kind of regular expression. Nonetheless, it serves to ensure the automaton is constructed according to the algorithm since final states are originally created by the base cases for $R = \varepsilon$ and $R = a$, where the assertion is unmistakably true.

```
let ghost function automatonLang (a: automaton) : fset word
  requires { forall f. mem f a.final_states
                → exists q, x. mem (q, x, f) a.transitions }
```

Surely, we know that if a word is in the language of the automaton then it must be an element of Σ^*, just like stated by mathematical definition and implementation.

```
  ensures { forall w. mem w result → mem w (sigma_ext a.alphabet) }
  ensures { forall w. mem w result → w ≠ Nil }
```

We also know that if there are no final states, no words are accepted by the automaton and, thus, the language is empty (line 1) and, certainly, if the language is not empty, then there must be acceptance states (line 2).

```
1  ensures { is_empty a.final_states → is_empty result }
2  ensures { result ≠ empty → a.final_states ≠ empty }
```

If the initial state of the automaton is also a final state, then the empty word ε is accepted by the automaton and therefore part of the language (line 1). In fact, if this is the case and, besides, there are no transitions at all, the language of the automaton is a set with only the empty word (line 2). While if there are no transitions but the start state is not final, the language is empty (line 4).

```
1  ensures { mem a.start a.final_states → mem (Cons eps Nil) result }
2  ensures { (is_empty a.transitions ∧ mem a.start a.final_states)
3              → result = singleton (Cons eps Nil) }
4  ensures { (is_empty a.transitions ∧ not mem a.start a.final_states)
5              → result = empty }
```

Similarly, if there is a direct transition from the start state of the automaton to a final state, the symbol performing the transition is an element of the language (line 1). And, if it is the only transition and the start state is not final, it is the only accepted word by the language (line 2).

```
1  ensures  { forall x f. mem f a.finalStates ∧ mem (a.start, x, f) a.transitions
     → mem (Cons x Nil) result }
2  ensures  { forall x f. mem f a.finalStates ∧ a.transitions = singleton (a.
     start, x, f) ∧ not mem a.start a.finalStates → result = (singleton (Cons x
     Nil)) }
```

Intuitively, following the previous reasoning and the one used when implementing automatonLang, we can conclude that either the language is empty or there is a word w such that $w \in \Sigma^*$, and it is accepted by the automaton $((\delta_E^*(s,w) \cap F) \neq \emptyset)$ (line 1). Equivalently, we can ensure that all words that are in the language are accepted by the automaton and are elements of Σ^* (line 2).

```
1  ensures { is_empty result  ∨  (exists w. mem w result ∧ mem w (sigma_ext a.
     alphabet) ∧ not is_empty (inter (delta_ext a.start w a) a.final_states)) }
2  ensures { forall w. mem w result → ( inter (delta_ext a.start w a) a.
     finalStates ≠ empty ∧ mem w (sigma_ext a.alphabet) ) }
```

Similarly to the postcondition of accepted_words (and, in fact, due to it), the computation of automatonLang should ensure that if a word is in the result then the set of final states is not empty, there is a path along that same word from the start state of the automaton to one of the final states, and the word is an element of Σ^*.

```
ensures { forall w. mem w result → (a.finalStates ≠ empty ∧ exists f. mem f a
  .finalStates ∧ path a.start w f a ∧ mem w (sigma_ext a.alphabet)) }
```

The reverse should also be true. And since Nil cannot be part of the language, whenever it is accepted, the word ε must be in the result (as if in replacement of Nil).

```
1  ensures { forall w. (w ≠ Nil ∧ mem w (sigma_ext a.alphabet)
```

```
    ∧ inter (delta_ext a.start w a) a.finalStates ≠ empty) → mem w result }
  ensures { forall w f. (w ≠ Nil ∧ mem w (sigma_ext a.alphabet) ∧ mem f a.
    finalStates ∧ path a.start (reverse w) f a) → mem w result }
  ensures { forall w. (w = Nil ∧ inter (delta_ext a.start w a) a.finalStates ≠
    empty) → mem (Cons eps Nil) result }
```

4.1.5 Extended Transition Function

We have now to formalise the function `delta_ext` that is used in Listing 4.14 to assess if a word is accepted by an automaton.

Recall the formal definition of extended transition function, presented in Sec. 3.6:

$$\delta^*(s,\varepsilon) = \text{ECLOSE}(s)$$
$$\delta^*(s,ua) = \{p \mid \forall r,q.\ r \in \delta^*(s,u)\ \wedge\ q \in \delta(r,a)\ \wedge\ p \in \text{ECLOSE}(q)\}$$

Notice that δ^* is defined recursively starting from the right end of the word $w = ua$. Based on this definition, we implement the extended transition function using pattern matching and structural recursion on a given word w, as can be seen in Listing 4.16. Given that the type `word` is defined as a list of symbols and pattern matching on lists in WhyML is done on a head-tail "style", if we were to receive a word that is structurally identical to w, we would not have direct access to the last symbol a. Therefore, to make an implementation resembling that of the mathematical definition, we assume the word `w` in `delta_ext` to be reversed, where the last symbol (of what would be the normal word) is at the head of the list.

Because we are reasoning over a recursive function, we have to define a condition to guarantee its termination. Since in this case we are doing structural recursion over the list of symbols `w`, and we are removing elements from the list in every recursive call, we can define the termination with a variant over the length of word `w`.

```
let rec ghost function delta_ext (s: state) (w: word) (a: automaton): fset state
  variant { length w }
=
  match w with
  | Nil → eclose s a
  | Cons x Nil → if eq x eps then eclose s a
                 else let rs = eclose s a in
                      let qs = fold_delta rs x a in
                          fold_eclose qs a
  | Cons x u →  let rs = delta_ext s u a in
                let qs = fold_delta rs x a in
                    fold_eclose qs a
  end
```

Listing 4.16: Implementation of the extended transition function δ^*.

Similarly to the formal definition, our function takes as arguments a single state, a word, and an automaton (which in contrast with the mathematical definition, needs to be an argument), and returns a set of reachable states under the given word, from the given state. Just like the mathematical definition, we begin by considering the case where the input string is the empty word ε, represented implicitly by `Nil` or explicitly by `eps`. In this case, we need only to compute the ε-closure of the given state (lines 5 and 6). Otherwise, we apply similar reasoning to the inductive step of δ^*. For the case where the word is a single input symbol ($w = a$), the recursive call would ($\delta^*(s, u)$) be the result of `ECLOSE` (since $u = \varepsilon$) on the given state (line 7). Otherwise, we perform the recursive call on δ^* to obtain all states reachable from `s` along a path `u` (line 10). From there, for each of those states, we compute the transition through symbol `x` (lines 9 and 11) — equivalent to the $\delta(r, a)$ step — and account for the possibility of subsequent ε-transitions (lines 9 and 12).

In the interest of keeping this section brief, we omit the implementation of functions `fold_delta` and `fold_eclose`. As implied by the above explanation of function `delta_ext`, `fold_delta` takes a set of states and a symbol and, by applying function `delta` (recall Listing 4.6) to each state in the given set, outputs a set with all reached sates. Function fold_eclose computes the ε-closure (which we will be formalizing in the next section), for each state in a given set of states, outputting a set with all reached states through ε-transitions. Both functions have also to be specified in order to help verify the correction of `delta_ext`. Usually annotated with postconditions that may directly resemble those of the main function `delta_ext` (which we will be specifying ahead), for example, ensuring that for all states in the result of `fold_eclose` there must be a path through a sequence of ε-transitions from one of the states in the initial set, or *vice versa*.

