Technische Universität München
Lehrstuhl für Logik und Verifikation

Formalizing Symbolic Decision Procedures for Regular Languages

Dmytro Traytel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Dr. h.c. Javier Esparza

Prüfer der Dissertation:

1. Univ.-Prof. Tobias Nipkow, Ph.D.
2. Prof. Dr. Alexandra Silva
   Radboud Universiteit Nijmegen, Niederlande

Die Dissertation wurde am 15.07.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15.10.2015 angenommen.

Abstract

This thesis studies decision procedures for the equivalence of regular languages represented symbolically as regular expressions or logical formulas.

Traditional decision procedures in this context rush to dispose of the concise symbolic representation by translating it into finite automata, which then are efficiently minimized and checked for structural equality.

We develop procedures that avoid this explicit translation by working with the symbolic structures directly. This results in concise functional algorithms that are easy to reason about, even formally. Indeed, the presented decision procedures are specified and proved correct in the proof assistant Isabelle.

The core idea, shared by all procedures under consideration, is the usage of a symbolic derivative operation that replaces the global transition table of the automaton. For regular expressions those are the increasingly popular Brzozowski derivatives and their cousins. For formulas, the development of such operations is the main theoretical contribution of this thesis.

The main technical contribution is the formalization of a uniform framework for deciding equivalence of regular languages and the instantiation of this framework by various symbolic representations. Overall, this yields formally verified executable decision procedures for the equivalence of various kinds of regular expressions, Presburger arithmetic formulas, and formulas of monadic second-order logic on finite words under two different existing semantics (WS1S and M2L(Str)).

Zusammenfassung

Diese Dissertation beschäftigt sich mit Entscheidungsprozeduren für die Äquivalenz regulärer Sprachen, die durch reguläre Ausdrücke oder logische Formeln symbolisch dargestellt sind.

In diesem Kontext entledigen sich traditionelle Entscheidungsprozeduren schnellstmöglich der kompakten symbolischen Darstellung, indem sie sie in endliche Automaten übersetzen. Die Automaten werden anschließend effizient minimiert und auf strukturelle Gleichheit überprüft.

Wir entwickeln Entscheidungsprozeduren, die diese explizite Übersetzung vermeiden und stattdessen mit der symbolischen Darstellung arbeiten. Dies resultiert in kompakten funktionalen Algorithmen, die sich leicht formal verifizieren lassen. Die präsentierten Entscheidungsprozeduren sind nämlich alle in dem Theorembeweiser Isabelle spezifiziert und ihre Korrektheit ist formal bewiesen.

Die allen betrachteten Entscheidungsprozeduren zu Grunde liegende Idee ist die Verwendung von symbolischen Ableitungen, die an Stelle der Übergangstabelle von Automaten treten. Angewendet auf reguläre Ausdrücke entspricht das den immer beliebter werdenden Brzozowski-Ableitungen und ihren Abwandlungen. Die Definition solcher Ableitungsoperationen für logische Formeln ist der zentrale theoretische Beitrag dieser Arbeit.

Die wichtigste technische Errungenschaft ist die Formalisierung einer generischen Entscheidungsprozedur für die Äquivalenz regulärere Sprachen und deren Instanziierung durch verschiedene symbolische Darstellungen. Als Ergebnis erhalten wir formal verifizierte, ausführbare Algorithmen für die Äquivalenz unterschiedlicher Arten von regulären Ausdrücken, sowie Formeln der Presburger-Arithmetik und zweier bekannter Variationen der monadischen Prädikatenlogik zweiter Stufe (WS1S and M2L(Str)).

Acknowledgment

If you are considering doing a Ph.D. and are not averse to theorem proving, I recommend you to do it under Tobias Nipkow's guidance. I cannot imagine a supervisor so liberal and at the same time so supportive (including constantly open doors), not to mention his magic touch for finding sweet spots in research. I very much appreciate that one of such sweet spots became the topic of my Ph.D. thesis. Although he does not quite consider codatatypes a sweet spot, I am grateful that he allowed me to spend a non-negligible amount of time working on this "hobby" of mine.

I was delighted to hear that Alexandra Silva accepted to act as a referee for my thesis. Although, I have not met her in person before the submission of this thesis, her research has been inspiring from the day when I started to think coalgebraically about regular languages.

The collaboration with Damien Pous on the MonaCo tool was fun. I am happy that he was interested in combining formula derivatives with his research. I hope that ideas that went into MonaCo will make some users eventually similarly happy.

Next to Tobias Nipkow, I want to thank two people for explicitly teaching me how to do research. Andrei Popescu and Jasmin Blanchette, the unanimous (co)leaders of the codatatypists gang, are chaos and order, in that order. Andrei taught me how to create precious theory from chaos by thinking abstractly, Jasmin—how to put the theory in order by thinking concretely, yielding usable and used tools. Taking the two together results in a self-reinforcing research machine.

While we are at the codatatypists gang: I am grateful to everyone who has contributed to this project, notably Jasmin Blanchette, Aymeric Bouzy, Martin Desharnais, Johannes Hölzl, Ondřej Kunčar, Andreas Lochbihler, Lorenz Panny, and Andrei Popescu, or at least knows what BNF really stands for.

Ondřej Kunčar, my officemate for the longest part of my Ph.D., deserves a special thanks for engaging me in interesting philosophical and local_theoretical discussions, especially when some procrastination was urgently required. Nobody will take our common, recently acquired hate for beach volleyball players away from us.

More generally, Tobias Nipkow's chair for logic and verification (formerly known as the theorem proving group) was an excellent team to be part of. I had the plea-

sure to work within this "flexible, client-driven organization" and want to thank the people who made it into that pleasant experience (some of which left even before I started, but still stay in touch with the group): Stefan Berghofer, Jasmin Blanchette, Sascha Böhme, Julian Brunner, Lukas Bulwahn, Manuel Eberl, Florian Haftmann, Johannes Hölzl, Brian Huffman, Lars Hupel, Fabian Immler, Gerwin Klein, Alexander Krauss, Ondřej Kunčar, Peter Lammich, Lars Noschinski, Andrei Popescu, Thomas Türk, Makarius Wenzel, the secretaries Silke Müller and Eleni Nikolaou-Weiß, as well as the IT-support team. I especially thank the subset of those people (extended with some external forces) who helped our pub-quiz team laserbroy01 to win various beer-oriented prizes.

I thank Jasmin Blanchette, Florian Haftmann, Johannes Hölzl, Ondřej Kunčar, Peter Lammich, Andrei Popescu, and Damien Pous for proofreading parts of my thesis and supplying numerous helpful comments.

My Ph.D. was funded by the DFG Doctorate Program PUMA which was a nice environment to be in. Helmut Seidl deserves the biggest credit here for his dedication as the coordinator. It was also a great fun to (co)organize the joint PUMA/RiSE workshops together with Fabian Immler, Bogdan Mihaila, and Ana Sokolova.

I thank David Basin, Lutz Schröder, and Christoph Weidenbach for inviting me to present my ongoing Ph.D. research to their groups.

I am grateful to my personal proof assistant, Isabelle, who did a marvelous job of checking my arguments.

But most importantly I thank my family: Anna—my love; Moritz—my true source of sunshine for over two years now; and my parents and co. for their continuous support and endless patience, especially during the time of thesis writing. I dedicate this thesis to them.

Contents

1 Introduction **1**
1.1 Plan of Attack . . . 1
1.2 Related Work . . . 3
1.3 Contributions and Structure . . . 6
1.4 Publications . . . 7

2 Isabelle/HOL and Regular Languages **9**
2.1 Programming and Proving in Isabelle/HOL . . . 10
2.2 Locales . . . 12
2.3 Partiality . . . 13
2.4 Regular Expressions . . . 14

3 Formal Languages, Coinductively **17**
3.1 Languages as Infinite Tries . . . 18
3.2 Regular Operations on Tries . . . 19
3.2.1 Primitively Corecursive Operations . . . 19
3.2.2 Reducing Corecursion Up-To to Primitive Corecursion . . . 21
3.3 Proving Equalities on Tries . . . 24
3.4 Coalgebraic Terminology . . . 28

4 Language Equivalence Framework **31**
4.1 The Algorithm, Informally . . . 31
4.2 Coalgebras as Locales . . . 33
4.3 The Algorithm, Formally . . . 35
4.4 Termination . . . 38

5 Regular Expression Equivalence **41**
5.1 Derivatives . . . 42
5.1.1 Brzozowski Derivatives . . . 42
5.1.2 Partial Derivatives . . . 45
5.2 Marked Regular Expressions . . . 47
5.2.1 Mark After Atom . . . 49
5.2.2 Mark Before Atom . . . 52
5.2.3 Comparison . . . 55

5.3 Empirical Comparison . . . 56
5.4 Extensions . . . 59

6 Π-Extended Regular Expressions 61
6.1 Syntax and Semantics . . . 61
6.2 Decision Procedure . . . 63
6.3 Atoms with More Structure . . . 65
6.4 Alternatives to Brzozowski Derivatives . . . 66

7 WMSO Equivalence via Regular Expression Equivalence 69
7.1 Syntax and Two Semantics . . . 70
7.2 From WMSO Formulas to Regular Expressions . . . 73
7.3 Decision Procedure . . . 78
7.4 Case Study: Finite-Word LTL . . . 79

8 Derivatives of Formulas 83
8.1 Formula Derivatives . . . 84
8.2 Accepting Formulas . . . 88
8.3 Decision Procedure . . . 91
8.4 Minimum Semantics Optimization . . . 93
8.5 Case Study: MonaCo . . . 97
8.6 Case Study: Presburger Arithmetic . . . 103

9 Conclusion 107
9.1 Results . . . 107
9.2 Future Work . . . 109

Bibliography 113

I was a-trembling, because I'd got to decide,
forever, betwixt two things, and I knowed it. I
studied a minute, sort of holding my breath, and
then says to myself, "All right, then, I'll *go* to hell."

— Mark Twain, *The Adventures of Huckleberry Finn* (1884)

Chapter 1

Introduction

Unlike the fundamental decision problem faced by Huckleberry Finn, regular languages are much closer to heaven. They play a central role in almost all branches of computer science, and yet (or rather because of) most of the interesting questions asked about them can be answered by a machine.

The main question under consideration in this thesis is *language equivalence*: given two concrete symbolic representations of regular languages, decide whether they denote the same language. Here, symbolic refers to a piece of syntax defined by an abstract datatype—more concretely to regular expressions and logical formulas. It explicitly does not refer to finite automata.

Traditionally, decision procedures for equivalence of such symbolic representations rush to dispose of the syntactic structure by compiling it to deterministic finite automata, which then are efficiently minimized and checked for structural equality. Here, we explore an alternative path that avoids this explicit compilation step.

The decision procedures presented here are specified and proved correct in the Isabelle proof assistant. Thus, there is virtually no doubt in their correctness with respect to a formal specification. The specification itself, however, needs to be inspected manually. Using Isabelle's code generator, we extract verified algorithms from the formalization in conventional functional programming languages.

1.1 Plan of Attack

Deciding equivalence of regular languages is a rather uncontroversial goal to pursue. The choice of a suitable representation for regular languages is more subtle. As with programming languages there is a spectrum of imperative and declarative alternatives. Various kinds of finite automata are situated at the imperative end of this spectrum. Here, we concentrate on the other end, which is populated by regular expressions and formulas of weak monadic second-order logic (WMSO).

Our mantra is not only to take the declarative symbolic representations as input, but also to stick to them in the course of the decision procedures. The key ingredient to achieve this for different representations are variations of Brzozowski derivatives of regular expressions [25], which symbolically compute the regular expression denoting what is left after reading one character and thus replace the explicit global transition function of automata.

The main motivation for working with symbolic representations is simplicity. Regular expressions and formulas are free datatypes equipped with induction and recursion principles and suitable for equational reasoning—the core competence of proof assistants and functional programming languages. In contrast, automata are arbitrary graphs and therefore not as easy to reason about in a structural fashion.

Symbolic decision procedures for regular expression equivalence based on variations of Brzozowski derivatives have been formalized before [5,24,35,76,85]. In this thesis, we unify the different existing approaches in a common framework that essentially constructs a syntactic bisimulation abstracting over concrete derivatives. Based on a few properties of abstract derivatives we prove total correctness and completeness of the bisimulation construction once and for all within the framework. All decision procedures presented in this thesis are then obtained by instantiating the framework with different derivative operations.

In his seminal work [26], Büchi envisioned WMSO to become a "more conventional formalism [that] can be used in place of regular expressions [...] for formalizing conditions on the behavior of automata". This vision has materialized—WMSO has been used to encode decision problems in hardware verification [11], program verification [63], network verification [12], synthesis [55], etc.

Equivalence of WMSO formulas is decidable via the famous logic-automaton connection [26, 39, 107], although the decision procedure's complexity is non-elementary [83]. Nevertheless, the MONA tool [58] shows that the daunting theoretical complexity can often be overcome in practice by employing a multitude of smart optimizations. Similarly to Büchi, MONA's manual [69] calls WMSO a "simple and natural notation" for regular languages.

Traditional decision procedures for WMSO do not try to benefit from the conventional, simple, and natural logical notation. Instead, by exploiting the logic-automaton connection, formulas are translated into finite automata, which are then minimized. During the translation all the rich algebraic formula structure including binders and high-level constructs is lost. On the other hand, the subsequent minimization might have benefited from some simplifications at the formula level.

We develop decision procedures for WMSO that are not employing automata and formalize them using the Isabelle proof assistant.

The first decision procedure reduces equivalence of WMSO formulas to equivalence of regular expressions. For a straightforward embedding of WMSO formulas

into regular expressions, an extension of regular expressions with negation and intersection as well as with a new projection operation is required. We develop an equivalence checker for regular expressions extended in that way by extending the derivative operation accordingly. We also define a language-preserving translation of formulas into regular expressions that together with the mentioned equivalence checker yields a decision procedure for WMSO.

Next, inspired by Brzozowski derivatives, we devise a notion of derivatives operating directly on formulas. Using formula derivatives, we obtain a second decision procedure for WMSO that does not require a translation of formulas into automata.

All of the above procedures for WMSO come in two flavors mirroring two existing semantics of WMSO: WS1S and M2L(Str).

Methodologically, the above procedures (as well as the generic framework) are based on the coalgebraic view on languages due to Rutten [99]. This theory is interesting in itself and formalized separately. We turn this formalization into a tutorial on corecursion and coinduction in Isabelle/HOL.

The obtained decision procedures are sound but not very efficient. Combining state-of-the-art optimizations, such as BDDs, hash consing, memoization, with formula derivatives is the basis of the more competitive (unverified) MonaCo tool for deciding WMSO, developed in an ongoing joint effort with Damien Pous.

Finally, we want to emphasize the value of formalization for our work. Besides the usual arguments about guaranteed correctness, working out all details, and eventually correcting mistakes that the literature is full of, we argue that the formalization helped us to clarify the landscape of various decision procedures. This happened nearly automatically in the course of formalization, because the latter is hard and forced us to take the simplest possible path, which often turns out to be the most direct one. In the end, the clarified situation has fostered new ideas such as derivatives of formulas.

1.2 Related Work

Our work stands on the shoulders of giants, which we introduce in several groups.

Language Theory Some classics of language theory are indispensable for our work. At dawn of language theory, Kleene discovered regular expressions [71] and proved Kleene's famous theorem: regular expressions are as expressive as finite automata. Brzozowski introduced derivatives of regular expressions [25] a decade later. This monumental work has the largest impact on this thesis of all given references. Brzozowski's main achievement was, besides the introduction of the actual operation, to prove that there are only finitely many derivatives modulo associativity, commutativity, and idempotence (ACI) of the union operator.

While traditional decision procedures for the equivalence of regular expressions were translating the latter into automata [50, 82], Ginzburg [49] used Brzozowski's derivatives of regular expressions to decide their equivalence. His procedure is the blueprint for our generic framework. Conway's monograph [34], besides being an entertaining read, also emphasizes the importance of the derivatives which he calls "a skill worth acquiring".

For regular expressions, different variations of Brzozowski derivatives have been proposed more recently. Most prominently, the notion of partial derivatives has been introduced by Antimirov [2, 3] and generalized by Caron et al. [29] to support complements and intersections. Partial derivatives capture ACI equivalence directly in the data structure of sets. Often partial derivatives are viewed as the non-deterministic counterpart of deterministic Brzozowski derivatives.

Stockmeyer and Meyer proved the PSPACE completeness of the equivalence problem for regular expressions [105].

Logic Even before Brzozowski, Büchi [26] proposed an alternative to regular expressions for specifying regular properties. Since he was a logician, it is not surprising that his language of choice was a logic: the weak monadic second-order logic of one successor (WS1S). The logic-automaton connection—the counterpart of Kleene's theorem for this logic—was discovered independently by Büchi, Elgot, and Trakhtenbrot [26, 39, 107]. Meyer [83] extended earlier results [105] to show that equivalence of WS1S formulas is of non-elementary complexity.

The daunting theoretical complexity has been faced by developers of the Mosel [65] and MONA [38, 58, 70] tools. The latter is the state of the art for WS1S today and shows that in practice reasonable things can be achieved despite the theoretical intractablity [11, 12, 55, 63].

More recently, some attempts to improve on the state of the art have been initiated. Fiedor et al. [41] employ the modern antichain technique for nondeterministic automata in the context of deciding WS1S. Their tool, dWiNA, outperforms MONA for certain classes of formulas (in particular for formulas in prefix normal form).

Ganzow and Kaiser [45, 46] present a very general, model theoretic decision procedure for weak monadic second-order logic on inductive structures that is inspired by Shelah's composition method [102]. Although they tackle the problem from a completely different angle, their computation of the next-state function is also performed symbolically on formulas and therefore has a similar flavor as our derivatives of formulas. However, after obtaining the next-state function in such a way, Ganzow and Kaiser escape the formula world by translating the problem into a finite reachability game. Their algorithm is implemented in the Toss tool. It outperforms MONA for certain formulas, by outsourcing parts of the computation to state-of-the-art SAT-solvers. Overall, the work by Ganzow and Kaiser can serve as a starting point on how to generalize our procedure beyond WS1S.

Coalgebra Given its young age the theory of coalgebra [9,62,100] is an extremely influential field. Rutten [99] has outlined the coalgebraic view on formal languages. Derivatives play a central role in this view. A little later, Jacobs coined the bialgebraic view on regular languages [61] where the syntax of regular expressions is considered from the inductive algebraic perspective and their semantics from the coinductive coalgebraic perspective.

This bialgebraic view caters for an elegant proof of Kleene's theorem for regular languages [61,99] and its generalization to other coalgebras [22,103]. Another highlight of the coalgebraic approach is an elegant exposition of Brzozowski's algorithm for minimization [19,20] A recent overview of landmark results gathered under the unifying umbrella of coalgebra is given by Silva [104].

Programing languages Recently, derivatives of regular expressions have received a noticeable increase of interest in the programming language community (especially functional programming). Owens et al. [92] were first to implement an efficient scanner generator for Standard ML based on Brzozowski derivatives, which according to them were "lost in the sands of time". Fischer et al. [43] use a variation of derivatives for matching to outperform a Google library.

Beyond regular expressions, generalizations of Brzozowski derivatives were developed for context-free grammars [36, 84] and Kleene algebra with tests (KAT) [73]. In the latter case, derivatives give rise to efficient coalgebraic decision procedures for KAT [96] and its network-targeted specialization NetKAT [44]. Our work can be seen in line with this work, albeit replacing "efficient" with "verified". The ultimate goal is to develop procedures that fulfill both predicates.

Interactive theorem proving Also the interactive theorem proving community has started to realize the elegance of derivatives only recently. This resulted in a series of formalized decision procedures for regular expressions equivalence.

The first verified equivalence checker for regular expressions was published by Braibant and Pous [24]. They worked with automata, not regular expressions, their theory was large and their algorithm efficient. In response, Krauss and Nipkow [76] gave a much simpler partial correctness proof for an equivalence checker for regular expressions based on derivatives. Coquand and Siles [35] showed total correctness of their equivalence checker for extended regular expressions based on derivatives. Asperti [5] presented an equivalence checker for regular expressions via marked regular expressions (as previously used by Fischer et al. [43]) and showed total correctness. Moreira et al. [85] presented an equivalence checker for regular expressions based on partial derivatives and showed its total correctness.

Outside of the application area of equivalence checking, Wu et al. [118, 119] benefited from the inductive structure of regular expressions and Antimirov's partial derivatives to formally verify the Myhill-Nerode theorem.

Beyond regular expressions, Berghofer and Reiter [13] formalized a decision procedure for Presburger arithmetic via automata in Isabelle/HOL.

MONA was linked as an external prover to Isabelle by Basin and Friedrich [10] and to PVS by Owre and Rueß [93]. In both cases, MONA is used as a trusted oracle for deciding formulas in the respective theorem prover.

1.3 Contributions and Structure

The main contribution of this thesis is the formalization of a uniform framework for deciding language equivalence based on the coalgebraic view on regular languages and the instantiation of this framework by various syntactic representations including regular expressions, WMSO formulas under different semantics, and formulas of Presburger arithmetics.

For regular expressions we reuse the wealth of syntactic representations that have been proposed in the literature. Our contribution here lies merely in their unification and formalization. The situation is different with formulas where both—the translation to Π-extended regular expressions (alongside the introduction of the latter) as well as the notion of formula derivatives—are new contributions.

In more detail, each of the following chapters except for preliminaries (Chapter 2) and conclusion (Chapter 9) offers one central contribution (often beyond other less important ones that are named in the respective chapter). The central contributions are listed below.

Chapter 3 Formalization of the coalgebraic view on formal languages

Chapter 4 Development (and formalization) of a generic framework for deciding language equivalence

Chapter 5 Instantiation of the framework with various existing derivative-like operations on regular expressions

Chapter 6 Introduction (and formalization) of Π-extended regular expressions and their derivatives

Chapter 7 Introduction (and formalization) of a semantics preserving translation of WMSO formulas Π-extended regular expressions

Chapter 8 Introduction (and formalization) of a formula derivatives for WMSO and Presburger arithmetic

The chapters are intended to be read in order. Chapter 3 forms an exception to this rule. Although it is helpful for understanding the coalgebraic setting, the following chapters do not rely on it formally. The two semantics of WMSO are introduced in Chapter 7. Moreover, Chapter 8 contains a case study on the ongoing development of the unverified but efficient MonaCo tool that is based on formula derivatives.

1.4 Publications

This thesis is built upon the following three refereed conference publications and one journal article ordered chronologically by the date of completion.

[113] D. Traytel and T. Nipkow. Verified decision procedures for MSO on words based on derivatives of regular expressions. In G. Morrisett and T. Uustalu, editors, *ICFP 2013*, pages 3–12. ACM, 2013.

[89] T. Nipkow and D. Traytel. Unified decision procedures for regular expression equivalence. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 450–466. Springer, 2014.

[115] D. Traytel and T. Nipkow. Verified decision procedures for MSO on words based on derivatives of regular expressions. *J. Funct. Program.*, 25, 2015. Extended version of [113].

[111] D. Traytel. A coalgebraic decision procedure for WS1S. In S. Kreutzer, editor, *CSL 2015*, volume 41 of *LIPIcs*, pages 487–503. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2015.

We reuse material from these publications with Tobias Nipkow's permission. The publications are associated to the individual chapters as follows: Chapter 5 is based on the ITP publication, Chapters 6 and 7 on the ICFP publication and its extended *J. Funct. Program.* version, and Chapter 8 on the CSL publication. The framework from Chapter 4 is not published yet at the level of generality as it is presented here. It constitutes the (so far) last step in a series of generalizations starting from a concrete decision procedure towards more and more abstractions.

Moreover, we describe formalizations that withstood the test of the most merciless and pedantic reviewer— the Isabelle proof assistant. The associated artifacts (formal proof developments) are published in the *Archive of Formal Proofs* (*AFP*).

[110] D. Traytel. A codatatype of formal languages. In *Archive of Formal Proofs*. 2013. `http://afp.sf.net/entries/Coinductive_Languages.shtml`.

[90] T. Nipkow and D. Traytel. Unified decision procedures for regular expression equivalence. In *Archive of Formal Proofs*. 2014. `http://afp.sf.net/entries/Regex_Equivalence.shtml`.

[114] D. Traytel and T. Nipkow. Decision procedures for MSO on words based on derivatives of regular expressions. In *Archive of Formal Proofs*. 2014. `http://afp.sf.net/entries/MSO_Regex_Equivalence.shtml`.

[112] D. Traytel. Derivatives of logical formulas. In *Archive of Formal Proofs*. 2015. `http://afp.sf.net/entries/Formula_Derivatives.shtml`.

The chronological order of those *AFP* entries (in which they are given) corresponds closely to the chapter order in this thesis. The formalizations still deviate slightly from the presentation here, mainly for historical reasons. The last version of the generic framework was introduced only in the last *AFP* entry [112] and we have not upgraded earlier formalizations to use the full generality yet. Thus the thesis presents an idealistic view on how things should have been formalized: uniformly, starting from the general framework. The actual formalizations are less uniform.

