David Miguel Ramalho Pereira

Towards Certified Program Logics for the Verification of Imperative Programs

Doctoral Program in Computer Science
of the Universities of Minho, Aveiro and Porto

April 2013

David Miguel Ramalho Pereira

Towards Certified Program Logics for the Verification of Imperative Programs

Thesis submitted to Faculty of Sciences of the University of Porto for the Doctor Degree in Computer Science within the Joint Doctoral Program in Computer Science of the Universities of Minho, Aveiro and Porto

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
April 2013

Para os meus pais David e Maria Clara e para a Ana...

Acknowledgments

I would like to start by thanking my supervisors Nelma Moreira and Simão Melo de Sousa for their guidance, for the fruitful discussions we had together, and mostrly for their friendship and encouragement along the past years.

I am particularly indebted to José Carlos Bacelar Almeida who was always available to discuss with me several technical issues related to Coq. I also thank Ricardo Almeida for our discussions about Kleene algebra with tests. To Rogério Reis, my sincere thanks for being such a good teacher.

Obviously, the work environment was a major help, and I wish to thank Alexandre, André, Andreia, Bruno, Besik, Cláudio, Jorge, Marco, and Vitor for the fun moments at the office of the Computer Science department.

Towards the end of the PhD, a group of people was also of great importance, as they received me in their laboratory, in order to start new research in the area of real-time systems. In particular I would like to thank Luís Miguel Pinho, Luís Nogueira, and also to my new colleges André Pedro, Cláudio Maia, and José Fonseca.

Regarding financial support, I am grateful to the Fundação para a Ciência e Tecnologia and the MAP-i doctoral programme for the research grant SFRH/BD/33233/2007 which allowed me to conduct the research activities over the past years. I am also thankful to the research projects CANTE (PTDC/EIA-CCO/101904/2008), FAVAS (PTDC/EIA-CCO/105034/2008), and RESCUE (PTDC/EIA/65862/2006).

Finally, I would like to thank my family and my close friends. Their flawless support and the fun moments we had together were essential to help me in this demanding adventure. My parents always believed and supported me by any means possible. My little nephews, Carolina and Rodrigo, were always there to make me smile and laugh, even in those darkest days where some unfinished proof turned me into an intolerable person. I could not finish without thanking Ana, my girlfrined, who was always by my side and that bared with me until the very end. Her unconditional belief, support and motivation literally ensured that I this dissertation was indeed possible to be finished.

Abstract

Modern proof assistants are mature tools with which several important mathematical problems were proved correct, and which are also being used as a support for the development of program logics libraries that can be used to certify software developments.

Using the Coq proof assistant we have formalised a library for regular languages, which contains a sound and complete procedure for deciding the equivalence of regular expressions. We also formalised a library for the language theoretical model of Kleene algebra with tests, the algebraic system that considers regular expressions extended with Boolean tests and that is particularly suited to the verification of imperative programs.

Also using the Coq proof assistant, we have developed a library for reasoning about shared-variable parallel programs in the context of Rely-Guarantee reasoning. This library includes a sound proof system, and can be used to prove the partial correctness of simple parallel programs.

The goal of the work presented in this dissertation is to contribute to the set of available libraries that help performing program verification tasks relying in the trusted base provided by the safety of the Coq proof system.

Resumo

Os assistentes de demonstração modernos são ferramentas complexas e nas quais foram formalizados e demonstrados correctos vários problemas matemáticos. São também usados como ferramentas de suporte nas quais é possível codificar lógicas de programas e usar essas mesmas codificações em tarefas de certificação de programas.

Utilizando o assistente de demonstração Coq, formalizamos uma biblioteca de linguagens regulares que contém um procedimento de decisão integro e completo para o problema da equivalência de expressões regulares. Formalizamos também uma biblioteca do modelo de linguagens da álgebra de Kleene com testes, que se trata de um sistema algébrico que considera como termos expressões regulares estendidas com testes e é particularmente adequado para a verificação de programas imperativos.

Utilizando também o assistente de demonstração Coq, desenvolvemos uma biblioteca que contém uma formalização que permite raciocinar sobre a correcção parcial de programas paralelos dotados de arquitecturas com variáveis partilhadas. Esta formalização enquadra-se no contexto do Rely-Guarantee. A biblioteca desenvolvida contém um sistema de inferência que pode ser usado para a demonstração da correcção parcial de programas paralelos simples.

O objectivo das formalizações que descrevemos ao longo desta dissertação é o de contribuir para o conjunto de bibliotecas formais disponíveis que permitem a verificação de programas e cujo nível de confiança é reforçado dado o uso das mesmas num ambiente seguro, providenciado pelo assistente de demonstração Coq.

Contents

Acknowledgments **vii**

Abstract **ix**

Resumo **xi**

List of Tables **xvii**

List of Figures **xix**

1 Introduction **1**

1.1 Contributions 3

1.2 Structure of the Thesis 5

2 Preliminaries **7**

2.1 The Coq Proof Assistant 7

2.1.1 The Calculus of Inductive Constructions 7

2.1.2 Inductive Definitions and Programming in Coq 9

2.1.3 Proof Construction 11

2.1.4 Well-founded Recursion 14

2.1.5 Other Features of Coq 17

2.1.6 Sets in Coq 17

2.2 Hoare Logic 20

2.3 Conclusions 25

3 Equivalence of Regular Expressions **27**

3.1 Elements of Language Theory . . . 27

3.1.1 Alphabets, Words, and Languages . . . 28

3.1.2 Finite Automata . . . 31

3.1.3 Regular Expressions . . . 33

3.1.4 Kleene Algebra . . . 37

3.2 Derivatives of Regular Expressions . . . 38

3.3 A Procedure for Regular Expressions Equivalence . . . 47

3.3.1 The Procedure EQUIVP . . . 47

3.3.2 Implementation . . . 50

3.3.3 Correctness and Completeness . . . 56

3.3.4 Tactics and Automation . . . 60

3.3.5 Performance . . . 68

3.4 Related Work . . . 70

3.5 Conclusions . . . 71

4 Equivalence of KAT Terms **73**

4.1 Kleene Algebra with Tests . . . 73

4.2 The Language Model of KAT . . . 74

4.3 Partial Derivatives of KAT Terms . . . 83

4.4 A Procedure for KAT Terms Equivalence . . . 89

4.4.1 The Procedure EQUIVKAT . . . 90

4.4.2 Implementation, Correctness and Completeness . . . 92

4.5 Application to Program Verification . . . 99

4.6 Related Work . . . 105

4.7 Conclusions . . . 105

5 Mechanised Rely-Guarantee in COQ **107**

5.1 Rely-Guarantee Reasoning . . . 108

5.2 The IMPp Programming Language . . . 110

5.3 Operational Semantics of IMPp 112
5.4 Reductions under Interference 115
5.5 A Proof System for Rely-Guarantee 118
5.6 Soundness of HL-RG 123
5.7 Examples and Discussion 131
5.8 Related Work 132
5.9 Conclusions 133

6 Conclusions and Future Work **135**

References **137**

List of Tables

3.1 Performance results of the tactic dec_re. 69

3.2 Comparison of the performances. 71

List of Figures

3.1 The transition diagram of the DFA D. 31

3.2 A NFA accepting sequences of 0's and 1's that end with a 1. 32

5.1 Rely and guarantee *vs.* preconditions and postconditions. 109

Chapter 1

Introduction

The increase of complexity and criticality of computer programs over the past decades has motivated a great deal of interest in formal verification. Using formal systems, software developers gain access to a framework in which they can construct rigorous specifications of the properties that a program must satisfy. Usually, these formal systems are associated with a proof system that allows users to prove the correctness of the system with respect to its specification. Therefore, the number of design and implementation errors in programs decreases drastically and the trust of the users on those programs increases considerably.

The roots of formal verification go back to the 19th century with the work of Frege, who introduced first-order logic formally and the notion of *formal proof*. A formal proof is a sequence of derivation steps, such that each of those steps can be checked to be well-formed with respect to the rules of the underlying proof system. Formal proofs are the key element of formal verification since they represent evidence that can be effectively checked for validity. This validity naturally implies the correctness of a property with respect to the specification of the system under consideration.

The process of constructing a formal proof in some formal system usually turns out to be non-trivial. The first incompleteness theorem of Gödel shows that there is no formal system that allows to deduce all the true statements as soon as it includes a certain fragment of the arithmetic of natural numbers. Later, Church and Turing proved that in particular first-order logic is undecidable in the sense that no procedure can be constructed that is able to prove the validity of all true first-order formulas. Nevertheless, this inherent problem of first-order logic impelled researchers to find fragments that are expressive enough and also decidable. This effort resulted in the advent of *automated theorem proving*, whose major representatives are the *satisfiability of Boolean formulas* (SAT) and the *satisfiability modulo theory* (SMT).

Parallel to the development of automated theorem proving, considerable evolutions were also registered in the development of *proof assistants*, also known as *interactive theorem provers*. A proof assistant is a piece of software that allows users to encode mathematics

on a computer, and assists them in checking that each of the proof's derivation steps are indeed correct with respect to the underlying formal system. The first project that addressed a primitive notion of proof assistant was Automath, led by De Brujin, and whose goal was to provide a mechanical verification system for mathematics. The result of this effort was a language where mathematical statements and proofs could be written rigorously, and also a set of implementations of proof-checkers responsible for verifying the correctness of the derivations contained in the proofs. The proofs are usually referred to as *proof scripts* and they usually do not correspond to the exact same language of the formal system underlying the proof assistant: a proof script is made of a set of definitions, statements of theorems, and sequences of instructions that convince the prover that those statements are indeed true.

De Brujin made also another important contribution to the development of proof assistants, by introducing the notion of *proof-as-objects* into this field, and which is tantamount at the notion of the *Curry-Howard isomorphism* that states that a proof P of a theorem ϕ, in a constructive logic, can be regarded as a λ-term P whose type is ϕ in a type system. Hence, proof-checking can be reduced to type-checking. As more powerful type theories were conceived, such as the one of Martin-Löf that lays type theory as a foundation of mathematics, more complex and powerful proof assistants were developed. As a consequence, type checking tools also became more complex, but they remain tractable problems and can be implemented and checked by humans, thus imposing a high-level of trust in the core of proof assistants. These small type-checkers that support the proof construction are usually developed under the so-called *De Brujin principle*: every proof can be written completely using just the set of rules of a small kernel, which corresponds exactly to the set of inference rules of the underlying formal system. Most modern proof assistants employ the De Brujin principle in their design.

Yet another motivating feature of some proof assistants is the *proof-as-programs* criteria: in constructive logic, a statement of the form $\forall x, \exists y,\ R(x, y)$ means that there is a total function f such that $\forall x,\ R(x, f(x))$. Hence, the function f is built in the proof and it can be extracted as a standard functional program. This process is known as *program extraction*. Rephrasing in the context of type theory, given a *specification* $\forall x : A, \exists y : B.R(x, y)$, its realisation will produce a functional program $f : A \to B$. This has important implications in the construction of programs: the proof checking system can be encoded in the prover and then extracted and serve as a correct proof checking kernel for future versions of the prover; a core component of a bigger software can be developed in this way, ensuring the correct behaviour of the programs, despite errors that may appear in the development of the other components of the program.

In the actual state-of-the-art, the role of proof assistants cannot be overlooked, both int the certification of mathematics and in the formal verification and development of computer systems. The mechanized proofs of the four-color theorem by Gonthier and Werner [40], the

Feit-Thomson theorem [41] by Gonthier *et. al.*, the certified compiler CompCert by Xavier Leroy [67], the certification of microkernel seL4 (Secure Embedded L4) [56] in HOL, and the certification of automated theorem prover ALT-ERGO in COQ [71] are some the main achievements of the usage of proof assistants.

1.1 Contributions

We have already seen that proof assistants are powerful and reliable tools with which one can specify mathematical theories and programs, and also verify their correctness. Usually, these tasks end up being tedious and, very often, difficult. However, the advantage of using proof assistants is that we define and certify concepts or libraries of interest once and for all. Moreover, we can incorporate these certified concepts and libraries into other developments, by fully trusting on them. With adequate support, we can even extract correct-by-construction software and then integrate this trusted code into larger software developments.

The subject of this thesis is that of using proof assistants to encode program logics which can be used to conduct program verification. We address both sequential programs, and parallel ones. For sequential programs we formalize regular expressions and one of its extensions with Boolean values, where deductive reasoning is replaced by equational reasoning. Moreover, both theories are able to capture choice and iteration and are decidable, which allows for some formulae to be proved automatically. For handling parallel programs, we formalize an extension of Hoare logic that is able to express properties of parallel execution in a compositional way.

Summing up, this thesis aims at contributing with the following three formalizations:

A decision procedure for regular expressions equivalence. Regular expressions are one of the most common and important concepts that can be found, in some way or another, in any field of Computer Science. Regular expressions are mostly recognized as a tool for matching patterns in languages, hence they are fundamental in the world-wide-web, compilers, text editors, etc. They were first introduced in the seminal work of Kleene [55] as a specification language for automata. Their compact representation, flexibility, and expressiveness lead to applications outside language processing, and have been applied with success, for instance, as runtime monitors for programs [92, 91]. Regular expressions can in fact be seen as a program logic that allows to express non-deterministic choice, sequence, and finite iteration of programs. Moreover, they were extended to address imperative programs [54, 59] and real-time systems [10, 89]. Hence, it is important to provide formalized theories about regular expressions, so that we can use them in program verification. A particular and interesting property of regular expressions is that their equivalence and containment are decidable properties. This means that we can also formalize the decision procedure and

extract it as a correct functional program. This is the goal of our first contribution, where we devise and prove correct a decision procedure for regular expression equivalence, following along the lines of *partial derivatives* of Antimirov and Mosses [9, 8]. Since regular expressions coincide with relational algebra, we also developed a tactic to decide relational equivalence.

A decision procedure for the equivalence of terms of Kleene algebra with tests. Regular expressions can be enriched with Boolean tests and, in this way, provide expressivity to capture the conditional and while-loop constructions of imperative languages. These expressions are terms of *Kleene algebra with tests*, introduced by Dexter Kozen [59]. We have formalized the language model of Kleene algebra with tests, and as with regular expressions, we follow the partial derivative approach. Moreover, we have implemented a decision procedure for the equivalence of terms. Some examples of the application of Kleene algebra with tests to program verification are presented.

A sound proof system for rely/guarantee. Our final contribution focus on the formal aspects and verification of parallel programs. We have chosen to specify and prove sound a proof system to verify the correctness of programs written in a simple imperative parallel programming language. The target programs are shared memory parallel programs. The semantics of the language follows the principle of Rely-Guarantee introduced by Cliff Jones [53], and latter described in [27, 78, 102] by a small-step reduction semantics that considers a fine-grained notion of interference caused by the environment. The source of the interference is modeled by the rely relation. The effect that the computation of the individual programs imposes in the environment is constrained by a guarantee relation. The proof system is an extension of Hoare logic, whose triples are enriched with the rely and guarantee relations.

Publications List

- "KAT and PHL in Coq". David Pereira and Nelma Moreira. *Computer Science and Information Systems*, 05(02), December 2008.

- "Partial derivative automata formalized in Coq". José Bacelar Almeida, Nelma Moreira, David Pereira, and Simão Melo de Sousa. In Michael Domaratzki and Kai Salomaa, editors. Proceedings of the 15th *International Conference on Implementation and Application of Automata (CIAA 2010)*, Winnipeg, MA, Canada, August, 2010. Volume 6482 of *Lecture Notes in Computer Science*, pages 59-68, Springer-Verlag, 2011.

- "Deciding regular expressions (in-)equivalence in Coq". Nelma Moreira, David Pereira, and Simão Melo de Sousa. In T. G. Griffin and W. Kahl, editors. Proceedings of the 13th *International Conference on Relational and Algebraic Methods in Computer Science (RAMiCS 13)*, Cambridge, United Kingdom, September, 2012. Volume 7560 of *Lecture Notes in Computer Science*, pages 98-133, Springer-Verlag, 2012.

- "PDCoq : Deciding regular expressions and KAT terms (in)equivalence in Coq through partial derivatives". Nelma Moreira, David Pereira, and Simão Melo de Sousa.
http://www.liacc.up.pt/~kat/pdcoq/

- "RGCoq : Mechanization of rely-guarantee in Coq". Nelma Moreira, David Pereira, and Simão Melo de Sousa.
http://www.liacc.up.pt/~kat/rgcoq/

1.2 Structure of the Thesis

This dissertation is organized as follows:

Chapter 2 introduces the Coq proof assistant, our tool of choice for the formalizations developed. It also describes Hoare logic, the program logic that is central to the application of these same formalizations to program verification.

Chapter 3 describes the mechanization of a decision procedure for regular expression equivalence. It also includes the formalization of a relevant fragment of regular language theory. We present experimental performance tests and compare the development with other formalizations that address the same goal, but that use different approaches.

Chapter 4 describes the extension of the development presented in Chapter 3 to Kleene algebra with tests. This extension includes a mechanization of the language theoretic model of Kleene algebra with tests, and also a decision procedure for the equivalence of terms of Kleene algebra with tests. We present some examples that show that this algebraic system can effectively be used to handle proofs about the equivalence and partial correctness of programs.

Chapter 5 describes the formalization of an extended Hoare inference system for proving the partial correctness of shared-memory parallel programs, based on the notions of rely and guarantee.

Chapter 6 reviews the contributions described in the previous three chapters, and establishes research lines that aim at solidifying our contributions in order to make them capable of addressing more complex programs.

Chapter 2

Preliminaries

In this chapter we introduce the Coq proof assistant [97], our tool of choice for the developments that are described along this dissertation. We also introduce Hoare logic [48], the well known program logic that has become the standard logic for addressing the correctness of computer programs.

2.1 The Coq Proof Assistant

In this section we provide the reader with a brief overview of the Coq proof assistant. In particular, we will look into the definition of (dependent) (co-)inductive types, to the implementation of terminating recursive functions, and to the proof construction process in Coq's environment. A detailed description of these topics and other Coq related subjects can be found in the textbooks of Bertot and Casterán [15], of Pierce *et.al.* [85], and of Chlipala [25].

2.1.1 The Calculus of Inductive Constructions

The Coq proof assistant is an implementation of Paulin-Mohring's *Calculus of Inductive Constructions* (CIC) [15], an extension of Coquand and Huet's *Calculus of Constructions* (CoC) [28] with (dependent) inductive definitions. In rigor, since version 8.0, Coq is an implementation of a weaker form of CIC, named the *predicative Calculus of Inductive Constructions* (pCIC), and whose rules are described in detail in the official Coq manual [98].

Coq is supported by a rich typed λ-calculus that features polymorphism, dependent types, very expressive (co-)inductive types, and which is built on the *Curry-Howard Isomorphism* (CHI) programs-as-proofs principle [51]. In CHI, a typing relation $t : A$ can be interpreted either as a term t of type A, or as t being a proof of the proposition A. A classical example of

the CHI is the correspondence between the implication elimination rule (or *modus ponens*)

$$\frac{A \to B \qquad A}{B},$$

and the function application rule of λ-calculus

$$\frac{f : A \to B \qquad x : A}{f(x) : B},$$

from where it is immediate to see that the second rule is "the same" as the first one if we erase the terms information. Moreover, interpreting the typing relation $x : A$ as the logical statement *"x proves A"*, and interpreting $f : A \to B$ as *"the function f transforms a proof of A into a proof of B"*, then we conclude that the application of the term x to function f yields the conclusion *"$f(x)$ proves B"*. Under this perspective of looking at logical formulae and types, CIC becomes both a functional programming language with a very expressive type system and, simultaneously, a higher-order logic where users can define specifications about the programs under development, and also build proofs that show that those programs are correct with respect to their specifications.

In the CIC, there is no distinction between terms and types. Henceforth, all types have their own type, called a *sort*. The set of sorts of CIC is the set

$$\mathcal{S} = \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i) \mid i \in \mathbb{N}\}.$$

The sorts Prop and Set ensure a strict separation between *logical types* and *informative types*: the former is the type of propositions and proofs, whereas the latter accommodates data types and functions defined over those data types. An immediate effect of the non-existing distinction between types and terms in CIC is that computations occurs both in programs and in proofs.

In CIC, terms are equipped with a built-in notion of *reduction*. A reduction is an elementary transformation defined over terms, and a computation is simply a finite sequence of reductions. The set of all reductions forms a confluent and strong normalising system, *i.e.*, all terms have a unique *normal form*. The expression

$$E, \Gamma \vdash t =_{\beta\delta\zeta\iota} t'$$

means that the terms t and t' are convertible under the set of reduction rules of the CIC, in a context Γ and in an environment E. In this case, we say that t and t' are $\beta\delta\zeta\iota$-convertible, or simply convertible. The reduction rules considered have the following roles, respectively: the reduction β, pronounced *beta reduction*, transforms a β-redex $(\lambda x : A.e_1)e_2$ into a term $e_1\{x/e_2\}$; the reduction δ, pronounced *delta reduction*, replaces an identifier with its definition; the reduction ζ, pronounced *zeta-reduction*, transforms a local definition

of the form $\mathsf{let}\, x := e_1 \,\mathsf{in}\, e_2$ into the term $e_2\{x/e_1\}$; finally, the reduction ι, pronounced *iota reduction*, is responsible for computation with recursive functions, and also for pattern matching.

A fundamental feature of Coq's underlying type system is the support for *dependent product types* $\Pi x : A.B$, which extends functional types $A \to B$ in the sense that the type of $\Pi x : A.B$ is the type of functions that map each instance of x of type A to a type of B where x may occur in it. If x does not occur in B then the dependent product corresponds to the function type $A \to B$.

2.1.2 Inductive Definitions and Programming in Coq

Inductive definitions are another key ingredient of Coq. An inductive type is introduced by a collection of *constructors*, each with its own arity. A value of an inductive type is a composition of such constructors. If T is the type under consideration, then its constructors are functions whose final type is T, or an application of T to arguments. Moreover, the constructors must satisfy *strictly positivity constraints* [83] for the sake of preserving the termination of the type checking algorithm, and for avoiding definitions that lead to logical inconsistencies [15].

One of the simplest examples is the classical definition of Peano numbers:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

The definition of `nat` is not written in pure CIC, but rather in the specification language Gallina. In fact, this definition yields four different definitions: the definition of the type `nat` in the sort `Set`, two *constructors* `O` and `S`, and an automatically generated induction principle `nat_ind` defined as follows.

```
∀ P:nat → Prop, P 0 → (∀ n:nat, P n → P (S n)) → ∀ n:nat, P n.
```

The induction principle expresses the standard way of proving properties about Peano numbers, and it enforces the fact that these numbers are built as a finite application of the two constructors `O` and `S`. By means of *pattern matching*, we can implement recursive functions by deconstructing the given term and produce new terms for each constructor. An example is the following function that implements the addition of two natural numbers:

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O ⇒ m
  | S p ⇒ S (p + m)
  end
```

```
where "n + m" := (plus n m).
```

The `where` clause, in this case, allows users to bind notations to definitions, thus making the code easier to read. The definition of `plus` is possible since it corresponds to an exhaustive pattern-matching, *i.e.*, all the constructors of `nat` are considered. Moreover, the recursive calls are performed on terms that are *structurally* smaller than the given recursive argument. This is a strong requirement of CIC which requires that all functions must be terminating. We will see ahead that non-structurally recursive functions still can be implemented in Coq via a translation into equivalent, structurally decreasing functions.

More complex inductive types can be defined, namely inductive definitions that depend on values. A classic example is the family of vectors of length $n \in \mathbb{N}$, whose elements have a type `A`:

```
Inductive vect (A : Type) : nat → Type :=
| vnil : vect A 0
| vcons : ∀ n : nat, A → vect A n → vect A (S n)
```

As an example, the code below shows how to create the terms representing the vectors $[a,b]$ and $[c]$ with lengths 2 and 1, respectively. The elements of these two vectors are the constructors of another inductively defined type `A`. In the code below, the values 0, 1, and 2 correspond to the Peano numerals `O`, `S O`, and `S (S O)`, respectively.

```
Inductive A:Type := a | b | c.

Definition v1 : vect A 2 := vcons A 1 a (vcons A 0 b (vnil A)).
Definition v2 : vect A 1 := vcons A 0 c (vnil A).
```

A natural definition over values of type `vect` is the concatenation of vectors. In Coq it goes as follows:

```
Fixpoint app(n:nat)(l1:vect A n)(n':nat)(l2:vect A n'){struct l1} :
 vect (n+n') :=
 match l1 in (vect _ m') return (vect A (m' + n')) with
  | vnil ⇒ l2
  | vcons n0 v l1' ⇒ vcons A (n0 + n') v (app n0 l1' n' l2)
 end.
```

Note that there is a difference between the pattern-matching construction `match` used in the definition of the `sum` function, and the one used in the implementation of `app`: in the latter, the returned type depends on the sizes of the vectors given as arguments. Therefore, the extended `match` construction in `app` has to bind the dependent argument m' to ensure that the final return type is a vector of size $n + n'$. The computation of `app` with arguments `v1`

and v2 yields the expected result, that is, the vector [a,b,c] of size 3 (since the value 2+1 is convertible to the value 3):

```
Coq <   Eval vm_compute in app A 2 v1 1 v2.
    = vcons 2 a (vcons 1 b (vcons 0 c (vnil A)))
    : vect (2 + 1)
```

The vm_compute command performs reductions within a virtual machine [42] which is almost as efficient as bytecode compiled OCaml code.

2.1.3 Proof Construction

The type system underlying Coq is an extended λ-calculus that does not provide built-in logical constructions besides universal quantification and the Prop sort. Logical constructions are encoded using inductive definitions and the available primitive quantification. For instance, the conjunction $A \wedge B$ is encoded by the inductive type and, defined as follows:

```
Inductive and (A B : Prop) : Prop :=
| conj : A → B → and A B
where "A ∧ B'' := (and A B).
```

The induction principle automatically generated for and is the following:

```
and_ind : ∀ A B P : Prop, (A → B → P) → A ∧ B → P
```

Disjunction is encoded in a similar way, and consists in two constructors, one for each branch of the disjunction. Negation is defined as a function that maps a proposition A into the constant False, which in turn is defined as the inductive type with no inhabitants. The constant True is encoded as an inductive type with a single constructor I. Finally, the existential quantifier $\exists\, x$:T, P(x) is defined by the following inductive definition:

```
Inductive ex (A:Type) (P : A → Prop) : Prop :=
| ex_intro : ∀ x:A, P x → ex P
```

The inductive definition ex requires us to provide a *witness* that the predicate P is satisfied by the term x, in the spirit of constructive logic, where connectives are seen as functions taking proofs as input, and producing new proofs as output.

The primitive way of building a proof in Coq is by explicitly constructing the corresponding CIC term. Thankfully, proofs can be constructed in a more convenient, interactive, and backward fashion by using a language of commands called *tactics*. Although tactics are commonly applied when the user is in the *proof mode* of Coq– activated by the Theorem command (or similar commands) – the tactics can be used also to construct programs

interactively. However, this must be done with care, since tactics usually produce undesirable large terms. Let us take a look at the example of the construction of a simple proof of the commutativity of the conjunction $A \wedge B$, where A and B are propositions. First, we need to tell Coq that we are want to enter the proof mode. For that, we use the command `Theorem`.

```
Coq < Theorem and_comm :
Coq <    forall A B:Prop,
Coq <       A /\ B -> B /\ A.
1 subgoal

  ============================
   forall A B : Prop, A /\ B -> B /\ A
```

The first part of the proof is to move the universally quantified propositions and the hypothesis `A ∧ B` into the context:

```
Coq <   intros A B H.
1 subgoal

  A : Prop
  B : Prop
  H : A /\ B
  ============================
   B /\ A
```

Next, we eliminate the hypothesis `H` to isolate the terms `A` and `B`. This is achieved by the `destruct` tactic:

```
Coq <   destruct H.
1 subgoal

  A : Prop
  B : Prop
  H : A
  H0 : B
  ============================
   B /\ A
```

Now that we know that both propositions `A` and `B` hold, we have to split the goal in order to isolate each of the components of the conjunction. This is carried out by the tactic `constructor` which applies the unique constructor `and`, yielding two new sub-goals, one to prove `A`, and another to prove `B`.

```
Coq <   constructor.
```

2 subgoals

A : Prop
B : Prop
H : A
H0 : B
==============================
B

subgoal 2 is:
A

To finish the proof it is enough to apply the tactic `assumption` that searches the hypothesis in the context and concludes that `A` and `B` are already known to be true.

```
Coq <   assumption.
```

1 subgoal

A : Prop
B : Prop
H : A
H0 : B
==============================
A

```
Coq <   assumption.
```

Proof completed.

```
Coq < Qed.
```

`Coq <` *and_comm is defined*

The command `Qed` marks the end of the proof. This command has a very important role: it checks that the term that was progressively constructed using the tactics is a well-formed inhabitant of the type of the theorem that we have allegedly proved. This allows one to develop new tactics without formal restrictions, and prevents possible bugs existing in the tactics from generating wrong proof terms, since the terms constructed are checked once again at the end of the proof. When using the command `Qed`, the proof term becomes *opaque* and cannot be unfolded or reduced. In order to have the contrary behaviour, the user must use the command `Defined`.

2.1.4 Well-founded Recursion

As pointed out earlier, all the functions defined in Coq must be provably terminating. The usual approach is to implement functions through the Fixpoint command and use one of the arguments as the structurally recursive argument. However, this is not possible to do for the implementation of all terminating functions. The usual way to tackle this problem is via an encoding of the original formulation of the function into an equivalent, structurally decreasing function. There are several techniques available to address the development of non-structurally decreasing functions in Coq, and these techniques are documented in detail in [15]. Here we will consider the technique for translating a general recursive function into an equivalent, *well-founded recursive function.*

A given binary relation $\mathcal{R}$ defined over a set S is said to be *well-founded* if for all elements $x \in S$, there exists no strictly infinite descendent sequence $(x, x_0, x_1, x_2, \dots)$ of elements of S such that $(x_{i+1}, x_i) \in \mathcal{R}$. Well-founded relations are available in Coq through the definition of the inductive predicate `Acc` and the predicate `well_founded` :

```
Inductive Acc (A : Type) (ℛ : A → A → Prop) (x : A) : Prop :=
| Acc_intro : (∀  y : A, ℛ y x → Acc A ℛ y) → Acc A ℛ x

Definition well_founded (A:Type)(ℛ:A → A → Prop) := ∀ a:A, Acc A ℛ a.
```

First, let us concentrate in the inductive predicate `Acc`. The inductive definition of `Acc` contemplates a single constructor, `Acc_intro`, whose arguments ensure the non existance of infinite $\mathcal{R}$-related sequences, that is, all the elements `y` that are related to `x` must lead to a finite descending sequence, since `y` satisfies `Acc`, which in turn is necessarily finite. The definition of `well_founded` universally quantifies over all the elements of type `A` that are related by $\mathcal{R}$.

The type `Acc` is inductively defined, and so it can be used as the structurally recursive argument in the definition of functions. Current versions of Coq provide two high level commands that ease the burden of manually constructing a recursive function over `Acc` predicates: the command Program [93, 94] and the command Function [13]. Here, we will only consider the Function command beacause it is the one used in our developments.

The command Function allows users to explicitly specify what is the recursive measure for the function to be implemented. In order to give an insight on how we can use Function to program non-structurally recursive functions, we present different implementations of the addition of two natural numbers. A first way of implementing addition is through the following function:

```
Function sum(x:nat)(y:nat){measure id x}:nat :=
  match x with
  | 0 ⇒ y
```

```
  | m ⇒ S (sum (m-1) y)
  end.
Proof.
  abstract(auto with arith).
Defined.
```

The annotation measure informs the Function command that the measure to be considered is the function id, applied to the argument x. A proof obligation is generated by Function, and is discharged by the given tactic. This obligation requires a proof that the x used in the recursive branch of sum is smaller than the original x under the less-than order defined over the natural numbers. The abstract tactic takes as argument another tactic that solves the current goal, and saves the proof of the goal as a separate lemma. The usage of abstract can be very useful, namely when the λ-term that proves the goal has a considerable large size, a fact that can have severe implications during computation or type-checking.

Another way to implement sum is to instruct the Function command so that it accepts as its recursive argument a proof term asserting that the relation less-than is well-founded.

```
Function sum1(x:nat)(y:nat){wf lt x}:nat :=
  match x with
  | 0 ⇒ y
  | m ⇒ S (sum1 (m-1) y)
  end.
Proof.
 abstract(auto with arith).
 exact(lt_wf).
Defined.
```

The definition of sum1 is identical to the one of sum, except for the annotation wf. In this case, Function yields two proof obligations: the first one is similar to the one of sum, and the second one asks for a proof that less-than is a well-founded relation. Both obligations are discharged automatically due to the auto tactic and the help of known lemmas and theorems available in the database arith.

The last way to use Function in order to build recursive definitions is to consider the struct annotation. In this case, functions are defined as if they were defined by the Fixpoint command.

```
Function sum2(x:nat)(y:nat){struct x}:nat :=
  match x with
  | 0 ⇒ y
  | S m ⇒ S (sum2 m y)
 end.
```

Besides allowing more general definitions of recursive functions than the `Fixpoint` command does, the command `Function` also automatically generates a fixpoint equation and an induction principle to reason about the recursive behaviour of the implemented function.

Performing reductions that involve well-founded induction proofs with a given relation is usually an issue in Coq. Such reductions may take too long to compute due to the complexity of the proof term involved. One way to get around is to use a technique proposed by Barras[1], whose idea is to add sufficient `Acc_intro` constructors, in a lazy way, on top of a `Acc` term, so that the original proof term is never reached during computation. The beauty of this technique is that the resulting term is logically equivalent to the original proof of the well founded relation. The general structure of the function is

```
Variable A : Type.
Variable R : relation A.
Hypothesis R_wf : well_founded R.

Fixpoint guard (n : nat)(wf : well_founded R) : well_founded R :=
  match n with
  | O ⇒ wf
  | S m ⇒ fun x ⇒ Acc_intro x (fun y _ ⇒ guard m (guard m wf) y)
  end.
```

In each recursive call, and when matching the term `Acc x H` constructed by the `guard` function, the reduction mechanisms find only `Acc_intro` terms, instead of some complex proof term. This improves computation considerably and yields better performances. For illustrating how the function `guard` can be used together with `Function`, we present a re-implementation of the function `sum1` where we discharge the second proof obligation by providing the type-checker with the result of `guard`:

```
Function sum1(x:nat)(y:nat){wf lt x}:nat :=
  match x with
  | O ⇒ y
  | m ⇒ S (sum1 (m-1) y)
  end.
Proof.
 abstract(auto with arith).
 exact(guard 100 _ lt_wf).
Defined.
```

[1] This technique has no official reference. To the best of our knowledge, the technique was communicated by Bruno Barras in a thread of the Coq-club mailing list.

2.1.5 Other Features of Coq

There are many other features of Coq that are very useful when conducting formalisations of mathematical theories, or certified program development. Below, we enumerate only the features that were relevant to the work presented in this thesis:

- an extraction mechanism, introduced by Paulin-Morhing [82], by Paulin-Morhing and Werner [84], and also by Letouzey [72]. This mechanism allows users to extract functional programs in OCaml, in Haskell, or in Scheme directly from their Coq developments. Based on the distinction between informative and logical types, extraction erases the logical contents and translates data types and function into one the functional languages mentioned above;
- it supports *type classes*. In Coq, the notion of type class extends the concept of a type class as seen in standard functional programming languages, in the sense that it allows proofs and dependent arguments in the type class definition. Type classes were developed by Sozeau and Oury [96] without extending the underlying Coq type system and relying on *dependent records*;
- a module system developed by Chrzaszcz [26] which allows users to conduct structured developments in a way similar to the one of OCaml;
- a *coercion* mechanism that automatically provides a notion of sub-typing;
- a new general rewriting mechanism implemented by Sozeau [95] that allows users to perform rewriting steps on terms where the underlying equality relation is not the one primitively available in Coq.

2.1.6 Sets in Coq

The Coq developments to be described in Chapters 5 and 6 make intensive use of sets. In this section we provide an overview of the available formalisations of sets in Coq that we have used. These formalisations are the Ensembles package of Coq's standard library and the Containers package, available in Coq's community contributions [69] and described in detail by Lescuyer in [70].

Sets as Predicates

A set is a collection of elements chosen from some universe. A set S can be determined extensionally, where a given predicate dictates which elements belong to S, and which do not. In this setting, such predicates are called *characteristic functions* mapping elements of universe into Boolean values. In the particular case of Coq, such notion of set is provided

by the package Ensembles, where the characteristic functions are implemented as predicates, *i.e.*, functions mapping values of a given type into propositions.

We do not use the package Ensembles directly in our development for the following reason: this package contains the specification of the axiom of *set extensionality* and therefore, whenever our development is subject to an extraction process, the extraction mechanism alerts the user for the existence of axioms that may lead to non-terminating functions and inconsistency, even when not using them at all. In order to avoid this kind of warning, we have incorporated in our development only the definitions present in Ensembles that we need. These definitions are

```
Section Sets.

 Variable U : Type.

 Definition Ensemble := U → Prop.

 Definition In(A:Ensemble) (x:U) : Prop := A x.

 Definition Included(B C:Ensemble) : Prop := ∀ x:U, In B x → In C x.

 Inductive Empty_set : Ensemble := .

 Inductive Singleton(x:U) : Ensemble :=
 |In_singleton : In (Singleton x) x.

 Inductive Union(B C:Ensemble) : Ensemble :=
 | Union_introl : ∀ x:U, In B x → In (Union B C) x
 | Union_intror : ∀ x:U, In C x → In (Union B C) x.

 Definition Same_set(B C:Ensemble) : Prop := Included B C ∧ Included C B.

End Sets.
```

Finite Sets and the Containers Library

Finite sets are provided in Coq's standard library by the packages FSets and MSets, with MSets being an evolution of FSets. Both libraries are implemented in a structured way using modules and functors. A detailed description of the FSets library is given in [37]. The main reason for not using any of these two libraries is the lack of flexibility that they have for our purposes: anytime we need a new kind of finite set, we have to instantiate a module signature

and them apply it into an adequate functor. If we need, for instance, sets of sets, we need to build a new signature and then instantiate it with a functor once again. The same happens if we need, for instance, sets of pairs whose components are of the type of an already existing set. Using the Containers package, these variations are obtained automatically, that is, once we define an instance of an ordered type, we can use sets of values of this type, and also sets of sets of this type, or sets of pairs of sets of this type, and so on. Summing up, the usage of the Containers library has the following advantages:

- finite sets are first-class values, an so they can be used like any other value, such like natural numbers;
- the development contains a vernacular command, `Generate OrderedType` that tries to automatically construct all the functional and logic content that is need to register the given inductive type t as an ordered type;
- sets of sets of a given type are always available for free due to the way the finite sets are defined in Containers. Each instance of a finite set contains a proof that asserts that the set itself is an ordered type.

The type class that defines an ordered type is the class `OrderedType`, which cotains the following parameters: a type `A`; an equality relation `_eq` and an order relation `_lt`; a proofs that `_eq` is an equivalence relation and a proof that `_lt` is a strict order; a computable comparison function `_cmp` defined over values of type `A`; finally, the soundness of the comparison function with respect to the order relation `_lt`. A particular instance of the class `OrderedType` is the class of ordered types where the considered equality relation is COQ's primitive equality. This class is named `UsualOrderedType`. Both classes are defined as follows:

```
Class OrderedType(A : Type) := {
  _eq : relation A;
  _lt : relation A;
  OT_Equivalence :> Equivalence _eq;
  OT_StrictOrder :> StrictOrder _lt _eq;
  _cmp : A → A → comparison;
  _compare_spec : ∀ x y, compare_spec _eq _lt x y (_cmp x y)
}.

Class SpecificOrderedType(A : Type)(eqA : relation A) := {
  SOT_Equivalence :> Equivalence eqA ;
  SOT_lt : relation A;
  SOT_StrictOrder : StrictOrder SOT_lt eqA;
  SOT_cmp : A → A → comparison;
  SOT_compare_spec : ∀ x y, compare_spec eqA SOT_lt x y (SOT_cmp x y)
```

```
}.

Notation "'UsualOrderedType' A" := (SpecificOrderedType A (@eq A)).
```

As an example, let us look at an excerpt of the Coq code that is needed to be able to use finite sets of values of `nat`. The equality relation considered is Coq's primitive equality. The comparison function is given by the fixpoint `nat_compare`.

```
Instance nat_StrictOrder : StrictOrder lt (@eq nat) := {
  StrictOrder_Transitive := lt_trans
}.

Fixpoint nat_compare (n m : nat) :=
 match n, m with
 | O, O ⇒ Eq
 | _, O ⇒ Gt
 | O, _ ⇒ Lt
 | S n, S m ⇒ nat_compare n m
 end.