Hence, we now have to specify the pre and postconditions to ensure the correct usage and function of `delta_ext`, which will be know of the function in the context of proof. Perhaps most obvious is the fact that every state in the set returned by `delta_ext` must be a state of the given automaton:

```
ensures  { subset result a.states }
```

More interesting would be to define postconditions that, in some way, help reason not only about the contents of the result, but also about the process of how we got there. Intuitively, we understand that, just like its mathematical counterpart in Sec. 3.6, `delta_ext` returns a set of states reachable along a *path* `w`. Therefore, we could be tempted to define a postcondition such as the one below:

```
ensures { forall r. path s w r a → mem r result }
```

However, recall that we are reasoning over a word that is flipped. Therefore, it is not state `r` that is reachable from state `s` along a path `w`, but the other way around. To tackle the issue, we have implemented a predicate `htap` that checks if there is a path along a word between an arriving state and a departing state, whose implementation follows very

closely that of path in Listing 4.8 and can be viewed on our online repository. We can now, rewrite the previous postcondition as:

```
ensures { forall r. htap r w s a → mem r result }
ensures { forall r. path s (reverse w) r a → mem r result }
```

Notice we have also included a postcondition using predicate path with the word w being reversed. Indeed, one might ask why not just use path with the flipped word instead of implementing another predicate that, surely, required a reverse thought process. However, and without venturing too much in the subject of Sec. 4.2, the postcondition with predicate htap is easier to prove since the predicate and the implementation of delta_ext follow the same flow (or sequence) of execution, consuming the word from the last symbol to the first. Whereas path with a flipped word starts by consuming the first symbol.

Similarly, it should also be true that, if a state is in the result of delta_ext, it is because there is a path between r and the given state s:

```
ensures { forall r. mem r result → htap r w s a }
ensures { forall r. mem r result → path r (reverse w) s a }
```

We have chosen not to specify preconditions since the only true pre-requisite on the function is the type of arguments it receives which is already guaranteed by its signature. On a first impression, one might think we should require the state we receive to be one of the defined states in the automaton, or the word to be composed by elements of the alphabet or ε. However, these would be stronger than necessary, as these functions have to naturally check these kinds of requirements while computing, not solely for avoiding execution problems but because it is actually how they work — for example checking if a transition exists or not — and, therefore, no problems might arise from not having these as preconditions. The same reasoning also lead us to not define preconditions, for example, for function delta in Listing 4.6.

Taking into account the issue regarding the necessity for the word to be reversed in order to compute delta_ext, all other functions depending on it (such as accepted_words and automatonLang) must also conform to these changes, for example changing their postconditions using path to work over reverse w instead. Regardless, in case we just want to use the specification without the implementation, this is not a problem, since we can assume a possible implementation where the word follows what would be an expected natural construction, leaving the correct specification of the function on our basis of trust. However, further down this chapter, we will look into a more elegant solution to this problem. Nonetheless, delta_ext proves to be a great example of how implementation can influence the specification of a program.

4.1.6 ECLOSE

The implementation of the ECLOSE function (Listing 4.17) presented a rather challenging task, just like its formal definition. We begin by recalling that this function is responsible

for computing the states reachable from a given state through a sequence of ε-transitions, which explains the use of a recursive definition. This also results in an implementation that receives a set of states instead of a single one, contrary to the formal definition of ECLOSE. With the help of `fold_next_states`, we compute the set of states reachable through *one* ε-transition, for each state in a given set, including that same state. If the obtained set is a subset of the initial — no new states were reached — no further calculations are needed. Otherwise, we must compute the `eclose` of the obtained states.

It may seem as though function `eclose` does not terminate and, indeed, the set computed by `fold_next_states`, can only increase in the number of elements or remain unchanged. However, we point to the fact that we are dealing with finite automata, which means that we are dealing with a finite number of states to explore, thus guaranteeing the function eventually returns a result.

```
1  (* Computes all states reachable from each state in ss through successive eps-
       transitios. *)
2  let rec ghost function eclose (ss: fset state) (a: automaton): fset state
3    variant  { cardinal a.states - cardinal ss }
4  =
5    if is_empty a.transitions then ss
6    else let ns = fold_next_states ss a in
7         if subset ns ss then ns
8         else eclose (union ss ns) a
9
10 (* Obtains the next states through eps-transitions from each state in s. *)
11 let rec ghost function fold_next_states (ss: fset state) (a: automaton): fset
       state
12   variant  { cardinal ss }
13 =
14   if is_empty ss then empty
15   else if is_empty a.transitions then ss
16   else let q = pick ss in
17        let nxt_sts = add q (delta q eps a) in
18        union nxt_sts (fold_next_states (remove q ss) a)
```

Listing 4.17: Implementation of the ECLOSE function.

The problem with this implementation is not only the fact that it is different in terms of the arguments it receives, compared to the mathematical definition, but in terms of the flow of execution. The breadth-first structure, besides being different from the mathematical definition of ECLOSE in Sec. 3.5, makes it harder to reason about the sequence in which we perform the transitions and, therefore, about the path we are following. Remember that the idea of the function is to find all states reachable from a given state s along a path of consecutive ε-transitions, so this should be something we are able to ensure.

In order to tackle the issue, we now dive in a depth-first approach to the formalization

of function `eclose`, that more closely resembles the formal definition, which we recall below.

$$\mathsf{ECLOSE} \triangleq \left\{ (s, Q) \;\middle|\; Q = \{s\} \cup \{\, r \mid \forall q.\ q \in \delta(s, \varepsilon) \ \wedge\ q \neq s \ \wedge\ r \in \mathsf{ECLOSE}(q) \,\} \right\}$$

We begin by creating a wrapper-function `eclose` that, just like the above definition, receives a single state as an argument, as well as the automaton. We need this function to make sure we start the execution with an empty sequence of visited nodes, a structure that will make sure we do not visit a state twice. The actual algorithm runs on function `eclose_n` that checks if a given state was already visited (line 10), computes its ε-transitions (line 12), and computes the ε-closure for each of the reached states (line 13). Function `fold_next` is thus responsible for repeating the algorithm for each of those states. As you can see, this happens using mutual recursion, which, on its own, is a problem when proving the termination of the functions.

```
1  (* Depth-first version of ECLOSE *)
2  let ghost function eclose (s: state) (a: automaton) : fset state
3  =
4    if is_empty a.transitions then singleton s
5    else eclose_n empty s a
6
7  (* Computes the eclose for state q: all states reachable from q through
       successive epsilon-transitios. *)
8  let rec ghost function eclose_n (visited: seq state) (q: state) (a: automaton) :
       fset state
9  =
10   if not mem q visited then
11     begin
12       let ns = (delta q eps a) in
13       fold_next (snoc visited q) ns a
14     end
15   else to_set visited
16
17 with ghost function fold_next (visited: seq state) (ns: fset state) (a:
       automaton) : fset state
18 =
19   if is_empty ns then to_set visited
20   else if subset ns (to_set visited) then to_set visited
21   else let q = pick (diff ns (to_set visited)) in
22        union (eclose_n visited q a) (fold_next visited (remove q ns) a)
```

Listing 4.18: Depth-first implementation of ECLOSE.

