

David Pereira

**Towards certified program logics
for the verification of imperative
programs**



**Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2012**

David Pereira

Towards certified program logics for the verification of imperative programs



*Tese submetida à Faculdade de Ciências da Universidade do Porto
para a obtenção do grau de Doutor em Ciência de Computadores*

verificar se não será MApi!

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2012

Abstract

Modern proof assistants are mature tools with which several important mathematical problems were proved correct, and **wich**^h 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 prova modernos são agora ferramentas maduras de uma considerável complexidade nas quais foram formalizados e provados correctos vários problemas matemáticos. São também usados como ferramentas de **suporta** nas quais é possível codificar lógicas de programas e usar essas codificações para tarefas de certificação de programs.

Utilizando o assistente de prova COQ, **foramlizamos** uma biblioteca de linguagens regulares que contém um procedimento de decisão integro e completo para a 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 que é particularmente adequado para a verificação de programas imperativos.

Utilizando também o **assisten de** prova COQ, desenvolvemos uma biblioteca que contém uma formalização que permite raciocinar acerca da 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 integro 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 **programs** e cujo nível de confiança é reforçado dado o uso das mesmas num ambiente seguro, providenciado pelo assistente de prova COQ.

**Para os meus pais David e Maria Clara e para a Ana,
que sempre acreditaram.**

Acknowledgments

I would like to thank my supervisors Nelma Moreira and Simão Sousa for their guidance, the fruitfully discussions related to the topic of this Ph.D. thesis proposal, and mainly for their friendship and encouragement.

I am also particularly in depth to José Carlos Bacelar Almeida who showed himself always available to discuss with me the biggest problems that I had while doing proofs with COQ. Besides **its** knowledge and enthusiasm, **its** encouragement helped me to go along with the PhD work. I thank also Ricardo Almeida, whom I get to know close to the end of this thesis, but whose discussions about Kleene algebra with tests were very fruitful and interesting.

Obviously, the work environment was also of great help. I wish to thank to André, Andreia, Bruno, Besik, Cláudio, Marco and Vitor for the fun moments that we have passed in the office.

In terms of financial support, I thank Fundação para a Ciência e Tecnologia, and the MAP-i organization for accepting my appliance to a Ph.D. scholarship. Without it, this work would never be possible.

Towards the end of the PhD work a set 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 the new colleges I made there, namely, André, Cláudio and José.

Finally, I would like to thank my family, and close friends, for all their flawless support in this stressing adventure. My parents always believed that I could end this, and always supported me by any means possible into it. My little nephews, Carolina and Rodrigo, were always able to make ^{me} smile, even in the days where proof construction made me one of the worst people to be nearby.

Obviously, I could not end without thanking my girlfriend Ana, who was always by my side and that bared with me until the very end and always unconditionally believing in me and supporting me in all the ways one can **image**. Literally, without her I would have never had the change to write this dissertation.

Contents

Abstract	v
Resumo	vii
Acknowledgments	xi
List of Tables	xv
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	18

2.2	Hoare Logic	21
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	37
3.3	A Procedure for Regular Expression Equivalence	47
3.3.1	The Procedure EQUIVP	47
3.3.2	Implementation	50
3.3.3	Correctness and Completeness	57
3.3.4	Tactics and Automation	61
3.3.5	Performance	69
3.4	Related Work	70
3.5	Conclusions	72
4	Equivalence of KAT Terms	75
4.1	Kleene Algebra with Tests	76
4.2	The Language Model of KAT	76
4.3	Partial Derivatives of <i>Kleene algebra with tests</i> (KAT) Terms	84
4.4	A Procedure for KAT Terms Equivalence	90
4.4.1	The Procedure EQUIVKAT	90
4.4.2	Implementation, Correctness and Completeness	92
4.5	Application to Program Verification	98

4.6	Related Work	104
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	111
5.3.1	Reductions Under Interference	115
5.4	A Proof System for Rely-Guarantee	117
5.4.1	Inference Rules	119
5.5	Soundness of RG-HL	123
5.6	Related Work	132
5.6.1	Discussion and Conclusion	133
5.7	Conclusions	134
6	Conclusions	137
6.1	Future Research Directions	138
	References	140

List of Tables

3.1	Performance results of the tactic <code>dec_re</code>	70
3.2	Comparison of the performances.	72

List of Figures

3.1	The transition diagram of the DFA D	31
3.2	NFA accepting sequences of 0's and 1' that end with an 1.	33
5.1	Rely and guarantee vs. preconditions and postconditions.	109

Chapter 1

Introduction

The growth in complexity of computer systems over the past decade, and the ever-more critical role that they assume in our daily lives, has led to a great research interest in formal verification. Using formal systems, designers and developers have access to a framework where they can rigorously specify the properties that a system has to meet, and it may serve also as a guide to further development. Usually, these formal systems also provide a proof system, which grants the possibility of proving the correctness of the behaviour of the system with respect to its specification. Using this approach, the number of design and implementation errors may decrease drastically, and the trust of the users of the systems will increase.

The origins of formal verification date back from the 19th century with Frege, who introduced first-order logic formally, which in turn resulted in the notion of what now is understood by *formal proof*. A formal proof is a sequence of derivation steps, such that each of those steps can be checked to be well-formed in the underlying formal 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 a non-trivial task. The first incompleteness theorem of Gödel showed that ^{no} ~~not~~ formal system exists that allows to deduce all true statements, as soon as it ^{includes} ~~considers~~ 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 asserts the validity of all true first-order formulas. Nevertheless, this inherent problem of first-order logic impelled researchers to find fragments that were expressive enough, and also

decidable. This effort resulted in the advent of *automated theorem proving*, whose **main** representatives are *satisfiability of Boolean formulas* (SAT) [] and *satisfiability modulo theory* (SMT) [].

Parallel to the development of automated theorem proving, great 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 enables the user to encode mathematics on a computer, and that assists him in checking that each of the proof's derivation steps are indeed correct with respect to the underlying formal system supporting the prover. The first project that addressed a primitive notion of proof assistant was Automath [22], led by De Bruijn, and whose objective was to provide a mechanical verification system for mathematics. The results of this effort was a language where mathematical statements could be written rigorously, as well as a set of implementations of proof-checkers responsible for verifying all the derivations contained in those proofs. These proofs are usually referred to as *proof scripts*, as 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 these statements are indeed true.

De Bruijn also made another important contribution to the development of proof assistant, by bringing the notion of *proof-as-objects* into this field, which is tantamount to the notion of the *Curry-Howard isomorphism* [55] that states that a proof P of a theorem ϕ , in a constructive logic, can be regarded as the 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 [78] that lie type theory as a foundation of mathematics, more complex and powerful the proof assistants became. As a consequence, type checking tools also become more complex, but they are still 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 *De Bruijn 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 Bruijn 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 $f : A \rightarrow B$. 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 rest of the program.

In the actual state-of-the-art, the role of proof assistants cannot be overlooked, being it on the certification of mathematics, or for the formal verification and development of computer systems. The mechanised proofs of the four-color theorem by Gonthier and Werner [43], the Feit-Thomson theorem [44] by Gonthier *et. al.*, the certified compiler CompCert by Xavier Leroy [72], the certification of microkernel seL4 (Secure Embedded L4)[61] in HOL, and the certification of automated theorem prover ALT-ERGO in COQ [76] are among 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 ends up being tedious and, very often, difficult. However, the advantage of using proof assistants it is that we define and certify the concepts once and for all, and we can incorporate them in other developments with full trust.

The subject of this thesis follows the lines of the usage of proof assistants to encode theories that can be used to conduct program verification tasks. We address both sequential programs, and parallel ones. For sequential programs, we formalise 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 formalise an extension of Hoare logic that is able to express properties of parallel execution in a compositional way.

Summing up, our thesis aims at contributing with the following three formalisations:

A decision procedure for regular expression equivalence. Regular expressions are amongst the most common, yet important concepts that are present, in some way or another, in any sub-area of computer science. They are most recognised as a tool for matching patterns in languages, hence they are present in strength in the world-wide-web, compilers, text editors,

etc. They were first introduced in the seminal work of Kleene [60] as a specification languages for automata. Their compactness, flexibility and expressiveness lead to applications outside language processing, and have been applied with success, for instance, as run-time monitors for programs [95, 94]. They can indeed be seen as a program logics that allow to express non-deterministic choice, sequence, and finite iteration of programs. Moreover, they were extended to address imperative programs [59, 64] and realtime systems [1, 92]. Hence, it is important to provide formalised theories about regular expressions, so that we can apply them to 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 formalise the decision procedure and extract it as a 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 [11, 10]. Since regular expressions coincide with relational algebra, we also developed a proof tactic to decide relational equivalence using the previously developed decision procedure for regular expressions.

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 that captures the conditional and while-loop constructions of imperative languages. These expressions are terms of *Kleene algebra with tests*, introduced by Dexter Kozen [64]. We have formalised 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 are presented which show the applicability of Kleene algebra with tests in program verification tasks.

A sound inference system for rely/guarantee. Our final contribution focus the formal aspects and verification of parallel programs. We have chosen to specify and prove sound an inference system for proving the correctness of programs of 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 [57], and latter described [28, 82, 104] by a small-step reduction semantics that considers a fine-grained notion of interference caused by the environment. The source of the interference is modelled by the rely relation. The effect that the computation of the individual programs imposes in the environment is constrained by a guarantee relation. Its inference 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". David Pereira, Nelma Moreira 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*, 2012.
- "Mechanisation of KAT terms equivalence". David Pereira, Nelma Moreira and Simão Melo de Sousa. *Submitted*.

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 formalisations developed, and also describes Hoare logic, the program logic that is central to the application of these same formalisations to program verification.

Chapter 3 describes the mechanisation of a decision procedure for regular expression equivalence. It also includes the formalisation 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 mechanisation 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 program equivalence and partial correctness.

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

Chapter 6 reviews the contributions given in the previous three chapters and establishes possible research lines that aim at solidifying the application of our contributions to more effective ways of applying them to the verification of more complex imperative programs.

Chapter 2

Preliminaries

In this chapter we introduce the COQ proof assistant [100], our tool of choice for the developments that are described along this dissertation. We also introduce Hoare logic [50], the well known program logic that has become the standard logic for addressing correctness of imperative 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.* [89], and of Chlipala [26].

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) [29] with (dependent) inductive definitions. In rigor, since version 8.0, COQ is an implementation of a weaker version of CIC, named the *predicative Calculus of Inductive Constructions* (*pCIC*), whose rules are described in detail in the official COQ manual [35].

COQ is supported by a rich typed λ -calculus that features polymorphism, dependent types and very expressive (co-)inductive types, which is built on the *Curry-Howard Isomorphism* (CHI) programs-as-proofs principle [55]. In CHI, a typing relation $t : A$ can either be seen 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*)

$$\frac{A \rightarrow B \quad A}{B},$$

and the function application rule of λ -calculus

$$\frac{f : A \rightarrow B \quad x : A}{f x : B},$$

from where it is immediate to see that the second rule is the same as the first, if we erase the typing information. Moreover, interpreting the typing relation $x : A$ as the logical statement " x proves A ", and interpreting $f : A \rightarrow B$ as "*the function f transforms a proof of A into a proof of B* ", then we conclude that the function application of the term x to 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 developed programs, and build proofs that show that such programs are correct with respect to the specifications defined.

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

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

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 computation occur 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 computation is simply a series reductions over a term. The set of all reductions form 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 $\text{let } x := e_1 \text{ in } e_2$ into the term $e_2\{x/e_1\}$; finally, the reduction ι , pronounced *iota reduction*, is responsible for computation with recursive functions.

A fundamental feature of COQ's underlying type system is the support for *dependent product types* $\Pi x : A.B$, which extends functional types $A \rightarrow B$ 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 $A \rightarrow B$.

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* [87] for the sake of preserving the termination of the type checking algorithm. One of the simplest examples is the classical definition of Peano numbers:

```
Inductive nat : Set :=
| 0 : 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* `0` and `S`, and an automatically generated induction principle `nat_ind` defined as follows.

$$\forall P : \text{nat} \rightarrow \text{Prop}, P \ 0 \rightarrow (\forall n : \text{nat}, P \ n \rightarrow P \ (S \ n)) \rightarrow \forall n : \text{nat}, P \ n.$$

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 `0` and `S`. By means of *pattern matching*, we can implement recursive functions

by deconstructing the given term and producing new terms for each constructor. An example is the following function that adds two natural numbers:

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | 0 => 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, and because recursive calls are performed on terms that are *structurally* smaller than the given recursive argument. This is a strong requirement of CIC that forces all functions to be terminating. We will see ahead that non-structurally recursive functions can still be programmed within COQ via a translation into equivalent structurally decreasing ones.

More complex inductive types can be defined, namely inductive definitions which 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]` of length 2 and 1, whose elements are the constructors of another inductively defined type `A`.

```
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, defined 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 l'1 => vcons A (n0 + n') v (app n0 l'1 n' l2)
  end.
```

Note that there is a difference between the pattern-matching construction `match` used in the addition of two natural numbers, and the one used in the concatenation of vectors: in the latter, the returning 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 the reductions within a virtual machine [45] ensuring a more efficient computation within COQ's environment.

2.1.3 Proof Construction

As we have seen, the type system behind 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 of two propositions $A \wedge B$ is encoded through 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 associated to `and`, and automatically generated by COQ, is the expected one:

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

Disjunction is encoded in a similar way, with two constructors, each corresponding to a each of the branches of the disjunction. Negation is defined as a function mapping 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 through the following inductive definition:

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

The inductive definition `ex` enforces that we have to provide a *witness* that the predicate `P` verifies the property expected on the term x , in the spirit of constructive logic, where connectives are seen as functions taking proofs and producing new proofs.

The primitive way of the COQ proof construction process is to explicitly build CIC terms. However, proofs can be built more conveniently and interactively in a backward fashion through a language of commands called *tactics*. Although tactics are commonly used when a user is in the *proof mode* of COQ, activated by the `Theorem` command (and similar commands), they can also be used to interactively construct programs. However, that must be done carefully, since tactics may produce undesirably large terms. Lets take a look at an example of constructing a simple proof of the commutativity of the conjunction of two propositions A and B . First, we need to tell COQ that we are going to enter in the proof mode, by using the `Theorem` command.

```
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 of the proof is to move the universally quantified propositions A and B to the context, together with the hypothesis $A \wedge B$:

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

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

```
=====
B /\ A
```

Next, we deconstruct the hypothesis H and obtain isolated terms A and B holding in the current proof context. 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 *A* and *B* hold, we have to deconstruct the goal in order to isolate each of the components of the conjunction. This is done by using the tactic `constructor` that applies, in this case, the unique constructor `and`, yielding two new sub-goals, one for proving *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 looks into the context and notices that both *A* and *B* are 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 in fact an inhabitant of the type of the theorem that we have allegedly just 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 they are checked once more at the end of the proof. When using `Qed` the proof term becomes *opaque* and cannot be unfolded and subjected to reductions. In order to have the contrary behaviour, the user must use the command `Defined` instead of `Qed` to terminate the proof.

2.1.4 Well-founded Recursion

As pointed out earlier, all the functions defined in COQ must be terminating. The usual approach is to implement functions through the `Fixpoint` command and using one of the arguments as the structurally recursive argument. However, this is not possible to be employed in 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, which are clearly documented in [15]. Here we will consider the particular technique for translating a general recursive functions into a equivalent *well-founded recursive function*.

A given binary relation \mathcal{R} over a set S is said to be *well-founded* if for all element $x \in S$, 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 ensures the inexistence of

infinite 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 R .

The definition of Acc is inductively defined, and so it can be used as the structurally recursive argument in the definition of functions. Current versions of COQ provides two high level commands that ease the burden of manually constructing a recursive function over Acc predicates: the command `Program` [96, 97] and the command `Function` [14]. The command `Function` allows the user to explicitly specify what is the recursive measure for the function being implemented. In order to give an insight on how we can use `Function` to program non-structurally recursive functions, we will present different implementations of the function that adds two natural numbers. A first way of implementing such a function is as follows:

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

Another way of implementing `sum` is by instructing `Function` to consider explicitly the recursive argument as being a term that proves that the relation $<$ 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 implementation of `sum` is identical to the implementation of `siml`, 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 asks for a proof that the relation `<` is well founded. Both obligations are discharged automatically due to `auto` tactics, with the help of know lemmas and theorems from the database `arith`. The third and final way of building functions using `Function` is by using the `struct` annotation. In this case, the definition will be carried out as structurally recursive function, like if it was defined using `Fixpoint`.

```
Function sum2 (x:nat) (y:nat) {struct x}:nat :=
  match x with
  | 0 => y
  | S m => S (sum2 m y)
  end.
Proof.
  abstract (auto with arith).
  exact lt_wf.
Defined.
```

Besides allowing more general definitions of recursive functions than the usage of `Fixpoint` allows, 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 proofs of well-founded induction with a given relation is usually an issue in COQ. Such computations may take too much time to compute due to the complexity of the proof term involved. One way to get around is to use a technique proposed by Barras, whose idea is to add sufficient `Acc_intro` constructors, in a lazy way, on top a `Acc` term, so that this 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 the function is the following:

```
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
  | 0 => wf
  | S m => fun x => Acc_intro x (fun y _ => guard m (guard m wf) y)
  end.
```

In each recursive call, when matching a term `Acc x H` constructed by the `guard` function,

the reduction mechanisms will find only `Acc_intro` terms, instead of a complex proof term. This increases computation considerably and yields better performance for the implemented function. For exemplifying how the function `guard` can be used when using the vernacular `Function`, we present a re-implementation of `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
  | 0 => y
  | m => S (sum1 (m-1) y)
end.
Proof.
  abstract (auto with arith).
  exact (guard 100 _ lt_wf).
Defined.
```

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. Here we enumerate only the features that are more relevant to the work presented in this thesis:

- an extraction mechanism, first introduced by Paulin-Morhing [86], by Paulin-Morhing and Werner [88], and also by Letouzey [77]. This mechanism allows users to extract functional programs in OCaml, in Haskell or in Scheme directly from CoQ developments. Based on the distinction between informative and logical types, extraction erases the logical contents and translates the informational ones into the functional languages mentioned above;
- it supports *type classes*, which extends the concept of 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 Orly [99] without extending the underlying CoQ type system, by relying on *dependent records*;
- a module system developed by Chrzaszcz [27] which allows users to conduct structured developments in a similar fashion to the one available in OCaml;
- a *coercion* mechanism that automatically provides a notion of sub-typing;

- a new general rewriting mechanism implemented by Sozeau [98] 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 4 and 5 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 [74] and described in detail by Lescuyer in [75].

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 the following:

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 [40]. 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 then apply it into an adequate functor. But, 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, and so they can be used like any other value, such like natural numbers;
- the development contains a plugin that provides a new vernacular command, `Generate OrderedType`, that automatically tries to construct all operations and proofs required to register an instance that the type under consideration is an ordered type. This command usually succeeds with any inductively defined type with no dependent arguments;
- sets of sets of a given type are always available for free due to the way the specification of finite sets is conceived since it contains a component guaranteeing that the set under

consideration itself is an ordered type.

The type class defining an ordered type is defined by the class `OrderedType`, containing the following parameters: a type `A`; an equality `_eq` and an order and `_lt`; proofs that `_eq` is an equivalence relation and that `_lt` is a strict order; a computable comparison function over values of `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 these 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`. The last function of the code presented below is the function `all_smaller` whose goal 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 than a given natural number `n`.

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

```
Fixpoint nat_compare (n m : nat) :=
  match n, m with
  | 0, 0 ⇒ Eq
```



```

| _, 0 ⇒ Gt
| 0, _ ⇒ 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
}.

```

```

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 3 and 4 target program verification of sequential and parallel programs, respectively. Both the works are closely related to the well known *Floyd-Hoare logic*, usually called solely Hoare logic, which resulted from the works of Floyd [41] and Hoare [51]. Using Hoare logic we are able to prove a program correct by applying a finite set of inference rules to an initial program specification of the form

$$\{P\} C \{Q\}, \quad (2.1)$$

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 will obligatorily terminate in a state where the assertion Q holds. The assertions P and Q are usually called *preconditions* and *postconditions*, respectively.