Program Instance nat_OrderedType : UsualOrderedType nat := {
  SOT_lt := lt;
  SOT_cmp := nat_compare;
  SOT_StrictOrder := nat_StrictOrder
}.
```

The goal of the following function is to build a finite set of natural numbers lower than the argument n. The Coq code below presents such instantiation and the construction of the set containing all the numbers smaller or equal than a given natural number n.

```
Fixpoint all_smaller(n:nat) : set nat :=
 match n with
 | 0 ⇒ singleton 0
 | S m ⇒ add (S m) (all_smaller m)
 end.
```

2.2 Hoare Logic

The contributions described in Chapters 4 and 5 target program verification of sequential and parallel programs, respectively. Both works are closely related to the well known *Floyd-Hoare logic*, usually known as Hoare logic [38, 48]. Using Hoare logic, we are able to prove

that a program is correct by applying a finite set of inference rules to an initial program specification of the form

$$\{P\}\, C\, \{Q\}, \tag{2.1}$$

such that P and Q are logical assertions, and C is a program. The intuition behind such a specification, widely known as *Hoare triple* or as *partial correctness assertion* (PCA), is that if the program C starts executing in a state where the assertion P is true, then if C terminates, it does so in a state where the assertion Q holds. The assertions P and Q are usually called *preconditions* and *postconditions*, respectively. Moreover, and since we assume that the program C might not terminate, we will be using Hoare logic for proving the partial correctness of programs.

A Simple Imperative Programming Language and its Semantics

The set of inference rules of Hoare logic is tightly connected to the inductive syntax of the target programming language, in the sense that each program construction is captured by an inference rule. Here we consider a typical imperative language with assignments, two-branched conditional instructions, and *while* loops. We will denote this language by IMP. The syntax of IMP programs is inductively defined by

$$\begin{aligned} C, C_1, C_2 \;::=\; & \mathsf{skip} \\ & \mid x \;:=\; E \\ & \mid C_1 \;;\; C_2 \\ & \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ C_1\ \mathsf{end}, \end{aligned}$$

where x is a variable of the language, E is an arithmetic expression, and B is Boolean expression. For the simplicity of the presentation, we omit the language of expressions and assume that variables of IMP can have only one of two following types: integers and Booleans. IMP programs are interpreted in a standard *small-step structural operational semantics* [86], where there exists the notion of state (a set of variables and corresponding assigned values). Programs are computed by means of a *evaluation function* that take *configurations* $\langle C, s\rangle$ into new configurations The expression

$$\langle C, s\rangle \Longrightarrow^{\star} \langle \mathsf{skip}, s'\rangle \tag{2.2}$$

intuitively states that operationally evaluating the program C in the state s leads to the termination of the program in the state s', using a finite number of individual evaluation steps guided by the syntactical structure of C. The individual evaluation rules for IMP programs are

$$\frac{}{\langle\, x := E, s \,\rangle \implies \langle\, \mathsf{skip}, s[[\![E]\!]_{\mathbb{E}}/x] \,\rangle}\ (\textsc{Assgn})$$

$$\frac{\langle\, C_1, s \,\rangle \implies \langle\, C_1', s' \,\rangle}{\langle\, C_1; C_2, s \,\rangle \implies \langle\, C_1'; C_2, s' \,\rangle}\ (\textsc{SeqStep})$$

$$\frac{}{\langle\, \mathsf{skip}; C_2, s \,\rangle \implies \langle\, C_2, s \,\rangle}\ (\textsc{SeqSkip})$$

$$\frac{[\![B]\!]_{\mathbb{B}}(s) \;=\; \mathbf{true}}{\langle\, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s \,\rangle \implies \langle\, C_1, s \,\rangle}\ (\textsc{IfTrue})$$

$$\frac{[\![B]\!]_{\mathbb{B}}(s) \;=\; \mathbf{false}}{\langle\, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s \,\rangle \implies \langle\, C_2, s \,\rangle}\ (\textsc{IfFalse})$$

$$\frac{}{\langle\, \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{end}, s \,\rangle \implies \langle\, \mathsf{if}\ B\ \mathsf{then}\ (C\ ;\ \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{end})\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}, s \,\rangle}\ (\textsc{While})$$

The function $[\![B]\!]_{\mathbb{B}}$ is a function that denotationaly evaluates Boolean expressions in states, and returns the corresponding Boolean value for the Boolean expression given as argument. The function $[\![E]\!]_{\mathbb{E}}$ evaluates arithmetic expressions also in a denotational way. This kind of semantics, and alternative ones, can be consulted in standard textbooks about programming language semantics such as [77, 100, 45].

Hoare Logic's Proof System

Hoare logic is a proof system formed by a set of inference rules that correspond to fundamental laws of programs. Each inference rule consists of zero or more premisses and a unique conclusion. Here, and along the remaining of this dissertation, we will be considering Hoare proof systems that are intended to be used for proving *partial correctness*. This means that the inference rules consider programs that may not terminate.

In Hoare logic, a *deduction* assumes the form of a tree whose nodes are labeled by specifications, and whose sub-trees are deductions of the premisses of the inference rules applied to the nodes. The leaves of the deduction trees are nodes to which no more inference rules

can be applied, and the root of the tree is the specification of the correctness of the program under consideration. These trees are usually named as *proof trees*, or *derivation trees*, and represent the proof of the correctness a program.

Before entering the technical details of the proof system, there is one more result about Hoare logic that allows us to be certain that the proofs that we construct are indeed correct. Using other words, Hoare logic is *sound* with respect to the semantics of the target programs, *i.e.*, all Hoare triples $\{P\}\,C\,\{Q\}$ that we are derivable are *valid* in the following sense: if the predicate P holds in some state s, and if the program C terminates, then it must do so in a state s' such that the predicate Q holds.

The Hoare logic inference rules for proving the partial correctness of IMP programs are the following:

Skip: the following rule simply states that the preconditions and the postconditions of a skip program must be the same, since there is no computation involved.

$$\frac{}{\{P\}\ \mathsf{skip}\ \{P\}}\ (\text{HL-Skip})$$

Assignment: if we want to show that the assertion P holds after the assignment of the expression E to the variable x, we must show that $P[E/x]$ (the substitution of the free occurrences of x by E in P) holds before the assignment. We will apply this rule backwards. We know P and we wish to find a precondition that makes P true after the assignment $x := E$.

$$\frac{}{\{P[E/x]\}\,x := E\,\{P\}}\ (\text{HL-Assgn})$$

Composition: if C_1 transforms a state satisfying P into a state satisfying Q', and that if C_2 takes a state satisfying Q' into a state satisfying Q, then if P is true, the execution of C_1 followed by C_2 takes the program into a state satisfying the postcondition Q.

$$\frac{\{P\}\,C_1\,\{Q'\} \qquad \{Q'\}\,C_2\,\{Q\}}{\{P\}\,C_1; C_2\,\{Q\}}\ (\text{HL-Seq})$$

Conditional: if b is true in the starting state, then C_1 is executed and Q becomes true; alternatively, if B is false, then C_2 is executed. The preconditions depend on the truth value of B. This additional information is often crucial for completing the corresponding sub-proofs.

$$\frac{\{B \wedge P\}\,C_1\,\{Q\} \qquad \{\neg B \wedge P\}\,C_2\,\{Q\}}{\{P\}\ \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}\ \{Q\}}\ (\text{HL-If})$$

While: given an invariant P which must be true in each iteration of the while loop, then when B is false, that is, when the loop condition fails, the invariant must be true, no matter how many times the loop has been repeated before.

$$\frac{\{B \wedge P\}\, C\, \{P\}}{\{P\}\,\mathsf{while}\, B\,\mathsf{do}\, C\,\mathsf{end}\ \{\neg B \wedge P\}}\ (\text{HL-W}\textsc{hile})$$

Weakening: if we have proved that $\{P'\}\ C\ \{Q'\}$, and if we also know that P implies P', and that Q' implies Q, then we can strengthen the precondition and weaken the postcondition.

$$\frac{P \rightarrow P' \qquad \{P'\}\, C\, \{Q'\} \qquad Q' \rightarrow Q}{\{P\}\, C\, \{Q\}}\ (\text{HL-W}\textsc{eak})$$

We denote this proof system by HL. We say that a Hoare triple $\{P\}\, C\, \{Q\}$ is *derivable* in HL, and write $\vdash_{\mathsf{HL}}\ \{P\}\, C\, \{Q\}$ if we can build a proof tree for the triple $\{P\}\, C\, \{Q\}$ using the previous rules. We may also have a derivation in the presence of a set of assumption $\mathcal{A}$ and we write $\mathcal{A} \vdash_{\mathsf{HL}}\ \{P\}\, C\, \{Q\}$. Side conditions are introduced by the usage of the (HL-Weak) rule in the derivation process. This rule allows to relate external first-order assertions with the local specifications.

Proof trees can be constructed by a special purpose algorithm called *verification condition generator* (VCGen), which uses specific heuristics to infer the side conditions from one particular derivation. The input for a VCGen algorithm is a Hoare triple, and the output is a set of first-order proof obligations. For this to happen, the inference rules of the proof system must be changed so that the following conditions always hold:

1. assertions occurring in the premisses must be sub-formulas of the conclusion, so that discovering intermediate assertions is required;
2. the set of inference rules must be unambiguous in order for the derivation tree construction process can be syntax-oriented.

Instead of HL, we can consider an alternative Hoare proof system that is syntax directed and that enjoys the sub-formula property. We consider a version of IMP with annotated commands, defined by following grammar:

$$\begin{aligned} C, C_1, C_2\ ::= &\ \mathsf{skip} \\ &\mid x\ :=\ E \\ &\mid C_1\ ;\ \{P\}\ C_2 \\ &\mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \\ &\mid \mathsf{while}\ B\ \mathsf{do}\ \{I\}\ \ C\ \mathsf{end}. \end{aligned}$$

The set of rules of the considered proof system, which we denote by HLa, is the following:

$$\frac{P \to Q}{\{P\}\ \mathsf{skip}\ \{Q\}}\ (\text{HL-AnnSkip})$$

$$\frac{P \to Q[E/x]}{\{P\}\ x := E\ \{Q\}}\ (\text{HL-AnnAssgn})$$

$$\frac{\{P\}\ C_1\ \{Q'\} \qquad \{Q'\}\ C_2\ \{Q\}}{\{P\}\ C_1;\ \{Q'\}\ C_2\ \{Q\}}\ (\text{HL-AnnSeq})$$

$$\frac{\{B \wedge P\}\ C_1\ \{Q\} \qquad \{\neg B \wedge P\}\ C_2\ \{Q\}}{\{P\}\ \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}\ \ \{Q\}}\ (\text{HL-AnnIf})$$

$$\frac{P \to I \qquad \{I \wedge B\}\ C\ \{I\} \qquad I \wedge \neg B \to Q}{\{P\}\ \mathsf{while}\ B\ \mathsf{do}\ \{I\}\ C\ \mathsf{end}\ \{Q\}}\ (\text{HL-AnnWhile})$$

The system HLa can be proved to infer the same proof trees as the system HL. Such proof is available in the work of Frade and Pinto [39], as well as the treatment of extensions to the underlying programming language and the formal treatment of such extensions at the level of the corresponding proof systems.

The concepts of Hoare logic presented until now are the ones that we require as a base for the contributions described in Chapter 4 and Chapter 5. There exists, however, much more to say about Hoare logic. In recent years, the particular subject of program verification by means of Hoare logic and related concepts has evolved considerably, mainly in terms of tool support, such as the Why3 system [17, 16] and the Boogie system [30, 31].

2.3 Conclusions

In this chapter we have introduced the Coq proof assistant, the tool that we have used to mechanize the theories to be introduced in the next three chapters. We have also described Hoare logic, the program logic that serves as the base for the contributions presented in Chapter 4 and Chapter 5.

Chapter 3

Equivalence of Regular Expressions

Formal languages are one of the pillars of Computer Science. Amongst the computational models of formal languages, that of *regular expression* is one of the most used, having applications in many areas. The notion of regular expressions has its origins in the seminal work of Kleene [55], where he introduced them as a specification language for *deterministic finite automata*.

Regular expressions are certainly very famous due to their capability of matching patterns, and they abound in the technologies deriving from the *World Wide Web*, in text processors, in structured languages such as XML, in the design of programming languages like Perl and Esterel [14]. More recently, regular expressions also found applications in the run-time monitoring of programs [91, 92].

In this chapter we describe the mechanisation, in the proof assistant Coq, of a considerable fragment of the theory of regular languages. We also present the implementation and the proofs of correctness and completeness of a decision procedure for regular expressions equivalence using the notion of derivative, an alternative to the approaches based on automata constructions. The Coq development is available online from [75].

3.1 Elements of Language Theory

In this section we introduce some classic concepts of formal languages that we will need in the work we are about to describe. These concepts can be found in the introductory chapters of classical textbooks such as the one by Hopcroft and Ullman [50] or the one by Kozen [58]. Along this section, the most relevant definitions are accompanied by the corresponding Coq code fragment.

3.1.1 Alphabets, Words, and Languages

Alphabet

An *alphabet* Σ is a non-empty finite set of objects usually called *symbols* (or *letters*). Standard examples of alphabets are the sets $\Sigma_1 = \{0, 1\}$ and $\Sigma_2 = \{a, b, c\}$. An alphabet is specified by the following module signature:

```
Module Type Alphabet.
  Parameter A : Type.
  Declare Instance AOrd : UsualOrderedType A.
End Alphabet.
```

The type `A` is the type of symbols, and it is an ordered type as specified by the type class instance `AOrd`. The equality relation over `A` is Coq's primitive equality, as imposed by the definition of the `UsualOrderedType` type class. Let us see an example of the definition of an alphabet by providing a concrete module for the module signature above.

Example 1. *Consider the following alphabet $\Sigma = \{a, b, c\}$. This alphabet is encoded in* Coq *by the following module:*

```
Module Alph : Alphabet.
  Inductive alph : Type := a | b | c.
  Definition A := alph.
  Generate OrderedType alph.
  Program Instance AOrd : UsualOrderedType A := {
    SOT_lt := A_lt ;
    SOT_cmp := A_cmp
  }.
End Alph.
```

The previous example makes use of the Coq command `Generate OrderedType` to generate the definitions and proofs that are required to establish the type `A` as an ordered type, considering Coq's primitive equality.

Words

A *word* (or *string*) over an alphabet Σ is a finite sequence of symbols from Σ. The natural way of defining words in Coq is by using the already available type of polymorphic list `list` and instantiating it with the type of the symbols that are used in Σ. We name this type `word`. We refer to words by the lower-case letters w, u, v, and so on. Naturally, the *empty word* ϵ is the term `@nil A`. The *concatenation* of two words w and u over Σ, denoted by $w \cdot u$, or simply by wu, is the same as standard list concatenation w `++` u.

Example 2. *Let $\Sigma = \{a, b, c\}$ be the alphabet under consideration. The set of all words on Σ is the set*

$$\{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, aba, \ldots\}$$

Languages

A *language* is any finite or infinite set of words over an alphabet Σ. The type of languages is the type of predicates over values of type `word`. Hence, we define predicates through the instantiation of the predicate `Ensemble`, introduced in Section 2.1.6, with the type `word`.

```
Definition language := Ensemble word.
```

Given an alphabet Σ, the set of all words over Σ, denoted by $\Sigma^\star$, is inductively defined as follows: the empty word ϵ is an element of $\Sigma^\star$ and, if $w \in \Sigma^\star$ and $a \in \Sigma$, then aw is also a member of $\Sigma^\star$. The set $\Sigma^\star$ is defined in Coq by the following inductive predicate:

```
Inductive sigma_star : language :=
| in_sigma_star_nil : [] ∈ sigma_star
| in_sigma_star : ∀ w:word, w ∈ sigma_star → ∀ a:A, a::w ∈ sigma_star.
```

The *empty language*, the language containing only ϵ, and the language containing only a symbol $a \in \Sigma$ are denoted, respectively, by $\emptyset$, $\{\epsilon\}$, and $\{a\}$. They are defined in the following way:

```
Definition empty_l := (Empty_set word).
Notation "∅" := empty_l.

Definition eps_l := (Singleton word []).
Notation "{ϵ}" := eps_l.

Inductive symb_l (a:A) : language := in_sing : [a] ∈ symb_l a.
Notation "{{x}}" := (symb_l x)(at level 0).
```

The operations over languages include the usual Boolean set operations (union, intersection, and complement), plus *concatenation*, *power* and *Kleene star*. The concatenation of two languages L_1 and L_2 is defined by

$$L_1L_2 \stackrel{\text{def}}{=} \{wu \,|\, w \in L_1 \wedge u \in L_2\}. \tag{3.1}$$

The *power* of a language L, denoted by L^n, with $n \in \mathbb{N}$, is inductively defined by

$$\begin{aligned} L^0 &\stackrel{\text{def}}{=} \{\epsilon\}, \\ L^{n+1} &\stackrel{\text{def}}{=} LL^n. \end{aligned} \tag{3.2}$$

The *Kleene star* of a language L is the union of all the finite powers of L, that is,

$$L^\star \stackrel{\text{def}}{=} \bigcup_{i\geq 0} L^i. \tag{3.3}$$

The operations (3.1-3.3) are defined in Coq through the next three inductive definitions.

```
Inductive conc_l (l₁ l₂:language) : language :=
|conc_l_app : ∀ w₁ w₂:word,
   w₁ ∈ l₁ → w₂ ∈ l₂ → (w₁ ++ w₂) ∈ (conc_l l₁ l₂)
Notation "x • y" := (conc_l x y).

Fixpoint conc_ln (l:language)(n:nat) : language :=
 match n with
| 0 ⇒ eps_l
| S m ⇒ l • (conc_ln l m)
Notation "x •• n" := (conc_ln x n).

Inductive star_l (l:language) : language :=
| starL_n : ∀ (n:nat)(w:word), w ∈ (l •• n) → w ∈ (star_l l)
Notation "x ∗" := (star_l x).
```

A language L over an alphabet Σ is a *regular language* if it is inductively defined as follows:

- the languages $\emptyset$ and $\{\epsilon\}$ are regular languages;
- for all $a \in \Sigma$, the language $\{a\}$ is a regular language;
- if L_1 and L_2 are regular languages, then $L_1 \cup L_2$, L_1L_2, and $L_1^\star$ are regular languages.

The predicate that determines whether a given language is regular or not is formalised in Coq as follows:

```
Inductive rl : language → Prop :=
| rl0 : rl ∅
| rl1 : rl {ϵ}
| rlsy : ∀ a, rl {{a}}
| rlp : ∀ l₁ l₂, rl l₁ → rl l₂ → rl (l₁ ∪ l₂)
| rlc : ∀ l₁ l₂, rl l₁ → rl l₂ → rl (l₁ • l₂)
| rlst : ∀ l, rl l → rl (l ∗).
```

We denote language equality by $L_1 = L_2$.

```
Definition leq (l₁ l₂:language) := l₁ ⊆ l₂ ∧ l₂ ⊆ l₁.
Notation "x == y" := (leq x y).
Notation "x != y" := (¬(x == y)).
```

Finally, we introduce the concept of the *left-quotient* of a language L with respect to a word $w \in \Sigma^\star$, and which we denote by $\mathcal{D}_w(L)$. The left-quotient is defined as follows:

$$\mathcal{D}_w(L) \stackrel{\text{def}}{=} \{v \mid wv \in L\}. \tag{3.4}$$

In particular, if in (3.4) we have $w = a$, with $a \in \Sigma$, then we say that $\mathcal{D}_a(L)$ is the left-quotient of L with respect to the symbol a. The Coq definitions of LQ and LQw given below are, respectively, the notions of left-quotient with respect to a symbol in Σ and the left-quotient with respect to a word in $\Sigma^\star$.

```
Inductive LQ (L:language) : A → language :=
| in_quo : ∀ (x:word)(a:A), (a::x) ∈ L → x ∈ (LQ L a)
Notation "x %Lq y" := (LQ x y).

Inductive LQw (L:language) : word → language :=
| in_quow : ∀ (w1 w2:word), (w1 ++ w2) ∈ L → w2 ∈ (LQw L w1)
Notation "x %Lqw y" := (LQw x y).
```

3.1.2 Finite Automata

A *deterministic finite automaton* (DFA) is a 5-tuple $D = (Q, \Sigma, \delta, q_0, F)$ such that Q is the set of *states*, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ is the *transition function*, q_0 is the *initial state*, and F is the set of *accepting states* (or *final states*).

A DFA D can be described graphically by a *transition diagram*, that is, a *digraph* such that each node is a state of D, each arc is labeled by a symbol $a \in \Sigma$ and represents a transition between two states, the initial state is marked by an unlabelled input arrow, and all the accepting states are marked by a double circle. Figure 3.1 presents the transition diagram of the following DFA D:

$$D = (\{q_0, q_1\}, \Sigma_1, \{(q_0, 0, q_0), (q_0, 1, q_1), (q_1, 0, q_0), (q_1, 1, q_1), q_0, \{q_1\}).$$

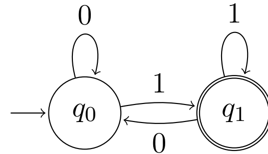


Figure 3.1: The transition diagram of the DFA D.

A DFA D processes an input word w through the *extended transition function* $\hat{\delta}$, which is

inductively defined as follows:

$$\begin{array}{rcl} \hat{\delta}(q,\epsilon) & \stackrel{\text{def}}{=} & q, \\ \hat{\delta}(q,aw) & \stackrel{\text{def}}{=} & \hat{\delta}(\delta(q,a),w). \end{array} \tag{3.5}$$

Considering (3.5), we define the notion of the *language accepted* (or *recognised*) by a DFA. Let $D = (Q, \Sigma, \delta, q_0, F)$ be the automaton under consideration. The language of D, denoted by $\mathcal{L}(D)$, is the set of all words that take D from its initial state into one of its final states, *i.e.*,

$$\mathcal{L}(D) \stackrel{\text{def}}{=} \{w \mid \hat{\delta}(q_0, w) \in F\}.$$

The language recognised by any DFA is always a regular language [55]. The next example shows how DFAs process words given as input, using the extended transition function.

Example 3. *Let D be the DFA presented in Figure 3.1. This automaton processes the word $w = 101$ in the following way:*

$$\begin{array}{rcl} \hat{\delta}(q_0,101) & = & \hat{\delta}(\delta(q_0,1),01) \\ & = & \hat{\delta}(q_1,01) \\ & = & \hat{\delta}(\delta(q_1,0),1) \\ & = & \hat{\delta}(q_0,1) \\ & = & \hat{\delta}(\delta(q_0,1),\epsilon) \\ & = & \hat{\delta}(q_1,\epsilon) \\ & = & q_1 \end{array}$$

Two DFAs D_1 and D_2 are *equivalent* if the languages they recognise are the same, that is, if $\mathcal{L}(D_1) = \mathcal{L}(D_2)$. We denote DFA equivalence by $D_1 \sim D_2$.

A DFA D is *minimal* if all the DFAs D' such that $D \sim D'$ have no less states than D. Furthermore, any minimal DFA is unique up to isomorphism.

A *non-deterministic finite automaton* (NFA) is a 5-tuple $N = (Q, \Sigma, \delta, q_0, F)$ such that Q is the finite set of states, Σ is the alphabet, q_0 is the initial state, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, and F is the set of accepting states. Like DFAs, a NFA can be represented by a transition diagram. Figure 3.2 presents the transition diagram of a NFA that recognises all the sequences of 0's and 1's that end with a 1.

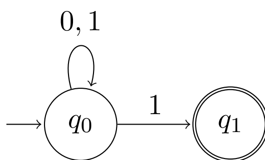


Figure 3.2: A NFA accepting sequences of 0's and 1's that end with a 1.

Note that the main difference between the definition of DFA and that of NFA is in the transition function: while the transition function of a DFA is a function from a state into another state, in a NFA the transition function returns a set of accessible states. For NFAs, the extended transition function is inductively defined as follows:

$$\begin{array}{rcl}\hat{\delta}(q,\epsilon) & \stackrel{\text{def}}{=} & \{q\}, \\ \hat{\delta}(q,aw) & \stackrel{\text{def}}{=} & \bigcup_{q' \in \delta(q,a)} \hat{\delta}(q',w).\end{array}$$

Next, we present an example on how NFAs process words using the definition given above.

Example 4. *Given the NFA presented in Figure 3.2 and the word $w = 101$, the computation of $\hat{\delta}(q_0, 101)$ goes as follows:*

$$\begin{array}{rcl}\hat{\delta}(q_0,101) & = & \hat{\delta}(\delta(q_0,1),01) \\ & = & \hat{\delta}(q_0,01) \cup \hat{\delta}(q_1,01) \\ & = & \hat{\delta}(\delta(q_0,0),1) \cup \hat{\delta}(\delta(q_1,0),1) \\ & = & \hat{\delta}(q_0,1) \cup \emptyset \\ & = & \hat{\delta}(\delta(q_0,1),\epsilon) \\ & = & \hat{\delta}(q_0,\epsilon) \cup \hat{\delta}(q_1,\epsilon) \\ & = & \{q_0,q_1\}\end{array}$$

The language recognised by a NFA N is the language

$$\mathcal{L}(N) \stackrel{\text{def}}{=} \{w \mid \hat{\delta}(q_0,w) \cap F \neq \emptyset\},$$

which is also a regular language. If the languages recognised by two NFAs N_1 and N_2 coincide, that is, if $\mathcal{L}(N_1) = \mathcal{L}(N_2)$, then N_1 and N_2 are equivalent NFAs and we write $N_1 \sim N_2$.

Our development does not consider formalisation of automata. Details on the formalisation of this particular subject in proof assistants can be found in the works of Filliâtre [36], Briais [20], and Braibant and Pous [18, 19].

3.1.3 Regular Expressions

Let Σ be an alphabet. Regular expressions over Σ are denoted by $\alpha, \beta, \alpha_1, \beta_1$, and are inductively defined as follows:

- the constants 0 and 1 are regular expressions;
- all the symbols $a \in \Sigma$ are regular expressions;
- if α and β are regular expressions, then their *union* $\alpha + \beta$ and their *concatenation* $\alpha\beta$ are regular expressions as well ;
- finally, if α is a regular expression, then so is its *Kleene star* $\alpha^\star$.

We denote the syntactical equality of two regular expressions α and β by $\alpha \equiv \beta$. The set of all regular expressions over an alphabet Σ is denoted by RE_Σ. In Coq, regular expressions are inhabitants of the following inductive type:

```
Inductive re : Type :=
| re0 : re                          Notation "0" := re0.
| re1 : re                          Notation "1" := re1.
| re_sy : A → re                    Notation "`a" := (re_sy a).
| re_union : re → re → re           Infix "+" := re_union.
| re_conc : re → re → re            Infix "·" := re_conc.
| re_star : re → re.                Notation "x *" := (re_star x).
```

Language of Regular Expressions

Regular expressions *denote* regular languages. The language denoted by a regular expression α, $\mathcal{L}(\alpha)$, is inductively defined in the expected way:

$$
\begin{aligned}
\mathcal{L}(0) &\stackrel{\text{def}}{=} \emptyset,\\
\mathcal{L}(1) &\stackrel{\text{def}}{=} \{\epsilon\},\\
\mathcal{L}(a) &\stackrel{\text{def}}{=} \{a\},\ a \in \Sigma\\
\mathcal{L}(\alpha+\beta) &\stackrel{\text{def}}{=} \mathcal{L}(\alpha) \cup \mathcal{L}(\beta),\\
\mathcal{L}(\alpha\beta) &\stackrel{\text{def}}{=} \mathcal{L}(\alpha)\mathcal{L}(\beta),\\
\mathcal{L}(\alpha^\star) &\stackrel{\text{def}}{=} \mathcal{L}(\alpha)^\star.
\end{aligned}
$$

In the code below, the language $\mathcal{L}(\alpha)$ is given by the definition of the recursive function `re_rel`.

```
Fixpoint re_rel (α:re) : language :=
  match α with
  | re0 ⇒ ∅
  | re1 ⇒ {ϵ}
  | re_sy a ⇒ {{a}}
  | re_union α1 α2 ⇒ (re_rel α1) ∪ (re_rel α2)
  | re_conc α1 α2 ⇒ (re_rel α1) • (re_rel α2)
  | re_star α1 ⇒ (re_rel α1)⋆
end.
Notation "ℒ(α)" := (re_rel α).

Coercion re2rel : re ↣ language.
```

Besides defining `re_rel`, we also mark it as a coercion from the type of regular expressions to the type of languages. This allows us to refer to the language of a regular expression, $\mathcal{L}(\alpha)$, simply by α, since the term `re_rel`(α) is automatically inferred by COQ, if possible. Easily, we prove that the result of `re_rel` is always a regular language.

```
Theorem re2rel_rl : ∀ α:re, rl (α).
```

Measures and Nullability

The *length* of a regular expression α, denoted $|\alpha|$, is the total number of constants, symbols and operators of α. The *alphabetic length* of a regular expression α, denoted $|\alpha|_\Sigma$, is the total number of occurences of symbols of Σ in α.

We say that a regular expression α is *nullable* if $\epsilon \in \mathcal{L}(\alpha)$ and *non-nullable* otherwise. Considering the definition

$$\varepsilon(\alpha) \stackrel{\text{def}}{=} \begin{cases} \texttt{true} & \text{if } \epsilon \in \mathcal{L}(\alpha), \\ \texttt{false} & \text{otherwise,} \end{cases}$$

we say that the regular expressions α and β are *equi-nullable* if $\varepsilon(\alpha) = \varepsilon(\beta)$. The function that we recursively define below in COQ determines if a given regular expression is nullable or not in a syntactical way[1]:

```
Fixpoint nullable(α:re) : bool :=
 match α with
  | re0 ⇒ false
  | re1 ⇒ true
  | re_sy _ ⇒ false
  | re_star _ ⇒ true
  | re_union α1 α2 ⇒ nullable α1 || nullable α2
  | re_conc α1 α2 ⇒ nullable α1 && nullable α2
 end.
Notation "ε(y)" := (nullable y).
```

The soundness and completeness of the nullability of regular expressions is given by the two next theorems, both of which are proved by induction on the structure of the given regular expression and using simple properties of the membership of the empty word in languages.

```
Theorem null_correct_true : ∀ α:re, ε(α) = true ↔ ϵ ∈ α.
Theorem null_correct_false : ∀ α:re, ε(α) = false ↔ ϵ ∉ α.
```

[1] `nullable` is a Boolean function; therefore, the binary operators `||` and `&&` correspond to the usual Boolean operations of disjunction and conjunction, respectively.

Finite Sets of Regular Expressions

Finite sets of regular expressions are defined by the type `set re`. The language of a finite set of regular expressions S is

$$\mathcal{L}(S) \stackrel{\text{def}}{=} \bigcup_{\alpha_i \in S} \mathcal{L}(\alpha_i). \tag{3.6}$$

Naturally, two sets of regular expressions S_1 and S_2 are equivalent if $\mathcal{L}(S_1) = \mathcal{L}(S_2)$, which we denote by $S_1 \sim S_2$. Equation (3.6) is defined in Coq through the predicate `SreL`, defined below.

```
Inductive SreL : set re → language :=
| in_sre_lang : ∀ (s:set re) (w:word) (α:re),
    α ∈ s → w ∈ α → w ∈ (SreL s).
Notation "ℒ(s)" := (SreL s).
```

Given a set of regular expressions $S = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, we define

$$\sum S \stackrel{\text{def}}{=} \alpha_1 + \alpha_2 + \ldots + \alpha_n.$$

Naturally, the language of such a sum of elements of S is trivially equivalent to $\mathcal{L}(S)$, and is defined by

$$\mathcal{L}\left(\sum S\right) \stackrel{\text{def}}{=} \mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2) \cup \cdots \cup \mathcal{L}(\alpha_n).$$

Nullability extends to sets of regular expressions in a straightforward way: a set S is nullable if $\varepsilon(\alpha)$ evaluates positively, that is, if $\varepsilon(\alpha) = \texttt{true}$ for at least one $\alpha \in S$. We denote the nullability of a set of regular expressions S by $\varepsilon(S)$. Two sets of regular expressions S_1 and S_2 are equi-nullable if $\varepsilon(S_1) = \varepsilon(S_2)$. Nullability of sets of regular expressions is expressed in our development by the next definition.

```
Definition null_set (s:set re) := fold (fun α:re ⇒ orb (ε(α))) s false.
Notation "ε(s)'' := (null_set s).
```

We also consider the *right-concatenation* $S \odot \alpha$ of a regular expression α with a set of regular expressions S, which is defined as follows:

$$S \odot \alpha = \begin{cases} \emptyset & \text{if } \alpha = 0, \\ S & \text{if } \alpha = 1, \\ \{\beta\alpha \mid \beta \in S\} & \text{otherwise.} \end{cases} \tag{3.7}$$

The *left-concatenation*, denoted $\alpha \odot S$, is defined in an analogous away. We usually omit the operator $\odot$ and write $S\alpha$ and αS instead. Equation (3.7) is defined in Coq in the following straightforward way:

```
Definition fold_conc(s:set re)(α:re) := map (fun β ⇒ β · α) s.

Definition dsr(s:set re)(α:re) : set re :=
  match α with
   | 0 ⇒ ∅
   | 1 ⇒ s
   | _ ⇒ fold_conc s α
 end.

Notation "s ⊙ α" := (dsr s α).
```

3.1.4 Kleene Algebra

An idempotent semiring is an algebraic structure $(K, +, \cdot, 0, 1)$, satisfying the following set of axioms:

$$x + x = x \tag{3.8}$$

$$x + 0 = x \tag{3.9}$$

$$x + y = y + x \tag{3.10}$$

$$x + (y + z) = (x + y) + z \tag{3.11}$$

$$0x = 0 \tag{3.12}$$

$$x0 = 0 \tag{3.13}$$

$$1x = x \tag{3.14}$$

$$x1 = x \tag{3.15}$$

$$x(yz) = (xy)z \tag{3.16}$$

$$x(y + z) = xy + xz \tag{3.17}$$

$$(x + y)z = xz + yz. \tag{3.18}$$

The natural partial ordering on such a semiring is $x \leq y \Leftrightarrow x + y = y$. A *Kleene algebra* (KA) is an algebraic structure $(K, +, \cdot, ^\star, 0, 1)$ such that the sub-algebra $(K, +, \cdot, 0, 1)$ is an idempotent semiring and that the operator * is characterised by the following set of axioms:

$$1 + pp^\star \leq p^\star \tag{3.19}$$

$$1 + p^\star p \leq p^\star \tag{3.20}$$

$$q + pr \leq r \rightarrow p^\star q \leq r \tag{3.21}$$

$$q + rp \leq r \rightarrow qp^\star \leq r \tag{3.22}$$

The algebraic structure $(\mathrm{RE}_\Sigma, +, \cdot, ^\star, 0, 1)$ is a KA, namely, the free KA on the generator Σ (the alphabet under consideration). The standard model of KA is the model

$(\mathrm{RL}_\Sigma, \cup, \cdot, ^\star, \emptyset, \{\epsilon\})$, where RL_Σ is the set of all regular languages over Σ. Kozen proved the completeness of KA with respect to this model through an algebraic treatment of the classical approach used to decide regular expression equivalence by means of automata [57]. Other models of KA include the model of binary relations, the model of matrices over a KA and the model of tropical semirings.

3.2 Derivatives of Regular Expressions

The notion of *derivative* of a regular expression α was introduced by Brzozowski in the 60's [22], and was motived by the construction of sequential circuits directly from regular expressions extended with intersection and complement. In the same decade, Mirkin introduced the notion of *prebase* and *base* of a regular expression as a method to construct NFA recognising the corresponding language [73]. His definition is a generalisation of Brzozowski's derivatives for NFAs and was independently re-discovered almost thirty years later by Antimirov [8], which coined it as *partial derivative* of a regular expression.

Brzozowski's Derivatives

Let Σ be an alphabet, α a regular expression over Σ, and $a \in \Sigma$. The *Brzozowski derivative* of α with respect to a, or simply *derivative* of α with respect to a, is recursively defined as follows:

$$
\begin{aligned}
a^{-1}(0) &\stackrel{\text{def}}{=} 0,\\
a^{-1}(1) &\stackrel{\text{def}}{=} 0,\\
a^{-1}(b) &\stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } a \equiv b,\\ 0 & \text{otherwise,}\end{cases}\\
a^{-1}(\alpha+\beta) &\stackrel{\text{def}}{=} a^{-1}(\alpha) + a^{-1}(\beta),\\
a^{-1}(\alpha\beta) &\stackrel{\text{def}}{=} \begin{cases} a^{-1}(\alpha)\beta + a^{-1}(\beta) & \text{if } \varepsilon(\alpha) = \texttt{true},\\ a^{-1}(\alpha)\beta & \text{otherwise,}\end{cases}\\
a^{-1}(\alpha^\star) &\stackrel{\text{def}}{=} a^{-1}(\alpha)\alpha^\star.
\end{aligned}
$$

The intuition behind the derivation function is that it acts as a residual operation on $\mathcal{L}(\alpha)$, since it removes, from each word of $\mathcal{L}(\alpha)$ the symbol $a \in \Sigma$ that is in the beginning of such a word.

The notion of derivative can be naturally extended to words in the following way, where $u \in \Sigma^\star$:

$$
\begin{aligned}
\epsilon^{-1}(\alpha) &\stackrel{\text{def}}{=} \alpha,\\
(ua)^{-1}(\alpha) &\stackrel{\text{def}}{=} a^{-1}(u^{-1}(\alpha)).
\end{aligned}
$$

The language denoted by $a^{-1}(\alpha)$ is the left-quotient of $\mathcal{L}(\alpha)$ with respect to a, with $a \in \Sigma$. For words $w \in \Sigma^\star$, the language denoted by $w^{-1}(\alpha)$ is tantamount to the language defined in (3.4). An important property of derivatives is their tight connection to word membership: in order to determine if a word $w \in \Sigma^\star$ is a member of the language denoted by a regular expression α it is enough to prove that

$$\varepsilon(w^{-1}(\alpha)) = \texttt{true}. \tag{3.23}$$

Symmetrically, we can conclude that

$$\varepsilon(w^{-1}(\alpha)) = \texttt{false} \tag{3.24}$$

implies that $w \notin \mathcal{L}(\alpha)$. The proofs of equations (3.23) and (3.24) are obtained by induction on the length of the word w and by some simple properties of the membership of the empty word. The next example, borrowed from Owens *et. al.* [80], shows the use of derivatives in pattern matching.

Example 5. *Let* $\Sigma = \{a, b\}$, $\alpha \stackrel{\text{def}}{=} ab^\star$, *and* $w = abb$. *The word* w *is accepted by the regular expression* α, *as shown by the following computation of Brzozowski's derivative:*

$$\begin{aligned}
(abb)^{-1}(\alpha) &= (abb)^{-1}(ab^\star) \\
&= b^{-1}((ab)^{-1}(ab^\star)) \\
&= b^{-1}(b^{-1}(a^{-1}(ab^\star))) \\
&= b^{-1}(b^{-1}(a^{-1}(a)b^\star)) \\
&= b^{-1}(b^{-1}(b^\star)) \\
&= b^{-1}(b^{-1}(b)b^\star) \\
&= b^{-1}(b^\star) \\
&= b^\star
\end{aligned}$$

Now, by testing the nullability of the resulting regular expression $b^\star$, *we obtain* $\varepsilon(b^\star) = \texttt{true}$. *Hence* $w \in \mathcal{L}(\alpha)$.

Similarly, it is also very easy to prove that a word does not belong to the language denoted by some regular expression.

Example 6. *Let* $\Sigma = \{a, b\}$, $\alpha \stackrel{\text{def}}{=} ab^\star$, *and* $w = aab$. *The word* w *is not accepted by the*

regular expression α*, since the computation of Brzozowski's derivative leads to*

$$\begin{aligned}(aab)^{-1}(\alpha) &= (aab)^{-1}(ab^\star)\\ &= b^{-1}((aa)^{-1}(ab^\star))\\ &= b^{-1}(a^{-1}(a^{-1}(ab^\star)))\\ &= b^{-1}(a^{-1}(a^{-1}(a)b^\star))\\ &= b^{-1}(a^{-1}(b^\star))\\ &= b^{-1}(a^{-1}(b)b^\star)\\ &= b^{-1}(0)\\ &= 0,\end{aligned}$$

and, by testing the nullability of the resulting regular expression 0*, we obtain* $\varepsilon(0) = \texttt{false}$*. Thus, by* (3.24)*,* $w \notin \mathcal{L}(\alpha)$*.*

In his seminal work, Brzozowski proved that the set of all derivatives of a regular expression α is finite when closed under the associativity, commutativity and idempotence of the operator $+$. The set $D(\alpha)$ of all the derivatives of α, modulo the previous properties, is the set defined by

$$D(\alpha) \stackrel{\text{def}}{=} \{\beta \mid \exists w \in \Sigma^\star,\ w^{-1}(\alpha) = \beta\}.$$

Example 7. *Let* $\Sigma = \{a, b\}$*, and let* $\alpha \stackrel{\text{def}}{=} ab^\star$*. The set* $D(\alpha)$*, of all the derivatives of* α*, is* $D(\alpha) = \{ab^\star, b^\star\}$*. A first round of derivation gives:* $\epsilon^{-1}(ab^\star) = ab^\star$*,* $a^{-1}(ab^\star) = b^\star$*, and* $b^{-1}(ab^\star) = 0$*. Next, we derive the values just calculated, obtaining* $a^{-1}(b^\star) = 0$ *and* $b^{-1}(b^\star) = b^\star$*, which renders the total of derivatives for* α*.*

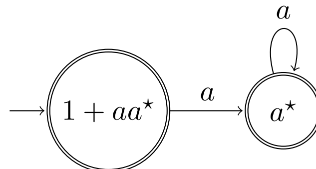
We can use $D(\alpha)$ to build a DFA that accepts the language denoted by α. The *derivative automaton* of α, denoted by $\mathcal{D}(\alpha)$ is the DFA defined as follows:

$$\mathcal{D}(\alpha) \stackrel{\text{def}}{=} (D(\alpha), \Sigma, \cdot^{-1}, \alpha, \{q \mid q \in D(\alpha),\ \varepsilon(q) = \texttt{true}\}).$$

Example 8. *Consider the regular expression* $\alpha = 1 + aa^\star$ *defined over the alphabet* $\Sigma = \{a\}$*. The corresponding derivative automaton is the DFA*

$$\mathcal{D}(\alpha) = (\{1 + aa^\star, a^\star\}, \Sigma, \cdot^{-1}, 1 + aa^\star, \{1 + aa^\star, a^\star\}),$$

represented by the following transition diagram:



Partial Derivatives

Partial derivatives were introduced by Antimirov [8] and are a generalisation of Brzozowski's derivatives to NFAs. In fact, this notion was first introduced by Mirkin [73], but only later was proved by Champarnaud and Ziadi [24] that both notions coincide. Let α be a regular expression and $a \in \Sigma$. The *set* $\partial_a(\alpha)$ *of partial derivatives* of the regular expression α with respect to a is inductively defined as follows:

$$\begin{aligned}
\partial_a(\emptyset) &\stackrel{\text{def}}{=} \emptyset,\\
\partial_a(\varepsilon) &\stackrel{\text{def}}{=} \emptyset,\\
\partial_a(b) &\stackrel{\text{def}}{=} \begin{cases} \{\varepsilon\} & \text{if } a \equiv b,\\ \emptyset & \text{otherwise,}\end{cases}\\
\partial_a(\alpha+\beta) &\stackrel{\text{def}}{=} \partial_a(\alpha) \cup \partial_a(\beta),\\
\partial_a(\alpha\beta) &\stackrel{\text{def}}{=} \begin{cases} \partial_a(\alpha)\beta \cup \partial_a(\beta) & \text{if } \varepsilon(\alpha) = \texttt{true},\\ \partial_a(\alpha)\beta & \text{otherwise,}\end{cases}\\
\partial_a(\alpha^\star) &\stackrel{\text{def}}{=} \partial_a(\alpha)\alpha^\star.
\end{aligned}$$

The operation of partial derivation is naturally extended to sets of regular expressions, as follows. Let S be a set of regular expressions and $a \in \Sigma$. The derivative with respect to a for the set S is defined by

$$\partial_a(S) \stackrel{\text{def}}{=} \bigcup_{\alpha \in S} \partial_a(\alpha).$$

Similarly to derivatives, the language of a partial derivative is the corresponding left-quotient

$$\mathcal{L}(\partial_a(\alpha)) = \mathcal{D}_a(\mathcal{L}(\alpha)). \tag{3.25}$$

Partial derivatives are implemented in Coq by the recursive function `pdrv` presented below. Lemma `pdrv_correct` proves the property (3.25). Finally, the extension of partial derivatives to finite sets of regular expressions is obtained in the expected way, as defined by `pdrv_set`.