As in the formal definition, we must ensure the ε-closure of a state includes the own state (line 2). No less obvious, is that all states in the result are states of the given automaton (line 3), or that, if there are no transitions, the ε-closure of one state is the singular set with the own state (line 4).

```
let ghost function eclose (s: state) (a: automaton) : fset state
  ensures { mem s result }
  ensures { subset result a.states }
  ensures { is_empty a.transitions → result = singleton s }
```

We could also specify postconditions regarding the contents of the result. For example, if a state is in the result of function `eclose`, it is either because it is the starting state or because one other state in the result can reach that state through an ε-transition (line 1). And the reverse case should also be true: if a state is part of the reached states and has a ε-transition to one other state, that state he can reach should also be part of the result (line 3).

```
  ensures {forall f. mem f result →
            f = s  ∨  (exists i. mem i result ∧ mem (i, eps, f) a.transitions)}
  ensures {forall i, f. mem i result ∧ mem (i, eps, f) a.transitions
            → mem f result}
```

However, we can be even more ambitious. In fact, recall that `eclose` finds states along a path of successive ε-transitions. Thus, we should be able to ensure that for all states to which we have an "ε-path", these are in the result of the function `eclose` (line 1). Similarly, if a state is in the result, surely there must be a path through ε-transitions leading to it from the starting state (line 2).

```
  ensures  { forall r. (path_eps s r a) → mem r result }
  ensures  { forall r. mem r result → path_eps s r a }
```

While it is also necessary to specify the functions `eclose_n` and `fold_next`, we omit the assertions we have designed for them, not only because it would make this section increasingly longer and cumbersome — for example, having to reason about the set of visited states in every assertion, or postconditions being interdependent because of the mutual recursion — but also because only `eclose`'s assertions will be present in the context of proof for other functions that use it (`eclose_n` and `fold_next`'s postconditions are, however, necessary to prove those in `eclose`). Moreover, for the same reason we have not defined preconditions for `delta_ext`, we also do not define them for `eclose`.

4.2 Verification

With the program implemented and fully specified, we can move on to the mechanical verification of the implemented procedures. For each lemma, axiom, annotated function, invariant or assertion, Why3 generates a *verification condition* (VC) to be proved. Since Why3 is oriented towards automatic proofs, supporting many external automatic theorem provers, to solve the validity of a verification condition we can call on provers such as Alt-Ergo, Z3 or CVC4. And while, as mentioned in Chapter 2, Why3 also supports interactive proof assistants such as Coq or Isabelle, in this thesis we are just focused on the use of SMTs.

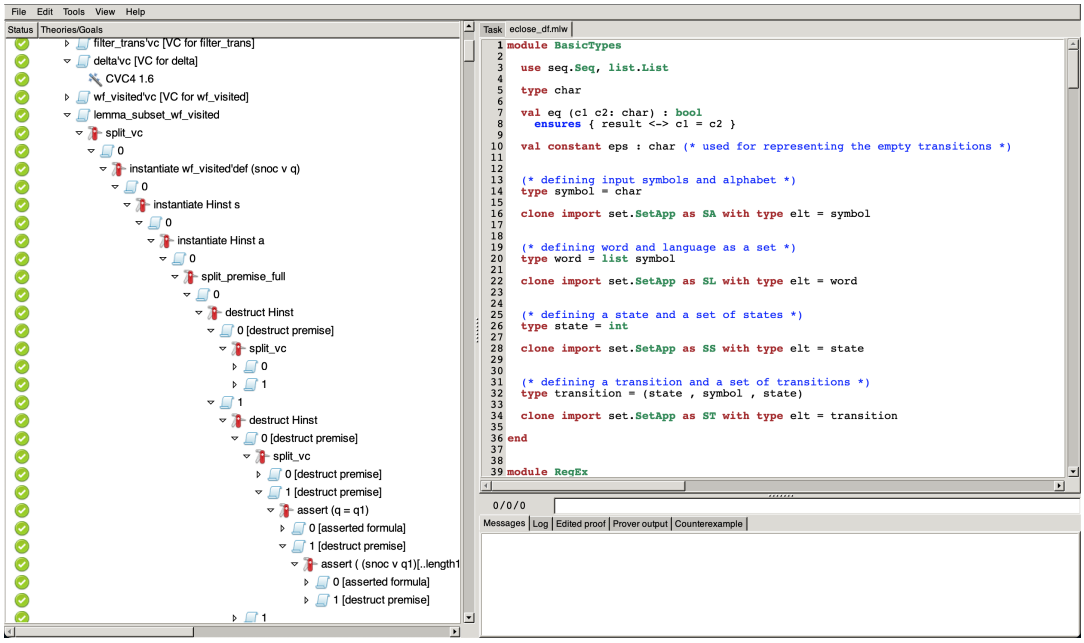


Figure 4.1: The Why3 IDE graphical interface.

It can happen that the provers can easily discharge a selected verification condition, just like `delta` in Fig. 4.1, or that we need to make formulas and annotations easier to prove by applying logical transformations in order to help provers discharge them, as for `lemma_subset_wf_visited`. Naturally, the feedback we get from the solvers in each VC — either because it was able to prove the contract, timed out because it was not able to complete the proof in a certain time, or even was able to refute it — will influence the way we proceed with our proof. An unsuccessful attempt to prove a verification condition can be an indicator of: an insufficient specification; the simple incapacity of the prover, leading to the necessity of applying *tactics*, we have mentioned before;or even a bug or incorrect implementation of the code.

During our verification effort, we came across many of these issues. Some of them, we have already mentioned in the previous sections. Our approach was to first implement every function we thought would be necessary to produce a complete program, with all concepts introduced in Chapter 3. Gradually annotating the code with pre and postconditions and introducing lemmas, as they became relevant to the proof of our implementations and in regards to what seemed to be lacking. This methodology brings both advantages and disadvantages. On one hand, focusing on proving the correctness of functions that are just needed for the logical context of proof, such as `eclose`, `delta_ext` or `automatonLang`, is time-consuming and draws us away from tackling what could be considered the main proof: the equality between the languages of regular expression and NFA. On the other hand, it creates an iterative process where we are able to build stronger specifications — annotating, asserting and creating lemmas as they become necessary — that not only might help in proving other functions correct, but also increase confidence

in the implementation, by becoming aware of properties that should be true and, most importantly, able to be proven regarding the implementation.

ECLOSE. Function `eclose` was perhaps the most challenging and time-consuming to implement and, especially, verify. A breadth-first implementation (Listing 4.17) makes it harder to reason over the concept of path, which becomes a problem when other implementations, such as that of `delta_ext`, depend on `eclose` and, therefore, its specification. A depth-first implementation solves this issue, following a flow of execution comparable to that of predicate `path`. However, this alternative introduces other issues that are equally difficult to solve. The necessity of avoiding checking a state twice introduces new structures that have to be specified and controlled, such as sequence `visited` in Listing 4.18. For example, by having to guarantee that states are added to `visited` sequentially, in order to ensure that all states along a path of ε are reached. Otherwise, `visited` could have any state — before and after the starting state — and could not guarantee all states along the path would be visited. Furthermore, mutual recursion raises problems related to the termination and interdependence between pre and postconditions of the functions `eclose_n` and `fold_next`, where the termination and contracts of a function have to be proven in relation to the other, which greatly increases the complexity of proof and, consequently, that of the wrapper function `eclose`.