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

$$\begin{aligned}
 C, C_1, C_2 ::= & \text{skip} \\
 & | x := e \\
 & | C_1 ; C_2 \\
 & | \text{if } b \text{ then } C_1 \text{ else } C_2 \\
 & | \text{while } b \text{ do } C_1 \text{ end,}
 \end{aligned}$$

where x is a variable of the language, and e is an arithmetic expression. For the simplicity of the presentation, here we omit the language of expressions and assume that variables of IMP can have only one of two types: integers and Booleans. **Programs of IMP** are interpreted in a standard *small-step structural operational semantics* [90], where there exists the notion of state (a set of variables and corresponding assign values) and programs are executed by means of a *evaluation function* that take *configurations* $\langle C, s \rangle$ into new configurations **The** expression

$$\langle C, s \rangle \Longrightarrow^* \langle \text{skip}, s' \rangle \quad (2.2)$$

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

$$\frac{}{\langle x := e, s \rangle \Longrightarrow \langle \text{skip}, s[e/x] \rangle} \text{ (SKIP)}$$

$$\frac{\langle C_1, s \rangle \Longrightarrow \langle C'_1, s' \rangle}{\langle C_1; C_2, s \rangle \Longrightarrow \langle C'_1; C_2, s' \rangle} \text{ (ASSGN)}$$

$$\frac{}{\langle \text{skip}; C_2, s \rangle \Longrightarrow \langle C_2, s \rangle} \text{ (SEQSKIP)}$$

$$\frac{\llbracket b \rrbracket(s) = \mathbf{true}}{\langle \text{if } b \text{ then } C_1 \text{ else } C_2, s \rangle \Longrightarrow \langle C_1, s \rangle} \text{ (SEQSTEP)}$$

$$\frac{\llbracket b \rrbracket(s) = \mathbf{false}}{\langle \text{if } b \text{ then } C_1 \text{ else } C_2, s \rangle \Longrightarrow \langle q, s \rangle} \text{ (IF)}$$

$$\frac{}{\langle \text{while } b \text{ do } C \text{ end}, s \rangle \Longrightarrow \langle \text{if } b \text{ then } (C ; \text{while } b \text{ do } C \text{ end}) \text{ else skip}, s \rangle} \text{ (WHILE)}$$

The function $\llbracket b \rrbracket$ is a function that denotationally evaluates Boolean expressions in states, and returns the corresponding Boolean value. This kind of semantics, and alternative ones can be seen in standard textbooks about programming language semantics such as [81, 102, 48].

Hoare Logic's Proof System

Hoare logic is a proof system made up of a set of inference rules that correspond to fundamental laws about programs. Each inference rule consists of zero or more premisses and a unique conclusion.

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 leaf nodes of 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 a correctness proof of a program.

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

Skip. the following rule simply states that the pre- and post-conditions of a **skip** program must be the same, since this command does not change the state of computation.

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

Assignment. if we want to show that b holds after the assignment of e to x , we must show that $b[x/e]$ (the substitution of the free occurrences of x by e , in b) holds before the assignment. We will apply this rule backwards. We know b and we wish to find a

pre-condition that makes b true after the assignment $x := e$.

$$\frac{}{\{P[e/x]\} x := e \{P\}} \text{ (HL-ASSGN)}$$

Composition. if C_1 brings a state satisfying P into a state satisfying Q' , and that if C_2 brings a state satisfying Q' into a state satisfying Q , then if P is true, the execution of C_1 followed by C_2 , brings the program into a state satisfying the post-condition Q .

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

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 are enforced depending on whether b is true or false. This additional information is often crucial for completing the respective sub-proofs.

$$\frac{\{b \wedge P\} C_1 \{Q\} \quad \{b \wedge \neg P\} C_2 \{Q\}}{\{P\} \text{ if } b \text{ then } C_1 \text{ else } C_2 \{Q\}} \text{ (HL-IF)}$$

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.

$$\frac{\{b \wedge P\} C \{P\}}{\{P\} \text{ while } b \text{ do } C \{ \neg b \wedge P \}} \text{ (HL-WHILE)}$$

Weakening. if we have proved that $\{P'\} C \{Q'\}$, and if we have a well-formed-formula P that implies P' , and also we have a well-formed-formula Q' such that Q' implies Q , then we can strengthen the pre-condition and weaken the post-condition.

$$\frac{P \rightarrow P' \quad \{P'\} C \{Q'\} \quad Q' \rightarrow Q}{\{P\} C \{Q\}} \text{ (HL-WEAK)}$$

We denote this proof system by HL. We say that a Hoare triple $\{P\} C \{Q\}$ is *derivable* in HL, and write $\vdash_{\text{HL}} \{P\} C \{Q\}$ 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 $\mathcal{A} \vdash_{\text{HL}} \{P\} C \{Q\}$. 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) [36], 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:

$$\begin{aligned}
 C, C_1, C_2 ::= & \text{skip} \\
 & | x := e \\
 & | C_1 ; \{P\} C_2 \\
 & | \text{if } b \text{ then } C_1 \text{ else } C_2 \\
 & | \text{while } b \text{ do } \{I\} C \text{ end.}
 \end{aligned}$$

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

$$\begin{aligned}
 & \frac{P \rightarrow Q}{\{P\} \text{ skip } \{Q\}} \text{ (HL-ANNSKIP)} \\
 & \frac{P \rightarrow Q[e/x]}{\{P[x/e]\} x := e \{Q\}} \text{ (HL-ANNASSGN)} \\
 & \frac{\{P\} C_1 \{Q'\} \quad \{Q'\} C_2 \{Q\}}{\{P\} C_1; \{Q'\} C_2 \{Q\}} \text{ (HL-ANNSEQ)} \\
 & \frac{\{b \wedge P\} C_1 \{Q\} \quad \{\neg b \wedge P\} C_2 \{Q\}}{\{P\} \text{ if } c \text{ then } C_1 \text{ else } C_2 \{Q\}} \text{ (HL-ANNIF)}
 \end{aligned}$$

$$\frac{b \rightarrow I \quad \{I \wedge b\} C \{I\} \quad I \wedge \neg b \rightarrow Q}{\{P\} \text{ while } b \text{ do } \{I\} C \{Q\}} \text{ (HL-ANNWHILE)}$$

The system **HL_a** can be proved to infer the same proof trees as the system **HL**. Such proof is available in the work of Frade and Pinto [42], 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 Why system [17, 16] and the Boogie system [31, 32].

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 [60], 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. More recently, regular expressions found also applications in run-time monitoring of programs [94, 95].

In this chapter we describe the mechanisation, in the proof assistant COQ, of a considerable fragment of regular language theory and also present the implementation and correctness of a decision procedure for regular expressions using the notion of derivative of a regular expression, an alternative to the construction of automata.

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 from Hopcroft and Ullman [54], or the one of Kozen [63]. Along the 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 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 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.

Example 1. Consider the following alphabet $\Sigma = \{a, b, c\}$. This alphabet is encoded in COQ by developing the corresponding COQ module, as follows:

```
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-letter 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 words defined on Σ is the set

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

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 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 Σ^* , is inductively defined as follows: the empty word ϵ is an element of Σ^* and, if $w \in \Sigma^*$ and $a \in \Sigma$, then aw is also a member of Σ^* . The set Σ^* 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 , by $\{\epsilon\}$, and by $\{a\}$. 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 language L_1 and L_2 is defined by

$$L_1 L_2 \stackrel{\text{def}}{=} \{wu \mid w \in L_1 \wedge u \in L_2\}. \quad (3.1)$$

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

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

considering that $n \in \mathbb{N}$. The *Kleene star* of a language L is the union of all the finite powers of L , that is,

$$L^* \stackrel{\text{def}}{=} \bigcup_{i \geq 0} L^i. \quad (3.3)$$

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 the 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 $L_1 \cup L_2$, $L_1 L_2$ and L_1^* are regular languages.

The predicate that determines whether or not a given language is regular 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 *).
```

Two languages L_1 and L_2 are equal if the sets of words that they represent are the same. 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 $w \in \Sigma^*$, and which we denote by a new $\mathcal{D}_w(L)$. The left-quotient is defined as follows:

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

Naturally, definition (3.4) can be specialised to words $w = a$, with $a \in \Sigma$. In this case, we refer to the left-quotient with respect a symbol a . The COQ definitions of LQ and LQw given below are, respectively, the notions of left-quotient with respect to a symbol, and with respect to a word.

Inductive $\text{LQ} (L:\text{language}) : A \rightarrow \text{language} :=$
 $| \text{in_quo} : \forall (x:\text{word}) (a:A), (a::x) \in L \rightarrow x \in (\text{LQ } L \ a)$
Notation " $x \% \text{Lq } y$ " $:= (\text{LQ } x \ y)$.

Inductive $\text{LQw} (L:\text{language}) : \text{word} \rightarrow \text{language} :=$
 $| \text{in_quow} : \forall (w_1 \ w_2:\text{word}), (w_1 ++ w_2) \in L \rightarrow w_2 \in (\text{LQw } L \ w_1)$
Notation " $x \% \text{Lqw } y$ " $:= (\text{LQw } x \ y)$.

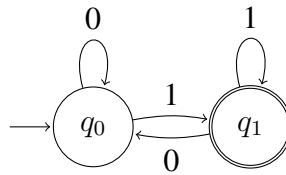
3.1.2 Finite automata

A *deterministic finite automata* (DFA) is a 5-tuple $D = (Q, \Sigma, \delta, q_0, F)$ such that Q is the set of *states*, Σ is the alphabet, $\delta : Q \times \Sigma \rightarrow Q$ 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, 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:

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

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 $\mathcal{L}(D)$, 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 [60]. 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:

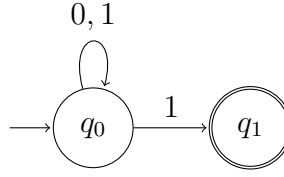
$$\begin{aligned}\hat{\delta}(q_0, 101) &= \hat{\delta}(\delta(q_0, 1), 01) \\ &= \hat{\delta}(q_1, 01) \\ &= \hat{\delta}(\delta(q_1, 0), 1) \\ &= \hat{\delta}(q_0, 1) \\ &= \hat{\delta}(\delta(q_0, 1), \epsilon) \\ &= \hat{\delta}(q_1, \epsilon) \\ &= q_1\end{aligned}$$

Two DFAs D_1 and D_2 are *equivalent* if the languages they recognise are the same, that is, whenever $\mathcal{L}(D_1) = \mathcal{L}(D_2)$. 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 automata* (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, $\delta : Q \times \Sigma \rightarrow 2^Q$ 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 an NFA that recognises all the sequences of 0's and 1's that end with an 1.

Note that the main difference between the definition of DFA and that of NFA is their 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 NFA, the extended transition function is inductively as follows:

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

Figure 3.2 NFA accepting sequences of 0's and 1' that end with an 1.

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

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

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

As with DFAs, 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 [39], of Briais [20], and of Braibant and Pous [18, 19].

3.1.3 Regular expressions

Let Σ be an alphabet. A regular expression $\alpha, \beta, \alpha_1, \beta_1, \dots$, over Σ is inductively defined as follows: the constants 0 and 1 are regular expressions; all symbols $a \in \Sigma$ are regular expressions; if α and β are regular expressions, then the *union* $\alpha + \beta$ and the *concatenation* $\alpha\beta$; finally, if α is a regular expression, then so is its *Kleene star* α^* . 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
| re1 : re
| re_sy : A → re
| re_union : re → re → re
| re_conc : re → re → re
| re_star : re → re.

Notation "0" := re0.
Notation "1" := re1.
Notation "a" := (re_sy a).
Infix "+" := re_union.
Infix "." := re_conc.
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: $\mathcal{L}(0) \stackrel{\text{def}}{=} \emptyset$ and $\mathcal{L}(1) \stackrel{\text{def}}{=} \{\epsilon\}$; if α and β are regular expressions, then the expressions $\mathcal{L}(\alpha + \beta) \stackrel{\text{def}}{=} \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$ and $\mathcal{L}(\alpha\beta) \stackrel{\text{def}}{=} \mathcal{L}(\alpha)\mathcal{L}(\beta)$; finally, if α is a regular expression, then $\mathcal{L}(\alpha^*) \stackrel{\text{def}}{=} \mathcal{L}(\alpha)^*$. 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. For reasons of completeness of the formalisation, 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 symbols from the alphabet Σ that occur in α (repeated symbols included). We say that a regular expression α is *nullable* if $\epsilon \in \mathcal{L}(\alpha)$, and *non-nullable* otherwise. We say that the regular expressions α and β are *equi-nullable* if $\varepsilon(\alpha) = \varepsilon(\beta)$. The following function¹ recursively determines if a given regular expression is nullable or not:

```

Fixpoint nullable( $\alpha$ :re) : bool :=
  match  $\alpha$  with
  | re0  $\Rightarrow$  false
  | re1  $\Rightarrow$  true
  | re_sy _  $\Rightarrow$  false
  | re_star _  $\Rightarrow$  true
  | re_union  $\alpha_1$   $\alpha_2$   $\Rightarrow$  nullable  $\alpha_1$  || nullable  $\alpha_2$ 
  | re_conc  $\alpha_1$   $\alpha_2$   $\Rightarrow$  nullable  $\alpha_1$  && nullable  $\alpha_2$ 
  end.
Notation " $\varepsilon(y)$ " := (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 : $\forall \alpha:\text{re}, \varepsilon(\alpha) = \text{true} \leftrightarrow \epsilon \in \alpha$.

Theorem null_correct_false : $\forall \alpha:\text{re}, \varepsilon(\alpha) = \text{false} \leftrightarrow \epsilon \notin \alpha$.

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

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

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  $\rightarrow$  language :=
| in_sre_lang :  $\forall (S:\text{set re}) (w:\text{word}) (\alpha:\text{re}),$ 
   $\alpha \in S \rightarrow w \in \alpha \rightarrow w \in (\text{SreL } S).$ 
Notation " $\mathcal{L}(s)$ " := (SreL  $s$ ).

```

The sum of a set of regular expressions $S = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$ is defined by

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

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

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

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

Nullability extends to sets of regular expressions in a straightforward way: a set S is nullable if $\varepsilon(\alpha)$ evaluates positively, that is, $\varepsilon(\alpha) = \text{true}$ 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 next definition.

Definition `null_set (S:set re) := fold (fun α:re => orb (ε(α))) S false.`

Notation " $\varepsilon(S)$ " := `(null_set S)`.

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

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

We usually omit the operator \odot whenever it is clear from the context, and write $S\alpha$ instead. The definition Equation (3.7) is defined in COQ in the obvious way:

Definition `fold_conc (S:set re) (α:re) := map (fun β => βα) S.`

Definition `dsr (α:re) (S:set re) : set re :=`
`match α with`
`| 0 => ∅`
`| 1 => S`
`| _ => fold_conc S α`
`end.`

Notation " $\alpha \odot S$ " := `(dsr α S)`.

3.1.4 Kleene Algebra

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

$$x + x = x \quad (3.8)$$

$$x + 0 = 0 \quad (3.9)$$

$$x + y = y + x \quad (3.10)$$

$$x + (y + z) = (x + y) + z \quad (3.11)$$

$$0x = 0 \quad (3.12)$$

$$1x = x \quad (3.13)$$

$$x(yz) = (xy)z \quad (3.14)$$

$$x(y + z) = xy + xz \quad (3.15)$$

$$(x + y)z = xz + yz. \quad (3.16)$$

The natural partial ordering on such a semiring is $x \leq y \Leftrightarrow x + y = y$. A *Kleene algebra* (KA) is an algebraic structure $(K, +, \cdot, *, 0, 1)$ 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:

$$1 + pp^* \leq p^* \quad (3.17)$$

$$1 + p^*p \leq p^* \quad (3.18)$$

$$q + pr \leq r \rightarrow p^*q \leq r \quad (3.19)$$

$$q + rp \leq r \rightarrow qp^* \leq r \quad (3.20)$$

The algebraic structure $(\text{RE}_\Sigma, +, \cdot, *, 0, 1)$ is a KA, namely, the free KA on the generator Σ (the alphabet under consideration). The standard model of KA is the model $(\text{RL}_\Sigma, \cup, \cdot, *, \emptyset, \{\epsilon\})$, where RL_Σ is the set of all regular languages over Σ . Kozen proved [62] the completeness of KA with respect to this model through an algebraic treatment of the classical approach to decide regular expression equivalence using automata. 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* [23] of a regular expression α was introduced by Brzozowski in the 60's, and was motivated by the construction of sequential circuits directly from regular

expressions extended with intersection and complement. In the same decade, Mirkin [80] introduced the notion of *prebase* and *base* of a regular expression as a method to construct NFAs recognising the language denoted by a given regular expression. His definition is a generalisation of Brzozowski's derivatives for NFAs and was independently re-discovered almost thirty years later by Antimirov [10], which coined it as *partial derivative* of a regular expression.

Brzozowski's Derivatives

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

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

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 those words.

The notion of derivative can be naturally extended to words in the following way:

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

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

$$\varepsilon(w^{-1}(\alpha)) = \text{true}. \quad (3.21)$$

Symmetrically, we can conclude that

$$\varepsilon(w^{-1}(\alpha)) = \text{false}. \quad (3.22)$$

implies that $w \notin \mathcal{L}(\alpha)$. The proof of equations (3.21) and (3.22) are obtained by induction on the length of the word w and some simple properties of the membership of the empty word. The next example, borrowed from [84], shows how matching of strings is easily computed using derivatives.

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

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

Now, by testing the nullability of the resulting regular expression b^* , we obtain $\varepsilon(b^*) = \text{true}$. 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 $\Sigma = \{a, b\}$, let $\alpha \stackrel{\text{def}}{=} ab^*$, and let $w = aab$. The word w is not accepted by the regular expressions α , since the computation of Brzozowski's derivative leads to

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

and, by testing the nullability of the resulting regular expression 0 , we obtain $\varepsilon(0) = \text{false}$. Thus, by (3.22), $w \notin \mathcal{L}(\alpha)$.

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

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

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

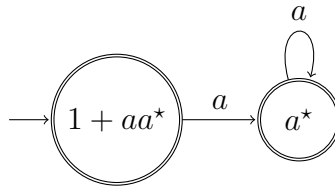
We can use $D(\alpha)$ to build a DFA that accepts the language denoted by α . The *derivative automaton* of α , denoted by $\mathcal{D}(\alpha)$, is the DFA defined as follows:

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

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

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

represented by the following transition diagram:



Partial Derivatives

Partial derivatives were introduced by Antimirov [10] and are a generalisation of Brzozowski's derivatives to NFAs. In fact, this notion was first introduced by Mirkin in [80], but only latter was proved by Champarnaud and Ziadi [25] that both notions coincide. Let α be a regular expression and let $a \in \Sigma$. The set $\partial_a(\alpha)$ of *partial derivatives* of the regular

expression α with respect to a is inductively defined as follows:

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

The operation of partial derivation is naturally extended to sets of regular expressions, as follows. Let S be a set of regular expressions, and let $a \in \Sigma$. 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

$$\mathcal{L}(\partial_a(\alpha)) = \mathcal{D}_a(\mathcal{L}(\alpha)). \quad (3.23)$$

Partial derivatives are implemented in COQ by the recursive function `pdrv` presented below. Lemma `pdrv_correct` proves the property (3.23). Finally, the extension of partial derivatives to finite sets of regular expressions is obtained in the expected way, as given by definition `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 a) ∪ (pdrv y a)
  | x · y ⇒ match ε(x) with
    | false ⇒ (pdrv x a) ∘ y
    | true ⇒ (pdrv x a) ∘ y ∪ (pdrv y a)
  end
  | x* ⇒ (pdrv x a) ∘ x*
end
Notation "∂a(α)" := (pdrv α a).

```

Lemma `pdrv_correct` : $\forall \alpha : \text{re}, \forall a : A, \mathcal{L}(\partial_a(\alpha)) == \text{LQ } \alpha \ a.$

Definition `pdrv_set` ($s : \text{set re}$) ($a : A$) := `fold` (`fun` $x : \text{re} \Rightarrow \text{union } (\partial_a(x))$) $s \ \emptyset.$

Notation `"x %dS y"` := (`pdrv_set` $x \ y$) (`at level 60`).

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

$$\begin{aligned} \partial_\varepsilon(\alpha) &\stackrel{\text{def}}{=} \{\alpha\}, \\ \partial_{wa}(\alpha) &\stackrel{\text{def}}{=} \partial_a(\partial_w(\alpha)). \end{aligned} \tag{3.24}$$

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,

$$\varepsilon(\partial_w(\alpha)) = \text{true} \leftrightarrow w \in \mathcal{L}(\alpha), \tag{3.25}$$

$$\varepsilon(\partial_w(\alpha)) = \text{false} \leftrightarrow w \notin \mathcal{L}(\alpha). \tag{3.26}$$

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

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

Now, by testing the nullability of the resulting set of regular expression $\{b^*\}$, we conclude that $\varepsilon(b^*) = \text{true}$. Thus, by (3.25), $w \in \mathcal{L}(\alpha)$.

Example 10. Let $\Sigma = \{a, b\}$, let $\alpha \stackrel{\text{def}}{=} ab^*$, and let $w = aab$. The word derivative of α with

respect to w corresponds to the following computation:

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

and, by testing the nullability on the empty set resulting from the derivation, by (3.26), leads to the conclusion that $w \notin \mathcal{L}(\alpha)$.

The implementation in COQ of (3.24) 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, (3.24) is implemented as a structurally recursive function as follows:

```

Reserved Notation "x %dw y" (at level 60).
Fixpoint pdrv_iw (sre:set re)(w:ilist) : set re :=
  match w with
  | <[] => sre
  | xs <:: x => (sre %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 $w \in \Sigma^*$. 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

$$PD(\alpha) \stackrel{\text{def}}{=} \bigcup_{w \in \Sigma^*} (\partial_w(\alpha)). \quad (3.27)$$

Antimirov proved [10] that, given a regular expression α , the set $PD(\alpha)$ is always finite and its cardinality has an upper bound of $|\alpha|_\Sigma + 1$, that is, $|PD(\alpha)| \leq |\alpha|_\Sigma + 1$.