```
Fixpoint pdrv (α:re)(a:A) :=
  match α with
  | 0 ⇒ ∅
  | 1 ⇒ ∅
  | ‘b ⇒ match _cmp a b with
         | Eq ⇒ {1}
         | _ ⇒ ∅
         end
  | x + y ⇒ (pdrv x s) ∪ (pdrv y s)
  | x · y ⇒ match ε(x) with
            | false ⇒ (pdrv x s) ⊙ y
```

```
              | true ⇒ (pdrv x s) ⊙ y ∪ (pdrv y s)
              end
  | x⋆ ⇒ (pdrv x s) ⊙ x⋆
end
Notation "∂_a(α)" := (pdrv α a).

Lemma pdrv_correct : ∀ α:re, ∀ a:A, ℒ(∂_a(α)) == LQ α a.

Definition pdrv_set(s:set re)(a:A) := fold (fun x:re ⇒ union (∂_a(x))) s ∅.
Notation "x %dS y" := (pdrv_set x y)(at level 60).
```

We now introduce partial derivatives with respect to words, which we denote by $\partial_w(\alpha)$, for $w \in \Sigma^\star$ and for a regular expression α. We overload the denotation of both the partial derivative with respect to a symbols, and of the partial derivative with respect to a word.

Given a regular expression α and a word $w \in \Sigma^\star$, the partial derivative $\partial_w(\alpha)$ of α with respect to w is defined inductively by

$$\begin{aligned} \partial_\varepsilon(\alpha) &\stackrel{\text{def}}{=} \{\alpha\}, \\ \partial_{wa}(\alpha) &\stackrel{\text{def}}{=} \partial_a(\partial_w(\alpha)). \end{aligned} \tag{3.26}$$

As with Brzozowski's, checking if a word belongs to $\mathcal{L}(\alpha)$ is syntactically obtained by applying nullability of sets to the set resulting from the derivation process, that is,

$$\varepsilon(\partial_w(\alpha)) = \texttt{true} \leftrightarrow w \in \mathcal{L}(\alpha), \tag{3.27}$$

$$\varepsilon(\partial_w(\alpha)) = \texttt{false} \leftrightarrow w \notin \mathcal{L}(\alpha). \tag{3.28}$$

In the examples that follow, we reconstruct the results we have provided as examples to Brzozowski's derivatives, but now using partial derivatives.

Example 9. *Let $\Sigma = \{a, b\}$, $\alpha \stackrel{\text{def}}{=} ab^\star$, and $w = abb$. The word derivative of α with respect to w corresponds to the following computation:*

$$\begin{aligned} \partial_{abb}(\alpha) &= \partial_{abb}(ab^\star) \\ &= \partial_b(\partial_{ab}(ab^\star)) \\ &= \partial_b(\partial_b(\partial_a(ab^\star))) \\ &= \partial_b(\partial_b(\partial_a(a)b^\star)) \\ &= \partial_b(\partial_b(\{b^\star\})) \\ &= \partial_b(\partial_b(b)b^\star) \\ &= \partial_b(\{b^\star\}) \\ &= \{b^\star\}. \end{aligned}$$

Now, by testing the nullability of the resulting set of regular expression $\{b^\star\}$*, we conclude that* $\varepsilon(b^\star) = \texttt{true}$*. Thus, by* (3.27)*,* $w \in \mathcal{L}(\alpha)$*.*

Example 10. *Let* $\Sigma = \{a, b\}$*,* $\alpha \stackrel{\text{def}}{=} ab^\star$*, and* $w = aab$*. The word derivative of* α *with respect to* w *corresponds to the following computation:*

$$\begin{aligned}
\partial_{aab}(\alpha) &= \partial_{aab}(ab^\star) \\
&= \partial_b(\partial_{aa}(ab^\star)) \\
&= \partial_b(\partial_a(\partial_a(ab^\star))) \\
&= \partial_b(\partial_a(\partial_a(a)b^\star)) \\
&= \partial_b(\partial_a(b^\star)) \\
&= \partial_b(\partial_a(b)b^\star) \\
&= \partial_b(\emptyset) \\
&= \emptyset,
\end{aligned}$$

and, by testing the nullability on the empty set resulting from the derivation, by (3.28)*, we conclude* $w \notin \mathcal{L}(\alpha)$*.*

The implementation in Coq of (3.26) is presented below. To ease its construction, we have defined the type `ilist` that represents lists defined from right to left.

```
Inductive ilist : Type :=
| inil : ilist
| icons : ilist → A → ilist.
Notation "<[]" := inil.
Infix "<::" := icons (at level 60, right associativity).
```

Therefore, the derivation with respect to words takes as arguments a regular expression and a word of type `ilist`. In this way, (3.26) is implemented as the following recursively defined function:

```
Reserved Notation "x %dw y" (at level 60).
Fixpoint pdrv_iw (s:set re)(w:ilist) : set re :=
  match w with
  | <[] ⇒ sre
  | xs <:: x ⇒ (s %dw xs) %dS x
end
where "x %dw y" := (pdrv_iw x y).

Definition wpdrv (r:re)(w:list A) :=
   pdrv_iw ({r}%set) (list_to_ilist w).
Notation "x %dW y" := (wpdrv x y)(at level 60).
```

```
Definition wpdrv_set (s:set re)(w:list A) :=
    s %dw (<@ w).
Notation "x %dWS y" := (wpdrv_set x y)(at level 60).
```

The definition `wpdrv_set` corresponds to the derivation of a set of regular expressions by a word, that is,

$$\partial_w(S) \stackrel{\text{def}}{=} \bigcup_{\alpha \in S} \partial_w(\alpha),$$

such that S is a set of regular expressions, and $w \in \Sigma^\star$. The function `wpdrv` is just a wrapper to `wpdrv_iw`, and its role is to transform a given word in its equivalent of type `ilist`.

Finally, we present the *set of partial derivatives* of a given regular expression α, which is defined by

$$PD(\alpha) \stackrel{\text{def}}{=} \bigcup_{w \in \Sigma^\star} (\partial_w(\alpha)).$$

Antimirov proved [8] that, given a regular expression α, the set $PD(\alpha)$ is always finite and its cardinality has an upper bound of $|\alpha|_\Sigma + 1$, *i.e.*, that $|PD(\alpha)| \leq |\alpha|_\Sigma + 1$.

Mirkin's Construction

Regular languages can be associated to a system of languages equations. Let N be a NFA such that $N = (Q, \Sigma, \delta, q_0, F)$, with $|Q| = n + 1$ and such that $Q = \{0, \ldots, n\}$ with $|\Sigma| = k$ and $n \in \mathbb{N}$. Also, let $q_0 = 0$ and let L_i be the language recognised by the automaton $(\{0, \ldots, n\}, \Sigma, \delta, i, F)$, for $i \in \{0, \ldots, n\}$, with $\mathcal{L}(N) = L_0$. Under the previous assumptions the following system of language equations

$$\begin{aligned} L_i &= \left(\bigcup_{j=1}^{k} \{a_j\} L_{ij}\right) \cup \begin{cases} \{\epsilon\} & \text{if } \epsilon \in L_i, \\ \emptyset & \text{otherwise,} \end{cases} \\ L_{ij} &= \bigcup_{m \in I_{ij}} L_m, \end{aligned} \tag{3.29}$$

is satisfied with $i \in \{0, \ldots, n\}$ and $I_{ij} = \delta(i, a_j) \subseteq \{0, \ldots, n\}$. Conversely any set of languages $\{L_0, \ldots, L_n\}$ that satisfies the set of equations (3.29) defines a NFA with initial state L_0. In particular if $L_0, \ldots, L_n$ are denoted by the regular expressions $\alpha_0, \ldots, \alpha_n$, respectively, then the following holds.

$$\begin{aligned} \alpha &\equiv \alpha_0, \\ \alpha_i &\sim a_1\alpha_{i1} + \ldots + a_k\alpha_{ik} + \begin{cases} 1 & \text{if } \varepsilon(\alpha) = \texttt{true}, \\ 0 & \text{otherwise,} \end{cases} \\ \alpha_{ij} &\sim \sum_{m \in I_{ij}} \alpha_m, \end{aligned} \tag{3.30}$$

with $i \in \{0, \ldots, n\}$ and $I_{ij} \subseteq \{0, \ldots, n\}$. Given a regular expression α, the task of finding a set of regular expressions that satisfies (3.30) is tantamount to finding a NFA whose recognised

language is $\mathcal{L}(\alpha)$. Mirkin [73] denoted by *support* any set of regular expressions that satisfies (3.30) and provided an algorithm for calculating it. He also proved that the size of such a solution is always upper bounded by $|\alpha|_\Sigma$. If S is a support for the regular expression α then the set $\{\alpha\} \cup S$ is a *prebase* of α.

Champarnaud and Ziadi [24] introduced an elegant way of calculating a support for a given regular expression α[2]. The function, denoted by $\pi(\alpha)$, is recursively defined as

$$\begin{array}{rclrcl} \pi(\emptyset) & \stackrel{\text{def}}{=} & \emptyset, & \pi(\alpha+\beta) & \stackrel{\text{def}}{=} & \pi(\alpha) \cup \pi(\beta), \\ \pi(\varepsilon) & \stackrel{\text{def}}{=} & \emptyset, & \pi(\alpha\beta) & \stackrel{\text{def}}{=} & \pi(\alpha)\beta \cup \pi(\beta), \\ \pi(\sigma) & \stackrel{\text{def}}{=} & \{\varepsilon\}, & \pi(\alpha^\star) & \stackrel{\text{def}}{=} & \pi(\alpha)\alpha^\star. \end{array} \tag{3.31}$$

Considering the previous definition, the set $\pi(\alpha) \cup \{\alpha\}$ forms a prebase of α. Champarnaud and Ziadi proved that prebases and the set of partial derivatives coincide, that is, $PD(\alpha) = \{\alpha\} \cup \pi(\alpha)$, from where we can conclude again that $|PD(\alpha)| \leq |\alpha|_\Sigma + 1$.

Example 11. *Let $\Sigma = \{a, b\}$, and let $\alpha \stackrel{\text{def}}{=} ab^\star$. The set $PD(\alpha)$ of all partial derivatives of α is calculated as follows:*

$$\begin{aligned} PD(\alpha) &= PD(ab^\star) \\ &= \{ab^\star\} \cup \pi(ab^\star) \\ &= \{ab^\star\} \cup \pi(a)b^\star \cup \pi(b^\star) \\ &= \{ab^\star\} \cup \{b^\star\} \cup \pi(b)b^\star \\ &= \{ab^\star, b^\star\} \cup \{b^\star\} \\ &= \{ab^\star, b^\star\}. \end{aligned}$$

The cardinality argument also holds, since

$$|PD(\alpha)| = |\{ab^\star, b^\star\}| = 2 \leq |ab^\star|_\Sigma + 1 = 3.$$

The definition in Coq of the system of equations (3.30), the function π and the theorem proving that the function π is a support are given below.

```
Inductive Support (α:re)(s:set re) : language :=
| mb_empty : ∀ w:word, w ∈ ε(α) → w ∈ (Support α s)
| mb : ∀ (w:word) (a:A), s != ∅ →
   (∃  β:re, ∃ s':set re,
      β ∈ s ∧ w ∈ L(aβ) ∧
      aβ ⊆ L(α) ∧ s' ⊆ s ∧
      β == L(s')) → w ∈ (Support α s).
```

[2]This definition was corrected by Broda *et al.* [21].

```
Fixpoint PI (α:re) : set re :=
 match α with
  | 0 ⇒ ∅
  | 1 ⇒ ∅
  | _ ⇒ {1}
  | x + y ⇒ (PI x) ∪ (PI y)
  | x · y ⇒ ((PI x)⊙y) ∪ (PI y)
  | x⋆ ⇒ (PI x)⊙(x⋆)
 end.
Notation "π(x)" := (PI x).

Theorem lang_eq_Support : ∀ α:re, α == Support α (π(α)).

Definition PD(α:re) := {α} ∪ π(α).
```

The proofs of the finiteness and of the upper bound of the set of partial derivatives is given by the lemmas that follow.

```
Theorem all_pdrv_in_PI : ∀ (α:re)(a:A), ∂a(α) ⊆ π(α).
Theorem PI_upper_bound : ∀ α:re, cardinal (π(α)) ≤ |α|Σ.

Lemma all_wpdrv_in_PD : ∀ (w:word)(αβ:re), β ∈ ∂w(α) → β ∈ PD(α).
Theorem PD_upper_bound : ∀ α:re, cardinal (PD(α)) ≤ |α|Σ + 1.
```

The proofs of these lemmas and theorems are all performed by induction on the structure of the given regular expressions and follow along the lines of the proofs originally conceived by the authors [8, 24].

Partial Derivatives and Regular Expression Equivalence

We now turn to the properties that allow us to use partial derivatives for deciding whether or not two given regular expressions are equivalent. A first property is that given a regular expression α, the equivalence

$$\alpha \sim \varepsilon(\alpha) \cup \bigcup_{a\in\Sigma} a\big(\sum \partial_a(\alpha)\big) \tag{3.32}$$

holds by induction on α and the properties of partial derivatives. We overload the notation $\varepsilon(\alpha)$ in the sense that in the current context $\varepsilon(\alpha) = \{1\}$ if α is nullable, and $\varepsilon(\alpha) = \emptyset$ otherwise. Following the equivalence (3.32), checking if $\alpha \sim \beta$ is tantamount to checking the equivalence

$$\varepsilon(\alpha) \cup \bigcup_{a\in\Sigma} a\big(\sum \partial_a(\alpha)\big) \sim \varepsilon(\beta) \cup \bigcup_{a\in\Sigma} a\big(\sum \partial_a(\beta)\big).$$

This will be an essential ingredient into our decision method because deciding if $\alpha \sim \beta$ resumes to checking if $\varepsilon(\alpha) = \varepsilon(\beta)$ and if $\partial_a(\alpha) \sim \partial_a(\beta)$, for each $a \in \Sigma$. Moreover, since partial derivatives are finite, and since testing if a word $w \in \Sigma^\star$ belongs to $\mathcal{L}(\alpha)$ is equivalent to checking syntactically that $\varepsilon(\partial_w(\alpha)) = \texttt{true}$, we obtain the following equivalence:

$$(\forall w \in \Sigma^\star, \varepsilon(\partial_w(\alpha)) = \varepsilon(\partial_w(\beta))) \leftrightarrow \alpha \sim \beta. \tag{3.33}$$

In the opposite direction, we can prove that α and β are not equivalent by showing that

$$\forall w \in \Sigma^\star,\ \varepsilon(\partial_w(\alpha)) \neq \varepsilon(\partial_w(\beta)) \rightarrow \alpha \not\sim \beta. \tag{3.34}$$

Equation (3.33) can be seen as an iterative process of testing regular expression equivalence by testing the equivalence of their derivatives. Equation (3.34) can be seen as the point where we find a counterexample of two derivatives during the same iterative process. In the next section we will describe a decision procedure that constructs a *bisimulation* that leads to equation (3.33) or that finds a counterexample like in (3.34) that shows that such a bisimulation does not exist.

3.3 A Procedure for Regular Expressions Equivalence

In this section we present the decision procedure EQUIVP for deciding regular expression equivalence, and describe its implementation in COQ. The base concepts for this mechanisation were already presented in the previous sections. The procedure EQUIVP follows along the lines of the work of Almeida *et. al.* [4], which has its roots in a rewrite system proposed by Antimirov and Mosses [9] to decide regular expression equivalence using Brzozowski's derivatives. In the next sections we describe the implementation choices that we have made in order to obtain a terminating, correct, and complete decision procedure that is also efficient from the point of view of determining if two regular expressions are equivalent using COQ's built-in computation. Although the current virtual machine of COQ has a performance that can be compared to the one of mainstream functional programming languages (namely with OCaml's bytecode compiled code), our procedure computes with terms that contain proofs (both in the finite set datatype used, and the specific representation of pairs of sets of regular expressions that we are about to present). Henceforth, the performances of our decision procedure within COQ's runtime are expected to be worst than those of a native implementation, or of the decision procedure that can be extracted from our development (in none of these we have the data representing proof terms).

3.3.1 The Procedure EQUIVP

Recall from the previous section that a proof of the equivalence of regular expressions can be obtained by an iterated process of checking the equivalence of their partial derivatives. Such

an iterated process is given in Algorithm 1 presented below. Given two regular expressions α and β the procedure EQUIVP corresponds to the iterated process of deciding the equivalence of their derivatives, in the way noted in equation (3.33). The procedure works over pairs of sets of regular expressions (S_α, S_β) such that $S_\alpha = \partial_w(\alpha)$ and $S_\beta = \partial_w(\beta)$, for some word $w \in \Sigma^\star$. From now on we will refer to these pairs of partial derivatives simply by derivatives.

Algorithm 1 The procedure EQUIVP.

Require: $S = \{(\{\alpha\}, \{\beta\})\}$, $H = \emptyset$
Ensure: true or false

```
 1: procedure EquivP(S, H)
 2:     while S ≠ ∅ do
 3:         (S_α, S_β) ← POP(S)
 4:         if ε(S_α) ≠ ε(S_β) then
 5:             return false
 6:         end if
 7:         H ← H ∪ {(S_α, S_β)}
 8:         for a ∈ Σ do
 9:             (S'_α, S'_β) ← ∂_a(S_α, S_β)
10:             if (S'_α, S'_β) ∉ H then
11:                 S ← S ∪ {(S'_α, S'_β)}
12:             end if
13:         end for
14:     end while
15: return true
16: end procedure
```

EQUIVP requires two arguments: a set H that serves as an accumulator for the derivatives (S_α, S_β) already processed; and a set S that serves as a working set that gathers new derivatives (S'_α, S'_β) yet to be processed. The set H ensures the termination of EQUIVP due to the finiteness of the set of partial derivatives. The set S has no influence in the termination argument. When EQUIVP terminates, then it must do so in one of two possible configurations: either the set H contains all the derivatives of α and β and all of them are equi-nullable; or a counterexample (S_α, S_β) such that $\varepsilon(S_\alpha) \neq \varepsilon(S_\beta)$ is found. By equation (3.33), we conclude that we have $\alpha \sim \beta$ in the first case, whereas in the second case we must conclude that $\alpha \not\sim \beta$.

This procedure follows along the lines of the work of Almeida *et. al.* [4, 3, 5], where the authors propose a functional algorithm that decides regular expressions equivalence based on partial derivatives. This procedure is a functional formulation of the rewrite system proposed by Antimirov and Mosses [9], but the goal of the latter is to decide regular expression

equivalence using Brzozowski's derivatives. The main difference between EQUIVP and the one proposed by the cited authors relies on the underlying representation of derivatives: their algorithm uses a notion of linearisation of regular expressions that includes derivation and a notion of linear regular expression; we simply use finite sets of regular expressions and the derivation previously introduced. We will get back to this and to other diferences in Section 3.5. To finish the current section we give below two examples that illustrate how EQUIVP computes. The first example considers the equivalence of regular expressions and the second one considers inequivalence.

Example 12. *Suppose we want to prove that* $\alpha = (ab)^\star a$ *and* $\beta = a(ba)^\star$ *are equivalent. Considering* $s_0 = (\{(ab)^\star a\}, \{a(ba)^\star\})$*, it is enough to show that*

$$\text{EQUIVP}(\{s_0\}, \emptyset) = \texttt{true}.$$

The computation of EQUIVP *for these particular* α *and* β *involves the construction of the new derivatives* $s_1 = (\{1, b(ab)^\star a\}, \{(ba)^\star\})$ *and* $s_2 = (\emptyset, \emptyset)$*. We can trace the computation by the following table*

i	S_i	H_i	drvs.
0	$\{s_0\}$	$\emptyset$	$\partial_a(s_0) = s_1, \partial_b(s_0) = s_2$
1	$\{s_1, s_2\}$	$\{s_0\}$	$\partial_a(s_1) = s_2, \partial_b(s_1) = s_0$
2	$\{s_2\}$	$\{s_0, s_1\}$	$\partial_a(s_2) = s_2, \partial_b(s_2) = s_2$
3	$\emptyset$	$\{s_0, s_1, s_2\}$	`true`

where i *is the iteration number, and* S_i *and* H_i *are the arguments of* EQUIVP *in that same iteration. The trace terminates with* $S_2 = \emptyset$ *and thus we can conclude that* $\alpha \sim \beta$*.*

Example 13. *Suppose we want to check that* $\alpha = b^\star a$ *and* $\beta = b^\star ba$ *are not equivalent. Considering* $s_0 = (\{b^\star a\}, \{b^\star ba\})$*, to prove so it is enough to check if*

$$\text{EQUIVP}(\{s_0\}, \emptyset) = \texttt{false}.$$

In this case, the computation of EQUIVP *creates the new derivatives ,* $s_1 = (\{1\}, \emptyset)$ *and* $s_2 = (\{b^\star a\}, \{a, b^\star ba\})$*, and takes two iterations to halt and return* `false`*. The counter example found is the pair* s_1*, as it is easy to see in the trace of computation presented in the table below.*

i	S_i	H_i	drvs.
0	$\{s_0\}$	$\emptyset$	$\partial_a(s_0) = s_1, \partial_b(s_0) = s_2$
1	$\{s_1, s_2\}$	$\{s_0\}$	`false`

3.3.2 Implementation

Representation of Derivatives

The main data type used in EQUIVP is the type of pairs of sets of regular expressions. Each pair (S_α, S_β) represents a word derivative $(\partial_w(\alpha), \partial_w(\beta))$, where $w \in \Sigma^\star$. The type of derivatives is given by the definition of `Drv` as follows:

```
Record Drv (α β:re) := mkDrv {
 dp :> set re * set re ;
 w : word ;
 cw : dp = (∂w(α),∂w(β))
}.
```

The type `Drv` is a *dependent record* composed of three parameters: a pair of sets of regular expressions `dp` that corresponds to the actual pair (S_α, S_β); a word `w`; a proof term `cw` that ensures that $(S_\alpha, S_\beta) = (\partial_{\mathtt{w}}(\alpha), \partial_{\mathtt{w}}(\beta))$. The use of the type `Drv` instead of a pair of sets of regular expressions is necessary because EQUIVP's domain is the set of pairs resulting from derivations and not arbitrary pairs of sets of regular expressions on Σ. The next example shows how to construct a value of type `Drv` representing the derivative of the original regular expressions α and β, by the empty word ϵ.

Example 14. *The function* `Drv_1st` *that returns the value of type* `Drv` α β*, and which represents the pair* $(\{\alpha\}, \{\beta\})$ *or, equivalently, the pair* $(\partial_\epsilon(\alpha), \partial_\epsilon(\beta))$ *is defined in our development as follows:*

```
Program Definition Drv_1st (α β:re) : Drv α β.
refine(mkDrv ({α},{β}) ϵ _).
abstract(reflexivity).
Defined.
```

The equality relation defined over `Drv` terms considers only the projection `dp`, that is, two terms `d1` and `d2` of type `Drv` α β are equal if `(dp d1)` $=$ `(dp d2)`. This implies that each derivative will be considered only once along the execution of EQUIVP. If the derivative `d1` is already in the accumulator set, then all derivatives `d2` that are computed afterwards will fail the membership test of line 10 of Algorithm 1. This directly implies the impossibility of the eventual non-terminating computations due to the repetition of derivatives.

As a final remark, the type `Drv` also provides a straightforward way to relate the result of the computation of EQUIVP to the (in-)equivalence of α and β: on one hand, if H is the set returned by EQUIVP, then checking the nullability of its elements is tantamount to proving the equivalence of the corresponding regular expressions, since we expect H to contain all the derivatives; on the other hand, if EQUIVP returns a term `t:Drv` α β, then ε(`t`) $=$ `false`,

which implies that the word `(w t)` is a witness of inequivalence, and can be presented to the user.

Extended Derivation and Nullability

The definition of derivative with respect to a symbol and with respect to a word are also extended to the type `Drv`. The derivation of a value of type `Drv` $\alpha\,\beta$ representing the pair (S_α, S_β) is obtained by calculating the derivative $\partial_a(S_\alpha, S_\beta)$, updating the word w, and also by automatically building the associated proof term for the parameter `cw`. The function implementing the derivation of `Drv` terms, and its extension to sets of `Drv` terms, and to the derivation with respect to a word, are given below. They are the definitions `Drv_pdrv`, `Drv_pdrv_set`, and `Drv_wpdrv`, respectively. Note that $\partial_a(S_\alpha, S_\beta) \stackrel{\text{def}}{=} (\partial_a(S_\alpha), \partial_a(S_\beta))$, and therefore $\partial_a(\partial_w(\alpha), \partial_w(\beta)) = (\partial_{wa}(\alpha), \partial_{wa}(\beta))$.

```
Definition Drv_pdrv(α β:re)(x:Drv α β)(a:A) : Drv α β.
refine(match x with
      | mkDrv α β K w P ⇒ mkDrv α β (pdrvp K a) (w++[a]) _
      end).
abstract((* Proof that ∂a(∂w(α),∂w(β)) = (∂wa(α),∂wa(β)) *)).
Defined.

Definition Drv_pdrv_set(x:Drv α β)(s:set A) : set (Drv α β) :=
 fold (fun y:A ⇒ add (Drv_pdrv x y)) s ∅.

Definition Drv_wpdrv (α β:re)(w:word) : Drv α β.
refine(mkDrv α β (∂w(α),∂w(β)) w _).
abstract((* Proof that (∂w(α),∂w(β)) = (∂w(α),∂w(β)) *)).
Defined.
```

We also extend the notion of nullable regular expression to terms of type `Drv`, and to sets of values of type `Drv`. Checking the nullability of a `Drv` term denoting the pair (S_α, S_β) is tantamount at checking that $\varepsilon(S_\alpha) = \varepsilon(S_\beta)$.

```
Definition c_of_rep(x:set re * set re) :=
 Bool.eqb (c_of_re_set (fst x)) (c_of_re_set (snd x)).

Definition c_of_Drv(x:Drv α β) := c_of_rep (dp x).

Definition c_of_Drv_set (s:set (Drv α β)) : bool :=
 fold (fun x ⇒ andb (c_of_Drv x)) s true.
```

All the previous functions were implemented using the proof mode of Coq instead of trying a direct definition. In this way we are able to wrap the proofs in the tactic `abstract`, which dramatically improves the performance of the computation.

Computation of New Derivatives

The *while-loop* of equivP – lines 2 to 14 of Algorithm 1 – describes the process of testing the equivalence of the derivatives of two given regular expressions α and β. In each iteration, either a witness of inequivalence is found, or new derivatives (S_α, S_β) are computed and the sets S and H are updated accordingly. The expected behaviour of each iteration of the loop is implemented by the function `step`, presented below, which also corresponds to the *for-loop* from lines 8 to 13 of Algorithm 1.

```
Definition step (H S:set (Drv α β))(Σ:set A) :
 ((set (Drv α β) * set (Drv α β)) * step_case α β) :=
 match choose S with
 |None ⇒ ((H,S),termtrue α β H)
 |Some (Sα,Sβ) ⇒
   if c_of_Drv _ _ (Sα,Sβ) then
    let H' := add (Sα,Sβ) H in
     let S' := remove (Sα,Sβ) S in
      let ns := Drv_pdrv_set_filtered α β (Sα,Sβ) H' Σ in
       ((H',ns ∪ S'),proceed α β)
   else
    ((H,S),termfalse α β (Sα,Sβ))
 end.
```

The `step` function proceeds as follows: it obtains a pair (S_α, S_β) from the set S, and tests it for equi-nullability. If S_α and S_β are not equi-nullable, then `step` returns a pair $((H,S)$,`termfalse` $\alpha\ \beta\ (S_\alpha, S_\beta))$ that serves as a witness for $\alpha \not\sim \beta$. If, on the contrary, S_α and S_β are equi-nullable, then `step` generates new derivatives (S'_α, S'_β) such that $(S'_\alpha, S'_\beta) = (\partial_a(S_\alpha), \partial_a(S_\beta))$, with $a \in \Sigma$ and (S'_α, S'_β) is not a member of $\{(S_\alpha, S_\beta)\} \cup H$. These new derivatives are added to S and (S_α, S_β) is added to H. The computation of new derivatives is performed by the function `Drv_pdrv_set_filtered`, defined as follows.

```
Definition Drv_pdrv_set_filtered(x:Drv α β)(H:set (Drv α β))
 (sig:set A) : set (Drv α β) :=
  filter (fun y ⇒ negb (y ∈ H)) (Drv_pdrv_set x sig).
```

Note that this is precisely what prevents the whole process from entering potential infinite loops, since each derivative is considered only once during the execution of equivP and because the number of derivatives is always finite.

Finally, we present the type `step_case`. This type is built from three constructors: the constructor `proceed` represents the fact that there is not yet information that allows to decide if the regular expressions under consideration are equivalent or not; the constructor `termtrue` indicates that no more elements exist in S and that H should contain all the derivatives; finally, the constructor `termfalse` indicates that `step` has found a proof of the inequivalence of the regular expressions under consideration.

```
Inductive step_case (α β:re) : Type :=
|proceed : step_case α β
|termtrue : set (Drv α β) → step_case α β
|termfalse : Drv α β → step_case α β.
```

Termination

Clearly, the procedure EQUIVP is general recursive rather than structural recursive. This implies that the procedure's iterative process cannot be directly encoded in COQ's type system. Therefore, we have devised a well-founded relation establishing a recursive measure that defines the course-of-values that makes EQUIVP terminate. This well-founded relation will be the structural recursive argument for our encoding of EQUIVP, using the methodology introduced in Section 2.1.4.

The decreasing measure (of the recursive calls) for EQUIVP is defined as follows: in each recursive call, the cardinality of the accumulator set H increases by one unit due to the computation of `step`. The maximum size that H can reach is upper bounded by

$$2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1$$

due to the upper bounds of the cardinalities of both $PD(\alpha)$ and $PD(\beta)$, the cardinality of the cartesian product, and the cardinality of the powerset. Therefore, if

$$\texttt{step}\, H\ S\ _ = (H', _, _),$$

then the following relation

$$(2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1) - |H'| < (2^{(|\alpha|_\Sigma+1)} \times 2^{(|\beta|_\Sigma+1)} + 1) - |H|, \tag{3.35}$$

holds. In terms of its implementation in COQ, we first define and prove the following:

```
Definition lim_cardN (z:N) : relation (set A) :=
  fun x y:set A ⇒ nat_of_N z - (cardinal x) < nat_of_N z - (cardinal y).
Lemma lim_cardN_wf : ∀ z, well_founded (lim_cardN z).
```

The relation `lim_cardN` is a generic version of (3.35). It is proved by first showing that

$$\forall (z,\, n,\, a : \texttt{nat}),\ (z - |a|) < n \to \texttt{Acc}\ (\texttt{lim_card}\ z)\ a, \tag{3.36}$$

such that Acc is the definition, in Coq's type theory, of the notion of accessibility. The relation lim_card is the same as relation lim_cardN, but the former uses Peano natural numbers whereas the later uses the type N. The type N is the binary representation of natural numbers and we use it as the main datatype here because it is more efficient when compared to nat. The proof of (3.36) follows from straightforward induction on n.

Next, we establish the upper bound of the number of derivatives, and define the relation LLim that is the relation that actually implements (3.35). The encoding in Coq goes as follows:

```
Definition MAX_RE(α:re) := |α|_Σ + 1.
Definition MAX(α β:re) := (2^MAX_RE(α) × 2^MAX_RE(β)) + 1.
Definition LLim(α β:re) := lim_cardN (Drv α β) (MAX α β).

Theorem LLim_wf(α β:re) : well_founded (LLim α β).
```

We note that there is still a missing piece in order for LLim to be used as the recursive argument for the main function: the sets S and H of EQUIVP must be proved to remain disjoint along the execution; otherwise, there is no guarantee that H actually increases in each recursive call, making LLim not adequate for its purpose. However, the proof of this property will be needed only when we implement EQUIVP, as we shall see very soon.

The Iterator

We now present the development of a recursive function that implements the main loop of Algorithm 1. This recursive function is an iterator that calls the function step a finite number of times starting with two initial sets S and H. This iterator, named iterate, is defined as follows:

```
Function iterate(α β:re)(H S:set (Drv α β))(sig:set A)(D:DP α β H S)
 {wf (LLim α β) H}: term_cases α β :=
  let ((H',S',next) := step H S in
   match next with
    |termfalse x ⇒ NotOk α β x
    |termtrue h ⇒ Ok α β h
    |proceed ⇒ iterate α β H' S' sig (DP_upd α β H S sig D)
  end.
Proof.
 abstract(apply DP_wf).
 exact(guard α β 100 (LLim_wf α β)).
Defined.
```

The function `iterate` is recursively decreasing on a proof term that `LLim` is well-founded. The type annotation `wf LLim` α β adds this information to the inner mechanisms of `Function`, so that `iterate` is constructed in such a way that Coq's type-checker accepts it. Moreover, in order to validate `LLim` along the computation of `iterate`, we must provide evidence that the sets S and H remain disjoint in all the recursive calls of `iterate`. The last parameter of the definition of `iterate`, D, has the type `DP` which packs together a proof that the sets H and S are disjoint (in all recursive calls) and that all the elements in the set H are equi-nullable. The definition of type `DP` is

```
Inductive DP (α β:re)(H S: set (Drv α β)) : Prop :=
| is_dp : H ∩ S = ∅ → c_of_Drv_set α β H = true → DP α β H S.
```

In the definition of the recursive branch of `iterate`, the function `DP_upd` is used to build a new term of type `DP` that proves that the updated sets H and S remain disjoint, and that all the elements in H remain equi-nullable.

```
Lemma DP_upd : ∀ (α β:re)(H S : set (Drv α β)) (sig : set A),
  DP α β H S →
  DP α β (fst (fst (step α β H S sig))) (snd (fst (step α β H S sig))).
```

The output of `iterate` is a value of type `term_cases`, which is defined as follows:

```
Inductive term_cases (α β:re) : Type :=
|Ok : set (Drv α β) → term_cases α β
|NotOk : Drv α β → term_cases α β.
```

The type `term_cases` is made of two constructors that determine what possible outcome we can obtain from computing `iterate`: either it returns a set S of derivatives, packed in the constructor `Ok`, or it returns a sole pair (S_α, S_β), packed in the constructor `NotOk`. The first should be used to prove equivalence, whereas the second should be used for exhibiting a witness of inequivalence.

As explained in Section 2.1.4, the `Function` vernacular produces proof obligations that must be discharged in order to be accepted by Coq's type checker. One of the proof obligations generated by `iterate` is that, when performing a recursive call, the new cardinalities of H and S still satisfy the underlying well-founded relation. The lemma `DP_wf` serves this purpose and is defined as follows:

```
Lemma DP_wf : ∀ (α β:re)(H S : set (Drv α β)) (sig : set A),
  DP α β H S → snd (step α β H S sig) = proceed α β →
  LLim α β (fst (fst (step α β H S sig))) H.
```

The second proof obligation generated by the `Function` command is just the exact accessibility term that allows for `iterate` to compute efficiently, as described in Section 2.1.4.

In the code below we give the complete definition of EQUIVP. The function `equivP` is simply a wrapper defined over `iterate`: it establishes the correct input for the arguments H and S and pattern matches over the result of `iterate`, returning the expected Boolean value.

```
Definition equivP_aux(α β:re)(H S:set(Drv α β))
                        (sig:set A)(D:DP α β H S):=
 let H' := iterate α β H S sig D in
  match H' with
  | Ok _ ⇒ true
  | NotOk _ ⇒ false
  end.

Definition mkDP_ini : DP α β ∅ {Drv_1st α β}.
abstract(constructor;[split;intros;try(inversion H)|vm_compute];
           reflexivity).
Defined.