Finally, the following research, which was conducted in parallel to the material presented here and is mostly about the implementation and usage of the new modular infrastructure for (co)datatypes in Isabelle/HOL, is beyond the scope of this thesis. Some of these refereed publications do form relevant related work (especially for Chapter 3) and are cited there accordingly.

D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *LICS 2012*, pages 596–605. IEEE, 2012.

J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.

J. C. Blanchette, A. Popescu, and D. Traytel. Cardinals in Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 111–127. Springer, 2014.

J. C. Blanchette, A. Popescu, and D. Traytel. Unified classical logic completeness—A coinductive pearl. In S. Demri, D. Kapur, and C. Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNCS*, pages 46–60. Springer, 2014.

J. C. Blanchette, L. Hupel, T. Nipkow, L. Noschinski, and D. Traytel. Experience report: The next 1100 Haskell programmers. In W. Swiestra, editor, *Haskell 2014*, pages 25–30. ACM, 2014.

J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In J. Vitek, editor, *ESOP 2015*, volume 9032 of *LNCS*, pages 359–382. Springer, 2015.

J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion. In J. Reppy, editor, *ICFP 2015*, pages 192–204. ACM, 2015.

J. Hölzl, A. Lochbihler, and D. Traytel. A formalized hierarchy of probabilistic system types—Proof pearl. In X. Zhang and C. Urban, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 203–220. Springer, 2015.

Yes yes yes! A machine proof is much easier if you already understand the intuition behind the traditional proof. It is a myth that machine assistance can replace the need for competence and understanding. The machine magnifies competence, but it also magnifies incompetence...

— Lawrence C. Paulson (2007)

Chapter 2

Isabelle/HOL and Regular Languages

Isabelle is a generic proof assistant. She follows the *LCF* tradition [51] of having a (relatively) small trusted kernel. Every theorem that is accepted by Isabelle is a consequence of the few primitive inferences belonging to the kernel. To make for a usable proving environment, Isabelle offers a wide variety of complex *definitional* facilities that reduce high-level user specifications to the kernel's primitives without introducing new axioms and advanced automatic proof tools (*methods*) that internally compose primitive inferences. Thereby, the LCF architecture constitutes a rather strong safeguard against inconsistencies.

Being generic means for Isabelle to support reasoning based on different logical foundations—the *object logics*. The most widely used and developed object logic is higher-order logic (HOL)— also our object logic of choice. HOL is based on Church's simple type theory [31]. It is the logic of Gordon's original HOL system [52] and of its many successors such as HOL4 [52], HOL Light [56], HOL Zero [1], and ProofPower–HOL [4]. Isabelle extends the classic HOL with overloading and type classes in a consistent fashion [79, 80]; this flavor is called Isabelle/HOL.

The Isabelle/HOL tutorial [88] introduces higher-order logic by the equation "HOL = Functional Programming + Logic". This minimalistic description fits very well to what we are planning to do in this thesis: programming (functional) decision procedures and proving their correctness. Our presentation of Isabelle/HOL will therefore abstract over for our purposes non-essential details, such as the separation into the object logic HOL and Isabelle's meta-logic Pure, foundational questions (what are the axioms?), Hilbert choice, and even the structural proof language Isar.[1] For a more comprehensive introduction to Isabelle/HOL we refer to the tutorial [88] and a more recent textbook [87, Part I].

[1] Only Chapter 3 will contain some Isar proofs, which by design [117] should be readable without any further explanation also for a non-expert.

2.1 Programming and Proving in Isabelle/HOL

In Isabelle/HOL types τ are built from type variables α, β, etc. via type constructors $\alpha\ \kappa$ written postfix. Types represent sets of values. Some special types are the sum type $\alpha + \beta$, the product type $\alpha \times \beta$, and the function type $\alpha \to \beta$, for which the type constructors are written infix. Infix operators bind less tightly than the postfix or prefix ones; we omit parentheses where possible. Other important types are the type of Booleans *bool* inhabited by exactly two different values $\top$ (truth) and $\bot$ (falsity) and the type $\alpha\ set$ of sets of elements of type α. Note that $\alpha\ set$ is isomorphic to $\alpha \to bool$ but those are two separate types in Isabelle/HOL.

The notation $t :: \tau$ means that term t has type τ. A term t is either a variable x, y, etc. or a constant c, d, or a lambda abstraction $\lambda x.\ t$, or an application $t\ t'$. As usual, application associates to the left: $f\ x\ y$ denotes $(f\ x)\ y$. The typing rules for terms are standard.

For Boolean connectives and sets common mathematical notation is used. Some less standard vocabulary follows: $\mathsf{fst} :: \alpha \times \beta \to \alpha$ and $\mathsf{snd} :: \alpha \times \beta \to \beta$ are the projections of the product type, $\mathsf{id} = \lambda x.\ x$ is the identity function, the predicate $\mathsf{finite} :: \alpha\ set \to bool$ asserts that a set contains only finitely many elements, and the function $\bullet$ (written infix) computes the image of a function over a set: $f \bullet X = \{f\ x \mid x \in X\}$. All types in HOL have to be non-empty. The set of all values of a type τ is denoted by $\mathsf{UNIV} :: \tau\ set$. A special constant is equality $= :: \alpha \to \alpha \to bool$, which is polymorphic (it exists for any type, including the function type, on which it is extensional).

Theorems are just terms of type *bool*, which were proven to be equivalent to $\top$ via the primitive kernel inferences, for example by backward reasoning via the barebones *rule* method that relies on higher-order unification [88], or more automatic methods like the rewriting engine (*simp*) or the powerful mixture of rewriting and classical reasoning (*auto*).

There are two primitives for introducing new types and constants: `typedef` and `definition`. With the `typedef` command, a new type can be defined by carving out a non-empty subset A (not referring to the new type) from an existing type. For example we can define the singleton type *unit* by carving out the singleton set containing $\top$ from *bool*.

$$\texttt{typedef}\ unit = \{\top\}$$

This command requires the user to prove $\{\top\} \neq \{\}$ and thereafter introduces the new type *unit* together with two bijections $\mathsf{Rep}_{unit} :: unit \to bool$ and $\mathsf{Abs}_{unit} :: bool \to unit$ between $\mathsf{UNIV} :: unit\ set$ and $\{\top\}$. The bijectivity is captured by a few axioms. The only inhabitant of *unit* is defined using the primitive for constant definitions.

$$\begin{array}{l}\texttt{definition}\ \mathsf{u} :: unit\ \texttt{where} \\ \quad \mathsf{u} = \mathsf{Abs}_{unit}\ \top\end{array}$$

Equations accepted by `definition` are introduced as axioms and thus restricted to a very simple format $\mathsf{f}\ x_1\ \ldots\ x_n = \mathsf{t}\ x_1\ \ldots\ x_n$, where f does not occur in t. With other words `definitions` may not be recursive.

Lifting constants to newly defined types and proving properties about those is a tedious labor due to the bijections that need to be inserted in the right places and eventually cancel out while proving. The Lifting and Transfer [60] tools automate this tedium making `typedefs` (quasi)transparent. In our formalization we use them extensively, however for the presentation here we keep silent about certain auxiliary `typedefs` and therefore will not need to introduce the syntax of those tools.

What we will need more often in the presentation are inductive types and recursive functions. Isabelle offers a `datatype` command that reduces an inductive type specification (in the spirit of a Standard ML `datatype`) to a primitive non-recursive `typedef` [16]. For example we can define the types of natural numbers and lists of elements of type α in a standard way.

$$\texttt{datatype}\ nat = 0 \mid nat + 1$$

$$\texttt{datatype}\ \alpha\, list = [] \mid (\mathsf{hd} : \alpha)\ \#\ (\mathsf{tl} : \alpha\, list)$$

Note that we may use (almost) arbitrary syntax for our constructors. Here $_ + 1$ is a unary constructor of natural numbers (often called Suc) and # is the infix list constructor. Moreover, the `datatype` command allows the user to annotate the arguments of a constructor with names for *selectors*. The list declaration thus defines $\mathsf{hd} :: \alpha\, list \to \alpha$ and $\mathsf{tl} :: \alpha\, list \to \alpha\, list$ to return a list's head and tail, respectively. We also obtain the associated characteristic equations such as $\mathsf{hd}\ (x\ \#\ xs) = x$. Note that, since $\mathsf{hd}\ []$ is not specified, there is nothing that can be proved about it (except that it constitutes a term of type α given that $[]$ is of type $\alpha\, list$).

After defining the type, the `datatype` command also derives a large amount of useful theorems for reasoning about the types including an induction rule. For natural numbers and lists the rules are the familiar ones (written as inference rules).

$$\frac{P\ 0 \quad \forall n.\ P\ n \longrightarrow P\ (n+1)}{\forall n.\ P\ n} \qquad \frac{P\ [] \quad \forall x\ xs.\ P\ xs \longrightarrow P\ (x\ \#\ xs)}{\forall xs.\ P\ xs}$$

Moreover, the `datatype` command also sets up the infrastructure for defining recursive functions. The definitions are performed by separate commands: `primrec` [14] for primitive recursion and `fun` [77] for general wellfounded recursion. We demonstrate `primrec`'s syntax that we use in this thesis (which is quite faithful to the actual Isabelle syntax) on the example of the append operation (written infix @). To use `fun`, no syntactic change is required, except for the command's name.

$\texttt{primrec}\ @ :: \alpha\,list \rightarrow \alpha\,list \rightarrow \alpha\,list\ \texttt{where}$
$\quad [\,]\ @\ ys = ys$
$\quad (x\ \#\ xs)\ @\ ys = x\ \#\ (xs\ @\ ys)$

Beside constructors, selectors, and @, lists come with the following vocabulary: functions set :: $\alpha\,list \rightarrow \alpha\,set$ (converting from lists to sets), as_list :: $\alpha\,set \rightarrow \alpha\,list$ (converting a finite set with a linear order attached to its elements to a list), map :: $(\alpha \rightarrow \beta) \rightarrow \alpha\,list \rightarrow \beta\,list$ (applying a function to every element in the list), rev :: $\alpha\,list \rightarrow \alpha\,list$ (reversing a list), and fold :: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \alpha\,list \rightarrow \beta \rightarrow \beta$ (the primitive recursion combinator) are all defined by primitive recursion. The length of a list xs is abbreviated by $|xs|$; the nth element of xs is accessed via $xs[n]$ (zero based and assuming that $n < |xs|$); the predicate distinct :: $\alpha\,list \rightarrow bool$ yields $\top$ iff its argument list does not contain any element more than once.

Besides inductive types, Isabelle has recently been extended to also support coinductive types via the `codatatype` command [14]. A prominent example of codatatypes are lazy lists— the coinductive counterparts of lists. We define them using the same syntax (including selectors) as for datatypes (except for the command's name).

$\texttt{codatatype}\ \alpha\,llist = [\,]_l \mid (\mathsf{lhd} : \alpha)\ \#_l\ (\mathsf{ltl} : \alpha\,llist)$

Codatatypes permit values of the type to be constructed from a finite or infinite number of constructors (much like Haskell's data due to lazy evaluation), while a value of a datatype is always finite. The above command provides many conveniences for reasoning about codatatypes, including a coinduction principle and a setup for primitively corecursive definitions. We explain the conveniences in more detail in Chapter 3.

A large fragment of HOL, including recursive functions over datatypes, is executable. That is Isabelle can generate code in functional languages from a given specification [54]. This includes functions on (finite) sets [53]. All the decision procedures presented in this thesis are carefully designed to be executable.

2.2 Locales

Locales [8] are Isabelle's tool for modelling parameterized systems. Recently, locales have been used to model institution theoretical reasoning about translations of logics [15]. We prefer to stick to a more standard presentation and usage. A locale fixes parameters and states assumptions about them.

$\texttt{locale}\ A =$
$\quad \texttt{fixes}\ x_1\ \texttt{and} \cdots \texttt{and}\ x_n$
$\quad \texttt{assumes}\ n_1 : P_1\ \overline{x}\ \texttt{and} \cdots \texttt{and}\ n_m : P_m\ \overline{x}$

In the context of the locale A, we can define constants that depend on the parameters x_i and prove properties about those constants using the assumptions P_i (accessed under the name n_i). To place a `definition`, `primrec`, or `theorem` into the context of a locale we annotate the command with a target modifier, for example `definition` (in A) $x_{1n} = (x_1, x_n)$. The parameters of a locale can be instantiated later via the `interpretation` command.

$$\texttt{interpretation}\ \mathsf{J} : A\ \texttt{where}\\ x_1 = t_1 \ldots x_n = t_n$$

The command issues proof obligations $P_i\ \bar{t}$ (that the user must discharge) and exports all constants (and theorems) defined (proved) in the context of the locale with x_i instantiated to t_i. Note that t_i may contain free variables which after instantiation become formal dependencies of the exported definitions and theorems. Multiple interpretations of the same locale are possible; the name prefix "J" disambiguates different instances. It is also possible to interpret a locale B while being inside of another locale A. This is usually modeled with the `sublocale` command.

$$\texttt{sublocale}\ A \leq B$$

Similarly to an interpretation, the `sublocale` command requires the user to discharge B's assumptions, assuming that A's assumption hold. Afterwards all definitions and theorems of B also become accessible in A. Moreover, as before an additional `where` clause may preinstantiate B's parameters with some terms that depend on A's parameters.

2.3 Partiality

Isabelle/HOL is a logic of total functions. Sometimes, we would still like to reason about functions, that may not terminate, but whose behavior in the non-terminating case can be uniquely specified. An example where this situation arises are tail recursive functions. Here it is sound to assign the function in the case of non-termination a single arbitrary but fixed value from the function's result type (note that all types are non-empty, hence such a value exists).

Tail recursive functions can be transformed into while loops, which are typically modeled in Isabelle via the while combinator.

$$\texttt{definition}\ \mathsf{while} :: (\alpha \to bool) \to (\alpha \to \alpha) \to \alpha \to \alpha\ option\ \texttt{where}\\ \mathsf{while}\ b\ c\ s\ =\ \texttt{if}\ \exists k.\ \neg b\ (c^k\ s)\ \texttt{then}\ \mathsf{Some}\ (c^{\mathsf{Least}\ k.\ \neg b\ (c^k s)}\ s)\ \texttt{else}\ \mathsf{None}$$

Here, $c^k s$ denotes the k-fold application $c\ (\cdots c\ (c\ s)\cdots)$ and $\mathsf{Least}\ k.\ P\ k$ specifies the smallest natural number m for which $P\ m$ holds. The option type $\alpha\ option$ has

two constructors None :: $\alpha\,option$ and Some :: $\alpha \to \alpha\,option$. Some lifts elements from the base type α to the option type, while None is usually used to indicate some exceptional (here non-terminating) behavior.

The above total specification of while is not yet executable. Supplying the following key theorem as a code equations to the code generator makes while executable.

```
theorem while b c s = if b s then while b c (c s) else Some s
```

The code generated from this tail recursive equation will return Some s in case the definition of while says so, but instead of returning None, it will not terminate. Thus, we can later prove termination if we can show that the result is ≠ None.

An alternative to the while-combinator would be to use the `partial_function` command [75], which automates definitions of tail recursive and monadic functions (for example on the $\alpha\,option$ monad). For tail-recursive functions defined with `partial_function`, stating the termination property is less convenient, which was the main reason for us to prefer the more low-level while-combinator.

2.4 Regular Expressions

The standard way to represent regular languages is to view them as a set of words (which in turn are represented by lists). While in the next chapter we explore an alternative representation, we briefly recapitulate here what Isabelle has to offer when using the standard representation.

Regular expressions are defined as a recursive datatype.

```
datatype α RE = ∅ | ε | A α | α RE + α RE | α RE · α RE | (α RE)*
```

They are assigned the usual (non-executable) semantics L by primitive recursion.

```
primrec L :: α RE → (α list) set where
  L ∅ = {}
  L ε = {[]}
  L (A a) = {[a]}
  L (r + s) = L r ∪ L s
  L (r · s) = L r · L s
  L (r*) = ⋃_{i::nat} (L r)^i
```

Here, $A \cdot B$ abbreviates the set $\{u \mathbin{@} v \mid u \in A,\ v \in B\}$ and A^i is defined recursively via $A^0 = \{[]\}$ and $A^{n+1} = A \cdot A^n$. In concrete regular expressions, we sometimes omit the constructor A for readability.

Next, we define two executable functions, which will accompany us throughout the thesis: the empty word acceptance test o and Brzozowski derivatives δ.

```
primrec o :: α RE → bool where
  o ∅ = ⊥
  o ε = ⊤
  o a = ⊥
  o (r + s) = o r ∨ o s
  o (r · s) = o r ∧ o s
  o (r*) = ⊤
```

```
primrec δ :: α → α RE → α RE where
  δ a ∅ = ∅
  δ a ε = ∅
  δ a b = if a = b then ε else ∅
  δ a (r + s) = δ a r + δ a s
  δ a (r · s) = δ a r · s + if o r then δ a s else ∅
  δ a (r*) = δ a r · r*
```

The empty word acceptance test o satisfies $o\ r \leftrightarrow [] \in \mathsf{L}\ r$. The Brzozowski derivative δ computes the regular expression denoting the *left quotient* of the language of the input expression with respect to some $a :: \alpha$; formally it is characterized by $\mathsf{L}\ (\delta\ a\ r) = \{v \mid a \# v \in \mathsf{L}\ r\}$. The extension of δ from single symbols to words $w :: \alpha\ list$ can be expressed as $\mathsf{fold}\ \delta\ w$; accordingly we have $\mathsf{L}\ (\mathsf{fold}\ \delta\ w\ r) = \{v \mid w\ @\ v \in \mathsf{L}\ r\}$.

Finally, we remark that in the course of the thesis we will extensively use overloading of the notation. For example, we will extend the type $\alpha\ RE$ with additional constructors and see many variations of o and δ in the thesis. Almost always those variations will be still called $\alpha\ RE$, o, and δ. Which variation exactly is meant will be clear from the context (usually the most recently introduced one).

In this case, as in many others, the process gives the minimal machine directly to anyone skilled in input differentiation. The skill is worth acquiring.

— John H. Conway (1971)

Chapter 3

Formal Languages, Coinductively

If one asks a computer scientist what a formal language is, the answer will most certainly be: a set of words. Here, we advocate another valid answer: an infinite trie. This is the coalgebraic view on languages introduced by Rutten [99], from whom Conway's very fitting (as we will see) quote is shamelessly stolen.

This chapter shows how to formalize the coalgebraic view on formal languages in Isabelle/HOL in the form of a tutorial on corecursion and coinduction. After introducing infinite tries as a codatatype (Section 3.1), we define regular operations on them by corecursion (Section 3.2) and prove that the defined regular operations satisfy the axioms of Kleene algebra by coinduction (Section 3.3). The target audience of the tutorial is the working formalizer, that is, somebody assumed not to be familiar with the theory of coalgebra. Thus, coalgebraic terminology will be used sparingly; indeed it will only be introduced at the very end (Section 3.4). The tutorial will make abundant use of notation overloading. We will use exactly the same notation that we have introduced for regular expressions for the corresponding notions on infinite tries. However, there will be not much usage of regular expressions here, such that no confusion should arise. The material presented in this chapter is based on the formalization constituting an AFP entry [110].

The literature is abound with tutorials on coinduction. So why bother writing yet another one? First, because we finally can do it in Isabelle/HOL using convenient high-level tools [16, 18, 116], without going through tedious manual constructions [94]. Second, coinductive techniques are still not as widespread as they could be (and we believe should be, since they constitute a convenient proof tool for questions about semantics). Third, many tutorials [30, 48, 59, 62, 74], with or without a theorem prover, exercise streams to that extent that one starts to believe having seen everything about streams. Even if this is not true, Rutten [99] demonstrates that it is entirely feasible to start with a structure slightly more complicated than streams, but familiar to everybody with a computer science degree. Finally, the decision procedures presented later in this thesis are based on the coalgebraic view. Therefore, the tutorial will gently set us in the right state of mind.

3.1 Languages as Infinite Tries

We define the type of formal languages as a codataype of infinite tries, that is, (prefix) trees of infinite depth branching over the alphabet. We represent the alphabet by the type α. Each node in a trie carries a Boolean label, which indicates whether the (finite) path to this node constitutes a word inside or outside of the language. The function type models branching: for each letter $x :: \alpha$ there is an x-subtree.

$$\texttt{codatatype}\ \alpha\, lang = \mathsf{L}\ (o : bool)\ (\delta : \alpha \to \alpha\, lang)$$

The `codatatype` command defines the type $\alpha\, lang$ together with a *constructor* $\mathsf{L} :: bool \to (\alpha \to \alpha\, lang) \to \alpha\, lang$ and two *selectors* $o :: \alpha\, lang \to bool$ and $\delta :: \alpha\, lang \to \alpha \to \alpha\, lang$.[1] For a binary alphabet $\alpha = \{a, b\}$, the trie *even* shown in Figure 3.1 is a typical inhabitant of $\alpha\, lang$. The label of its root is given by $o\ even = \top$ and its subtrees by another trie $odd = \delta\ even\ a = \delta\ even\ b$. Similarly, we have $o\ odd = \bot$ and $even = \delta\ odd\ a = \delta\ odd\ b$. Note that we could have equally written $even = \mathsf{L}\ \top\ (\lambda_.\ odd)$ and $odd = \mathsf{L}\ \bot\ (\lambda_.\ even)$ to obtain the same mutual characterization of *even* and *odd*.

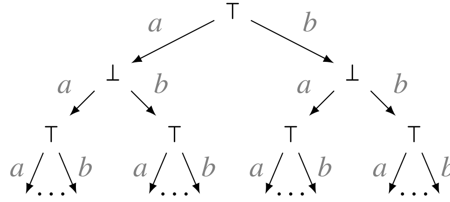


Figure 3.1: Infinite trie *even*

We gave our type the name $\alpha\, lang$, to remind us to think of its inhabitants as formal languages. Nevertheless, we call its inhabitants tries, because that is what they are.

Beyond defining the type and the constants, the `codatatype` command also proves a wealth of properties about them such as $o\ (\mathsf{L}\ b\ d) = b$, the injectivity of L, or more interestingly the coinduction rule, which we will study more carefully later.

Clearly, we would like to identify the trie *even* with the regular language of all words of even length $\{w \in \{a, b\}^* \mid |w| \bmod 2 = 0\}$, also represented by the regular expression $((a + b) \cdot (a + b))^*$. Therefore, we define the notion of word membership $\in$ on tries by primitive recursion on the word as follows.

$$\begin{array}{l}\texttt{primrec}\ \in\ :: \alpha\, list \to \alpha\, lang \to bool\ \texttt{where} \\ \quad [\,] \in L = o\ L \\ \quad (x\ \#\ w) \in L = w \in \delta\ L\ x\end{array}$$

[1] Note that the argument order for this selector differs from other usages of δ in this thesis.

Using $\in$, each trie can be assigned a language in the traditional set of lists view.

```
definition out :: α lang → α list set where
  out L = {w | w ∈ L}
```

For example out $even = \{w \in \{a, b\}^* \mid |w| \bmod 2 = 0\}$. We will see later that $\alpha\ lang$ is isomorphic to the set of lists view on languages by showing that out is a bijection.

3.2 Regular Operations on Tries

So far, we have only specified some concrete infinite tries informally. In Isabelle, following the LCF approach, any corecursive specification must be reduced to a primitive non-recursive definition. The `primcorec` command automates this reduction for the class of primitively corecursive functions [14]. Primitive corecursion is dual to primitive recursion. Primitive recursion destructs one layer of constructors before proceeding recursively. Primitive corecursion constructs (or produces) one constructor whose arguments are allowed to be either non-recursive terms or a corecursive call (applied to arbitrary non-recursive arguments). Beyond this, there is no tool support for corecursive specifications at the point of writing. While sophisticated methods involving domain, measure, and category theory have been proposed [17,81], we explore here how far primitive corecursion can get us.

3.2.1 Primitively Corecursive Operations

We start with some simple examples: the languages of the base cases of regular expressions. Intuitively, the trie $\varnothing$ representing the empty language is labeled with $\bot$ everywhere and the trie ε representing the empty word language is labeled with $\top$ at its root and with $\bot$ everywhere else. The trie A a representing the singleton language of the one letter word a is labeled with $\bot$ everywhere except for the root of its a-subtree. This intuition is easy to capture formally.

```
primcorec ∅ :: α lang where
  ∅ = L ⊥ (λx. ∅)

primcorec ε :: α lang where
  ε = L ⊤ (λx. ∅)

primcorec A :: α → α lang where
  o (A a) = ⊥
  δ (A a) = λx. if a = x then ε else ∅
```

Out of these three definitions only $\varnothing$ is corecursive. (The `primcorec` command also gracefully handles non-recursive specifications.)