However, we have managed to solve many of these issues and discharge most of the verification conditions for `eclose`, `eclose_n` and `fold_next`. However, the proof of termination of the last two functions remains an open issue. And so is true for the proof of a postcondition ensuring that, if a state is in the result, there must be a path from the starting state to it. The proof of this postconditions is dependent on a similar proof for `eclose_n` and `fold_next`, but requires a more thorough exploration due to the problems we have mentioned above, regarding mutual recursion.

In Sec. 3.5, we have made reference to two properties of ECLOSE, that could be interesting to include in our program. Even though, at this point, they are not entirely necessary for our procedures, we formalise them in Listing 4.19. Lemma `close_eclose` states that, if a state q is in the ε-closure of a state s ($q \in \text{ECLOSE}(s)$) and there is an ε-transition to any other state r, then state r must also be an element of ECLOSE(s). Lemma `subset_eclose` states that, for any state $q \in \text{ECLOSE}(s)$, its ε-closure is a subset of that of s ($\text{ECLOSE}(q) \subseteq \text{ECLOSE}(s)$).

```
1  (* If q in ECLOSE(s) and there is an eps-transition from q to a state r, then
     r in ECLOSE(s) *)
2  lemma close_eclose:
3    forall a:automaton, s q r:state.
4      mem s a.states ∧ mem q (eclose s a) ∧ mem (q, eps, r) a.transitions
5        → mem r (eclose s a)
6
7  (* If q in ECLOSE(s), then ECLOSE(q) must be a subset of ECLOSE(s). *)
```

```
8  lemma subset_eclose:
9    forall a:automaton, s:state, q:state. let ecl = (eclose s a) in
10     mem s a.states ∧ mem q ecl → subset (eclose q a) ecl
```

Listing 4.19: Properties of function `eclose`.

The former is easily proven by Why3. The latter, however, might seem obvious, especially when paired with the first, but is dependent (and proven) by a property of transitivity for `path_eps` — if there is a ε-path from a state x to a state y and from y to a state z, then there is a path from x to z — but which we have yet to prove.

Extended Transition Function. Assuming the specification of `eclose`, the proof of `delta_ext` did not require much effort, as we were able to prove each and every one of its postconditions. However, as you may recall from Sec. 4.1.5, predicate `path` parses the word in natural order, from the first symbol of the word to the last. This makes it harder to compare with the execution of function delta_ext, which does it in the reverse order. For this reason, we have defined a predicate `htap`, that works very similarly to `path` but receives a word assumed to be flipped, just like `delta_ext` and, therefore, without much transformations, is simple to prove. Since other functions are dependent on `path`, we have also created postconditions using the normal predicate but with the reversed word (that which would be considered in the normal order). However, the problem remains that `path` and `delta_ext` parse the word in different directions. With the creation of `htap` we can now define a lemma ensuring that having one implies the other, as shown in the listing below.

```
1  lemma path_htap:
2  forall x y: state, w: word, a:automaton.
3    path x w y a ↔ htap y (reverse w) x a
```

Listing 4.20: A path between x and y through w, implies a path between y and x through an inverted word w.

Due to its complexity, and the reason that drove us to define `htap` in the first place, we have not begun the proof of this lemma yet. Trusting that its result is true and assured that it should be proven, its proof is one of our future tasks.

An alternative in consideration is the redefinition of type `word` as a sequence of symbols, rather than a list. This would not grant us the commodities of pattern-matching, but would allow us to work with the word in its correct order — since sequences allow direct access to the last element — and perhaps be easier to prove a postcondition with `path`, even if they parse the sequence in different orders. Furthermore, it would allow us to work solely with the symbols, not having to consider the case where the list (or word) is `Nil`. Which, even though representing ε, as you could see in the implementation of language in Sec. 4.1.4, cannot be an element of the language. Hence, this change would also simplify any confusion that might be introduced by a `Nil` word.

Language of an NFA. The specification of `delta_ext` proved to be strong enough since, allied with the specification of `sigma_ext` (Listing 4.13), it helped discharge the post-conditions proposed for the implementation of `automatonLang` (Listing 4.15). This, of course, having in mind the fact that we are now considering the word in reverse because of `delta_ext`, which, as you may recall, is necessary for the definition of `accepted_words`.

Unfortunately, we have noticed a problem raised by the use of reversed words, as necessary for `delta_ext`: the elements of the language are also reversed words, since these are the ones we are being filtered by predicated `accepted_words` (Listing 4.14). This means that, when we compute the automaton language for any of the inductive steps (the basis does not have this problem, given that the language is either empty or has only one symbol), the words will be different from those of the language of the regular expression and, therefore, make it impossible to actually prove their equality.

There are several alternatives to this. The first we have already presented, which would be to change the type of `word` from list to sequence, and consequently have it and its predicates use the word in a natural order. However, this solution would require a complete restructuring of the code, something that does not answer to the immediacy of the problem. The second alternative would be to reverse all of the obtained words in the implementation of `automatonLang`, i.e., after the computation of the function, reverse all of the elements in the language. Fortunately, there is a solution that is both more elegant and easy to implement, without a complete redefinition of either `delta_ext` or `automatonLang`, or of the code as a whole.

As we have mentioned above, predicate `accepted_words` is the one responsible for filtering the words that are accepted by the language, by checking which words can reach the final states, starting from the initial state of the automaton. Recall the current definition of predicate `accepted_words` in the listing below, according to the changes made necessary by the implementation of `delta_ext`.

```
1 let ghost predicate accepted_words (a: automaton) (w: word)
2   ensures { result ↔
3               exists f. mem f a.final_states ∧ path a.start (reverse w) f a }
4 = not (is_empty (inter (delta_ext a.start w a) a.final_states))
```

Listing 4.21: Current formalization of predicate `accepted_words`.

A smarter solution would be to consider word `w` to be ordered naturally, and instead of flipping it for predicate `path` in the postcondition, do it for `delta_ext`, as in Listing 4.22 below. This will allows us to use word `w` normally in function `automatonLang` as well, and just like in predicate `accepted_words`, use `(reverse w)` for calls on `delta_ext` and `w` for calls on predicate `path`. Additionally, this also makes it easier to read and understand the code of functions depending on `path` or `delta_ext`.

```
1 let ghost predicate accepted_words (a: automaton) (w: word)
2   ensures { result ↔ exists f. mem f a.final_states ∧ path a.start w f a }
```

```
= not (is_empty (inter (delta_ext a.start (reverse w) a) a.final_states))
```

Listing 4.22: Correct formalization of predicate `accepted_words`.

Proof of Equivalence. Much like the mathematical proof introduced in the previous chapter, the proof for the base cases ($R = \emptyset$, $R = \varepsilon$ and $R = a$) was quite simple and with little exploration. The other postconditions introduced in Sec. 4.1.2 for function `compile` were mostly straightforward as well, both for the basis and inductive steps. However, much like the mathematical proof, the complexity of the proof becomes evident when we try to prove the equality of the languages for regular expression and automaton, for the concatenation and union. There are several issues that might have to be considered here, one of them quite delicate.

Recall that, by definition of the concatenation of languages, presented both in Sec. 3.3 and Listing 4.2, every character of the concatenation is explicit, including ε. Whereas in an ε-NFA, the computation of ε is automatic since $\varepsilon w = w$ and $w\varepsilon = w$. This does not mean that any of the computations is wrong or that it prevents the proof of equality. It does, however, require a bit more delicate exploration of the proof.