Definition equivP (α β:re) :=
 equivP_aux α β ∅ {Drv_1st α β} (setSy α ∪ setSy β) (mkDP_ini α β).
```

The function `mkDP_ini` ensures that the inital sets S and H are disjoint, with $H = \emptyset$, and also ensures that $\varepsilon(\emptyset) =$ `false` holds. The final decision procedure, `equivP`, calls the function `equivP_aux` with the adequate arguments, and the function `equivP_aux` simply pattern matches over a term of `term_cases` and returns a Boolean value accordingly.

We note that in the definition of `equivP` we instantiate the parameter representing the input alphabet by the union of two sets, both computed by the function `setSy`. This function returns the set of all symbols that exist in a given regular expression. It turns out that for deciding regular expressions (in)equivalence we need not to consider a fixed alphabet Σ, since only the symbols that exist in the regular expressions being tested are important and used in the derivations. In fact, the input alphabet can even be an infinite alphabet.

3.3.3 Correctness and Completeness

We now give the proofs of the soundness and of the completeness of EQUIVP with respect to language equivalence. The correctness and completeness of `equiv` with respect to language (in)equivalence follows from proving the following two statements: if `equivP` α β returns `true`, then $\alpha \sim \beta$; otherwise, `equivP` α β implies $\alpha \not\sim \beta$. If we unfold the definitions of `equivP` and `equivP_aux`, the previous statements can be rephrased in terms of the function `iterate`. Thus, and considering the alphabet $\Sigma =$ (`setSy` $\alpha \cup$ `setSy` β), and the value

$D = (\texttt{DP_1st}\ \alpha\ \beta)$, if it holds

$$\texttt{iterate}\ \alpha\ \beta\ \emptyset\ (\texttt{Drv_1st}\ \alpha\ \beta)\ \Sigma\ D = \texttt{Ok}\ \alpha\ \beta\ H,$$

then $\alpha \sim \beta$ must be true. The returned value H is a set $\texttt{Drv}\ \alpha\ \beta$. On the contrary, if we obtain

$$\texttt{iterate}\ \alpha\ \beta\ \emptyset\ (\texttt{Drv_1st}\ \alpha\ \beta)\ \Sigma\ D = \texttt{NotOk}\ \alpha\ \beta\ \gamma, \tag{3.37}$$

then $\alpha \not\sim \beta$ must hold, with γ of type $\texttt{Drv}\ \alpha\ \beta$. In what follows, we describe how the previous two statements were proved.

Correctness. In order to prove the correctness of `equivP` with respect to language equivalence, we proceed as follows. Suppose that $\texttt{equivP}\ \alpha\ \beta = \texttt{true}$. To prove that this implies regular expression equivalence we must prove that the set of all the derivatives is computed by the function `iterate`, and also that all the elements of that set are equi-nullable. This leads to (3.33), which in turn implies its language equivalence.

To prove that `iterate` computes the desired set of derivatives we must show that, in each of its recursive calls, the accumulator set H keeps a set of values whose derivatives have been already computed (they are also in H), or that such derivatives are still in the working set S, waiting to be selected for further processing. This property is formally defined in Coq as follows:

```
Definition invP (α β:re)(H S:set (Drv α β))(Σ:set A) :=
 ∀ x:Drv α β, x ∈ H → ∀ a:A, a ∈ Σ → (Drv_pdrv α β x a) ∈ (H ∪ S).
```

Now, we must prove that `invP` is an invariant of `iterate`. For that, we must first prove that `invP` is satisfied by the computation of `step`.

Proposition 1. *Let α and β be two regular expressions. Let S, S', H, and H' be finite sets of values of type $\texttt{Drv}\ \alpha\ \beta$. Moreover, let Σ be an alphabet. If $\texttt{invP}\ H\ S$ holds and if*

$$\texttt{step}\ \alpha\ \beta\ H\ S\ \Sigma = ((H',S'),\texttt{proceed}\ \alpha\ \beta),$$

then $\texttt{invP}\ H'\ S'$ also holds.

The next step is to prove that `invP` is an invariant of `iterate`. This proof indeed shows that if `invP` is satisfied in all the recursive calls of `iterate`, then this function must return a value $\texttt{Ok}\ \alpha\ \beta\ H'$ and $\texttt{invP}\ H'\ \emptyset$ must be satisfied, as stated by the lemma that follows.

Proposition 2. *Let α and β be two regular expressions. Let S, H, and H' be finite sets of values of type $\texttt{Drv}\ \alpha\ \beta$, and let Σ be an alphabet. If $\texttt{invP}\ H\ S$ holds, and if*

$$\texttt{iterate}\ \alpha\ \beta\ H\ S\ \Sigma\ D = \texttt{Ok}\ \alpha\ \beta\ H',$$

then $\texttt{invP}\ H'\ \emptyset$ also holds.

In Coq, the two previous propositions are defined as follows:

```
Lemma invP_step : ∀ α β H S Σ,
  invP α β H S Σ → invP α β (fst (fst (step α β H S Σ)))
                    (snd (fst (step α β H S Σ))) Σ.
```

```
Lemma invP_iterate : ∀ α β H S Σ D x,
  invP α β H S Σ →
    iterate α β H S Σ D = Ok α β x →
     invP α β x ∅.
```

Proposition 1 and Proposition 2 are still not enough to prove the correctness of `equivP` with respect to language equivalence. We must prove that these derivatives are all equi-nullable, and that the pair containing the regular expressions being tested for equivalence are in the set of derivatives returned by `iterate`. For that, we strengthen the invariant `invP` with this property, as follows:

```
Definition invP_final(α β:re)(H S:set (Drv α β))(s:set A) :=
  (Drv_1st α β) ∈ (H ∪ S) ∧
   (∀  x:Drv α β, x ∈ (H ∪ S) → c_of_Drv α β x = true) ∧
     invP α β H S s.
```

We start by proving that, if we are testing $\alpha \sim \beta$, then the pair $\{(\{\alpha\},\{\beta\})\}$ is an element of the set returned by `iterate`. But first we must introduce two generic properties that will allow us to conclude that.

Proposition 3. *Let α and β be two regular expressions. Let H, H', and S' be sets of values of type* `Drv` *α β. Finally, let Σ be an alphabet, and let D be a value of type* `DP` *$\alpha\ \beta\ H\ S$. If it holds that* `iterate` *$\alpha\ \beta\ H\ S\ \Sigma\ D =$* `Ok` *$\alpha\ \beta\ H'$, then it also holds that $H \subseteq H'$.*

Corollary 1. *Let α and β be two regular expressions. Let γ be a value of type* `Drv` *α β. Let H, H', and S' be sets of values of type* `Drv` *α β. Finally, let Σ be an alphabet, and let D be a value of type* `DP` *$\alpha\ \beta\ H\ S$. If it holds that* `iterate` *$\alpha\ \beta\ H\ S\ \Sigma\ D =$* `Ok` *$\alpha\ \beta\ H'$ and that* `choose` *$S =$* `Some` *γ, then it also holds that $\{\gamma\} \cup H \subseteq H'$.*

From Proposition 3 and Corollary 1 we are able to prove that the original pair is always returned by the `iterate` function, whenever it returns a value `Ok` $\alpha\ \beta\ H$.

Proposition 4. *Let α and β be two regular expressions, let H' be a finite set of values of type* `Drv` *α β, let Σ be an alphabet, and let D be a value of type* `DP` *$\alpha\ \beta\ \emptyset\ \{(\{\alpha\},\{\beta\})\}$. Hence,*

$$\texttt{iterate}\ \alpha\ \beta\ \emptyset\ \{(\{\alpha\},\{\beta\})\}\ \Sigma\ D = \texttt{Ok}\ \alpha\ \beta\ H' \rightarrow (\{\alpha\},\{\beta\}) \in H'.$$

Now, we proceed in the proof by showing that all the elements of the set packed in a value `Ok` α β H' enjoy equi-nullability. This is straightforward, due to the last parameter of `iterate`. Recall that a value of type `DP` always contains a proof of that fact.

Proposition 5. *Let α and β be two regular expressions. Let H, H', and S' be sets of values of type `Drv` α β. Finally, let Σ be an alphabet and D be a value of type `DP` α β H S. If it holds that `iterate` α β H S Σ D = `Ok` α β H', then it also holds that $\forall \gamma \in H'$, $\varepsilon(\gamma) =$ `true`.*

Using Propositions 4 and 5 we can establish the intermediate result that will take us to prove the correctness of `equivP` with respect to language equivalence.

Proposition 6. *Let α and β be two regular expressions. Let H, H', and S' be sets of values of type `Drv` α β. Finally, let Σ be an alphabet, and let D be a value of type `DP` α β H S. If it holds that `iterate` α β H S Σ D = `Ok` α β H', then `invP_final` α β H' $\emptyset$.*

The last intermediate logical condition that we need to establish to finish the proof of correctness is that `invP_final` implies language equivalence, when instantiated with the correct parameters. The following lemma gives exactly that.

Proposition 7. *Let α and β be two regular expressions. Let H' be a set of values of type `Drv` α β. If it holds that `invP_final` α β H' $\emptyset$ (`setSy` α $\cup$ `setSy` β), then it is true that $\alpha \sim \beta$.*

Finally, we can state the theorem that ensures that if `equivP` returns `true`, then we have the equivalence of the regular expressions under consideration.

Lemma 1. *Let α and β be two regular expressions. Thus, if `equivP` α β = `true`, then $\alpha \sim \beta$.*

Completeness. To prove that (3.37) implies non equivalence of two given regular expressions α and β, we must prove that the value γ in the returned term `NotOk` α β γ is a witness that there is a word $w \in \Sigma^\star$ such that $w \in \mathcal{L}(\alpha)$ and $w \notin \mathcal{L}(\beta)$, or the other way around. This leads us to the following lemma about `iterate`.

Proposition 8. *Let α and β be regular expressions, let S and H be sets of values of type `Drv` α β. Let Σ be an alphabet, γ a term of type `Drv`, and D a value of type `DP` α β S H. If*

$$\texttt{iterate}\ \alpha\ \beta\ S\ H\ \Sigma\ D = \texttt{NotOk}\ \alpha\ \beta\ \gamma,$$

then, considering that γ represents the pair of sets of regular expressions (S_α, S_β), we have $\varepsilon(S_\alpha) \neq \varepsilon(S_\beta)$.

Next, we just need to prove that the pair in the value returned by `iterate` does imply inequivalence.

Proposition 9. *Let α and β be regular expressions, let S and H be sets of values of type* $\mathtt{Drv}\ \alpha\ \beta$*, let Σ be an alphabet, and let D be a value of type* $\mathtt{DP}\ \alpha\ \beta\ S\ H$*. Hence,*

$$\mathtt{iterate}\ \alpha\ \beta\ S\ H\ \Sigma\ D = \mathtt{NotOk}\ \alpha\ \beta\ \gamma \to \alpha \not\sim \beta.$$

The previous two lemmas allow us to conclude that `equivP` is correct with respect to the inequivalence of regular expressions.

Lemma 2. *Let α and β be two regular expressions. Hence,*

$$\mathtt{equivP}\ \alpha\ \beta = \mathtt{false} \ \to\ \alpha \not\sim \beta$$

holds.

3.3.4 Tactics and Automation

In this section we describe two Coq proof tactics that are able to automatically prove the (in)equivalence of regular expressions, as well as relational algebra equations.

Tactic for Deciding Regular Expressions Equivalence

The expected way to prove the equivalence of two regular expressions α and β, using our development, can be summarised as follows: first we look into the goal, which must be of the form $\alpha \sim \beta$ or $\alpha \not\sim \beta$; secondly, we transform such goal into the equivalent one that is formulated using `equivP`, on which we can perform computation. The Coq code for this tactic is

```
Ltac re_inequiv :=
  apply equiv_re_false;vm_compute;
  first [ reflexivity | fail 2 "Regular expressions are not inequivalent" ].

Ltac re_equiv :=
  apply equiv_re_true;vm_compute;
  first [ reflexivity | fail 2 "Regular expressions are not equivalent" ].

Ltac dec_re :=
  match goal with
   | |- re2rel (?R1) ∼ re2rel (?R2) ⇒ re_equiv
   | |- re2rel (?R1) !∼ re2rel (?R2) ⇒ re_inequiv
   | |- re2rel (?R1) ≤ re2rel (?R2) ⇒
        unfold lleq;change (R1 ∪ R2) with (re2rel (R1 + R2));
          re_equiv
```

```
  | |- _ ⇒ fail 2 "Not a regular expression (in)equivalence equation."
 end.
```

The main tactic, `dec_re`, pattern matches on the goal and decides whether the goal is an equivalence, an inequivalence, or a containment relation. In the former two cases, `dec_re` applies the corresponding auxiliary tactics, `re_inequiv` or `re_equiv`, and reduces the (in)equivalence into a call to `equivP`, and then performs computation in order to try to solve the goal by reflexivity. In case the goal represents a containment relation, `dec_re` first changes it into an equivalence (since we know that $\alpha \leq \beta \stackrel{\text{def}}{=} \alpha + \beta \sim \beta$) and, after that, call the auxiliary tactic `re_equiv` to prove the goal.

The tactic `dec_re` is straightforward, and in the next section we will comment on its efficiency within Coq's environment.

Tactic for Deciding Relation Algebra Equations

One application of our development is the automation of proof of equations of relational algebra. The idea of using regular expression equivalence to decide relation equations, based on derivatives, was first pointed out by Nipkow and Krauss [66]. In this section we adapt their ideia to our development by providing a reflexive tactic that reduces relation equivalence to regular expressions equivalence, and then apply the tactic for deciding regular expressions equivalence to finish the proof.

A formalisation of relations is already provided in Coq's standard library, but no support for automation is given. Here, we consider the following encoding of binary relations:

```
Parameter B : Type.

Definition EmpRel : relation B := fun _ _:B ⇒ False.

Definition IdRel : relation B := fun x y :B ⇒ x = y.

Definition UnionRel (x y:relation B) : relation B := union _ x y.

Definition CompRel (R S: relation B): relation B :=
   fun i k ⇒ ∃ j, R i j ∧ S j k.

Inductive TransRefl (R:relation B) : relation B :=
| trr : ∀ x, TransRefl R x x
| trt : ∀ x y, R x y → ∀ z, TransRefl R y z → TransRefl R x z.

Definition rel_eq (R S:relation B) : Prop :=
```

```
same_relation B R S.
```

The definitions `EmpRel`, `IdRel`, `UnionRel`, `CompRel` and `TransRefl` represent, respectively, the empty relation, the identity relation, the union of relations, the composition of relations, and the transitive and reflexive closure of a relation. If R_1 and R_2 are relations, we denote the previous constants and operations on relations by $\emptyset$, $\mathcal{I}$, $R_1 \cup R_2$, $R_1 \circ R_2$, and $R^\star$, respectively. The definition `rel_eq` corresponds to the equivalence of relations, and is denoted by $R_1 \sim_{\mathcal{R}} R_2$.

Regular expressions can be easily transformed into binary relations. For this, we need to consider a function v that maps each symbol of the alphabet under consideration into a relation. With this function, we can define another recursive function that, by structural recursion on a given regular expression α, computes the corresponding relation. Such a function is defined in Coq as follows:

```
Fixpoint reRel(v:nat→ relation B)(α:re) : relation B :=
 match α with
 | 0 ⇒ EmpRel
 | 1 ⇒ IdRel
 | ‘a ⇒ v a
 | x + y ⇒ UnionRel (reRel v x) (reRel v y)
 | x · y ⇒ CompRel (reRel v x) (reRel v y)
 | x⋆ ⇒ TransRefl (reRel v x)
 end.
```

The following example shows how `reRel` is, in fact, a function that generates a relation considering a particular definition of the function `v`.

Example 15. *Let $\Sigma = \{a, b\}$, let R_a and R_b be two binary relations over a set of values of type* `B`*, and let $\alpha = a(b + 1)$ be a regular expression. Moreover, let v be a function that maps the symbol a to the relation R_a, and that maps b to the relation R_b. The computation of* `reRel` α `v` *gives the relation $R_a \circ (R_b \cup \mathcal{I})$, and can be described as follows:*

$$\begin{aligned}
\texttt{reRel}\ \alpha\ v &= \texttt{reRel}\ (a(b+1))\ v \\
&= \texttt{CompRel}\ (\texttt{reRel}\ a\ v)\ (\texttt{reRel}\ (b+1)\ v) \\
&= \texttt{CompRel}\ R_a\ (\texttt{reRel}\ (b+1)\ v) \\
&= \texttt{CompRel}\ R_a\ (\texttt{UnionRel}\ (\texttt{reRel}\ b\ v)\ (\texttt{reRel}\ 1\ v)) \\
&= \texttt{CompRel}\ R_a\ (\texttt{UnionRel}\ R_b\ (\texttt{reRel}\ 1\ v)) \\
&= \texttt{CompRel}\ R_a\ (\texttt{UnionRel}\ R_b\ \texttt{IdRel}).
\end{aligned}$$

Naturally, a word $w = a_1 a_2 \ldots a_n$ can also be interpreted as a relation, namely, the compo-

sition of the relations $v(a_i)$, where v is the function that maps a symbol a_i to a relation R_{a_i}, with $1 \leq i \leq n$. Such interpretation of words as relations is implemented as follows:

```
Fixpoint reRelW (v:A → relation B)(w:word) : relation B :=
 match w with
 | nil ⇒ IdRel
 | x::xs ⇒ CompRel (v x) (reRelW v xs)
 end.
```

Example 16. *Let $\Sigma = \{a, b\}$, let R_a and R_b be two binary relations over a set of values of type `B`. Let w be a word defined by $w = abba$. Moreover, let v be a function that maps the symbol a to the relation R_a, and that maps b to the relation R_b. The function `reRelW` v w yields the relation $R_a \circ R_b \circ R_b \circ R_a \circ \mathcal{I}$, and its computation is summarised as*

$$\begin{aligned} \texttt{reRelW}\ v\ w &= \texttt{reRelW}\ v\ abba \\ &= R_a \circ (\texttt{reRelW}\ v\ bba) \\ &= R_a \circ R_b \circ (\texttt{reRelW}\ v\ ba) \\ &= R_a \circ R_b \circ R_b \circ (\texttt{reRelW}\ v\ a) \\ &= R_a \circ R_b \circ R_b \circ R_a \circ (\texttt{reRelW}\ v\ \epsilon) \\ &= R_a \circ R_b \circ R_b \circ R_a \circ \mathcal{I}. \end{aligned}$$

Now we connect the previous interpretations to regular expression equivalence and relation equivalence. First we present the following inductive predicate, `ReL`, which defines a relation that contains all the pairs (a, b) such that a and b are related by the interpretation of `reRelW` over the elements of the language denoted by some regular expression α.

```
Inductive ReL (v:A → relation B) : re → relation B :=
| mkRel : ∀ α:re, ∀ w:word,
              w ∈ re2rel α → ∀ a b, (reRelW v w) a b → ReL α a b.
```

If two regular expressions α and β are equivalent, then their interpretations in terms of the function `reRelW` must necessarily lead to the equivalence of the values returned by `reRelW` $v\ \alpha\ w$ and by `reRelW` $v\ \beta\ w$, for $w \in \Sigma^\star$. This means that the pairs (a, b) in `ReL` v α and in `ReL` $v\ \beta$ must be the same. This takes us to the main property that is necessary to establish to rely on regular expression equivalence to decide equations involving relations.

Lemma 3. *Let α and β be regular expressions. Let v be a function that maps symbols to relations. Hence, $\alpha \sim \beta \rightarrow$ `ReL` $v\ \alpha \sim_{\mathcal{R}}$ `ReL` $v\ \beta$.*

In order to use Lemma 3 to decide relation equivalence, we must relate `ReL` and `reRel`. As the next lemma shows, both notions end up being equivalent relations.

Lemma 4. *Let α be a regular expression, and let v be a function mapping symbols α of the alphabet to relations. Thus* `reRel` $v\ \alpha \sim_{\mathcal{R}}$ `ReL` $v\ \alpha$.

Together, Lemma 3 and Lemma 4 allow us to prove that if two regular expressions are equivalent, then so are their interpretations on binary relations.

Theorem 1. *Let α and β be regular expressions. Let v be a function that maps symbols to relation. Hence,* $\alpha \sim \beta \rightarrow$ `reRel` $v\ \alpha \sim_{\mathcal{R}}$ `reRel` $v\ \beta$.

Lemmas 3 and 4, and Theorem 1 are defined in our development as follows:

```
Lemma Main : ∀ α:re,
   rel_eq (reRel v α) (ReL α).

Lemma Rel_red_Leq_aux : ∀ α β:re,
   α ∼ β → rel_eq (ReL v α) (ReL v β).

Theorem Rel_red_Leq_final : ∀ α β:re,
   α ∼ β → rel_eq (reRel v α) (reRel v β).
```

We will now describe the details of the tactic construction. The first step is to construct the function that maps symbols of the alphabet into relations. For the underlying alphabet we consider the set of natural numbers, *i.e.*, the values of type `nat`. This function is obtained directly from the individual relations that exist in the equations that our tactic is intended to solve. To obtain such function, we need the following definitions that are responsible for building and finding representatives of relations.

```
Definition STD(T:Type) : relation T := IdRel T.

Definition emp_m(T:Type) : Map[nat,relation T] :=
  @empty nat _ _ (relation T).

Definition add_mm(T:Type)(x:nat)(r:relation T)(m:Map[nat,relation T]) :=
  @add nat _ _ (relation T) x r m.

Definition find_mm(T:Type)(x:nat)(m:Map[nat,relation T]) :=
  match @find nat _ _ (relation T) x m with
  | None ⇒ STD T
  | Some y ⇒ y
  end.

Definition f(T:Type)(v:Map[nat,relation T])(x:nat) :=
  find_mm T x v.
```

The function `add_mm` is responsible for adding a map from a natural number into a relation. The function `find_mm` is responsible for returning the relation that corresponds to a given natural number. The next tactics construct a map `Map[nat,relation` T`]` by following the structure of the goal representing the relation equation. The type `Map` is provided by the Containers library [70], and its usage is similar to the usage of finite sets.

```
Ltac in_there t n m v max :=
 let ev := eval vm_compute in (beq_nat n max) in
   match ev with
    | true ⇒ constr:max
    | _ ⇒ let k := eval vm_compute in (find_mm t n m) in
      match constr:(k=v) with
      | ?X = ?X ⇒ constr:n
      | _ ⇒ in_there t (S n) m v max
      end
  end.

Ltac mk_reif_map t m n v :=
  match v with
  | UnionRel t ?A1 ?A2 ⇒
      let R1 := mk_reif_map t m n A1 in
      let m1 := eval simpl in (fst R1) in
      let m2 := eval simpl in (snd R1) in
      let R2 := mk_reif_map t m1 m2 A2 in
      let m1' := eval simpl in (fst R2) in
      let m2' := eval simpl in (snd R2) in
                 constr:(m1',m2')
  (* ... *)
  | IdRel t ⇒ constr:(m,n)
  | EmpRel t ⇒ constr:(m,n)
  | ?H ⇒
    let y := in_there t 0 m H n in
    let r := eval vm_compute in (beq_nat n y) in
     match r with
     | false ⇒ constr:(m,n)
     | _ ⇒ constr:(add_mm t n H m,S n)
     end
  end.

Ltac reif_rel t n m v max :=
```

```
 let x := in_there t n m v max in
 let b := eval vm_compute in (beq_nat x max) in
  match b with
  | false ⇒ constr:(reRel t (f t m) (‘x))
  | _ ⇒ constr:(reRel t (f t m) 0)
  end.
```

The tactic `in_there` checks for the next natural number to be associated with a relation. The tactic `mk_reif_map` is responsible for following the structure of a given relation and updating the map from natural numbers into relations. Finally, the tactic `reif_rel` is responsible for constructing a term `reRel` considering the map already available (represented by the variable `m`).

Now, we present the tactic that reifies the original goal.

```
Ltac reif t v m mx :=
  match v with
  | EmpRel t ⇒ constr:(reRel t (f t m) 0)
  | IdRel t ⇒ constr: (reRel t (f t m) 1)
  | UnionRel t ?A1 ?A2 ⇒
     let l1 := reif t A1 m mx with l2 := reif t A2 m mx in
       constr:(UnionRel t l1 l2)
  | CompRel t ?A1 ?A2 ⇒
     let l1 := reif t A1 m mx with l2 := reif t A2 m mx in
        constr:(CompRel t l1 l2)
  | trans_refl t ?A1 ⇒
     let l1 := reif t A1 m mx in
        constr:(trans_refl t l1)
  | ?H ⇒ let l := reif_rel t 0 m v mx in
        constr:l
  end.

Ltac reify :=
 match goal with
 | |- rel_eq ?K ?V1 ?V2 ⇒
   let v := set_reif_env K V1 V2 in
   let m := eval simpl in (fst v) with mx := eval simpl in (snd v) in
   let x := fresh "Map_v" with y := fresh "MAX" in
     set(x:=m) ; set(y:=mx) ;
       let t1 := reif K V1 m mx with t2 := reif K V2 m mx in
        change V1 with t1 ; change V2 with t2
```

```
  end.
```

The previous tactic transforms a goal of the form $R_1 = R_2$ into an equivalent goal where the relations involved are changed to their respective interpretations. For example, if the goal is $R_1 \cup R_1 = R_1$, then the tactic `reify` changes it into the goal

$$(\texttt{reRel v `0}) \cup (\texttt{reRel v `0}) = \texttt{reRel v `0},$$

provided that `v` is a function that maps the regular expression `!0` into the relation R_1. Now, a final intermediate tactic is needed to reify the operations over relations.

```
Ltac normalize v :=
 match v with
 | reRel ?T ?F 1 ⇒ constr:1
 | reRel ?T ?F 0 ⇒ constr:0
 | reRel ?T ?F (`X) ⇒ constr:(`X)
 | UnionRel ?T ?X ?Y ⇒
   let x := normalize X with y := normalize Y in
     constr:(x + y)
 | CompRel ?T ?X ?Y ⇒
   let x := normalize X with y := normalize Y in
     constr:(x · y)
 | trans_refl ?T ?X ⇒
     let x := normalize X in
      constr:(x*)
 end.
```

Considering the previous example, the goal we would obtain after applying the tactic `normalize` is the expected one, that is, `reRel v (`0 + `0)` $=$ `reRel v `0`. The final tactic `solve_rel` is defined as follows and integrates all the previous intermediate tactics in order to automatically prove regular equations.

```
Ltac solve_rel :=
  let x := fresh "Map_v" with
   y := fresh "Max" with
   h := fresh "hash" with
   n1 := fresh "norm1" with
   n2 := fresh "norm2" in
   match goal with
   | |- rel_eq ?K ?V1 ?V2 ⇒
     let v := set_reif_env K V1 V2 in
     let m := eval simpl in (fst v) with mx := eval simpl in (snd v) in
       set(x:=m) ; set(y:=mx) ; fold x ;
```

```
    let t1 := reif K V1 m mx with t2 := reif K V2 m mx in
      change V1 with t1 ; change V2 with t2 ; set(h:=f K m) ;
        let k1 := normalize t1 with k2 := normalize t2 in
          set(n1:=reRel K h k1);set(n2:=reRel K h k2)
  end; match goal with
          | |- rel_eq ?T ?V1 ?V2 ⇒
              change V1 with n1 ; change V2 with n2 ; unfold n1,n2 ; clear
                  n1 n2
         end ; apply Rel_red_Leq_final ; dec_re.
```

The version of the tactic `solve_rel` that we have described here uses solely Coq tactical language. More recently, we have developed a new version of the tactic here described, but using OCaml and Coq's API. The new tactic is shorter and more efficient, since it relies on general purpose data structures, such as hash tables, provided by OCaml. However, the usage of the tactic requires an OCaml compiler available, and was not tested enough, thus we have decided to present this first version in the thesis. We now present a simple example of the usage of the tactic `solve_rel` in the automatic generation of a proof of equivalence between two relations.

Example 17. *Let R be a binary relation over a set of values of type T. In what follows, we show the proof of the equation $R_1 \circ R_1^\star \sim_{\mathcal{R}} R_1^\star \circ R_1$ in Coq, using our developed tactic.*

```
Lemma example_1 :
  ∀ T:Type, ∀ R1:relation T, rel_eq T (R1 ∘ R1*) (R1* ∘ R1).
Proof.
  intros;solve_rel.
Qed.
```

3.3.5 Performance

Although the main goal of our development was to provide certified evidence that the decision algorithm suggested by Almeida *et. al.* [3] is correct, it is of obvious interest to understand the usability and efficiency of `equivP` and of the corresponding tactic while being computed within Coq's interactive environment. For that, we have experimented our tactic with several data sets of randomly generated regular expressions, in a uniform way.

The data sets were generated by the FAdo tool [2], and each such data set is composed of 10000 pairs of regular expressions, so that the results are statistically relevant. The experiments were conducted on a Virtual Box environment with 8 Gb of RAM, using coq-8.3pl4. The virtual environment executes on a dual six-core processor AMD Opteron(tm) 2435 processor with 2.60 GHz, and with 16 Gb of RAM. Table 3.1 provides the results obtained from our experiments.

k	$n = 25$		$n = 50$		$n = 100$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
10	0.151	0.026	0.416	0.032	1.266	0.044
20	0.176	0.042	0.442	0.058	1.394	0.072
30	0.183	0.050	0.478	0.074	1.338	0.097
40	0.167	0.058	0.509	0.088	1.212	0.108
50	0.167	0.065	0.521	0.097	1.457	0.141
k	$n = 250$		$n = 500$		$n = 1000$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
10	12.049	0.058	38.402	0.081	-	0.125
20	5.972	0.083	24.674	0.105	58.904	0.181
30	5.511	0.128	17.408	0.157	43.793	0.226
40	5.142	0.147	19.961	0.181	43.724	0.271
50	5.968	0.198	17.805	0.203	46.037	0.280

Table 3.1: Performance results of the tactic `dec_re`.

Each entry in Table 3.1 corresponds to the average time that was required to compute the decision procedure over 10000 pairs of regular expressions. The tests consider both equivalent – denoted by the rows labeled by *eq* – and inequivalent regular expressions – denoted by the rows labeled by *ineq*. The variable k ranges over the sizes of the sets of symbols from which the regular expressions are built. The variable n ranges over the sizes of the regular expressions generated, that is, the total number of constants, symbols and operators of the regular expression. All the timings presented in Table 3.1 are in seconds.

The results presented in Table 3.1 allow us to conclude that the procedure is efficient, since it is able to decide the equivalence of large regular expressions in less that 1 minute. However, the procedure has its pitfalls: whenever the size of the alphabet is small and the size of the regular expressions is considerably large, *e.g.*, for configurations where $k = 10$ and the size of the regular expressions is 1000, or where $k = 2$ and the size of the regular expressions is 250, the decision procedure – and therefore, the tactic – take a long time to give a result. This dues to the fact that the derivations computed along the execution of the procedure tend to produce fewer derivatives resulting in the pair $(\emptyset, \emptyset)$ and so, more recursive calls are needed.

When testing inequivalences, our decision procedure is very efficient, even for the larger families of regular expressions considered. This can bring advantages when using our tactic, for instance, as an argument for the `try` tactic. Although the tactic presented here does not present counter-examples to the user, it is quite easy to change it in order to do so. In the next section we present a comparison of the performances exhibited by our procedure with other developments that are available.

3.4 Related Work

The subject of developing certified algorithms for deciding regular expression equivalence within theorem provers is not new. In recent years, much attention was directed to this particular subject, resulting in several formalizations, some of which are based on derivatives, and spawn along three different interactive theorem provers, namely Coq, Isabelle [79], and Matita [12].

The most complete of the developments is the one of Braibant and Pous [18]: the authors formalised Kozen's proof of the completeness of KA [57] and developed also efficient tactics to decide KA equalities by computational reflection. Their construction is based on the classical automata process for deciding regular expressions equivalence without minimisation of the involved automata. Moreover, they use a variant of Illie and Yu's method [52] for constructing automata from regular expressions, and the comparison is performed using Karp's [49] direct comparison of automata. The resulting development is quite general (it is able to prove (in)equivalence of expression of several models of Kleene algebra) and it is also quite efficient due to a careful choice of the data structures involved.

The works that are closer to ours are the works of Coquand and Siles [29], and of Nipkow and Krauss [66]. Coquand and Siles implemented a procedure for regular expression equivalence based on Brzozowski's derivative method, supported by a new construction of finite sets in type theory. They prove their algorithm correct and complete. Nipkow and Krauss' development is also based in Brzozowski's derivative, and it is a compact and elegant development carried out in the Isabelle theorem prover. However, the authors did not formalise the termination and completeness of the algorithm. In particular, the termination is far from being a trivial subject, as demonstrated by the work presented in this thesis, and in the work of Coquand and Siles.

More recently, Asperti presented a development [11] of an algorithm based on pointed regular expressions, which are regular expressions containing internal points. These points serve as indicators of the part of the regular expression that was already processed (transformed into a DFA) and therefore which part of the regular expression remains to be processed. The development is also quite short and elegant and provides an alternative to the algorithms based on Brzozowski's derivatives, since it does not require normalisation modulo a suitable set of axioms to prove the finiteness of the number of the states of the corresponding DFA.

In Table 3.2 we provide results about a comparison between our development and the one of Braibant and Pous. We no not present comparison with the other two Coq developments since they clearly exhibit worst performances than ours and the previous one. For technical reasons, we were not able to test the development of Asperti. In these experiments we have used a datasets of 1000 uniform randomly generated regular expressions, and they were conducted in a Macbook Pro 15", with a 2.3 GHz Intel Core i7 processor with 4 GB of RAM

alg./(k,n)	$(2,5)$		$(2,10)$		$(2,20)$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
equivP	0.003	0.002	0.008	0.003	0.020	0.004
ATBR	0.059	0.016	0.080	0.042	0.258	0.099
	$(4,20)$		$(4,50)$		$(10,100)$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
equivP	0.035	0.004	0.172	0.010	0.776	0.016
ATBR	0.261	0.029	0.436	0.358	1.525	0.874
	$(20,200)$		$(50,500)$		$(50,1000)$	
	eq	*ineq*	*eq*	*ineq*	*eq*	*ineq*
equivP	2.211	0.048	9.957	0.121	17.768	0.149
ATBR	3.001	1.654	5.876	2.724	16.682	12.448

Table 3.2: Comparison of the performances.

memory.

It is clear from Table 3.2 that the work of Braibant and Pous scales better than ours for larger families of regular expressions but it is drastically slower than ours with respect to regular expression inequivalence. For smaller families of regular expressions, our procedure is also faster than theirs in both cases. The values k and n in Table 3.2 are the same measures that were used in Table 3.1, presented in the previous section for the analysis of the performance of `equivP`.

3.5 Conclusions

In this chapter we have described the mechanisation, within the Coq proof assistant, of the procedure equivP for deciding regular expressions equivalence based on the notion of partial derivatives. This procedure decides the (in)equivalence of regular expressions by an iterated method of comparing the equivalence of their partial derivatives. The main advantage of our method, when compared to the ones based on Brzozowski's derivatives, is that it does not require normalisation modulo the associativity, commutativity and idempotence of the $+$ operator in order to prove the finiteness of the number of derivatives and of the termination of the corresponding algorithms. The performances exhibited by our algorithm are satisfactory. Nevertheless, there is space for improvement. A main point of improvement is the development of intermediate tactics that are able to automate common proof steps.

An interesting continuation of our development is its extension to support extended regular expressions, that is, regular expressions containing intersection and complement. The recent work of Caron, Champarnaud and Mignot [23] extends the notion of partial derivative to

handle these extended regular expressions and its addition to our formalisation should not carry any major difficulty.

Another point that we wish to address is the representation of partial derivatives similarly to the work of Almeida *et. al.*, where partial derivatives are represented in a linear way. This representation has the advantage of reducing the number of symbols involved in the derivation process whenever some of the symbols lead to derivatives whose result is the empty set.

Chapter 4

Equivalence of KAT Terms

Kleene algebra with tests (KAT) [59, 64] is an algebraic system that extends Kleene algebra, the algebra of regular expressions, by considering a subset of *tests* whose elements satisfy the axioms of Boolean algebra. The addition of tests brings a new level of expressivity in the sense that in KAT we are able to express imperative program constructions, rather than just non-deterministic choice, sequential composition and iteration on a set of actions, as it happens with regular expressions.

KAT is specially fitted to capture and verify properties of simple imperative programs since it provides an equational way to deal with partial correctness and program equivalence. In particular KAT subsumes *propositional Hoare logic* (PHL) [65, 60] in the sense that PHL's deductive rules become theorems of KAT. Consequently, proving that a given program p is partially correct using the deductive system of PHL is tantamount to checking if p is partially correct by equational reasoning in KAT. Moreover, some Horn formulas [43, 44] of KAT can be reduced into standard equalities which can then be decided automatically using one of the available methods [101, 64, 62].

In this chapter we present a mechanically verified implementation of a procedure to decide KAT terms equivalence using partial derivatives. The decision procedure is an extension of the procedure already introduced and described in the previous chapter. The Coq development is available in [75].

4.1 Kleene Algebra with Tests

A KAT is a KA extended with an embedded *Boolean algebra* (BA). Formally, a KAT is an algebraic structure

$$(K, T, +, \cdot, ^\star, ^-, 0, 1),$$

such that $(K, +, \cdot, ^\star, 0, 1)$ is a KA, $(T, +, \cdot, ^-, 0, 1)$ is a Boolean algebra and $T \subseteq K$. Therefore, KAT satisfies the axioms of KA and the axioms of Boolean algebra, that is, the set of axioms (3.8–3.22) and the following ones, for $b, c, d \in T$:

$$bc = cb \tag{4.1}$$

$$b + (cd) = (b + c)(b + d) \tag{4.2}$$

$$\overline{b + c} = \overline{b}\overline{c} \tag{4.3}$$

$$b + \overline{b} = 1 \tag{4.4}$$

$$bb = b \tag{4.5}$$

$$b + 1 = 1 \tag{4.6}$$

$$b + 0 = b \tag{4.7}$$

$$\overline{bc} = \overline{b} + \overline{c} \tag{4.8}$$

$$b\overline{b} = 0 \tag{4.9}$$

$$\overline{\overline{b}} = b \tag{4.10}$$

4.2 The Language Model of KAT

Primitive Tests, Primitive Actions, and Atoms

Let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a non-empty finite set whose elements are called *primitive tests*. Let $\overline{\mathcal{B}} = \{\overline{b} \mid b \in \mathcal{B}\}$ be the set such that each element $l \in \mathcal{B} \cup \overline{\mathcal{B}}$ is called a *literal*. An *atom* is a finite sequence of literals $l_1 l_2 \ldots l_n$, such that each l_i is either b_i or $\overline{b_i}$, for $1 \leq i \leq n$, where $n = |\mathcal{B}|$. We will refer to atoms by $\alpha, \alpha_1, \alpha_2, \ldots$, and to the set of all atoms on $\mathcal{B}$ by At. The set At can be regarded as the set of all truth assignments to elements of $\mathcal{B}$. Consequently, there are exactly $2^{|\mathcal{B}|}$ atoms. The following example shows the set of atoms for a given set of primitive tests.

Example 18. *Let $\mathcal{B} = \{b_1, b_2\}$. The set of all atoms over $\mathcal{B}$ is $\mathsf{At} = \{b_1 b_2, b_1\overline{b_2}, \overline{b_1}b_2, \overline{b_1 b_2}\}$.*

Given an atom $\alpha \in \mathsf{At}$ and a primitive test $b \in \mathcal{B}$, we write $\alpha \leq b$ if $\alpha \to b$ is a propositional tautology. For each primitive test $b \in \mathcal{B}$ and for each atom $\alpha \in \mathsf{At}$ we always have $\alpha \leq b$ or $\alpha \leq \overline{b}$.

Example 19. *Let $\mathcal{B} = \{b_1, b_2\}$. The subset of At formed by those atoms α such that $\alpha \leq b_1$ is the set $B = \{b_1 b_2, b_1\overline{b_2}\}$.*

Besides primitive tests we also have to consider a finite set of symbols representing atomic programs, whose role is the same of the alphabet in the case of regular expressions. Such set in KAT is called the set of *primitive actions* $\Sigma = \{p_1, \ldots, p_m\}$ and is represented in our formalisation by the type of integers `Z` available in Coq's standard library.

In the Coq development we have encoded primitive tests and atoms as terms of the type of finite ordinal numbers `Ord` n which we present below. The type `Ord` n consists of two parameters: a natural number `nv` and a proof term `nv_lt_n` witnessing that `nv` is strictly smaller that n. The functions for constructing an ordinal containing the value zero and for calculating the successor of an ordinal are also included in the code excerpt. We consider the existence of a global value `ntests` of type `nat` that establishes the cardinality of $\mathcal{B}$. The value of the parameter `ntests` is defined by instantiating the module type `KAT_Alph`. This module is a parameter for the rest of the modules that compose our development.

```
Module Type KAT_Alph.
  Parameter ntests : nat.
End KAT_Alph.

Record Ord (n:nat) := mk_Ord {
  nv :> nat;
  nv_lt_n : ltb nv n = true
}.

Definition ord0(n:nat) : Ord (S n) := @mk_Ord (S n) 0 eq_refl.
Definition ordS(n:nat)(m:Ord n) := @mk_Ord (S n) (S m) ((ord_lt n m)).
```

A member in $\mathcal{B}$ is a term of type `Ord (ntests)` and an atom in At is an inhabitant of the type `Ord` ($2^{\texttt{ntests}}$). We can calculate the set of all atoms as given by the function `ords_up_to` introduced below. The statement that proves the fact that `ords_up_to` calculates the set At on $\mathcal{B}$ is the lemma `all_atoms_in_ords_up_to` given below.