The specifications on the left differ syntactically from the one on the right. The constants $\varnothing$ and ε are defined using the so called *constructor view*. It allows the user to enter equations of the form constant or function equals constructor, where the arguments of the constructor are restricted as described above (modulo some further

syntactic conveniences, such as lambda abstractions, `case`-, and `if`-expressions). This kind of definitions should be familiar to any (lazy) functional programmer.

In contrast, the specification of A utilizes the *destructor view*. Here, we specify the constant or function being defined by observations or experiments via *selector equations*. The allowed experiments on a trie are given by its selectors o and δ. We can observe the label at the root and the subtrees. Specifying the observation, again restricted to be either a non-recursive term or a direct corecursive call, for each selector yields a unique characterization. In general, for codatatypes with more than one constructor, the situation is slightly more elaborate [14], but not relevant here.

It is straightforward to rewrite specifications in either of the views into the other one. The `primcorec` command performs this rewriting internally and outputs the theorems corresponding to the user's input specification in all views. The constructor view theorems serve as executable code equations (which make sense in programming languages with lazy evaluation), while the destructor view offers safe simplification rules even when applied eagerly as done by Isabelle's simplifier.

Now that the basic building blocks $\varnothing$, ε, and A are in place, we turn our attention to how to combine them to obtain more complex languages. We start with the simpler combinators for union, intersection, and negation, before moving to the more interesting concatenation and iteration. The union + of two tries should denote set union of languages (i.e. $\mathtt{out}\ (L + K) = \mathtt{out}\ L \cup \mathtt{out}\ K$ should hold). It is defined corecursively by traversing the two tries in parallel and computing for each pair of labels their disjunction. Intersection $\cap$ is the same, but replacing conjunction for disjunction. Negation $\neg$ simply inverts every label in the trie.

$$\begin{array}{l}
\mathtt{primcorec}\ + :: \alpha\ lang \to \alpha\ lang \to \alpha\ lang\ \mathtt{where} \\
\quad o\ (L + K) = o\ L \vee o\ K \\
\quad \delta\ (L + K) = \lambda x.\ \delta\ L\ x + \delta\ K\ x
\end{array}$$

$$\begin{array}{l}
\mathtt{primcorec}\ \cap :: \alpha\ lang \to \alpha\ lang \to \alpha\ lang\ \mathtt{where} \\
\quad o\ (L \cap K) = o\ L \wedge o\ K \\
\quad \delta\ (L \cap K) = \lambda x.\ \delta\ L\ x \cap \delta\ K\ x
\end{array}$$

$$\begin{array}{l}
\mathtt{primcorec}\ \neg :: \alpha\ lang \to \alpha\ lang\ \mathtt{where} \\
\quad o\ (\neg\ L) = \neg\ o\ L \\
\quad \delta\ (\neg\ L) = \lambda x.\ \neg\ \delta\ L\ x
\end{array}$$

The attentive reader will have noticed: we are about to rediscover Brzozowski derivatives and the acceptance test on regular expressions in the destructor view equations for the selectors δ and o. There is an important difference though: while Brzozowski derivatives work with syntactic objects, our tries are semantic objects on which equality denotes language equivalence. For example, we will later prove $\varnothing + r = r$ for tries, while $\varnothing + r \neq r$ holds for regular expressions. The coinductive

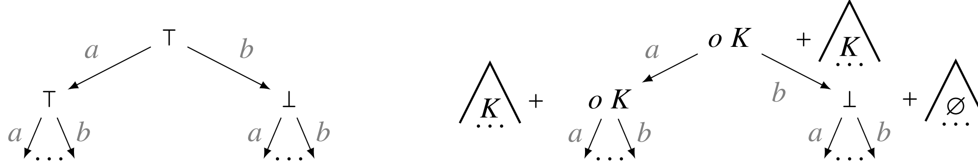


Figure 3.2: Tries for L (left) and the concatenation $L \cdot K$ (right)

view reveals that derivatives and the acceptance test are the two fundamental ingredients that completely characterize regular languages and arise naturally even when only considering the semantics.

3.2.2 Reducing Corecursion Up-To to Primitive Corecursion

Concatenation $\cdot$ is next on the list of regular operations that we want to define on tries. Thinking of Brzozowski derivatives and the acceptance test, we would usually specify it by the following two equations.

$$o\ (L \cdot K) = o\ L \wedge o\ K$$
$$\delta\ (L \cdot K) = \lambda x.\ (\delta\ L\ x \cdot K) + (\texttt{if}\ o\ L\ \texttt{then}\ \delta\ K\ x\ \texttt{else}\ \varnothing)$$

A difficulty arises here, since this specification is not primitively corecursive—the right hand side of the second equation contains a corecursive call but not at the topmost position (which is + here). We call this kind of corecursion *up to* +.

Without tool support for corecursion up-to, concatenation must be defined differently—as a composition of other primitively corecursive operations. Intuitively, we would like to separate the above δ-equation into two along the + and sum them up afterwards. Technically, the situation is more involved. Since the δ-equation is corecursive, we can not just create two tries by primitive corecursion.

Figure 3.2 depicts the trie that should result from concatenating an arbitrary trie K to the concrete given trie L. Procedurally, the concatenation must replace every subtree T of L with label $\top$ at the root (those are positions where words from L end) by the trie $U + K$ where U is the trie obtained from T by changing its root from $\top$ to $o\ K$. For uniformity with the above δ-equation, we imagine subtrees F of L with label $\bot$ at the root as also being replaced by $F + \varnothing$, which, as we will prove later, has the same effect as leaving F alone.

Figure 3.3 presents one way to bypass the restrictions imposed by primitive corecursion. We are not allowed to use + *after* proceeding corecursively, but we may arrange the arguments of + in a broader trie over a doubled alphabet formally modeled by pairing letters of the alphabet with a Boolean flag. In Figure 3.3 we write a for $(a, \top)$ and a' for $(a, \bot)$. Because it defers the summation, we call this primitively corecursive procedure *deferred concatenation* $\hat{\cdot}$.

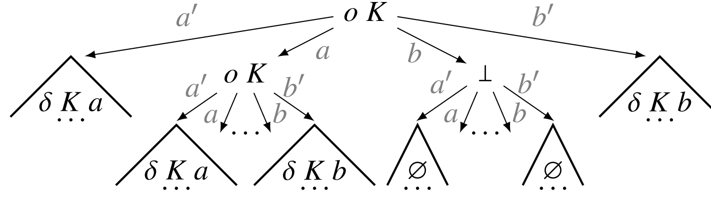


Figure 3.3: Trie for deferred concatenation $L \mathbin{\hat{\cdot}} K$

$$\texttt{primcorec}\ \hat{\cdot} :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang} \to (\alpha \times \mathit{bool})\ \mathit{lang}\ \texttt{where}$$
$$o\ (L \mathbin{\hat{\cdot}} K) = o\ L \wedge o\ K$$
$$\delta\ (L \mathbin{\hat{\cdot}} K) = \lambda(x, b).\ \texttt{if}\ b\ \texttt{then}\ \delta\ L\ x \mathbin{\hat{\cdot}} K\ \texttt{else if}\ o\ L\ \texttt{then}\ \delta\ K\ x \mathbin{\hat{\cdot}} \varepsilon\ \texttt{else}\ \varnothing$$

Note that unlike in the Figure 3.3, where we informally plug the trie $\delta\ K\ x$ as some x'-subtrees, the formal definition must be more careful with the types. More precisely, $\delta\ K\ x$ is of type $\alpha\ \mathit{lang}$, while something of type $(\alpha \times \mathit{bool})\ \mathit{lang}$ is expected. This type mismatch is resolved by further concatenate ε to $\delta\ K\ x$ (again in a deferred fashion) without corrupting the intended semantics.

Once the trie for the deferred concatenation has been build, the desired trie for concatenation can be obtained by a second primitively corecursive traversal that sums the x- and x'-subtrees *before* proceeding corecursively.

$$\texttt{primcorec}\ \hat{\oplus} :: (\alpha \times \mathit{bool})\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where}$$
$$o\ (\hat{\oplus}\ L) = o\ L$$
$$\delta\ (\hat{\oplus}\ L) = \lambda x.\ \hat{\oplus}\ (\delta\ L\ (x, \top) + \delta\ L\ (x, \bot))$$

Finally, we can define the concatenation as the composition of $\hat{\cdot}$ and $\hat{\oplus}$. The earlier standard selector equations for $\cdot$ are provable for this definition.

$$\texttt{definition}\ \cdot :: \alpha\ \mathit{lang} \to \alpha\ \mathit{lang} \to \alpha\ \mathit{lang}\ \texttt{where}$$
$$L \cdot K = \hat{\oplus}\ (L \mathbin{\hat{\cdot}} K)$$

The situation with iteration is similar. The selector equations following the Brzozowski derivative of L^* yield a non-primitively recursive specification. In more detail, it is corecursive up to $\cdot$.

$$o\ (L^*) = \top$$
$$\delta\ (L^*) = \lambda x.\ \delta\ L\ x \cdot L^*$$

As before, the restriction is circumvented by altering the operation slightly. We define the binary operation *deferred iteration* $L \mathbin{\hat{*}} K$, whose language should represent $L \cdot K^*$ (although we have not defined $_^*$ yet). When constructing the subtrees of

$L \mathbin{\hat{*}} K$ we keep pulling copies of the second argument into the first argument before proceeding corecursively (the second argument itself stays unchanged).

primcorec $\hat{*}$ :: α *lang* → α *lang* → α *lang* where
$o\ (L \mathbin{\hat{*}} K) = o\ L$
$\delta\ (L \mathbin{\hat{*}} K) = \lambda x.\ (\delta\ (L \cdot (\varepsilon + K))\ x) \mathbin{\hat{*}} K$

Supplying ε as the first argument to $\hat{*}$ defines iteration that satisfies the original selector equations.

definition $_^*$ :: α *lang* → α *lang* where
$L^* = \varepsilon \mathbin{\hat{*}} L$

We have defined all the standard regular operations on tries! Later we will prove that those definitions satisfy the axioms of Kleene algebra, meaning that the behave as expected. Already now we can compose the operation to define new tries, for example the introductory $\mathit{even} = ((\mathsf{A}\ a + \mathsf{A}\ b) \cdot (\mathsf{A}\ a + \mathsf{A}\ b))^*$.

Let us step back a little bit: why was it possible to reduce the above corecursive up-to equations to primitive corecursion? The formal answer to this question is beyond the scope of this thesis. Still we try to convey a vague intuition here. The reduction was possible because the up-to operations $+$ and $\cdot$ are so called *friendly* operations [17]. An operation is friendly if, under lazy evaluation, it does not peek too deeply into its arguments, before producing at least one constructor.

For example, $L + K = \mathsf{L}\ (o\ L \vee o\ K)\ (\lambda x.\ \delta\ L\ x + \delta\ K\ x)$ destructs only one layer of constructors, in order to produce the topmost L. In contrast the primitively corecursive equation $L! = \mathsf{L}\ (o\ L)\ (\lambda x.\ (\delta\ (\delta\ L\ x)\ x)!)$ destructs two layers of constructors (via δ) before producing one and is therefore not friendly. Indeed, we will not be able to reduce the equation $\mathit{bad} = \mathsf{L}\ \top\ (\lambda_.\ \mathit{bad}!)$ (which is corecursive up-to !) to a primitively corecursive specification. And there is a good reason for it: *bad* is not uniquely specified by the above equation—assigning it a unique value through primitive corecursion would immediately lead to a contradiction.

Friendly operations are formally captured using parametricity [17]. Automation in Isabelle for the large class of friendly functions is underway at the moment of writing. The employed reduction to primitive corecursion follows an abstract, category theory inspired construction. Yet, what it yields in practice is very close to what we have seen on the example of concatenation. (In contrast, the iteration example takes some shortcuts, that the abstract view does not offer.)

Beyond Regular Languages Before we turn to proving, let us exercise one more corecursive definition. Earlier, we have assigned each trie a set of lists via the function out. Primitive corecursion enables us to define the converse operation.

$\texttt{primcorec in} :: \alpha\ \textit{list set} \rightarrow \alpha\ \textit{lang}\ \texttt{where}$
$\quad o\ (\mathsf{in}\ L) = [\,] \in L$
$\quad \delta\ (\mathsf{in}\ L) = \lambda x.\ \mathsf{in}\ \{w \mid x \# w \in L\}$

The function out and in are both bijections. We show this by proving that their compositions (either way) are both the identity. One direction, out (in L) = L, follows by set extensionality and a subsequent induction on words. The reverse direction requires a proof by coinduction, which is the topic of the next section.

Using in we can turn every (even undecidable) set of lists into a trie. This is somewhat counterintuitive, since, given a concrete trie, its word problem seems easily decidable (via the function $\Subset$). But of course in order to compute the trie out a list of sets L via in the word problem for L must be solved—we are reminded that HOL is not a programming language where everything is executable, but a logic in which we write down specifications (not programs).

For the regular operations from the previous section the situation is different. For example, Isabelle's code generator can produce valid Haskell code that evaluates $[a, b, a, a] \Subset (\mathsf{A}\ a \cdot (\mathsf{A}\ a + \mathsf{A}\ b))^*$ to $\top$.

3.3 Proving Equalities on Tries

We have seen the definitions of many operations, justifying their meaningfulness by appealing to the reader's intuition. This is often not enough to guarantee correctness, especially if we have a theorem prover at hand. So let us formally prove in Isabelle that the regular operations on tries form a Kleene algebra by proving Kozen's famous axioms [72] as (in)equalities on tries.

Codatatypes are equipped with the perfect tool for proving equalities: the coinduction principle. Intuitively, the coinduction principle says that the existence of a relation R that is closed under the codatatype's observations (given by selectors) implies that elements related by R are equal. In other words, if we cannot distinguish elements of a codatatype by (finite) observations, they must be equal. Formally, for our codatatype $\alpha\ \textit{lang}$ we obtain the following coinduction rule.

$$\frac{R\ L\ K \qquad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow o\ L_1 \leftrightarrow o\ L_2 \land \forall x.\ R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x)}{L = K}$$

The second assumption of this rule formalizes the notion of being closed under observations. Concretely, if two tries are related then their root's labels are the same and all their subtrees are related. A relation that satisfies this assumption is called a *bisimulation*. Thus, proving an equation by coinduction amounts to exhibiting a bisimulation witness that relates the left and the right hand sides.

Let us observe the coinduction rule, called $coinduct_{lang}$ below, in action. We prove the Kleene algebra axiom that the empty language is the left identity of union via a detailed Isar proof that is accepted by Isabelle.

```
theorem ∅ + L = L
proof (rule coinduct_lang)
    def R L1 L2 = (∃K. L1 = ∅ + K ∧ L2 = K)

    show R (∅ + L) L by simp

    fix L1 L2
    assume R L1 L2
    then obtain K where L1 = ∅ + K and L2 = K by simp
    then show o L1 ↔ o L2 ∧ ∀x. R (δ L1 x) (δ L2 x) by simp
qed
```

The proof has three parts. First, we supply a definition of a witness relation R. Second, we prove that R relates the left and the right hand side. Third, we prove that R is a bisimulation. While the second and the third part is rather automatic, the first one a priori seems to require some ingenuity.

Let us delve into the automatic part first. Proving $R\ (\varnothing + L)\ L$ for our particular definition of R is trivial after instantiating the existentially quantified K with L. Proving the bisimulation property is barely harder: for two tries L_1 and L_2 related by R we need to show $o\ L_1 \leftrightarrow o\ L_2$ and $\forall x.\ R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x)$. Both properties follow by simple calculations rewriting L_1 and L_2 in terms of a trie K, whose existence is guaranteed by $R\ L_1\ L_2$, and simplifying with the selector equations for $+$ and $\varnothing$.

$$o\ L_1 \leftrightarrow o\ (\varnothing + K) \leftrightarrow o\ \varnothing \lor o\ K \leftrightarrow \bot \lor o\ K \leftrightarrow o\ K \leftrightarrow o\ L_2$$

$$\begin{aligned}&R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x) \leftrightarrow R\ (\delta\ (\varnothing + K)\ x)\ (\delta\ K\ x)\\ &\leftrightarrow R\ (\delta\ \varnothing\ x + \delta\ K\ x)\ (\delta\ K\ x) \leftrightarrow R\ (\varnothing + \delta\ K\ x)\ (\delta\ K\ x)\\ &\leftrightarrow \exists K'.\ \varnothing + \delta\ K\ x = \varnothing + K' \land \delta\ K\ x = K' \leftrightarrow \top\end{aligned}$$

The last step is justified by instantiating K' with $\delta\ K\ x$.

How did our definition of R come about? In general, when proving a conditional equation $P\ \overline{x} \longrightarrow l\ \overline{x} = r\ \overline{x}$ by coinduction, where $\overline{x}$ denotes a list of variables occurring freely in the equation, the canonical choice for the bisimulation witness is $R\ a\ b = (\exists \overline{x}.\ a = l\ \overline{x} \land b = r\ \overline{x} \land P\ \overline{x})$. There is no guarantee that this definition yields a bisimulation. However, after performing a few proofs by coinduction, this particular pattern emerges as a natural first choice to try.

Indeed, the choice is so natural, that it was worth to automate it in the *coinduction* proof method [14]. This method instantiates the coinduction rule $coinduct_{lang}$ with

the canonical bisimulation witness constructed from the goal, where the existentially quantified variables must be given explicitly using the *arbitrary* modifier. Then it applies the instantiated rule in a resolution step, discharges the first assumption, and unpacks the existential quantifiers from R in the remaining subgoal (the `obtain` step in the above Isar proof). Using this convenience many proofs, including the one above, collapse to an automatic one-liner. Some examples follow.

theorem $\varnothing + L = L$
 by (*coinduction arbitrary*: L) *auto*

theorem $L + L = L$
 by (*coinduction arbitrary*: L) *auto*

theorem $(L_1 + L_2) + L_3 = L_1 + L_2 + L_3$
 by (*coinduction arbitrary*: $L_1\ L_2\ L_3$) *auto*

theorem $\mathsf{in}\ (\mathsf{out}\ L) = L$
 by (*coinduction arbitrary*: L) *auto*

theorem $\mathsf{in}\ (L \cup K) = \mathsf{in}\ L + \mathsf{in}\ K$
 by (*coinduction arbitrary*: $L\ K$) *auto*

As indicated earlier, sometimes the *coinduction* method does not succeed. It is instructive to consider one example where this is the case: $o\ L \longrightarrow \varepsilon + L = L$.

If we attempt to prove the above statement by coinduction instantiated with the canonical bisimulation witness $R\ L_1\ L_2 = (\exists K.\ L_1 = \varepsilon + K \wedge L_2 = K \wedge o\ K)$, after some simplification we are stuck with proving that R is a bisimulation.

$$
\begin{aligned}
&R\ (\delta\ L_1\ x)\ (\delta\ L_2\ x) \leftrightarrow R\ (\delta\ (\varepsilon + K)\ x)\ (\delta\ K\ x)\\
&\leftrightarrow R\ (\delta\ \varepsilon\ x + \delta\ K\ x)\ (\delta\ K\ x) \leftrightarrow R\ (\varnothing + \delta\ K\ x)\ (\delta\ K\ x)\\
&\leftrightarrow R\ (\delta\ K\ x)\ (\delta\ K\ x) \leftrightarrow \exists K'.\ \delta\ K\ x = \varepsilon + K' \wedge \delta\ K\ x = K' \wedge o\ K'
\end{aligned}
$$

The remaining goal is not provable: we would need to instantiate K' with $\delta\ K\ x$, but then, we are left to prove $o\ (\delta\ K\ x)$, which we cannot deduce (we only know $o\ K$). If we, however, instantiate the coinduction rule with $R^{=}\ L_1\ L_2 = R\ L_1\ L_2 \vee L_1 = L_2$, we are able to finish the proof. This means that our canonical bisimulation witness R is not a bisimulation, but $R^{=}$ is. In such cases R is called a *bisimulation up to equality*.

Instead of modifying the *coinduction* to instantiate the rule $\mathit{coinduct}_{lang}$ with $R^{=}$, it is more convenient to capture this improvement directly in the coinduction rule. This results in the following rule which we call *coinduction up to equality* or $\mathit{coinduct}^{=}_{lang}$.

$$
\frac{R\ L\ K \qquad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow o\ L_1 \leftrightarrow o\ L_2 \wedge \forall x.\ R^{=}\ (\delta\ L_1\ x)\ (\delta\ L_2\ x)}{L = K}
$$

Note that $\mathit{coinduct}^{=}_{lang}$ is not just an instantiation of $\mathit{coinduct}_{lang}$. Otherwise all occurrences of R in $\mathit{coinduct}_{lang}$ would have been replaced with $R^{=}$ in $\mathit{coinduct}^{=}_{lang}$. Instead, after the instantiation, the first assumption is simplified to $R\ L\ K$—we would not use coinduction in the first place, if we could prove $R^{=}\ L\ K$ by reflexivity. Similarly, the reflexivity part in the occurrence on the left of the implication in the second assumption is needless and therefore eliminated. Using this coinduction up to equality rule, we have regained the ability of writing one line proofs.

`theorem` $o\ L \longrightarrow \varepsilon + L = L$
`by` $(\mathit{coinduction\ arbitrary}\colon L\ \mathit{rule}\colon \mathit{coinduct}^{=}_{lang})\ \mathit{auto}$

The next hurdle, however, is not long in coming. Suppose that we already have proved the standard selector equations for concatenation $\cdot$. (This required finding some auxiliary properties of $\hat{\cdot}$ and $\hat{\oplus}$ but was an overall straightforward usage of coinduction up to equality.) Next, we want to reason about $\cdot$. For example, we prove its distributivity over $+$: $(L + K) \cdot M = (L \cdot M) + (K \cdot M)$. As before, we are stuck proving that the canonical bisimulation witness $R\ L_1\ L_2 = (\exists L'\ K'\ M'.\ L_1 = (L' + K') \cdot M' \wedge L_2 = (L' \cdot M') + (K' \cdot M'))$ is a bisimulation (and this time even for up to equality).

$$
\begin{aligned}
&R^{=}\,(\delta\ L_1\ x)\ (\delta\ L_2\ x) \leftrightarrow R^{=}\,(\delta\ ((L' + K') \cdot M')\ x)\ (\delta\ ((L' \cdot M') + (K' \cdot M'))\ x)\\
&\leftrightarrow \begin{cases}
R^{=}\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M') \\ \quad ((\delta\ L'\ x \cdot M') + (\delta\ K'\ x \cdot M')) & \texttt{if}\ \neg o\ L' \wedge \neg o\ K' \\
R^{=}\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M' + \delta\ M'\ x) \\ \quad ((\delta\ L'\ x \cdot M' + \delta\ M'\ x) + (\delta\ K'\ x \cdot M')) & \texttt{if}\ o\ L' \wedge \neg o\ K' \\
R^{=}\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M' + \delta\ M'\ x) \\ \quad ((\delta\ L'\ x \cdot M') + (\delta\ K'\ x \cdot M' + \delta\ M'\ x)) & \texttt{if}\ \neg o\ L' \wedge o\ K' \\
R^{=}\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M' + \delta\ M'\ x) \\ \quad ((\delta\ L'\ x \cdot M' + \delta\ M'\ x) + (\delta\ K'\ x \cdot M' + \delta\ M'\ x)) & \texttt{if}\ o\ L' \wedge o\ K'
\end{cases}\\
&\leftrightarrow \begin{cases}
\top & \texttt{if}\ \neg o\ L' \wedge \neg o\ K' \\
R^{=}\,((\delta\ L'\ x + \delta\ K'\ x) \cdot M' + \delta\ M'\ x) \\ \quad ((\delta\ L'\ x \cdot M' + \delta\ K'\ x \cdot M') + \delta\ M'\ x) & \text{otherwise}
\end{cases}
\end{aligned}
$$

The remaining subgoal cannot be discharged by the definition of R. In principle it could be discharged by equality (the two tries are equal), but this is almost exactly the property we originally started proving, so we have not simplified the problem by coinduction but rather are going in circles here. However, if our relation could somehow split its arguments across the outermost $+$, we could prove the left pair being related by R and the right pair by $=$. The solution is easy: we allow the

relation to do just that. Therefore, we define the congruence closure R^{+} of a relation R under + inductively by the following rules.

$$\frac{L = K}{R^{+}\ L\ K} \qquad \frac{R\ L\ K}{R^{+}\ L\ K} \qquad \frac{R^{+}\ L\ K}{R^{+}\ K\ L} \qquad \frac{R^{+}\ L_1\ L_2 \quad R^{+}\ L_2\ L_3}{R^{+}\ L_1\ L_3} \qquad \frac{R^{+}\ L_1\ K_1 \quad R^{+}\ L_2\ K_2}{R^{+}\ (L_1 + L_2)\ (K_1 + K_2)}$$

The closure R^{+} is then used to strengthen the coinduction rule, just as the earlier reflexive closure $R^{=}$ strengthening. Note that the closure $R^{=}$ can also be viewed as inductively generated by the first two of the above rules. In summary, we obtain *coinduction up to congruence of* +, denoted by $coinduct^{+}_{lang}$.

$$\frac{R\ L\ K \quad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow o\ L_1 \leftrightarrow o\ L_2 \wedge \forall x.\ R^{+}\ (\delta\ L_1\ x)\ (\delta\ L_2\ x)}{L = K}$$

As intended this rule makes the proof of distributivity into an another automatic one-liner. This is due to the fact that our new proof principle, coinduction up to congruence of +, matches exactly the definitional principle of corecursion up to + used in the selector equations of ·.

```
theorem (L + K) · M = (L · M) + (K · M)
  by (coinduction arbitrary: L K M rule: coinduct⁺_lang) auto
```

For properties involving iteration $_^{*}$, whose selector equation are corecursive up to ·, we will need a further strengthening of the coinduction rule (using the congruence closure under ·). Overall, the most robust solution to keep track of the different rules is to maintain a coinduction rule up to all previously defined operation on tries: we define $R^{\star}$ to be the congruence closure of R under +, ·, and $_^{*}$ and then use the following rule for proving.

$$\frac{R\ L\ K \quad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow o\ L_1 \leftrightarrow o\ L_2 \wedge \forall x.\ R^{\star}\ (\delta\ L_1\ x)\ (\delta\ L_2\ x)}{L = K}$$

We will not spell out all axioms of Kleene algebra [72] and their formal proofs [110] here. Most proofs are automatic; some require a few manual hints in which order to apply the congruence rules. Note that the axioms of Kleene algebra also contain some inequalities, such as $\varepsilon + L \cdot L^{*} \le L^{*}$, and even conditional inequalities, such as $L + M \cdot K \le M \longrightarrow L \cdot K^{*} \le M$. However, $L \le K$ is defined as $L + K = K$, such that proofs by coinduction also are applicable here.