Mirkin's Construction

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

$$\begin{aligned} L_i &= \left(\bigcup_{j=1}^k \{a_j\} L_{ij} \right) \cup \begin{cases} \{\epsilon\} & \text{if } \epsilon \in L_i, \\ \emptyset & \text{otherwise.} \end{cases} \\ L_{ij} &= \bigcup_{m \in I_{ij}} L_m, \end{aligned} \quad (3.28)$$

with $i \in \{0, \dots, n\}$ and $I_{ij} = \delta(i, a_j) \subseteq \{0, \dots, n\}$. Conversely any set of languages $\{L_0, \dots, L_n\}$ that satisfies the set of equations (3.28) defines an NFA with initial state L_0 . In particular if L_0, \dots, L_n are denoted by the regular expressions $\alpha_0, \dots, \alpha_n$, respectively, then the following holds.

$$\begin{aligned} \alpha &\equiv \alpha_0 \\ \alpha_i &\sim a_1 \alpha_{i1} + \dots + a_k \alpha_{ik} + \begin{cases} 1 & \text{if } \varepsilon(\alpha) = \text{true}, \\ 0 & \text{otherwise.} \end{cases} \\ \alpha_{ij} &\sim \sum_{m \in I_{ij}} \alpha_m, \end{aligned} \quad (3.29)$$

with $i \in \{0, \dots, n\}$ and $I_{ij} \subseteq \{0, \dots, n\}$. Given a regular expression α , the task of finding a set of regular expressions that satisfies (3.29) is tantamount to find a NFA whose language it recognises is the language of α . Mirkin [80] denoted by *support* any set of regular expressions that satisfies (3.29), provided an algorithm for calculating it, and proved that such solution's size is always 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 [25] introduced an elegant definition of calculating² a support for a given regular expression α . Such function, denoted by $\pi(\alpha)$, is recursively defined as follows.

$$\begin{aligned} \pi(\emptyset) &\stackrel{\text{def}}{=} \emptyset & \pi(\alpha + \beta) &\stackrel{\text{def}}{=} \pi(\alpha) \cup \pi(\beta) \\ \pi(\varepsilon) &\stackrel{\text{def}}{=} \emptyset & \pi(\alpha\beta) &\stackrel{\text{def}}{=} \pi(\alpha)\beta \cup \pi(\beta) \\ \pi(\sigma) &\stackrel{\text{def}}{=} \{\varepsilon\} & \pi(\alpha^*) &\stackrel{\text{def}}{=} \pi(\alpha)\alpha^*. \end{aligned} \tag{3.30}$$

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 that $|PD(\alpha)| \leq |\alpha|_\Sigma + 1$.

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

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

The cardinality argument also holds, since

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

The equation system (3.29), 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,
```

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

$$\begin{aligned}
& \beta \in s \wedge w \in \mathcal{L}(a\beta) \wedge \\
& a\beta \subseteq \mathcal{L}(\alpha) \wedge s' \subseteq s \wedge \\
& \beta == \mathcal{L}(s') \rightarrow w \in (\text{Support } \alpha \ s).
\end{aligned}$$

```

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 " $\pi(x)$ " := (PI x).

Theorem lang_eq_Support : $\forall \alpha:\text{re}, \alpha == \text{Support } \alpha \ (\pi(\alpha))$.

Definition PD($\alpha:\text{re}$) := $\{\alpha\} \cup (\pi(\alpha))$.

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

Theorem all_pdrv_in_PI : $\forall (\alpha:\text{re}) (a:A), \partial_a(\alpha) \subseteq \pi(\alpha)$.

Theorem PI_upper_bound : $\forall \alpha:\text{re}, \text{cardinal } (\pi(\alpha)) \leq |\alpha|_\Sigma$.

Lemma all_wpdrv_in_PD : $\forall (w:\text{word}) (\alpha\beta:\text{re}), \beta \in \partial_w(\alpha) \rightarrow \beta \in \text{PD}(\alpha)$.

Theorem PD_upper_bound : $\forall \alpha:\text{re}, \text{cardinal } (\text{PD}(\alpha)) \leq |\alpha|_\Sigma + 1$.

The proofs of the previous 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 [10, 25].

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 , it holds that

$$\alpha \sim \varepsilon(\alpha) \cup \bigcup_{a \in \Sigma} a \left(\sum \partial_a(\alpha) \right), \quad (3.31)$$

by induction on α and the properties of partial derivatives. We overload the notation $\varepsilon(\alpha)$, in the sense in the current context, $\varepsilon(\alpha) = \{1\}$ if α is nullable, and $\varepsilon(\alpha) = \emptyset$, otherwise.

Following the equivalence (3.31), checking if $\alpha \sim \beta$ is tantamount to check the equivalence

$$\varepsilon(\alpha) \cup \bigcup_{a \in \Sigma} a \left(\sum \partial_a(\alpha) \right) \sim \varepsilon(\beta) \cup \bigcup_{a \in \Sigma} a \left(\sum \partial_a(\beta) \right). \quad (3.32)$$

This will be an essential ingredient to our decision method because deciding if $\alpha \sim \beta$ resumes to check 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 $w \in \Sigma^*$ belongs to $\mathcal{L}(\alpha)$ is equivalent to check syntactically that $\varepsilon(\partial_w(\alpha))$ is nullable, we obtain the following equivalence:

$$(\forall w \in \Sigma^*, \varepsilon(\partial_w(\alpha)) = \varepsilon(\partial_w(\beta))) \leftrightarrow \alpha \sim \beta. \quad (3.33)$$

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

$$\varepsilon(\partial_w(\alpha)) \neq \varepsilon(\partial_w(\beta)) \rightarrow \alpha \not\sim \beta. \quad (3.34)$$

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 a point where we find a counter example 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), and that finds a counter-example like in (3.34) that shows that such bisimulation does not exist.

3.3 A Procedure for Regular Expression 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.* [5], which has its roots in a rewrite system proposed by Antimirov and Mosses [11] 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, which is known to be slower than computation of mainstream functional programming language.

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 regular expressions (S_α, S_β) such that $S_\alpha = \partial_w(\alpha)$ and $S_\beta = \partial_w(\beta)$, for some word $w \in \Sigma^*$. 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  $S \neq \emptyset$  do
3:      $(S_\alpha, S_\beta) \leftarrow POP(S)$ 
4:     if  $\varepsilon(S_\alpha) \neq \varepsilon(S_\beta)$  then
5:       return false
6:     end if
7:      $H \leftarrow H \cup \{(S_\alpha, S_\beta)\}$ 
8:     for  $a \in \Sigma$  do
9:        $(S'_\alpha, S'_\beta) \leftarrow \partial_a(S_\alpha, S_\beta)$ 
10:      if  $(S'_\alpha, S'_\beta) \notin H$  then
11:         $S \leftarrow S \cup \{(S'_\alpha, S'_\beta)\}$ 
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 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 counter-example (S_α, S_β) such that $\varepsilon(S_\alpha) \neq \varepsilon(S_\beta)$ was 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.* [6, 4, 7], where the

authors propose a functional algorithm that decides regular expression equivalence based on partial derivatives. This procedure is a functional formulation of the rewrite system proposed that Antimirov and Mosses proposed [11], but whose goal was 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 include 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 other differences 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 in-equivalence.

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

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

*The computation of EQUIVP for these particular α and β involves the construction of the new derivatives $s_1 = (\{1, b(ab)^*a\}, \{(ba)^*\})$ 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 if $\alpha = b^*a$ and $\beta = b^*ba$ are not equivalent. Considering $s_0 = (\{b^*a\}, \{b^*ba\})$, to prove so it is enough to check if*

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

*In this case, the computation of EQUIVP creates the new derivatives, $s_1 = (\{1\}, \emptyset)$ and $s_2 = (\{b^*a\}, \{a, b^*ba\})$, 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	$\{s_1, s_2\}$	$\{s_0\}$	$\varepsilon(s_1) = \text{false}$

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 $w \in \Sigma^*$. The type of derivatives is given by the definition of `Drv` as follows:

```
Record Drv ( $\alpha \beta$ :re) := mkDrv {
  dp :> set re * set re ;
  w : word ;
  cw : dp = ( $\partial_w(\alpha), \partial_w(\beta)$ )
}.
```

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 $(S_\alpha, S_\beta) = (\partial_w(\alpha), \partial_w(\beta))$. Using 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 $\alpha \beta$` , 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 ( $\alpha \beta$ :re) : Drv  $\alpha \beta$ .
  refine (Build_Drv ({ $\alpha$ }, { $\beta$ })  $\epsilon$  _).
  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 $\alpha \beta$` 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 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 an straightforward way to relate the result of the computation of `EQUIVP` with the (in-)equivalence of α and β : on one hand, if H is the set returned by `EQUIVP`, then checking the nullability its elements of H is tantamount to prove 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 $t : \text{Drv } \alpha \beta$, then $\varepsilon(t) = \text{false}$, which implies that the word $w \ t$ is a witness of in-equivalence, 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 set of `Drv` terms, and to the derivation with regard to a word, are given below. They are the definitions `Drv_pdrv`, `Drv_pdrv_set`, and `Drv_wpdrv`, respectively. These definition are presented below. 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 ( $\alpha \beta : \text{re}$ ) ( $x : \text{Drv } \alpha \beta$ ) ( $a : A$ ) : Drv  $\alpha \beta$ .
refine (match x with
  | mkDrv  $\alpha \beta$   $K$   $w$   $P \Rightarrow$  mkDrv  $\alpha \beta$  (pdrv  $K$   $a$ ) ( $w++[a]$ ) _
end) .
abstract ((* Proof that  $\partial_a(\partial_w(\alpha), \partial_w(\beta)) = (\partial_{wa}(\alpha), \partial_{wa}(\beta))$  *) ) .
Defined.

Definition Drv_pdrv_set ( $x : \text{Drv } \alpha \beta$ ) ( $s : \text{set } A$ ) : set (Drv  $\alpha \beta$ ) :=
  fold (fun  $y : A \Rightarrow$  add (Drv_pdrv  $x$   $y$ ))  $s$   $\emptyset$ .

Definition Drv_wpdrv ( $\alpha \beta : \text{re}$ ) ( $w : \text{word}$ ) : Drv  $\alpha \beta$ .
refine (mkDrv  $\alpha \beta$  ( $\partial_w(\alpha), \partial_w(\beta)$ )  $w$  _).
abstract ((* Proof that  $(\partial_w(\alpha), \partial_w(\beta)) = (\partial_w(\alpha), \partial_w(\beta))$  *) ) .
Defined.

```

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

```

Definition c_of_rep ( $x : \text{set re} * \text{set re}$ ) :=
  Bool.eqb (c_of_re_set (fst  $x$ )) (c_of_re_set (snd  $x$ )).

Definition c_of_Drv ( $x : \text{Drv } \alpha \beta$ ) := c_of_rep (dp  $x$ ).

```

```

Definition c_of_Drv_set (s:set (Drv  $\alpha$   $\beta$ )) : bool :=
  fold (fun x  $\Rightarrow$  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 in-equivalence 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, and which also corresponds to the *for-loop* from lines 8 to 12 of Algorithm 1.

```

Definition step (H S:set (Drv  $\alpha$   $\beta$ )) (sig:set A) :
  ((set (Drv  $\alpha$   $\beta$ ) * set (Drv  $\alpha$   $\beta$ )) * step_case  $\alpha$   $\beta$ ) :=
  match choose s with
  |None  $\Rightarrow$  ((H, S), termtrue  $\alpha$   $\beta$  H)
  |Some ( $S_\alpha, S_\beta$ )  $\Rightarrow$ 
    if c_of_Drv _ _ ( $S_\alpha, S_\beta$ ) then
      let H' := add ( $S_\alpha, S_\beta$ ) H in
      let S' := remove ( $S_\alpha, S_\beta$ ) S in
      let ns := Drv_pdrv_set_filtered  $\alpha$   $\beta$  ( $S_\alpha, S_\beta$ ) H' sig in
      ((H', ns  $\cup$  S'), proceed  $\alpha$   $\beta$ )
    else
      ((H, S), termfalse  $\alpha$   $\beta$  ( $S_\alpha, S_\beta$ ))
  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 triple $((H, S), \text{termfalse } \alpha \beta (S_\alpha, S_\beta))$, that serves as a witness of $\alpha \not\sim \beta$. If, on the contrary, S_α and S_β are equi-nullable, then `step` generates a new set of derivatives by the symbols $a \in \Sigma$, $(S'_\alpha, S'_\beta) = (\partial_a(S_\alpha), \partial_a(S_\beta))$, such that (S'_α, S'_β) are not elements 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  $\alpha$   $\beta$ ) (H:set (Drv  $\alpha$   $\beta$ ))
  (sig:set A) : set (Drv  $\alpha$   $\beta$ ) :=
  filter (fun y  $\Rightarrow$  negb (y  $\in$  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` below. 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 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 bounded by

$$2^{(|\alpha|_{\Sigma}+1)} \times 2^{(|\beta|_{\Sigma}+1)} + 1 \quad (3.35)$$

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

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

then the following relation

$$(2^{(|\alpha|_{\Sigma}+1)} \times 2^{(|\beta|_{\Sigma}+1)} + 1) - |H'| < (2^{(|\alpha|_{\Sigma}+1)} \times 2^{(|\beta|_{\Sigma}+1)} + 1) - |H|, \quad (3.36)$$

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 (3.36). It is proved by first showing that

$$\forall z, n, a, (z - |a|) < n \rightarrow \text{Acc}(\text{lim_card } z) a, \quad (3.37)$$

such that `Acc` is the definition, in COQ's type theory, of the notion of accessibility. The relation `lim_card` is the same as `lim_cardN`, but that uses Peano natural numbers, instead of the binary encoding of natural numbers of the type `N`. We use `N` as the main datatype here, because it is very efficient when compared to `nat`. The proof that (3.37) 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 (3.36). The encoding in COQ goes as follows:

Definition `MAXfst(α:re) := |α|Σ + 1.`
Definition `MAXsnd(β:re) := |β|Σ + 1.`
Definition `MAX(α β:re) := (2MAXfst(α) × 2MAXsnd(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 to `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, rendering `LLim` as 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  $\Rightarrow$  NotOk  $\alpha$   $\beta$  x
|termtrue h  $\Rightarrow$  Ok  $\alpha$   $\beta$  h
|progress  $\Rightarrow$  iterate  $\alpha$   $\beta$   $H'$   $S'$  sig (DP_upd  $\alpha$   $\beta$   $H$   $S$  sig  $D$ )
end.
Proof.
  abstract (apply DP_wf) .
  exact (guard r1 r2 100 (LLim_wf r1 r2)) .
Defined.

```

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

```

Inductive term_cases  $\alpha$   $\beta$  : Type :=
|Ok : set (Drv  $\alpha$   $\beta$ )  $\rightarrow$  term_cases  $\alpha$   $\beta$ 
|NotOk : Drv  $\alpha$   $\beta$   $\rightarrow$  term_cases  $\alpha$   $\beta$ .

```

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 in-equivalence.

The function `iterate` is recursively decreasing on a proof 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. As we have noted earlier, using `LLim` is not enough: we must also provide evidence that the sets S and H remain disjoint in all the recursive calls of `iterate` in order for `LLim` is true in all such calls. This is way `iterate` has the parameter D of type `DP`. The type `DP` packs together a proof that H and S are disjoint (in all recursive calls), and that all the elements in H are equi-nullable. The definition of type `DP` is the following:

```

Inductive DP ( $\alpha$   $\beta$ :re) ( $H$   $S$ : set (Drv  $\alpha$   $\beta$ )) : Prop :=
| is_dp :  $H \cap S = \emptyset \rightarrow$  c_of_Drv_set  $\alpha$   $\beta$   $H$  = true  $\rightarrow$  DP  $\alpha$   $\beta$   $H$   $S$ .

```

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 :  $\forall$  ( $h$   $s$  : set (Drv  $\alpha$   $\beta$ )) ( $sig$  : set  $A$ ), DP  $\alpha$   $\beta$   $h$   $s$   $\rightarrow$ 
  DP  $\alpha$   $\beta$  (fst (fst (step  $\alpha$   $\beta$   $h$   $s$   $sig$ ))) (snd (fst (step  $\alpha$   $\beta$   $h$   $s$   $sig$ ))).

```

As explained in Section 2.1.4, the `Function` vernacular produces proof obligations that we need to discharge in order to COQ's type checker to accept if. 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 : ∀ (h s : set (Drv r1 r2)) (sig : set A),
  DP _ _ h s → snd (step _ _ h s sig) = Process' _ _ →
  LLim _ _ (fst (fst (step _ _ 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 instead 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` builds the term of type `DP` that ensures that $\{(\{\alpha\}, \{\beta\})\} \cap \emptyset = \emptyset$ and $\varepsilon(\emptyset) = \text{false}$ hold. 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 expression (in-)equivalence we need not to consider a fixed alphabet Σ , since only the derivations by symbols that exist in the regular expressions being tested is important. The input alphabet can even be infinite.

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.

Correctness

The correctness of `equiv` with respect to language (in-)equivalence follows the same ideas of the proof that we have devised to ensure that from proving two statements: if `equivP` $\alpha \beta$ returns `true`, then $\alpha \sim \beta$; otherwise, `equivP` $\alpha \beta$ 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 $\Sigma = (\text{sySet } \alpha \cup \text{sySet } \beta)$, and the value $D = (\text{DP_1st } \alpha \beta)$, if it holds

$$\text{iterate } \alpha \beta \emptyset (\text{Drv_1st } \alpha \beta) \Sigma D = \text{Ok } \alpha \beta H, \quad (3.38)$$

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

$$\text{iterate } \alpha \beta \emptyset (\text{Drv_1st } \alpha \beta) \Sigma D = \text{NotOk } \alpha \beta \gamma, \quad (3.39)$$

then $\alpha \not\sim \beta$ must hold, with γ of type $\text{Drv } \alpha \beta$. In what follows, we describe the way we proved the previous two statements, in a separate way.

Correctness wrt in-equivalence. To prove that (3.39) implies non equivalence of two given regular expressions α and β , we must prove that the value γ in the returned term `NotOk` $\alpha \beta \gamma$ is a witness that there is a word $w \in \Sigma^*$ 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`.

Lemma 1. *Let α and β be regular expressions, let S and H be set of values of type $\text{Drv } \alpha \beta$, let Σ be an alphabet, and let D be a values of type $\text{DP } \alpha \beta S H$. If*

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

then $\varepsilon(\text{fst } \gamma) \neq \varepsilon(\text{snd } \gamma)$.

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

Lemma 2. *Let α and β be regular expressions, let S and H be set of values of type $\text{Drv } \alpha \beta$, let Σ be an alphabet, and let D be a values of type $\text{DP } \alpha \beta S H$. Hence,*

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

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

Theorem 1. *Let α and β be two regular expressions. It holds that*

$$\text{equivP } \alpha \beta = \text{false} \rightarrow \alpha \not\sim \beta.$$

Correctness wrt equivalence. In order to prove the correctness of `equivP` with respect to language equivalence, we proceed as follows. Suppose that `equivP` $\alpha \beta = \text{true}$. 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 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 selecte for further processing. This property is formally defined in COQ as follows:

Definition `invP` $(\alpha \beta : \text{re}) (H S : \text{set } (\text{Drv } \alpha \beta)) (sig : \text{set } A) :=$
 $\forall x : \text{Drv } \alpha \beta, x \in H \rightarrow \forall a : A, a \in sig \rightarrow (\text{Drv_pdrv } \alpha \beta \ x \ a) \in (H \cup S).$

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`.

Lemma 3. *Let α and β be two regular expressions. Let S, S', H , and H' be finite sets of values of type $\text{Drv } \alpha \beta$. Moreover, let Σ be an alphabet. If `invP` $H S$ holds, and*

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

then `invP` $H' S'$ also holds.

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

Lemma 4. *Let α and β be two regular expressions. Let S, H , and H' be finite sets of values of type $\text{Drv } \alpha \beta$, and let Σ be an alphabet. If `invP` $H S$ holds, and if*

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

then `invP` $H' \emptyset$ also holds.

In COQ, the two previous lemmas are defined as follows:

Lemma `invP_step` : $\forall H S \Sigma,$
 $\text{invP } \alpha \beta H S \Sigma \rightarrow \text{invP } (\text{fst } (\text{fst } (\text{step } \alpha \beta H S \Sigma)))$
 $(\text{snd } (\text{fst } (\text{step } \alpha \beta H S \Sigma))) \Sigma.$

Lemma `invP_iterate` : $\forall \alpha \beta H S \Sigma D x,$
 $\text{invP } \alpha \beta H S \Sigma \rightarrow$
 $\text{iterate } \alpha \beta H S \Sigma D = \text{Ok } \alpha \beta x \rightarrow$
 $\text{invP } \alpha \beta x \emptyset$

The lemmas 3 and 4 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` ($H S$:set (Drv $\alpha \beta$)) (s :set A) :=
 $(\text{Drv_1st } \alpha \beta) \in (H \cup S) \wedge$
 $(\forall x:\text{Drv } \alpha \beta, x \in (H \cup S) \rightarrow \text{c_of_Drv } \alpha \beta x = \text{true}) \wedge$
 $\text{invP } \alpha \beta H S s.$

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`. For that, we must first show two generic lemmas that will allow us to conclude that.

Lemma 5. *Let α and β be two regular expressions. Let H , H' , and S' be set of values of type $\text{Drv } \alpha \beta$. Finally, let Σ be an alphabet, and let D be a value of type $\text{DP } \alpha \beta H S$. If it holds that $\text{iterate } \alpha \beta H S \Sigma D = \text{Ok } \alpha \beta H'$, then is also holds that $H \subseteq H'$.*

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

From lemma 5 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 $\alpha \beta H$` .

Lemma 6. *Let α and β be two regular expressions, let H' be a finite set of values of type $\text{Drv } \alpha \beta$, let Σ an alphabet, and let D be a value of type $\text{DP } \alpha \beta \emptyset \{(\{\alpha\}, \{\beta\})\}$. Hence,*

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

Now, we proceed in the proof by showing that all the elements of the set packed in a value $\text{ok } \alpha \beta H'$ 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.

Lemma 7. *Let α and β be two regular expressions. Let H , H' , and S' be set of values of type $\text{Drv } \alpha \beta$. Finally, let Σ be an alphabet, and let D be a value of type $\text{DP } \alpha \beta H S$. If it holds that $\text{iterate } \alpha \beta H S \Sigma D = \text{ok } \alpha \beta H'$, then it holds that $\forall \gamma \in H', \varepsilon(\gamma) = \text{true}$.*

Using lemmas 6 and 8 we can establish the intermediate result that will take us to prove the correctness of `equivP` with respect to language equivalence.

Lemma 8. *Let α and β be two regular expressions. Let H , H' , and S' be set of values of type $\text{Drv } \alpha \beta$. Finally, let Σ be an alphabet, and let D be a value of type $\text{DP } \alpha \beta H S$. If it holds that $\text{iterate } \alpha \beta H S \Sigma D = \text{ok } \alpha \beta H'$, then $\text{invP_final } \alpha \beta H' \emptyset$.*

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.

Lemma 9. *Let α and β be two regular expressions. Let H' be a set of values of type $\text{Drv } \alpha \beta$. If it holds that $\text{invP_final } \alpha \beta H' \emptyset (\text{sySet } \alpha \cup \text{sySet } \beta)$, 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.

Theorem 2. *Let α and β be two regular expressions. Thus, if $\text{equivP } \alpha \beta = \text{true}$, then $\alpha \sim \beta$.*

Completeness

As in the case of the correctness of `equivP`, we have to prove two statements to ensure that `equivP` is complete with respect to language equivalence. Considering two regular expressions α and β , the statements that we need to show that are valid are that if $\alpha \sim \beta$ then `equivP` return `true`, and that if $\alpha \not\sim \beta$ then `iterate` must return `false`.

In order to obtain the proofs for the two aforementioned statements, it is enough to reason by contradiction. To prove the first one, we pattern match over the results of `equivP`, which is a Boolean value and therefore leads to two cases: if `equivP` returns `true`, then by theorem ?? this implies $\alpha \sim \beta$, which trivially solves the goal; if `equivP` returns `false`, then by theorem ??, this implies $\alpha \not\sim \beta$, but this is a contradiction since $\alpha \sim \beta$, thus trivially proving the second case. The proof for the second statement follows a similar reasoning.

Theorem 3. *Let α and β be regular expressions. Hence, if $\alpha \sim \beta$ then $\text{equivP } \alpha \beta = \text{true}$.*

Theorem 4. *Let α and β be regular expressions. Hence, if $\alpha \not\sim \beta$ then $\text{equivP } \alpha \beta = \text{false}$.*

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 Expression 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 the following:

```
Ltac re_inequiv :=
  apply equiv_re_false; vm_compute;
  first [ reflexivity | fail 2 "Regular expressions 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 in-equivalence, or a subset 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 the case of the goal representing a subset 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 usability within COQ's environment for deciding regular expression (in-)equivalence.

Tactic for Deciding Relational 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 relational equations, based on derivatives, was first pointed out by Nipkow and Krass [71]. In this section we adapt their idea to our development by providing a reflexive tactic that reduces relational equivalence to regular expression equivalence, and then apply the tactic for deciding regular expression 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 relation, the composition of relations, and the transitive and reflexive closure of a relation. If R_1 and R_2 are relations, denote the previous constants and operations on relations, by \emptyset , \mathcal{I} , $R_1 \cup R_2$, $R_1 \circ R_2$, and R^* , 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 that, we need to consider a function v such that maps each symbols 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 r 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 symbols 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:

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

Naturally, a word $w = a_1 a_2 \dots a_n$ can also be interpreted as a relation, namely, the composition 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 symbols a to the relation R_a , and that maps b to the relation R_b . The function $\text{reRelW } v \ w$ yields the relation $R_a \circ R_b \circ R_b \circ R_a \circ \mathcal{I}$, and its computation is summarised as

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

Now we connect the previous interpretations to regular expression equivalence and relational 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 $\text{ReL} \ (v:A \rightarrow \text{relation } B) : \text{re} \rightarrow \text{relation } B :=$
 $| \text{mkRel} : \forall \alpha:\text{re}, \forall w:\text{word},$
 $w \in \text{re2rel } \alpha \rightarrow \forall a \ b, (\text{reRelW } v \ w) \ a \ b \rightarrow \text{ReL } \alpha \ a \ b.$

If two regular expressions α and β are equivalent, then the interpretations of reRelW of their words must necessarily be the same. This means that the pairs (a, b) in $\text{ReL } v \ \alpha$ and in $\text{ReL } v \ \beta$ must also 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 10. Let α and β be regular expressions. Let v be a function that maps symbols to relation. Hence,

$$\alpha \sim \beta \rightarrow \text{ReL } v \ \alpha \sim_{\mathcal{R}} \text{ReL } v \ \beta.$$

In order to use theorem 10 to decide relational equivalence, we must relate ReL and reRel . But, as the next lemma show, both these notion end up being equivalent relations.

Lemma 11. Let α be a regular expression, and let v be a function mapping symbols of the alphabet to relations. Thus,

$$\text{reRel } v \ \alpha \sim_{\mathcal{R}} \text{ReL } v \ \alpha.$$

Together, lemma 10 and 11 allow us to prove that if two regular expressions are equivalent, then so are their interpretation on binary relations.

Theorem 5. *Let α and β be regular expressions. Let v be a function that maps symbols to relation. Hence,*

$$\alpha \sim \beta \rightarrow \text{reRel } v \alpha \sim_{\mathcal{R}} \text{reRel } v \beta.$$

The last lemmas and theorem are defined in our development as follows:

Lemma Main : $\forall \alpha:\text{re},$
 $\text{rel_eq } (\text{reRel } v \alpha) (\text{ReL } \alpha).$

Lemma Rel_red_Leq_aux : $\forall \alpha \beta:\text{re},$
 $\alpha \sim \beta \rightarrow \text{rel_eq } (\text{ReL } v \alpha) (\text{ReL } v \beta).$

Theorem Rel_red_Leq_final : $\forall \alpha \beta:\text{re},$
 $\alpha \sim \beta \rightarrow \text{rel_eq } (\text{reRel } v \alpha) (\text{reRel } v \beta).$

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. We consider the underlying alphabet the set of natural numbers, i.e., the values of type `nat`. This function is obtained directly from the individual in the equations that our tactic is intended to solve. For obtaining such function, we need the following definitions that are responsible for building and finding representatives of relations.

Definition STD($T:\text{Type}$) : `relation T` := `IdRel T`.

Definition emp_m($T:\text{Type}$) : `Map[nat, relation T]` :=
`@empty nat _ _ (relation T).`

Definition add_mm($T:\text{Type}$) ($x:\text{nat}$) ($r:\text{relation T}$) ($m:\text{Map[nat, relation T]}$) :=
`@add nat _ _ (relation T) x r m.`

Definition find_mm($T:\text{Type}$) ($x:\text{nat}$) ($m:\text{Map[nat, relation T]}$) :=
`match @find nat _ _ (relation T) x m with`
`| None \Rightarrow STD T`
`| Some $y \Rightarrow y$`
`end.`

Definition f($T:\text{Type}$) ($v:\text{Map[nat, relation T]}$) ($x:\text{nat}$) :=
`find_mm T x v.`

The functions `add_mm` and `find_mm` are responsible for adding and a map from a natural number to a relation, respectively. The next tactics construct a map `Map[nat, relation T]` by following the structure of the goal representing the relational equation. The type `Map` is provided by the `Containers` library [75], 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 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

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

provided that `v` is a function that maps the regular expression `!0` into the relation R_1 . Now, a final intermediate tactic is need 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 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 $R_1 \circ R_1^* \sim_{\mathcal{R}} R_1^* \circ R_1$ 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 a certified evidence that the decision algorithm suggested by Almeida *et. al.* [1] 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. These data sets were generated by the FAdo tool [3], 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. Each entry in the table is 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 *eq* – and inequivalent regular expressions – denoted by the rows labeled *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 allows us to conclude that the procedure is efficient, since it is able to decide the equivalence of regular expressions considerably large in less than 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 when $k = 2$ 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 is due to the fact that the derivations computed along the

k	$n = 25$		$n = 50$		$n = 100$	
	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>
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	$n = 250$		$n = 500$		$n = 1000$	
	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>
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`.

execution of the procedure tend to produce few 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. In the next section we present a comparison of the performances exhibited by our procedure with other developments 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 that spawn along three different interactive theorem provers, namely COQ, ISABELLE [83] and MATITA [13].

The most complete of the developments is the one of Braibant and Pous [18]: the authors formalised Kozens' completeness theorem for Kleene algebra [], and have developed efficient tactics to decide Kleene algebra equalities by 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 [56] for constructing automata from regular expressions, and the comparison is performed using Karp’s [53] 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 is also quite efficient due to a careful choice of the data structures involved.

The works that are more close to ours are the works of Coquand and Siles [30], and Nipkow and Krauss [71]. 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 formalised 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 [12] 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, since the other two COQ developments clearly exhibit worst performances. For technical reasons, we were not able to test the development of Asperti. In these experiments we have used dataset of 1000 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 memory.

It is clear from table 3.2 that the work of Braibant and Pous scales better than ours for larger families of regular expression but it is drastically slower than ours with respect to regular expression in-equivalence. For smaller families of regular expressions, our procedure is also faster than theirs. 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 analysing the performance of `equivP`.

alg./(k, n)	(2, 5)		(2, 10)		(2, 20)	
	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>
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)	
	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>
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)	
	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>	<i>eq</i>	<i>ineq</i>
equivP	2.211	0.048	9.957	0.121		0.149
ATBR	3.001	1.654	5.876	2.724		12.448

Table 3.2: Comparison of the performances.

3.5 Conclusions

In this chapter we have described the mechanisation, within the COQ proof assistant, of the procedure EQUIVP for deciding regular expression equivalence based in the notion of partial derivatives. This procedure decides the (in-)equivalence of regular expression by an iterated method of comparing the equivalence of the respective partial derivatives. The main advantage of our method, when compared to the ones based on Brzowski's derivatives, is that it does not requires normalisation modulo *ACI* 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 satisfying. Nevertheless, there exists always 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 Paron, Champarnaud and Mignot [24] extend the notion of partial derivative to handle these extended regular expressions and its addition to our formalisation will not, in principle, 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

KAT [64, 68] is an algebraic system that extends Kleene algebra [60], 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) [70, 65] 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 check if p is partially correct by equational reasoning in KAT. Moreover, some Horn formulas [46, 47] of KAT can be reduced into standard equalities which can then be decided automatically using one of the available methods [103, 69, 67].

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.

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

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

such that $(K, +, \cdot, *, 0, 1)$ 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 (3.8–3.20) and the following ones, for $b, c, d \in T$:

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

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

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

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

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

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

$$\overline{bc} = \bar{b} + \bar{c} \tag{4.7}$$

$$b\bar{b} = 0 \tag{4.8}$$

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

4.2 The Language Model of KAT

Primitive Tests, Primitive Actions, and Atoms

Let $\mathcal{B} = \{b_1, \dots, b_n\}$ be a non-empty whose elements are called *primitive tests*. Let $\bar{\mathcal{B}} = \{\bar{b} \mid b \in \mathcal{B}\}$ be the set such that each element $l \in \mathcal{B} \cup \bar{\mathcal{B}}$ is called a *literal*. An *atom* is a finite sequence of literals $l_1 l_2 \dots l_n$, such that each l_i is either b_i or \bar{b}_i , for $1 \leq i \leq n$, where $n = |\mathcal{B}|$. We will refer to atoms by $\alpha, \alpha_1, \alpha_2, \dots$, and 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.

Example 18. Let $\mathcal{B} = \{b_1 b_2\}$. The set of all atoms over \mathcal{B} is $\text{At} = \{b_1 b_2, b_1 \bar{b}_2, \bar{b}_1 b_2, \bar{b}_1 \bar{b}_2\}$.

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

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 $B = \{b_1 b_2, b_1 \bar{b}_2\}$.

Besides primitive tests we have also to consider a finite set of symbols representing atomic programs, whose role is the same of an alphabet in regular expressions. Such a set in KAT is called the set of *primitive actions* $\Sigma = \{p_1, \dots, p_m\}$.

In terms of the COQ development, we define a module signature `KAT_Alph` where we specify the type of primitive tests `bv`, of primitive actions `sy`, and of atoms `atom`. In this same module type, we specify the sets \mathcal{B} , Σ and At through the parameters `sigmaB`, `sigmaP`, and `atset`, respectively. Moreover, we assume the existence of the function `eval_atom` whose objective is to evaluate primitive tests with respect to atoms as defined by $\alpha \leq b$.

Module Type `KAT_Alph`.

```
Parameter sy bv atom : Type.
Declare Instance atom_ord : UsualOrderedType sy.
Declare Instance atom_ord : UsualOrderedType bv.
Declare Instance atom_ord : UsualOrderedType atom.
Parameter sigmaP : set sy.
Parameter sigmaB : set bv.
Parameter atset : set atom.
Parameter eval_atom_bv : atom → bv → bool.
```

End `KAT_Alph`.

In addition to the sets of primitive tests and primitive actions, we have decided to incorporate in the definition of `KAT_Alph` an abstract notion of atom and of the set of all atoms by the parameter `atset`. In this configuration, we are able to encode the theory of KAT without having to provide a concrete structure to atoms, except when instantiating `KAT_Alph` to test with concrete examples. The parameter `eval_atom_bv` specifies the function responsible for deciding if $\alpha \leq b$.

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 \mathcal{T} . In this setting, the operators \cdot , $+$ and $\overline{}$ are interpreted as Boolean operations of conjunction, 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 $e_1 e_2$, and their Kleene star e_1^* .

The set of all KAT terms is denoted by $\mathcal{K}_{\mathcal{B},\Sigma}$, and we denote syntactical equality between $e_1, e_2 \in \mathcal{K}_{\mathcal{B},\Sigma}$ by $e_1 \equiv e_2$. In COQ, the type of KAT terms and the type of tests is defined as expected by

```

Inductive test : Type :=
| ba0 : test
| ba1 : test
| baV : bv → test
| baN : test → test
| baAnd : test → test → test
| baOr : test → test → test.

Inductive kat : Type :=
| kats : sy→ 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 $t \in \mathcal{T}$ and $\alpha \in \text{At}$, we denote evaluation of tests with respect to atoms by $\alpha \leq t$.

```

Fixpoint evalT( $\alpha$ :atom) ( $t$ :test) {struct t} : bool :=
  match  $\alpha$  with
  | ba0 ⇒ false
  | ba1 ⇒ true
  | baV  $b$  ⇒ eval_atom_bv  $\alpha$   $b$ 
  | baN  $t_1$  ⇒ negb (evalT  $\alpha$   $t_1$ )
  | baAnd  $\alpha_1$   $\alpha_2$  ⇒ (evalT  $\alpha$   $\alpha_1$ ) && (evalT  $\alpha$   $\alpha_2$ )
  | baOr  $\alpha_1$   $\alpha_2$  ⇒ (evalT  $\alpha$   $\alpha_1$ ) || (evalT  $\alpha$   $\alpha_2$ )
  end.

```

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

$$\begin{aligned} \overline{b_1 b_2} \leq t &= \text{evalT}(\overline{b_1 b_2})(b_1 + \overline{b_2}) \\ &= \text{evalT}(\overline{b_1 b_2})(b_1) \parallel \text{evalT}(\overline{b_1 b_2})(\overline{b_2}) \\ &= \text{evalT}(\overline{b_1 b_2})(b_1) \parallel \text{negb}(\text{evalT}(\overline{b_1 b_2})(b_2)) \\ &= \text{true} \parallel \text{negb}(\text{true}) = \text{true} \end{aligned}$$

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* [59, 66]. A guarded string is a sequence

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

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

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

For guarded string x we define $\text{first}(x) \stackrel{\text{def}}{=} \alpha_0$ and $\text{last}(x) \stackrel{\text{def}}{=} \alpha_n$. We say that two guarded strings x and y are *compatible* if $\text{last}(x) = \text{first}(y)$. Obviously, if $n = 0$ then $\text{last}(x) = \text{first}(x)$ always hold. 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 $\text{last}(x)$, or $\text{first}(y)$. 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 $\text{last}(y) = b_1 \overline{b_2} \neq b_1 b_2 = \text{first}(x)$.

In COQ we have implemented the fusion product of two guarded strings x and y not as a partial function, but rather as a total function containing 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 (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 KAT terms e_1 and e_2 , 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 $\alpha p \in (\text{At} \cdot \Sigma)$ and $x' \in \text{GS}_{\mathcal{B}, \Sigma}$, then x' and y remain compatible. The main properties of the fusion product over compatible guarded strings are the following: if $x, y, z \in \text{GS}_{\mathcal{B}, \Sigma}$ 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 operations `first` and `last` are not affected by the fusion product of guarded strings, i.e. x and y , i.e., $\text{last}(xy) = \text{last}(y)$ and $\text{first}(xy) = \text{first}(x)$.

Languages of Guarded Strings

In 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 $\text{GS}_{\mathcal{B}, \Sigma}$. The constant languages are the empty language \emptyset and the language containing all atoms. These notion are formalised in COQ as follows:

Definition $gl := gs \rightarrow \text{Prop}$.

Inductive $gl_emp : gl := \emptyset$

Notation " \emptyset " $:= gl_emp$.

Inductive $gl_eps : gl :=$

| $in_gl_eps : \forall \alpha : \text{atom}, \alpha \in gl_eps$.

Notation " At " $:= gl_eps$.

The language of a primitive test b is the set of all atoms $\alpha \in At$ 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 $g_bv : bv \rightarrow gl :=$

| $in_g_bv : \forall (b : bv) (\alpha : \text{atom}), \alpha \leq b \rightarrow b \in g_bv\ b$.

Inductive $gl_test : gl : \text{test} \rightarrow gl :=$

| $in_gl_test : \forall (\alpha : \text{atom}) (t : \text{test}), \alpha \leq t \rightarrow \alpha \in gl_test\ t$.

Inductive $gl_at_only (a : \text{atom}) : gl :=$

| $mk_gl_atom_only : (gs_end\ a) \in gl_at_only\ a$.

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

Inductive $gl_sy : sy \rightarrow gl :=$

| $in_gl_sy : \forall (p : sy) (\alpha \beta : \text{atom}), \alpha p \beta \in gl_sy\ p$.

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 $gl_conc(L_1\ L_2 : gl) : gl :=$

| $mk_gl_conc : \forall (x\ y : gl) (T : \text{compatible}\ x\ y),$

$x \in L_1 \rightarrow y \in L_2 \rightarrow (\text{fusion_prod}\ x\ y\ T) \in (gl_conc\ L_1\ L_2)$.

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

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

The power of and the Kleene star of a language L , denoted respectively by L^n and L^* , are defined as expected.

```

Inductive gl_conc (L1 L2:gl) : gl :=
| mkgl_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
| 0 ⇒ {1}
| S m ⇒ L ⋄ (conc_gln L m)
end.

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

```

As in 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 string. Thus, given a KAT term e , the language that e denotes, $G(e)$, is recursively defined on the structure of e as follows:

$$\begin{aligned}
G(p) &\stackrel{\text{def}}{=} \{\alpha p \beta \mid \alpha, \beta \in \text{At}\}, \quad p \in \Sigma \\
G(t) &\stackrel{\text{def}}{=} \{\alpha \in \text{At} \mid \alpha \leq t\}, \quad t \in T \\
G(e_1 + e_2) &\stackrel{\text{def}}{=} G(e_1) \cup G(e_2) \\
G(e_1 e_2) &\stackrel{\text{def}}{=} G(e_1) G(e_2) \\
G(e^*) &\stackrel{\text{def}}{=} \bigcup_{n \geq 0} G(e)^n.
\end{aligned}$$

From the previous definition it is easy to conclude that $G(1) = \text{At}$ and that $G(0) = \emptyset$. If $x \in \text{GS}_{\mathcal{B}, \Sigma}$, then the language is $G(x) = \{x\}$. 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 $G(e_1) = G(e_2)$.

Example 23. Let $\mathcal{B} = \{b_1, b_2\}$ and $\Sigma = \{p, q\}$. Let $e = b_1 p + q \overline{b_2}$. The language denoted

by the KAT term e is the following:

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

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

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

$$\mathcal{D}_w(L) \stackrel{\text{def}}{=} \{x \mid wx \in L\}, \quad (4.10)$$

and are specialised to elements $w \in (\text{At} \cdot \Sigma)$ by

$$\mathcal{D}_{\alpha p}(L) \stackrel{\text{def}}{=} \{x \mid \alpha p x \in L\}. \quad (4.11)$$

In COQ we have 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 words and elements of $(\text{At} \cdot \Sigma)$.