```
Definition ordS_map (n:nat) (s:set (Ord n)) : set (Ord (S n)) :=
   map (@ordS n) s.

Fixpoint ords_up_to (n:nat) {struct n} : set (Ord n) :=
  match n with
    | 0 ⇒ ∅
    | S m ⇒ add (ord0 m) (@ordS_map m (ords_up_to m))
  end.

Lemma all_atoms_in_ords_up_to : ∀ (n:nat)(i:Ord n), i ∈ (ords_up_to n).
```

In order to finish the development of primitive tests and atom related concepts we need to define how we evaluate tests with respect to these structures. Let $b \in \mathcal{B}$ be a term of type `Ord` n, and let m be the natural number it represents. In order to check that $\alpha \leq b$, where α is represented by a value of type `Ord` ($2^{\texttt{ntests}}$), we simply look at the mth bit of the value of α. If that bit is 1 then $\alpha \leq b$ is true, and false otherwise. The code below presents the Coq

code for making this decision, where `N.testbit_nat` is a function that converts a value of type `nat` into its corresponding bit representation and returns the nth bit, where n is given as argument.

```
Definition nth_bit(m:nat)(k:Ord m)(n:nat) : bool :=
  N.testbit_nat (N.of_nat k) n.
```

Tests and Terms

The syntax of KAT terms extends the syntax of regular expressions – or the syntax of KA – with elements called *tests*, which can be regarded as Boolean expressions on the underlying Boolean algebra of any KAT. A test is inductively defined as follows:

- the constants 0 and 1 are tests;
- if $b \in \mathcal{B}$ then b is a test;
- if t_1 and t_2 are tests, then $t_1 + t_2$, $t_1 \cdot t_2$, and $\overline{t_1}$ are tests.

We denote the set of tests on $\mathcal{B}$ by T. In this setting, the operators $\cdot$, $+$, and $^-$ are interpreted as Boolean operations of conjuntion, disjunction, and negation, respectively. The operators $\cdot$ and $+$ are naturally overloaded with respect to their interpretation as operators over elements of the underlying KA, where they correspond to non-deterministic choice and sequence, respectively.

A KAT *term* e is inductively defined as follows:

- if t is a test then t is a KAT term;
- if $p \in \Sigma$, the p is a KAT term;
- if e_1 and e_2 are KAT terms, then so are their union $e_1 + e_2$, their concatenation $e_1 \cdot e_2$, and their Kleene star $e_1^\star$.

The set of all KAT terms is denoted by $\mathsf{K}_{\mathcal{B},\Sigma}$, and we denote syntactical equality between $e_1, e_2 \in \mathsf{K}_{\mathcal{B},\Sigma}$ by $e_1 \equiv e_2$. As usual, we ommit the concatenation operator in $e_1 \cdot e_2$ and write e_1e_2 instead. In Coq, the type of KAT terms and the type of tests are defined as expected by

```
Inductive test : Type :=
| ba0 : test
| ba1 : test
| baV : Ord ntests → test
| baN : test → test
```

```
| baAnd : test → test → test
| baOr : test → test → test.

Inductive kat : Type :=
| kats : Z→  kat
| katb : test → kat
| katu : kat → kat → kat
| katc : kat → kat → kat
| katst : kat → kat.
```

Like primitive tests, tests are evaluated with respect to atoms for validity. The function `evalT` below implements this evaluation following the inductive structure of tests. For $t \in \mathsf{T}$ and $\alpha \in \mathsf{At}$, we denote evaluation of tests with respect to atoms by $\alpha \leq t$. For evaluating $\alpha \leq t$, where t is a test, we consider the following recursive function `evalT` which is defined as the evaluation of a Boolean expression with respect to a Boolean valuation of the set of Boolean variables under consideration (in our case, the set $\mathcal{B}$).

```
Function evalT(α:Ord (pow2 ntests))(t:test) : bool :=
 match α with
 | ba0 ⇒ false
 | ba1 ⇒ true
 | baV b ⇒ nth_bit _ α b
 | baN t1 ⇒ negb (evalT α t1)
 | baAnd t1 t2 ⇒ (evalT α t1) && (evalT α t2)
 | baOr t1 t2 ⇒ (evalT α t1) || (evalT α t2)
 end.
```

Below we give an example of the evaluation of a test with respect to a given atom.

Example 20. *Let $\mathcal{B} = \{b_1, b_2\}$, and let $t = b_1 + \overline{b_2}$. The evaluation $\overline{b_1 b_2} \leq t$ holds since*

$$\begin{aligned}
\overline{b_1 b_2} \leq t &= \texttt{evalT}\ (\overline{b_1 b_2})\ (b_1 + \overline{b_2}) \\
&= \texttt{evalT}\ (\overline{b_1 b_2})\ (b_1)\ \texttt{||}\ \texttt{evalT}\ (\overline{b_1 b_2})\ (\overline{b_2}) \\
&= \texttt{nth_bit _}\ (\overline{b_1 b_2})\ (b_1)\ \texttt{||}\ \texttt{evalT}\ (\overline{b_1 b_2})\ (\overline{b_2}) \\
&= \texttt{false}\ \texttt{||}\ \texttt{evalT}\ (\overline{b_1 b_2})\ (\overline{b_2}) \\
&= \texttt{evalT}\ (\overline{b_1 b_2})\ (\overline{b_2}) \\
&= \texttt{negb}\ (\texttt{evalT}\ (\overline{b_1 b_2})\ (b_2)) \\
&= \texttt{nth_bit _}\ (\overline{b_1 b_2})\ (b_2) \\
&= \texttt{negb}\ (\texttt{false}) = \texttt{true}
\end{aligned}$$

Note that the previous example considers the atom $\overline{b_1 b_2}$ which corresponds to the term of type `Ord (pow2 2)` representing the natural number 0.

Guarded Strings and Fusion Products

There are various models for KAT but, as in the case of KA, its standard model is the one of sets of regular languages, whose elements in the case of KAT are *guarded strings* [54, 61]. A guarded string is a sequence

$$x = \alpha_0 p_0 \alpha_1 p_1 \ldots p_{(n-1)} \alpha_n,$$

with $\alpha_i \in \mathsf{At}$ and $p_i \in \Sigma$. Guarded strings start and end with an atom. When $n = 0$, then the guarded string is a single atom $\alpha_0 \in \mathsf{At}$. We use $x, y, x_0, y_0, \ldots$ to refer to guarded strings. The set of all guarded strings over the sets $\mathcal{B}$ and Σ is denoted by $\mathsf{GS}_{\mathcal{B},\Sigma}$. Guarded strings are defined in Coq by the following inductive type.

```
Inductive gs : Type :=
|gs_end : atom → gs
|gs_conc : atom → Z → gs → gs.
```

For guarded string x we define $\mathsf{first}(x) \stackrel{\text{def}}{=} \alpha_0$ and $\mathsf{last}(x) \stackrel{\text{def}}{=} \alpha_n$. We say that two guarded strings x and y are *compatible* if $\mathsf{last}(x) = \mathsf{first}(y)$. Both operations and the notion of compatibility between guarded strings are defined in Coq as follows.

```
Definition first(x:gs) : atom :=
 match x with
 | gs_end k ⇒ k
 | gs_conc k _ _ ⇒ k
 end.

Fixpoint last(x:gs) : atom :=
 match x with
 | gs_end k ⇒ k
 | gs_conc _ _ k ⇒ last k
 end.

Definition compatible (x y:gs) := last x = first y.
```

If two guarded strings x and y are compatible, then the *fusion product* $x \diamond y$, or simply xy, is the standard word concatenation but omitting one of the common atoms $\mathtt{last}(x)$, or $\mathtt{first}(y)$. The fusion product of two guarded strings x and y is a partial function since it is only defined when x and y are compatible.

Example 21. *Let $\mathcal{B} = \{b_1, b_2\}$ and let $\Sigma = \{p, q\}$. Let $x = b_1 b_2 p \overline{b_1 b_2}$ and $y = \overline{b_1 b_2} q b_1 \overline{b_2}$. The fusion product $x \diamond y$ is the guarded string $b_1 b_2 p \overline{b_1 b_2} q b_1 \overline{b_2}$. On the contrary, the fusion product $y \diamond x$ is not defined since $\mathsf{last}(y) = b_1 \overline{b_2} \neq b_1 b_2 = \mathsf{first}(x)$.*

In Coq we have implemented the fusion product of two guarded strings x and y by means of a dependent recursive function whose arguments are x, y, and an explicit proof of the compatibility of x and y, *i.e.*, a term of type `compatible` x y. The function `fusion_prod` implements the fusion product based on this criteria.

```
Lemma compatible_tl:
 ∀ (x y x':gs)(α:atom)(p:sy),
  ∀ (h:compatible x y)(l:x = gs_conc x p x'), compatible x' y.

Fixpoint fusion_prod (x y:gs)(h:compatible x y) : gs :=
 match x as x' return x = x' → gs with
 |gs_end _ ⇒ fun (_:(x = gs_end _)) ⇒ y
 |gs_conc k s t ⇒ fun (h0:(x = gs_conc k s t)) ⇒
                         let h' := compatible_tl x y h k s t h0 in
                          gs_conc k s (fusion_prod t y h')
end (refl_equal x).
```

Since the parameter h depends on the guarded strings x and y it must recursively decrease accordingly. The lemma `compatible_tl` states that if two guarded strings x and y are compatible, and if $x = \alpha p :: x'$, for some $\alpha p \in (\mathsf{At} \cdot \Sigma)$ and $x' \in \mathsf{GS}_{\mathcal{B},\Sigma}$, then x' and y remain compatible. The main properties of the fusion product over compatible guarded strings are the following: if $x, y, z \in \mathsf{GS}_{\mathcal{B},\Sigma}$ then the fusion product is associative, *i.e.*, $(xy)z = x(yz)$; the fusion product of a guarded string x with a compatible atom α is an absorbing operation on the left or right of α, *i.e.*, $\alpha x = x$ and $x\alpha = x$; the function `last` is left-invariant with respect to the fusion product, *i.e.*, $\texttt{last}(xy) = \texttt{last}(y)$; conversly, the function `first` is right-invariant with respect to the fusion product, *i.e.*, $\texttt{first}(xy) = \texttt{first}(x)$.

Languages of Guarded Strings

In the language theoretic model of KAT, a *language* is a set of guarded strings over the sets $\mathcal{B}$ and Σ, *i.e.*, a subset of $\mathsf{GS}_{\mathcal{B},\Sigma}$. The empty language $\emptyset$ and the language containing all atoms are formalised in Coq as follows:

```
Definition gl := gs → Prop.

Inductive gl_emp : gl := ∅
Notation "∅" := gl_emp.

Inductive gl_eps : gl :=
| in_gl_eps : ∀ α:atom, α ∈ gl_eps.
Notation "At" := gl_eps.
```

The language of a primitive test b is the set of all atoms $\alpha \in \mathsf{At}$ such that $\alpha \leq b$, and the language of a test t is the set of all the atoms α such that $\alpha \leq t$. The language containing a single guarded string x is also defined.

```
Inductive g_bv : bv → gl :=
| in_g_bv : ∀ (b:bv)(α:atom), α ≤ b → b ∈ gl_bv b.

Inductive gl_test : gl : test → gl :=
| in_gl_test : ∀ (α:atom)(t:test), α ≤ t → α ∈ gl_test t.

Inductive gl_at_only (a:atom) : gl :=
| mk_gl_atom_only : (gs_end a) ∈ gl_at_only a.
```

The language of a symbol $p \in \Sigma$ is the set of all guarded strings x such that $x = \alpha p \beta$, with $\alpha, \beta \in \mathsf{At}$.

```
Inductive gl_sy : sy → gl :=
| in_gl_sy : ∀ (p:sy)(α β:atom), αpβ ∈ gl_sy p.
```

The union of two languages L_1 and L_2 is defined as the usual union of sets. Given two languages L_1 and L_2 we define the set $L_1 L_2$ as the set of all the fusion products xy such that $x \in L_1$ and $y \in L_2$.

```
Inductive gl_conc(L1 L2:gl) : gl :=
|mkg_gl_conc : ∀ (x y:gl)(T:compatible x y),
    x ∈ L1 → y ∈ L2 → (fusion_prod x y T) ∈ (gl_conc L1 L2).
```

Example 22. *Let $\mathcal{B} = \{b_1, b_2\}$ and $\Sigma = \{p, q\}$. Let $L_1 = \{b_1 b_2 p \overline{b_1 b_2}, \overline{b_1 b_2}, b_1 b_2 q b_1 \overline{b_2}\}$ and $L_2 = \{\overline{b_1 b_2} p b_1 b_2, \overline{b_1 b_2}, b_1 \overline{b_2} q b_1 b_2\}$. The fusion product of the languages of guarded strings L_1 and L_2 is the language*

$$L_1 L_2 = \{b_1 b_2 p \overline{b_1 b_2} p b_1 b_2, b_1 b_2 p \overline{b_1 b_2}, \overline{b_1 b_2} p b_1 b_2, \overline{b_1 b_2}, b_1 b_2 q b_1 \overline{b_2} q b_1 b_2\}.$$

The power and the Kleene star of a language L, denoted respectively by L^n and $L^\star$, are defined as expected. Their formalisation in Coq is as follows:

```
Inductive gl_conc(L1 L2:gl) : gl :=
|mkg_gl_conc : ∀ (x y:gl)(T:compatible x y),
    x ∈ L1 → y ∈ L2 → (fusion_prod x y T) ∈ (gl_conc L1 L2).
Notation "x ⋄ y" := (gl_conc x y).

Fixpoint conc_gln(L:gl)(n:nat) : gl :=
 match n with
 | O ⇒ {1}
```

```
 | S m ⇒ L ◇ (conc_gln L m)
 end.

Inductive gl_star(L:gl) : gl :=
|mk_gl_star : ∀ (n:nat)(x:gs), x ∈ (conc_gln L n) → x ∈ (gl_star L).
Notation "x⋆" := (gl_star x).
```

As with regular expressions, the equality of languages of guarded strings is set equality and is defined in Coq as follows:

```
Definition gl_eq (L1 L2:gl) := Same_set _ L1 L2.
Notation "x == y" := (gl_eq x y).
Notation "x != y" := (¬(x == y)).
```

KAT terms are syntactical expressions that denote languages of guarded strings. Thus, given a KAT term e, the language that e denotes, $\mathsf{G}(e)$, is recursively defined on the structure of e as follows:

$$\begin{aligned}
\mathsf{G}(p) &\stackrel{\text{def}}{=} \{\alpha p\beta \,|\, \alpha,\beta \in \mathsf{At}\}\ ,\, p \in \Sigma\\
\mathsf{G}(t) &\stackrel{\text{def}}{=} \{\alpha \in \mathsf{At} \,|\, \alpha \leq t\}\ ,\, t \in T\\
\mathsf{G}(e_1+e_2) &\stackrel{\text{def}}{=} \mathsf{G}(e_1) \cup \mathsf{G}(e_2)\\
\mathsf{G}(e_1e_2) &\stackrel{\text{def}}{=} \mathsf{G}(e_1)\mathsf{G}(e_2)\\
\mathsf{G}(e^\star) &\stackrel{\text{def}}{=} \cup_{n\geq 0}\mathsf{G}(e)^n.
\end{aligned}$$

From the previous definition it is easy to conclude that $\mathsf{G}(1) = \mathsf{At}$ and that $\mathsf{G}(0) = \emptyset$. If $x \in \mathsf{GS}_{\mathcal{B},\Sigma}$, then its language is $\mathsf{G}(x) = \{x\}$. If e_1 and e_2 are two KAT terms, we say that e_1 and e_2 are equivalent, and write $e_1 \sim e_2$, if and only if $\mathsf{G}(e_1) = \mathsf{G}(e_2)$.

Example 23. *Let $\mathcal{B} = \{b_1, b_2\}$ and $\Sigma = \{p, q\}$. Let $e = b_1p + q\overline{b_2}$. The language denoted by the* KAT *term e is the following:*

$$\begin{aligned}
\mathsf{G}(e) &= \mathsf{G}(b_1p + q\overline{b_2})\\
&= \mathsf{G}(b_1p) \cup \mathsf{G}(q\overline{b_2})\\
&= \mathsf{G}(b_1)\mathsf{G}(p) \cup \mathsf{G}(q)\mathsf{G}(\overline{b_2})\\
&= \{\alpha|\alpha \in \mathsf{At}, \alpha \leq b_1\} \diamond \{\alpha p\beta|\alpha,\beta \in \mathsf{At}\} \cup \{\alpha q\beta|\alpha,\beta \in \mathsf{At}\} \diamond \{\alpha|\alpha \in \mathsf{At}, \alpha \leq \overline{b_2}\}\\
&= \{\alpha p\beta|\alpha,\beta \in \mathsf{At}, \alpha \leq b_1\} \cup \{\alpha q\beta|\alpha,\beta \in \mathsf{At}, \beta \leq \overline{b_2}\}.
\end{aligned}$$

We naturally extend the function G to sets S of KAT terms by $\mathsf{G}(S) \stackrel{\text{def}}{=} \cup_{e\in S}\mathsf{G}(e)$. If S_1 and S_2 are sets of KAT terms then $S_1 \sim S_2$ if and only if $\mathsf{G}(S_1) = \mathsf{G}(S_2)$. Moreover, if e is a KAT term and S is a set of KAT terms then $e \sim S$ if and only if $\mathsf{G}(e) = \mathsf{G}(S)$.

We also have to consider the left-quotient of languages $L \subseteq \mathsf{GS}_{\mathcal{B},\Sigma}$. Quotients with respect to words $w \in (\mathsf{At} \cdot \Sigma)^\star$ are defined by

$$\mathcal{D}_w(L) \stackrel{\text{def}}{=} \{ x \mid wx \in L \},$$

and are specialised to elements $\alpha p \in (\mathsf{At} \cdot \Sigma)$ by

$$\mathcal{D}_{\alpha p}(L) \stackrel{\text{def}}{=} \{ x \mid \alpha p x \in L \}.$$

In Coq we define the function `kat2gl` that implements the function G, and the inductive predicates `LQ` and `LQw` that implement, respectively, the left-quotients of a language with respect to guarded strings and elements of $(\mathsf{At} \cdot \Sigma)$.

```
Fixpoint kat2gl(e:kat) : gl :=
 match e with
 | kats x ⇒ gl_sy x
 | katb b ⇒ gl_atom b
 | katu e1 e2 ⇒ gl_union (kat2gl e1) (kat2gl e2)
 | katc e1 e2 ⇒ gl_conc (kat2gl e1) (kat2gl e2)
 | katst e' ⇒ gl_star (kat2gl e')
 end.
Notation G := kat2gl.

Inductive LQ (l:gl) : atom → sy → gl :=
|in_quo : ∀ (a:atom)(p:sy)(y:gs), (gs_conc a p y) ∈ l → y ∈ LQ l a p.

Inductive LQw (l:gl) : gstring → gl :=
|in_quow : ∀ (x w:gs)(T:compatible w x),
    (fusion_prod w x T) ∈ l → x ∈ LQw l w.
```

Moreover, we will also need the notion of quotient with respect to words in $(\mathsf{At}\cdot\Sigma)^\star$. For that we introduce functions to obtain a word from a guarded string and vice-versa. The functions are defined as follows:

```
Fixpoint to_gstring (l:list (atom*Z))(g:gstring) : gstring :=
 match l with
  | [] ⇒ g
  | x::xs ⇒ gs_conc (fst x) (snd x) (to_gstring xs g)
 end.

Fixpoint from_gstring (g:gstring) : list (atom*Z) :=
 match g with
  | gs_end a ⇒ []
```

```
 | gs_conc a p x ⇒ (a,p)::from_gstring x
end.
```

The following properties hold between words and guarded strings:

```
Lemma to_gstring_app :
  ∀ w1 w2 x, to_gstring (w1 ++ w2) x = to_gstring w1 (to_gstring w2 x).

Lemma from_to_gstring :
  ∀ w, w = (to_gstring (from_gstring w) (gs_end (last w))).

Lemma from_string_correct :
  ∀ x, x = to_gstring (from_gstring x) (gs_end (last x)).
```

Finally, we introduce another notion of left-quotient, this time with respect to words. Its definition in Coq is as follows:

```
Inductive LQw_alt (l:gl) : list (atom*Z) → gl :=
| in_quow_alt : ∀ x w, (to_gstring x w) ∈ l → w ∈ (LQw_alt l x).
```

The role of the predicate `LQw_alt` is going to be the same as the role of left-quotients with respect to words in the case of regular expressions, that is, the predicate `LQw_alt` will serve as the language model of partial derivatives of KAT terms. We will get back to this particular point briefly.

4.3 Partial Derivatives of KAT Terms

The notion of derivative of a KAT term was introduced by Kozen in [62] as an extension of the Brzozowski's derivatives. In the same work, Kozen also introduces the notion of *set derivative*, to which we will call partial derivative of a KAT term.

Before formally introducing partial derivatives, we have to introduce the notion of nullability of a KAT term. Given an atom α and a KAT term e, the function inductively defined by

$$
\begin{aligned}
\varepsilon_\alpha(p) &\stackrel{\text{def}}{=} \texttt{false},\\
\varepsilon_\alpha(t) &\stackrel{\text{def}}{=} \begin{cases} \texttt{true} & \text{if } \alpha \leq t,\\ \texttt{false} & \text{if } \alpha \not\leq t.\end{cases}\\
\varepsilon_\alpha(e_1+e_2) &\stackrel{\text{def}}{=} \varepsilon_\alpha(e_1) \mathbin{||} \varepsilon_\alpha(e_2),\\
\varepsilon_\alpha(e_1e_2) &\stackrel{\text{def}}{=} \varepsilon_\alpha(e_1) \mathbin{\&\&} \varepsilon_\alpha(e_2),\\
\varepsilon_\alpha(e^\star) &\stackrel{\text{def}}{=} \texttt{true},
\end{aligned}
$$

determines if e is nullable. The function $\varepsilon_\alpha(\cdot)$ is extended to the set of all atoms At by

$$\mathsf{E}(e) \stackrel{\text{def}}{=} \{\alpha \in \mathsf{At} \mid \varepsilon_\alpha(e) = \texttt{true}\}.$$

As with the notion of nullable regular expression, we can relate the results of $\varepsilon_\alpha(e)$ with language membership by

$$\varepsilon_\alpha(e) = \texttt{true} \rightarrow \alpha \in \mathsf{G}(e),$$

and, symmetrically, by

$$\varepsilon_\alpha(e) = \texttt{false} \rightarrow \alpha \notin \mathsf{G}(e).$$

For KAT terms e_1 and e_2, if $\varepsilon_\alpha(e_1) = \varepsilon_\alpha(e_2)$ holds for all $\alpha \in \mathsf{At}$, then we say that e_1 and e_2 are *equi-nullable.*

Example 24. *Let $\mathcal{B} = \{b_1, b_2\}$, let $\Sigma = \{p, q\}$, and let $e = b_1 p + q\overline{b_2}$. The computation of $\varepsilon_{b_1 b_2}(e)$ goes as follows:*

$$\begin{aligned}
\varepsilon_{b_1b_2}(e) &= \varepsilon_{b_1b_2}(b_1p + q\overline{b_2}) \\
&= \varepsilon_{b_1b_2}(b_1p) \mid\mid \varepsilon_{b_1b_2}(q\overline{b_2}) \\
&= (\varepsilon_{b_1b_2}(b_1) \;\&\&\; \varepsilon_{b_1b_2}(p)) \mid\mid (\varepsilon_{b_1b_2}(q) \;\&\&\; \varepsilon_{b_1b_2}(\overline{b_2})) \\
&= (b_1b_2 \leq b_1 \;\&\&\; \texttt{false}) \mid\mid (\texttt{false} \;\&\&\; b_1b_2 \leq \overline{b_2}) \\
&= (\texttt{true} \;\&\&\; \texttt{false}) \mid\mid (\texttt{false} \;\&\&\; \texttt{false}) \\
&= \texttt{false}.
\end{aligned}$$

However, if we consider $e = b_1 + \overline{b_2}$, we obtain a positive result. The computation of $\varepsilon_{b_1b_2}(e)$ goes as follows:

$$\begin{aligned}
\varepsilon_{b_1b_2}(\alpha) &= \varepsilon_{b_1b_2}(b_1 + \overline{b_2}) \\
&= \varepsilon_{b_1b_2}(b_1) \mid\mid \varepsilon_{b_1b_2}(\overline{b_2}) \\
&= b_1b_2 \leq b_1 \mid\mid b_1b_2 \leq \overline{b_2} \\
&= \texttt{true} \mid\mid \texttt{false} \\
&= \texttt{true}.
\end{aligned}$$

Nullability is extended to sets in the following way:

$$\varepsilon_\alpha(S) \stackrel{\text{def}}{=} \begin{cases} \texttt{true} & \text{if } \exists e \in S, \varepsilon_\alpha(e) = \texttt{true}; \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Two sets S_1 and S_2 of KAT terms are *equi-nullable* if $\varepsilon_\alpha(S_1) = \varepsilon_\alpha(S_2)$. For sets of KAT terms we also define the concatenation of a set with a KAT term by

$$S \odot e \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } e = 0, \\ S & \text{if } e = 1, \\ \{e'e \mid e' \in S\} & \text{otherwise.} \end{cases}$$

As usual, we omit the operator $\odot$ whenever possible.

Let $\alpha p \in (\mathsf{At} \cdot \Sigma)$ and let e be a KAT term. The *set* $\partial_{\alpha p}(e)$ *of partial derivatives* of e with respect to αp is inductively defined by

$$\begin{aligned}
\partial_{\alpha p}(t) &\stackrel{\text{def}}{=} \emptyset, \\
\partial_{\alpha p}(q) &\stackrel{\text{def}}{=} \begin{cases} \{1\} & \text{if } p \equiv q, \\ \emptyset & \text{otherwise.} \end{cases} \\
\partial_{\alpha p}(e_1 + e_2) &\stackrel{\text{def}}{=} \partial_{\alpha p}(e_1) \cup \partial_{\alpha p}(e_2), \\
\partial_{\alpha p}(e_1 e_2) &\stackrel{\text{def}}{=} \begin{cases} \partial_{\alpha p}(e_1)e_2 \cup \partial_{\alpha p}(e_2) & \text{if } \varepsilon_\alpha(e_1) = \texttt{true}, \\ \partial_{\alpha p}(e_1)e_2, & \text{otherwise.} \end{cases} \\
\partial_{\alpha p}(e^\star) &\stackrel{\text{def}}{=} \partial_{\alpha p}(e)e^\star.
\end{aligned}$$

Example 25. *Let $\mathcal{B} = \{b_1, b_2\}$, $\Sigma = \{p, q\}$, and $e = b_1 p + q\overline{b_2}$. The partial derivative of e with respect to $b_1 b_2 p$ is the following:*

$$\begin{aligned}
\partial_{b_1 b_2 p}(e) &= \partial_{b_1 b_2 p}(b_1 p + q\overline{b_2}) \\
&= \partial_{b_1 b_2 p}(b_1 p) \cup \partial_{b_1 b_2 p}(q\overline{b_2}) \\
&= \partial_{b_1 b_2 p}(b_1)p \cup \partial_{b_1 b_2 p}(p) \cup \partial_{b_1 b_2}(q)\overline{b_2} \\
&= \partial_{b_1 b_2 p}(b_1)p \cup \partial_{b_1 b_2 p}(p) \\
&= \partial_{b_1 b_2 p}(p) \\
&= \{1\}.
\end{aligned}$$

Partial derivatives of KAT terms can be inductively extended to words $w \in (\mathsf{At} \cdot \Sigma)^\star$ in the following way:

$$\begin{aligned}
\partial_\epsilon(e) &\stackrel{\text{def}}{=} \{e\} \\
\partial_{w\alpha p}(e) &\stackrel{\text{def}}{=} \partial_{\alpha p}(\partial_w(e)),
\end{aligned}$$

where ϵ is the empty word . The set of all partial derivatives of a KAT term is the set

$$\partial_{(\mathsf{At}\cdot\Sigma)^\star}(e) \stackrel{\text{def}}{=} \bigcup_{w \in (\mathsf{At}\cdot\Sigma)^\star} \{e' \mid e' \in \partial_w(e)\}.$$

Example 26. *Let $\mathcal{B} = \{b_1, b_2\}$, $\Sigma = \{p, q\}$, and $e = b_1 p(b_1 + b_2)q$. The partial derivative of*

e with respect to the sequence $b_1b_2p\overline{b_1}b_2q$ *is the following:*

$$
\begin{aligned}
\partial_{b_1b_2p\overline{b_1}b_2q}(e) &= \partial_{b_1b_2p\overline{b_1}b_2q}(b_1p(b_1+b_2)q)\\
&= \partial_{\overline{b_1}b_2q}(\partial_{b_1b_2p}(b_1p(b_1+b_2)q))\\
&= \partial_{\overline{b_1}b_2q}(\partial_{b_1b_2p}(b_1)(p(b_1+b_2)q)\cup\partial_{b_1b_2p}(p(b_1+b_2)q))\\
&= \partial_{\overline{b_1}b_2q}(\partial_{b_1b_2p}(b_1)(p(b_1+b_2)q))\cup\partial_{\overline{b_1}b_2q}(\partial_{b_1b_2p}(p(b_1+b_2)q))\\
&= \partial_{\overline{b_1}b_2q}(\partial_{b_1b_2p}(p)(b_1+b_2)q)\\
&= \partial_{\overline{b_1}b_2q}((b_1+b_2)q)\\
&= \partial_{\overline{b_1}b_2q}(b_1+b_2)q\cup\partial_{\overline{b_1}b_2q}(q)\\
&= \partial_{\overline{b_1}b_2q}(q)\\
&= \{1\}.
\end{aligned}
$$

Similarly to partial derivatives of regular expressions, the language of partial derivatives of KAT terms are the left-quotients, that is, for $w \in (\mathsf{At}\cdot\Sigma)^\star$ and for $\alpha p \in (\mathsf{At}\cdot\Sigma)$, the following equalities $\mathsf{G}(\partial_w(e)) = \mathcal{D}_w(\mathsf{G}(e))$ and $\mathsf{G}(\partial_{\alpha p}(e)) = \mathcal{D}_{\alpha p}(\mathsf{G}(e))$ hold.

The next excerpt of the Coq development shows the previous definitions and theorems.

```
Fixpoint nullable(e:kat)(α:atom) : bool :=
 match e with
 | kats p ⇒ false
 | katb b ⇒ evalT α b
 | katu e1 e2 ⇒ nullable e1 α || nullable e2 α
 | katc e1 e2 ⇒ nullable e1 α && nullable t2 α
 | katst e1 ⇒ true
 end.

Definition nullable_set(s:set kat)(a:atom) :=
   fold (fun x ⇒ orb (nullable x a)) s false.

Fixpoint pdrv(e:kat)(α:atom)(p:sy) : set kat :=
 match e with
 | kats p′ ⇒ match compare p′ p with
             | Eq ⇒ {1}
             | _ ⇒ ∅
             end
 | katb b ⇒ ∅
 | katu e1 e2 ⇒ pdrv e1 α p ∪ pdrv e2 α p
 | katc e1 e2 ⇒ if nullable e1 α then
                    (pdrv e1 α p) ⊙ e2 ∪ pdrv e2 α p
```

```
                  else
                     (pdrv e1 α p) ⊙ e2
 | katst e1 ⇒ (pdrv e1 α p) ⊙ (katst e1)
 end.
```

```
Theorem pdrv_correct : ∀ e α p, G(pdrv e α p) ∼ LQ (G(e)) α p.
```

```
Theorem wpdrv_correct : ∀ e w, G(wpdrv e w) ∼ LQw_alt (G(e)) w.
```

Finiteness of the Set of Partial Derivatives

Kozen showed [62] that the set of partial derivatives is finite by means of the closure properties of a sub-term relation over KAT terms. As we have seen in the previous chapter, in the case of regular expressions the same problem can be solved using Mirkin's *pre-bases* [73]. Here we extend this method to KAT terms. We obtain an upper bound on the number of partial derivatives that is bounded by the number of primitive programs of Σ, and not the number of sub-terms as in [62].

Definition 1. *Let e be a* KAT *term. The function $\pi(e)$ from* KAT *terms to sets of* KAT *terms is recursively defined as follows:*

$$\begin{aligned} \pi(t) &\stackrel{\text{def}}{=} \emptyset, \\ \pi(p) &\stackrel{\text{def}}{=} \{1\}, \\ \pi(e_1 + e_2) &\stackrel{\text{def}}{=} \pi(e_1) \cup \pi(e_2), \\ \pi(e_1 e_2) &\stackrel{\text{def}}{=} \pi(e_1)e_2 \cup \pi(e_2), \\ \pi(e^\star) &\stackrel{\text{def}}{=} \pi(e)e^\star. \end{aligned}$$

Example 27. *Let $\mathcal{B} = \{b_1, b_2\}$, $\Sigma = \{p, q\}$, and $e = b_1p(b_1 + b_2)q$. The set of* KAT *terms computed by $\pi(e)$ is the following:*

$$\begin{aligned} \pi(e) &= \pi(b_1p(b_1 + b_2)q) \\ &= \pi(b_1)p(b_1 + b_2)q \cup \pi(p(b_1 + b_2)q) \\ &= \pi(p(b_1 + b_2)q) \\ &= \pi(p)(b_1 + b_2)q \cup \pi((b_1 + b_2)q) \\ &= \{(b_1 + b_2)q\} \cup \pi(b_1 + b_2)q \cup \pi(q) \\ &= \{(b_1 + b_2)q\} \cup \pi(q) \\ &= \{1, (b_1 + b_2)q\}. \end{aligned}$$

Let $|e|_\Sigma$ be the measure that gives us the number of primitive programs in e, which is recursively defined as follows:

$$\begin{aligned}
|t|_\Sigma &\stackrel{\text{def}}{=} 0,\ \ t \in T,\\
|p|_\Sigma &\stackrel{\text{def}}{=} 1,\ \ p \in \Sigma,\\
|e_1 + e_2|_\Sigma &\stackrel{\text{def}}{=} |e_1|_\Sigma + |e_2|_\Sigma,\\
|e_1 e_2|_\Sigma &\stackrel{\text{def}}{=} |e_1|_\Sigma + |e_2|_\Sigma,\\
|e_1^\star|_\Sigma &\stackrel{\text{def}}{=} |e_1|_\Sigma.
\end{aligned}$$

We now show that this is an upper bound of $\pi(e)$, which requires a lemma stating that π is a closed operation on KAT terms.

Proposition 10. *Let e be a* KAT *term over the set of primitive tests $\mathcal{B}$ and the set of primitive programs Σ. Hence, it holds that*

$$\forall e\ e',\ e' \in \pi(e) \to \forall e'',\ e'' \in \pi(e') \to e'' \in \pi(e).$$

Lemma 5. *Let e be a* KAT *term over the set of primitive tests $\mathcal{B}$ and the set of primitive programs Σ. Hence, $|\pi(e)| \leq |e|_\Sigma$.*

Now let $\mathsf{KD}(e) \stackrel{\text{def}}{=} \{e\} \cup \pi(e)$, with e being a KAT term. It is easy to see that $|\mathsf{KD}(e)| \leq |e|_\Sigma + 1$, since $|\pi(e)|_\Sigma \leq |e|_\Sigma$. We will now show that $\mathsf{KD}(e)$ establishes an upper bound on the number of partial derivatives of e. For that, we prove that $\mathsf{KD}(e)$ contains all the partial derivatives of e. First we prove that the partial derivative $\partial_{\alpha p}(e)$ is a subset of $\pi(e)$, for all $\alpha \in \mathsf{At}$ and $p \in \Sigma$. Next, we prove that if $e' \in \partial_{\alpha p}(e)$ then $\pi(e')$ is a subset of $\pi(e)$, which allow us to prove, by induction on the length of a word $w \in (\mathsf{At} \cdot \Sigma)^\star$ that all the derivatives of e are members of $\mathsf{KD}(e)$.

Lemma 6. *Let e be a* KAT *term, and let $\alpha p \in (\mathsf{At} \cdot \Sigma)$. Hence, if the* KAT *term e' is a member of $\pi(e)$, then $\partial_{\alpha p}(e') \subseteq \pi(e)$.*

Theorem 2. *Let e be a* KAT *term, and let $w \in (\mathsf{At} \cdot \Sigma)^\star$. Thus, $\partial_w(e) \subseteq \mathsf{KD}(e)$.*

In the code excerpt below we present the definition of π and of KD, as well as the proof of the cardinality of the set of all the partial derivatives of a KAT term.

```
Fixpoint PI (e:kat) : set kat :=
 match e with
 | katb b ⇒ ∅
 | kats _ ⇒ {katb ba1}
 | katu x y ⇒ (PI x) ∪ (PI y)
 | katc x y ⇒ (PI x)⊙ y ∪ (PI y)
```

```
 | katst x ⇒ (PI x) ⊙ (katst x)
 end.
Notation "π(x)" := (PI x).

Definition KD(r:kat) := {r} ∪ (XI r).

Fixpoint sylen (e:kat) : nat :=
 match e with
 | katb _ ⇒ 0
 | kats _ ⇒ 1
 | katu x y ⇒ sylen x + sylen y
 | katc x y ⇒ sylen x + sylen y
 | katst x ⇒ sylen x
 end.
Notation "|e|_Σ" := (sylen e).

Theorem KD_upper_bound : ∀ e, cardinal (KD e) ≤ (sylen e) + 1.