3.4 Coalgebraic Terminology

We briefly connect the intuitive notions, such as tries, from earlier sections with the key coalgebraic concepts and terminology that is usually used to present this

$$\begin{array}{ccc} \alpha & \xrightarrow{s} & \alpha\ F \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{map}_F\ f} \\ \beta & \xrightarrow{t} & \beta\ F \end{array}$$

Figure 3.4: Commutation property of a coalgebra morphism

view on formal languages. The working formalizer is still the target audience of this section and is expected to learn how particularly useful abstract objects gave rise to concrete tools in Isabelle/HOL. More theoretical and detailed introductions to coalgebra can be found elsewhere [62, 100].

Given a functor F an *(F-)coalgebra* is a *carrier* object A together with a map $A \to F\ A$—the *structural map* of a coalgebra. In the context of higher-order logic—that is in the category of types[1]—a functor is a type constructor F together with a map function $\mathsf{map}_F :: (\alpha \to \beta) \to \alpha\ F \to \beta\ F$ that preserves identity and composition: $\mathsf{map}_F\ \mathsf{id} = \mathsf{id}$ and $\mathsf{map}_F\ (f \circ g) = \mathsf{map}_F\ f \circ \mathsf{map}_F\ g$. An F-coalgebra in HOL is therefore simply a function $s :: \alpha \to \alpha\ F$. A function $f :: \alpha \to \beta$ is a coalgebra *morphism* between two coalgebras $s :: \alpha \to \alpha\ F$ and $t :: \beta \to \beta\ F$ if it satisfies the commutation property $t \circ f = \mathsf{map}_F\ f \circ s$, also depicted by the commutative diagram in Figure 3.4.

A particularly useful coalgebra is the *final (F-)coalgebra*: an F-coalgebra to which there exists an unique morphism from any other coalgebra. Not all functors F admit a final coalgebra. Two different final coalgebras are necessarily isomorphic. Final coalgebras are interesting, because in Isabelle/HOL they correspond to codatatypes. Isabelle's facility for defining codatatypes maintains a large class of functors—bounded natural functors [116]—for which a final coalgebra does exists. Not only that, for any bounded natural functor F, Isabelle can construct its final coalgebra with the codatatype CF as the carrier and define a bijective constructor $\mathsf{C}_F :: CF\ F \to CF$ and the inverse destructor $\mathsf{D}_F :: CF \to CF\ F$. The latter takes the role of the structural map of the coalgebra.

$\texttt{codatatype}\ CF = \mathsf{C}_F\ (\mathsf{D}_F : CF\ F)$

Finally, we are ready to connect these abstract notions to our tries. The codatatype of tries $\alpha\ lang$ is the final coalgebra of the functor $\beta\ D = bool \times (\alpha \to \beta)$ with the associated map function $\mathsf{map}_D\ g = \mathsf{id} \times (\lambda f.\ g \circ f)$, where $(f \times g)\ (x, y) = (f\ x, g\ y)$. The structural map of this final coalgebra is the function $\mathsf{D}_D = \langle o, \delta \rangle$, where $\langle f,\ g \rangle\ x = (f\ x, g\ x)$.

The finality of $\alpha\ lang$ gives rise to the definitional principle of primitive corecursion. In Isabelle this principle is incarnated by the primitive corecursor $\mathsf{corec} :: (\tau \to \tau\ D) \to \tau \to \alpha\ lang$, that assigns the given D-coalgebra the unique morphism from it

[1] This category consists of types as objects and of functions between types as arrows.

$$\begin{array}{ccc} \tau & \xrightarrow{s} & \tau\ D \\ {\scriptstyle \mathsf{corec}\ s}\downarrow & & \downarrow{\scriptstyle \mathsf{map}_D\ (\mathsf{corec}\ s)} \\ \alpha\ lang & \xrightarrow{\langle o, \delta \rangle} & \alpha\ lang\ D \end{array}$$

Figure 3.5: Unique morphism corec s to the final coalgebra $(\alpha\, lang, \langle o, \delta \rangle)$

to the final coalgebra. In other words, the primitive corecursor allows us to define functions of type $\tau \to \alpha\, lang$ by providing an D-coalgebra on τ, i.e. a function of type $\tau \to bool \times (\alpha \to \tau)$ that essentially describes a deterministic (not necessarily finite) automaton without an initial state. To clarify this automaton analogy, it is customary to present the F-coalgebra s as two functions $s = \langle \mathsf{o}, \mathsf{d} \rangle$ with τ being the states of the automaton, $\mathsf{o} : \tau \to bool$ denoting accepting states, and $\mathsf{d} : \alpha \to \tau \to \tau$ being the transition function. From a given s, we uniquely obtain the function corec s that assigns to a separately given initial state $t : \tau$ the language corec $s\ t : \alpha\, lang$ and makes the diagram in Figure 3.5 commute.

The `primcorec` command [14] reduces a user given specification to a non-recursive definition using the corecursor. For example, the union operation + is internally defined as $\lambda L\, K.\ \mathsf{corec}\ (\lambda(L, K).\ (o\ L \wedge o\ K, \lambda a.\ (\delta\ L\ a, \delta\ K\ a)))\ (L, K)$. The D-coalgebra given as the argument to corec resembles the right hand sides of the selector equations for + (with the corecursive calls stripped away). As end users, most of the time we are happy being able to write the high-level corecursive specifications, without having to explicitly supply coalgebras.

It is worth noting that the final coalgebra $\alpha\, lang$ itself corresponds to the automaton, whose states are languages, acceptance is given by $\mathsf{o}\ L = o\ L$, and the transition function by $\mathsf{d}\ a\ L = \delta\ L\ a$. For this definitions, we obtain $\mathsf{corec}\ \langle \mathsf{o}, \mathsf{d} \rangle\ (L : \alpha\, lang) = L$. For regular languages this automaton corresponds to the minimal automaton (since equality on tries corresponds to language equivalence), which is finite by the Myhill-Nerode theorem. This correspondence is not very practical though, since we typically label states of automata with something finite, in particular not with languages (represented by infinite tries).

A second consequence of the finality of $\alpha\, lang$ is the coinduction principle that we have seen earlier. It follows from the fact that final coalgebras are quotients by bisimilarity, where bisimilarity is defined as the existence of a bisimulation relation.

Inspired by the coalgebraic view on formal languages, we tackle deciding their equivalence generically in the next chapter.

You don't have to be a genius or a visionary
or even a college graduate to be successful.
You just need a framework and a dream.

— Michael Dell (1999)

Chapter 4

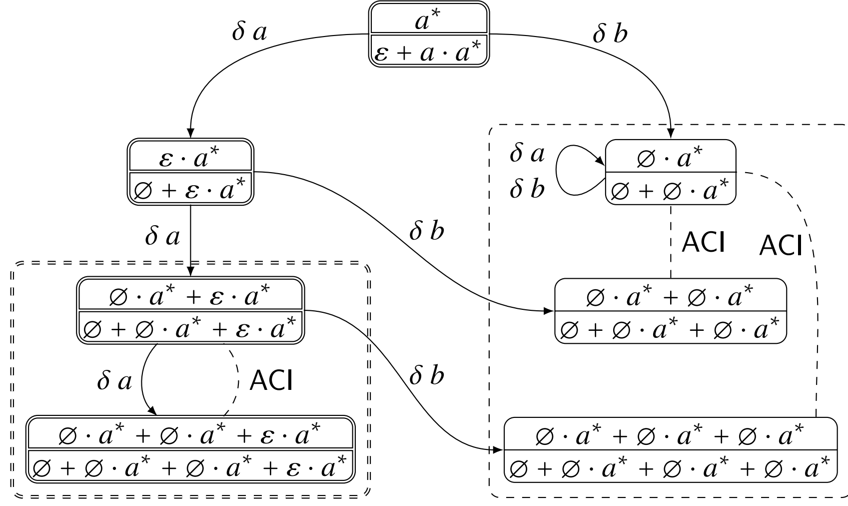
Language Equivalence Framework

Our dream is deciding equivalence of regular languages and in this chapter we present a formalized framework for doing so. The central idea of the generic algorithm is rooted in the coinduction principle $\mathit{coinduct}_{\mathit{lang}}$ from the previous chapter. Essentially, the algorithm iteratively constructs a bisimulation.

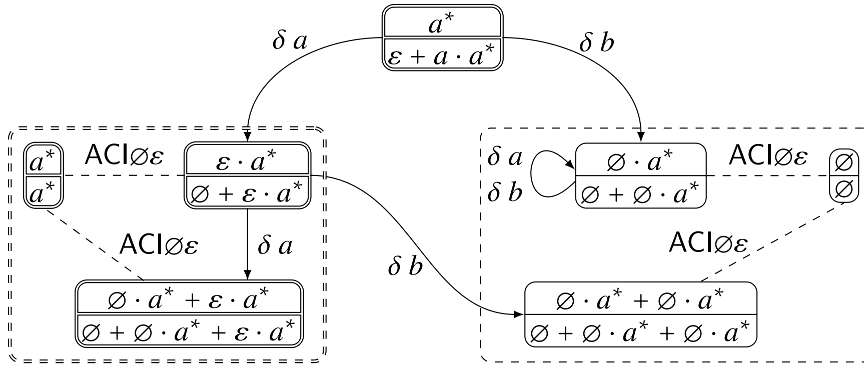
4.1 The Algorithm, Informally

Before presenting our abstract solution, we informally describe what the decision procedure does when instantiated with regular expressions, using for the latter the notation introduced in Chapter 2. The coalgebraic procedure for regular expression equivalence iteratively constructs a relation $P : \alpha\, RE \to \alpha\, RE \to \mathit{bool}$ whose direct image under the regular expressions' semantics L is a bisimulation. For example, suppose we want to prove the regular expressions r and s being equivalent. We start with the pair (r, s), check whether both are *consistently accepting* the empty word via $o\ r \leftrightarrow o\ s$ and add it to the relation P. Then we close P under the pairwise Brzozowski derivative δ for all letters of the alphabet. Whenever a new pair (t, u) is added to P, it is checked to fulfill $o\ t \leftrightarrow o\ u$. New means here that the pair is different from all already explored pairs modulo some syntactic equivalence relation R that is finer than language equivalence, such as associativity, commutativity, and idempotence (ACI) of the + constructor. The choice of R will be crucial for the termination and the performance of the algorithm. Once P is closed under derivatives and all empty word acceptance checks have been passed we know that it is a (syntactic) bisimulation and therefore by coinduction the languages of the input regular expressions r and s coincide.

Alternatively, one can view this procedure as the well-known product automaton construction, where one regards the derivatives as an implicit or lazy description of an automaton whose states are labeled with expressions. Language equivalence means then $o\ \ r \leftrightarrow o\ \ s$ for all states (r, s) of this product automaton. On the other hand, using an automaton usually implies that one is interested in transitions be-



(a) Product automaton/bisimulation modulo ACI



(b) Product automaton/bisimulation modulo ACI$\varnothing\varepsilon$

Figure 4.1: Checking the equivalence of a^* and $\varepsilon + a \cdot a^*$ for $\Sigma = \{a, b\}$

tween states. We are not really interested in that, but only in the state space of the product automaton (given by the relation P above). Thus, the bisimulation terminology is more suitable.

We observe the decision procedure at work, before we tackle its formal (and abstract) definition by looking at the regular expressions a^* and $\varepsilon + a \cdot a^*$ for some $a \in \Sigma = \{a, b\}$. For presentation purposes, the correspondence of derivatives to automata is useful. Figure 4.1 shows two automata, the states of which are equivalence classes of pairs of regular expressions indicated by a dashed fringe (which is omitted for singleton classes). The equivalence classes of automaton (a) are modulo ACI, while those of automaton (b) are modulo a stronger relation ACI$\varnothing\varepsilon$, that for example also relates $\varnothing + r$ with r, $\varnothing \cdot r$ with $\varnothing$, and $\varepsilon \cdot r$ with r and thereby makes the automaton smaller. Transitions correspond to pairwise derivatives and dou-

bled margins denote states for which the associated pairs of regular expressions are pairwise accepting the empty word. The absence of pairs (r, s) for which r is accepting the empty word and s is not accepting it (or vice versa) proves the language equivalence of all pairs in the automaton, including the initial pair $(a^*, \varepsilon + a \cdot a^*)$.

Let us mention an obvious performance deficit of our algorithm. The algorithm constructs a bisimulation (not even a bisimulation up to equality). This effectively means that even when applied on two identical expressions, the algorithm would still enumerate all derivatives. There is a whole hierarchy of possible improvements: bisimulation up to equality, equivalence, congruence, and congruence and context, which have been successfully employed in unverified derivative-based decision procedures [21, 96]. However, when starting to verify an algorithm one has to settle for a solution somewhere in between of efficiency and simplicity.

4.2 Coalgebras as Locales

The above procedure is not new [49] and its formalization and proof of soundness in Isabelle [76] precedes the work in this thesis. Indeed this early formalization was our starting point for carrying out several generalizations. First, we have refined the procedure to cover extended regular expressions and additionally proved completeness [113] (see also Chapter 6). Second, we have abstracted the procedure over the kind of derivatives, thereby enabling, for example, the usage of partial derivatives instead of Brzozowski derivatives [89] (Chapter 5). Third, we have also generalized the syntactic structures that are checked for equivalence [111]. This enables deciding equivalence of logical formulas instead of regular expressions (Chapter 8).

What we present below culminates this series of generalizations by additionally allowing the two structures that are checked for equivalence to be different. This can be used to check whether formulas of different logics (or regular expressions) denote the same regular language. To achieve this, our algorithm is parametrized by two syntactic D-coalgebras (as introduced in Section 3.4), both equipped with their own notions of derivatives.

This requires the notion of a D-coalgebra to be formalized in Isabelle. This is done by means of the following locale, to be read column-wise.

```
locale coalg_D =
  fixes Σ :: α list              assumes distinct Σ
  and   i :: τ → σ               and wf^τ t ⟶ wf^σ (i t)
  and   o :: σ → bool            and wf^τ t ⟶ L^σ (i t) = L^τ t
  and   d :: α → σ → σ           and wf^σ s ∧ w ∈ L^σ s ⟶ w ∈ (set Σ)*
  and   L^σ :: σ → α lang        and wf^σ s ∧ a ∈ set Σ ⟶ wf^σ (d a s)
  and   wf^σ :: σ → bool         and wf^σ s ∧ a ∈ set Σ ⟶ L^σ (d a s) = δ (L^σ s) a
  and   L^τ :: τ → α lang        and wf^σ s ⟶ o s = o (L^σ s)
  and   wf^τ :: τ → bool
```

This locale is more complex than one would expect at first, so let us slowly understand the design decisions behind its parameters and assumptions. The parameters contain three type variables: α as the type of letters, τ as the type of structures about which we want to reason (for example regular expressions), and σ as the type of modified structures for which we actually supply the coalgebra (for example regular expressions normalized in some way).

We assume that the alphabet is given by the list Σ. This simultaneously ensures that the alphabet is finite and allows us to take subsets of the type α as the alphabet (without having to define a subtype of α). The list Σ is assumed to be duplicate-free.

The initial transformation i translates τ-structures into σ-structures. Later, i will be instantiated either with an identity, or with a certain normalization function, or sometimes even with a real translation (of formulas into regular expressions).

The actual D-coalgebra is given by the two functions o and d. Note that in principle, we could use this (syntactic) coalgebra to define a language semantics of σ-structures as $\mathsf{corec}\ \langle \mathsf{o}, \mathsf{d} \rangle :: \sigma \to \alpha\ lang$. However, we prefer to put the burden of supplying the semantics of σ-structures via the L^σ parameter on the side of whoever interprets the locale for two reasons. First, a σ-structure will typically be an inductively defined piece of syntax. Therefore, an inductive definition of the semantics will be a better fit on the interpreter's side than the above uniform coinductive definition. Second, we want to allow that not all σ-structures are wellformed (again bypassing a subtype construction). For non-wellformed structures the semantics then does not need to be defined at all. Wellformedness is specified by yet another parameter wf^σ.

The semantics of τ-structures (as well as the wellformedness predicate wf^τ) must also be supplied separately . But since we do not require a coalgebraic structure on τ, there is no other way to obtain the semantics anyway.

The assumptions of the locale relate its parameters in a straightforward fashion. In particular they formalize the preservation of wellformedness by i and d, and the expected semantic effects of i, d, and o on wellformed structures.

Note that unlike in the concrete description of the procedure, there is no notion of an equivalence relation modulo which σ-structures are compared. Later the equivalence relation will be directly captured in the very notion of σ-structures, either instantiating them with equivalence classes directly or with canonical representatives computed via a normalization function.

In the context of coalg_D, a decision procedure for the word problem (that is matching) on τ structures is easy to define and prove correct. The formal proof by induction takes one line and is omitted (as most of the formal proofs will be).

```
definition (in coalg_D) match :: τ → α list → bool where
  match t w = o (fold d w (i t))
```

$\texttt{theorem}$ ($\texttt{in}$ coalg_D) $\mathsf{match}\ r\ w \leftrightarrow w \in \mathsf{L}^\tau\ t$

When instantiating the locale parameters o, d, and i with executable functions, we obtain an executable algorithm for matching via Isabelle's code generator.

We strive for a decision procedure for language equivalence and not matching. Let us tackle this next. To be as flexible as possible, we assume that we are given two coalgebras—each in form of the locale coalg_D—that do not even need to share the same alphabet. This is formalized by another locale.

$\texttt{locale}$ coalgs_D =
  $\mathsf{A} : \mathsf{coalg}_D$ +
  $\mathsf{B} : \mathsf{coalg}_D$ +
  $\texttt{fixes}$ $\triangledown :: \mathsf{A}.\alpha \to \mathsf{B}.\alpha \to bool$
  $\texttt{assumes}$ $a \triangledown b \longrightarrow a \in \mathsf{set}\ \mathsf{A}.\Sigma \leftrightarrow b \in \mathsf{set}\ \mathsf{B}.\Sigma$

The additional parameter $\triangledown$ relates letters of the two (potentially different) alphabets in an alphabet-respecting fashion. A priori, it is unclear what language equivalence of structures over different alphabet should be. The constant $\triangledown$ helps to clarify this by defining *$\triangledown$-equivalence* as follows.

$\texttt{definition}$ ($\texttt{in}$ coalgs_D) $\equiv_\triangledown\ :: \mathsf{A}.\alpha\ lang \to \mathsf{B}.\alpha\ lang \to bool$ $\texttt{where}$
  $L \equiv_\triangledown K \leftrightarrow \forall w, v.\ w \triangledown^* v \longrightarrow w \in L \leftrightarrow v \in K$

Here, the relation $\triangledown^*$ denotes $\triangledown$ lifted to words. More precisely we define, $w \triangledown^* v = |w| = |v| \wedge (\forall i < |v|.\ w[i] \triangledown v[i])$. Two languages are then $\triangledown$-equivalent if all pairs of words of the same length, that are related letterwise by $\triangledown$, behave equally with respect to language membership. Note that $\triangledown$-equivalence also caters for a nice coinductive characterization, in the spirit of the coinduction rules for $\alpha\ lang$. In this characterization o and δ denote the semantic operations (selectors) on tries.

$$\frac{R\ L\ K \qquad \forall L_1\ L_2.\ R\ L_1\ L_2 \longrightarrow o\ L_1 \leftrightarrow o\ L_2 \wedge (\forall a\ b.\ a \triangledown b \longrightarrow R\ (\delta\ L_1\ a)\ (\delta\ L_2\ b))}{L \equiv_\triangledown K}$$

It is easy to show that instantiating $\triangledown$ with equality, yields the usual notion of language equivalence as trie (or set) equality: $L \equiv_= K \leftrightarrow L = K$.

4.3 The Algorithm, Formally

In the context of coalgs_D, we can finally introduce our procedure for checking $\triangledown$-equivalence of $t :: \mathsf{A}.\tau$ and $u :: \mathsf{B}.\tau$. Therefore, we iteratively enumerate the following set of pairwise derivatives of $\mathsf{A}.\mathsf{i}\ t$ and $\mathsf{B}.\mathsf{i}\ u$, which constitutes a bisimulation

given that each pair ($r :: \mathsf{A}.\sigma$, $s :: \mathsf{B}.\sigma$) in this set is consistently accepting—that is it satisfies $\mathsf{A}.o\ r \leftrightarrow \mathsf{B}.o\ s$.

$$\{(\mathsf{fold}\ \mathsf{A.d}\ w\ (\mathsf{A.i}\ t), \mathsf{fold}\ \mathsf{B.d}\ v\ (\mathsf{B.i}\ u)) \mid w \in (\mathsf{set}\ \mathsf{A}.\Sigma)^* \wedge v \in (\mathsf{set}\ \mathsf{B}.\Sigma)^* \wedge w \triangledown^* v\}$$

So far, there is no guarantee that this set is finite. We postpone the discussion of termination until after the presentation of the enumeration. Here is a recursive worklist algorithm that performs the enumeration, given as functional pseudocode.

```
Σ :: (A.α × B.α) list
Σ = [(a, b) | a ← A.Σ, b ← B.Σ, a ▽ b]

closure :: (A.σ × B.σ) list × (A.σ × B.σ) set → bool
closure ([], _) = ⊤
closure ((r, s) # ws, P) = if A.o r ≠ B.o s then ⊥ else
    let
        add (a, b) (ws, P) =
            let rs = (A.d a r, B.d b s)
            in if rs ∈ P then (ws, P) else (rs # ws, {rs} ∪ P)
    in closure (fold add Σ (ws, P))

eqv :: A.τ → B.τ → bool
eqv t u = wf^τ t ∧ wf^τ u ∧ let rs₀ = (A.i t, B.i u) in closure ([rs₀], {rs₀})
```

The core of the algorithm is the tail recursive closure function. It operates on a state consisting of a worklist *ws* of pairs of σ-structures and a set P of so far encountered pairs (including those that are in the worklist). The only way to terminate the recursion is to empty the worklist. In each iteration, the head pair of the worklist is checked to be consistently accepting and after passing this check is replaced by its pairwise derivatives that were not previously encountered.

The above pseudocode is not yet the formalization of eqv in Isabelle/HOL. Since HOL is a logic of total functions, in order to guarantee totality, a function has to either be terminating or its non-terminating behavior has to be uniquely specified. Since the conditions under which closure will terminate will be discussed only later, we opt for the second possibility.