In Sec. 3.9, we proved that the language of the concatenation of two automata is equal to the concatenation of the languages of those two automata, based on the idea that all words accepted by the resulting automaton, naturally, have to go through the first automaton of the concatenation and then the second. In other words, this means that words accepted by the resulting automaton are concatenations of words accepted by the first automaton and words accepted by the second automaton. We defined this by means of Lemma 3, recalled below.

$$\begin{aligned}&\forall w \in \Sigma^*.\ \delta^*_{EG}(s_{0_E}, w) \cap F_G \neq \emptyset \\ &\quad \implies \exists u \in \Sigma^*_E,\ \exists v \in \Sigma^*_G.\ \ w = uv \ \wedge\ (\delta^*_E(s_{0_E}, u) \cap F_E) \neq \emptyset \ \wedge\ (\delta^*_G(s_{0_G}, v) \cap F_G) \neq \emptyset\end{aligned}$$

We would like to have a similar result in our program, in the hopes that this can help our mechanically checked proof. As such, we implemented the previous lemma:

```
lemma word_concat:
  forall a b c: automaton, w: word. c = automaton_concat a b ∧
    mem w (sigma_ext c.alphabet) ∧
    inter (delta_ext a.start (reverse w) c) b.finalStates ≠ empty →
      exists u v: word.
        mem u (sigma_ext a.alphabet) ∧ mem v (sigma_ext b.alphabet) ∧
        w = Append.(++) u v ∧
        inter (delta_ext a.start (reverse u) a) a.finalStates ≠ empty ∧
        inter (delta_ext b.start (reverse v) b) b.finalStates ≠ empty

lemma alang_concat:
  forall a b c: automaton, w: word.
    let le = automatonLang a in let lf = automatonLang b in
```

```
14      let lef = automatonLang c in
15        c = automaton_concat a b ∧
16        mem w lef → exists u v. w = Append.(++) u v ∧ mem u le ∧ mem v lf
```

Listing 4.23: Lemma 3 formalised in WhyML.

Lemma `alang_concat` just better translates the meaning of the first one, and therefore is easily proven, while being even more helpful. Note that we are omitting the conditions that respect the preconditions of `automatonLang`, in the interest of simplicity and legibility. However, while the definition of Lemma 3 may be clear enough for us to conclude the mathematical proof, `Why3` requires a greater exploration or the definition of clearer properties. In our case, we had to define a property regarding the equality of languages (or sets):

```
1  lemma set_equality:
2    forall x y: fset 'a.
3      (forall a. mem a x → mem a y) ∧ (forall a. mem a y → mem a x)
4        → x = y
```

Listing 4.24: Set equality.

This lemma is obvious and easy to discharge, and means that, if we want to prove that, for example, `lef = (concat le lf)`, it is not enough to prove that every word in `lef` is in `(concat le lf)`, but also the other way around. The former condition is easy to prove using the previous two lemmas, the latter is not as simple. Therefore, we must define a lemma similar to Lemma 3 (or `word_concat`) but stating that, for any two words u and v, respectively accepted by an automaton A and B, the concatenation of those two words is accepted by the automaton resulting from the concatenation of A and B:

Lemma 5.

$$\forall u \in \Sigma_A^*,\ v \in \Sigma_B^*,\ w = uv.\ (\delta_A^*(s_{0_A}, u) \cap F_A) \neq \emptyset\ \wedge\ (\delta_B^*(s_{0_B}, v) \cap F_B) \neq \emptyset \implies w \in \Sigma_{AB}^*\ \wedge\ (\delta_{AB}^*(s_{0_A}, w) \cap F_B) \neq \emptyset$$

```
1  lemma concat_word:
2    forall a b c: automaton, u v w: word. ∧
3      mem u (sigma_ext a.alphabet) ∧ mem v (sigma_ext b.alphabet) ∧
4      inter (delta_ext a.start (reverse u) a) a.finalStates ≠ empty ∧
5      inter (delta_ext b.start (reverse v) b) b.finalStates ≠ empty ∧
6      w = Append.(++) u v ∧ c = automaton_concat a b →
7        mem w (sigma_ext c.alphabet) ∧
8        inter (delta_ext a.start (reverse w) c) b.finalStates ≠ empty
9
10 lemma concat_alang:
11  forall a b c: automaton, u v w: word.
12   let le = automatonLang a in let lf = automatonLang b in
13   let lef = automatonLang c in
```

```
14    mem u le ∧ mem v lf ∧ w = Append.(++) u v ∧ c = automaton_concat a b →
15      mem w lef
```

Listing 4.25: Lemma 5 formalised in WhyML.

Once again, lemma `concat_alang` better translates the meaning of the first one and, therefore, is easily discharged, while being more helpful for the proof. One other interesting possibility would be to define a transitivity property for predicate `path`, similar to the one we have discussed before for `path_eps`.

The proof of the inductive step of the union of regular expressions will also require the formalization of certain concepts, mainly that of Lemma 4 (recalled below), which states that a word accepted by the automaton resulting from the union of two others is either accepted by one or the other.

$$\forall w \in \Sigma^*.\ \delta^*_{E+G}(i,w) \cap F_{E+G} \neq \emptyset \implies \left(w \in \Sigma^*_E \wedge (\delta^*_E(s_{0_E},w) \cap F_E) \neq \emptyset\right) \vee \left(w \in \Sigma^*_G \wedge (\delta^*_G(s_{0_G},w) \cap F_G) \neq \emptyset\right)$$

```
1  lemma word_union:
2    forall a b c: automaton, w: word, i: state.
3      c = automaton_union i a b ∧ mem w (sigma_ext c.alphabet) ∧
4      inter (delta_ext c.start (reverse w) c) c.finalStates ≠ empty →
5        (mem w (sigma_ext a.alphabet) ∧
6            inter (delta_ext a.start (reverse w) a) a.finalStates ≠ empty)  ∨
7        (mem w (sigma_ext b.alphabet) ∧
8            inter (delta_ext b.start (reverse w) b) b.finalStates ≠ empty)
9
10 lemma alang_union:
11   forall a b c: automaton, w: word, i: state.
12     let le = automatonLang a in let lf = automatonLang b in
13     let lef = automatonLang c in
14       c = automaton_union i a b ∧ mem w lef → mem w le  ∨  mem w lf
```

Listing 4.26: Lemma 4 formalised in WhyML.

Once again, the formalization of Lemma 4 (`word_union` in Listing 4.26) is accompanied by the definition of one other lemma that simplifies its meaning and the proof as well. Moreover, we have to keep in mind the definition of set equality, we have mentioned before, that led us to define two new lemmas to prove the equality of the concatenation. The proof of the union is no different. Therefore, similarly to Lemma 4, we have to define a new lemma stating that for any word w accepted either by an automaton A or B, the automaton resulting from the union of these two automata must also accept w:

Lemma 6.

$$\forall w.\ (w \in \Sigma^*_A \wedge (\delta^*_A(s_{0_A},w) \cap F_A) \neq \emptyset) \vee (w \in \Sigma^*_B \wedge (\delta^*_B(s_{0_B},w) \cap F_B) \neq \emptyset) \implies (\delta^*_{A+B}(s_{0_{A+B}},w) \cap (F_A \cup F_B)) \neq \emptyset$$

```
lemma union_word:
  forall a b c: automaton, w: word, i: state.
    ((mem w (sigma_ext a.alphabet)
        ∧ inter (delta_ext a.start (reverse w) a) a.finalStates ≠ empty)  ∨
     (mem w (sigma_ext b.alphabet)
        ∧ inter (delta_ext b.start (reverse w) b) b.finalStates ≠ empty)) ∧
    c = automaton_union i a b →
      mem w (sigma_ext c.alphabet) ∧
      inter (delta_ext c.start (reverse w) c) c.finalStates ≠ empty

lemma union_alang:
  forall a b c: automaton, w: word, i: state.
    let le = automatonLang a in let lf = automatonLang b in
    let lef = automatonLang c in
      (mem w le  ∨  mem w lf) ∧ c = automaton_union i a b → mem w lef
```

Listing 4.27: Lemma 6 formalised in WhyML.