Theorem all_wpdrv_in_KD : ∀ w x r, x ∈ (wpdrv e w) → x ∈ KD(r).
```

We finish this section by establishing a comparison between our method for establishing the finiteness of partial derivatives with respect to the one introduced by Kozen [62]. The method introduced by Kozen establishes an upper bound on the number of derivatives of a KAT term considering its number of subterms. In particular, Kozen establishes that, given a KAT term e, the number of derivatives is upper bounded by $|e| + 1$ elements, where here $|e|$ denotes the number of subterms of e given by a closure $cl(e)$. This upper bound is larger than the one we obtain using our definition of $\mathsf{KD}(e)$. For instance, for the KAT term used in Example 27 the upper bound given by $cl(e)$ is

$$|\{b_1p(b_1 + b_2)q, b_1, p(b_1 + b_2)q, p, b_1 + b_2, b_2, q\}| + 1$$

which is considerably larger that the upper bounded that we obtain with the application of $\mathsf{KD}(e)$.

4.4 A Procedure for KAT Terms Equivalence

In this section we introduce a procedure for deciding KAT terms equivalence that is based on the notion of partial derivative. This procedure is the natural extension of the procedure EQUIVP described in the last chapter. Most of the structures of both the development on regular expressions and this one overlap and, for this reason, we will mainly focus on the details where the difference between the two developments are most notorious.

4.4.1 The Procedure EQUIVKAT

The kind of reasoning that takes us from partial derivatives of KAT terms into solving their (in)equivalence is very similar to the one we have followed with respect to regular expresion (in)equivalence. Given a KAT term e we know that

$$e \sim \mathsf{E}(e) \cup \left(\bigcup_{\alpha p \in (\mathsf{At} \cdot \Sigma)} \alpha p \partial_{\alpha p}(e) \right).$$

Therefore, if e_1 and e_2 are KAT terms, we can reformulate the equivalence $e_1 \sim e_2$ as

$$\mathsf{E}(e_1) \cup \left(\bigcup_{\alpha p \in (\mathsf{At} \cdot \Sigma)} \alpha p \partial_{\alpha p}(e_1) \right) \sim \mathsf{E}(e_2) \cup \left(\bigcup_{\alpha p \in (\mathsf{At} \cdot \Sigma)} \alpha p \partial_{\alpha p}(e_2) \right),$$

which is tantamount at checking if

$$\forall \alpha \in \mathsf{At},\ \varepsilon_\alpha(e_1) = \varepsilon_\alpha(e_2)$$

and

$$\forall \alpha p \in (\mathsf{At} \cdot \Sigma),\ \partial_{\alpha p}(e_1) \sim \partial_{\alpha p}(e_2).$$

Since we know that to check if a guarded string x is a member of the language denoted by some KAT term e we need to prove that the derivative of e with respect to x must be nullable, we can finitely iterate over the previous equations and reduce the (in)equivalence of e_1 and e_2 to one of the next equivalences:

$$e_1 \sim e_2 \leftrightarrow \forall \alpha \in \mathsf{At}, \forall w \in (\mathsf{At} \cdot \Sigma)^\star, \varepsilon_\alpha(\partial_w(e_1)) = \varepsilon_\alpha(\partial_w(e_2)) \tag{4.11}$$

and

$$(\exists\, w\, \exists\, \alpha,\ \varepsilon_\alpha(\partial_w(e_1)) \neq \varepsilon_\alpha(\partial_w(e_2))) \leftrightarrow e_1 \not\sim e_2. \tag{4.12}$$

The terminating decision procedure EQUIVKAT, presented in Algorithm 2, describes the computational interpretation of the equivalences (4.11) and (4.12). This procedure corresponds to the iterated process of deciding the equivalence of their partial derivatives.

The computational behaviour of EQUIVKAT is very similar to the behaviour of EQUIVP, described in the previous chapter. Both are iterative processes that decide (in)equivalences by testing the (in)equivalence of the corresponding partial derivatives.

Clearly, EQUIVKAT is computationally more expensive than EQUIVP : the code in lines 4 to 8 performs $2^{|\mathcal{B}|}$ comparisons to determine if the components of the derivative (Γ, Δ) are equi-nullable or not, whereas EQUIVP performs one single operation to determine equi-nullability; the code in lines 10 to 15 performs $2^{|\mathcal{B}|}|\Sigma|$ derivations, while in the case of EQUIVP only $|\Sigma|$ derivations are calculated. In the next section we describe the implementation of EQUIVKAT in COQ.

Algorithm 2 The procedure EQUIVKAT.

Require: $S = \{(\{e_1\}, \{e_2\})\}$, $H = \emptyset$
Ensure: `true` or `false`

```
 1: procedure EquivKAT(S, H)
 2:     while S ≠ ∅ do
 3:         (Γ, Δ) ← POP(S)
 4:         for α ∈ At do
 5:             if ε_α(Γ) ≠ ε_α(Δ) then
 6:                 return false
 7:             end if
 8:         end for
 9:         H ← H ∪ {(Γ, Δ)}
10:         for αp ∈ (At · Σ) do
11:             (Λ, Θ) ← ∂_αp(Γ, Δ)
12:             if (Λ, Θ) ∉ H then
13:                 S ← S ∪ {(Λ, Θ)}
14:             end if
15:         end for
16:     end while
17: return true
18: end procedure
```

We finish this section by providing two examples that describe the course of values produced by EQUIVKAT, one for the equivalence of KAT terms, and another for the case of inequivalence.

Example 28. *Let $\mathcal{B} = \{b\}$ and let $\Sigma = \{p\}$. Suppose we want to prove that $e_1 = (pb)^\star p$ and $e_2 = p(bp)^\star$ are equivalent. Considering $s_0 = (\{(pb)^\star p\}, \{p(bp)^\star\})$, it is enough to show that* EQUIVKAT$(\{s_0\}, \emptyset) =$ `true`. *The first step of the computation generates the two new following pairs of derivatives:*

$$\partial_{bp}(e_1, e_2) = (\{1, b(pb)^\star\}, \{(bp)^\star\}),$$
$$\partial_{\overline{b}p}(e_1, e_2) = (\{1, b(pb)^\star\}, \{(bp)^\star\}).$$

Since the new pairs are the same, only one of them is added to the working set S, and the original pair (e_1, e_2) is added to the historic set H. Hence, in the next iteration of EQUIVKAT *considers $S = \{s_1\}$, with $s_1 = (\{1, b(pb)^\star\}, \{(bp)^\star\})$, and $H = \{s_0\}$. Once again,*

new derivatives are calculated and they are the following:

$$\partial_{bp}(\{1, b(pb)^\star\}, \{(bp)^\star\}) = (\{b(pb)^\star\}, \{(bp)^\star\}),$$
$$\partial_{\overline{b}p}(\{1, b(pb)^\star\}, \{(bp)^\star\}) = (\emptyset, \emptyset).$$

The next iteration of the procedure will have $S = \{s_2, s_3\}$ and $H = \{s_0, s_1\}$, considering that $s_2 = (\{b(pb)^\star\}, \{(bp)^\star\})$ and $s_3 = (\emptyset, \emptyset)$. Since the derivative of s_2 is either s_2 or s_3 and since the same holds for the derivatives of s_3, the procedure will terminate in two iterations with $S = \emptyset$ and $H = \{s_0, s_1, s_2, s_3\}$. Hence, we conclude that $e_1 \sim e_2$.

Example 29. *Suppose that now we are interested in checking if $e_1 = (\overline{b}p)^\star b$ and $e_2 = b(p\overline{b})^\star$ are not equivalent. Like the previous example, the procedure starts by calculating the derivatives with respect to the elements bp and $\overline{b}p$, which yields*

$$\partial_{bp}(e_1, e_2) = (\emptyset, \{\overline{b}(p\overline{b})^\star\}),$$
$$\partial_{\overline{b}p}(e_1, e_2) = (\{(\overline{b}p)^\star b\}, \emptyset).$$

The new pairs of derivatives computed are enough for the procedure to conclude that $e_1 \not\sim e_2$. This is because $\varepsilon_{\overline{b}}(\partial_{bp}(e_1)) \neq \varepsilon_{\overline{b}}(\partial_{bp}(e_2))$, and also because $\varepsilon_b(\partial_{\overline{b}p}(e_1)) \neq \varepsilon_b(\partial_{\overline{b}p}(e_2))$, meaning that in the next recursive call of EQUIVP *a check for the equi-nullability between the components of the selected pair will yield non-equivalence.*

4.4.2 Implementation, Correctness and Completeness

In this section we provide the details of the implementation of EQUIVKAT in the COQ proof assistant. This implementation follows along the lines of the implementation of the decision procedure for deciding regular expression equivalence presented along Section 3.3.2.

Pairs of KAT Derivatives

As in the case of regular expressions, the main data structure used in EQUIVKAT is the one of pairs of derivatives of KAT terms, and we define a similar dependent record to implement them. The differences are the expected ones: in the case of EQUIVKAT, the derivative dp is now a pair of set of KAT terms, w is a word from $(\mathsf{At} \cdot \Sigma)^\star$, and cw is a parameter holding a proof that $dp = (\partial_w(\Gamma), \partial_w(\Delta))$.

```
Record Drv (e1 e2:kat) := mkDrv {
 dp :> set kat * set kat ;
 w : list AtSy ;
 cw : dp = (∂w(e1),∂w(e2))
}.
```

The derivation and nullability functions were also adapted to the type `Drv`. As an example, we present the extended derivation functions below. The type `AtSy` used in the definition of `Drv_pdrv_set` represents values $\alpha p \in (\mathsf{At} \cdot \Sigma)$.

```
Definition Drv_pdrv (x:Drv e1 e2)(α:atom)(p:sy) : Drv e1 e2.
Proof.
 refine(match x with mkDrv k w p ⇒
          mkDrv e1 e2 (pdrvp k a s) (w++[(α,p)]) _
        end).
(* ... *)
Defined.

Definition Drv_wpdrv (w:list AtSy) : ReW e1 e2.
Proof.
 refine(Build_Drv e1 e2 (wpdrvp ({e1},{e2}) w) w _).
(* ... *)
Defined.

Definition Drv_pdrv_set(s:Drv e1 e2)(sig:set AtSy) : set (Drv e1 e2) :=
 fold (fun x:AtSy ⇒ add (Drv_pdrv s (fst x) (snd x))) sig ∅.
```

Implementation of EQUIVKAT in Coq

As in the case of the definitions of the derivation operations on `Drv` terms, the extended nullability functions were enriched with atoms.

```
Definition nullable_p(x:set kat * set kat)(a:atom) :=
   eqb (nullable_set (fst x) a) (nullable_set (snd x) a).

Definition nullable_at_set(x:set kat * set kat)(ats:set atom) :=
   fold (fun p ⇒ andb (nullable_p x p)) ats true.

Definition nullableDrv(x:Drv e1 e2)(a:set atom) := nullable_at_set x a.

Definition nullableDrv_set(s:set (Drv e1 e2)(a:set atom) :=
  fold (fun p ⇒ andb (nullableDrv a p)) s true.

Definition newDrvSet(x:Drv e1 e2)(h:set (Drv e1 e2))
                         (sig:set (atom*z)) : set (Drv e1 e2) :=
   filter (fun x ⇒ negb (x ∈ h)) (Drv_pdrv_set x sig).
```

We now describe the implementation of Algorithm 2. First, we encode the function `step`, presented below, that implements one step of the algorithm's loop. This function makes $2^{|\mathcal{B}|}$ checks for equi-nullability of the derivative taken from the working set S. The function `step` returns a value of type `step_case` that indicates how the whole decision procedure must proceed. The function `step` corresponds to the code from lines 3 to 15 of Algorithm 2.

```
Inductive step_case (e1 e2:kat) : Type :=
|proceed : step_case e1 e2
|termtrue : set (Drv e1 e2) → step_case e1 e2
|termfalse : Drv e1 e2 → step_case e1 e2.

Definition step(h s:set (Drv e1 e2))(ats:set atom) (atsig:set (atom*Z)) :
 ((set (Drv e1 e2) * set (Drv e1 e2)) * step_case e1 e2) :=
 match choose s with
 |None ⇒ ((h,s),termtrue e2 e1 h)
 |Some (de1,de2) ⇒
   if nullableDrv e1 e2 (de1,de2) ats then
    let h' := add (de1,de2) h in
     let rsd' := in
      let s' := newDrvSet e1 e2 (de1,de2) H' atsig in
        (h',s' ∪ (s \ {(de1,de2)}),proceed e1 e2)
   else
    ((h,s),termfalse e1 e2 (de1,de2))
 end.
```

Next we encode the iterator representing the main loop of Algorithm 2. We use a modified version of the type `DP` presented in Section 3.3.2, to accomodate a proof that all the derivatives considered in the accumulator set are equi-nullable. The function `iterate` recursively calls the function `step` until a termination state is reached. Moreover, `iterate` is implemented using the well-founded relation `LLim` that was the one used in the last chapter to implement the iterator for the decision procedure for regular expressions (in)equivalence.

```
Inductive term_cases (e1 e2:kat) : Type :=
|Ok : set (Drv e1 e2) → term_cases e1 e2
|NotOk : Drv e1 e2 → term_cases e1 e2.

Inductive DP (e1 e2:kat)(h s:set (Drv e1 e2))(ats:set atom) : Prop :=
|is_dp : h ∩ s === ∅ →
             nullableDrv_set e1 e2 h ats = true → DP e1 e2 h s ats.

Lemma DP_upd : ∀ (e1 e2:kat)(h s : set (Drv e1 e2))
```

```
                      (ats:set atom)(atsig : set (atom*Z)),
   DP e1 e2 H S →
    DP e1 e2 (fst (fst (step e1 e2 h s ats atsig)))
              (snd (fst (step e1 e2 h s ats atsig))).

Lemma DP_wf : ∀ (e1 e2:kat)(h s : set (Drv e1 e2))(ats:set atom)
                    (atsig : set (atom*Z)),
  DP e1 e2 h s ats → snd (step e1 e2 h s ats atsig) = proceed e1 e2 →
   LLim e1 e2 (fst (fst (step e1 e2 h s ats atsig))) h.

Function iterate(e1 e2:kat)(h s:set (Drv e1 e2))(ats:set atom)
                         (atsig:set (atom*Z)) (D:DP e1 e2 h s ats)
                         {wf (LLim e1 e2) h}: term_cases e1 e2 :=
  let ((h',s'),next) := step e1 e2 h s ats atsig in
   match next with
   |termfalse x ⇒ NotOk e1 e2 x
   |termtrue h ⇒ Ok e1 e2 h
   |proceed ⇒ iterate e1 e2 h' s' sig ats (DP_upd e1 e2 h s ats atsig D)
  end.
Proof.
 (* Proof obligation 1 : proof that LLim is a decreasing measure for iterate *)
 abstract(apply DP_wf).
 (* Proof obligation 2: proof that LLim is a well founded relation. *)
 exact(guard e1 e2 100 (LLim_wf e1 e2)).
Defined.
```

The function `equivkat_aux` that we present below lifts the result of `iterate` to Boolean values. The function `equivkat` finishes the encoding of the full EQUIVKAT procedure, and is simply a call to `equivkat_aux` with the correct values for the working set S and the accumulator set H, as described in Algorithm 2.

```
Definition equivkat_aux(e1 e2:kat)(h s:set (Drv e1 e2))
                              (ats:set atom)(atsig:set (atom*Z))
                              (D:DP e1 e2 h s ats) : bool :=
    let h' := iterate e1 e2 h s ats atsig D in
      match h' with
      | Ok _ ⇒ true
      | NotOk _ ⇒ false
      end.

Definition mkDP_1st(ats:set atom) : DP e1 e2 ∅ {Drv_1st e1 e2} ats.
```

```
Definition equivkat(e1 e2:kat) : bool :=
  equivkat_aux e1 e2 ∅ {Drv_1st e1 e2} At (At*(setSy e1 ∪ setSy e2))
                 (mkDP_1st e1 e2 At).
```

Correctness and Completeness

The correctness and completeness proofs for `equivkat` follow the same steps of the proofs that we have devised for `equivP`. Here we recall these ideas for `equivkat` and establish their formal definitions, and give the corresponding theorems. In what follows, we will be assuming the following: the variables e_1 and e_2 denote the KAT terms that are under consideration for (in)equivalence; the variables d_{e_1} and d_{e_2} denote a derivative of $\{e_1\}$ and $\{e_2\}$, respectively; the variables S, S', H, and H' denote sets of pairs of sets of `Drv` e_1 e_2 terms; finally, $\mathcal{B}$ and Σ denote the set of primitive tests and the set of primitive programs, respectively.

Correctness. The proof of the correctness of `equivkat` proceedes as follows. We start by defining an invariant over `iterate` that allows us to conclude KAT term equivalence by proving that all the partial derivatives are computed and also that they are equi-nullable. This invariant is given by the definition of `invK_final` below.

```
Definition invK(H S:set (Drv e1 e2))(At:set atom)(atsig:set (atom*Z)) :=
 ∀ d, d ∈ H → ∀ p, p ∈ atsig →
    (Drv_pdrv e1 e2 d (fst p) (snd p)) ∈ (H ∪ S).

Definition invK_final(H S:set (Drv e1 e2))(At:set atom)(atsig:set (atom*Z))
   :=
 (Drv_1st e1 e2) ∈ (H ∪ S) ∧
  (∀  d, d ∈ (H ∪ S) → nullable_Drv_set e1 e2 d At = true) ∧
    invK H S At atsig.
```

By the recursive behaviour of `iterate` and by the definition of `step` we prove that `invK_final` leads to KAT term equivalence. First we prove that a successful computation of `iterate` yields a set containing all the partial derivatives of e_1 and e_2. For that, consider the following propositions and Corollary 2.

Proposition 11. *If* `invK` H S *holds, and if*

$$\texttt{step}\ e_1\ e_2\ H\ S\ \mathsf{At}\ (\mathsf{At}\times\Sigma) = ((H',S'),\texttt{proceed}\ e_1\ e_2)$$

also holds, then `invK` H' S' At $(\mathsf{At}\times\Sigma)$.

Proposition 12. *Let* D *be a term of type* `DP` e_1 e_2 S H At*. If* `invK` H S At $(\mathsf{At}\times\Sigma)$ *holds, and if*

$$\texttt{iterate}\ e_1\ e_2\ H\ S\ \mathsf{At}\ (\mathsf{At}\times\Sigma)\ D = \texttt{Ok}\ e_1\ e_2\ H',$$

then $\mathtt{invK}\ H'\ \emptyset\ \mathsf{At}\ (\mathsf{At} \times \Sigma)$ *also holds.*

Proposition 13. *Let* D *be a value of type* $\mathtt{DP}\ e_1\ e_2\ H\ S\ \mathsf{At}$. *If it is the case that*

$$\mathtt{iterate}\ e_1\ e_2\ H\ S\ \mathsf{At}\ (\mathsf{At} \times \Sigma)\ D = \mathtt{Ok}\ e_1\ e_2\ H'$$

holds, then $H \subseteq H'$.

Corollary 2. *Let* D *be a value of type* $\mathtt{DP}\ e_1\ e_2\ H\ S\ \mathsf{At}$. *If*

$$\mathtt{iterate}\ e_1\ e_2\ H\ S\ \mathsf{At}\ (\mathsf{At} \times \Sigma)\ D = \mathtt{Ok}\ e_1\ e_2\ H',$$

and if $\mathtt{choose}\ S = \mathtt{Some}\ (d_{e_1}, d_{e_2})$ *holds, then it is the case that* $\{(d_{e_1}, d_{e_2})\} \cup H \subseteq H'$.

Proposition 14. *Let* D *be a value of type* $\mathtt{DP}\ e_1\ e_2\ \emptyset\ \{(\{e_1\},\{e_2\})\}\ \mathsf{At}$. *Hence,*

$$\mathtt{iterate}\ e_1\ e_2\ \emptyset\ \{(\{e_1\},\{e_2\})\}\ \mathsf{At}\ (\mathsf{At} \times \Sigma)\ D = \mathtt{Ok}\ e_1\ e_2\ H' \to (\{e_1\},\{e_2\}) \in H'.$$

We proceed by showing that all the elements of H' in the value `Ok` $e_1\ e_2\ H'$ enjoy equi-nullability. This is straightforward, due to the parameter D in the definition of `iterate`. Recall that the definition of D explicitly contains a proof that all the elements in the given set are equi-nullable.

Proposition 15. *Let* D *be a value of type* $\mathtt{DP}\ e_1\ e_2\ H\ S\ \mathsf{At}$. *If*

$$\mathtt{iterate}\ e_1\ e_2\ H\ S\ \mathsf{At}\ (\mathsf{At} \times \Sigma)\ D = \mathtt{Ok}\ e_1\ e_2\ H',$$

then $\forall \alpha \in \mathsf{At}, \forall \gamma \in H',\ \varepsilon_\alpha(\gamma) = \mathtt{true}$ *holds.*

Using Proposition 14 and Proposition 15 we can establish the intermediate result that will take us to prove the correctness of `equivkat` with respect to language equivalence.

Proposition 16. *Let* D *be a value of type* $\mathtt{DP}\ e_1\ e_2\ H\ S\ \mathsf{At}$. *If*

$$\mathtt{iterate}\ e_1\ e_2\ H\ S\ \mathsf{At}\ (\mathsf{At} \times \Sigma)\ D = \mathtt{Ok}\ e_1\ e_2\ H',$$

then $\mathtt{invK_final}\ e_1\ e_2\ H'\ \emptyset\ \mathsf{At}\ (\mathsf{At} \times \Sigma)$ *holds.*

The last intermediate logical condition that we need to establish is that `inv_final` implies KAT terms equivalence, considering that it is instantiated with the correct parameters. The following lemma gives us exactly that.

Lemma 7. *If* $\mathtt{inv_final}\ e_1\ e_2\ H'\ \emptyset\ \mathsf{At}\ (\mathsf{At} \times \Sigma)$ *holds, then it is true that* $e_1 \sim e_2$.

Finally, we can state the theorem that ensures that if `equivkat` returns `true`, then we have the equivalence of the KAT terms under consideration.

Theorem 3. *Let e_1 and e_2 be two* KAT *terms. Hence,*

$$\texttt{equivkat}\ e_1\ e_2 = \texttt{true} \rightarrow e_1 \sim e_2.$$

Completeness. The proof of the completeness of `equivkat` goes as follows. First we show that if `equivkat` finds a term d of type `Drv` e_1 e_2 such that $\varepsilon_\alpha(d)$ is false for some atom $\alpha \in \mathsf{At}$, then there exists a guarded string x that is not simultaneously a member of $\mathsf{G}(e_1)$ and of $\mathsf{G}(e_2)$. Naturally, this conclusion implies $e_1 \not\sim e_2$.

Proposition 17. *Let D be a term of type* `DP` e_1 e_2 S H At*. If*

$$\texttt{iterate}\ e_1\ e_2\ S\ H\ \mathsf{At}\ (\mathsf{At} \times \Sigma)\ D = \texttt{NotOk}\ e_1\ e_2\ (d_{e_1}, d_{e_2})$$

holds, then $\varepsilon_\alpha(d_{e_1}) \neq \varepsilon_\alpha(d_{e_2})$ also holds for some $\alpha \in \mathsf{At}$.

Proposition 18. *Let D be a term of type* `DP` e_1 e_2 S H At*. Hence, it holds that*

$$\texttt{iterate}\ e_1\ e_2\ S\ H\ \mathsf{At}\ (\mathsf{At} \times \Sigma)\ D = \texttt{NotOk}\ e_1\ e_2\ (d_{e_1}, d_{e_2}) \rightarrow e_1 \not\sim e_2.$$

Lemma 8. *If* `equivkat` *returns with the value of* `false` *then e_1 and e_2 are inequivalent* KAT *terms, i.e.,*

$$\texttt{equivkat}\ e_1\ e_2 = \texttt{false} \ \rightarrow\ e_1 \not\sim e_2.$$

Performance and Usability of EQUIVKAT

As pointed out earlier, the performance of EQUIVKAT is not expected to be so efficient as EQUIVP. The algorithm contains two exponential computations, one for checking if a derivative is equi-nullable and another to compute the set of new derivatives. Currently we have not yet found a way to improve on these performances. To the best of our knowledge we are only aware of the work of Worthington [101] who presents an efficient conversion from KAT terms into regular expressions and decides the equivalence of these regular expression using Kozen's procedure based in automata. The formalisation of this approach requires working with matrices, which is out of the scope of our work since we base ourselves in syntactical operations.

In its current state of development, our procedure can be used to automatically decide (in)equivalence involving small KAT terms. Such small terms occur very frequently in proofs of KAT equations and our procedure can be used to help on finishing such proofs, by automatically solving eventual sub-goals. Moreover, and due to the capability of extraction of COQ, we can obtain a program that is correct by construction and that can be used outside COQ's environment and exhibit better performances.

4.5 Application to Program Verification

As we have written earlier, KAT is suited to several verification tasks, as proved by several reported experiments. In this section we try to motivate the reader to the reasons that lead us to formalise KAT in Coq: to have a completely correct development which can serve as a certified environment to build proofs of the correctness and equivalence of simple imperative programs.

We begin this section by introducing some examples borrowed from [59] that show how KAT can be useful to prove the equivalence between certain classes of simple imperative programs. We consider the imperative language introduced in Section 2.2. Afterwards, we show how KAT and Hoare logic are related by PHL and how deductive reasoning in PHL reduces to equational reasoning in KAT. We also present some motivating examples about the usefulness of KAT to the partial correctness of programs.

Equivalence of Programs Through KAT

Recall from Sections 4.1 and 4.2 that the terms of KAT are regular expressions enriched with Boolean tests. The addition of tests gives extra expressivity to KAT terms when compared to regular expressions because they allow us to represent imperative program constructions such as conditionals and while loops. Since KAT is propositional, it does not allow to express assignments or other first order constructions. Nevertheless, and under an adequate encoding of the first order constructions at the propositional level, we can encode programs as KAT terms.

Recall the simple imperative programs IMP, introduced in Chapter 2. An important remark is that assignments of IMP are not directly represented in KAT because KAT is propositional, and the same applies to the Boolean expressions in IMP. Hence, in all the definitions, properties, and examples that follow, we will be assuming that IMP assignments and Boolean expressions are represented by primitive programs and tests, respectively. Under these assumptions, if e_1 and e_2 are terms encoding the IMP programs C_1 and C_2, and if the Boolean test t is the encoding of the IMP Boolean expression B, then we can encode sequence, and conditional instructions and while loops in KAT as follows:

$$C_1; C_2 \stackrel{\text{def}}{=} e_1 e_2,$$
$$\textsf{if } B \textsf{ then } C_1 \textsf{ else } C_2 \textsf{ fi} \stackrel{\text{def}}{=} (te_1 + \bar{t}e_2),$$
$$\textsf{while } B \textsf{ do } C_1 \textsf{ end} \stackrel{\text{def}}{=} (te_1)^\star \bar{t}.$$

We now present a set of examples that illustrate how we can address program equivalence in KAT. We begin by an example that we can show that while loops and do-loops are equivalent for certain programs using these constructions. A do-loop is defined in KAT as the term

$p(bp)^\star\overline{b}$, that is, it always begin with a computation of the body of the do-loop. We also consider $\texttt{skip} \stackrel{\text{def}}{=} 1$.

Example 30. *Let $\mathcal{B} = \{b\}$ and $\Sigma = \{p\}$ be the set of primitive tests and set of primitive programs, respectively, and let P_1 and P_2 be the programs defined as follows:*

$$P_1 \stackrel{\text{def}}{=} \mathsf{do\ if}\ B\ \mathsf{then}\ C\ \mathsf{else\ skip\ fi\ while}\ (B),$$
$$P_2 \stackrel{\text{def}}{=} \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{end}.$$

Considering that $B = b$ and that $C = p$, we encode the programs P_1 and P_2 as KAT *terms as follows:*

$$e_1 \stackrel{\text{def}}{=} (bp + \overline{b})(b(bp + \overline{b}))^\star\overline{b},$$
$$e_2 \stackrel{\text{def}}{=} (bp)^\star\overline{b}.$$

The procedure ***equivkat*** *decides the equivalence $e_1 \sim e_2$ in 0.028 seconds.*

An example similar to the previous one is presented by Kozen and Patron [63], in the context of the certification of compiler optimisations using KAT.

Example 31. *One of the possible optimisations established for compilers is* loop unrolling*, in order to try to reduce the number of tests and jumps executed by a loop. If we have a loop*

$$P_1 \stackrel{\text{def}}{=} \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{end},$$

unrolling it produced the equivalent program

$$P_2 \stackrel{\text{def}}{=} \mathsf{while}\ B\ \mathsf{do}\ \ C\,;\ \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else\ skip\ fi\ end}.$$

With $B = b$ and $C = p$, the programs P_1 and P_2 are encoded as KAT *terms $e_1 \stackrel{\text{def}}{=} (bp)^\star\overline{b}$ and $e_2 \stackrel{\text{def}}{=} (bp(bp+\overline{b}))^\star\overline{b}$, respectively. Our procedure is able to decide the equivalence $e_1 \sim e_2$ in 0.058 seconds. On the contrary, the equational proof presented in [63] is considerably long despite the fact that the* KAT *terms under consideration are relatively simple.*

The next example shows how we can use extra variables and commutativity assumption in KAT to prove the equivalence of programs. The original program of the example is first rewritten manually and then automatically solved by equivkat.

Example 32. *Let $\mathcal{B} = \{b\}$ and $\Sigma = \{p, q, r\}$ be the set of primitive tests and set of primitive programs, respectively. Let $C_1 = p$, $C_2 = q$, $C_3 = r$, $B = b$, and let P_1 and P_2 be the following two programs:*

$$P_1 \stackrel{\text{def}}{=} \mathsf{if}\ B\ \mathsf{then}\ C_1; C_2\ \mathsf{else}\ C_1; C_3\ \mathsf{fi}$$
$$P_2 \stackrel{\text{def}}{=} C_1; \mathsf{if}\ B\ \mathsf{then}\ C_2\ \mathsf{else}\ C_3\ \mathsf{fi}$$

These programs are encoded in KAT *as the terms* $e_1 = (bpq + \overline{b}pr)$ *and* $e_2 = p(bq + \overline{b}r)$, *respectively. Note that the program* P_2 *is expected to be equivalent to program* P_1 *only if the value of the test* b *is preserved by execution of* C_1, *which we do not know. Hence, we must introduce extra information on both programs without compromising their meaning. We proceed by considering a new test* c *that remembers the effect of* p *in* c, i.e., *such that the commutativity equality* $pc = cp$ *holds and replace the test* b *in the subterm* $(bq + \overline{b}r)$ *by* c. *Then we add at the top of both programs the two possible interactions between the tests* b *and* c, *which corresponds to the term* $bc + \overline{b}\overline{c}$. *The resulting modified* KAT *terms are*

$$e_3 = (bc + \overline{b}\overline{c})(bpq + \overline{b}pr) \ \ \textit{and} \ \ e_4 = (bc + \overline{b}\overline{c})p(cq + \overline{c}r),$$

respectively. Using equivkat *to try to solve the equivalence* $e_3 \sim e_4$ *does not work yet. The solution is to first propagate the program* p *in* e_4 *and then use the commutativity condition* $pc = cp$ *to obtain the new term* $e_4' = (bc + \overline{b}\overline{c})(cpq + \overline{c}pr)$. *The procedure now proves the equivalence* $e_3 \sim e_4'$ *in 0.06 seconds.*

The final example concerns with proving that programs with two while loops have an equivalent program with just a single loop.

Example 33. *Let* $\mathcal{B} = \{b, c\}$ *and* $\Sigma = \{p, q\}$ *be the set of primitive tests and set of primitive programs, respectively, and let* P_1 *and* P_2 *be the following two programs:*

$$P_1 \stackrel{\text{def}}{=} \textsf{while}\ B\ \textsf{do}\ C_1; \textsf{while}\ B'\ \textsf{do}\ C_2\ \textsf{end end}$$

$$P_2 \stackrel{\text{def}}{=} \textsf{if}\ B\ \textsf{then}\ C_1; \textsf{while}\ B + B'\ \textsf{do if}\ B'\ \textsf{then}\ C_2\ \textsf{else}\ C_1\ \textsf{fi end else skip fi}$$

Let $C_1 = p$, $C_2 = q$, $B = b$ *and* $B' = c$. *The programs* P_1 *and* P_2 *are encoded in* KAT *as*

$$e_1 = (bp((cq)^{\star}\overline{c}))^{\star}\overline{b} \ \ \textit{and} \ \ e_2 = bp((b + c)(cq + \overline{c}p))^{\star}\overline{(b + c)} + \overline{b},$$

respectively. The procedure decides the equivalence $e_1 \sim e_2$ *in 0.053 seconds.*

Hoare Logic and KAT

Hoare logic, already introduced in Section 2.2 uses triples of the form $\{P\}\ C\ \{Q\}$, where P is the precondition and Q is the postcondition of the program C. The meaning of these triples, called Hoare triples or PCAs, is the following: *if* P *holds when* C *starts executing then* Q *will necessarily hold when* C *terminates, if that is the case.* Hoare logic consists of a set of inference rules which we can successively apply to triples in order to prove the partial correctness of the underlying program.

PHL is a weaker Hoare logic that does not come equipped with the assignment inference rule, since it is a propositional logic. KAT subsumes PHL and therefore the inference rules of PHL can be encoded as KAT theorems. This implies that deductive reasoning within PHL

proof system reduces to equational reasoning in KAT. In KAT Hoare triples $\{P\}\ C\ \{Q\}$ are expressed by the KAT equations $te = tet'$, or equivalently by $te\overline{t'} = 0$, or by $te \leq t'$, such that $t, t' \in T$ and e is a KAT term. The inference rules of PHL are encoded as follows:

Sequence:

$$t_1e_1 = t_1e_1t_2 \wedge t_2e_2 = t_2e_2t_3 \rightarrow t_1e_1e_2 = t_1e_1e_2t_3, \tag{4.13}$$

Conditional:

$$t_1t_2e_1 = t_1t_2e_1t_3 \wedge \bar{t_1}t_2e_2 = \bar{t_1}t_2e_2t_3 \rightarrow t_2(t_1e_1 + \bar{t_1}e_2) = t_2(t_1e_1 + \bar{t_1}e_2)t_3, \tag{4.14}$$

While-loop:

$$t_1t_2e = t_1t_2et_2 \rightarrow t_2(t_1e)^*\bar{t_1} = t_2(t_1e)^*\bar{t_1}\bar{t_1}t_2. \tag{4.15}$$

Weakening:

$$t_1 \leq t_1' \wedge t_1e = t_1et_2 \wedge t_2 \leq t_2' \rightarrow t_1'e = t_1'et_2' \tag{4.16}$$

The equations (4.13), (4.14), (4.15), and (4.16) (which were mechanically checked in our development) correspond, respectively, to the deductive rules (HL-Seq), (HL-If), (HL-While), and (HL-Weak), and were already introduced in Chapter 2. No assignment rule is considered here because PHL is propositional. Our decision procedure may be of little or no help here since we need to reason under sets of equational hypothesis, and we need to use them in a way that cannot be fully automated [59]. However, derivable PHL rules of the form

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \cdots \{P_n\}\ C_n\ \{Q_n\}}{\{P\}\ C\ \{Q\}} \tag{4.17}$$

correspond to KAT equational implications

$$t_1p_1\overline{t_1'} = 0 \wedge \ldots \wedge t_np_n\overline{t_n'} = 0 \wedge \ldots \wedge t_{n+1} \leq t_{n+1}' \wedge \ldots\ \ t_{n+m} \leq t_{n+m}' \rightarrow te\overline{t'} = 0. \tag{4.18}$$

It has been shown in [43] that for all KAT terms $r_1, \ldots, r_n, e_1, e_2$, over a set of primitive programs $\Sigma = \{p_1, \ldots, p_m\}$, the equational implication of the form

$$r_1 = 0 \wedge \ldots \wedge r_n = 0 \rightarrow e_1 = e_2$$

is a theorem of KAT if and only if

$$e_1 + uru = e_2 + uru, \tag{4.19}$$

where $u \stackrel{\text{def}}{=} (p_1 + \ldots + p_m)^\star$ and $r \stackrel{\text{def}}{=} r_1 + \ldots + r_n$. At this point our decision procedure can be used to decide $e_1 \sim e_2$. In order to infer the set of hypothesis that is required to obtain the previous equation, we need annotated programs whose language was introduced

in Section 2.2. Recall that the corresponding encoding of such programs as KAT terms is defined by

$$C_1; \{P\}\ C_2 \stackrel{\text{def}}{=} e_1 t e_2,$$
$$\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi} \stackrel{\text{def}}{=} (te_1 + \bar{t}e_2),$$
$$\text{while } B \text{ do } \{I\}\ C \text{ end} \stackrel{\text{def}}{=} (tie)^\star \bar{t},$$

with C, C_1, and C_2 encoded by the KAT terms e, e_1, and e_2, and with P and I encoded by the tests t and i, respectively. Given an IMP program we can obtain its annotated counterpart using a VCGen algorithm such as the one presented by Frade and Pinto [39] or, at the level of KAT only, using the algorithm proposed by Almeida *et. al.* [6].

We will now present two examples that show the usage of KAT in verifying the partial correctness of simple imperative programs. Once more, the abstraction of the first order program constructs is performed manually, by mapping assignments to primitive programs, and Boolean assertions to primitive tests.

Example 34. *Consider the program* Sum3 $\stackrel{\text{def}}{=} y := x; y := x + x + y$. *The program* Sum3 *computes the triple of the value assigned to the variable* x. *The Hoare triple specifying the correctness of such property is* $\{\text{true}\}$ Sum3 $\{y = 3x\}$ *and in order to prove this triple partially correct, we proceed by first annotating* Sum3 *as described in the table below. We name* AnnSum3 *the program* Sum3 *with annotations. Moreover, considering the set of primitive tests* $\mathcal{B} = \{b_1, b_2, b_3\}$ *and the set of primitive programs* $\Sigma = \{p_1, p_2\}$, *we assign primitive tests and primitive programs to the annotations and instructions, respectively.*

AnnSum3	*Encoding*
$\{x + x + x = 3y\}$	b_1
$y := x;$	p_1
$\{x + x + y = 3x\}$	b_2
$y := x + x + y$	p_2

Next, we establish the hypotheses of the form $tp\overline{t'} = 0$ *so that we can produce the* KAT *equivalence to which we can apply our decision procedure. Considering* $b_0 \stackrel{\text{def}}{=} \text{true}$ *and* $b_3 \stackrel{\text{def}}{=} y = 3x$, *the set of hypotheses is the following:*

$$\Gamma = \{b_0 \leq b_1, b_1 p_1 \overline{b_2} = 0, b_2 p_2 \overline{b_3} = 0\}.$$

The triple $\{\text{true}\}$ AnnSum3 $\{y = 3x\}$ *is encoded as a* KAT *equation as follows:*

$$b_0 b_1 p_1 b_2 p_2 \overline{b_3} = 0,$$

from where we can define $u = (p_1 + p_2)^\star$ *and* $r = b_0\overline{b_1} + b_1 p_1 \overline{b_2} + b_2 p_2 \overline{b_3}$. *We can now apply the decision procedure to the term*

$$b_0 b_3 p_1 b_1 p_2 \overline{b_2} + uru = 0 + uru$$

and obtain the expected equivalence in 0.912 seconds.

Example 35. *Let us consider the following program:*

$$\mathsf{Fact} \stackrel{\text{def}}{=} y := 1; z := 0; \mathsf{while}\, \neg(z = x)\, \mathsf{do}\;\; z := z + 1; y := y * z\; \mathsf{end}.$$

The program Fact *computes the factorial of the value given by the variable x. The specification we wish to prove partially correct is thus* $\{\mathsf{true}\}\, \mathsf{Fact}\, \{y = x!\}$. *As in the previous example, the table below presents the annotated version of* Fact *and the corresponding encoding of instructions and Boolean conditions into primitive programs in* $\mathcal{B} = \{b_0, b_1, b_2, b_3, b_4, b_5\}$ *and tests in* $\Sigma = \{p_1, p_2, p_3, p_4\}$.

AnnFact	*Encoding*
y := 1	p_1
$\{y = 0!\}$	b_1
z := 0 ;	p_2
$\{y = z!\}$	b_2
while $\neg(z = x)$ do	b_3
$\{y = z!\}$	b_2
z := z + 1 ;	p_3
$\{y \times z = z!\}$	b_4
y := y * z ;	p_4
end	

Assuming that $b_0 \stackrel{\text{def}}{=} \mathsf{true}$ *and that* $b_5 \stackrel{\text{def}}{=} y = x!$, *the Hoare triple* $\{\mathsf{true}\}\, \mathsf{AnnFact}\, \{y = x!\}$ *is encoded as the equality*

$$b_0 p_1 b_1 p_2 b_2 (b_3 b_2 p_3 b_4 p_4)^\star \overline{b_3 b_5} = 0. \tag{4.20}$$

To prove (4.20) *we need to obtain a set of hypotesis, that can be obtained in a backward fashion [6] using a* weakest precondition generator. *The set of hypothesis for* (4.20) *is the following:*

$$\Gamma = \{b_0 p_1 \overline{b_1} = 0, b_1 p_2 \overline{b_2} = 0, b_3 b_2 p_3 \overline{b_4} = 0, b_4 p_2 \overline{b_2} = 0, b_2 \overline{b_3 b_5} = 0\}.$$

With Γ *and* $\Sigma = \{p_1, p_2, p_3, p_4\}$, *we know that*

- $u = (p_1 + p_2 + p_3 + p_4)^\star$;
- $r = b_0 p_1 \overline{b_1} + b_1 p_2 \overline{b_2} + b_3 b_2 p_3 \overline{b_4} + b_4 p_2 \overline{b_2} + b_2 \overline{b_3 b_5}$.

The equation

$$b_0 p_1 b_1 p_2 b_2 (b_3 b_2 p_3 b_4 p_4)^\star \overline{b_3 b_5} + uru = 0 + uru$$

is provable by the decision procedure in 22 seconds.

4.6 Related Work

Although KAT can be applied to several verification tasks, there exists few tool support for it. Kozen and Aboul-Hosn [1] developed the KAT-ML proof assistant. KAT-ML allows one to reason about KAT terms. It also provides support to reason with assignments and other first-order programming constructs, since the underlying theory of KAT-ML is *Schematic Kleene algebra with tests* (SKAT), an extension of KAT with a notion of assignment, characterised by an additional set of axioms. However, KAT-ML provides no automation.

Recently, Almeida *et. al.* [7, 6] presented a new development of a decision procedure for KAT equivalence. The implementation was made using the OCaml programming language, is not mechanically certified, but includes a new method for proving the partial correctness of programs that dispenses the burden of constructing the terms r and u introduced in the previous section.

Finally, the work that is more related to ours is the one of Pous [87]. This work extends the previous work of the author in the automation of Kleene algebra in Coq which was already discussed in the previous chapter. The author decides KAT term via a procedure based on partial derivatives like we do. The decision procedure suffers from the same exponential behaviour on the size of terms involved and no completeness proof is given for the latter algorithm. However, this development provides a tactic that automatically solves KAT equations with hypothesis of the kind of the ones used for proving the partial correctness of programs that we have shown in the previous section. Moreover, the development contains a completeness proof of KAT following along the lines of the work of Kozen and Smith [64].

4.7 Conclusions

In this chapter we have presented a decision procedure for KAT terms equivalence. We have described its implementation in Coq, as well has its proofs of correctness and completeness with respect to the language theoretic model of KAT. The decision procedure is based on the notion of partial derivative of KAT terms, and a new way of calculating their finiteness based on the method introduced by Mirkin is presented.

Although KAT works at the propositional level, it still can be used as a framework to perform several verifications tasks, namely, program equivalence and partial correctness of programs. However, in such approaches, we must provide the necessary abstractions of the first order constructions as new tests and primitive actions and, some times, consider extra commutativity conditions over these abstractions. Moreover, verification tasks of this kind must still rely on external tools that must ensure that the first order constructions considered are valid.

In terms of future work, we consider the mechanisation of a new algorithm that decides KAT terms introduced by Almeida *et. at.* [7, 6]. This new method refines the one presented in the previous section, in the sense that it dispenses the creation of the KAT terms r and u that are required to automate the proof of partial correctness of imperative programs encoded as KAT terms. Moreover, we are also interested in extending our development in order to support SKAT, which we believe that it will approximate the usage of KAT to a more realistic notion of program verification, since at the level of SKAT we have access to first order constructions in programs.

Chapter 5

Mechanised Rely-Guarantee in Coq

In the previous two chapters we have addressed the mechanisation of terminating and correct decision procedures for regular expressions and KAT terms. Both theories and decision procedures can effectively be used to perform formal verification of sequential programs. The next step in this line of research is to consider further extensions of regular expressions and KAT terms to address other programming paradigms such as concurrent programming, parallel programming, or event real-time system programming. Of particular interest is parallel programming, since new advances in computer technology are pushing this paradigm as a future standard (consider, for instance, the efforts being done to mature *cloud computing*).

The most recent development of the family of regular expressions with respect to concurrent and parallel programming is Hoare's *et. al. concurrent Kleene algebra* (CKA) [46, 47]. This algebraic system is based in *modal Kleene algebra* (MKA) [33, 74, 32], an extension of KA with *domain operators*. CKA provides a language with operators that are able to specify concurrent composition with or without dependencies between the involved programs, as well as the operators for sequential programming as in regular expressions. Extensions to tests are not considered in the current stage of CKA. The underlying model of CKA is the one of execution traces on which dependencies can be specified. In terms of decidable decision procedures, no results have been presented so far.

In order to be able to mechanise CKA in Coq in the future (possibly with new models as well as decision procedures) we have decided to gather knowledge and some experience with concurrency and parallelism through a mechanisation of one of targets of CKA, namely, Jones' *rely-guarantee* (RG) [53]. RG is one of the well established formal approaches to the verification of shared-variable parallel programs. In particular, our study and mechanisation follows very closely the work described by Coleman and Jones in [27]. We note that RG was already addressed in terms of its encoding within a proof-assistant through the work of Nieto [78], who mechanised a parametric version of RG, and whose proofs follow the ones previously introduced by Xu *et. al.* [102].

Therefore, in this chapter we describe what we believe to be the first effort to provide a complete formalisation of RG for the COQ proof assistant. We encode the programming language, its small-step operational semantics, a Hoare-like inference system, and a mechanised proof of its soundness with respect to operational semantics. In particular, we allow nested parallelism, whereas the work of Nieto [78] does not.

It is also important to stress that RG is the base for more recent formal systems that address concurrency and parallelism. Such works result from a synergy between RG and Reynolds' *separation logic* [90] and include RGSEP by Vafeiadis *et. al.* [99], *local rely-guarantee reasoning* by Feng *et. al.* [35], and also *deny-guarantee* by Dodds *et. al* [34]. Although none of the cited works are addressed in this thesis, it is our conviction that the contribution we give with our mechanisation can be a guide for the mechanisation of the cited formal systems in the future, within the COQ proof assistant. The development is available online in [76].

5.1 Rely-Guarantee Reasoning

The first formal system that addressed the specification and verification of parallel programs was the one developed by Owiki and Gries [81]. In their approach, a sequential proof had to be carried out for each parallel process, which also had to incorporate information that established that each sequential proof does not interfere with the other sequential proofs. This makes the whole proof system non-compositional, as it depends on the information of the actual implementation details of the sequential processes. The inference rule for this approach is summarised as follows:

$$\frac{\{P_1\}C_1\{Q_1\} \qquad \{P_2\}C_2\{Q_2\} \qquad \begin{array}{c}C_1 \text{ does not interfere with } C_2\\ C_2 \text{ does not interfere with } C_1\end{array}}{\{P_1 \wedge P_2\}\,\mathsf{par}\; C_1 \;\mathsf{with}\; C_2 \;\mathsf{end}\{Q_1 \wedge Q_2\}}$$

Based on the previous system, Jones introduced RG in his PhD thesis [53], which resulted in a formal approach to shared-variable parallelism that brings the details of interference into specification, in an explicit way. In RG, besides preconditions and postconditions, specifications are enriched with *rely conditions* and *guarantee conditions*: a rely condition describes the effects of the steps of execution of the environment; a guarantee condition describes the effects of the steps of execution of the program. Therefore, in the context of parallel program specification and design, the rely condition describes the level of interference that the program is able to tolerate from the environment, whereas the guarantee condition describes the level of interference that the program imposes on the environment. From the specification point-of-view, the rely condition can be seen as a way of requiring the developer to make the necessary assumptions about the environment in which the program is going to

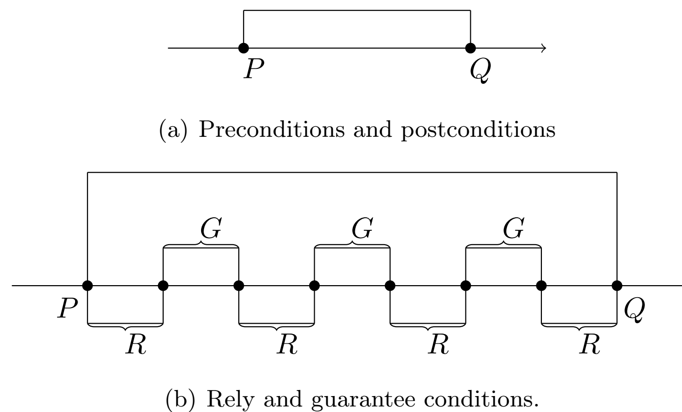
execute. From the user's point-of-view, it is his responsibility to ensure that the environment complies with the previous assumptions.

As a final remark, note that when the program's computation is described by rely and guarantee conditions, which are to be decomposed during the proof construction process, the result of such decomposition can only have at least the same level of interference as their parent conditions, that is, they cannot produce more interference. Still, and from the logical point of view, these decompositions may be weakened or strengthen, but they still must comply with the conditions from which they have originated. The point here is that weakening or strengthening decomposed rely and guarantee conditions may allow to establish a larger number of environments where the complete parallel program may be deployed.

Preconditions and Postconditions *v.s.* Rely and Guarantee Conditions

The difference between rely and guarantee conditions and preconditions and postconditions can be stated in the following way: preconditions and postconditions view the complete execution of the underlying program as a whole, whereas rely and guarantee conditions analyse each possible step of the execution, either resulting from the interference of the environment, or by a step of computation of the program. This is captured graphically in Figure 5.1, borrowed from Coleman and Jones [27].

Figure 5.1: Rely and guarantee *vs.* preconditions and postconditions.



(a) Preconditions and postconditions

(b) Rely and guarantee conditions.

Considering Figure 5.1, the soundness of a parallel program C with respect to precondition P and post condition Q, and also with respect with the rely condition R and guarantee condition G can be described informally as follows: if $P(s)$ holds for some state s, and if all the interference is bound by R, and all the intermediate computations of C are bounded by G then, if the program C terminates, it must do so in a state s' such that $Q(s')$ holds. This notion will be formally given further ahead in this chapter. Before finishing this section, we present a simple parallel program and characterize it in terms of its correctness.

Example 36. *Consider the program C, defined as follows:*

$$C \stackrel{\text{def}}{=} \mathsf{par}\ x := x + 1\ \mathsf{with}\ x := x + 2\ \mathsf{end}.$$

Informally, the program increments the variable x, in two possible ways: in one way, the program updates the variable x by 1, and then by 2; in the other way, the program first updates the variable x by 2, and then by 1.

Considering the behaviour of the program C, we can prove that if x is initially greater or equal than 0, then x will be greater or equal to 2 after C terminates. For that, we must define the adequate precondition and postcondition, $P \stackrel{\text{def}}{=} \lambda s.[\![x]\!]_{\mathbb{N}}(s) \geq 0$ and $Q \stackrel{\text{def}}{=} \lambda s.[\![x]\!]_{\mathbb{N}}(s) \geq 2$, respectively.

We also need to establish the rely and guarantee conditions that allow us to conclude the correctness of C. In this case, it is enough to consider both conditions as relations that assert that the value of x never decreases, that is, $R \stackrel{\text{def}}{=} \lambda s \lambda s'.[\![x]\!]_{\mathbb{N}}(s) \leq [\![x]\!]_{\mathbb{N}}(s')$ and G defined in the same way.

With the above definitions, it is straightforward to conclude that the postcondition always hold (namely, the program C always terminates in this case), and also that the computation of the subprograms $x := x + 1$ and $x := x + 2$ trivialy satisfy both R and G.

In Section 5.7 we recover the example above and give a formal proof of its validity with respect to the axiomatic semantics that we will introduce in Section 5.5.

5.2 The IMPp Programming Language

IMPp is a simple parallel imperative programming language that extends IMP, which was already introduced in Chapter 2. The IMPp language extends IMP by introducing an instruction for the atomic execution of programs, and also one instruction for parallel execution of programs. Moreover, it also considers lists of natural numbers as part of the datatypes primitively supported.

As in IMP, the language IMPp considers a language of arithmetic expressions and a language of Boolean expressions. We denote these languages of expressions by AExp and BExp, respectively, and we inductively define them by the following grammars:

$$\mathsf{AExp} \ni E, E_1, E_2 \ ::= \ x \mid n \in \mathbb{N} \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2,$$

$$\mathsf{BExp} \ni B, B_1, B_2 \ ::= \mathsf{true} \mid \mathsf{false} \mid \neg B \mid B_1 \wedge B_2 \mid E_1 = E_2 \mid E_1 < E_2,$$

where x is a variable identifier. The language of IMPp programs is inductively defined by

$$\begin{array}{rcl} \mathsf{IMPp} \ni C, C_1, C_2 & ::= & \mathsf{skip} \\ & | & x := E \\ & | & \mathsf{atomic}(C) \\ & | & C_1; C_2 \\ & | & \mathsf{if}\ B\ \mathtt{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \\ & | & \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{done} \\ & | & \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, \end{array}$$

where x is a variable, $E \in \mathsf{AExp}$, and $B \in \mathsf{BExpr}$. The program C' that is the argument of the atomic instruction must be a sequential program built using the grammar of IMP programs. Given the following definition of variable identifiers (borrowed from Pierce's *et. al* [85]),