We can use the while combinator to redefine our equivalence check, this time formally in Isabelle. In our case, the state s of the while loop has the same type $\gamma = (\mathsf{A}.\sigma \times \mathsf{B}.\sigma)\ \mathit{list} \times (\mathsf{A}.\sigma \times \mathsf{B}.\sigma)\ \mathit{set}$ as the argument to closure. We define the arguments to the while combinator b and c to obtain an executable formal definition of closure. A further helper function s, that initializes the state of the loop, completes the executable formal definition of the equivalence check eqv.

```
definition (in coalgs_D) b :: γ → bool where
    b ([], _) = ⊥
```

```
    b ((r, s) # _, _) = A.o r ↔ B.o s

definition (in coalgs_D) c :: γ → γ where
    c ((r, s) # ws, P) =
        let
            add (a, b) (ws, P) =
                let rs = (A.d a r, B.d b s)
                in if rs ∈ P then (ws, P) else (rs # ws, {rs} ∪ P)
        in fold add Σ (ws, P)

definition (in coalgs_D) closure :: γ → bool where
    closure x = case while b c x of Some ([], _) ⇒ ⊤ | _ ⇒ ⊥

definition (in coalgs_D) s :: A.τ → B.τ → γ where
    s t u = let rs_0 = (A.i t, B.i u) in ([rs_0], {rs_0})

definition (in coalgs_D) eqv :: A.τ → B.τ → bool where
    eqv t u = A.wf^τ t ∧ B.wf^τ u ∧ closure (s t u)
```

Next we prove that this algorithm is sound and refutationally sound. Soundness means that if eqv terminates with the result $\top$, the input τ-structure are $\triangledown$-equivalent. Refutational soundness means that if eqv terminates with the result $\bot$, the input τ-structure are $\triangledown$-inequivalent. Note that eqv is also logically specified to be $\bot$ in case of non-termination, however the extracted code will actually not terminate in this case—Isabelle's code generator guarantees only partial correctness.

As the key step of the proof, we note that, given two initial wellformed τ-structures t and u, the following invariant invar holds for the states of the while loop.

```
definition (in coalgs_D) invar :: (A.τ × B.τ) → γ → bool where
    invar (t, u) (ws, P) =
        set ws ⊆ P ∧
        (∀(r, s) ∈ P. ∃w ∈ (set A.Σ)*. ∃v ∈ (set B.Σ)*.
            w ▽* v  ∧  r  =  fold  A.d w (A.i t)  ∧  s  =  fold  B.d v (B.i u))  ∧
        (∀(r, s) ∈ P ∖ set ws. A.o r ↔ B.o s ∧
            (∀a ∈ set A.Σ. ∀b ∈ set A.Σ. a ▽ b ⟶ (A.d a r, B.d b s) ∈ P))
```

It is not hard to prove invar being indeed a loop invariant—that is to show that invar (t, u) (s t u) and $\forall x.$ invar (t, u) $x \longrightarrow$ b $x \longrightarrow$ invar (t, u) (c x) hold.

Then, soundness follows by coinduction (for $\equiv_\triangledown$) where the bisimulation witness has been instantiated with the set P (viewed as a binary relation) constructed by the while loop. The third conjunct of invar (t, u) $([], P)$ immediately implies that P is a bisimulation in the sense of the used coinduction rule. We obtain the following formalized theorem. The assumption eqv t u also includes the termination of the procedure and the fact that the input structures are wellformed.

theorem (in coalgs_D) $\mathsf{eqv}\ t\ u \longrightarrow \mathsf{A.L}^\tau\ t \equiv_\triangledown \mathsf{B.L}^\tau\ u$

Refutational soundness also follows from the invariant. Here we may assume that the procedure has terminated in a state (ws, P) with a non-empty worklist. Then, from invar (t, u) (ws, P) and $\neg\mathsf{b}$ (ws, P) (the only way to exit the loop), we obtain a pair of $\triangledown^*$-related words w and v with $\mathsf{A.match}\ w\ t \not\leftrightarrow \mathsf{B.match}\ v\ u$. Using the characteristic property of match, we deduce $w \in \mathsf{A.L}^\tau\ t \not\leftrightarrow v \in \mathsf{B.L}^\tau\ u$, which contradicts $\triangledown$-equivalence. To capture exactly refutational soundness, the formal statement has to explicitly exclude other cases where eqv t u returns $\bot$, such as one of the input formulas being not wellformed but also non-termination of the while loop.

theorem (in coalgs_D)
$\mathsf{A.wf}^\tau\ t \wedge \mathsf{B.wf}^\tau\ u \wedge \mathsf{while\ b\ c}\ (\mathsf{s}\ t\ u) \neq \mathsf{None} \wedge \neg\mathsf{eqv}\ t\ u \longrightarrow \mathsf{A.L}^\tau\ t \not\equiv_\triangledown \mathsf{B.L}^\tau\ u$

4.4 Termination

Now that partial correctness of our equivalence check is established, we can turn our attention to termination. Clearly, from our abstract assumptions we can not deduce that the set of all word derivatives of a σ-structure is finite, and therefore cannot guarantee termination without further ado.

Even for our concrete algorithm description from the beginning of this section, it is not clear if the procedure always terminates. Indeed, it often does not terminate, when plain Brzozowski derivatives are used to compute the next step for the case of regular expressions. However, if after each derivative step the expressions are brought into a certain normal form the termination of the procedure is magically guaranteed. The magic is explained by Brzozowski's fundamental result, that there are only finitely many distinct word derivatives modulo a certain equivalence relation. We will revisit and formalize this result in the next chapter. Here, it is important to understand that later instantiations for the parameter d of coalg_D will always be some derivative operation followed by a normalization function.

In our general setting, we add the assumption that there are only finitely many distinct (normalized) word derivatives of wellformed τ-structures in a separate locale.

locale $\mathsf{fin_coalg}_D$ =
coalg_D +
assumes $\mathsf{wf}^\tau\ t \longrightarrow \mathsf{finite}\ \{\mathsf{fold\ d}\ w\ (\mathsf{i}\ t) \mid w \in (\mathsf{set}\ \Sigma)^*\}$

As before, we want to check equivalence of two different coalgebras.

locale $\mathsf{fin_coalgs}_D$ =
A : $\mathsf{fin_coalg}_D$ +

$\mathsf{B} : \mathsf{fin_coalg}_D +$
`fixes` $\triangledown :: \mathsf{A}.\alpha \to \mathsf{B}.\alpha \to \mathit{bool}$
`assumes` $a \triangledown b \longrightarrow a \in \mathsf{set}\ \mathsf{A}.\Sigma \leftrightarrow b \in \mathsf{set}\ \mathsf{B}.\Sigma$

Of course, we prefer not to redefine all our machinery for equivalence checking developed within the coalgs_D locale in $\mathsf{fin_coalgs}_D$. Indeed every instance of $\mathsf{fin_coalgs}_D$ can be seen as an instance of coalgs_D. The locale mechanism conveniently allows us to include everything from the less restricted locale coalgs_D into the more restricted $\mathsf{fin_coalgs}_D$ via the `sublocale` command.

`sublocale` $\mathsf{fin_coalgs}_D \leq \mathsf{coalgs}_D$

Now, the equivalence check eqv is also available in the locale $\mathsf{fin_coalgs}_D$. The additional assumptions allow us to prove termination. The key idea is that in each step of the while loop either the worklist becomes shorter or the set P is added a new element. Note however that due to our invariant P is a subset of the finite set $\{\mathsf{fold}\ \mathsf{A}.\mathsf{d}\ w\ (\mathsf{A}.\mathsf{i}\ t) \mid w \in (\mathsf{set}\ \mathsf{A}.\Sigma)^*\} \times \{\mathsf{fold}\ \mathsf{B}.\mathsf{d}\ v\ (\mathsf{B}.\mathsf{i}\ u) \mid w \in (\mathsf{set}\ \mathsf{B}.\Sigma)^*\}$.

`theorem` $(\mathsf{in}\ \mathsf{fin_coalgs}_D)\ \mathsf{A}.\mathsf{wf}^\tau\ t \wedge \mathsf{B}.\mathsf{wf}^\tau\ u \longrightarrow \mathsf{while}\ \mathsf{b}\ \mathsf{c}\ (\mathsf{s}\ t\ u) \neq \mathsf{None}$

Using this the refutational soundness theorem becomes a completeness property.

`theorem` $(\mathsf{in}\ \mathsf{fin_coalgs}_D)\ \mathsf{A}.\mathsf{wf}^\tau\ t \wedge \mathsf{B}.\mathsf{wf}^\tau\ u \wedge \mathsf{A}.\mathsf{L}^\tau\ t \equiv_\triangledown \mathsf{B}.\mathsf{L}^\tau\ u \longrightarrow \mathsf{eqv}\ t\ u$

The additional finiteness assumptions of $\mathsf{fin_coalgs}_D$ are the hardest ones to discharge. Moreover, the usage of stronger normalization functions as part of d (thereby identifying more equivalent structures syntactically) reduces the search space of the algorithm. Typically, the stronger the normalization, the more difficult it is to formally prove the finiteness assumption—this is somewhat counterintuitive but will be clarified, when we consider concrete instantiations. Often, we will even settle for a partially correct algorithm (instantiating the locale coalgs_D rather than $\mathsf{fin_coalgs}_D$), in cases where we were not able to prove finiteness formally (although we strongly believe that it holds). A verified partially correct algorithm that terminates on all the tested examples is still better than an inefficient verified totally correct algorithm that runs out of resources on all but trivial examples.

We conclude this chapter by noticing that any instance of coalg_D can be seen as an instance of coalgs_D by instantiating the latter with twice the coalgebra structure of the former and with equality for the $\triangledown$ relation. This is again expressed by a `sublocale` command (also for the locales with the additional finiteness assumption) and pulls the definition of the $\triangledown$-equivalence check into the coalg_D locale. In coalg_D, $\triangledown$-equivalence then coincides with standard language equivalence.

```
sublocale coalg_D X ≤ coalgs_D where
  A = X    B = X    a ▽ b = (a = b)

sublocale fin_coalg_D X ≤ fin_coalgs_D where
  A = X    B = X    a ▽ b = (a = b)
```

Some people, when confronted with a problem,
think "I know, I'll use regular expressions."
Now they have two problems.

— Jamie Zawinski (1997)

Chapter 5

Regular Expression Equivalence

Equivalence of regular expressions is a perennial topic in computer science. Recently it has spawned a number of formalized and verified decision procedures for this task in interactive theorem provers [5,24,35,76,85]. Except for the formalization by Braibant and Pous [24], all these decision procedures operate directly on variations of regular expressions. Although they (implicitly) build automata, the states of the automata are labeled with regular expressions, and there is no global transition table but the next-state function is computable from the regular expressions, resembling a lot the coalgebras in our framework. Yet all these decision procedures look very different. Of course, the next-state functions all differ, but so do the actual decision procedures and their correctness, completeness and termination proofs.

The primary contribution of this chapter is the instantiation of our framework from the previous chapter with all the above approaches (Sections 5.1 and 5.2). This unifies the different presentations and allows us to reuse proofs of correctness, completeness, and termination that were performed once and for all in the framework. Further secondary contributions include:

- A new perspective on partial derivatives that recasts them as Brzozowski derivatives followed by some rewriting (Section 5.1).
- The discovery that Asperti's algorithm is not the one by McNaughton-Yamada [82], as stated by Asperti [5], but a dual construction which apparently had not been considered in the literature and which produces uniformly smaller automata (Section 5.2).
- A linear time version (for a fixed alphabet) of Asperti's quadratic next-state function (Section 5.2).
- An empirical comparison of the performance of the different instantiations (Section 5.3).

This chapter is largely based on the material presented at ITP 2014 [89].

5.1 Derivatives

In 1964, Brzozowski [25] showed how to compute left quotients syntactically—as derivatives of regular expressions. Derivatives have been rediscovered in proof assistants by Krauss and Nipkow [76] and Coquand and Siles [35]. Our first instantiations of the framework reuse infrastructure from earlier formalizations in Isabelle [76, 113].

A refinement of Brzozowski's approach—partial derivatives—was introduced by Antimirov [3] and formalized by Moreira et al. [85] in Coq and by Wu et al. [119] in Isabelle. Partial derivatives operate on finite sets of regular expressions. They can be viewed either as a nondeterministic automaton with regular expressions as states or as the corresponding deterministic automaton obtained by the subset construction.

In the following, we integrate the two notions in our framework and show how derivatives can be used to simulate partial derivatives without invoking sets explicitly.

5.1.1 Brzozowski Derivatives

Since we have already seen the definitions of Brzozowski derivatives δ and the empty word acceptance test o for regular expressions in Chapter 2, we can immediately proceed with our first instantiation of the coalg_D locale. Here, we assume the type α being finite and ordered, such that we can write as_list UNIV for the list of all elements of α; note that set (as_list UNIV) = UNIV in this case.

```
interpretation coalg_D where
  Σ = as_list UNIV
  i r = r
  o r = o r
  d a r = δ a r
  L^σ r = L^τ r = L r
  wf^σ r = wf^τ r = ⊤
```

Unfortunately, the partially correct equivalence checker that is produced by this interpretation is completely useless in practice, because it will rarely terminate. For example, the regular expression a^* has infinitely many distinct derivatives, as all derivatives with respect to words a^n are distinct: $\mathsf{fold}\ \delta\ a^1\ a^* = \varepsilon \cdot a^*$; $\mathsf{fold}\ \delta\ a^{n+1}\ a^* = \emptyset \cdot a^* + \mathsf{fold}\ \delta\ a^n\ a^*$.

Fortunately, Brzozowski showed that there are finitely many equivalence classes of derivatives modulo associativity, commutativity and idempotence (ACI) of the + constructor. We prove that the number of distinct derivatives of r modulo ACI is finite: $\mathsf{finite}\ \{[\mathsf{fold}\ \delta\ w\ r]_\sim \mid w :: \alpha\ list\}$ where $[r]_\sim = \{s \mid r \sim s\}$ denotes the equivalence class of r and ACI equivalence $\sim$ is defined inductively as follows.

$$r + (s + t) \sim (r + s) + t \qquad r + s \sim s + r \qquad r + r \sim r$$

$$r \sim r \qquad \frac{r \sim s}{s \sim r} \qquad \frac{r \sim s \quad s \sim t}{r \sim t}$$

$$\frac{r_1 \sim s_1 \quad r_2 \sim s_2}{r_1 + r_2 \sim s_1 + s_2} \qquad \frac{r_1 \sim s_1 \quad r_2 \sim s_2}{r_1 \cdot r_2 \sim s_1 \cdot s_2} \qquad \frac{r \sim s}{r^* \sim s^*}$$

ACI-equivalent regular expressions $r \sim s$ have the same languages, and their equivalence is preserved by the derivative: $\delta\ a\ r \sim \delta\ a\ s$ for all $a :: \alpha$. This enables the following interpretation of the $\mathsf{fin_coalg}_D$ locale that operates on ACI equivalence classes. We obtain a first totally correct and complete equivalence checker $\mathsf{D}_\sim.\mathsf{eqv}$ in Isabelle/HOL.

interpretation $\mathsf{D}_\sim : \mathsf{fin_coalg}_D$ **where**
$\Sigma = \mathsf{as_list\ UNIV}$
$\mathsf{i}\ r = [r]_\sim$
$\mathsf{o}\ [r]_\sim = o\ r$
$\mathsf{d}\ a\ [r]_\sim = [\delta\ a\ r]_\sim$
$\mathsf{L}^\sigma\ [r]_\sim = \mathsf{L}^\tau\ r = \mathsf{L}\ r$
$\mathsf{wf}^\sigma\ [r]_\sim = \mathsf{wf}^\tau\ r = \top$

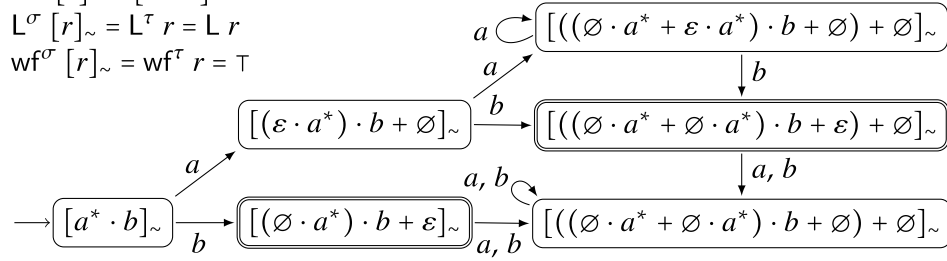


Figure 5.1: Derivative automaton modulo ACI for $a^* \cdot b$

Technically, the formalization defines a quotient type [64] of "regular expressions modulo ACI" to represent equivalence classes and uses Lifting and Transfer [60] to lift operations on regular expressions to operations on equivalence classes. The above presentation of definitions of the locale parameters by "pattern matching" on equivalence classes resembles the code generated by Isabelle for quotients (a pseudo-constructor [53], $[_]_\sim$, wraps a concrete representative r), rather than the actual definitions by Lifting.

Since the equivalence checker must compare equivalence classes, the code generation for quotients requires an executable equality (that is a decision procedure for $\sim$-equivalence). We achieve this through an ACI normalization function $\langle_\rangle$ that maps a regular expression r to a canonical representative of $[r]_\sim$ by sorting all summands with respect to an arbitrary fixed linear order $\leq$ while removing duplicates. The definition of $\langle_\rangle$ employs a smart (simplifying) constructor $\oplus$, whose equations are matched sequentially. Note that smart constructors adhere to the invariant that they yield canonical representatives when applied to canonical representatives .

```
primrec ⟨_⟩ :: α RE → α RE where
  ⟨∅⟩ = ∅
  ⟨ε⟩ = ε
  ⟨A a⟩ = A a
  ⟨r + s⟩ = ⟨r⟩ ⊕ ⟨s⟩
  ⟨r · s⟩ = ⟨r⟩ · ⟨s⟩
  ⟨r*⟩ = ⟨r⟩*
```

```
fun ⊕ :: α RE → α RE → α RE where
  (r + s) ⊕ t = r ⊕ (s ⊕ t)
  r ⊕ (s + t) = if r = s then s + t
    else if r ≤ s then r + (s + t)
      else s + (r ⊕ t)
  r ⊕ s = if r = s then r
    else if r ≤ s then r + s else s + r
```

We obtain an executable decision procedure for ACI equivalence: $r \sim s \leftrightarrow \langle r \rangle = \langle s \rangle$. This makes $\mathsf{D}_\sim$.eqv executable, yielding verified code in different functional programming languages via Isabelle's code generator. Yet, the performance of the generated code is disappointing. Figure 5.1 shows why; it shows the automaton that results from visualizing our syntactic coalgebra given in $\mathsf{D}_\sim$ applied to the initial state $a^* \cdot b$ (arrows represent derivative computations via d and states with a double margin are exactly those for which the empty word acceptance test o yields ⊤). We see that derivatives clutter concrete representatives with duplicated summands. Further derivative steps perform the same computation repeatedly and hence become increasingly expensive. This bottleneck is avoided by taking canonical ACI-normalized representatives as states yielding a second interpretation.

```
interpretation D : fin_coalg_D where
  Σ = as_list UNIV
  i r = ⟨r⟩
  o r = o r
  d a r = ⟨δ a r⟩
  L^σ r = L^τ r = L r
  wf^σ r = wf^τ r = ⊤
```

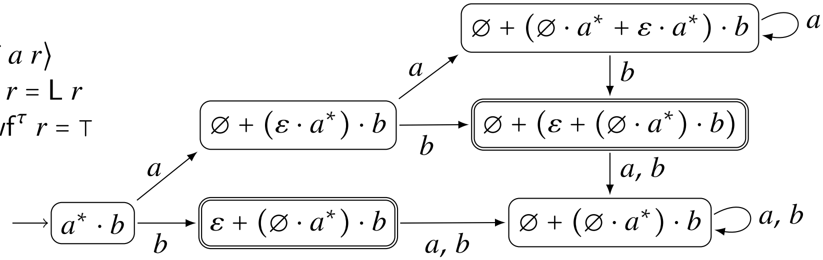


Figure 5.2: ACI-normalized derivative automaton for $a^* \cdot b$

A few points are worth mentioning here: First, D does not use the quotient type—it operates directly on canonical representatives and therefore can use structural equality for comparison (rather than ~). Second, the interpretations $\mathsf{D}_\sim$ and D yield structurally the same automata, although with different labels. Figure 5.2 shows the automaton produced by D for $a^* \cdot b$. This observation—which enables us to reuse the technically involved finiteness proof for $\mathsf{D}_\sim$ to discharge the finiteness assumption for D—relies crucially on our normalization function ⟨_⟩ being idempotent and well-behaved with respect to derivatives.

```
theorem ⟨⟨r⟩⟩ = ⟨r⟩
theorem ⟨δ a ⟨r⟩⟩ = ⟨δ a r⟩
```

The automaton from Figure 5.2 shows that the state labels still contain superfluous information, notably in the form of ∅s and εs. A coarser relation than ∼-equivalence, denoted ≈, addresses this concern. We omit the straightforward inductive definition of ≈, which cancels ∅s and εs where possible and takes the associativity of concatenation · into account. Coarseness ($[r]_\sim \subseteq [r]_\approx$) together with the finiteness assumption of $\mathsf{D}_\sim$ implies finiteness of equivalence classes of derivatives modulo ≈: $\mathsf{finite}\ \{[\mathsf{fold}\ \delta\ w\ r]_\approx \mid w :: \alpha\ list\}$.

As before, to avoid working with equivalence classes, we use a recursively defined ≈-normalization function $\langle\!\langle _ \rangle\!\rangle$ similar to $\langle _ \rangle$ (it corresponds to the *norm* function from the formalization by Krauss and Nipkow [76]). This intermediate normalization might seem harmless, but it is not clear anymore that the number of derivatives interspersed with normalization is finite[1]. We were not able to prove finiteness of such derivatives interspersed with $\langle\!\langle - \rangle\!\rangle$ (although we conjecture that it holds). Note that, unlike $\langle _ \rangle$, the normalization $\langle\!\langle _ \rangle\!\rangle$ (also ≈) is not well-behaved with respect to derivatives: for example, $\langle\!\langle \delta\ a\ \langle\!\langle ((a + \varepsilon) \cdot (a \cdot a)) \cdot b \rangle\!\rangle \rangle\!\rangle \neq \langle\!\langle \delta\ a\ (((a + \varepsilon) \cdot (a \cdot a)) \cdot b) \rangle\!\rangle$. The normalization would need to take the distributivity of · over + into account to prevent this inequality, but even with this addition a formal proof of well-behavedness seems difficult. Furthermore, our evaluation (Section 5.3) suggests that not too much energy should be invested in finding this proof. Thus, the following interpretation gives only a partial correctness result.

```
interpretation N : fin_coalg_D where
  Σ = as_list UNIV
  i r = ⟪r⟫
  o r = o r
  d a r = ⟪δ a r⟫
  L^σ r = L^τ r = L r
  wf^σ r = wf^τ r = ⊤
```

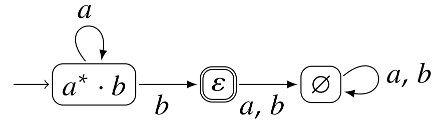


Figure 5.3: Normalized derivative automaton for $a^* \cdot b$

In practice, we did not find an input for which N would not terminate. For the example $a^* \cdot b$ it even yields the minimal automaton shown in Figure 5.3.

5.1.2 Partial Derivatives

Partial derivatives split certain +-constructors into sets of regular expressions, thus capturing ACI equivalence directly in the data structure. The automaton construction for a regular expression r starts with the singleton set $\{r\}$. More precisely, partial derivatives ∂ are defined recursively as follows.

[1] For example, using the terminating normalization function that does the same ACI simplifications as $\langle\!\langle - \rangle\!\rangle$, but additionally soundly rewrites $\varepsilon \cdot a^*$ to $a^* \cdot a^*$ for a fixed symbol a will result in an infinite number of derivatives when applied at intermediate steps to the initial expression a^*.

$$
\begin{array}{l}
\texttt{primrec}\ \partial :: \alpha \to \alpha\, RE \to \alpha\, RE\, set\ \texttt{where}\\
\quad \partial\ _\ \varnothing = \{\}\\
\quad \partial\ _\ \varepsilon = \{\}\\
\quad \partial\ a\ (\mathsf{A}\ b) = \texttt{if}\ a = b\ \texttt{then}\ \{\varepsilon\}\ \texttt{else}\ \{\}\\
\quad \partial\ a\ (r + s) = \partial\ a\ r \cup \partial\ a\ s\\
\quad \partial\ a\ (r \cdot s) = \partial\ a\ r \odot s \cup \texttt{if}\ o\ r\ \texttt{then}\ \partial\ a\ s\ \texttt{else}\ \{\}\\
\quad \partial\ a\ (r^*) = \partial\ a\ r \odot r^*
\end{array}
$$

Above, $R \odot s$ is used as a shorthand notation for $\{r \cdot s \mid r \in R\}$. The definition yields the characteristic property of partial derivatives by induction on r.

$$\{w \mid a \# w \in \mathsf{L}\ r\} = \bigcup_{s \in \partial\ a\ r} \mathsf{L}\ s$$

Following this characteristic property, we can interpret the locale $\mathsf{fin_coalg}_D$. The automaton constructed by P for our running example is shown in Figure 5.4.

$$
\begin{array}{l}
\texttt{interpretation}\ \mathsf{P} : \mathsf{fin_coalg}_D\ \texttt{where}\\
\quad \Sigma = \mathsf{as_list}\ \mathsf{UNIV}\\
\quad \mathsf{i}\ r = \{r\}\\
\quad \mathsf{o}\ R = \exists r \in R.\ o\ r\\
\quad \mathsf{d}\ a\ R = \bigcup_{r \in R} \delta\ a\ r\\
\quad \mathsf{L}^{\sigma}\ R = \bigcup_{r \in R} \mathsf{L}\ r\\
\quad \mathsf{L}^{\tau}\ r = \mathsf{L}\ r\\
\quad \mathsf{wf}^{\sigma}\ r = \mathsf{wf}^{\tau}\ r = \top
\end{array}
$$

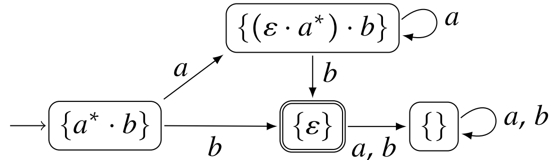


Figure 5.4: Partial derivative automaton for $a^* \cdot b$

The assumptions of coalg_D (inherited by $\mathsf{fin_coalg}_D$) are easy to discharge. Just as for Brzozowski derivatives, only the proof of finiteness of the reachable state space poses a challenge. We were able to reuse the proof by Wu *et al.* [119] who show finiteness when proving one direction of the Myhill-Nerode theorem. Compared with the proof of finiteness for the interpretation D, the formal reasoning about partial derivatives appears to be more succinct.