The definition of the last few lemmas successfully concludes the correctness proof of function `compile` and, therefore, the mechanically checked proof of the expressive equivalence of regular expressions and finite automata. However, while lemmas `alang_concat`, `concat_alang`, `alang_union`, and `union_alang` are easily discharged, given their predecessors, the proof of lemmas `word_concat`, `concat_word`, `word_union`, and `union_word` remains an open issue. Even though these might seem natural, given the definition of concatenation and union, they have proved to be rather complex. One of the possibilities to consider, particularly in the case of the concatenation, is the use of the result of transitivity for predicate `path`, which, as we have mentioned before, we hope to formalise and prove in the future.

4.3 Discussion and Related Work

The theoretical and practical significance of formal languages and automata theory has driven many to formalise and develop mechanically-checked proofs for various of their concepts, representations, and algorithms related to the Kleene algebra.

In 1997, Filliâtre [35] produced one of the earliest works on formalizing and verifying the Kleene theorem, proving the expressive equivalence of regular expressions and finite automata, within the Coq proof assistant, and extracting a functional program that translates a regular expression into a finite automaton. Kaiser's [52] recent approach to the formalization of the Kleene theorem, which uses Coq as well, does not allow automata with ε-transitions, working on top of an alternative to the classic algorithm using deterministic finite automata instead of nondeterministic finite automata, in order to avoid the issues and complexity introduced by concepts such as the ε-closure. More recently, and following on this work, Doczkal et al. [33, 34], propose a formalization of

regular language representations in Coq's constructive type theory, including the equivalence between different automaton representations and a constructive approach to the Myhill-Nerod theorem.

Other works on the formalization of automata theory include Constable et al.'s [24] approach to constructively formalise the Myhill-Nerod theorem on the minimization of finite automata in NuPRL, and, more recently, Paulson's [72] formalised automata theory in Isabelle/HOL, including the Myhill-Nerode theorem and Brzozowski's minimization algorithm [12], based on hereditarily finite sets.

There are also substantial developments in the mechanization of decision procedures for the equivalence of regular expressions. Coquand and Siles' [25] approach was to mechanically check a decision procedure for equivalence based on Brzozowski's algebraic method [13]. Braibant and Pous [11] use Coq to verify a reflexive decision procedure for equalities in Kleene algebras based on finite automata, and prove it sound and complete. Moreira et al. [66, 67] developed a mechanically verified decision procedure based on the equivalence of the partial derivatives of regular expressions, following on Pereira and Moreira's [73] formalization of Kleene algebras with tests, Almeida et al.'s [3] correctness proof of a partial derivative automata construction from regular expressions using the Coq proof assistant, and their work on the mechanization of Kleene algebra within Coq [65].

Although the work presented in this chapter is not the first to mechanise a correction proof for the conversion of regular expressions into finite automata, our approach is original. Compared to Filliâtre's take [35], which, despite also being based on the conversion method proposed by Hopcroft et al. [46], skips on the formalization of concepts such as the ε-closure (`ECLOSE`) or the extended transition function (δ^*), by defining the acceptance of words and, therefore, the language by means of a predicate path, our approach is comparable to the theoretical work not only for the actual translation, but also every definition that precedes it, which makes it more complex but more complete as well. Moreover, as of the moment of this writing, ours is the first work to propose a mechanised proof for the equivalence of regular expressions and finite automata, or any formalization of formal language theory, within the Why3 framework.

CHAPTER 5

Conclusion

With this dissertation, we prove that regular expressions and finite automata are equivalent descriptions of formal languages, both mathematically and mechanically. We show that the two are equally expressive by proving that, after converting a regular expression into a nondeterministic fine automaton, the language of the two representations is the same — the conversion is language-preserving.

The results we prove have been the subject of much work and studied in seminal publications since Kleene first introduced regular expressions as a specification language for finite automata [54], leading to the equivalence of regular expressions and finite automata in a work to be known as Kleene's theorem. However, contributions on the theory of formal languages and automata across the years have been quite informal, leaving space for more rigorous mathematical definitions of involved algorithms and proofs, which can be especially helpful and rewarding when the end goal is to implement mechanically certified programs based on these concepts. Furthermore, these existing works prove the equivalence between regular expressions and finite automata on the possibility to generate one from the other. Yet, if we want to validate a procedure that performs such conversion, it is not enough to "show" that it is possible. We need a more fine and accurate method for proving their equivalence and that is to show that some properties remain unchanged. In particular, being descriptions of regular languages, the proof of equivalence comes down to a proof of correction of the conversion algorithm showing that the language of a regular expression is equal to that of the resulting automaton.

This led us to redefine some of the textual concepts that are needed for the definition of automaton language in a more formal manner, making them simpler to reason about in a context of proof. Additionally, we present an adapted version of the conversion algorithm, in a more formal mathematical interpretation, and finally define the proof for the equality of the languages in a *pen-and-paper* style, using the new redefinitions.

The mechanization of algorithms and proofs is a delicate process. An incomplete or

faulty specification is enough to weaken the validity of the procedure. To reach a formalization of the algorithm and consequent mechanization of the proof, this was a necessary step. The redesign of many of the concepts in a more formal manner allowed us to implement them in a way that is faithful to the mathematical development. At the same time, detailing a formal proof of equivalence on the basis of language equality, made it easier to understand the sequence of proof and underlying properties necessary to the conclusion of the correctness. In particular, through the definition of lemmas, as well as possible pre and postconditions. The Why3 mechanization, thus, follows the pen-and-paper proof closely. The choice to implement every introduced concept preceding the formal proof, instead of just using what would be an expected specification, was time-consuming, but at the same time allowed us to build a more confident specification. The process of implementing and specifying becomes iterative and, in some sense, gives us feedback on our work. If the theorem provers are not able to discharge the verification conditions, something is wrong in our procedure: either the implementation has a bug, or the specification is faulty or insufficient. In the end, this proves to be an advantage, resulting in a more complete program and greater confidence in the validity of our mechanized proof.

The present work shows that the Why3 framework is a program validation environment capable of being used in the development of realistic verified software. Particularly in our case, allowing for a mathematically rewarding mechanization of results on formal languages and automata theory, with reasonable effort.

However, a work of this dimension is not without its challenges or faults along the way. Although Why3 presented to be very powerful in regards to specification and, especially, very versatile in terms of the theorem provers we can use, their feedback (for example offering counterexamples), and the immensity of tactics that can be applied in the context of proof, we have noticed a bit of a nondeterministic behaviour. It happened that on some occasions we would not get the same result twice, for example, having two equal programs where Why3 would be able to discharge some contracts in one, but not the other. This, of course, can be influenced by the processing power of the machine in use, how much memory was available, etc. Furthermore, there were instances where obvious properties had to be further asserted in order to be discharged and induction could be cumbersome.