```
Inductive id : Type := Id : nat → id.
```

the syntax of IMPp is defined in Coq as follows:

```
Inductive aexp : Type :=
| ANum : nat → aexp
| AId : id → aexp
| APlus : aexp → aexp → aexp
| AMinus : aexp → aexp → aexp
| AMult : aexp → aexp → aexp.

Inductive bexp : Type :=
| BTrue : bexp
| BFalse : bexp
| BEq : aexp → aexp → bexp
| BLt : aexp → aexp → bexp
| BNot : bexp → bexp
| BAnd : bexp → bexp → bexp.

Inductive stmt : Type :=
| Stmt_Skip : stmt
| Stmt_Ass : id → aexp → stmt
| Stmt_Seq : stmt → stmt → stmt
| Stmt_If : bexp → stmt → stmt → stmt
| Stmt_While : bexp → stmt → stmt
| Stmt_Atom : stmt → stmt
| Stmt_Par : stmt → stmt → stmt.
```

```
Notation "'skip'" := Stmt_Skip.
Notation "x ':=' e" := (Stmt_Ass x e).
Notation "C1 ; C2" := (Stmt_Seq C1 C2).
Notation "'while' b 'do' C 'end'" := (Stmt_While b C).
Notation "'if' b 'then' C1 'else' C2 'fi'" := (Stmt_If b C1 C2).
Notation "'par' C1 'with' C2 'end'" := (Stmt_Par C1 C2).
Notation "'atomic(' C ')'" := (Stmt_Atom C).
```

5.3 Operational Semantics of IMPp

IMPp programs are evaluated by means of a small-step operational semantics, in the style of Plotkin's *structural operational semantics* [86]. The semantics must be small-step in order to capture a fine-grained interleaving between the computation of the program under consideration, and the interference caused by other parallel processes running in the environment. Formally, the semantics of IMPp is a relation

$$\stackrel{c}{\Longrightarrow} : \langle \mathsf{IMPp}, \Sigma\rangle \to \langle \mathsf{IMPp}, \Sigma\rangle \tag{5.1}$$

between pairs $\langle C, s\rangle$, called configurations, such that C is a IMPp program, and s is a state (set of mappings of variables to values). The set of all states is denoted by Σ. The type of values that are supported by the semantics, and the notion of state (and the particular case of the empty state) are defined in Coq as follows:

```
Definition val := nat.

Definition st := id → val.

Definition empty_st : st := fun _ ⇒ 0.
```

Naturally, a configuration $\langle C, s\rangle$ has the type (stmt*st). Moreover, we define the operation of updating a variable in a state in the following way:

```
Definition upd (s : st) (x:id) (e : val) : st :=
  fun x':id ⇒ if beq_id x x' then x else s x'.
```

Before giving the structure of the relation $\stackrel{c}{\Longrightarrow}$ we describe the interpretation of arithmetic and Boolean expressions.We can define a recursive function that evaluates arithmetic expressions into their final result in type val. Such a function is defined as follows:

```
Function aeval (s : st) (e : aexp) {struct e} : val :=
  match e with
  | ANum n ⇒ n
```

```
  | AId x ⇒ s x
  | APlus e1 e2 ⇒ aeval s e1 + aeval s e2
  | AMinus e1 e2 ⇒ aeval s e1 - aeval s e2
  | AMult e1 e2 ⇒ aeval s e1 * aeval s e2
  end.
Notation "⟦e⟧𝔼(s)" := (aeval s e).
```

The same approach is taken for Boolean expressions.

```
Function beval (s : st) (b : bexp){struct b} : bool :=
 match b with
 | BTrue ⇒ true
 | BFalse ⇒ false
 | BEq e1 e2 ⇒ beq_nat (aeval s e1) (aeval s e2)
 | BLt e1 e2 ⇒ blt_nat (aeval s e1) (aeval s e2)
 | BNot b1 ⇒ negb (beval s b1)
 | BAnd b1 b2 ⇒ andb (beval s b1) (beval s b2)
 end.
Notation "⟦b⟧𝔹(s)" := (beval s b).
```

Given the interpretation functions for arithmetic and Boolean expressions, we can now describe the relation that captures the computation of an IMPp program, C starting in some state s. The relation $\stackrel{c}{\Longrightarrow}$ is inductively as follows:

$$\frac{}{\langle x := E, s\rangle \stackrel{c}{\Longrightarrow} \langle \mathsf{skip}, s[\llbracket E\rrbracket_{\mathbb{E}}/x]\rangle}\ (\textsc{Assgn})$$

$$\frac{\langle C, s\rangle \stackrel{\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle}{\langle \mathsf{atomic}(C), s\rangle \stackrel{c}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle}\ (\textsc{Atomic})$$

$$\frac{}{\langle \mathsf{skip}; C, s\rangle \stackrel{c}{\Longrightarrow} \langle C, s\rangle}\ (\textsc{Seq-1})$$

$$\frac{\langle C_1, s\rangle \stackrel{c}{\Longrightarrow} \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \stackrel{c}{\Longrightarrow} \langle C_1'; C_2, s'\rangle}\ (\textsc{Seq-2})$$

$$\frac{\llbracket B\rrbracket_{\mathbb{B}}(s) = \mathsf{true}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rangle \stackrel{c}{\Longrightarrow} \langle C_1, s\rangle}\ (\textsc{If-true})$$

$$\frac{\llbracket B\rrbracket_{\mathbb{B}}(s) = \mathsf{false}}{\langle \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\rangle \stackrel{c}{\Longrightarrow} \langle C_2, s\rangle}\ (\textsc{If-false})$$

$$\frac{}{\langle \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{done}, s\rangle \stackrel{c}{\Longrightarrow} \langle \mathsf{if}\ B\ \mathsf{then}\ C; (\mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{done})\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}, s\rangle}\ (\textsc{While})$$

$$\frac{\langle C_1, s\rangle \overset{c}{\Longrightarrow} \langle C_1', s'\rangle}{\langle \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{par}\ C_1'\ \mathsf{with}\ C_2\ \mathsf{end}, s'\rangle}\ (\text{PAR-1})$$

$$\frac{\langle C_2, s\rangle \overset{c}{\Longrightarrow} \langle C_2', s'\rangle}{\langle \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{par}\ C_1\ \mathsf{with}\ C_2'\ \mathsf{end}, s'\rangle}\ (\text{PAR-2})$$

$$\frac{}{\langle \mathsf{par\ skip\ with\ skip\ end}, s\rangle \overset{c}{\Longrightarrow} \langle \mathsf{skip}, s\rangle}\ (\text{PAR-END})$$

The above set of rules is encoded in Coq through the following inductive predicate:

```
Inductive cstep : (stmt * st) → (stmt * st) → Prop :=
|CS_Ass: ∀ s x e,
  cstep ((x := e),s) (skip,s [aeval s e/x])
|CS_Atom : ∀ C s s′,
   star _ (step) (C,s) (skip,s′) → cstep (atomic(C),s) (skip,s′)
|CS_SeqStep : ∀ s C1 C1′ s′ C2,
  cstep (C1,s) (C1′,s′) → cstep ((C1;C2),s) ((C1′;C2),s′)
|CS_SeqFinish : ∀ s C2,
  cstep ((skip;C2),s) (C2,s)
|CS_IfFalse : ∀ s C1 C2 b,
  ¬b2assrt b s → cstep (if b then C1 else C2 fi,s) (C2,s)
|CS_IfTrue : ∀ s C1 C2 b,
  b2assrt b s → cstep (if b then C1 else C2 fi,s) (C1,s)
|CS_While : ∀ s b C ,
  cstep (while b do C end,s) (if b then (C;while b do C end) else skip fi,s)
|CS_Par1 : ∀ s C1 C1′ C2 s′ ,
  cstep (C1,s) (C1′,s′) → cstep (par C1 with C2 end,s) (par C1′ with C2 end,s′)
|CS_Par2 : ∀ s C1 C2 C2′ s′,
  cstep (C2,s) (C2′,s′) → cstep (par C1 with C2 end,s) (par C1 with C2′ end,s′)
|CS_Par_end : ∀ s,
  cstep (par skip with skip end,s) (skip,s).

Infix "=c=>" := cstep.
```

The constructors that form the `cstep` correspond to the reduction rules presented before, in the exact same order of occurrence. In the example that follows, we show a reduction performed by applying the relation $\overset{c}{\Longrightarrow}$.

Example 37. *Let x_1 and x_2 be two variables. Let s be the state such that the values of x_1 and x_2 are 0. Let C be the* IMPp *program defined as follows:*

$$\mathsf{par}\ x_1 := 1\ \mathsf{with}\ x_2 := 2\ \mathsf{end}$$

*Two reductions may occur from $\langle C, s \rangle$: either it reduces by the rule (*PAR-1*) and updates s by mapping the value 1 to the variable x_1, that is,*

$$\langle \mathsf{par}\ x_1 := 1\ \mathsf{with}\ x_2 := 2\ \mathsf{end}, s \rangle \overset{c}{\Longrightarrow} \langle \mathsf{par}\ \mathsf{skip}\ \mathsf{with}\ x_2 := 2\ \mathsf{end}, s[1/x_1] \rangle,$$

*or it reduces by rule (*PAR-2*) and updates s by mapping 2 to the variable x_2, that is,*

$$\langle \mathsf{par}\ x_1 := 1\ \mathsf{with}\ x_2 := 2\ \mathsf{end}, s \rangle \overset{c}{\Longrightarrow} \langle \mathsf{par}\ x_1 := 1\ \mathsf{with}\ \mathsf{skip}\ \mathsf{end}, s[2/x_2] \rangle.$$

5.4 Reductions under Interference

The semantics of the relation $\overset{c}{\Longrightarrow}$ is not enough to capture interference from the environment since it assumes the programs run in isolation, even in the case of the parallel computation rules. In order to capture interference adequately, we need an extended notion of transition between configurations $\langle C, s \rangle$ that takes into account a possible preemption of C by an external program. If this is the case, then the resulting configuration must keep C unchanged, but the state s may be subject to an update, caused exactly by the interference of that external program's operation. Formally, we consider a new relation between configurations as follows

$$\langle C, s\rangle \overset{R}{\Longrightarrow} \langle C', s'\rangle \overset{\text{def}}{=} (\langle C, s\rangle \overset{c}{\Longrightarrow} \langle C', s'\rangle) \vee (C = C' \wedge (s, s') \in R), \tag{5.2}$$

such that R is a relation on states that determines if a state can change into another state by the interference of the environment. The relation (5.2) is encoded in Coq as follows:

```
Definition interf R :=
    fun cf cf′:stmt*st ⇒ (fst cf) = (fst cf′) ∧ R (snd cf) (snd cf′).

Definition prog_red R :=
    fun cf cf′:stmt*st ⇒ cf =c=> cf′ ∨ interf R cf cf′.
```

Example 38. *Let s be a state such that the variable x_1 has the value 1 and the variable x_2 has the value 1 also. Let R be the rely condition defined as $R \overset{\text{def}}{=} \{(x, x+1) \mid x \in \{x_1, x_2\}\}$, that is, that tolerates that the environment increases the value of a given variable by 1, and let C be the* IMPp *program defined as follows:*

$$x_1 := 1; x_2 := 2$$

The following are the two possible reductions of the configuration $\langle C, s \rangle$, considering interference:

$$\langle x_1 := 1; x_2 := 2, s\rangle \overset{R}{\Longrightarrow} \langle \mathsf{skip}; x_2 := 2, s[1/x_1]\rangle$$

or

$$\langle x_1 := 1; x_2 := 2, s\rangle \overset{R}{\Longrightarrow} \langle x_1 := 1; x_2 := 2, s[2/x_2]\rangle.$$

Now that we already have a definition for one step of computation of a program under interference, we extend it to a finite number of reductions, thus capturing the behaviour of computations. This is tantamount to the reflexive and transitive closure of $\overset{R}{\Longrightarrow}$, that is,

$$\langle C, s\rangle \overset{R^\star}{\Longrightarrow} \langle C', s'\rangle. \tag{5.3}$$

Obviously, we may also consider a predetermined number of computations, in order to analyse just a fixed number of steps of reduction under interference, instead of considering all the possible computations. In Coq we have the following definitions for fixed number of reductions, and also for any finite number of reductions:

```
Inductive starn (A:Type)(R:relation A) : nat → A → A → Prop :=
|starn_refl : ∀ x:A,
   starn A R 0 x x
|starn_tran : ∀ x y:A,
   R x y → ∀ (n:nat)(z:A), starn A R n y z → starn A R (S n) x z.

Definition prog_red_n R n :=
  fun cf cf' ⇒ starn (stmt*st) (prog_red R) n cf cf'.
Notation "cf =R^n=> cf'" := (prog_red_n R n cf cf').

Inductive star (A:Type)(R:relation A) : A → A → Prop :=
| star_refl : ∀ x:A, star A R x x
| star_trans : ∀ x y:A, R x y → ∀ z:A, star A R y z → star A R x z.

Definition prog_red_star R :=
  fun cf cf' ⇒ star (stmt*st) (prog_red R) cf cf'.
Notation "cf =R*=> cf'" := (prog_red_star R cf cf').
```

Example 39. *Let s be a state and let $C \stackrel{\text{def}}{=} x_1 := 1; x_2 := 2$ be the program under consideration. We are able to prove that, after four reductions, the computation of C leads to a state where the variable x_1 contains the value 2. This property is obtained by first performing three reductions leading the configuration $\langle\, C, s\,\rangle \overset{c^3}{\Longrightarrow} \langle\, \mathsf{skip}, s[1/x_1][2/x_2]\,\rangle$. With one more reduction, and because the relation R is defined as $(x, x+1) \in R$, we can prove $\langle\, \mathsf{skip}, s[1/x_1][2/x_2]\,\rangle \overset{R}{\Longrightarrow} \langle\, \mathsf{skip}, s[1/x_1][2/x_2][2/x_1]\,\rangle$.*

Respecting Guarantees

In RG, the role of the guarantee condition is to bound the amount of interference that a program may impose in the environment. In particular, the guarantee condition of a program is part of the rely condition of all the other programs running in parallel with it. Therefore,

if the configuration $\langle C, s \rangle$ reduces to a configuration $\langle C', s' \rangle$ after some finite number of steps, and if $\langle C', s' \rangle \overset{c}{\Longrightarrow} \langle C'', s'' \rangle$ holds, then we must show that $(s', s'') \in G$, where G is the established guarantee condition. Proving all such reductions that occur along the execution of C ensures that the complete computation of C satisfies the constraints imposed by G. The satisfaction of this property was introduced by Coleman and Jones in [27], and is formally defined by

$$\begin{array}{c}\mathsf{within}(R, G, C, s)\\ \overset{\mathrm{def}}{=}\\ \forall C' s',\ (\langle C, s\rangle \overset{R^\star}{\Longrightarrow} \langle C', s'\rangle) \to \forall C'' s'',\ (\langle C', s'\rangle \overset{c}{\Longrightarrow} \langle C'', s''\rangle) \to (s', s'') \in G.\end{array} \tag{5.4}$$

An important consequence of the previous definition is that given two programs C and C', we can prove that the states resulting from their parallel computation are members of the set of states that result from the reflexive and transitive closure of the rely and guarantee conditions of each other. Formally, and considering commands C_1, C_1', and C_2, and also considering states s and s', we have

$$(\langle\, \mathsf{par}\, C_1\, \mathsf{with}\, C_2\, \mathsf{end}, s\,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{par}\, C_1'\, \mathsf{with}\, C_2\, \mathsf{end}, s'\,\rangle) \to (s, s') \in (R \cup G_1)^\star, \tag{5.5}$$

$$(\langle\, \mathsf{par}\, C_1\, \mathsf{with}\, C_2\, \mathsf{end}, s\,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{par}\, C_1'\, \mathsf{with}\, C_2'\, \mathsf{end}, s'\,\rangle) \to (s, s') \in (R \cup G_2)^\star, \tag{5.6}$$

where G_1 and G_2 are, respectively, the guarantee conditions of the programs C_1 and C_2. These properties will be fundamental for proving the soundness of the parallel computation inference rule for the RG proof system HL-RG. Other properties of within are the following: if a reduction of the program exists, or one step of interference occurs, then within still holds, that is,

$$\forall s\ s',\ (\langle C, s\rangle \overset{c}{\Longrightarrow} \langle C', s'\rangle) \to \mathsf{within}(R, G, C, s) \to \mathsf{within}(R, G, C', s')$$

and

$$\forall s\, s',\ (s, s') \in R \to \forall C,\ \mathsf{within}(R, G, C, s) \to \mathsf{within}(R, G, C, s').$$

The previous properties are naturally extended to a finite set of reductions under interference starting from a configuration $\langle C, s\rangle$. Formally,

$$\forall s\ s',\ (\langle C, s\rangle \overset{R^\star}{\Longrightarrow} \langle C', s'\rangle) \to \mathsf{within}(R, G, C, s) \to \mathsf{within}(R, G, C', s'). \tag{5.7}$$

Another property of interest that we will need for the soundness proof of the parallel statement, is that if $\mathsf{within}(R, G, C, s)$ holds and if $\langle C, s\rangle$ reduces to $\langle C', s'\rangle$ then this reduction can be interpreted as a finite series of intermediate steps, where each step is captured either by R – meaning that the environment has interveen – or by G – meaning that a program reduction occurred. Formally, this notion is expressed as follows:

$$\forall s\ s',\ (\langle C, s\rangle \overset{R^\star}{\Longrightarrow} \langle C', s'\rangle) \to \mathsf{within}(R, G, C, s) \to (s, s') \in (R \cup G)^\star. \tag{5.8}$$

5.5 A Proof System for Rely-Guarantee

In this section we introduce an inference system for proving the partial correctness of IMPp programs along the lines of RG. This system, which we name HL-RG, extends sequential Hoare logic with a notion of interference that is explicit at the specification level. Let R be a relation establishing the rely condition, and let G be a relation establishing the guarantee condition. A triple in HL-RG has the form of

$$\{R, P\}\ C\ \{Q, G\},$$

and we shall write

$$\vdash\ \{R, P\}\ C\ \{Q, G\}$$

if we can prove from the set of inference rules of HL-RG that the program C is partially correct with respect to its rely and guarantee conditions, and also with respect to its preconditions and postconditions.

The soundness of HL-RG requires the notion of *Hoare validity*. In classic Hoare logic we state this condition as follows: if a program C starts its computation in a state where the precondition P holds then, if C terminates, it terminates in a state where the postcondition Q holds. In the case of parallel programs, this notion must be extended to comply with the rely and guarantee conditions. Thus, the validity of a specification $\{R, P\}\ C\ \{Q, G\}$, which we write $\models\ \{R, P\}\ C\ \{Q, G\}$, has the following reading: if a program C starts its computation in a state where the precondition P holds and if the interference of the environment is captured by the rely condition R then, if C terminates, it terminates in a state where the postcondition Q holds, and also all the intermediate program reduction steps satisfy the guarantee condition G.

Before presenting the inference system HL-RG, we must introduce the notion of *stability*. In RG, we say that an assertion P is *stable* with respect to the interference of the environment, captured by a relation R, if P remains invariant with respect to the interference caused by R, that is

$$\mathsf{stable}\ R\ P \stackrel{\text{def}}{=} \forall s, s' \in \Sigma,\ P(s) \to (s, s') \in R \to P(s'). \tag{5.9}$$

The particular effect of stability conditions on preconditions and postconditions can be described as follows: if P is a precondition of the program C that is satisfied, and if the environment R acts upon P, then P remains satisfied, allowing in this way to "move" the satisfiability of P into the state where the actual execution of C starts; the same applies to a postcondition Q, that is, if Q is stable with respect to R, then Q can me moved into the state where the program C has finished its execution. We now present the definition of stability in Coq. But for that, we introduce first the definitions of assertions.

```
Definition assrt := st → Prop.
```

```
Definition b2assrt (b:bexp) : assrt :=
  fun s:st ⇒ beval s b = true.

Definition assg_subs (x:id) (e:aexp) (Q:assrt) : assrt :=
  fun s:st ⇒ Q (s [(aeval s e)/x]).

Definition assrtT (b: bexp) : assrt :=
  fun s:st ⇒ b2assrt b s.

Definition assrtF (b: bexp) : assrt :=
  fun s:st ⇒ ¬b2assrt b s.

Definition assrtAnd (P Q:assrt) : assrt :=
  fun s:st ⇒ P s ∧ Q s.

Definition assrtOr (P Q: assrt) : assrt :=
  fun s:st ⇒ P s ∨ Q s.

Definition assrtImp (P Q: assrt) : Prop :=
  ∀ s:st, P s → Q s.
```

With the previous definitions, we can define the notion of stability in Coq, as follows:

```
Definition stable (R:relation st)(P:assrt) :=
  ∀ x y:st, P x ∧ R x y → P y.

Lemma stable_starn :
  ∀ n:nat, ∀ R P, stable R P → stable (starn _ R n) P.

Lemma stable_star :
  ∀ R P, stable R P → stable (star _ R) P.

Lemma stable_and :
  ∀ R1 R2 P, stable R1 P → stable R2 P → stable (rstAnd R1 R2) P.

Lemma stable_impl :
  ∀ R1 R2 P, stable R2 P → rstImp R1 R2 → stable R1 P.
```

We now give a simple example that shows an assertion being stable with respect to a possible rely condition.

Example 40. *Let R be a relation defined by $R \stackrel{\text{def}}{=} \{(x, x+k) \mid k > 0\}$, and let $P(s) \stackrel{\text{def}}{=} s > 0$.*

It is easy to see that the assertion P is stable with respect to R since if we know that $P(x)$ holds, then x must be a positive number, and due to the action of R, we obtain $P(x+k)$ which is also true.

Inference Rules

We will now describe each of the inference rules of the HL-RG inference system. This system extends sequential Hoare logic by adding two new rules, one for each of the commands that extends IMPp with respect to IMP. Moreover, the rules for the sequential part, as well as the rules for atomic execution and parallel execution of commands are enriched with the stability conditions required for the rules to be sound. In Coleman and Jone's presentation, such stability rules are implicit, but when conducting the development in a proof system like Coq, the stability conditions must be made explicit. We now introduce the inference rules of the HL-RG proof system.

Skip. In the case of the skip command, no program reductions exist. Thus, only the environment R can change the underlying state, and so, for the precondition P and the postcondition Q, the hypotheses must establish their stability with respect to R. The inference rule for skip is defined as follows:

$$\frac{\mathsf{stable}\ R\ P \qquad \mathsf{stable}\ R\ Q \qquad P \rightarrow Q}{\{R,P\}\ \mathsf{skip}\ \{Q,G\}}\ (\text{HG-Skip})$$

Assignment. In the case of assignment, the environment R may cause interference in the precondition P or the postcondition Q, but it does not affect the execution of the assignment. Moreover, it must be known in advance that the change in the state due the assignment must satisfy the guarantee condition G. The inference rule for the assignment is defined as follows:

$$\frac{\begin{array}{l}\mathsf{stable}\ R\ P \\ \mathsf{stable}\ R\ Q \qquad (\forall s \in \Sigma, (s, s[E/x]) \in G) \qquad P \rightarrow Q[E/x]\end{array}}{\{R,P\}\ x := E\ \{Q,G\}}\ (\text{HG-Assgn})$$

Sequence. In the case of the sequential composition of programs C_1 and C_2, we need to prove that C_1 behaves correctly with respect to its specification and, if that is the case, we have to prove that C_2 respects the same condition, considering that the postcondition of C_1 becomes the precondition of C_2. The inference rule for the composition of programs is defined as follows:

$$\frac{\{R,P\}\ C_1\ \{Q',G\} \qquad \{R,Q'\}\ C_2\ \{Q,G\}}{\{R,P\}\ C_1;C_2\ \{Q,G\}}\ (\text{HG-Seq})$$

Conditional choice. In the case of the conditional statement, as long as the specifications for the statements of the branches are given, we can prove the correct specification of the whole conditional. Still, the assertion stating the result of evaluating the Boolean guard must be immune to the interference of the environment. With the stability ensured, there is no risk that the interference of the environment breaks the expected flow of the execution of the program. The inference rule for the conditional choice command is defined as follows:

$$\frac{\begin{array}{ccc}\mathsf{stable}\ R\ [\![B]\!]_{\mathbb{B}} & & \{R, P \wedge B\}\ C_1\ \{Q, G\} \\ \mathsf{stable}\ R\ [\![\neg B]\!]_{\mathbb{B}} & \mathsf{stable}\ R\ P & \{R, P \wedge \neg B\}\ C_2\ \{Q, G\}\end{array}}{\{R, P\}\ \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}\ \{Q, G\}}\ (\text{HG-If})$$

Loops. In the case of while loops, the classic inference rule of Hoare logic is extended with stability conditions, in a similar way as in the conditional rule. The environment may interfere with the Boolean guard, and so stability must be ensured in order to preserve the correct evaluation of loops. The inference rule for the while loop is defined as follows:

$$\frac{\mathsf{stable}\ R\ [\![B]\!]_{\mathbb{B}} \qquad \mathsf{stable}\ R\ [\![\neg B]\!]_{\mathbb{B}} \qquad \{R, B \wedge P\}\ C\ \{P, G\}}{\{R, P\}\ \mathsf{while}\ B\ \mathsf{do}\ C\ \mathsf{done}\ \{\neg B \wedge P, G\}}\ (\text{HG-While})$$

Atomic execution. Atomic statement execution ensures that a given program executes with no interference of the environment whatsoever. Hence, the rely condition in this case is the identity relation, here denoted by ID. Moreover, the command C that is going to be executed atomically must be a valid sequential program, and the precondition and postcondition can still suffer interference from the environment, hence they must be proved stable with respect to the global rely condition R. The inference rule for the atomic execution of programs is defined as follows:

$$\frac{\begin{array}{cc}\mathsf{stable}\ R\ P & \\ \mathsf{stable}\ R\ Q & \{P\}\ C\ \{Q\}\end{array}}{\{R, P\}\ \mathsf{atomic}(C)\ \{Q, G\}}\ (\text{HG-Atomic})$$

Consequence. The consequence rule is just a simple extension of the consequence rule in sequential Hoare logic, where the rely and guarantee conditions R and G can be strengthened or weakened. The inference rule for the consequence is defined as follows:

$$\frac{\begin{array}{ccc}P \to P' & R \subseteq R' & \\ Q' \to Q & G' \subseteq G & \{R', P'\}\ C\ \{Q', G'\}\end{array}}{\{R, P\}\ C\ \{Q, G\}}\ (\text{HG-Conseq})$$

Parallel composition. In the case of parallel composition of two programs C_1 and C_2 we assume that the specifications of the individual programs ensure that they not interfere with

each other. Hence, the hypotheses must contain evidences that the guarantee condition of one of the component programs becomes part of the environment of the other component program, and vice versa. The adequate stability conditions for both the component programs are also required. The inference rule for the parallel composition of programs is defined as follows:

$$\frac{\begin{array}{ccc} (G_1 \cup G_2) \subseteq G & & \\ (R_l \cup G_1) \subseteq R_r & \mathsf{stable}\,(R_r \cup G_2)\,Q_1 & \\ (R_r \cup G_2) \subseteq R_l & \mathsf{stable}\,(R_l \cup G_1)\,Q_2 & \\ (R_l \cap R_r) \subseteq R & \mathsf{stable}\,(R_r \cup G_2)\,P & \{R_l, P\}\,C_1\,\{Q_1, G_1\} \\ (Q_l \wedge Q_r) \rightarrow Q & \mathsf{stable}\,(R_l \cup G_1)\,P & \{R_r, P\}\,C_2\,\{Q_2, G_2\} \end{array}}{\{R, P\}\ \mathsf{par}\,C_1\,\mathsf{with}\,C_2\,\mathsf{end}\ \{Q, G\}}\ (\text{HG-PAR})$$

In Coq, we define the inference system HL-RG by the following inductive predicate:

```
Inductive triple_rg (R G:StR) : assrt → stmt → assrt → Prop :=
| RSkip: ∀ (P Q:assrt),
   Reflexive G → stable R P → stable R Q → P[→]Q →
   triple_rg R G P skip Q

| RAsgn : ∀ v a P Q,
   stable R P → stable R Q → (∀  s, G s (upd s v (aeval s a))) →
   (∀  s, P s → Q (upd s v (aeval s a))) →
   triple_rg R G P (v := a) Q

| RAtom : ∀ P Q c b,
   (∀  x y, star _ G x y → G x y) →
   stable R P → stable R Q → triple G P c Q →
   triple_rg R G P (atomic c end) Q

| RIf: ∀ P c1 c2 Q b,
   Reflexive G →
   stable R (assrtT b) → stable R (assrtF b) →
   stable R P →
   triple_rg R G (assrtAnd (assrtT b) P) c1 Q →
   triple_rg R G (assrtAnd (assrtT b) P) c2 Q →
   triple_rg R G P (ifb b then c1 else c2 fi) Q

| RSequence: ∀ c1 c2 P K Q,
   Reflexive G →
   triple_rg R G P c1 K → triple_rg R G K c2 Q →
```