There is a direct connection between ∂ and δ that seems not to have been covered in the literature. It is best expressed in terms of a recursive function pset that translates derivatives to partial derivatives: $\mathsf{pset}\ (\delta\ a\ r) = \partial\ a\ r$.

$$
\begin{array}{l}
\texttt{primrec}\ \mathsf{pset} :: \alpha\, RE \to \alpha\, RE\, set\ \texttt{where}\\
\quad \mathsf{pset}\ \varnothing = \{\}\\
\quad \mathsf{pset}\ \varepsilon = \{\varepsilon\}\\
\quad \mathsf{pset}\ (\mathsf{A}\ a) = \{\mathsf{A}\ a\}\\
\quad \mathsf{pset}\ (r + s) = \mathsf{pset}\ r \cup \mathsf{pset}\ s\\
\quad \mathsf{pset}\ (r \cdot s) = \mathsf{pset}\ r \odot s\\
\quad \mathsf{pset}\ (r^*) = \{r^*\}
\end{array}
$$

A finite set R of regular expressions can be represented uniquely by a single regular expression $\sum R$, a sum ordered with respect to $\leq$. Hence, we have $\sum \mathsf{pset}\ (\delta\ a\ r) = \sum \partial\ a\ r$, meaning that we can devise a normalization function $⟪r⟫ = \sum \mathsf{pset}\ r$ that allows us to simulate partial derivatives while operating on plain regular expressions. Alternatively, $⟪_⟫$ can be defined using smart constructors (with sequentially matched equations):

```
primrec ⟪_⟫ :: α RE → α RE where
    ⟪∅⟫ = ∅
    ⟪ε⟫ = ε
    ⟪A a⟫ = A a
    ⟪r + s⟫ = ⟪r⟫ ⊞ ⟪s⟫
    ⟪r · s⟫ = ⟪r⟫ ⊡ s
    ⟪r*⟫ = r*

primrec ⊡ :: α RE → α RE → α RE where
    ∅ ⊡ r = ∅
    (r + s) ⊡ t = (r ⊡ s) ⊞ (s ⊡ t)
    r ⊡ s = s · t

primrec ⊞ :: α RE → α RE → α RE where
    ∅ ⊞ r = r
    r ⊞ ∅ = r
    (r + s) ⊞ t = r ⊞ (s ⊞ t)
    r ⊞ (s + t) = if r = s then s + t
        else if r ≤ s then r + (s + t)
            else s + (r ⊞ t)
    r ⊞ s = if r = s then r
        else if r ≤ s then r + s
            else s + r
```

This definition allows to contrast the implicit quotienting performed by partial derivatives with the qoutienting modulo ACI equivalence ($\sim$). They turn out to be incomparable: $⟪_⟫$ does not simplify the second argument of concatenation $\cdot$ and the argument of iteration *, but erases $\emptyset$s and uses left distributivity.

Finally, we obtain a last derivative-based interpretation using the characteristic property $⟪\delta\ a\ r⟫ = \sum(\partial\ a\ r)$ and reusing the proof of P's finiteness assumption.

```
interpretation PD : fin_coalg_D where
    Σ = as_list UNIV
    i r = r
    o r = o r
    d a r = ⟪δ a r⟫
    L^σ r = L^τ r = L r
    wf^σ r = wf^τ r = ⊤
```

Whenever P yields an automaton for r with states labeled with finite sets of regular expressions X_i, the automaton constructed by PD for r is structurally the same and has its states labeled with $\sum X_i$.

5.2 Marked Regular Expressions

One of the oldest methods for converting a regular expression into an automaton is based on the idea of identifying the states of the automaton with positions in the

regular expression. Both McNaughton and Yamada [82] and Glushkov [50] mark the atoms in a regular expression with numbers to identify positions uniquely. In this section, we formalize two recent reincarnations of this approach due to Fischer *et al.* [43] and Asperti [5, 6]. They are based on the realization that in a functional programming setting, it is most convenient to represent positions in a regular expression by marking some of its atoms. First we define an infrastructure for working with marked regular expressions. Then we define and relate both reincarnations in terms of this infrastructure.

Marked regular expressions are formalized by the following type synonym (where the value $\top$ denotes a marked atom).

$$\texttt{type_synonym}\ \alpha\ mRE = (bool \times \alpha)\ RE$$

We convert easily between RE and mRE with the help of map_RE, the map function on regular expressions. The language L_m of a marked regular expression is the set of words that start at some marked atom.

$$\begin{array}{l}
\texttt{definition}\ \pi_{mRE} :: \alpha\ mRE \to \alpha\ RE\ \texttt{where} \\
\quad \pi_{mRE} = \mathsf{map_RE}\ \mathsf{snd} \\
\texttt{definition}\ \iota_{mRE} :: \alpha\ RE \to \alpha\ mRE\ \texttt{where} \\
\quad \iota_{mRE} = \mathsf{map_RE}\ (\lambda r.\ (\bot, r))
\end{array}$$

$$\begin{array}{l}
\texttt{primrec}\ \mathsf{L}_m :: \alpha\ mRE \to \alpha\ lang\ \texttt{where} \\
\quad \mathsf{L}_m\ \varnothing = \{\} \\
\quad \mathsf{L}_m\ \varepsilon = \{\} \\
\quad \mathsf{L}_m\ (\mathsf{A}\ (m, a)) = \texttt{if}\ m\ \texttt{then}\ \{[a]\}\ \texttt{else}\ \{\} \\
\quad \mathsf{L}_m\ (r + s) = \mathsf{L}_m\ r \cup \mathsf{L}_m\ s \\
\quad \mathsf{L}_m\ (r \cdot s) = \mathsf{L}_m\ r \cdot \mathsf{L}\ (\pi_{mRE}\ s) \cup \mathsf{L}_m\ s \\
\quad \mathsf{L}_m\ (r^*) = \mathsf{L}_m\ r \cdot \mathsf{L}\ (\pi_{mRE}\ r)^*
\end{array}$$

The function final tests if some atom at the "end" of a regular expression is marked.

$$\begin{array}{l}
\texttt{primrec}\ \mathsf{final} :: \alpha\ mRE \to bool\ \texttt{where} \\
\quad \mathsf{final}\ \varnothing = \bot \\
\quad \mathsf{final}\ \varepsilon = \bot \\
\quad \mathsf{final}\ (\mathsf{A}\ (m, a)) = m \\
\quad \mathsf{final}\ (r + s) = \mathsf{final}\ r \lor \mathsf{final}\ s \\
\quad \mathsf{final}\ (r \cdot s) = \mathsf{final}\ s \lor o\ s \land \mathsf{final}\ r \\
\quad \mathsf{final}\ (r^*) = \mathsf{final}\ r
\end{array}$$

Marks are moved around a regular expression by two operations. The function read a r unmarks all atoms in r except a.

```
definition read :: α → α mRE → α mRE where
  read a = map_RE (λ(m, x). (m ∧ a = x, x))
```

Its characteristic lemma is that it restricts $\mathsf{L}_m\ r$ to words whose head is a.

```
theorem L_m (read a r) = {w ∈ L_m r | w ≠ [] ∧ hd w = a}
```

The function follow $m\ r$ moves all marks in r to the "next" atom, much like an ε-closure; the mark m is newly pushed in from the left.

```
primrec follow :: bool → α mRE → α mRE where
  follow m ∅ = ∅
  follow m ε = ε
  follow m (A (_, a)) = A (m, a)
  follow m (r + s) = follow m r + follow m s
  follow m (r · s) = follow m r · follow (final r ∨ m ∧ o r) s
  follow m (r*) = (follow (final r ∨ m) r)*
```

The characteristic lemma about follow shows that the marks are moved forward, thereby chopping off the first letter (in the generated language), and that the parameter m indicates whether every "first" atom should be marked:

```
theorem L_m (follow m r) = {tl w | w ∈ L_m r} ∪
                           (if m then L(π_mRE r) else {}) − {[]}
```

5.2.1 Mark After Atom

In the work of McNaughton-Yamada-Glushkov, the mark indicates which atom has just been read—the mark is located "after" the atom. Therefore the initial state is special because nothing has been read yet. Thus we express the states of the automaton as a pair of a boolean (⊤ means that nothing has been read yet) and a marked regular expression. The boolean can be viewed as a mark in front of the automaton. (Alternatively, one could work with an explicit start symbol in front of the regular expression.) We interpret the locale $\mathsf{fin_coalg}_D$ as follows.

```
interpretation A : fin_coalg_D where
  Σ = as_list UNIV
  i r = (⊤, ι_mRE r)
  o (m, r) = final r ∨ m ∧ o r
  d a (m, r) = (⊥, read a (follow m r))
  L^σ (m, r) = L_m(follow m r) ∪ if o (m, r) then {[]} else {}
  L^τ r = L r
  wf^σ (m, r) = wf^τ r = ⊤
```

The definition of A.d expresses that we first build the ε-closure starting from the marked atoms (via follow) and then read the next atom. With the characteristic lemmas about read and follow (and a few auxiliary lemmas), the locale assumptions are easily proved. This yields our first procedure based on marked regular expressions. The finiteness assumption is easily proved via the following lemma.

$$\texttt{theorem}\ \mathsf{fold}\ \mathsf{A.d}\ w\ (\mathsf{A.i}\ \ r) \in \{\top, \bot\} \times \mathsf{mrexps}\ r$$

Above, $\mathsf{mrexps} :: \alpha\, RE \to (\alpha\ mRE)\, set$ maps a regular expression to the finite set of all its marked variants, that is $\mathsf{mrexps}\ r = \{r' \mid \pi_{mRE}\ r' = r\}$; its actual recursive definition is straightforward and omitted.

Now we take a closer look at the work of Fischer *et al.* [43], which inspired the preceding formalization. They present a number of (not formally verified) matching algorithms on marked regular expressions in Haskell that follow McNaughton-Yamada-Glushkov. Their basic transition function is called shift.

```
primrec shift :: bool → α mRE → α → α mRE where
  shift _ ∅ _ = ∅
  shift _ ε _ = ε
  shift m (A (_, a)) c = A (m ∧ (a = c), a)
  shift m (r + s) c = shift m r c + shift m s c
  shift m (r · s) c = shift m r c · shift (final r ∨ m ∧ o r) s c
  shift m (r*) c = (shift (final r ∨ m) r c)*
```

A simple induction proves that their shift is our A.d.

$$\texttt{theorem}\ \mathsf{shift}\ m\ r\ c\ =\ \mathsf{read}\ c\ (\mathsf{follow}\ m\ r)$$

Thus we have verified their shift function. Fischer et al. optimize shift further, which is still quadratic due to the calls of the recursive functions final and o. They simply cache the values of final and o at all nodes of a regular expression by adding additional fields to each constructor. We have verified this optimization step as well. Until the end of this subsection, $\alpha\ mRE$ therefore will be not just an extension of the atoms of $\alpha\, RE$ as before, but an own datatype that includes the additional fields.

$$\texttt{datatype}\ \alpha\ mRE = \hat{\varnothing} \mid \hat{\varepsilon} \mid \hat{\mathsf{A}}^{bool}\ \alpha \mid \alpha\ mRE +^{bool}_{bool} \alpha\ mRE \mid \alpha\ mRE \cdot^{bool}_{bool} \alpha\ mRE \mid (\alpha\ mRE)^{*,bool}$$

In the above definition the superscripted Booleans cache the values of final, while the subscripted ones cache the values of o. We can now redefine those operations to access the cached values directly instead of descending recursively.

```
primrec final :: α mRE → bool where
  final ∅̂ = ⊥
  final ε̂ = ⊥
  final (Â^m a) = m
  final (r +^m_b s) = m
  final (r ·^m_b s) = m
  final (r^{*,m}) = m

primrec o :: α mRE → bool where
  o ∅̂ = ⊥
  o ε̂ = ⊤
  o (Â^m a) = ⊥
  o (r +^m_b s) = b
  o (r ·^m_b s) = b
  o (r^{*,m}) = ⊤
```

Moreover, we define smart constructors $\hat{+}$, $\hat{\cdot}$, and $_^{\hat{*}}$ for $+$, $\cdot$, and $_^*$ that combine the cached values in a meaningful way and use those to transform plain regular expressions meaningfully into caching expressions via the function ι_{mRE}. The inverse function π_{mRE} strips the cached values to obtain a plain marked regular expression.

```
definition +̂ :: α mRE → α mRE → α mRE where
  r +̂ s = r +^{final r ∨ final s}_{o r ∨ o s} s

definition ·̂ :: α mRE → α mRE → α mRE where
  r ·̂ s = r ·^{final r ∧ o s ∨ final s}_{o r ∧ o s} s

definition _^*̂ :: α mRE → α mRE where
  r^*̂ = r^{*, final r}
```

```
primrec ι_mRE :: α RE → α mRE where
  ι_mRE ∅ = ∅̂
  ι_mRE ε = ε̂
  ι_mRE (A a) = Â^⊥ a
  ι_mRE (r + s) = ι_mRE r +̂ ι_mRE s
  ι_mRE (r · s) = ι_mRE r ·̂ ι_mRE s
  ι_mRE (r^*) = (ι_mRE r)^*̂

primrec π_mRE :: α mRE → (bool × α) RE where
  π_mRE ∅̂ = ∅
  π_mRE ε̂ = ε
  π_mRE (Â^m a) = A (m, a)
  π_mRE (r +^m_b s) = π_mRE r + π_mRE s
  π_mRE (r ·^m_b s) = π_mRE r · π_mRE s
  π_mRE (r^{*,m}) = (π_mRE r)^*
```

What meaningful means is specified by a wellformedness predicate wf_{mRE} that checks that the cached values are actually equal to the earlier recursive computation. All constructors that carry a hat ˆ produce wellformed expressions, when applied to wellformed arguments. Note that the functions final and o on the right hand sides of the following equations are the ones defined on plain marked regular expressions (without any caching).

```
fun wf_mRE :: α mRE → bool where
  wf_mRE (r +^m_b s) = wf_mRE r ∧ wf_mRE s ∧
      let t = π_mRE r + π_mRE s in m = final t ∧ b = o t
  wf_mRE (r ·^m_b s) = wf_mRE r ∧ wf_mRE s ∧
      let t = π_mRE r · π_mRE s in m = final t ∧ b = o t
  wf_mRE (r^{*,m}) = wf_mRE r ∧ m = final (π_mRE r)
  wf_mRE _ = ⊤
```

Finally, the transition function shift is basically the same as in the case of plain marked regular expressions, except for the usage of smart constructors and the optimized final and o functions.

$$
\begin{array}{l}
\texttt{primrec}\ \mathsf{shift} :: bool \to \alpha\ mRE \to \alpha \to \alpha\ mRE\ \texttt{where} \\
\quad \mathsf{shift}\ _\ \hat{\varnothing}\ _ = \hat{\varnothing} \\
\quad \mathsf{shift}\ _\ \hat{\varepsilon}\ _ = \hat{\varepsilon} \\
\quad \mathsf{shift}\ m\ (\hat{\mathsf{A}}^{-}\ a)\ c = \hat{\mathsf{A}}^{m \wedge (a=c)}\ a \\
\quad \mathsf{shift}\ m\ (r +_{-}^{-} s)\ c = \mathsf{shift}\ m\ r\ c\ \hat{+}\ \mathsf{shift}\ m\ s\ c \\
\quad \mathsf{shift}\ m\ (r \cdot_{-}^{-} s)\ c = \mathsf{shift}\ m\ r\ c\ \hat{\cdot}\ \mathsf{shift}\ (\mathsf{final}\ r \vee m \wedge o\ r)\ s\ c \\
\quad \mathsf{shift}\ m\ (r^{*\prime -})\ c = (\mathsf{shift}\ (\mathsf{final}\ r \vee m)\ r\ c)^{\hat{*}}
\end{array}
$$

Overall, this yields another interpretation which is very similar to A (also proofwise), especially since we have overloaded the notation. It also constitutes our first usage of the wellformedness check. However, all cached marked regular expressions that the equivalence checker will see, will be wellformed by construction (since ι_{mRE} produces wellformed equations and shift preserves wellformedness). Therefore, the function wf_{mRE} will never be executed in the generated code.

$$
\begin{array}{l}
\texttt{interpretation}\ \mathsf{A}_2 : \mathsf{fin_coalg}_D\ \texttt{where} \\
\quad \Sigma = \mathsf{as_list\ UNIV} \\
\quad \mathsf{i}\ r = (\top, \iota_{mRE}\ r) \\
\quad \mathsf{o}\ (m, r) = \mathsf{final}\ r \vee m \wedge o\ r \\
\quad \mathsf{d}\ a\ (m, r) = (\bot, \mathsf{shift}\ m\ r\ a) \\
\quad \mathsf{L}^{\sigma}\ (m, r) = \mathsf{A.L}^{\sigma}\ (m, \pi_{mRE}\ r) \\
\quad \mathsf{L}^{\tau}\ r = \mathsf{L}\ r \\
\quad \mathsf{wf}^{\sigma}\ (m, r) = \mathsf{wf}_{mRE}\ r \\
\quad \mathsf{wf}^{\tau}\ r = \top
\end{array}
$$

5.2.2 Mark Before Atom

Instead of imagining the mark to be after an atom, it can also be viewed to be in front of it; in that case it marks possible next atoms. This is somewhat dual to the McNaughton-Yamada-Glushkov construction. It leads to the following interpretation of our locale. (Note that we are again using the non-caching expressions.)

$$
\begin{array}{l}
\texttt{interpretation}\ \mathsf{B} : \mathsf{fin_coalg}_D\ \texttt{where} \\
\quad \Sigma = \mathsf{as_list\ UNIV} \\
\quad \mathsf{i}\ r = (\mathsf{follow}\ \top\ (\iota_{mRE}\ r), o\ r) \\
\quad \mathsf{o}\ (r, m) = m \\
\quad \mathsf{d}\ a\ (r, m) = \texttt{let}\ r' = \mathsf{read}\ a\ r\ \texttt{in}\ (\mathsf{follow}\ \bot\ r', \mathsf{final}\ r') \\
\quad \mathsf{L}^{\sigma}\ (r, m) = \mathsf{L}_m\ r \cup \texttt{if}\ m\ \texttt{then}\ \{[]\}\ \texttt{else}\ \{\} \\
\quad \mathsf{L}^{\tau}\ r = \mathsf{L}\ r \\
\quad \mathsf{wf}^{\sigma}\ (r, m) = \mathsf{wf}^{\tau}\ r = \top
\end{array}
$$

The definition of B.d expresses that we first read an atom and then build the ε-closure. The assumptions of coalg_D and $\mathsf{fin_coalg}_D$ are proved easily just like in the previous interpretations with marked regular expressions.

The interesting point is that this happens to be the algorithm formalized by Asperti [5]. Although he says that he has formalized McNaughton-Yamada, he actually formalized the dual algorithm. This is not easy to see because Asperti's formalization is considerably more involved than ours, with many auxiliary functions. Strictly speaking, his algorithm is a variation of ours that produces the same automata. The complete proof of this fact can be found elsewhere [57]. Because of the size of Asperti's formalization, we omit it here. However, we can take a step towards his formulation and merge follow and read into one function move, the analogue of his homonymous function.

$$
\begin{array}{l}
\texttt{primrec}\ \mathsf{move} :: \alpha \to \alpha\ mRE \to bool \to \alpha\ mRE\ \texttt{where} \\
\quad \mathsf{move}\ _\ \varnothing\ _ = \varnothing \\
\quad \mathsf{move}\ _\ \varepsilon\ _ = \varepsilon \\
\quad \mathsf{move}\ _\ (\mathsf{A}\ (_, a))\ m = \mathsf{A}\ (m, a) \\
\quad \mathsf{move}\ c\ (r + s)\ m = \mathsf{move}\ c\ r\ m + \mathsf{move}\ c\ s\ m \\
\quad \mathsf{move}\ c\ (r \cdot s)\ m = \mathsf{move}\ c\ r\ m \cdot \mathsf{move}\ c\ s\ (c \in \mathsf{final1}\ r \vee m \wedge o\ r) \\
\quad \mathsf{move}\ c\ (r^*)\ m = (\mathsf{move}\ c\ r\ (c \in \mathsf{final1}\ r \vee m))^*
\end{array}
$$

Here, final1 is an auxiliary recursive function (not shown here) with the characteristic property that $c \in \mathsf{final1}\ r = \mathsf{final}\ (\mathsf{read}\ c\ r)$. A simple induction proves that move combines follow and read as in B.d.

$$\texttt{theorem}\ \mathsf{move}\ c\ r\ m\ =\ \mathsf{follow}\ m\ (\mathsf{read}\ c\ r)$$

The function move has quadratic complexity for the same reason as shift. Unfortunately, it cannot be made linear with the same ease as for shift. The problem is that we need to cache the value of $c \in \mathsf{final1}\ r$ in the previous step, before we know c. We solve this by caching the whole set $\mathsf{final1}\ r$ of letters. In the worst case, the whole alphabet must be stored in certain inner nodes. However, for an alphabet of fixed size this guarantees linear time complexity. This optimization constitutes a last interpretation B_2 in the spirit of A_2. We again start by extending the datatype of regular expression with additional fields for caching.

$$
\begin{array}{l}
\texttt{datatype}\ \alpha\ mRE = \hat{\varnothing} \mid \hat{\varepsilon} \mid \hat{\mathsf{A}}^{bool}\ \alpha \\
\qquad\qquad\qquad \mid \alpha\ mRE +^{\alpha\ set}_{bool} \alpha\ mRE \mid \alpha\ mRE \cdot^{\alpha\ set}_{bool} \alpha\ mRE \mid (\alpha\ mRE)^{*,\alpha\ set}
\end{array}
$$

In the above definition the superscripted values cache sets of letters produced by final1 (except for the $\hat{\mathsf{A}}$ constructor), while the subscripted ones cache the values of o. As for A_2, we redefine those operations to rely on the cached values.

```
primrec final1 :: α mRE → α → bool where
  final1 ∅̂ = {}
  final1 ε̂ = {}
  final1 (Â^m a) = if m then {a} else {}
  final1 (r +^M_b s) = M
  final1 (r ·^M_b s) = M
  final1 (r^{*,M}) = M
```

```
primrec o :: α mRE → bool where
  o ∅̂ = ⊥
  o ε̂ = ⊤
  o (Â^m a) = ⊥
  o (r +^M_b s) = b
  o (r ·^M_b s) = b
  o (r^{*,M}) = ⊤
```

The key difference to A_2 lies in the following definitions of the smart constructors $\hat{+}$, $\hat{\cdot}$, and $_^{\hat{*}}$ for $+$, $\cdot$, and $_^{*}$ that combine the cached values. The casts ι_{mRE} and π_{mRE} between the different kinds of regular expressions and the wellformedness predicate wf_{mRE} are then using the smart constructors abstraction and therefore (almost) identical to the equivalent operations used in the A_2 interpretation.

```
definition +̂ :: α mRE → α mRE → α mRE where
  r +̂ s = r +^{final1 r ∪ final1 s}_{o r ∨ o s} s

definition ·̂ :: α mRE → α mRE → α mRE where
  r ·̂ s = r ·^{(if o s then final1 r else {}) ∪ final1 s}_{o r ∧ o s} s

definition _^*̂ :: α mRE → α mRE where
  r^*̂ = r^{*, final1 r}
```

```
primrec ι_mRE :: α RE → α mRE where
  ι_mRE ∅ = ∅̂
  ι_mRE ε = ε̂
  ι_mRE (A a) = Â^⊥ a
  ι_mRE (r + s) = ι_mRE r +̂ ι_mRE s
  ι_mRE (r · s) = ι_mRE r ·̂ ι_mRE s
  ι_mRE (r^*) = (ι_mRE r)^*̂
```

```
primrec π_mRE :: α mRE → (bool × α) RE where
  π_mRE ∅̂ = ∅
  π_mRE ε̂ = ε
  π_mRE (Â^m a) = A (m, a)
  π_mRE (r +^M_b s) = π_mRE r + π_mRE s
  π_mRE (r ·^M_b s) = π_mRE r · π_mRE s
  π_mRE (r^{*,M}) = (π_mRE r)^*
```

```
fun wf_mRE :: α mRE → bool where
  wf_mRE (r +^M_b s) = wf_mRE r ∧ wf_mRE s ∧
      let t = π_mRE r + π_mRE s in M = final1 t ∧ b = o t
  wf_mRE (r ·^M_b s) = wf_mRE r ∧ wf_mRE s ∧
      let t = π_mRE r · π_mRE s in M = final1 t ∧ b = o t
  wf_mRE (r^{*,M}) = wf_mRE r ∧ M = final1 (π_mRE r)
  wf_mRE _ = ⊤
```

Finally, our simplified transition function move is basically the same as in the case of plain marked regular expressions, except for its usage of smart constructors and the optimized final1 and o functions.

```
primrec move :: α → α mRE → bool → α mRE where
  move _ ∅̂ _ = ∅̂
```

$$
\begin{aligned}
&\mathsf{move}\ _\ \hat{\varepsilon}\ _ = \hat{\varepsilon}\\
&\mathsf{move}\ c\ (\hat{\mathsf{A}}\text{-}\ a)\ m = \hat{\mathsf{A}}^{m}\ a\\
&\mathsf{move}\ c\ (r \mathbin{\underline{+}} s)\ m = \mathsf{move}\ c\ r\ m \mathbin{\hat{+}} \mathsf{move}\ c\ s\ m\\
&\mathsf{move}\ c\ (r \mathbin{\underline{\cdot}} s)\ m = \mathsf{move}\ c\ r\ m \mathbin{\hat{\cdot}} \mathsf{move}\ c\ s\ (c \in \mathsf{final1}\ r \lor m \land o\ r)\\
&\mathsf{move}\ c\ (r^{*\prime}\text{-})\ m = (\mathsf{move}\ c\ r\ (c \in \mathsf{final1}\ r \lor m))^{\hat{*}}
\end{aligned}
$$

The same applies to the functions follow and final, which were not needed in the A_2 interpretation, but are required for the parameter i of the forthcoming B_2. We omit the straightforward equations for those functions. Those were all the ingredients we need for the actual interpretation. The locale's assumptions are (as it is always the case with marked regular expressions) easy to discharge, especially when reusing the proofs that were already carried out for the interpretation B.