```
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. llowin
```

```
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.
```

4.3 Partial Derivatives of KAT Terms

The notion of derivative of a KAT term was introduced by Dexter Kozen in [67] as an extension of 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 nullable KAT term. Given an atom α and a KAT term e , the function is inductively defined by

$$\begin{aligned}\varepsilon_\alpha(p) &\stackrel{\text{def}}{=} \text{false} \\ \varepsilon_\alpha(t) &\stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \alpha \leq t, \\ 0 & \text{if } \alpha \not\leq t. \end{cases} \\ \varepsilon_\alpha(e_1 + e_2) &\stackrel{\text{def}}{=} \varepsilon_\alpha(e_1) \parallel \varepsilon_\alpha(e_2) \\ \varepsilon_\alpha(e_1 e_2) &\stackrel{\text{def}}{=} \varepsilon_\alpha(e_1) \&\& \varepsilon_\alpha(e_2) \\ \varepsilon_\alpha(e^*) &\stackrel{\text{def}}{=} \text{true}\end{aligned}$$

determines if e is nullable. The function ε is extended to the set of all atoms At by

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

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

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

and, symmetrically, by

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

For KAT terms e_1 and e_2 , if it holds $\varepsilon_\alpha(e_1) = \varepsilon_\alpha(e_2)$ for all $\alpha \in \text{At}$, 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 $\varepsilon_{b_1 b_2}(\alpha)$ goes as follows:

$$\begin{aligned}\varepsilon_{b_1 b_2}(\alpha) &= \varepsilon_{b_1 b_2}(b_1 p + q \overline{b_2}) \\ &= \varepsilon_{b_1 b_2}(b_1 p) \parallel \varepsilon_{b_1 b_2}(q \overline{b_2}) \\ &= (\varepsilon_{b_1 b_2}(b_1) \&\& \varepsilon_{b_1 b_2}(p)) \parallel (\varepsilon_{b_1 b_2}(q) \&\& \varepsilon_{b_1 b_2}(\overline{b_2})) \\ &= (b_1 b_2 \leq b_1 \&\& \text{false}) \parallel (\text{false} \&\& b_1 b_2 \leq \overline{b_2}) \\ &= (\text{true} \&\& \text{false}) \parallel (\text{false} \&\& \text{false}) = \text{false}.\end{aligned}$$

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

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

Nullability is extended to sets in the following way:

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

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 $\alpha p \in (\text{At} \cdot \Sigma)$ 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

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

Example 25. Let $\mathcal{B} = \{b_1, b_2\}$, let $\Sigma = \{p, q\}$, and let $e = b_1 p + q \overline{b_2}$. The partial derivative

of e with respect to b_1b_2p is the following:

$$\begin{aligned}
\partial_{b_1b_2p}(e) &= \partial_{b_1b_2p}(b_1p + q\overline{b_2}) \\
&= \partial_{b_1b_2p}(b_1p) \cup \partial_{b_1b_2p}(q\overline{b_2}) \\
&= \partial_{b_1b_2p}(b_1)p \cup \partial_{b_1b_2p}(p) \cup \partial_{b_1b_2}(q)\overline{b_2} \\
&= \partial_{b_1b_2p}(b_1)p \cup \partial_{b_1b_2p}(p) \\
&= \partial_{b_1b_2p}(p) \\
&= \{1\}.
\end{aligned}$$

Partial derivatives of KAT terms can be inductively extended to sequences $w \in (\text{At} \cdot \Sigma)^*$ 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 sequence. The set of all partial derivatives is the set

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

Example 26. Let $\mathcal{B} = \{b_1, b_2\}$, let $\Sigma = \{p, q\}$, and let $e = b_1p(b_1 + b_2)q$. The partial derivative of e with respect to the sequence $b_1b_2p\overline{b_1}b_2q$ is the following:

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

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

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

```

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

Definition ewp_set(s:set kat) (a:atom) :=
  fold (fun x ⇒ orb (ewp 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 ewp 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 : $\forall e \alpha p, G(\text{pdrv } e \alpha p) \sim \text{LQ } (G(e)) \alpha p$.

Theorem wpdrv_correct : $\forall e w, G(\text{wpdrv } e w) \sim \text{LQw } (G(e)) w$.

Finiteness of Partial Derivatives

Kozen showed [67] 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* [80]. 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 [67].

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

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

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

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

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

Lemma 12. 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

$$\forall e' \quad e' \in \pi(e) \rightarrow \forall e'' \quad e'' \in \pi(e') \rightarrow e'' \in \pi(e).$$

Lemma 13. 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 $\text{KD}(e) \stackrel{\text{def}}{=} \{e\} \cup \pi(e)$, with e being a KAT term. It is easy to see that $|\text{KD}(e)| \leq |e|_{\Sigma} + 1$, since $|\pi(e)|_{\Sigma} \leq |e|_{\Sigma}$. We will now show that $\text{KD}(e)$ established an upper bound on the number of partial derivatives of e . For that, we prove that $\text{KD}(e)$ contains all the partial derivatives of e . First prove that the partial derivative $\partial_{\alpha p}(e)$ is a subset of $\pi(e)$. 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 sequence $w \in (\text{At} \cdot \Sigma)^*$ that all the derivatives of e are members of $\text{KD}(e)$.

Lemma 14. *Let e be a KAT term, and let $\alpha p \in (\text{At} \cdot \Sigma)$. Hence, if the KAT term e' is a member of $\pi(e)$, then $\partial_{\alpha p}(e') \subseteq \pi(e)$.*

Theorem 6. *Let e be a KAT term, and let $w \in (\text{At} \cdot \Sigma)^*$. Thus, $\partial_w(e) \subseteq \text{KD}(e)$.*

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

```

Fixpoint PI (e:kat) : set kat :=
  match e with
  | katb b  $\Rightarrow$   $\emptyset$ 
  | kats _  $\Rightarrow$  {katb bal}
  | katu x y  $\Rightarrow$  (PI x)  $\cup$  (PI y)
  | katc x y  $\Rightarrow$  (PI x)  $\odot$  y  $\cup$  (PI y)
  | katst x  $\Rightarrow$  (PI x)  $\odot$  (katst x)
  end.
Notation " $\pi(x)$ " := (PI x).

```

```

Definition KD(r:kat) := {r}  $\cup$  (XI r).

```

```

Fixpoint sylen (e:kat) : nat :=
  match e with
  | katb _  $\Rightarrow$  0
  | kats _  $\Rightarrow$  1
  | katu x y  $\Rightarrow$  sylen x + sylen y
  | katc x y  $\Rightarrow$  sylen x + sylen y
  | katst x  $\Rightarrow$  sylen x
  end.
Notation " $|e|_{\Sigma}$ " := (sylen e).

```

```

Theorem KD_upper_bound :  $\forall e$ , cardinal (KD e)  $\leq$  (sylen e) + 1.

```

```

Theorem all_wpdrv_in_KD :  $\forall w x r$ ,  $x \in (\text{wpdrv } e w) \rightarrow x \in \text{KD}(r)$ .

```

4.4 A Procedure for KAT Terms Equivalence

In this section we introduce a procedure for deciding KAT term 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 development is 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 expression (in)equivalence. Given a KAT term e we know that

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

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

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

which is tantamount at checking if

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

and

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

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:

$$e_1 \sim e_2 \leftrightarrow \forall \alpha \in \text{At}, \forall w \in (\text{At} \cdot \Sigma)^*, \varepsilon_\alpha(\partial_w(e_1)) = \varepsilon_\alpha(\partial_w(e_2)) \quad (4.12)$$

and

$$(\exists w \exists \alpha, \varepsilon_\alpha(\partial_w(e_1)) \neq \varepsilon_\alpha(\partial_w(e_2))) \leftrightarrow e_1 \not\sim e_2. \quad (4.13)$$

The terminating decision procedure EQUIVKAT, presented in Algorithm 2, describes the computational interpretation of the equivalences (4.12) and (4.13). This procedure corresponds to the iterated process of deciding the equivalence of their partial derivatives.

Algorithm 2 The procedure EQUIVKAT.

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

```

1: procedure EQUIVKAT( $S, H$ )
2:   while  $s \neq \emptyset$  do
3:      $(\Gamma, \Delta) \leftarrow POP(s)$ 
4:     for  $\alpha \in \text{At}$  do
5:       if  $\varepsilon_\alpha(\Gamma) \neq \varepsilon_\alpha(\Delta)$  then
6:         return false
7:       end if
8:     end for
9:      $H \leftarrow H \cup \{(\Gamma, \Delta)\}$ 
10:    for  $\alpha p \in (\text{At} \cdot \Sigma)$  do
11:       $(\Lambda, \Theta) \leftarrow \partial_{\alpha p}(\Gamma, \Delta)$ 
12:      if  $(\Lambda, \Theta) \notin H$  then
13:         $S \leftarrow S \cup \{(\Lambda, \Theta)\}$ 
14:      end if
15:    end for
16:  end while
17: return true
18: end procedure

```

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 requiring 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-nullable; 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.

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 to the case of inequivalence.

Example 28. Let $\mathcal{B} = \{b\}$ and let $\Sigma = \{p, q\}$. Suppose we want to prove that $e_1 = (bp)^*q$

and $e_2 = b(pq)^*$ are not equivalent. Considering $s_0 = (\{(ab)^*a\}, \{a(ba)^*\})$, it is enough to show that

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

The computation of EQUIVP for these particular α and β involves the construction of the new derivatives $s_1 = (\{1, b(ab)^*a\}, \{(ba)^*\})$ 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 29. Suppose we want to check if $\alpha = b^*a$ and $\beta = b^*ba$ are not equivalent. Considering $s_0 = (\{b^*a\}, \{b^*ba\})$, to prove so it is enough to check if

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

In this case, the computation of EQUIVP creates the new derivatives, $s_1 = (\{1\}, \emptyset)$ and $s_2 = (\{b^*a\}, \{a, b^*ba\})$, 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	$\{s_1, s_2\}$	$\{s_0\}$	$\varepsilon(s_1) = \text{false}$

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 sequence from $(\text{At} \cdot \Sigma)^*$, and cw is a parameter holding a proof that $dp = (\partial_w(\Gamma), \partial_w(\Delta))$.

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

The derivation and nullability functions were also changed to accomodate the new type `KDrv`. As an example, we present the extended derivation functions below. The type `AtSy` used in the definition of `KDrv_pdrv_set` represents values $\alpha p \in (\text{At} \cdot \Sigma)$.

```
Definition KDrv_pdrv (x:KDrv e1 e2) (α:atom) (p:sy) : KDrv e1 e2.
```

Proof.

```
  refine(match x with Build_ReW k w p ⇒
    Build_KDrv e1 e2 (pdrv k a s) (w++[(α,p)]) _
  end).
```

(* ... *)

Defined.

```
Definition KDrv_wpdrv (w:list AtSy) : ReW e1 e2.
```

Proof.

```
  refine(Build_KDrv e1 e2 (wpdrv ({e1},{e2}) w) w _).
```

(* ... *)

Defined.

```
Definition KDrv_pdrv_set (s:KDrv e1 e2) (sig:set AtSy) : set (KDrv e1 e2) :=
  fold (fun x:AtSy ⇒ add (KDrv_pdrv s (fst x) (snd x))) sig ∅.
```

Implementation of EQUIVKAT in Coq

Like in the case of the definitions of the derivation operations on `KDrv` terms, the extended nullability functions are were enriched with atoms.

```
Definition ewp_p(x:set kat * set kat) (a:atom) := eqb (ewp_set (fst x) a)
  (ewp_set (snd x) a).
```

Definition `ewp_at_set` (x :`set` `kat` * `set` `kat`) (ats :`set` `atom`) := `fold` (`fun` p
 \Rightarrow `andb` (`ewp_p` x p)) ats `true`.

Definition `ewpKDrv` (x :`KDrv` e_1 e_2) (a :`set` `atom`) := `ewp_at_set` x a .

Definition `newKDrvSet` (x :`KDrv` e_1 e_2) (h :`set` (`KDrv` e_1 e_2)) (sig :`set` `AtSy`) : `set`
`(KDrv` e_1 e_2) :=
`filter` (`fun` $x \Rightarrow$ `negb` ($x \in h$)) (`KDrv_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 loop of the algorithm. This function makes $2^{|B|}$ 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 14 of Algorithm 2.

Inductive `step_case` (e_1 e_2 :`kat`) : `Type` :=
`|``proceed` : `step_case` e_1 e_2
`|``termtrue` : `set` (`KDrv` e_1 e_2) \rightarrow `step_case` e_1 e_2
`|``termfalse` : `KDrv` e_1 e_2 \rightarrow `step_case` e_1 e_2 .

Definition `step` (h s :`set` (`KDrv` e_1 e_2)) (sig :`set` `sy`) (ats :`set` `atom`) :
`((set` (`KDrv` e_1 e_2) * `set` (`KDrv` e_1 e_2)) * `step_case` e_1 e_2) :=
`match` `choose` s `with`
`|``None` \Rightarrow (`(h, s), termtrue` e_2 e_1 h)
`|``Some` (d_{e_1}, d_{e_2}) \Rightarrow
`if` `ewpKDrv` e_1 e_2 (d_{e_1}, d_{e_2}) ats `then`
`let` h' := `add` (d_{e_1}, d_{e_2}) h `in`
`let` rsd' := `in`
`let` s' := `newKDrvSet` e_1 e_2 (d_{e_1}, d_{e_2}) H' sig ats `in`
`(h', s' \cup ($s \setminus \{(d_{e_1}, d_{e_2})\}$), proceed e_1 e_2)`
`else`
`((h, s), termfalse` e_1 e_2 (d_{e_1}, d_{e_2}))
`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, page 54, 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 expression (in-)equivalence.

Inductive `term_cases` e_1 e_2 : `Type` :=
`|``Equiv` : `set` (`KDrv` e_1 e_2) \rightarrow `term_cases` e_1 e_2

```
|NotEquiv : KDrv e1 e2 → term_cases e1 e2.
```

```
Inductive DP (h s:set (KDrv e1 e2)) (ats:set atom) : Prop :=
|is_dp : h ∩ s == ∅ → (∀ x:atom, x ∈ ats) → ewpKDrv_set e1 e2 h ats =
  true → DP h s ats.
```

```
Function iterate(e1 e2:kat) (h s:set (KDrv e1 e2)) (sig:set A) (d:DP e1 e2 h s)
{wf (LLim e1 e2) h}: term_cases e1 e2 :=
  let ((h', s'), next) := step h s in
  match next with
  |termfalse x ⇒ NotEquiv e1 e2 x
  |termtrue h ⇒ Equiv e1 e2 h
  |progress ⇒ iterate e1 e2 h' s' sig (DP_upd e1 e2 h s sig 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 (KDrv e1 e2)) (sig:set sy) (d:DP e1
  e2 h s) :=
  let h' := iterate e1 e2 h s sig D in
  match h' with
  | Ok _ ⇒ true
  | NotOk _ ⇒ false
end.
```

```
Definition mkDP_1st : DP e1 e2 ∅ {KDrv_1st e1 e2}.
```

```
Definition equivkat(e1 e2:kat) := equivkat_aux e1 e2 ∅ {KDrv_1st e1 e2}
  sigmaP (mkDP_1st e1 e2).
```

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, H' denote sets of pairs of sets of $\text{KDrv } e_1 \ e_2$ terms; finally, \mathcal{B} and Σ denote the set of primitive tests, and the set of primitive programs, respectively.

For the case of the correctness of `equivkat`, we divide the proof in two cases. In the first case, we prove $e_1 \not\sim e_2$ whenever `equivkat` returns `false`. In the second case, we define an invariant on `kiterate` which gives the properties that allows us to conclude $e_1 \sim e_2$.

For the first case, we show that if `equivkat` finds a term d of type $\text{KDrv } e_1 \ e_2$ such that $\varepsilon_\alpha(d)$ is false for some atom $\alpha \in \text{At}$, then there exists a guarded string x that is not simultaneously a member of $G(e_1)$ and of $G(e_2)$. Naturally, this conclusion implies $e_1 \not\sim e_2$.

Proposition 1. *Let D be a term of type $\text{KDP } e_1 \ e_2 \ S \ H$. If*

$$\text{kiterate } e_1 \ e_2 \ S \ H \ \mathcal{B} \ \Sigma \ D = \text{NotOk } e_1 \ e_2 \ (d_{e_1}, d_{e_2})$$

holds, then $\varepsilon_\alpha(d_{e_1}) \neq \varepsilon_\alpha(d_{e_2})$ also holds for some $\alpha \in \text{At}$.

Proposition 2. *Let D be a term of type $\text{KDP } e_1 \ e_2 \ S \ H$. Hence, it holds that*

$$\text{kiterate } e_1 \ e_2 \ S \ H \ \mathcal{B} \ \Sigma \ D = \text{NotOk } e_1 \ e_2 \ (d_{e_1}, d_{e_2}) \rightarrow e_1 \not\sim e_2.$$

Lemma 15. *If `equivkat` returns with the value of `false` then e_1 and e_2 are in-equivalent KAT terms, i.e.,*

$$\text{equivkat } e_1 \ e_2 = \text{false} \rightarrow e_1 \not\sim e_2.$$

For the second case, we proceed by defining an invariant over `kiterate` that allow us to conclude KAT term equivalence by proving that all the partial derivatives are computed and also they are equi-nullable. This invariant is given by the definition of `invK_final` below.

Definition `invK`($H \ S:\text{set } (\text{KDrv } e_1 \ e_2)$) ($\mathcal{B}:\text{set } \text{atom}$) ($\Sigma:\text{set } \text{sy}$) :=
 $\forall d, d \in H \rightarrow \forall p, p \in \Sigma \rightarrow \forall \alpha, \alpha \in \mathcal{B} \rightarrow$
 $(\text{KDrv_pdrv } e_1 \ e_2 \ d \ \alpha \ p) \in (H \cup S).$

Definition `invK_final`($H \ S:\text{set } (\text{KDrv } e_1 \ e_2)$) ($\mathcal{B}:\text{set } \text{atom}$) ($\Sigma:\text{set } \text{sy}$) :=
 $(\text{KDrv_1st } e_1 \ e_2) \in (H \cup S) \wedge$
 $(\forall d, d \in (H \cup S) \rightarrow \text{ewp_KDrv } e_1 \ e_2 \ d \ \mathcal{B} = \text{true}) \wedge$
`invK` $H \ S \ \mathcal{B} \ \Sigma.$

By the recursive behaviour of `kiterate`, and by the definition of `kstep` we prove that `invK_final` leads to KAT term equivalence. First we prove that a successful computation of `kiterate` yields a set containing all the partial derivatives of e_1 and e_2 .

Proposition 3. *If $\text{invK } H \ S$ holds, and if*

$$\text{kstep } e_1 \ e_2 \ H \ S \ \Sigma = ((H', S'), \text{proceed } e_1 \ e_2)$$

also holds, then $\text{invK } H' \ S'$.

Proposition 4. *Let D be a term of type $\text{KDP } e_1 \ e_2 \ S \ H$. If $\text{invK } H \ S$ holds, and if*

$$\text{kiterate } e_1 \ e_2 \ H \ S \ \Sigma \ D = \text{ok } e_1 \ e_2 \ H',$$

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

Proposition 5. *Let D be a value of type $\text{KDP } e_1 \ e_2 \ H \ S$. If it the case that $\text{kiterate } e_1 \ e_2 \ H \ S \ \mathcal{B} \ \Sigma \ D = \text{ok } e_1 \ e_2 \ H'$ hold, then $H \subseteq H'$.*

Corollary 2. *Let D be a value of type $\text{KDP } e_1 \ e_2 \ H \ S$. If*

$$\text{kiterate } e_1 \ e_2 \ H \ S \ \mathcal{B} \ \Sigma \ D = \text{ok } e_1 \ e_2 \ H',$$

and if choose $S = \text{Some } (d_{e_1}, d_{e_2})$ holds, then it is the case that $\{(d_{e_1}, d_{e_2})\} \cup H \subseteq H'$.

Proposition 6. *Let D be a value of type $\text{KDP } e_1 \ e_2 \ \emptyset \ \{(\{e_1\}, \{e_2\})\}$. Hence,*

$$\text{kiterate } e_1 \ e_2 \ \emptyset \ \{(\{e_1\}, \{e_2\})\} \ \mathcal{B} \ \Sigma \ D = \text{ok } e_1 \ e_2 \ H' \rightarrow (\{e_1\}, \{e_2\}) \in H'.$$

We proceed by showing that all the elements of the H' in the value $\text{ok } e_1 \ e_2 \ H'$ 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 7. *Let D be a value of type $\text{KDP } e_1 \ e_2 \ H \ S$. If $\text{kiterate } e_1 \ e_2 \ H \ S \ \mathcal{B} \ \Sigma \ D = \text{ok } e_1 \ e_2 \ H'$, then $\forall \alpha \in \text{At}, \forall \gamma \in H', \varepsilon_\alpha(\gamma) = \text{true}$ holds.*

Using lemmas 6 and 8 we can establish the intermediate result that will take us to prove the correctness of equivP with respect to language equivalence.