```
    triple_rg R G P (c1;c2) Q

| RConseq: ∀ R' G' P P' Q Q' c,
    assrtImp P P' → assrtImp Q' Q →
    rstImp R R' → rstImp G' G →
    triple_rg R' G' P' c Q' →
    triple_rg R G P c Q

| RLoop : ∀ P b c,
    Reflexive G → stable R P →
    stable R (assrtT b) → stable R (assrtF b) →
    triple_rg R G (assrtAnd (assrtT b) P) c P →
    triple_rg R G P (while b do c end) (assrtAnd (assrtT b) P)

| RConcur : ∀ Rl Rr G1 G2 P Q1 Q2 Q cr cl,
   Reflexive G1 → Reflexive G2 →
   rstImp R (rstAnd Rl Rr) → rstImp (rstOr G1 G2) G →
   rstImp (rstOr Rl G1) Rr → rstImp (rstOr Rr G2) Rl →
   assrtImp (assrtAnd Q1 Q2) Q →
   stable (rstOr Rr G2) Q1 → stable (rstOr Rl G1) Q2 →
   stable (rstOr Rr G2) P → stable (rstOr Rl G1) P →
   triple_rg Rl G1 P cl Q1 →
   triple_rg Rr G2 P cr Q2 →
   triple_rg R G P (par cl with cr end) Q.
```

In the specification of the `RAtom` constructor, we use as premise the term `triple`. This represents a valid deduction tree using the sequential Hoare proof system, which we proved correct with respect to the sequential fragment of IMPp, but that we do not present it in this dissertation. The proof of the soudness of sequential Hoare logic captured by `triple` follows along the lines of the works of Leroy [68] and Pierce *at al.* [85], and is based also in a small-step reduction semantics.

5.6 Soundness of HL-RG

We will now proceed with the proof of soundness of HL-RG in the Coq proof assistant, following along the lines of our reference work, Coleman and Jones [27]. Here we do not describe in detail the actual Coq scripts that were required to build the proofs. Instead, we provide proof sketches that indicate the way those scripts were constructed. Our development is available online in [76].

Soundness Proof

Recall that the validity of a specification $\{R, P\}\ C\ \{Q, G\}$, which we write

$$\models\ \{R, P\}\ C\ \{Q, G\},$$

has the following meaning: if a program C starts its computation in a state where the precondition P holds and if the interference of the environment is captured by the rely condition R then, if C terminates, it terminates in a state where the postcondition Q holds; moreover, all the intermediate program reduction steps f C satisfy the guarantee condition G. Formally, the definition of Hoare validity for $\mathsf{HL\text{-}RG}$ is defined as

$$\begin{array}{c} \models\ \{R, P\}\ C\ \{Q, G\} \\ \stackrel{\text{def}}{=} \\ \forall C\, s, P(s) \rightarrow \forall s', (\langle C, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle) \rightarrow Q(s') \wedge \mathsf{within}(R, G, C, s). \end{array} \tag{5.10}$$

The soundness of the proof system goes by induction on the size of the proof tree, and by case analysis on the last rule applied. Since the proof system is encoded as the inductive type `tripleRG`, a proof obligation is generated for each of its constructors. For each constructor C_i in the definition of type `tripleRG` a proof obligation of the form

$$\vdash\ \{R, P\}\ C_i\ \{Q, G\}\ \rightarrow \models\ \{R, P\}\ C_i\ \{Q, G\},$$

is generated. This means that we have to prove

$$\vdash\ \{R, P\}\ C_i\ \{Q, G\} \rightarrow\ \forall s,\ P(s) \rightarrow \forall s',\ (\langle C_i, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle) \rightarrow Q(s') \tag{5.11}$$

and also

$$\vdash\ \{R, P\}\ C_i\ \{Q, G\} \rightarrow\ (\forall s,\ P(s) \rightarrow \forall s',\ (\langle C_i, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle) \rightarrow \mathsf{within}(R, G, C_i, s)). \tag{5.12}$$

We call to (5.11) the *Hoare part* of the proof, and we call to (5.12) the *Guarantee part* of the proof, respectively.

Skip

The statement skip produces no reduction. Therefore, the only transition available to reason with is the environment, which satisfies the rely relation.

Hoare part. From $\langle \mathsf{skip}, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$ we know that $(s, s') \in R^\star$ and, from the stability of Q with respect to R, we obtain $Q(s)$ from $Q(s')$. The rest of the proof trivially follows from the hypothesis $P \rightarrow Q$.

Guarantee part. From the definition of within and from $P(s)$ for some state s, we know that $\langle \mathsf{skip}, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$ and $\langle \mathsf{skip}, s\rangle \stackrel{c}{\Longrightarrow} \langle C, s'\rangle$ for some state s'. This is, however, an absurd since no reduction $\langle \mathsf{skip}, s\rangle \stackrel{c}{\Longrightarrow} \langle C, s'\rangle$ exists in the definition of $\stackrel{c}{\Longrightarrow}$.

Assignment

The assignment is an indivisible operation which updates the current state with a variable x containing a value given by an expression e. The precondition P and the postcondition Q can be streched by the interference of the environment R due to the stability conditions. This only happens right before, or right after the execution of the assignment.

Hoare part. By induction on the length of the reduction $\langle x := E, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$ we obtain $\langle \mathsf{skip}, s\rangle \stackrel{c}{\Longrightarrow} \langle \mathsf{skip}, s[E/x]\rangle$ and $\langle \mathsf{skip}, s[E/x]\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$. Since skip implies the impossibility of reductions, we are able to infer that $(s[E/x], s') \in R^\star$. By the stability of the postcondition, we obtain $Q(s[E/x])$ from $Q(s')$.

In what concerns the case where the environment causes interference, we prove by induction on the length of $\langle x := E, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$ that exists $s'' \in \Sigma$ such that $(s, s'') \in R$ and $\langle x := E, s''\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle$. By the stability of the precondition $P(s)$, we conclude $P(s'')$. From the induction hypothesis, we conclude that $Q(s')$.

Guarantee part. For proving the guarantee satisfaction, *i.e.*, to prove $\mathsf{within}(R,G,x := E,s)$, we first obtain that if $\langle x := E, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle C', s'\rangle$ then both $C' = \mathsf{skip}$ and $s' = s[E/x]$ must hold. Hence, we conclude $(s, s[E/x]) \in G$ by the hypotheses.

Sequence

For the conditional statement, the proof follows closely the proof that is constructed to prove the soundness of the inference rule in the case of sequential Hoare logic, for both the cases of the Hoare part and the guarantee part.

Hoare part. The proof goes by showing that since we have the reduction

$$\langle\, C_1; C_2, s\,\rangle \stackrel{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$$

for $s, s' \in \Sigma$, then there exists an intermediate state $s'' \in \Sigma$ such that $\langle\, C_1, s\,\rangle \stackrel{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s''\,\rangle$ and $\langle\, C_2, s''\,\rangle \stackrel{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$ hold. Using $\langle\, C_1, s\,\rangle \stackrel{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s''\,\rangle$ and the induction hypotheses, we show that the postcondition of C_1 is the precondition of C_2, and by $\langle\, C_2, s''\,\rangle \stackrel{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$ we obtain the postcondition of C_2, which finishes the proof.

Guarantee part. For proving $\mathsf{within}(R, G, C_1; C_2, s)$ we need the following intermediate lemma:

$$\begin{array}{c} \mathsf{within}(R, G, C_1, s) \\ \to \\ (\forall s' \in \Sigma,\ \langle C_1, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle \to \mathsf{within}(R, G, C_2, s')) \to \mathsf{within}(R, G, C_1; C_2, s). \end{array} \tag{5.13}$$

By applying (5.13) to $\mathsf{within}(R,G,C_1; C_2,s)$, we are left to prove first that $\mathsf{within}(R, G, C_1, s)$, that is immediate from the hypothesis $\models \{R, P\}\ C_1\ \{Q', G\}$. From the same inductive

hypothesis, we obtain $Q'(s')$, where $s' \in \Sigma$ is the state where C_1 finishes its execution. For the second part of the proof, which corresponds to prove that

$$\forall s' \in \Sigma,\ \langle C_1, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle \to \mathsf{within}(R, G, C_2, s'),$$

we obtain $Q'(s')$, and from $Q'(s')$ and $\models \{R, Q'\}\ C_2\ \{Q, G\}$ we obtain $\mathsf{within}(R, G, C_2, s')$, which closes the proof.

Conditional

For the conditional statement, the proof follows closely the proof that is constructed to prove the soundness of the inference rule in the case of sequential Hoare logic, for both the cases of the Hoare part and the guarantee part.

Hoare part. The proof goes by induction on the structure of the reduction

$$\langle\, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s\,\rangle \stackrel{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$$

and by case analysis in the value of the guard B. For the cases where no interference occurs, the proof follows immediately from the hypothesis. When interference occurs, we use the stability of the guard B with respect to the rely condition R, which keeps the evaluation of B unchanged. Once this is proved, the rest of the proof follows also from the hypotheses.

Guarantee part. In order to prove

$$\mathsf{within}(R, G, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s),$$

we require the following auxiliary lemmas:

$$\mathsf{within}(R, G, C_1, s) \to [\![B]\!]_{\mathbb{B}}(s) \to \forall C_2,\ \mathsf{within}(R, G, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s) \tag{5.14}$$

and

$$\mathsf{within}(R, G, C_2, s) \to [\![\neg B]\!]_{\mathbb{B}}(s) \to \forall C_1,\ \mathsf{within}(R, G, \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}, s). \tag{5.15}$$

Since $\mathsf{within}(R, G, C_1, s)$ and $\mathsf{within}(R, G, C_2, s)$ are already in the hypotheses, we just perform a case analysis on the value of the guard B, and directly apply (5.14) and (5.15) for each of the cases, which finishes the proof.

While Loop

The proof of the soundness of the rule for the while is obtained using an adequate elimination principle because the induction on the length of the derivation is not enough to ensure that the loop amounts to a fixpoint. The same principle needs to be used for both the Hoare part and the guarantee part of the proof.

Hoare part. To prove the soundness for the Hoare part, we first prove the validity of the following (generic) elimination principle:

$$
\begin{array}{l}
\forall x : A, \forall P : A \to A \to \texttt{Prop}, (x,x) \in P \to \\
(\forall n\, x\, y\, z,\ (x,y) \in R \to (y,z) \in R^n \to \\
(\forall y_1\, k,\ k \leq n \to (y_1,z) \in R^k \to (y1,z) \in P) \to (x,z) \in P) \to \\
\forall x\, y,\ (x,y) \in R \to (x,y) \in P.
\end{array}
$$

This inductive argument states that for a predicate P to hold along a reduction defined by the closure of the relation R, then it must hold for the case (x,x) and, if after n reductions it satisfies (y,z), then for all reductions carried in less that n steps P must hold. The idea is to instantiate this elimination principle to the case of the while loop. In order to correctly apply this predicate, we first need to transform the validity condition

```
∀ s, P(s) →
  ∀ s', star _ (prog_red R) (while b do c end, s) (skip,s') →
    (assrtAnd (assrtT b) P) s'.
```

into its equivalent form

```
∀ s, P(s) →
  (∀  s', star _ (prog_red R) (while b do c end, s) (skip,s') →
    ∀ p p',
      star _ (prog_red R) p p' →
      fst p = (while b do c end) →
      fst p' = skip →
      P (snd p) → (assrtAnd (assrtF b) P) (snd p')).
```

Once we apply the elimination principle to our goal, we are left with two subgoals: the first goal considers the case where the number of reductions is zero, so we are asked to prove that `while b do c end = skip`. This goal is trivially proved by the injectivity of the constructors.

The second goal states that the current state of the program results from a positive number of reductions. Hence, we perform case analysis on the last reduction, which originates two new subgoals: one for the case when the reduction is the result of the program execution; and another when the reduction is due to the interference of the environment. In the former case, we know that the loop has reduced to a conditional instruction, which leaves us with two cases:

- if the Boolean guard is false, we are left with a reduction $\langle\, \mathsf{skip}, s\,\rangle \overset{R^n}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$, which implies that $(s,s') \in R^n$. We use the latter fact and the stability of the guard with respect to the rely condition to move the postcondition $P \wedge [\![\neg B]\!]_{\mathbb{B}}(s')$ to $P \wedge [\![\neg B]\!]_{\mathbb{B}}(s)$, which is available from the hypotheses;

- if the Boolean guard evaluates to a true, we know that the loop reduces to

$$\langle\, C; \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end}, s \,\rangle \stackrel{R^n}{\Longrightarrow} \langle\, \mathsf{skip}, s' \,\rangle,$$

  which we decompose into

$$\langle\, C, s \,\rangle \stackrel{R^m}{\Longrightarrow} \langle\, \mathsf{skip}, s'' \,\rangle,$$

  and $\langle\, \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end}, s'' \,\rangle \stackrel{R^{(n-m)}}{\Longrightarrow} \langle\, \mathsf{skip}, s' \,\rangle$, for some $m \in \mathbb{N}$ such that $m < n$. The rest of the proof follows from simple logical reasoning with the hypotheses.

For the case where the last reduction has been performed by the interference of the environment, we first move the precondition to the state resulting from the action of the rely condition and end the proof by the same kind of reasoning used above.

Guarantee part. The proof of the satisfaction of the guarantee relation goes in a similar way as the Hoare part. Since, by the hypotheses, we know that all the program reductions of C are constrained by the guarantee condition G, then a finite composition of C should also by constrained by G, which should allow us to conclude that the loop satisfies the guarantee condition. Up until now, we were not able to complete this proof within the Coq formalisation and we believe the reason is that we have not found and used the adequate elimination for this purpose. Here we give the partial proof that we have obtained and show the point where we are blocked. The goal is to prove that

$$\mathsf{within}(R, G, C, s) \;\rightarrow\; \mathsf{within}(R, G, \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end}, s),$$

regarding that we already know that $\{R, P\}\, C\, \{Q, G\}$ holds. From this last hypothesis and because we know that $\langle\, \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end}, s \,\rangle \stackrel{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s' \,\rangle$, we also know that the postcondition $(P \wedge [\![\neg B]\!]_{\mathbb{B}})(s')$ holds. Next, we perform case analysis on the value of the Boolean guard B, that is:

- if $[\![\neg B]\!]_{\mathbb{B}}(s)$ holds, then we know that $\mathsf{within}(R, G, \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end}, s)$ can be replaced by

$$\mathsf{within}(R, G, \mathsf{if}\, B\, \mathsf{then}\, (\mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end})\, \mathsf{else}\, \mathsf{skip}\, \mathsf{fi}, s)$$

  and from the latter statement we are able to conclude $\mathsf{within}(R, G, \mathsf{skip}, s)$ holds and that is true, as we have showed before.

- if $[\![B]\!]_{\mathbb{B}}(s)$ holds, then we know that $\mathsf{within}(R, G, \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end}, s)$ can be replaced by

$$\mathsf{within}(R, G, \mathsf{if}\, B\, \mathsf{then}\, (\mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end})\, \mathsf{else}\, \mathsf{skip}\, \mathsf{fi}, s)$$

  which in turn is equivalent to $\mathsf{within}(R, G, C; \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end}, s)$. By the properties of the within predicate for sequential execution we reduce the previous goal to a proof that $\mathsf{within}(R, G, C, s)$ holds (which is immediate from the hypotheses) and to a proof that

$$\forall n \in \mathbb{N}, \forall s'' \in \Sigma, (\langle\, C, s \,\rangle \stackrel{R^n}{\Longrightarrow} \langle\, \mathsf{skip}, s'' \,\rangle) \rightarrow \mathsf{within}(R, G, \mathsf{while}\, B\, \mathsf{do}\, C\, \mathsf{end}, s'')$$

  which is still to prove.

Consequence

Proving both the Hoare part and the guarantee part for the inference rule is pretty straightforward. The proof goes by induction on the length of $\langle\, C, s\,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$, and by the properties of the implication on the preconditions and postconditions, and also by the properties of the implication on the rely and the guarantee conditions.

Atomic Execution

For proving the soundness of the inference rule for the atomic execution of programs, we must show that the environment causes interference either right before, or right after the execution of program given as argument for the atomic statement. Moreover, we have to show that if $\langle\, C, s\,\rangle \overset{\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle$ then $Q(s')$ and $(s, s') \in G^\star$ hold.

Hoare part. First we obtain the hypotheses gives the possible interference of the environment, and the atomic execution of C. Considering that we have

$$\langle\, \mathsf{atomic}(C), s\,\rangle \overset{R^\star}{\Longrightarrow} \langle\, \mathsf{skip}, s'\,\rangle,$$

those hypothesis are the following:

$$(s, t) \in R^\star, \tag{5.16}$$

$$(t', s') \in R^\star, \tag{5.17}$$

$$\langle C, t\rangle \overset{ID^\star}{\Longrightarrow} \langle \mathsf{skip}, t'\rangle, \tag{5.18}$$

considering that $t, t' \in \Sigma$. Next, we prove that $P(t)$ can be inferred from $P(s)$ by the stability condition on (5.16). Moreover, we use the soundness of sequential Hoare logic to prove that from hypothesis $\{P\}\, C\, \{Q\}$ and from $P(t)$ we have $\langle\, C, t\,\rangle \overset{\star}{\Longrightarrow} \langle\, \mathsf{skip}, t'\,\rangle$. From the previous reduction we conclude that $Q(t')$ holds, and by (5.17) we also conclude that $Q(s')$ also holds, which finishes the proof.

Guarantee part. Considering the hypotheses (5.16), (5.17) and (5.18), we deduce $(t, t') \in G^\star$ using the same reasoning that we employed to obtain $Q(t')$. By the transitivity of the guarantee condition we know that $(t, t') \in G$, wich allows us to conclude that the atomic execution of C respects its guarantee condition.

Parallel Execution

To prove of the soundness of the inference rule (HG-PAR) we are required to build the proofs that show that the program C_1 satisfies its commitments when executing under the interference of C_2, and the other way around.

Hoare part. The proof of the Hoare validity for a statement concerning the parallel computation of two programs C_1 and C_2 is carried in two steps. First we prove that starting with C_1

leads to an intermediate configuration where C_1 finishes and also that in that configuration, the original program C_2 has reduced to C_2'. We then prove that C_2' terminates and, by the stability condition, we stretch the postcondition of $C_1$to the same state where C_2 finished. The second phase consists in an equivalent reasoning, but starting with the execution of C_2 and using a similar approach.

The first part requires the reasoning that follows. We start the proof by obtaining a new set of hypotheses that allows us to conclude $Q_1(s')$. From the hypothesis

$$\langle \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, s\rangle \stackrel{(R_1\wedge R_2)^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle, \tag{5.19}$$

and from the hypotheses $\{R_1, P\}\ C_1\ \{Q_1, G_1\}$, $\{R_2, P\}\ C_2\ \{Q_2, G_2\}$, and $P(s)$ we obtain

$$\mathsf{within}(R_1, G_1, C_1, s), \tag{5.20}$$

$$\mathsf{within}(R_2, G_2, C_2, s), \tag{5.21}$$

$$(\langle C_1, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle) \to Q_1(s'), \tag{5.22}$$

$$(\langle C_2, s\rangle \stackrel{R^\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle) \to Q_2(s'). \tag{5.23}$$

From (5.19), (5.20) and (5.21) we can conclude that there exists state s'' such that the program C_1 finishes executing in s'', and such that the program C_2' stars its execution in s'', where C_2' is the result of the execution of C_2 under the interference of the environment, which also contains the interference caused by C_1. Hence, the following properties hold:

$$\langle C_1, s\rangle \stackrel{R_1^\star}{\Longrightarrow} \langle \mathsf{skip}, s''\rangle, \tag{5.24}$$

$$\langle C_2, s\rangle \stackrel{R_2^\star}{\Longrightarrow} \langle C_2', s''\rangle, \tag{5.25}$$

$$\langle \mathsf{par}\ \mathsf{skip}\ \mathsf{with}\ C_2'\ \mathsf{end}, s''\rangle \stackrel{(R_1\wedge R_2)\star}{\Longrightarrow} \langle \mathsf{skip}, s'\rangle. \tag{5.26}$$

Since both (5.21) and (5.25) hold, then by (5.7) we conclude $\mathsf{within}(R_2, G_2, C_2', s'')$. Moreover, from $\mathsf{within}(R_2, G_2, C_2', s'')$ and (5.26) we also conclude

$$\mathsf{within}(R_2, G_2, \mathsf{par}\ \mathsf{skip}\ \mathsf{with}\ C_2'\ \mathsf{end}, s''). \tag{5.27}$$

To conclude this part of the proof, we need to show that $Q_1(s') \wedge Q_2(s')$. For that, we split $Q_1(s') \wedge Q_2(s')$ and show that $Q_1(s')$ holds by the stability conditions. To prove $Q_2(s')$, we reason as before, but considering that the command C_2 ends its execution first that C_1.

Guarantee part. The proof goes by applying the following property of within with respect to parallel computation:

$$\begin{array}{c}\mathsf{within}(R \cup G_2, G_1, C_1, s) \to \mathsf{within}(R \cup G_1, G_2, C_2, s) \to \\ \mathsf{within}(R, G, \mathsf{par}\ C_1\ \mathsf{with}\ C_2\ \mathsf{end}, s).\end{array} \tag{5.28}$$

Using (5.28) we are left to prove $\mathsf{within}(R \cup G_2, G_1, C_1, s)$ and $\mathsf{within}(R \cup G_1, G_2, C_2, s)$. Here we present only the proof of the former, since the proof of the later is obtained by similar reasoning.

From the hypotheses we know that $\mathsf{within}(R \cup G_2, G_1, C_1, s)$ is equivalent to

$$\mathsf{within}((R_1 \cap R_2) \cup G_2, G_1, C_1, s).$$

From the hypotheses, we know that $\mathsf{within}(R_1, G_1, C_1, s)$ and that $(R_1 \cap R_2) \cup G_2 \rightarrow R_1$. By the properties of the implication and the predicate within we conclude the proof.

5.7 Examples and Discussion

We will now analyse the effectiveness of our development of HL-RG in the verification of simple parallel programs. We start by the following example.

Example 41. *This example is a classic one in the realm of the verification of concurrency. The idea is to do a parallel assignment to a variable x initialised beforehand with some value greater of equal to 0. The corresponding* IMPp *program code is the following:*

$$C \stackrel{\text{def}}{=} \mathsf{par}\ x := x + 1\ \mathsf{with}\ x := x + 2\ \mathsf{end}.$$

The rely condition states that the value of the variable x after a reduction is greater or equal than before the reduction occurs. The guarantee is defined in the exact same way, that is,

$$R \stackrel{\text{def}}{=} G \stackrel{\text{def}}{=} \lambda s \lambda s'. [\![x]\!]_{\mathbb{N}}(s) \leq [\![x]\!]_{\mathbb{N}}(s').$$

Finally, the precondition states that initially the value of x is greater or equal to 0, and the postcondition states that the final value of x is greater or equal to 2, that is,.

$$P \stackrel{\text{def}}{=} \lambda x. [\![x]\!]_{\mathbb{N}}(s) \geq 0,$$

$$Q \stackrel{\text{def}}{=} \lambda x. [\![x]\!]_{\mathbb{N}}(s) \geq 2.$$

The full specification corresponds to $\{R, P\}\ C\ \{Q, G\}$ and the first thing we do to prove this specification valid is to apply the inference rule HG-PAR. *The application of* HG-PAR *requires us to provide the* Coq *proof assistant with the missing value, namely, the rely conditions R_1 and R_2 such that $R_1 \wedge R_2 \rightarrow R$; the two guarantee conditions G_1 and G_2 such that $G_1 \vee G_2 \rightarrow G$ and the postconditions C_1 and C_2 such that their conjuntion implies C. We instantiate these new variables with*

$$\begin{array}{lll} R_1 & \stackrel{\text{def}}{=} & R \\ G_1 & \stackrel{\text{def}}{=} & G \\ C_1 & \stackrel{\text{def}}{=} & x \geq 1 \end{array} \qquad \begin{array}{lll} R_2 & \stackrel{\text{def}}{=} & R \\ G_2 & \stackrel{\text{def}}{=} & G \\ C_2 & \stackrel{\text{def}}{=} & x \geq 2. \end{array}$$

The goal associated with the stability conditions are proved in a straightforward way. Next, we prove

$$\{R_1, P\}\ x := x + 1\{Q_1, G_1\}$$

and

$$\{R_2, P\}\ x := x + 2\{Q_2, G_2\}$$

correct by applying the inference rule (HG-ASSGN)*, and by simple arithmetic reasoning.*

One of the main difficulties of using RG is the definition of the rely and guarantee conditions. This is because the rely and the guarantee conditions have to describe the state of computation of the programs as a whole, which is not always easy. In order to cope with such difficulties, rules for adding "ghost" variables into the programs under consideration were introduced [102]. In the case of our reference work and in our development, this rule is not taken into consideration. Unfortunatelly, the absence of such rule is a strong constraint to the set of parallel programs that we can address with our development. Moreover, our proof system is more restricted than our reference work: we don't allow interference during the evaluation of both arithmetic and Boolean expressions, because these are seen as atomic functions; we also do not have support for reasoning about arrays, which renders out a considerable set of interesting programs commonly used as examples of reference in the literature. This means that, in the future, we have to rethink the design choices that we have made and find ways to improve the development in order to make it more usable.

5.8 Related Work

In this chapter we have described a formalisation, within the COQ proof assistant, of a proof system for RG, following the concepts introduced by Coleman and Jones in [27]. Related work includes the work of these authors, and also the work of Prensa Nieto [78, 88] in the Isabelle proof assistant.

Our formalisation essentially confirms most of the work introduced by Coleman and Jones [27], but extended with atomic execution of programs. Thus, our development shows that the ideas forwarded by these authors seem to be correct and assuming that we were not able to finish the guarantee part for the inference rule for loops not because it is unsound, but because we still have to find the adequate way of addressing it. Therefore, we consider that our development effort can serve as a guide for future formalisations within COQ that address other approaches to RG, or to some of its extensions.

In what concerns to the comparison of our work with the one of Nieto[78, 88], the main diferences are the following: the author formalised a notion of parameterised parallelism, that is, parallel programs are defined as a list of sequential programs. This restriction unable the specification of nested parallelism. Nevertheless, the system mechanised by Nieto contains

an extra rule for introducing "ghost" variables into the original program's code and eases the recording og interference. This allowed for that work to be succesfully used to prove the correctness of larger set of examples than us.

5.9 Conclusions

In this chapter we have described the mechanisation of an Hoare-like inference system for proving the partial correctness of simple, shared-variable parallel programs. The work presented follows very closely the work of Jones and Coleman [27], but ended up in a more restrictive proof system. This is mainly the consequence of the set of hypothesis that are required to show that the parallel execution rule is sound with respect to the small-step semantics that we have decided to use. Still, the main goal of the work we have presented is achieved: we have decided follow this line of work in order to get a better knowledge of the difficulties that arise when formalising a proof system shared-variable parallel within the context of RG principle. In particular we have understood how the definition of the rely and guarantee conditions can be a hard job, even for very simple programs.

These programs are written in the IMPp language, that extends IMP, introduce in Chapter 2 with instructions for atomic and parallel execution of programs. We mechanise a small-step operational semantics that captures a fine-grained notion of computation under interference of the environment. We have also proved the soundness of the inference system HL-RG, which is an extension of the inference system proposed by Coleman and Jones in [27] with a command for the atomic execution of programs.

Although RG has become a mature theory and is a well-known method for verification of shared-variable parallel programs, it is usually difficult to define in it rely and guarantee conditions that specify the behaviours of parallel programs over the whole execution state. Nevertheless, we believe that our formalisation that can serve as a starting step to develop more modern and suited models [99, 35, 34] that handle parallelism and concurrency in a more adequate and flexible way. It is included in our list of future research topics to extend our formalisation in that way.

Another important outcome of this work is our increase in the knowledge of RG that will allow us to have a stronger base to address our next goal, which is to investigate CKA, an algebraic framework for reasoning about concurrent programs. In particular, we are interested in the way it handles RG reasoning, and how we can devise an extension of the ideas of derivation that were studied in Chapter 3 and in Chapter 4 and, with them, try do define decision procedures for RG reasoning within the context of CKA.

Chapter 6

Conclusions and Future Work

Along this dissertation we have described three contributions that we believe can be useful in the verification of imperative programs using the Coq proof assistant. With these contributions, we broaden the set of mechanically verified theories that have direct application to the verification of sequential programs – in the cases of regular expressions and KAT terms – and of shared variable parallel programs – in the case of the RG proof system.

Our first contribution is a library of regular expressions that provides a correct and efficient algorithm for deciding their (in-)equivalence. Since regular expressions can be regarded also as a program logic for programs with non-deterministic choice and loops, we can use the decision procedure to reason about the execution of program traces. This has application, for instance, in run-time verification, where monitors analyse the order of events and deliberate if such order is tolerable, or if it is erroneous. The formalisation of regular expressions contains also a proof tactic that automates reasoning about equations involving binary relations. Since binary relations are one of the core concepts for reasoning about programs, our formalisation can be useful for assisting users in building proofs about programs represented as relations, as it is usually done in point-free approaches. Finally, the development can be extracted into functional code of one of the languages supported by the extraction mechanism of Coq, and that code can be integrated in other developments that require the treatment of regular expressions.

We also implemented a correct algorithm for deciding the equivalence of KAT terms, which have the expressivity to specify propositional programs, *i.e.*, programs with no assignments. The assignments, and other possible first order constructs can be encoded either as Boolean tests or as primitive program symbols. Programs following these encodings are expressive enough to capture several problems related to program verification, that can be solved equationally through KAT. The usage of KAT as a way to encode Hoare specifications allows for the automatic generation of proof of correctness for some programs.

Our last contribution is a sound inference system for reasoning about shared-variable parallel

programs following the RG approach. The support language is simple, but it allows us to capture and verify some of the essencial properties of parallel programs. It is also, for the best of our knowledge, the first formalisation of an RG inference system developed in Coq. Our main objective with this contribution is to define a simple system carrying the foundational properties for reasoning about parallel programs, which can be progressively extended with other concepts that will allow us, somewhere in the future, to conduct more realistic verification tasks using the Coq proof assistant.

Future Research Directions

In what concerns the development of regular expressions, we are interested in extending the development to address regular expressions with intersection and complement, possibly along the lines of [23]. We are also interested in making the development more robust, by developing proof tactics that automate proof steps that are commun to a considerable fragment of the development such as, for instance, tactics to normalise expressions modulo some set of axioms, and also to automate tractable fragments of the underlying model of regular languages. Finally, we would like to investigate ways to improve the efficiency of the decision procedure. One way possible is to use Almeida's [3] representation of derivatives, which consider less symbols to derivate with as the number of symbols in the regular expressions start decreasing due to previous derivations.

In the case of the development of KAT terms, an important research line to be followed is to try to find and equivalent definition of partial derivative that reduces the number of atoms to be considered. Such results will certainly imply considerable important increases in the performance of the decision procedure. We are also interested in the mechanisation of SKAT, which is an extension of KAT that explicitly considers assignments. With such an idealised formalisation, we can use Coq's extraction mechanism to obtain a trusted reasoning mechanism that can be the basis of a certified kernel for a proof assistant similar to KAT-ML that was developed by Aboul-Hosn and Kozen [1].

Finally, we point some research lines to be followed in order to improve our development of RG. A natural path will be to build on the experience we have gained with the development, and extend it to handle memory properties along the lines of RGSep [99] or *deny guarantee* [34]. However, we are more interested in the algebraic approach followed by Hoare *et. al.* with CKA [47]. We wish to mechanise the developed theory, in particular the encoding of RG, but also to search for methods that use derivatives and that may lead to decision procedures that may help, at some point, in the automation of proof construction of specifications of concurrent and parallel programs.

References

[1] K. Aboul-Hosn and D. Kozen. KAT-ML: An interactive theorem prover for Kleene algebra with tests. *Journal of Applied Non-Classical Logics*, 16(1–2):9–33, 2006.

[2] A. Almeida, M. Almeida, J. Alves, N. Moreira, and R. Reis. FAdo and GUItar: tools for automata manipulation and visualization. In S. Maneth, editor, *Proc. 14th CIAA'09*, volume 5642 of *LNCS*, pages 65–74. SV, 2009.

[3] M. Almeida. *Equivalence of regular languages: an algorithmic approach and complexity analysis*. PhD thesis, FCUP, 2011. http://www.dcc.fc.up.pt/~mfa/thesis.pdf.

[4] M. Almeida, N. Moreira, and R. Reis. Antimirov and Mosses's rewrite system revisited. *Int. J. Found. Comput. Sci.*, 20(4):669–684, 2009.

[5] M. Almeida, N. Moreira, and R. Reis. Testing regular languages equivalence. *JALC*, 15(1/2):7–25, 2010.

[6] R. Almeida. Decision algorithms for Kleene algebra with tests and Hoare logic. Master's thesis, Faculdade de Ciências da Universidade do Porto, 2012.

[7] R. Almeida, S. Broda, and N. Moreira. Deciding KAT and Hoare logic with derivatives. In *GandALF*, pages 127–140. Electronic Proceedings in Theoretical Computer Science, September 2012.

[8] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[9] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. In G. Rozenberg and A. Salomaa, editors, *DLT*, pages 195 – 209. World Scientific, 1994.

[10] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, Mar. 2002.

[11] A. Asperti. A compact proof of decidability for regular expression equivalence. In L. Beringer and A. Felty, editors, *Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012.*, number 7406 in LNCS. Springer-Verlag.

[12] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. The Matita interactive theorem prover. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 64–69. Springer, 2011.

[13] G. Barthe and P. Courtieu. Efficient reasoning about executable specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *TPHOLs*, volume 2410 of *LNCS*, pages 31–46. Springer, 2002.

[14] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer, 1984.

[15] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[16] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 platform, version 0.72.* LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.72 edition, May 2012. https://gforge.inria.fr/docman/view.php/2990/7919/manual-0.72.pdf.

[17] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[18] T. Braibant and D. Pous. An efficient Coq tactic for deciding Kleene algebras. In *Proc. 1st ITP*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.

[19] T. Braibant and D. Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1), 2012.

[20] S. Briais. Finite automata theory in Coq. http://sbriais.free.fr/tools/Automata_080708.tar.gz. Online, last accessed August 2011.

[21] S. Broda, A. Machiavelo, N. Moreira, and R. Reis. On the average state complexity of partial derivative automata: An analytic combinatorics approach. *IJFCS*, 22(7):1593–1606, 2011.

[22] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[23] P. Caron, J.-M. Champarnaud, and L. Mignot. Partial derivatives of an extended regular expression. In A. H. Dediu, S. Inenaga, and C. Martín-Vide, editors, *LATA*, volume 6638 of *LNCS*, pages 179–191. Springer, 2011.

[24] J.-M. Champarnaud and D. Ziadi. From Mirkin's prebases to Antimirov's word partial derivatives. *Fundam. Inform.*, 45(3):195–205, 2001.

[25] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. http://adam.chlipala.net/cpdt/.

[26] J. Chrzaszcz. Implementing modules in the Coq system. In D. A. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 270–286. Springer, 2003.

[27] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. Comput.*, 17(4):807–841, 2007.

[28] T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[29] T. Coquand and V. Siles. A decision procedure for regular expression equivalence in type theory. In J.-P. Jouannaud and Z. Shao, editors, *CPP 2011, Kenting, Taiwan, December 7-9, 2011.*, volume 7086 of *LNCS*, pages 119–134. Springer-Verlag, 2011.

[30] F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors. *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *LNCS*. Springer, 2006.

[31] R. Deline and Rustan. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (MSR), Mar. 2005.

[32] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *CoRR*, cs.LO/0310054, 2003.

[33] J. Desharnais, B. Möller, and G. Struth. Modal Kleene algebra and applications – a survey, 2004.

[34] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] X. Feng. Local rely-guarantee reasoning. *SIGPLAN Not.*, 44(1):315–327, Jan. 2009.

[36] J.-C. Filliâtre. Finite Automata Theory in Coq: A constructive proof of Kleene's theorem. Research Report 97–04, LIP - ENS Lyon, February 1997.

[37] J.-C. Filliâtre and P. Letouzey. Functors for proofs and programs. In D. A. Schmidt, editor, *ESOP*, volume 2986 of *LNCS*, pages 370–384. Springer, 2004.

[38] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[39] M. J. Frade and J. S. Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252 – 277, 2011.

[40] G. Gonthier. The four colour theorem: Engineering of a formal proof. Springer-Verlag, Berlin, Heidelberg, 2008.

[41] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. The formalization of the Odd Order theorem., September 2012. http://www.msr-inria.inria.fr/Projects/math-components/feit-thompson.

[42] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. *SIGPLAN Not.*, 37(9):235–246, Sept. 2002.

[43] C. Hardin. Modularizing the elimination of r=0 in Kleene algebra. *Logical Methods in Computer Science*, 1(3), 2005.

[44] C. Hardin and D. Kozen. On the elimination of hypotheses in Kleene algebra with tests. Technical report, 2002.

[45] M. Hennessy. *Semantics of programming languages - an elementary introduction using structural operational semantics.* Wiley, 1990.

[46] C. A. Hoare, B. Möller, G. Struth, and I. Wehrman. Foundations of concurrent Kleene algebra. In *Proceedings of the 11th International Conference on Relational Methods in Computer Science and 6th International Conference on Applications of Kleene Algebra: Relations and Kleene Algebra in Computer Science*, RelMiCS '09/AKA '09, pages 166–186, Berlin, Heidelberg, 2009. Springer-Verlag.

[47] C. A. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.

[48] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[49] J. Hopcroft and R. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 71 -114, University of California, Berkeley, California, 1971.

[50] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[51] W. Howard. *The formulae-as-types notion of construction*, pages 479–490.

[52] L. Ilie and S. Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.

[53] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981.

[54] D. Kaplan. *Regular Expressions and the Equivalence of Programs*. Memo (Stanford Artificial Intelligence Project). 1969.

[55] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, shannon, C. and McCarthy, J. edition.

[56] 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, pages 207–220, New York, NY, USA, 2009. ACM.

[57] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[58] D. Kozen. *Automata and Computability*. Springer-Verlag, New York, 1997.

[59] D. Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

[60] D. Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1):60–76, July 2000.

[61] D. Kozen. Automata on guarded strings and applications. *Matématica Contemporânea*, 2001.

[62] D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report http://hdl.handle.net/1813/10173, Computing and Information Science, Cornell University, March 2008.

[63] D. Kozen and M.-C. Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 568–582, London, UK, UK, 2000. Springer-Verlag.

[64] D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *LNCS*, pages 244–259. Springer-Verlag, 1996.

[65] D. Kozen and J. Tiuryn. On the completeness of propositional Hoare logic. In *RelMiCS*, pages 195–202, 2000.

[66] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Autom. Reasoning*, 49(1):95–106, 2012.

[67] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[68] X. Leroy. Mechanized semantics. In *Logics and languages for reliability and security*, volume 25 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 195–224. IOS Press, 2010.

[69] S. Lescuyer. Containers: a typeclass-based library of finite sets/maps. http://coq.inria.fr/pylons/contribs/view/Containers/v8.3.

[70] S. Lescuyer. First-class containers in Coq. *Studia Informatica Universalis*, 9(1):87–127, 2011.

[71] S. Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq*. Thèse de doctorat, Université Paris-Sud, Jan. 2011.

[72] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[73] B. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.

[74] B. Möller and G. Struth. Modal kleene algebra and partial correctness. In C. Rattray, S. Maharaj, and C. Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2004.

[75] N. Moreira, D. Pereira, and S. Melo de Sousa. PDCoq : Deciding regular expression and KAT terms (in)equivalence in Coq through partial derivative. http://www.liacc.up.pt/~kat/pdcoq/.

[76] N. Moreira, D. Pereira, and S. Melo de Sousa. RGCoq : Mechanization of rely-guarantee in the Coq proof-assistant. http://www.liacc.up.pt/~kat/rgcoq/.

[77] H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[78] L. P. Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002. Available at http://tumb1.biblio.tu-muenchen.de/publ/diss/allgemein.html.

[79] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[80] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, Mar. 2009.

[81] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[82] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Prog ramming Languages*, Austin, Jan. 1989. ACM.

[83] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, 664:328–345, 1993.

[84] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[85] B. C. Pierce, C. Casinghino, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey. *Software Foundations.* Electronic textbook, 2012. http://www.cis.upenn.edu/~bcpierce/sf.

[86] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[87] D. Pous. Relation algebra and KAT in Coq. http://perso.ens-lyon.fr/damien.pous/ra/, December 2012. Last accessed: 30-12-2012.

[88] L. Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In P. Degano, editor, *European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 348–362, 2003.

[89] R. Pucella. On equivalences for a class of timed regular expressions. *Electron. Notes Theor. Comput. Sci.*, 106:315–333, Dec. 2004.

[90] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[91] U. Sammapun, A. Easwaran, I. Lee, and O. Sokolsky. Simulation of simultaneous events in regular expressions for run-time verification. *Electron. Notes Theor. Comput. Sci.*, 113:123–143, Jan. 2005.

[92] K. Sen and G. Rosu. Generating optimal monitors for extended regular expressions. *Electr. Notes Theor. Comput. Sci.*, 89(2):226–245, 2003.

[93] M. Sozeau. Program-ing finger trees in Coq. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 13–24, New York, NY, USA, 2007. ACM.

[94] M. Sozeau. Subset coercions in Coq. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *LNCS*, pages 237–252. Springer Berlin Heidelberg, 2007.

[95] M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, Dec. 2009.

[96] M. Sozeau and O. Nicolas. First-Class Type Classes. *LNCS*, Aug. 2008.

[97] The Coq Development team. The Coq proof assistant. http://coq.inria.fr.

[98] The Coq development team. Coq reference manual. http://coq.inria.fr/distrib/V8.3/refman/.

[99] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *IN 18TH CONCUR*, pages 256–271. Springer, 2007.

[100] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993.

[101] J. Worthington. Automatic proof generation in Kleene algebra. In *RelMiCS'08/AKA'08*, volume 4988 of *LNCS*, pages 382–396, Berlin, Heidelberg, 2008. Springer-Verlag.

[102] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.