$\texttt{interpretation}\ \mathsf{B}_2 : \mathsf{fin_coalg}_D\ \texttt{where}$

$$
\begin{aligned}
&\Sigma = \mathsf{as_list\ UNIV}\\
&\mathsf{i}\ r = (\mathsf{follow}\ \top\ (\iota_{mRE}\ r), o\ r)\\
&\mathsf{o}\ (r, m) = m\\
&\mathsf{d}\ a\ (r, m) = (\mathsf{move}\ a\ r\ \bot, a \in \mathsf{final1}\ r)\\
&\mathsf{L}^{\sigma}\ (r, m) = \mathsf{B.L}^{\sigma}\ (\pi_{mRE}\ r, m)\\
&\mathsf{L}^{\tau}\ r = \mathsf{L}\ r\\
&\mathsf{wf}^{\sigma}\ (r, m) = \mathsf{wf}_{mRE}\ r\\
&\mathsf{wf}^{\tau}\ r = \top
\end{aligned}
$$

Even for a fixed alphabet, the actual move function from Asperti's formalization has quadratic complexity when faced with a tower of stars: each recursive call of move can trigger a call of a function eclose, which has linear complexity. Asperti aimed for compact proofs, not maximal efficiency.

5.2.3 Comparison

The two constructions may look similar, but they do not produce isomorphic automata. Considering our running example, we display the mark by a "•" before or after the atom. The two resulting automata are shown in Figure 5.5. There are special states that cannot be denoted by marking atoms only: $\bullet r$ in A's automaton is the completely unmarked regular expression that is the initial state and $r\bullet$ in B's automaton is a final state.

It turns out that the "before" automaton is a homomorphic image of the "after" automaton. To verify this we specify the homomorphism $\varphi\,(m, r) = (\mathsf{follow}\ m\ r, \mathsf{A.o}\ (m, r))$ and prove that it preserves initial states and commutes with the transition function.

```
theorem φ (A.i r) = B.i r
theorem φ (A.d a s) = B.d a (φ s)
theorem φ (fold A.d w s) = fold B.d w (φ s)
```

A direct consequence is that Asperti's "before" construction always generates automata with at most as many states as the McNaughton-Yamada-Glushkov construction. Formally, in the context of locale coalg_D we have defined an executable computation of the reachable state space $\{\mathsf{fold}\ \mathsf{d}\ w\ (\mathsf{i}\ r) \mid w \in (\mathsf{set}\ \Sigma)^*\}$ of the automaton as follows (where $\mathsf{the}\ (\mathsf{Some}\ x) = x$) and proved the aforementioned cardinality relation (where |_| is the cardinality of a set).

```
definition (in coalg_D) reachable :: σ → σ set where
  reachable r = snd (the (while (λ(ws, _). ws = [])
    (λ(s # ws, P). fold (λa (ws, P).
      let s′ = d a s in if s′ ∈ P then (ws, P) else (s′ # ws, P ∪ {s′})))
    Σ ([i r], {i r})))

theorem |B.reachable r| ≤ |A.reachable r|
```

In early drafts of this paper, we only conjectured the above statement and unsuccessfully tried to refute it with Isabelle's Quickcheck facility [27]. Later, Helmut Seidl has communicated an informal proof using the above homomorphism to us.

Let us abbreviate the statement of the above theorem to $n_b \le n_a$. One may think that n_a is only slightly larger than n_b, but it seems that n_b and n_a are more than a constant summand apart: for a two-element alphabet Quickcheck could refute $n_a \le n_b + k$ even for $k = 100$.

5.3 Empirical Comparison

We compare the efficiency with respect to both matching and deciding equivalence of the Standard ML code generated from eight described interpretations: ~-normalized derivatives (D), ≈-normalized derivatives (N), partial derivatives (P), derivatives simulating partial derivatives (PD), mark "after" atom (A), mark "after" atom with caching (A_2), mark "before" atom (B), and mark "before" atom with caching (B_2). The interpretation using the quotient type for derivatives ($\mathsf{D}_\sim$) is not

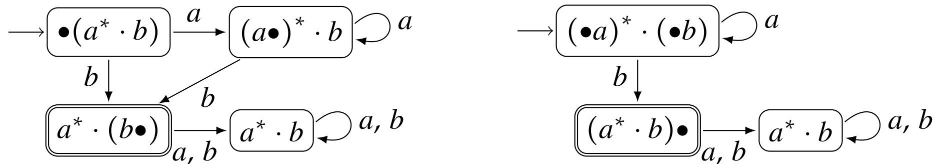


Figure 5.5: Marked regular expression automata (A left, B right) for $a^* \cdot b$

in this list, as it is clearly superseded by D. The results of the evaluation, performed on an Intel Core i7-2760QM machine with 8 GB of RAM, are shown in Figure 5.6. Solid lines depict the four derivative-based algorithms. Dashed lines are used for the algorithms based on marked regular expressions.

The first two tests, MATCH-R and MATCH-L, measure the time required to match the word a^n against the regular expression $(a + \varepsilon)^n \cdot a^n$—a standard benchmark also used by Fischer et al. [43]. The difference between the two tests is the definition of r^n. MATCH-R defines it as the n-fold concatenation associated to the right: $r^4 = r \cdot (r \cdot (r \cdot r))$, whereas MATCH-L associates to the left: $r^4 = ((r \cdot r) \cdot r) \cdot r$. In both tests, marked regular expressions outperform derivatives by far. The normalization performed by the derivative-based approaches (required to obtain a finite number of states for the equivalence check) decelerates the computation of the next state. Marked regular expressions benefit from a fast next state computation. The test MATCH-L exhibits the quadratic nature of the unoptimized matchers A and B (their curves are almost identical and therefore hard to distinguish in Figure 5.6). In contrast, A_2 and B_2 perform equally well in both tests, A_2 being approximately 1.5 times faster due to lighter cache annotations.

The next test goes back to Antimirov [2]: We measure the time (with a timeout of ten seconds) for proving the equivalence of a^* and $(a^0 + \ldots + a^{n-1}) \cdot (a^n)^*$. Again two tests, EQ-R and EQ-L, distinguish the associativity of concatenation in r^n. Here, the derivative-based equivalence checkers (except for D) perform better then the ones based on marked regular expressions. In particular, both version of partial derivatives, P and PD, outperform N—since this example was crafted by Antimirov to demonstrate the strength of partial derivatives, this is not wholly unexpected. Comparing EQ-R and EQ-L, the associativity barely influences the runtime.

Finally, to avoid bias towards a particular algorithm, we have devised the randomized test EQ-RND. There we measure the average time (with a timeout of ten seconds) to prove the equivalence of r with itself for 100 randomly generated expressions with n inner nodes ($+$, $\cdot$, or *). Proving $r \equiv r$ is of course a trivial task, but our algorithms do not stop the exploration when the bisimulation is extended with a new pair of two equal states. This optimization, which is a must for any practical algorithm, is the first step towards the rewarding usage of bisimulation up to equivalence (or even up to congruence) [21]. Without any such optimization, the task of proving $r \equiv r$ amounts to enumerating all derivatives of r, which is exactly what we want to compare. To generate random regular expression we use the infrastructure of SpecCheck [101]—a Quickcheck clone for Isabelle/ML. For computing the average, a timeout counts as 10 second (although the actual computation would likely have taken longer)—an approximation that skews the curves to converge to the margin of 10 seconds. We stopped measuring a method for increasing n when the average approached 5 seconds.

The results of EQ-RND are summarized as follows: $\mathsf{D} \gg \mathsf{N} \gg \mathsf{P}, \mathsf{PD} \gg \mathsf{A}, \mathsf{A}_2, \mathsf{B}, \mathsf{B}_2$, where $X \gg Y$ means that Y is an order of magnitude faster than X. The algorithm P

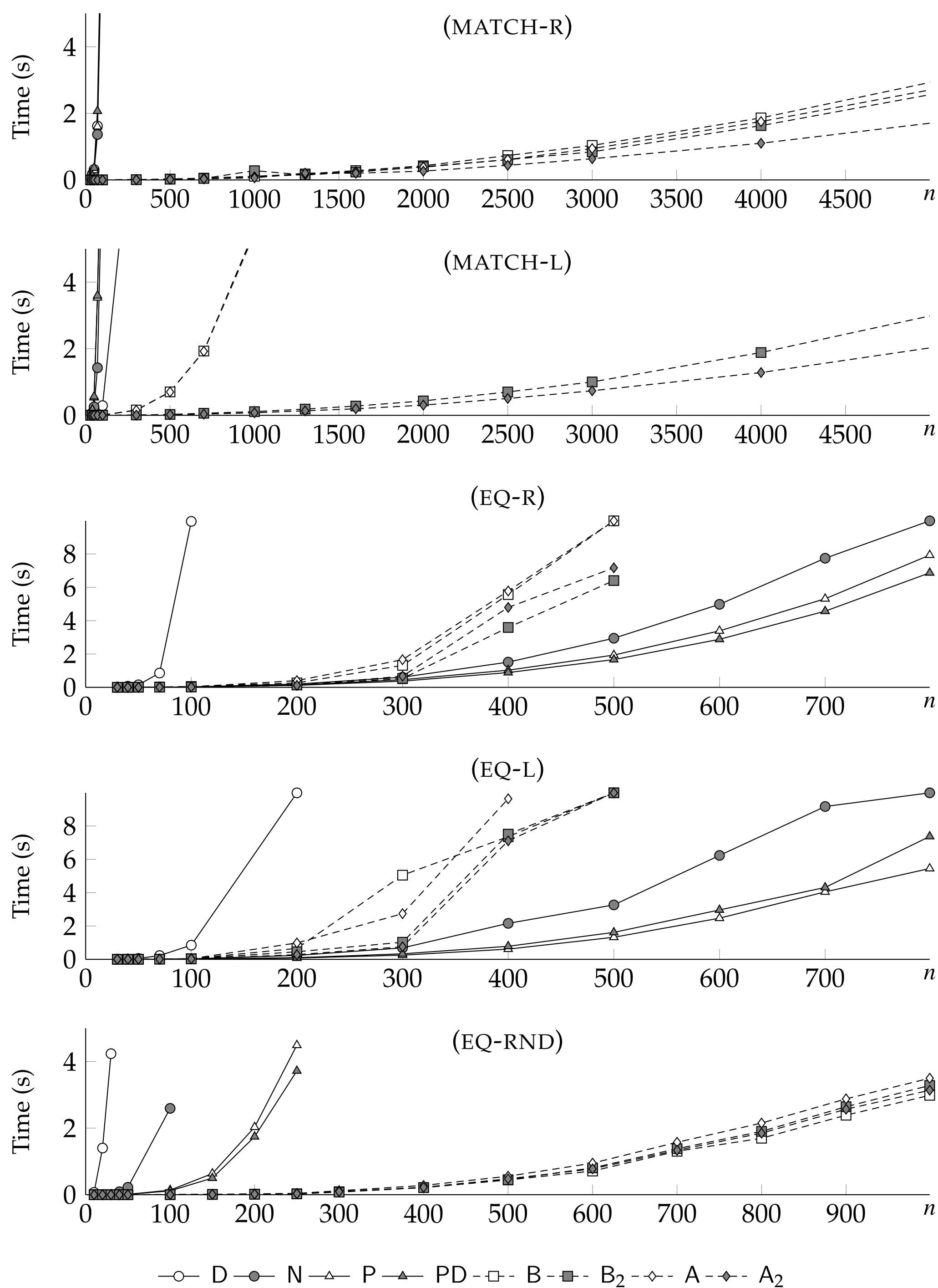


Figure 5.6: Evaluation results

is noticeably slower than PD—avoiding sets reduces the overhead. Among A, A_2, B, B_2, Asperti's unoptimized algorithm B performs best by a narrow margin. Regular expressions where the caching overhead pays off are rare and therefore not visible in the randomized test results. The same holds for expressions where B produces much smaller automata than A (such as the counterexample to $n_a \leq n_b + 100$ from Subsection 5.2.3).

Our evaluation shows that A_2 is the best choice for matching. For equivalence checking, the winner is not as clear cut: B (especially when applied to normalized input to avoid quadratic runtime without caching) and PD seem to be the best choices.

5.4 Extensions

Brzozowski's derivatives are easily extendable to regular expressions intersection and negation—indeed Brzozowski performed this extension already in his original work [25]. The number of such extended derivatives is still finite when quotiented modulo ACI.

In the next chapter we consider regular expressions further extended with projection. This extension caters for a simple semantics-preserving translation of formulas of different monadic second-order logics on finite words into regular expressions (Chapter 7).

Extending partial derivatives with intersection and negation is more involved [29]. An additional layer of sets must be used for intersections. With other words the states of our automaton would then be sets of sets of regular expressions. In Section 5.3, we have seen that already one layer of sets incurs some overhead. Hence, the view on partial derivatives as derivatives followed by some normalization is expected to be even more profitable for the extension—indeed in the next chapter we will take this route. In contrast, extending partial derivatives with projection is an easy exercise.

It is unclear how to extend marked regular expressions to handle negation and intersection. The number of possible markings for a regular expression of alphabetic width n is 2^n. However, there exist regular expressions of alphabetic width n using intersection, whose minimal automata have 2^{2^n} states [47].

Giving one such strange expressions–
Sullen, stupid, pert expressions.

— Lewis Carroll, *Hiawatha's Photographing* (1869)

Chapter 6

Π-Extended Regular Expressions

So far we have considered plain regular expressions and different decision procedures for their equivalence. In this chapter, we alter the input structures: we consider extended regular expressions (with intersection and complement) and further extend them by an additional operation: *projection* Π.

In Chapter 7 MSO formulas will be translated into regular expressions such that encodings of models of a formula correspond exactly to words in the regular language. Thereby, equivalence of formulas is reduced to the equivalence of regular expressions. Regular expressions extended with intersection and complement will allow us to encode Boolean operators on formulas in a straightforward fashion. The projection Π will play the crucial role of encoding existential quantifiers.

This chapter therefore constitutes an important interlude on our journey from regular expressions to logical formulas. The extension of regular expressions with projection is situated somewhere in between. The chapter is largely based on our ICFP 2013 publication [113] and its extended version [115].

6.1 Syntax and Semantics

Π-*extended* regular expressions (to distinguish them from mere extended regular expressions) are defined by extending the recursive data type $\alpha\, RE$ as follows.

$$\texttt{datatype}\ \alpha\, RE = \varnothing \mid \varepsilon \mid \mathsf{A}\ \alpha \mid \alpha\, RE + \alpha\, RE \mid \alpha\, RE \cdot \alpha\, RE \mid (\alpha\, RE)^* \\ \mid \alpha\, RE \cap \alpha\, RE \mid \neg\, (\alpha\, RE) \mid \Pi\, (\alpha\, RE)$$

Note that much of the time we will omit the "Π-extended" and simply speak of regular expressions if there is no danger of confusion.

We assume that type α is partitioned into a family of alphabets Σ_n that depend on a natural number n and there is a function $\pi :: \Sigma_{n+1} \to \Sigma_n$ that translates between

the different alphabets.[1]. In our later application of translating MSO formulas into regular expressions, n will represent the number of free variables of the translated formula. For now Σ_n and π are just parameters of our setup.

We focus on wellformed regular expressions where all atoms come from the same alphabet Σ_n. This will guarantee that the language of such a wellformed expression is a subset of Σ_n^*. The projection operation complicates wellformedness a little. Because projection is meant to encode existential quantifiers, projection should transform a regular expression over Σ_{n+1} into a regular expression over Σ_n, just as the existential quantifier transforms a formula with $n+1$ free variables into a formula with n free variables. Thus projection changes the alphabet. Wellformedness is defined recursively. We call a regular expression r *n-wellformed* if wf n r holds.

```
primrec wf :: nat → α RE → bool where
  wf n ∅ = ⊤
  wf n ε = ⊤
  wf n (A a) = a ∈ Σn
  wf n (r + s) = wf n r ∧ wf n s
  wf n (r · s) = wf n r ∧ wf n s
  wf n (r*) = wf n r
  wf n (r ∩ s) = wf n r ∧ wf n s
  wf n (¬ r) = wf n r
  wf n (Π r) = wf (n + 1) r
```

Note that in a system with dependent types, the parameter n could be a dependent parameter of the recursive data type, whereas the convention in the simply typed Isabelle/HOL is to use the predicate as a precondition if necessary.

The *language* L of a regular expression is defined as usual, except for the equations for complement and projection. For an n-wellformed regular expression the definition yields a subset of Σ_n^*.

```
primrec L :: nat → α RE → α list set where
  L n ∅ = {}
  L n ε = {[]}
  L n (A a) = {a}
  L n (r + s) = L n r ∪ L n s
  L n (r · s) = L n r · L n s
  L n (r*) = (L n r)*
  L n (r ∩ s) = L n r ∩ L n s
  L n (¬ r) = Σn* ∖ L n r
  L n (Π r) = map π • L (n + 1) r
```

[1]Due to Isabelle's lack of dependent types the type of π is $\alpha \to \alpha$. The more refined type $\Sigma_{n+1} \to \Sigma_n$ is realized via a locale that relates its parameters π and Σ with the assumption $\pi \bullet \Sigma_{n+1} \subseteq \Sigma_n$

The first unusual point is the parametrization with n. It expresses that we expect a regular expression over Σ_n and is necessary for the definition $\mathsf{L}\ n\ (\neg\ r) = \Sigma_n^* \setminus \mathsf{L}\ n\ r$.

The definition $\mathsf{L}\ n\ (\Pi\ r) = \mathsf{map}\ \pi \bullet \mathsf{L}\ (n+1)\ r$ is parameterized by the fixed parameter $\pi :: \Sigma_{n+1} \to \Sigma_n$. The projection Π denotes the homomorphic image under π. In more detail: map lifts π homomorphically to words (lists), and $\bullet$ lifts it to sets of words. Therefore Π transforms a language over Σ_{n+1} into a language over Σ_n.

To understand the "projection" terminology, it is helpful to think of elements of Σ_n as lists of fixed length n over some alphabet Σ and of π as the tail function on lists that drops the first element of the list. A word over Σ_n is then a list of lists. We will use this instantiation for π and Σ in Chapter 7.

6.2 Decision Procedure

Now we turn our attention to deciding equivalence of Π-extended regular expressions. Therefore we extend the empty word acceptance test and Brzozowski derivatives to the newly added cases.

primrec $o :: \alpha\, RE \to bool$ where

$$o\ \emptyset = \bot$$
$$o\ \varepsilon = \top$$
$$o\ (\mathsf{A}\ a) = \bot$$
$$o\ (r + s) = o\ r \lor o\ s$$
$$o\ (r \cdot s) = o\ r \land o\ s$$
$$o\ (r^*) = \top$$
$$o\ (r \cap s) = o\ r \land o\ s$$
$$o\ (\neg\ r) = \neg o\ r$$
$$o\ (\Pi\ r) = o\ r$$

primrec $\delta :: \alpha \to \alpha\, RE \to \alpha\, RE$ where

$$\delta\ a\ \emptyset = \emptyset$$
$$\delta\ a\ \varepsilon = \emptyset$$
$$\delta\ a\ (\mathsf{A}\ b) = \mathtt{if}\ a = b\ \mathtt{then}\ \varepsilon\ \mathtt{else}\ \emptyset$$
$$\delta\ a\ (r + s) = \delta\ a\ r + \delta\ a\ s$$
$$\delta\ a\ (r \cdot s) = \delta\ a\ r \cdot s + \mathtt{if}\ o\ r\ \mathtt{then}\ \delta\ a\ s\ \mathtt{else}\ \emptyset$$
$$\delta\ a\ (r^*) = \delta\ a\ r \cdot r^*$$
$$\delta\ a\ (r \cap s) = \delta\ a\ r \cap \delta\ a\ s$$
$$\delta\ a\ (\neg\ r) = \neg\ \delta\ a\ r$$
$$\delta\ a\ (\Pi\ r) = \Pi\ (\textstyle\bigoplus_{b \in \pi^- a} \delta\ b\ r)$$

The projection case introduced some new syntax that deserves some explanation. The preimage π^- applied to a letter $a \in \Sigma_n$ denotes the set $\{b \in \Sigma_{n+1} \mid \pi\ b = a\}$. Our alphabets Σ_n are finite for each n, hence so is the preimage of a letter. The summation $\bigoplus$ over a finite set denotes the iterated application of the $+$-constructor of regular expressions. Summation over the empty set is defined as $\emptyset$.

Since the projection acts homomorphically on words, it is clear that the derivative of $\Pi\ r$ with respect to a letter a can be expressed as a projection of derivatives of r. The concrete definition is a consequence of the following identity of left quotients for $a \in \Sigma_n$ and $A \subseteq \Sigma_{n+1}^*$:

$$\{w \mid a \# w \in \mathsf{map}\ \pi \bullet A\} = \mathsf{map}\ \pi \bullet \bigcup_{b \in \pi^- a} \{w \mid b \# w \in A\}$$

The characteristic properties of o and δ follow by structural induction.

theorem $o\ r \leftrightarrow [\,] \in \mathsf{L}\ n\ r$
theorem $\mathsf{wf}\ n\ r \longrightarrow \mathsf{wf}\ n\ (\delta\ a\ r)$
theorem $\mathsf{wf}\ n\ r \longrightarrow \mathsf{L}\ n\ (\delta\ a\ r) = \{w \mid a \# w \in \mathsf{L}\ n\ r\}$

As before, the number of distinct word derivatives modulo ACI of + is finite.

theorem finite $\{\langle \mathsf{fold}\ \delta\ w\ r\rangle \mid w :: \alpha\ list\}$

The function $\langle_\rangle :: \alpha\ RE \rightarrow \alpha\ RE$ extends the ACI normalization function from Chapter 5 for the cases of intersection, complement, and projection as shown below. The other cases remain unchanged (including the smart constructor $\oplus$ for union).

$$\langle r \cap s\rangle = \langle r\rangle \cap \langle s\rangle$$
$$\langle \neg\ r\rangle = \neg\ \langle r\rangle$$
$$\langle \Pi\ r\rangle = \Pi\ \langle r\rangle$$

The above finiteness theorem is among the most difficult ones to prove formally in the present work. Let us try to understand why.