5.1 Future Work

There are several, more direct and imminent, developments to complement the works presented in this dissertation. In particular, as we have made clear before, we would like to narrow our trust basis. Firstly, by closing the correctness of `ECLOSE`, proving its termination and discharging the remaining postconditions. Related to the latter, as discussed in Sec. 4.2, we are also looking into proving certain properties of predicates `path` and `path_eps`, such as transitivity. Lastly, and once again taking advantage of the previous results, we would like to prove the lemmas `word_concat`, `concat_word`, `word_union` and `union_word` (Listings 4.23, 4.25, 4.26 and 4.27, respectively). Another

interesting possibility would be to take advantage of Why3's capability of extracting correct-by-construction code, to extract certified OCaml code of the current development, obtaining a verified program for translating regular expressions into finite automaton. While we could have already performed this last step, we felt it would not pose a great contribution to what we have already developed, since it would only be natural and easy to use if we implemented a parser transforming the user's input into our structure for regular expressions.

The results from this work can also be followed by the mechanization of concepts such as the minimization of finite automata or the equivalence between nondeterministic finite automata and deterministic finite automata. However, we believe that above these, the most interesting next step would be to produce the converse proof for the conversion from finite automata to regular expressions. In this case, the proof would be based on the same principle as the one developed herein: to prove that the translation preserves the language. Nonetheless, requiring a great deal of mathematical redefinitions and formalizations, akin to the work we present in this document.

Regardless, the contributions of this dissertation open the possibility for developments beyond regular languages. Particularly, we intend on applying the insights and learnings from this work to the development of verified versions of the algorithms and tool for the bi-directional translation between Mungo typestates and deterministic object automata, we have introduced in [85]. For an initial stage of development, this means defining concepts similar to those we have introduced in Chapter 3, such as the language of deterministic object automata or the acceptance of *words*. Leading to the development of a proof of correctness of the conversion algorithms where, here too, the language must be preserved. Our interest in developing a proof of correction in both directions of the conversion is what makes it important to develop a proof of correction for the conversion from finite automata to regular expressions, as well.

Finally, we intend on converting the existing OCaml code in WhyML and complement it, by formalizing the necessary concepts from the previous phase, in order to develop a mechanized proof of the equivalence between Mungo typestates and DOA. Resulting in verified procedures that can be, later on, adapted to build a certified version of the tool we have presented. Allowing programmers to use it with confidence in the results it produces.

Bibliography

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich. "Deductive Software Verification–The KeY Book." In: *Lecture Notes in Computer Science* 10001 (2016).

[2] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. "Typestate-oriented programming." In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 2009, pp. 1015–1022.

[3] M. Almeida, N. Moreira, and R. Reis. "Antimirov and Mosses's rewrite system revisited." In: *International Journal of Foundations of Computer Science* 20.04 (2009), pp. 669–684.

[4] D. Ancona, N. Publishers, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. Gay, N. Gesbert, E. Giachino, et al. *Behavioral Types in Programming Languages*. Foundations and trends in programming languages. Now Publishers, 2016. ISBN: 9781680831351. URL: https://books.google.pt/books?id=EhfYvQEACAAJ.

[5] *ANR Project VOCaL*. URL: https://vocal.lri.fr.

[6] T. Ball. "The concept of dynamic analysis." In: *Software Engineering—ESEC/FSE'99*. Springer. 1999, pp. 216–234.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. "Cvc4." In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 171–177.

[8] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[9] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. "Bounded model checking." In: *Advances in computers* 58.11 (2003), pp. 117–148.

[10] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. "The Alt-Ergo automated theorem prover." In: *URL: http://alt-ergo. lri. fr* (2008).

[11] T. Braibant and D. Pous. "An efficient Coq tactic for deciding Kleene algebras." In: *International Conference on Interactive Theorem Proving*. Springer. 2010, pp. 163–178.

[12] J. A. Brzozowski. "Canonical regular expressions and minimal state graphs for definite events." In: *Mathematical theory of Automata* 12.6 (1962), pp. 529–561.

[13] J. A. Brzozowski. "Derivatives of regular expressions." In: *Journal of the ACM (JACM)* 11.4 (1964), pp. 481–494.

[14] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. "Symbolic model checking: 1020 states and beyond." In: *Information and computation* 98.2 (1992), pp. 142–170.

[15] C. Calcagno and D. Distefano. "Infer: An automatic program verifier for memory safety of C programs." In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 459–465.

[16] L. Cardelli. "Type systems." In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 263–264.

[17] *CertiKOS*. URL: http://flint.cs.yale.edu/certikos.

[18] A. Charguéraud, J.-C. Filliâtre, M. Pereira, and F. Pottier. *VOCAL – A Verified OCAml Library*. ML Family Workshop 2017. Sept. 2017. URL: https://hal.inria.fr/hal-01561094.

[19] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind. "Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification." In: *Proc. ACM Program. Lang.* 1.ICFP (), 24:1–24:30. ISSN: 2475-1421. DOI: 10.1145/3110268.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263.

[21] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. "Model Checking and the State Explosion Problem." In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by B. Meyer and M. Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. DOI: 10.1007/978-3-642-35746-6_1.

[22] M. Collins. "Formal methods." In: *Dependable Embedded Systems* (1998).

[23] *CompCert*. URL: http://compcert.inria.fr.

[24] R. L. Constable, P. B. Jackson, P. Naumov, and J. C. Uribe. "Constructively formalizing automata theory." In: *Proof, language, and interaction*. 2000, pp. 213–238.

[25] T. Coquand and V. Siles. "A decision procedure for regular expression equivalence in type theory." In: *International Conference on Certified Programs and Proofs*. Springer. 2011, pp. 119–134.

[26] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: Jan. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[27] P. Cousot and R. Cousot. "Systematic Design of Program Analysis Frameworks." In: Jan. 1979, pp. 269–282. DOI: 10.1145/567752.567778.

[28] O. Dardha, S. J. Gay, D. Kouzapas, R. Perera, A. L. Voinea, and F. Weber. "Mungo and StMungo: Tools for Typechecking Protocols in Java." In: *Behavioural Types: from Theory to Tools* (2017), pp. 309–328.

[29] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[30] *DeepSpec*. URL: https://deepspec.org.

[31] E. W. Dijkstra. "The humble programmer." In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[32] E. W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs." In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[33] C. Doczkal, J.-O. Kaiser, and G. Smolka. "A Constructive Theory of Regular Languages in Coq." In: *Certified Programs and Proofs*. Ed. by G. Gonthier and M. Norrish. Cham: Springer International Publishing, 2013, pp. 82–97.

[34] C. Doczkal and G. Smolka. "Regular language representations in the constructive type theory of Coq." In: *Journal of Automated Reasoning* 61.1-4 (2018), pp. 521–553.

[35] J.-C. Filliâtre. "Finite Automata Theory in Coq: A constructive proof of Kleene's theorem." In: *Ecole Normale Supérieure de Lyon Research Report* (1997).

[36] J.-C. Filliâtre. "Deductive software verification." In: *International Journal on Software Tools for Technology Transfer* 13.5 (2011), pp. 397–403. DOI: 10.1007/s10009-011-0211-0.

[37] J.-C. Filliâtre, L. Gondelman, and A. Paskevich. "The spirit of ghost code." In: *Formal Methods in System Design* 48.3 (2016), pp. 152–174.