Proposition 8. *Let D be a value of type $\text{KDP } e_1 \ e_2 \ H \ S$. If $\text{kiterate } e_1 \ e_2 \ H \ S \ \mathcal{B} \ \Sigma \ D = \text{ok } e_1 \ e_2 \ H'$, then $\text{invK_final } e_1 \ e_2 \ H' \ \emptyset$ holds.*

The last intermediate logical condition that we need to establish is that invK_final implies KAT terms equivalence, considering that it is instantiated with the correct parameters. The following lemma gives us exactly that.

Lemma 16. *If $\text{invK_final } e_1 \ e_2 \ H' \ \emptyset \ \mathcal{B} \ \Sigma$ holds, then it is true that $e_1 \sim e_2$.*

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

Theorem 7. *Let e_1 and e_2 be two KAT terms. Hence, $\text{equivkat } e_1 \ e_2 = \text{true}$ iff $e_1 \sim e_2$.*

Performance and Usability of EQUIVKAT

As pointed out earlier, the performance of the implementation of EQUIVKAT is not expected to show the performances exhibited by EQUIVP. The algorithm contains two exponential computations, one for checking if a derivative is equi-nullable, and another to compute new derivatives. In the setting of partial derivatives, and to the best of our knowledge, we are unable to avoid computing with the set of all atoms. There exists an alternative [?] to this problem, but it relies on matrices of KAT terms for transforming KAT terms into KA terms. Matrices are not addressed and where purposely set aside from our development.

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 correct by construction program that can be used outside COQ's environment and exhibit better performances.

4.5 Application to Program Verification

As we have said 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 motives 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. Having a correct decision procedures is an extra tool for such verification tasks.

We begin this section by introducing some examples borrowed from [64] that show how KAT can be useful to prove the equivalence between certain classes of simple imperative programs. We will consider the imperative language introduced in Section 2.2. Afterwards, we show how KAT and Hoare logic are related, namely, with respect to the weaker PHL and how reasoning in PHL can be reduced to 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. If e_1 and e_2 are terms encoding the **IMP** programs C_1 and C_2 , then we can encode sequence, conditional instructions and while loops in KAT as follows.

$$\begin{aligned} C_1; C_2 &\stackrel{\text{def}}{=} e_1 e_2, \\ \text{if } b \text{ then } C_1 \text{ else } C_2 &\stackrel{\text{def}}{=} (b e_1 + \bar{b} e_2), \\ \text{while } b \text{ do } C_1 \text{ done} &\stackrel{\text{def}}{=} (b e_1)^* \bar{b}. \end{aligned}$$

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 for certain programs using these constructions. A **do**-loop is defined in KAT as the term $p(bp)^* \bar{b}$, that is, it always begin with a computation of the body of the **do**-loop. We also consider $\text{skip} \stackrel{\text{def}}{=} 1$.

Example 30. Let $\mathcal{B} = \{b\}$ and $\Sigma = \{p\}$ be the set of primitive tests and set of primitive programs, respectively, and let $P1$ and $P2$ be the following two programs:

$P1 : \text{do } \{$ if b then: $C;$ else: skip; $\} \text{ while } b$	$P2 : \text{while } b \text{ do } :$ $C;$ end
---	---

Considering that the Boolean guard b is denoted by the same primitive symbol in \mathcal{B} , and assuming that the **IMP** program C is denoted by the primitive program p , then the programs $P1$ and $P2$ are encoded as KAT terms by

$$e_1 = (bp + \bar{b}1)(b(bp + \bar{b}1))^* \bar{b}$$

and

$$e_2 = (bp)^* \bar{b},$$

respectively. The procedure `equivkat` decides the equivalence $e_1 \sim e_2$ in 0.028 seconds.

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 31. Let $\mathcal{B} = \{b\}$ and $\Sigma = \{p, q, r\}$ be the set of primitive tests and set of primitive programs, respectively, and let $P3$ and $P4$ be the following two programs:

$P3 : \text{ if } b \text{ then:}$ $p; q;$ else: $p; r;$	$P4 : p;$ if b then: $q;$ else: $r;$
---	--

These programs are encoded in KAT as

$$e_3 = (bpq + \bar{b}pr)$$

and

$$e_4 = p(bq + \bar{b}r),$$

respectively. Program $P4$ is expected to be equivalent to program $P3$ if the value of b is preserved by the program p . Since we do not know that, we must introduce extra information on both programs, but preserving their meaning. We introduce a new test c that remembers the effect of p in c and replace the test b in $P4$ with 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 $bc + \bar{b}\bar{c}$. The resulting modified programs are the following:

$P3 : bc + \bar{b}\bar{c};$ if b then: $p; q;$ else: $p; r;$	$P4 : bc + \bar{b}\bar{c};$ $p;$ if c then: $q;$ else: $r;$
--	--

These new programs are encoded as the KAT terms

$$e'_3 = (bc + \bar{b}\bar{c})(bpq + \bar{b}pr)$$

and

$$e'_4 = (bc + \bar{b}\bar{c})p(cq + \bar{c}r),$$

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

$$e''_4 = s(bc + \bar{b}\bar{c})(cpq + \bar{c}pr).$$

The procedure now proves the equivalence $e'_3 \sim e''_4$ in 0.06 seconds.

The final example concerns with proving that programs with **While** loops have an equivalent program with just a single **While** loop.

Example 32. Let $\mathcal{B} = \{b, c\}$ and $\Sigma = \{p, q\}$ be the set of primitive tests and set of primitive programs, respectively, and let $P5$ and $P6$ be the following two programs:

$P5$: while b do: p ; while c do: q ; end end q ;	$P6$: if b then: p ; while $b+c$ do: if c then: q ; else: p ; end
--	---

Programs $P5$ and $P6$ are encoded in KAT as

$$e_5 = (bp((cq)^*\bar{c}))^*\bar{b}$$

and

$$e_6 = bp((b+c)(cq + \bar{c}p))^*\overline{(b+c)} + \bar{b},$$

respectively. The procedure **equivkat** decides the equivalence $e_5 \sim e_6$ in 0.053 seconds.

Hoare Logic and KAT

Hoare logic was already introduced in Section 2.2. There we have seen that Hoare logic uses triples of the form $\{b\}p\{c\}$, where b is the pre-condition and c is the post-condition of the program p . The meaning of these triples, called Hoare triples or PCAs, is the following: *if b holds when p starts executing then c will necessarily hold when p 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 $\{b\}p\{c\}$ are expressed by the equation $bp = bpc$, or equivalently by $bp\bar{c} = 0$, or by $bp \leq c$. The inference rules of PHL are encoded as follows: the program sequence rule is encoded as

$$bp = bcp \wedge cq = cqd \rightarrow bpq = bpqd, \quad (4.14)$$

the conditional rule as

$$bcp = bcpd \wedge \bar{b}cq = \bar{b}cqd \rightarrow c(bp + \bar{b}q) = c(bp + \bar{b}q)d, \quad (4.15)$$

and, finally, the while loop rule as

$$bpc = bcpc \rightarrow c(bp)^*\bar{b} = c(bp)^*\bar{b}bc. \quad (4.16)$$

The equations (4.14–4.16) thus correspond to the deductive rules (??–??) – the assignment rule (??) is not accounted since PHL is propositional – and were mechanically checked during our development. Our decision procedure may be of few 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 [64]. There exists however a fragment of Horn formulas of KAT that can be reduced to equalities where our procedure can be applied. These are of the family of formulas

$$b_1p_1\bar{b}'_1 = 0 \wedge \dots \wedge b_n p_n \bar{b}'_n = 0 \wedge \dots \wedge c_1 \leq c'_1 \wedge \dots \wedge c_m \leq c'_m \rightarrow bp\bar{b}' = 0. \quad (4.17)$$

It has been showed in [46, 47] that for all KAT terms $r_1, \dots, r_n, e_1, e_2$ over a set of primitive programs $\Sigma = \{p_1, \dots, p_m\}$, the equations of the form

$$r_1 = 0 \wedge \dots \wedge r_n \rightarrow e_1 = e_2$$

is a theorem of KAT if and only if

$$e_1 + uru = e_2 + uru, \quad (4.18)$$

where $u \stackrel{\text{def}}{=} (p_1 + \dots + p_m)^*$ and $r \stackrel{\text{def}}{=} r_1 + \dots + r_n$. At this point our decision procedure can be used to decide (4.18). 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. The corresponding encoding of such programs in KAT is the following:

$$\begin{aligned} p; \{b\}q &\stackrel{\text{def}}{=} e_1 b e_2 \\ \text{if } b \text{ then } p \text{ else } q &\stackrel{\text{def}}{=} (b e_1 + \bar{b} e_2) \\ \text{while } b \text{ do } \{i\} p &\stackrel{\text{def}}{=} (bip)^*\bar{b} \end{aligned}$$

Given a program P we can obtain its annotated counterpart using a VCGEN algorithm such as the one presented by Frade and Pinto [42] or, at the level of KAT only, using the algorithm proposed by Almeida *et. al.* [8].

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 assertion to primitive tests.

Example 33. Consider the program **Sum3** given below. This program calculates the triple of a number.

Sum3	Encoding
$\{true\}$	t_0
$y := x$	p_1
$\{x + x + y = 3x\}$	t_1
$y := x + x + y ;$	p_2
$\{y = 3x\}$	t_2

To prove the partial correctness of **Sum3** we must infer the corresponding set of hypotheses of the form $bp\bar{b}' = 0$ so that we can produce the KAT equivalence to which we can apply our decision procedure. Such set of hypothesis is the following:

$$\Gamma = \{t_0 \leq t_3, t_3 p_1 \bar{t}_1 = 0, t_1 p_2 \bar{t}_2 = 0\}.$$

The triple $\{true\}\{\text{Sum3}\}\{y = 3x\}$ is encoded as a KAT equation as follows:

$$t_0 t_3 p_1 t_1 p_2 \bar{t}_2 = 0,$$

from where we can define $u = (p_1 + p_2)^*$ and $r = t_0 \bar{t}_3 + t_3 p_1 \bar{t}_1 + t_1 p_2 \bar{t}_2$. We can now apply the decision procedure to the term

$$t_0 t_3 p_1 t_1 p_2 \bar{t}_2 + uru = 0 + uru$$

and obtain a positive answer in 0.912 seconds.

Example 34. Lets consider the program implementing the factorial of a number, presented in the table presented below, together with its pre- and post-conditions.

Fact	Encoding
$\{true\}$	t_0
$y := 1$	p_1
$\{y = 0!\}$	t_1
$z := 0 ;$	p_2
$\{y = z!\}$	t_2
$while \neg(z=x) do$	t_3
$\{$	
$\{y = z!\}$	t_2
$z := z + 1 ;$	p_3
$\{y \times z = z!\}$	t_4
$y := y * z ;$	p_4
$\}$	
$\{y = x!\}$	t_5

The Hoare triple $\{true\}\mathbf{Fact}\{y = x!\}$ is encoded as the equality

$$t_0 p_1 t_1 p_2 t_2 (t_3 t_2 p_3 t_4 p_4)^* \overline{t_3 t_5} = 0. \quad (4.19)$$

To prove (4.19) we need to obtain a set of hypotesis, that can be obtained in a backward fashion [8] using a weakest pre-condition generator. The set of hypothesis for (4.19) is the following:

$$\Gamma = \{t_0 p_1 \overline{t_1} = 0, t_1 p_2 \overline{t_2} = 0, t_3 t_2 p_3 \overline{t_4} = 0, t_4 p_2 \overline{t_2} = 0, t_2 \overline{t_3 t_5} = 0\}.$$

With Γ and $\Sigma = \{p_1, p_2, p_3, p_4\}$, we know that

- $u = (p_1 + p_2 + p_3 + p_4)^*$ and;
- $r = t_0 p_1 \overline{t_1} + t_1 p_2 \overline{t_2} + t_3 t_2 p_3 \overline{t_4} + t_4 p_2 \overline{t_2} + t_2 \overline{t_3 t_5}$.

The equation

$$t_0 p_1 t_1 p_2 t_2 (t_3 t_2 p_3 t_4 p_4)^* \overline{t_3 t_5} + uru = 0 + uru$$

is now provable by **equivkat**, which shows that **Fact** is correct. The procedure answered positively in 22 seconds.

4.6 Related Work

Although KAT can be applied to several verification tasks, there exist few tool support for it. Kozen and Aboul-Hosn [2] developed the KAT-ML proof assistant. KAT-ML allows one to

reason about KAT terms, and 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.* [9, 8] 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.

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 as 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. al.* [9, 8]. 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 is 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) [49, 52]. This algebraic system is based in *modal Kleene algebra* (MKA) [34, 79, 33], 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 based on extensions of the results presented in Chapter 3 and Chapter 4, we have decided before that, the best option to go into concurrency and parallelism was to understand and mechanise one of targets of CKA, namely, Jones' *rely-guarantee* (RG) [57]. 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 [28]. We note that RG was already addressed in terms of its encoding within a proof-assistant through the work of Nieto [82], who mechanised a parametric version of RG, and whose proofs follow the ones previously introduced by Xu *et. al.* [105].

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 for the later system.

It is also important to stress that RG is the base for more recent formal system, namely those that extend RG with Reynolds' *separation logic* [93]. Recent formal systems resulting from this synergy between RG and separation logic are RGSEP by Vafeiadis *et. al.* [101], *local rely-guarantee reasoning* by Feng *et. al.* [?], and also *deny-guarantee* by Dodds *et. al.* [37]. 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.

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 [85]. 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:

$$\frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\} \quad \begin{array}{l} C_1 \text{ does not interfere with } C_2 \\ C_2 \text{ does not interfere with } C_1 \end{array}}{\{P_1 \wedge P_2\} \text{par } C_1 \text{ with } C_2 \text{ end} \{Q_1 \wedge Q_2\}}$$

Based on the previous system, Jones introduced RG in his PhD thesis [58], 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 models steps of execution of the environment; a guarantee condition describes steps of execution of the program. Therefore, in the context of parallel program specification and design,

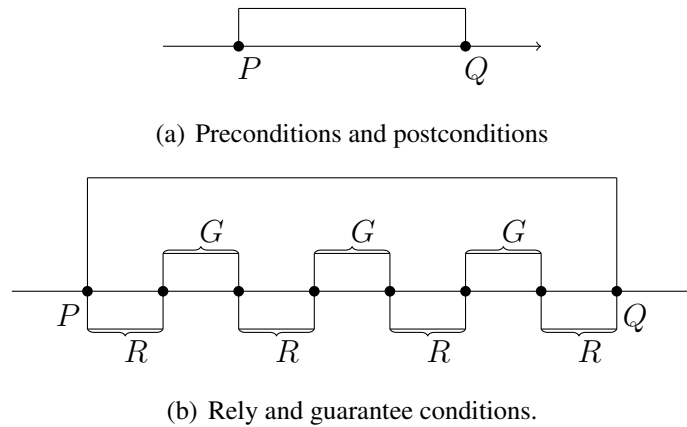
the rely conditions describes the level of interference that the program is able to tolerate from the environment, while the guarantee conditions describes the level of interference that the program imposes on the environment. From the specification point-of-view, the rely condition can be seen as a way of requiring the developer to make the necessary assumptions about the environment in which the program is going to execute. From the user point-of-view, it is the responsibility of the user itself to ensure that the environment complies with the previous assumptions.

As a final remark, ^{note} **not** 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 strengthened, 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 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 [28].

Figure 5.1 Rely and guarantee vs. preconditions and postconditions.



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 defined them by the following grammars:

$$\text{AExp} \ni e, e_1, e_2 ::= x \mid n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid [] \mid n :: l \mid \text{hd } l \mid \text{tl } l,$$

$$\text{BExp} \ni b, b_1, b_2 ::= \text{true} \mid \text{false} \mid \neg b \mid b_1 \wedge b_2 \mid e_1 = e_2 \mid e_1 < e_2 \mid \text{is_cons } l,$$

where x is a variable identifier and l is a list of natural numbers. The operators `hd` and `tl` correspond to the head and the tail of a given list l , respectively. The Boolean operator `is_cons` has the goal of checking if the input list l is empty or not. The language of IMPp programs is inductively defined by

$$\begin{aligned} \text{IMPp} \ni C, C_1, C_2 ::= & \text{skip} \\ & \mid x ::= e \\ & \mid \text{atomic } c(C') \\ & \mid C_1; C_2 \\ & \mid \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi} \\ & \mid \text{while } b \text{ do } C \text{ done} \\ & \mid \text{par } C_1 \text{ with } C_2 \text{ end,} \end{aligned}$$

where x is a variable, $e \in \text{AExp}$, and $b \in \text{BExpr}$. 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* [89]),

`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`

```

| AHead : aexp → aexp
| ATail : aexp → aexp
| ACons : aexp → aexp → aexp
| ANil : aexp.

Inductive bexp : Type :=
| BTrue : bexp
| BFalse : bexp
| BEq : aexp → aexp → bexp
| BLt : aexp → aexp → bexp
| BNot : bexp → bexp
| BAnd : bexp → bexp → bexp
| BIsCons : aexp → 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* [90]. The semantic 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

$$\xRightarrow{c} : \langle \text{IMPP}, \Sigma \rangle \rightarrow \langle \text{IMPP}, \Sigma \rangle \quad (5.1)$$

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:

```
Inductive val : Type :=
| VNat : nat → val
| VList : list nat → val.
```

```
Definition st := id → val.
```

```
Definition empty_st : st := fun _ => VNat 0.
```

Moreover, we defined the update of 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 e else s x'.
```

Before giving the structure of the relation \xRightarrow{c} we describe the interpretation of arithmetic and Boolean expressions. First we define some auxiliary functions that evaluate terms of type **AExp** into natural numbers or lists of natural numbers, depending on the result of the pattern matching over the structure of those terms.

```
Definition asnat (e : val) : nat :=
  match v with
  | VNat n => n
  | VList _ => 0
  end.
```

```
Definition aslist (e : val) : list nat :=
  match v with
  | VNat n => []
  | VList xs => xs
  end.
```

```
Definition tail (l : list nat) :=
  match l with
  | x::xs => xs
  | [] => []
  end.
```

```
Definition head (l : list nat) :=
  match l with
  | x::xs => x
  | [] => 0
  end.
```

With the help of the previous definitions, 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 ⇒ VNat n
  | AId x ⇒ s x
  | APlus e1 e2 ⇒ VNat ((asnat (aeval s e1)) + (asnat (aeval s e2)))
  | AMinus e1 e2 ⇒ VNat ((asnat (aeval s e1)) - (asnat (aeval s e2)))
  | AMult e1 e2 ⇒ VNat ((asnat (aeval s e1)) * (asnat (aeval s e2)))
  | ATail l ⇒ VList (tail (aslist (aeval s l)))
  | AHead l ⇒ VNat (head (aslist (aeval s l)))
  | ACons e1 l ⇒ VList (asnat (aeval s e1) :: aslist (aeval s l))
  | ANil ⇒ VList []
  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 ⇒
    if beq_nat (asnat (aeval s e1)) (asnat (aeval s e2)) then true else false
  | BLt e1 e2 ⇒
    if blt_nat (asnat (aeval s e1)) (asnat (aeval s e2)) then true else false
  | BNot b1 ⇒ negb (beval s b1)
  | BAnd b1 b2 ⇒ andb (beval s b1) (beval s b2)
  | BIsCons l ⇒
    match aslist (aeval s l) with
    | _::_ ⇒ true
    | [] ⇒ false
    end
  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 \Longrightarrow^c is inductively as follows:

$$\begin{array}{c}
\frac{}{\langle x ::= e, s \rangle \Longrightarrow^c \langle \text{skip}, s[e/x] \rangle} \text{(ASSGN)} \\
\\
\frac{\langle C, s \rangle \Longrightarrow^* \langle \text{skip}, s' \rangle}{\langle \text{atomic}(C), s \rangle \Longrightarrow^c \langle \text{skip}, s' \rangle} \text{(ATOMIC)} \\
\\
\frac{}{\langle \text{skip}; C, s \rangle \Longrightarrow^c \langle C, s \rangle} \text{(SEQ-1)} \\
\\
\frac{\langle C_1, s \rangle \Longrightarrow^c \langle C'_1, s' \rangle}{\langle C_1; C_2, s \rangle \Longrightarrow^c \langle C'_1; C_2, s' \rangle} \text{(SEQ-2)} \\
\\
\frac{\llbracket b \rrbracket_{\mathbb{B}}(s) = \text{true}}{\langle \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi}, s \rangle \Longrightarrow^c \langle C_1, s \rangle} \text{(IF-TRUE)} \\
\\
\frac{\llbracket b \rrbracket_{\mathbb{B}}(s) = \text{false}}{\langle \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi}, s \rangle \Longrightarrow^c \langle C_2, s \rangle} \text{(IF-FALSE)} \\
\\
\frac{}{\langle \text{while } b \text{ do } C \text{ done}, s \rangle \Longrightarrow^c \langle \text{if } b \text{ then } C; (\text{while } b \text{ do } C \text{ done}) \text{ else skip fi}, s \rangle} \text{(WHILE)} \\
\\
\frac{\langle C_1, s \rangle \Longrightarrow^c \langle C'_1, s' \rangle}{\langle \text{par } C_1 \text{ with } C_2 \text{ end}, s \rangle \Longrightarrow^c \langle \text{par } C'_1 \text{ with } C_2 \text{ end}, s' \rangle} \text{(PAR-1)} \\
\\
\frac{\langle C_2, s \rangle \Longrightarrow^c \langle C'_2, s' \rangle}{\langle \text{par } C_1 \text{ with } C_2 \text{ end}, s \rangle \Longrightarrow^c \langle \text{par } C_1 \text{ with } C'_2 \text{ end}, s' \rangle} \text{(PAR-2)} \\
\\
\frac{}{\langle \text{par skip with skip end}, s \rangle \Longrightarrow^c \langle \text{skip}, s \rangle} \text{(PAR-END)}
\end{array}$$

The above set of rules is encoded in COQ through the following inductive predicate, considering that the definition `(stmt*st)` stands for the type of configurations:

```

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 C'1 s' C2,
  cstep (C1, s) (C'1, s') → cstep ((C1; C2), s) ((C'1; C2), s')

```

```

|CS_SeqFinish :  $\forall s C_2,$ 
  cstep ((skip;  $C_2$ ),  $s$ ) ( $C_2, s$ )
|CS_IfFalse :  $\forall s C_1 C_2 b,$ 
   $\neg b \text{assrt } b \ s \rightarrow \text{cstep (if } b \text{ then } C_1 \text{ else } C_2 \text{ fi, } s) (C_2, s)$ 
|CS_IfTrue :  $\forall s c_1 c_2 b,$ 
   $b \text{assrt } b \ s \rightarrow \text{cstep (if } b \text{ then } C_1 \text{ else } C_2 \text{ fi, } s) (C_1, s)$ 
|CS_While :  $\forall s b C,$ 
  cstep (while  $b$  do  $C$  end,  $s$ ) (if  $b$  then ( $C$ ; while  $b$  do  $C$  end) else skip fi,  $s$ )
|CS_Par1 :  $\forall s C_1 C'_1 C_2 s',$ 
  cstep ( $C_1, s$ ) ( $C'_1, s'$ )  $\rightarrow \text{cstep (par } C_1 \text{ with } C_2 \text{ end, } s) (\text{par } C'_1 \text{ with } C_2 \text{ end, } s')$ 
|CS_Par2 :  $\forall s C_1 C_2 C'_2 s',$ 
  cstep ( $C_2, s$ ) ( $C'_2, s'$ )  $\rightarrow \text{cstep (par } C_1 \text{ with } C_2 \text{ end, } s) (\text{par } C_1 \text{ with } C'_2 \text{ end, } s')$ 
|CS_Par_end :  $\forall s,$ 
  cstep (par skip with skip end,  $s$ ) (skip,  $s$ ) .