[38] J.-C. Filliâtre and A. Paskevich. "Why3 — Where Programs Meet Provers." In: *Programming Languages and Systems*. Ed. by M. Felleisen and P. Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. ISBN: 978-3-642-37036-6.

[39] K. Fisher, J. Launchbury, and R. Richards. "The HACMS program: using formal methods to eliminate exploitable bugs." In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017), p. 20150401.

[40] R. W. Floyd. "Assigning meanings to programs." In: *Program Verification*. Springer, 1993, pp. 65–81.

[41] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. "CertiKOS: A Certified Kernel for Secure Cloud Computing." In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys '11. ACM, 2011, 3:1–3:5. ISBN: 978-1-4503-1179-3. DOI: 10.1145/2103799.2103803.

[42] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Nov. 2016, pp. 653–669. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[43] J. Y. Halpern, R. Harper, N. Immerman, P. G. Kolaitis, M. Y. Vardi, and V. Vianu. "On the unusual effectiveness of logic in computer science." In: *Bulletin of Symbolic Logic* 7.2 (2001), pp. 213–236.

[44] C. A. R. Hoare. "Proof of a program: FIND." In: *Communications of the ACM* 14.1 (1971), pp. 39–45.

[45] C. A. R. Hoare. "An axiomatic basis for computer programming." In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[46] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321462254.

[47] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[48] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, et al. "Foundations of session types and behavioural contracts." In: *ACM Computing Surveys (CSUR)* 49.1 (2016), pp. 1–36.

[49] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. "VeriFast: A powerful, sound, predictable, fast verifier for C and Java." In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 41–55.

[50] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681.

[51] C. B. Jones. *Turing and Software Verification*. Computing Science, Newcastle University, 2014.

[52] J.-O. Kaiser. "Constructive formalization of regular languages." Bachelor's thesis. Saarland University, 2012.

[53] *Kami*. URL: http://plv.csail.mit.edu/kami/.

[54] S. C. Kleene. "Representation of Events in Nerve Nets and Finite Automata." In: *Automata Studies. (AM-34), Volume 34*. Princeton: Princeton University Press, 31 Dec. 1956, pp. 3 –42. ISBN: 9781400882618. DOI: https://doi.org/10.1515/9781400882618-002.

[55] 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. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

[56] D. Kozen. *Automata and Computability*. Undergraduate Texts in Computer Science. Springer New York, 2007. ISBN: 9780387949079.

[57] *L4hq*. URL: http://l4hq.org.

[58] K. R. M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness." In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by E. M. Clarke and A. Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[59] X. Leroy. "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant." In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. ACM, 2006, pp. 42–54. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111042.

[60] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall Software Series. Prentice-Hall, 1998. ISBN: 9780132624787.

[61] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984.

[62] *MetaCoq project page*. URL: https://metacoq.github.io/metacoq/.

[63] R. Milner. "A theory of type polymorphism in programming." In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[64] J. F. Monin and M. G. Hinchey. *Understanding Formal Methods*. Berlin, Heidelberg: Springer-Verlag, 2001. ISBN: 1852332476.

[65] N. Moreira, D. Pereira, and S. M. de Sousa. *On the mechanization of Kleene algebra in Coq*. Tech. rep. Technical Report DCC-2009-03, DCC-FC&LIACC, Universidade do Porto, 2009.

[66] N. Moreira, D. Pereira, and S. M. de Sousa. "Deciding regular expressions (in-) equivalence in Coq." In: *International Conference on Relational and Algebraic Methods in Computer Science*. Springer. 2012, pp. 98–113.

[67] N. Moreira, D. Pereira, and S. M. de Sousa. "Deciding kleene algebra terms equivalence in Coq." In: *Journal of Logical and Algebraic Methods in Programming* 84.3 (2015), pp. 377–401.

[68] F. L. Morris and C. B. Jones. "An early program proof by Alan Turing." In: *IEEE Annals of the History of Computing* 2 (1984), pp. 139–143.

[69] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. "Use of formal methods at Amazon Web Services." In: *See http://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf* (2014).

[70] O. Nierstrasz. "Regular types for active objects." In: *ACM sigplan Notices* 28.10 (1993), pp. 1–15.

[71] M. J. Parreira Pereira. "Tools and Techniques for the Verification of Modular Stateful Code." Theses. Université Paris-Saclay, Dec. 2018. URL: https://tel.archives-ouvertes.fr/tel-01980343.

[72] L. C. Paulson. "A formalisation of finite automata using hereditarily finite sets." In: *International Conference on Automated Deduction*. Springer. 2015, pp. 231–245.

[73] D. Pereira and N. Moreira. "KAT and PHL in Coq." In: *Computer Science and Information Systems* 5.2 (2008), pp. 137–160.

[74] B. C. Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[75] 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. Sydney, Australia: NICTA, July 2014.

[76] J. C. Reynolds. "Separation logic: A logic for shared mutable data structures." In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74.

[77] C. Sernadas. *Introdução à teoria da computação. (Portuguese) [Introduction to the theory of computation]*. Lisbon, Portugal: Presença, 1993.

[78] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012. ISBN: 9781285401065.

[79] M. Sipser. *Introduction to the Theory of Computation, Lecture 7*. 2013. URL: https://courses.engr.illinois.edu/cs373/sp2013/Lectures/lec07.pdf.

[80] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter. "The MetaCoq Project." working paper or preprint. June 2019. URL: https://hal.inria.fr/hal-02167423.

[81] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq." In: *Proceedings of the ACM on Programming Languages* (Jan. 2020). DOI: 10.1145/3371076. URL: https://hal.archives-ouvertes.fr/hal-02380196.

[82] R. E. Strom and S. Yemini. "Typestate: A programming language concept for enhancing software reliability." In: *IEEE Transactions on Software Engineering* 1 (1986), pp. 157–171.

[83] *The HACMS project*. URL: http://ts.data61.csiro.au/projects/TS/SMACCM/.

[84] *The seL4 Microkernel*. URL: https://sel4.systems.

[85] A. Trindade, J. Mota, and A. Ravara. "Typestates to Automata and back: a tool." In: *13th Interaction and Concurrency Experience (ICE 2020)*. Ed. by J. Lange, A. Mavridou, L. Safina, and A. Scalas. Vol. 324. EPTCS. 2020, pp. 25–42. DOI: 10.4204/EPTCS.324.4. URL: https://doi.org/10.4204/EPTCS.324.4.

[86] A. M. Turing. "Checking a large routine." In: *Report of a Conference on High Speed Automatic Calculing Machines* (1949), pp. 67–69.

[87] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave. "Modular Deductive Verification of Multiprocessor Hardware Designs." In: *Proceedings of Computer Aided Verification CAV'15*. Ed. by D. Kroening and C. S. Păsăreanu. Springer International Publishing, 2015, pp. 109–127. ISBN: 978-3-319-21668-3.

[88] M. Wenzel, L. C. Paulson, and T. Nipkow. "The isabelle framework." In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2008, pp. 33–38.

[89] *Why3 project page*. URL: http://why3.lri.fr.

[90] B. A. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward, and D. Marsh. "Industrial perspective on static analysis." In: *Software Engineering Journal* 10.2 (1995), pp. 69–75.

[91] N. Yoshida, R. Hu, R. Neykova, and N. Ng. "The Scribble protocol language." In: *International Symposium on Trustworthy Global Computing*. Springer. 2013, pp. 22–41.