```

Infix " \xRightarrow{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 \xRightarrow{c} .

Example 35. 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:

$$\text{par } \{ x_1 := 1 \} \text{ with } \{ x_2 := 2 \} \text{ end}$$

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,

$$\langle \text{par} \{ x_1 := 1 \} \text{ with } \{ x_2 := 2 \} \text{ end, } s \rangle \xRightarrow{c} \langle \text{par } \{ \text{skip} \} \text{ with } \{ x_2 := 2 \} \text{ end, } s[1/x_1] \rangle,$$

or it reduces by rule (PAR-2) and updates s by mapping 2 to the variable x_2 , that is,

$$\langle \text{par} \{ x_1 := 1 \} \text{ with } \{ x_2 := 2 \} \text{ end, } s \rangle \xRightarrow{c} \langle \text{par } \{ x_1 := 1 \} \text{ with } \{ \text{skip} \} \text{ end, } s[2/x_2] \rangle.$$

5.3.1 Reductions Under Interference

The semantics of the relation \xRightarrow{c} is not enough to capture possible interference between parallel programs. In order to capture it 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:

$$\langle C, s \rangle \xRightarrow{R} \langle C', s' \rangle \stackrel{\text{def}}{=} (\langle C, s \rangle \xRightarrow{c} \langle C', s' \rangle) \vee (C = C' \wedge (s, s') \in R). \quad (5.2)$$

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 <=> cf' ∨ interf R cf cf'.
```

Example 36. Let s be a state such that the variable x_1 has the value 1 and the variable x_2 has de value 1 also. Let R be the rely condition defined by $(x, x + 1) \in R$, 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:

$$\langle x_1 := 1; x_2 := 2, s \rangle \Longrightarrow \langle \text{skip}; x_2 := 2, s[1/x_1] \rangle$$

or

$$\langle x_1 := 1; x_2 := 2, s \rangle \Longrightarrow \langle x_1 := 1; x_2 := 2, s[2/x_2] \rangle.$$

Now that we already have a definition for one step of computation of 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 \xRightarrow{R} , that is,

$$\langle C, s \rangle \xRightarrow{R^*} \langle C', s' \rangle. \quad (5.3)$$

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  $\xRightarrow{R^n}$  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  $\xRightarrow{R^*}$  cf'" := (prog_red_star R cf cf').
```

Example 37. *Let us consider the same initial state and program as in the previous example. 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 $\langle C, s \rangle \xRightarrow{c^3} \langle \text{skip}, s[1/x_1][2/x_2] \rangle$. With one more reduction, and because the relation R is defined as $R \in (x, x + 1)$, we can prove $\langle \text{skip}, s[1/x_1][2/x_2] \rangle \xRightarrow{R} \langle \text{skip}, s[1/x_1][2/x_2][2/x_1] \rangle$.*

5.4 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.

Before presenting the inference system HL-HG, 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 is not changed due to the actions caused by R , that is

$$\text{stable } R P \stackrel{\text{def}}{=} \forall s, s' \in \Sigma, P(s) \rightarrow (s, s') \in R \rightarrow P(s'). \quad (5.4)$$

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 be 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` \rightarrow `Prop`.

Definition `b2assrt` (b :`bexp`) : `assrt` :=
`fun s:st \Rightarrow beval s b = true.`

Definition `assg_subs` (x :`id`) (e :`aexp`) (Q :`assrt`) : `assrt` :=
`fun s:st \Rightarrow Q (s [(aeval s e)/x]).`

Definition `assrtT` (b : `bexp`) : `assrt` :=
`fun s:st \Rightarrow b2assrt b s.`

Definition `assrtF` (b : `bexp`) : `assrt` :=
`fun s:st \Rightarrow \neg b2assrt b s.`

Definition `assrtAnd` (P Q :`assrt`) : `assrt` :=
`fun s:st \Rightarrow P s \wedge Q s.`

Definition `assrtOr` (P Q : `assrt`) : `assrt` :=
`fun s:st \Rightarrow P s \vee Q s.`

Definition `assrtImp` (P Q : `assrt`) : `Prop` :=
 $\forall s:st, P s \rightarrow Q s.$

With the previous definitions, we can define the notion of stability in COQ, as follows:

Definition `stable` (R :`relation st`) (P :`assrt`) :=
 $\forall x y:st, P x \wedge R x y \rightarrow P y.$

Lemma `stable_starn` :
 $\forall n:nat, \forall R P, \text{stable } R P \rightarrow \text{stable } (\text{starn } _ R n) P.$

Lemma `stable_star` :
 $\forall R P, \text{stable } R P \rightarrow \text{stable } (\text{star } _ R) P.$

Lemma `stable_and` :
 $\forall R_1 R_2 P, \text{stable } R_1 P \rightarrow \text{stable } R_2 P \rightarrow \text{stable } (\text{rstAnd } R_1 R_2) P.$

Lemma `stable_impl` :

$$\forall R_1 R_2 P, \text{stable } R_2 P \rightarrow \text{rstImp } R_1 R_2 \rightarrow \text{stable } R_1 P.$$

We now give a simple example that shows an assertion being stable with respect to a possible rely condition.

Example 38. Let R be a relation defined by $R = \{(x, x+k) \mid k > 0\}$, and let $P(s) = s > 0$. 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.

5.4.1 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, there exists no program reductions. Thus, only the environment R can change the underlying state, and so, 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:

$$\frac{\text{stable } R P \quad \text{stable } R Q \quad P \rightarrow Q}{\{R, P\} \text{ skip } \{Q, G\}} \text{ (HG-SKIP)}$$

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:

$$\frac{\text{stable } R \ P \quad \text{stable } R \ Q \quad (\forall s \in \Sigma, (s, s[e/x]) \in G) \quad P \rightarrow Q[a/x]}{\{R, P\} \ x ::= e \ \{Q, G\}} \text{ (HG-ASSGN)}$$

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:

$$\frac{\{R, P\} \ C_1 \ \{Q', G\} \quad \{R, Q'\} \ C_2 \ \{Q, G\}}{\{R, P\} \ C_1; C_2 \ \{Q, G\}} \text{ (HG-SEQ)}$$

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:

$$\frac{\text{stable } R \ \llbracket b \rrbracket \quad \{R, P \wedge \llbracket b \rrbracket_{\mathbb{B}}\} \ C_1 \ \{Q, G\} \quad \text{stable } R \ \llbracket \bar{b} \rrbracket \quad \text{stable } R \ P \quad \{R, P \wedge \llbracket \bar{b} \rrbracket_{\mathbb{B}}\} \ C_2 \ \{Q, G\}}{\{R, P\} \ \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi } \ \{Q, G\}} \text{ (HG-IF)}$$

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. Moreover, stability must also apply to the invariant. The inference rule for the while loop is defined as follows:

$$\frac{\text{stable } R \ \llbracket b \rrbracket_{\mathbb{B}} \quad \text{stable } R \ \llbracket \bar{b} \rrbracket_{\mathbb{B}} \quad \{R, \llbracket b \rrbracket_{\mathbb{B}} \wedge P\} \ C \ \{Q, G\}}{\{R, P\} \ \text{while } b \text{ do } C \text{ done } \ \{\llbracket \bar{b} \rrbracket_{\mathbb{B}} \wedge P, G\}} \text{ (HG-WHILE)}$$

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:

$$\frac{\text{stable } R \ P \quad \text{stable } R \ Q \quad \{ID, P\} \ C \ \{Q\}}{\{R, P\} \ \text{atomic}(C) \ \{Q, G\}} \text{ (HG-ATOMIC)}$$

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:

$$\frac{\begin{array}{ccc} P \rightarrow P' & R \rightarrow R' & \\ Q' \rightarrow Q & G' \rightarrow G & \{R', P'\} \ C \ \{Q', G'\} \end{array}}{\{R, P\} \ C \ \{Q, G\}} \text{ (HG-CONSEQ)}$$

Parallel composition. In the case of parallel composition of two programs C_1 and C_2 assumes that the specifications of the individual programs ensure that they not interfere with each. 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:

$$\frac{\begin{array}{lll} (G_l \cup G_r) \rightarrow G & & \\ (R_l \cup G_l) \rightarrow R_r & \text{stable } (R_r \cup G_r) \ Q_1 & \\ (R_r \cup G_r) \rightarrow R_l & \text{stable } (R_l \cup G_l) \ Q_2 & \\ (R_l \wedge R_r) \rightarrow R & \text{stable } (R_r \cup G_r) \ P & \{R_l, P\} \ C_1 \ \{Q_1, G_l\} \\ (Q_l \wedge Q_r) \rightarrow Q & \text{stable } (R_l \cup G_l) \ P & \{R_r, P\} \ C_2 \ \{Q_2, G_r\} \end{array}}{\{R, P\} \ \text{par } C_1 \ \text{with } C_2 \ \text{end} \ \{Q, G\}} \text{ (HG-PAR)}$$

In COQ, we define the inference system RG-HL 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 :  $\forall$  (P Q:assrt) c b,
  ( $\forall$  x y, star _ G x y  $\rightarrow$  G x y)  $\rightarrow$ 
  stable R P  $\rightarrow$  stable R Q  $\rightarrow$  triple G P c Q  $\rightarrow$ 
  triple_rg R G P (wait b do c end) Q
| RIf:  $\forall$  P c1 c2 Q b,
  Reflexive G  $\rightarrow$ 
  stable R (assrtT b)  $\rightarrow$  stable R (assrtF b)  $\rightarrow$ 
  stable R P  $\rightarrow$ 
  triple_rg R G (([T]b)[ $\wedge$ ]P) c1 Q  $\rightarrow$ 
  triple_rg R G (([ $\perp$ ]b)[ $\wedge$ ]P) c2 Q  $\rightarrow$ 
  triple_rg R G P (ifb b then c1 else c2 fi) Q
| RSequence:  $\forall$  c1 c2 P K Q,
  Reflexive G  $\rightarrow$  triple_rg R G P c1 K  $\rightarrow$  triple_rg R G K c2 Q  $\rightarrow$ 
  triple_rg R G P (c1;c2) Q
| RConseq:
 $\forall$  (R' G':Env) P P' Q Q' c,
  triple_rg R' G' P' c Q'  $\rightarrow$ 
  assrtImp P P'  $\rightarrow$  assrtImp Q' Q  $\rightarrow$ 
  rstImp R R'  $\rightarrow$  rstImp G' G  $\rightarrow$ 
  triple_rg R G P c Q
| RLoop :
 $\forall$  P b c,
  Reflexive G  $\rightarrow$  stable R P  $\rightarrow$ 
  stable R (assrtT b)  $\rightarrow$  stable R (assrtF b)  $\rightarrow$ 
  triple_rg R G (([T]b)[ $\wedge$ ](P)) c P  $\rightarrow$ 
  triple_rg R G P (while b do c end) (([ $\perp$ ]b)[ $\wedge$ ]P)
| RConcur :
 $\forall$  (Rl Rr Gl Gr:StR) P Q1 Q2 Q cr cl,
  Reflexive Gl  $\rightarrow$  Reflexive Gr  $\rightarrow$ 
  rstImp R (rstAnd Rl Rr)  $\rightarrow$ 
  rstImp (rstOr Gl Gr) G  $\rightarrow$ 
  rstImp (rstOr Rl Gl) Rr  $\rightarrow$ 
  rstImp (rstOr Rr Gr) Rl  $\rightarrow$ 
  (assrtAnd Q1 Q2)[ $\rightarrow$ ]Q  $\rightarrow$ 
  stable (rstOr Rr Gr) Q1  $\rightarrow$ 
  stable (rstOr Rl Gl) Q2  $\rightarrow$ 
  stable (rstOr Rr Gr) P  $\rightarrow$ 
  stable (rstOr Rl Gl) P  $\rightarrow$ 
  triple_rg Rl Gl P cl Q1  $\rightarrow$ 
  triple_rg Rr Gr P cr Q2  $\rightarrow$ 
  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 in this dissertation. The proof of the soundness of sequential Hoare logic captured by `triple` follows along the lines of the works of Leroy [73] and Pierce *et al.* [89], and is based also in a small-step reduction semantics.

5.5 Soundness of RG-HL

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 [28], we prove the soundness of the system with respect to its functional correctness, *i.e.*, that preconditions and postconditions are ensured, and we also prove that it satisfies the constraints imposed by the guarantee condition. Together, these lead to the correctness of the whole proof system.

Here we do not present the actual COQ scripts required to build the proofs. Instead, we give proof sketches for each of the cases of the proof, that give an overview on how the actual proofs were mechanised. The actual COQ code is available in [].

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 $\langle C', s' \rangle \xRightarrow{c} \langle C'', s'' \rangle$ holds, then we must show that $(s', s'') \in G$, where G is the established guarantee condition. Proving that 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 [28], and is formally defined by

$$\begin{aligned} & \text{within}(R, G, C, s) \\ & \quad \stackrel{\text{def}}{=} \\ & \quad \forall C' s', (\langle C, s \rangle \xRightarrow{R^*} \langle C', s' \rangle) \rightarrow \forall C'' s'', \langle C', s' \rangle \xRightarrow{c} \langle C'', s'' \rangle \rightarrow (s', s'') \in G. \end{aligned} \tag{5.5}$$

An important consequence of the previous definition is that given two programs C and C' , we can prove 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, we have

$$\forall C_1 C_2 C'_1 s', \langle \text{par } C_1 \text{ with } C_2 \text{ end}, s \rangle \xRightarrow{R^*} \langle \text{par } C'_1 \text{ with } C_2 \text{ end}, s' \rangle \rightarrow (s, s') \in (R \cup G_l)^* \quad (5.6)$$

$$\forall C_1 C_2 C'_2 s', \langle \text{par } C_1 \text{ with } C_2 \text{ end}, s \rangle \xRightarrow{R^*} \langle \text{par } C_1 \text{ with } C'_2 \text{ end}, s' \rangle \rightarrow (s, s') \in (R \cup G_r)^*, \quad (5.7)$$

where G_l and G_r are, respectively, the guarantee conditions of the programs C_1 and C_2 . This properties will be fundamental for proving the soundness of the parallel computation inference rule for the RG proof system that we will introduce briefly. Other properties of *within* are the following: if a reduction of the program exists, or one step of interference occurs, then the *within* still holds, that is,

$$\forall s s', (\langle C, s \rangle \xRightarrow{c} \langle C', s' \rangle) \rightarrow \text{within}(R, G, C, s) \rightarrow \text{within}(R, G, C', s')$$

and

$$\forall s s', (s, s') \in R \rightarrow \forall C, \text{within}(R, G, C, s) \rightarrow \text{within}(R, G, C, s').$$

The previous properties are naturally extended to a finite set of reductions under interference starting from a configuration $\langle C, s \rangle$. Formally,

$$\forall s s', (\langle C, s \rangle \xRightarrow{R^*} \langle C', s' \rangle) \rightarrow \text{within}(R, G, C, s) \rightarrow \text{within}(R, G, C', s'). \quad (5.8)$$

Another property of interest and that we will need for the soundness proof of the parallel statement is that if $\text{within}(R, G, C, s)$ holds and if the $\langle C, s \rangle$ 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:

$$\forall s s', (\langle C, s \rangle \xRightarrow{R^*} \langle C', s' \rangle) \rightarrow \text{within}(R, G, C, s) \rightarrow (s, s') \in (R \cup G)^*. \quad (5.9)$$

Soundness Proof

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 . Formally, the definition of Hoare validity for HL-RG is defined as

$$\begin{aligned} & \models \{R, P\} C \{Q, G\} \\ & \stackrel{\text{def}}{=} \\ & \forall C, s, P(s) \rightarrow \forall s', (\langle C, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle) \rightarrow Q(s') \wedge \text{within}(R, G, C, s). \end{aligned} \quad (5.10)$$

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 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

$$\vdash \{R, P\} C_i \{Q, G\} \rightarrow \forall s, P(s) \rightarrow \forall s', \langle C_i, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle \rightarrow Q(s') \quad (5.11)$$

and also

$$\vdash \{R, P\} C_i \{Q, G\} \rightarrow (\forall s, P(s) \rightarrow \forall s', \langle C_i, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle \rightarrow \text{within}(R, G, C_i, s)). \quad (5.12)$$

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 $\langle \text{skip}, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$ we know that $(s, s') \in R^*$ 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 $\langle \text{skip}, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$ and $\langle \text{skip}, s \rangle \xRightarrow{c} \langle C, s' \rangle$ for some state s' . This is, however, an absurd since no reduction $\langle \text{skip}, s \rangle \xRightarrow{c} \langle C, s' \rangle$ exists in the definition of \xRightarrow{c} .

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 stretched 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 $\langle x ::= e, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$ we obtain $\langle \text{skip}, s \rangle \xRightarrow{c} \langle \text{skip}, s[e/x] \rangle$ and $\langle \text{skip}, s[e/x] \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$. Since **skip** implies the impossibility of reductions, we are able to infer that $(s[e/x], s') \in R^*$. 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 $\langle x ::= e, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$ that exists $s'' \in \Sigma$ such that $(s, s'') \in R$ and $\langle x ::= e, s'' \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$. 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 $\text{within}(R, G, x ::= e, s)$, we first obtain that if $\langle x ::= e, s \rangle \xRightarrow{R^*} \langle C', s' \rangle$ then both $C' = \text{skip}$ 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

$$\langle C_1; C_2, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$$

for $s, s' \in \Sigma$, then there exists intermediate state $s'' \in \Sigma$ such that $\langle C_1, s \rangle \xRightarrow{R^*} \langle \text{skip}, s'' \rangle$ and $\langle C_2, s'' \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$ hold. Using $\langle C_1, s \rangle \xRightarrow{R^*} \langle \text{skip}, s'' \rangle$ and the induction hypotheses, we show that the postcondition of C_1 is the precondition of C_2 , and by $\langle C_2, s'' \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$ we obtain the postcondition of C_2 , which finishes the proof.

Guarantee part. For proving $\text{within}(R, G, C_1; C_2, s)$ we need the following intermediate

lemma:

$$\begin{aligned} & \text{within}(R, G, C_1, s) \\ & \quad \rightarrow \\ & (\forall s' \in \Sigma, \langle C_1, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle \rightarrow \text{within}(R, G, C_2, s')) \rightarrow \text{within}(R, G, C_1; C_2, s). \end{aligned} \quad (5.13)$$

By applying (5.13) to $\text{within}(R, G, C_1; C_2, s)$, we are left to prove first that $\text{within}(R, G, C_1, s)$, 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

$$\forall s' \in \Sigma, \langle C_1, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle \rightarrow \text{within}(R, G, C_2, s'),$$

we obtain $Q'(s')$, and from $Q'(s')$ and $\models \{R, Q'\} C_2 \{Q, G\}$ we obtain $\text{within}(R, G, C_2, s')$, 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

$$\langle \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi}, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$$

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 rely condition R with respect to the value of guard b , which keeps its value unchanged. Once this is proved, the rest of the proof follows also from the hypotheses.

Guarantee part. In order to prove

$$\text{within}(R, G, \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi}, s),$$

we require the following auxiliary lemmas:

$$\text{within}(R, G, C_1, s) \rightarrow \llbracket b \rrbracket_{\mathbb{B}}(s) \rightarrow \forall C_2, \text{within}(R, G, \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi}, s) \quad (5.14)$$

and

$$\text{within}(R, G, C_2, s) \rightarrow \overline{\llbracket b \rrbracket}_{\mathbb{B}}(s) \rightarrow \forall C_1, \text{within}(R, G, \text{if } b \text{ then } C_1 \text{ else } C_2 \text{ fi}, s). \quad (5.15)$$

Since $\text{within}(R, G, C_1, s)$ and $\text{within}(R, G, C_2, s)$ 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 adequate elimination scheme 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 scheme:

$$\begin{aligned} & \forall x : A, \forall P : A \rightarrow A \rightarrow \text{Prop}, (x, x) \in P \rightarrow \\ & (\forall n \ x \ y \ z, (x, y) \in R \rightarrow (y, z) \in R^n \rightarrow \\ & (\forall y_1 \ k, k \leq n \rightarrow (y_1, z) \in R^k \rightarrow (y_1, z) \in P) \rightarrow (x, z) \in P) \rightarrow \\ & \forall x \ y, (x, y) \in R \rightarrow (x, y) \in P. \end{aligned}$$

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 than n steps P must hold. The idea is to instantiate this elimination scheme to the case of the while loop. In order to correctly apply this predicate, we first need to transform the validity condition

$$\begin{aligned} & \forall s, P(s) \rightarrow \\ & \forall s', \text{star_} \ (\text{prog_red } R) \ (\text{while } b \ \text{do } c \ \text{end}, s) \ (\text{skip}, s') \rightarrow \\ & ([\perp]b) [\wedge]P(s'). \end{aligned}$$

into its equivalent form

$$\begin{aligned} & \forall s, P \ s \rightarrow \\ & (\forall s', \text{star_} \ (\text{prog_red } R) \ (\text{while } b \ \text{do } c \ \text{end}, s) \ (\text{skip}, s') \rightarrow \\ & \forall p \ p', \\ & \text{star_} \ (\text{prog_red } R) \ p \ p' \rightarrow \\ & \text{fst } p = (\text{while } b \ \text{do } c \ \text{end}) \rightarrow \\ & \text{fst } p' = \text{skip} \rightarrow \\ & P \ (\text{snd } p) \rightarrow ([\perp]b) [\wedge]P \ (\text{snd } p')). \end{aligned}$$

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 $\langle \text{skip}, s \rangle \xRightarrow{R^n} \langle \text{skip}, s' \rangle$, 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 $P \wedge \llbracket \bar{b} \rrbracket_{\mathbb{B}}(s')$ to $P \wedge \llbracket \bar{b} \rrbracket_{\mathbb{B}}(s)$, which is available from the hypotheses;
- if the Boolean guard evaluates to a true, we know that the loop reduces to

$$\langle C; \text{while } b \text{ do } C \text{ end}, s \rangle \xRightarrow{R^n} \langle \text{skip}, s' \rangle,$$

which we decompose into

$$\langle C, s \rangle \xRightarrow{R^m} \langle \text{skip}, s'' \rangle,$$

and $\langle \text{while } b \text{ do } C \text{ end}, s'' \rangle \xRightarrow{R^{(n-m)}} \langle \text{skip}, s' \rangle$, 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 is also constrained by G , which allows us to conclude that the loop satisfies the guarantee condition.

Unfortunately, we were not able to complete this proof within the COQ formalisation. Here we give the partial proof that we have obtained and show the point where we are stuck. The goal is to prove that

$$\text{within}(R, G, C, s) \rightarrow \text{within}(R, G, \text{while } b \text{ do } C \text{ end}, s),$$

regarding that we already know that $\{R, P\} C \{Q, G\}$ holds. From this last hypothesis and because we know that $\langle \text{while } b \text{ do } C \text{ end}, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$, we also know that the postcondition $(P \wedge \llbracket \bar{b} \rrbracket)(s')$ holds. Next, we perform case analysis on the value of the Boolean guard b , that is:

- if $\llbracket b \rrbracket_{\mathbb{B}}(s) = \text{false}$ then we know that $\text{within}(R, G, \text{while } b \text{ do } C \text{ end}, s)$ can be replaced by

$$\text{within}(R, G, \text{if } b \text{ then } (\text{while } b \text{ do } C \text{ end}) \text{ else skip fi}, s)$$

and from the latter statement we are able to conclude $\text{within}(R, G, \text{skip}, s)$ holds and that is true, as we have showed before.

- if $\llbracket b \rrbracket_{\mathbb{B}}(s) = \text{true}$, then we know that $\text{within}(R, G, \text{while } b \text{ do } C \text{ end}, s)$ can be replaced by

$$\text{within}(R, G, \text{if } b \text{ then } (\text{while } b \text{ do } C \text{ end}) \text{ else skip fi}, s)$$

, which in turn is equivalent to $\text{within}(R, G, C; \text{while } b \text{ do } C \text{ end}, s)$. By the properties of the **within** predicate on sequential execution, we reduce the following goal to a proof that $\text{within}(R, G, C, s)$ (which is immediate from the hypotheses) and to a proof that

$$\forall n \in \mathbb{N}, \forall s'' \in \Sigma, (\langle C, s \rangle \xRightarrow{R^n} \langle \text{skip}, s'' \rangle) \rightarrow \text{within}(R, G, \text{while } b \text{ do } C \text{ end}, s'').$$

At this point we got stuck in the proof, since we were not able to find an appropriate logical condition that allow us to prove it.

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 $\langle C, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle$, 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 $\langle C, s \rangle \xRightarrow{*} \langle \text{skip}, s' \rangle$ then $Q(s')$ and $(s, s') \in G^*$ 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

$$\langle \text{atomic}(C), s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle,$$

those hypothesis are the following:

$$(s, t) \in R^*, \tag{5.16}$$

$$(t', s') \in R^*, \tag{5.17}$$

$$\langle C, t \rangle \xRightarrow{ID^*} \langle \text{skip}, t' \rangle, \quad (5.18)$$

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 $\langle C, t \rangle \xRightarrow{*} \langle \text{skip}, t' \rangle$. 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 $(t, t') \in G^*$ using the same reasoning that we employed to obtain $Q(t')$. By the transitivity of the guarantee condition we know that $(t, t') \in G$, which 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

$$\langle \text{par } C_1 \text{ with } C_2 \text{ end}, s \rangle \xRightarrow{(R_1 \wedge R_2)^*} \langle \text{skip}, s' \rangle, \quad (5.19)$$

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

$$\text{within}(R_1, G_1, C_1, s), \quad (5.20)$$

$$\text{within}(R_2, G_2, C_2, s), \quad (5.21)$$

$$(\langle C_1, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle) \rightarrow Q_1(s'), \quad (5.22)$$

$$(\langle C_2, s \rangle \xRightarrow{R^*} \langle \text{skip}, s' \rangle) \rightarrow Q_2(s'). \quad (5.23)$$

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 starts 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:

$$\langle C_1, s \rangle \xRightarrow{R_1^*} \langle \text{skip}, s'' \rangle, \quad (5.24)$$

$$\langle C_2, s \rangle \xRightarrow{R_2^*} \langle C'_2, s'' \rangle, \quad (5.25)$$

$$\langle \text{par skip with } C'_2 \text{ end}, s'' \rangle \xRightarrow{(R_1 \wedge R_2)^*} \langle \text{skip}, s' \rangle. \quad (5.26)$$

Since both (5.21) and (5.25) hold, then by (5.8) we conclude $\text{within}(R_2, G_2, C'_2, s'')$. Moreover, from $\text{within}(R_2, G_2, C'_2, s'')$ and (5.26) we also conclude

$$\text{within}(R_2, G_2, \text{par skip with } C'_2 \text{ end}, s''). \quad (5.27)$$

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:

$$\begin{aligned} \text{within}(R \vee G_2, G_1, C_1, s) &\rightarrow \text{within}(R \vee G_1, G_2, C_2, s) \rightarrow \\ &\text{within}(R, G, \text{par } C_1 \text{ with } C_2 \text{ end}, s). \end{aligned} \quad (5.28)$$

Using (5.28) we are left to prove $\text{within}(R \vee G_2, G_1, C_1, s)$ and $\text{within}(R \vee G_1, G_2, C_2, s)$. 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 $\text{within}(R \vee G_2, G_1, C_1, s)$ is equivalent to

$$\text{within}((R_1 \wedge R_2) \vee G_2, G_1, C_1, s).$$

From the hypotheses, we know that $\text{within}(R_1, G_1, C_1, s)$ and that $(R_1 \wedge R_2) \vee G_2 \rightarrow R_1$. By the properties of the implication and the predicate within we conclude the proof.

5.6 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 [28]. Related

work includes the work of these authors, and also the work of Prensa Nieto [82, 91] in the Isabelle proof assistant.

Our formalisation essentially confirms most of the work introduced by Coleman and Jones [28], but extended with atomic execution of programs. Thus, our development shows that the ideas forwarded by these authors seem to be correct, despite an incorrect rule they **use** to decompose sequential composition of programs, and also assuming that we were not able to finish the guarantee part for the inference rule for loops not because it is unsound, but just because we did not yet found the correct way of doing 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. used

In what concerns to the comparison of our work with the one of Prensa Nieto, the main differences are the following: Nieto formalises a notion of parameterised parallelisms, where a parallel program is seen as a list of sequential programs. Therefore, it cannot handle nested parallelism. Nevertheless, the author was able to **prove more useful than us**, mainly because her notion of RG is more flexible than ours. não se entende

5.6.1 Discussion and Conclusion

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 39. *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 or equal to 0. The corresponding IMPp program code is the following:*

$$C \stackrel{\text{def}}{=} \text{par } x ::= x + 1 \text{ with } x ::= x + 2 \text{ end.}$$

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,

$$R \stackrel{\text{def}}{=} G \stackrel{\text{def}}{=} \lambda s \lambda s'. \llbracket x \rrbracket_{\mathbb{N}}(s) \leq \llbracket x \rrbracket_{\mathbb{N}}(s').$$

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,.

$$P \stackrel{\text{def}}{=} \lambda x. \llbracket x \rrbracket_{\mathbb{N}}(s) \geq 0,$$

$$Q \stackrel{\text{def}}{=} \lambda x. \llbracket x \rrbracket_{\mathbb{N}}(s) \geq 2.$$

The full specification corresponds to $\{R, P\} \vdash \{Q, G\}$ 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 conjunction implies C . We instantiate these new variables with

$$\begin{array}{ll} R_1 \stackrel{\text{def}}{=} R & R_2 \stackrel{\text{def}}{=} R \\ G_1 \stackrel{\text{def}}{=} G & G_2 \stackrel{\text{def}}{=} G \\ C_1 \stackrel{\text{def}}{=} x \geq 1 & C_2 \stackrel{\text{def}}{=} x \geq 2. \end{array}$$

The goal associated with the stability conditions are proved in a straightforward way. Next, we prove

$$\{R_1, P\} \vdash x := x + 1 \{Q_1, G_1\}$$

and

$$\{R_2, P\} \vdash x := x + 2 \{Q_2, G_2\}$$

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 dues to the fact that rely and guarantee conditions must describe the state of computation 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 [1]. In the case of our reference work, such rules are not **accounter** for, and neither did we. Unfortunately, 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 **the** our reference work. This means that, in the future, we have to rethink the choices we have made to the work presented here, and find ways to improve it in some way. ?

5.7 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 [1], but ended up in a proof system more restricted than this one. 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 to go through this line in order to get a better knowledge of the difficulties that arise when formalizing a proof system for the certification of shared-variable parallel programs based on the 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, introduced in [chapter ??](#) 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 HG, which is an extension of the inference system proposed by Coleman and Jones in [28] 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 [101, 38, 37] 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 [chapters 3 and 4](#) and, with them, try to define decision procedures for RG reasoning within the context of CKA.

Chapter 6

Conclusions

Along this dissertation we have described three contributions that we believe ~~that~~ can be useful in tasks of 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 expression ~~s~~ 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 deliberates 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 build proofs about programs represented as relations, ~~like~~ ^{as} it is usually done in a point-free approach. 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 developments require the treatment of regular expressions.

We also implemented a correct algorithm for deciding the equivalence of KAT terms. KAT terms have the expressivity to specify propositional programs, *i.e.*, programs with no assignments. The assignments, and other possible first order constructs can be encoded as either Boolean tests or 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 essential 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, and that 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.

6.1 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 following along the ~~de~~-lines of [1]. We are also interested in making the development more robust, by developing proof tactics that automate proof steps that are **common** 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 [2] 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 an 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 bases of a certified kernel for a proof assistant similar to KAT-ML that was developed by Aboul-Hosn and Kozen [2].

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 develop-

ment, and extend it to handle memory properties along the lines of RGSEP [101] or *deny guarantee* [37]. However, we are more interested in the algebraic approach followed by Hoare *et. al.* with CKA. 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 procedure that may help, at some point, in the automation of proof construction of specifications of concurrent and parallel programs.

References

falta autor

- [1] **Timed regular expressions.** *J. ACM*, 49(2):172–206, Mar. 2002.
- [2] 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.
- [3] 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.
- [4] 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>.
- [5] M. Almeida, N. Moreira, and R. Reis. Antimirov and Mosses’s rewrite system revisited. *Int. J. Found. Comput. Sci.*, 20(4):669–684, 2009.
- [6] M. Almeida, N. Moreira, and R. Reis. Antimirov and Mosses’s rewrite system revisited. *International Journal Of Foundations Of Computer*, 20(04):669 – 684, 2009.
- [7] M. Almeida, N. Moreira, and R. Reis. Testing regular languages equivalence. *JALC*, 15(1/2):7–25, 2010.
- [8] R. Almeida. Decision algorithms for Kleene algebra with tests and Hoare logic. Master’s thesis, Faculdade de Ciências da Universidade do Porto, 2012.
- [9] 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.
- [10] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

- [11] 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.
- [12] 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.
- [13] 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 *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011.
- [14] 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.
- [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. Briaïs. Finite automata theory in Coq. http://sbriaïs.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] N. Bruijn. *Automath: A Language for Mathematics*. Séminaire de Mathématiques Supérieures. Les Presses de L’Université de Montréal, 1973.

- [23] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
- [24] 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 *Lecture Notes in Computer Science*, pages 179–191. Springer, 2011.
- [25] J.-M. Champarnaud and D. Ziadi. From Mirkin’s prebases to Antimirov’s word partial derivatives. *Fundam. Inform.*, 45(3):195–205, 2001.
- [26] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. <http://adam.chlipala.net/cpdt/>.
- [27] J. Chrzaszcz. Implementing modules in the coq system. In D. A. Basin and B. Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003.
- [28] 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.
- [29] T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.
- [30] 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.
- [31] 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 *Lecture Notes in Computer Science*. Springer, 2006.
- [32] 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.
- [33] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *CoRR*, cs.LO/0310054, 2003.
- [34] J. Desharnais, B. Möller, and G. Struth. Modal kleene algebra and applications – a survey, 2004. tech report?
- [35] T. C. development team. Coq reference manual. <http://coq.inria.fr/distrib/V8.3/refman/>.

- [36] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.
- [37] 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.
- [38] X. Feng. Local rely-guarantee reasoning. *SIGPLAN Not.*, 44(1):315–327, Jan. 2009.
- [39] J.-C. Filliâtre. Finite Automata Theory in Coq: A constructive proof of Kleene’s theorem. Research Report 97–04, LIP - ENS Lyon, February 1997.
- [40] J.-C. Filliâtre and P. Letouzey. Functors for proofs and programs. In D. A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2004.
- [41] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.
- [42] M. J. Frade and J. S. Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252 – 277, 2011.
- [43] G. Gonthier. Computer mathematics. chapter The Four Colour Theorem: Engineering of a Formal Proof, pages 333–333. Springer-Verlag, Berlin, Heidelberg, 2008.
- [44] 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>.
- [45] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. *SIGPLAN Not.*, 37(9):235–246, Sept. 2002.
- [46] C. Hardin. Modularizing the elimination of $r=0$ in kleene algebra. *Logical Methods in Computer Science*, 1(3), 2005.
- [47] C. Hardin and D. Kozen. On the elimination of hypotheses in kleene algebra with tests. Technical report, 2002.

- [48] M. Hennessy. *Semantics of programming languages - an elementary introduction using structural operational semantics*. Wiley, 1990.
- [49] 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.
- [50] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [51] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.
- [52] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent **kleene** algebra and its foundations. *J. Log. Algebr. Program.*, 80(6):266–296, 2011.
- [53] J. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 71 -114, University of California, Berkeley, California, 1971.
- [54] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company, Reading, Massachusetts, USA, 1979.
- [55] W. Howard. *The formulae-as-types notion of construction*, pages 479–490.
- [56] L. Ilie and S. Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.
- [57] C. B. Jones. *Development Methods for Computer Programs including a **No- tion** of Interference*. PhD thesis, Oxford University, June 1981.
- [58] C. B. Jones. *Development Methods for Computer Programs including a **Notion of Interference***. PhD thesis, Oxford University, June 1981.
- [59] D. Kaplan and S. U. C. S. D. A. I. Laboratory. *Regular Expressions and the Equivalence of Programs*. Memo (Stanford Artificial Intelligence Project). The author, 1969.
- [60] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, shannon, C. and McCarthy, J. edition.

- [61] 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.
- [62] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.
- [63] D. Kozen. *Automata and Computability*. Springer-Verlag, New York, 1997.
- [64] D. Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.
- [65] D. Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1):60–76, July 2000.
- [66] D. Kozen. Automata on guarded strings and applications. Technical report, Cornell University, Ithaca, NY, USA, 2001.
- [67] D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Computing and information science technical reports, Cornell University, March 2008.
- [68] D. Kozen and F. Smith. Kleene algebra with tests: Completeness and decidability. In *CSL*, pages 244–259, 1996.
- [69] 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 *Lecture Notes in Computer Science*, pages 244–259. Springer-Verlag, 1996.
- [70] D. Kozen and J. Tiuryn. On the completeness of propositional Hoare logic. In *RelMiCS*, pages 195–202, 2000.
- [71] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Autom. Reasoning*, 49(1):95–106, 2012.
- [72] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
- [73] 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.

- [74] S. Lescuyer. Containers: a typeclass-based library of finite sets/maps. <http://coq.inria.fr/pylons/contribs/view/Containers/v8.3>.
- [75] S. Lescuyer. First-class containers in Coq. *Studia Informatica Universalis*, 9(1):87–127, 2011.
- [76] S. Lescuyer. *Formalisation et développement d’une tactique réflexive pour la démonstration automatique en Coq*. Thèse de doctorat, Université Paris-Sud, Jan. 2011.
- [77] P. Letouzey. *Programmation fonctionnelle certifiée – L’extraction de programmes dans l’assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.
- [78] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, 1984.
- [79] B. Möller and G. Struth. Modal kleene algebra and partial correctness. Technical report, **INSTITUT FÜR INFORMATIK, UNIVERSITÄT AUGSBURG**, 2003.
- [80] B. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.
- [81] H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.
- [82] 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://tumblr.biblio.tu-muenchen.de/publ/diss/allgemein.html>.
- [83] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [84] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, Mar. 2009.
- [85] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.
- [86] C. Paulin-Mohring. Extracting F_ω ’s programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Jan. 1989. ACM.

- [87] 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.
- [88] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
- [89] B. C. Pierce, C. Casinghino, M. Greenberg, C. Hrițcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2012. <http://www.cis.upenn.edu/~bcpierce/sf>.
- [90] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
- [91] 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.
- [92] R. Pucella. On equivalences for a class of timed regular expressions. *Electron. Notes Theor. Comput. Sci.*, 106:315–333, Dec. 2004.
- [93] 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.
- [94] 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.
- [95] K. Sen and G. Rosu. Generating optimal monitors for extended regular expressions. *Electr. Notes Theor. Comput. Sci.*, 89(2):226–245, 2003.
- [96] 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.
- [97] M. Sozeau. Subset coercions in **coq**. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin Heidelberg, 2007.
- [98] M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, Dec. 2009.

- [99] M. Sozeau and O. Nicolas. First-Class Type Classes. *Lecture notes in computer science*, Aug. 2008.
- [100] The Coq Development Team. The Coq proof assistant. <http://coq.inria.fr>.
- [101] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *IN 18TH CONCUR*, pages 256–271. Springer, 2007.
- [102] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.
- [103] 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.
- [104] Q. Xu, W. de Roever, J. He, and J. Hu. *Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs*. Kiel Bericht. Christian-Albrechts-Universität, Institut für Informatik und Praktische Mathematik, 1995.
- [105] Q. Xu, W. paul De Roever, and T. P. S. U. C. Archives. The rely-guarantee method for verifying shared variable concurrent programs. 1997.