Brzozowski showed that the number of ~-equivalence classes for a fixed regular expression r is finite by structural induction on r. The inductive steps require proving finiteness by representing equivalence classes of derivatives of the expression in terms of equivalence classes of derivatives of subexpressions. This is technically complicated, especially for concatenation, iteration and projection, since it requires a careful choice of representatives of equivalence classes to reason about them, and Isabelle's automation can not help much with the finiteness arguments.

When carrying out this induction, on a high-level most cases follow Brzozowski's original proof [25]. The only exception is the newly introduced constructor $\Pi\ r$, where we proceed as follows: By induction hypothesis we know that r has a finite set D of distinct derivatives modulo ACI. Some of the formulas in D can have a sum as the topmost constructor. If we repeatedly split such outermost sums in D until none are left, we obtain a finite set X of expressions. Each word derivative $\mathsf{fold}\ \delta\ w\ r$ is ACI equivalent to some $\Pi\ (\bigoplus Y)$ for some $Y \subseteq X$. Since X is finite, its powerset is also finite. Hence, there are only finitely many distinct $\mathsf{fold}\ \delta\ w\ r$ modulo ACI.

The above proof sketch is very informal. The corresponding formal proof is technically more challenging: we need to define precisely in which way $\mathsf{fold}\ \delta\ w\ r$ is ACI equivalent to $\Pi\ (\bigoplus Y)$ for arbitrary words w. Here we employ the ACI normalization function and its equivalent abstract characterization: After the application of $\langle_\rangle$ all sums in the expression are associated to the right and the summands are sorted with respect to $\leq$ and duplicated summands are removed. Again, it is useful to note that $\langle_\rangle$ is idempotent and well-behaved with respect to derivatives.

Finally, we obtain a decision procedure for the equivalence of Π-extended regular expressions by instantiating our locale.

$$
\begin{array}{l}
\texttt{interpretation}\ \mathsf{D}_\Pi : \mathsf{fin_coalg}_D\ \texttt{where} \\
\quad \Sigma = \Sigma_n \\
\quad \mathsf{i}\ r = \langle r \rangle \\
\quad \mathsf{o}\ r = o\ r \\
\quad \mathsf{d}\ a\ r = \langle \delta\ a\ r \rangle \\
\quad \mathsf{L}^\sigma\ r = \mathsf{L}^\tau\ r = \mathsf{L}\ n\ r \\
\quad \mathsf{wf}^\sigma\ r = \mathsf{wf}^\tau\ r = \mathsf{wf}\ n\ r
\end{array}
$$

Note that the free variable n occurring in the interpretation becomes an additional parameter of the procedure $\mathsf{D}_\Pi.\mathsf{eqv}$. Also π and Σ are not yet instantiated.

6.3 Atoms with More Structure

Owens et al. [92] advocate a more compact regular expression structure where the language of an atom denotes a set of one letter words. The gained compactness is beneficial especially for expressions over a large alphabet. In our setting, this would mean using the type $(\alpha\, set)\, RE$ instead of $\alpha\, RE$ (without changing the underlying alphabet type α). We will see later that our alphabet is indeed large—exponential in the number of free variables.

We generalize this idea without committing to a fixed type for the atoms yet. Instead of $\alpha\, RE$, the regular expressions over the alphabet type α on which the algorithm operates will be of type $\beta\, RE$, where the relationship between α and the new atoms β is given by a function $\mathsf{mem}^\mathsf{A} :: \beta \to \alpha \to bool$. The new semantics $\mathsf{L} :: nat \to \beta\, RE \to (\alpha\, list)\, set$ of such regular expressions is defined just as the old L except for the atom case. A similar adjustment is required for the new derivative $\delta :: \alpha \to \beta\, RE \to \beta\, RE$.

$$\mathsf{L}\ n\ (\mathsf{A}\ b) = \{a \mid \mathsf{mem}^\mathsf{A}\ b\ a\} \qquad \delta\ a\ (\mathsf{A}\ b) = \texttt{if}\ \mathsf{mem}^\mathsf{A}\ b\ a\ \texttt{then}\ \varepsilon\ \texttt{else}\ \varnothing$$

Furthermore, the function $\mathsf{wf}^\mathsf{A} :: nat \to \beta \to bool$ is used to detect whether a β-atom is wellformed. The wellformedness check for regular expressions $\mathsf{wf} :: nat \to \beta\, RE \to bool$ will use wf^A in the atom case: $\mathsf{wf}\ n\ b = \mathsf{wf}^\mathsf{A}\ n\ b$. The functions mem^A and wf^A are two further parameters of our procedure. We obtain the original procedure by instantiating β with α and defining $\mathsf{mem}^\mathsf{A}\ (b :: \alpha)\ a \leftrightarrow (a = b)$ and $\mathsf{wf}^\mathsf{A}\ n\ b \leftrightarrow (b \in \Sigma_n)$. For the data structure from Owens *et al.* [92], one would instantiate β with $\alpha\, set$ and define $\mathsf{mem}^\mathsf{A}\ (B :: \alpha\, set)\ a \leftrightarrow (a \in B)$ and $\mathsf{wf}^\mathsf{A}\ n\ B \leftrightarrow (\forall a \in B.\ a \in \Sigma_n)$.

We will benefit from the abstract formulation by instantiating β with a set representation tailored to the particularities of the used regular expressions. In our case, the regular expressions will be translated MSO formulas and a few very particular sets of letters arise from the translation. Therefore, in Section 7 we will define a data type *atom* matching exactly those particularities and instantiate β, mem^A, and wf^A accordingly.

6.4 Alternatives to Brzozowski Derivatives

We have seen earlier the choice of the normalization is crucial for the size of the bisimulation relation. In the previous chapter we have showed that partial derivatives of ordinary regular expressions can be represented by a composition $⟪_⟫ \circ \delta\ a$ where $⟪_⟫$ is a particular normalization function defined using smart constructors and observe that $⟪_⟫$ tends to maintain a better balance between the size of the resulting bisimulation and the ease to compute the normal form than other ad hoc choices. To use partial derivatives here, we extend this particular function $⟪_⟫$ to Π-extended regular expressions as follows. The equations for the smart constructors $\boxplus$, $\boxdot$, $\boxtimes$, $\boxminus$, and $\amalg$ are matched sequentially.

```
primrec ⟪_⟫ :: α RE → α RE where
  ⟪∅⟫ = ∅
  ⟪ε⟫ = ε
  ⟪a⟫ = a
  ⟪r + s⟫ = ⟪r⟫ ⊞ ⟪s⟫
  ⟪r · s⟫ = ⟪r⟫ ⊡ s
  ⟪r*⟫ = r*
  ⟪r ∩ s⟫ = ⟪r⟫ ⊠ ⟪s⟫
  ⟪¬ r⟫ = ⊟ ⟪r⟫
  ⟪Π r⟫ = ∐ ⟪r⟫

primrec ⊡ :: α RE → α RE → α RE where
  ∅ ⊡ r = ∅
  ε ⊡ r = r
  (r + s) ⊡ t = (r ⊡ s) ⊞ (s ⊡ t)
  r ⊡ s = s · t

primrec ⊟ :: α RE → α RE where
  ⊟ (r + s) = (⊟ r) ⊠ (⊟ s)
  ⊟ (r ∩ s) = (⊟ r) ⊞ (⊟ s)
  ⊟ (¬ r) = r
  ⊟ r = ¬ r

primrec ∐ :: α RE → α RE where
  ∐ ∅ = ∅
  ∐ ε = ε
  ∐ (r + s) = (∐ r) ⊞ (∐ s)
  ∐ r = Π r

primrec ⊞ :: α RE → α RE → α RE where
  ∅ ⊞ r = r
  r ⊞ ∅ = r
  (r + s) ⊞ t = r ⊞ (s ⊞ t)
  r ⊞ (s + t) = if r = s then s + t
    else if r ≤ s then r + (s + t)
      else s + (r ⊞ t)
  r ⊞ s = if r = s then r
    else if r ≤ s then r + s
      else s + r

primrec ⊠ :: α RE → α RE → α RE where
  ∅ ⊠ r = ∅
  r ⊠ ∅ = ∅
  (¬ ∅) ⊠ r = r
  r ⊠ (¬ ∅) = r
  (r + s) ⊠ t = (r ⊠ t) ⊞ (s ⊠ t)
  r ⊠ (s + t) = (r ⊠ s) ⊞ (r ⊠ t)
  (r ∩ s) ⊠ t = r ⊠ (s ⊠ t)
  r ⊠ (s ∩ t) = if r = s then s ∩ t
    else if r ≤ s then r ∩ (s ∩ t)
      else s ∩ (r ⊠ t)
  r ⊠ s = if r = s then r
    else if r ≤ s then r ∩ s
      else s ∩ r
```

It is worth noticing that $⟪_⟫$ does not descend recursively into right hand side of concatenation and into iteration. Also, $\boxtimes$ distributes over $\boxplus$, which establishes something like a disjunctive normal form with respect to intersection (conjunction) and union (disjunction). Our motivation for this design goes back to Caron *et al.* [29], who show how to extend partial derivatives to negation and intersection

using sets of sets of regular expressions. The outer level of sets there represents unions, the inner intersections. We conjecture that the usage of our $\langle\!\langle _ \rangle\!\rangle$ as the normalization function produces isomorphic bisimulations to those obtained by working with the extended partial derivatives by Caron et al. [29] directly, but do not attempt to prove it. This conjecture is irrelevant for our purpose, since there is anyway only empirical evidence that partial derivatives perform better than other normalizations for Π-extended regular expression, yet it is an interesting problem to work on in the future. To employ $\langle\!\langle _ \rangle\!\rangle$ in the algorithm, it is sufficient to prove that it preserves wellformedness and languages—an easy exercise in induction.

$$\texttt{theorem}\ \mathsf{wf}\ n\ r \longrightarrow \mathsf{wf}\ n\ \langle\!\langle r \rangle\!\rangle$$
$$\texttt{theorem}\ \mathsf{wf}\ n\ r \longrightarrow \mathsf{L}\ n\ \langle\!\langle r \rangle\!\rangle = \mathsf{L}\ n\ r$$

Using $\langle\!\langle _ \rangle\!\rangle$ we obtain a second interpretation for Π-extended regular expressions.

$$\begin{array}{l}\texttt{interpretation}\ \mathsf{P}_\Pi : \mathsf{coalg}_D\ \texttt{where}\\ \quad \Sigma = \Sigma_n\\ \quad \mathsf{i}\ r = \langle\!\langle r \rangle\!\rangle\\ \quad \mathsf{o}\ r = o\ r\\ \quad \mathsf{d}\ a\ r = \langle\!\langle \delta\ a\ r \rangle\!\rangle\\ \quad \mathsf{L}^\sigma\ r = \mathsf{L}^\tau\ r = \mathsf{L}\ n\ r\\ \quad \mathsf{wf}^\sigma\ r = \mathsf{wf}^\tau\ r = \mathsf{wf}\ n\ r\end{array}$$

Here, we did not succeed in proving finiteness. Indeed, during our attempts in doing so we have discovered a mistake in an informal argument [28] that appeared to prove it. There, a basic claimed de Morgan property of clausal forms—that is sets of sets of expressions for modeling partial derivatives of extended regular expressions [29]—does not hold. Translated to our setting, the source of the problem is that our smart constructor $\boxtimes$ is not idempotent. To see this, assuming $a \preceq b \preceq a \cap b$, we calculate.

$$\begin{aligned}(a + b) \boxtimes (a + b) &= (a \boxtimes (a + b)) \boxplus (b \boxtimes (a + b))\\ &= ((a \boxtimes a) \boxplus (a \boxtimes b)) \boxplus ((b \boxtimes a) \boxplus (b \boxtimes b))\\ &= a \boxplus (a \cap b) \boxplus (a \cap b) \boxplus b\\ &= a + b + (a \cap b)\end{aligned}$$

Consequently, the de Morgan law $\langle\!\langle \neg\ (r + s) \rangle\!\rangle = \langle\!\langle \neg\ r \cap \neg\ s \rangle\!\rangle$ does not hold. One could argue that this is a bad design of the normalization, which is modeled after the operations on sets of sets of expressions given elsewhere [29]. (Those operations suffer from the same limitations.) However, the performance of $\mathsf{P}_\Pi.\mathsf{eqv}$ in practice seems reasonable and changing the normalization function to make $\boxtimes$ idempotent (for example by giving up distributivity of $\cap$ over $+$) resulted in a perceivable decrease in performance.

Nevertheless, an interesting question is whether one can find a fast normalization function that induces the following inductively defined equivalence relation $\approx$. Note that, unlike $\sim$ (ACI), the relation $\approx$ is only an equivalence, not a congruence. Not being able to do so we leave this question as future work.

$$\emptyset + r \approx r \qquad r + \emptyset \approx r \qquad \emptyset \cdot r \approx \emptyset \qquad \varepsilon \cdot r \approx r$$
$$(\neg\, \emptyset) \cap r \approx r \qquad r \cap (\neg\, \emptyset) \approx r \qquad \emptyset \cap r \approx \emptyset \qquad r \cap \emptyset \approx \emptyset$$
$$r + (s + t) \approx (r + s) + t \qquad r + s \approx s + r \qquad r + r \approx r$$
$$r \cap (s \cap t) \approx (r \cap s) \cap t \qquad r \cap s \approx s \cap r \qquad r \cap r \approx r$$
$$r \cap (s + t) \approx (r \cap s) + (r \cap t) \qquad (r + s) \cap t \approx (r \cap t) + (s \cap t)$$
$$(r + s) \cdot t \approx (r \cdot t) + (s \cdot t) \qquad \neg\, (\neg\, r) \approx r$$
$$\neg\, (r + s) \approx (\neg\, r) \cap (\neg\, s) \qquad \neg\, (r \cap s) \approx (\neg\, r) + (\neg\, s)$$
$$\Pi\, (r + s) \approx \Pi\, r + \Pi\, s \qquad \Pi\, \emptyset \approx \emptyset \qquad \Pi\, \varepsilon \approx \varepsilon$$
$$r \approx r \qquad \frac{r \approx s}{s \approx r} \qquad \frac{r \approx s \quad s \approx t}{r \approx t} \qquad \frac{r_1 \approx s_1 \quad r_2 \approx s_2}{r_1 + r_2 \approx s_1 + s_2}$$
$$\frac{r_1 \approx s_1 \quad r_2 \approx s_2}{r_1 \cap r_2 \approx s_1 \cap s_2} \qquad \frac{r_1 \approx s_1}{r_1 \cdot t \approx s_1 \cdot t} \qquad \frac{r \approx s}{\neg\, r \approx \neg\, s} \qquad \frac{r \approx s}{\Pi\, r \approx \Pi\, s}$$

Another promising alternative to Brzozowski derivatives is the data type $\alpha\, RE^{\mathsf{op}}$ of *dual regular expressions* [91]. The data type is obtained by modifying $\alpha\, RE$ as following: drop the negation and intersection constructors and add to every remaining n-ary constructor $\bullet$ a Boolean flag b with the following semantics $\mathsf{L}^{\mathsf{op}} :: nat \to \alpha\, RE^{\mathsf{op}} \to bool$.

$$\mathsf{L}^{\mathsf{op}}_n(\bullet\; b\; r_1 \cdots r_n) = \mathsf{L}_n(\texttt{if}\; b\; \texttt{then}\; \bullet\; r_1 \cdots r_n\; \texttt{else}\; \neg\, (\bullet\; (\neg\, r_1) \cdots (\neg\, r_n)))$$

In the formalization [114] we define wellformedness, derivatives and some ad hoc normalization on $\alpha\, RE^{\mathsf{op}}$ and instantiate our framework to work on those expressions (again without proving finiteness). A later evaluation will show that the decision procedure we obtain by using dual regular expression performs better in certain cases—a phenomenon for which we do not have an explanation yet.

Exponential is the new polynomial.
— Moshe Y. Vardi (2012)

Chapter 7

WMSO Equivalence via Regular Expression Equivalence

Formulas of weak monadic second-order logic (WMSO) constitute a very concise language for expressing regular properties. Conciseness does not come for free: although equivalence of WMSO formulas is decidable [26, 39, 107], the problem's complexity is non-elementary [83]. This fact made Vardi reconsider the meaning of being tractable, as the above quote shows. Nevertheless, the MONA tool [58] teaches us that the fear of the exponential tower is often unnecessary in practice.

Two different semantics for WMSO can be encountered in the literature: WS1S and M2L(Str). Weak monadic second-order logic of one successor (WS1S) is a logic that supports first-order quantification over natural numbers and second-order quantification over finite (therefore "Weak") sets of natural numbers, and beyond this has few additional special predicates, such as $<$ to compare first-order variables. This is the number theoretic approach to WMSO. It is contrasted by the automaton theoretic approach: M2L(Str), the monadic second-order logic on finite strings—we will drop the (Str) from now on. Although the syntax is the same as in WS1S, M2L formulas are considered in the context of a formal word, with variables representing positions in the word. First-order variables denote single positions and second-order variables denote finite sets of positions. The key difference to WS1S is that the fixed length of the formal word bounds the values (positions) that may be assigned to variables. While WS1S seems to be the more natural semantics, M2L has its occasional uses due to a better complexity for certain problems like bounded model construction [7]. Equivalence, however, is also non-elementary for M2L. An in-depth discussion of benefits and drawbacks of the two semantics can be found elsewhere [7, 67].

In this chapter, we introduce WMSO formulas and equip them with the two different semantics (Section 7.1). Then we define translations of formulas into Π-extended regular expressions that preserve the different semantics (Section 7.2) and use those to obtain decision procedures for WMSO based on the ones for Π-

extended regular expressions introduced in the previous chapter (Section 7.3). In a small case study we observe the obtained procedures in action (Section 7.4).

This chapter is based upon the same publications as the previous one [113, 115].

7.1 Syntax and Two Semantics

We briefly introduce WS1S and M2L (in that order), as well as a standard encoding of interpretations as formal words. More thorough introductions are given elsewhere [69, 106]. Formulas are defined inductively as follows.

$$\texttt{datatype}\ \Phi = \mathit{var} < \mathit{var} \mid \mathit{var} \in \mathit{var} \mid \Phi \vee \Phi \mid \neg\, \Phi \mid \exists_1\, \Phi \mid \exists_2\, \Phi$$

There are two kinds of quantifiers: $\exists_1$ quantifies over *first-order* variables, which denote natural numbers; $\exists_2$ quantifies over *second-order* variables, which denote finite sets of natural numbers. We use *de Bruijn indices* for variable bindings—therefore, no names are attached to quantifiers. An occurrence of a bound variable is just an index of type $\mathit{var} = \mathit{nat}$, that refers to its binder by counting the number of quantifiers between the occurrence and the binder. For example, the formula $\exists_1\ \mathsf{FO}\ 0 \vee (\exists_2\ 1 \in 0)$ would be customarily written as $\exists x.\ \mathsf{FO}\ x \vee (\exists X.\ x \in X)$ indicating second-order variables by capital letters. The first 0 and the 1 refer to the outer first-order quantifier, while the second 0 refers to the inner second-order quantifier. *Loose* de Bruijn indices, that are lacking a binder, are considered to be free. For example, $0 < 1$ corresponds to the formula $x < y$ for free x and y.

De Bruijn indices must be manipulated with great care and are hard to read. However, we prefer their usage for three reasons: First, they enable equality of formulas to hold modulo renaming of variables without further ado. Second, for any formula, its free variables are naturally ordered by the de Bruijn index. On several occasions, we will benefit from this order by using a simple list, the n-th element of which is associated to the variable with the index n. A priori this does not seem to be an advantage over using a functional assignment. However one of the occasions will be the underlying alphabet for the formula's language—an alphabet consisting of arbitrary functions is not a good idea for a decision procedure.[1] Third, de Bruijn indices were successfully employed in the formalization. Later we will also sketch how to translate conventional formulas with explicit bindings into our datatype.

The usual abbreviations define conjunction $\varphi \wedge \psi = \neg\, (\neg\, \varphi \vee \neg\, \psi)$, universal quantification $\forall_i\ \varphi = \neg\, \exists_i\ (\neg\, \varphi)$ for $i \in \{1, 2\}$, falsity $\mathsf{F} = \exists_1\ x < x$, and truth $\mathsf{T} = \neg\, \mathsf{F}$.

The truth of a formula depends on an *interpretation* of its free variables. An interpretation assigns each free variable of a formula a value. In our case, an interpretation

[1] And even if we restrict our attention to Boolean functions over a finite domain, we would like to use some more efficient data structure (lists or BDDs), that would impose an order on variables anyway, as the representation.

is represented by a list of finite sets of natural numbers; we abbreviate its type by *interp* = *nat fset list* where α *fset* denotes the *finite* powerset type, just as α *set* is the ordinary powerset type. The n-th entry of an interpretation is the assignment to the free variable of the formula denoted by the loose index n. Therefore, a first-order variable x is at first also assigned a finite set $I[x]$, with the condition that it must be a singleton set (predicate sing). The value assigned by the interpretation to x is then taken to be the sole element of $I[x]$ written, it $(I[x])$.

We can now define the WS1S-semantics of a formula φ with respect to an interpretation I, written $I \models \varphi$. We assume that I has the right length such that all accesses to it are not out of bounds. This assumption will be formalized later in the language notion for WS1S.

$$
\begin{array}{l}
\texttt{primrec} \models\ \mathit{interp} \to \Phi \to \mathit{bool}\ \texttt{where} \\
\quad I \models (x < y) = \mathsf{sing}\ (I[x]) \wedge \mathsf{sing}\ (I[y]) \wedge \mathsf{it}\ (I[x]) < \mathsf{it}\ (I[y]) \\
\quad I \models (x \in X) = \mathsf{sing}\ (I[x]) \wedge \mathsf{it}\ (I[x]) \in I[X] \\
\quad I \models (\varphi \vee \psi) = I \models \varphi \vee I \models \psi \\
\quad I \models (\neg\, \varphi) = \neg(I \models \varphi) \\
\quad I \models (\exists_1\, \varphi) = \exists p.\ (\{p\}\ \#\ I) \models \varphi \\
\quad I \models (\exists_2\, \varphi) = \exists P.\ |P| < \infty \wedge (P\ \#\ I) \models \varphi
\end{array}
$$

Intuitively, $x < y$ compares assignments of first-order variables, $x \in X$ checks that the assignment to x is contained in the assignment to X. Boolean connectives and quantifiers behave as expected. De Bruijn indices allow us to conveniently extend the interpretation by simply prepending the value for the most recently quantified variable when recursively entering the scope of a quantifier. If $I \models \varphi$ holds, we say *I satisfies* φ or I is a *model* of φ.

Formulas may have free variables. The functions $\mathcal{V}_1$ and $\mathcal{V}_2$ collect the free first-order and second-order variables.

$$
\begin{array}{l}
\texttt{primrec}\ \mathcal{V}_1 :: \Phi \to \mathit{nat\ set}\ \texttt{where} \\
\quad \mathcal{V}_1\ (m_1 < m_2) = \{m_1, m_2\} \\
\quad \mathcal{V}_1\ (m \in M) = \{m\} \\
\quad \mathcal{V}_1\ (\varphi \vee \psi) = \mathcal{V}_1\ \varphi \cup \mathcal{V}_1\ \psi \\
\quad \mathcal{V}_1\ (\neg\, \varphi) = \mathcal{V}_1\ \varphi \\
\quad \mathcal{V}_1\ (\exists_1\, \varphi) = \lfloor \mathcal{V}_1\ \varphi \setminus \{0\} \rfloor \\
\quad \mathcal{V}_1\ (\exists_2\, \varphi) = \lfloor \mathcal{V}_1\ \varphi \rfloor
\end{array}
$$

$$
\begin{array}{l}
\texttt{primrec}\ \mathcal{V}_2 :: \Phi \to \mathit{nat\ set}\ \texttt{where} \\
\quad \mathcal{V}_2\ (m_1 < m_2) = \{\} \\
\quad \mathcal{V}_2\ (m \in M) = \{M\} \\
\quad \mathcal{V}_2\ (\varphi \vee \psi) = \mathcal{V}_2\ \varphi \cup \mathcal{V}_2\ \psi \\
\quad \mathcal{V}_2\ (\neg\, \varphi) = \mathcal{V}_2\ \varphi \\
\quad \mathcal{V}_2\ (\exists_1\, \varphi) = \lfloor \mathcal{V}_2\ \varphi \rfloor \\
\quad \mathcal{V}_2\ (\exists_2\, \varphi) = \lfloor \mathcal{V}_2\ \varphi \setminus \{0\} \rfloor
\end{array}
$$

The notation $\lfloor X \rfloor$ is shorthand for $(\lambda x.\ x - 1) \bullet X$, which reverts the increasing effect of an existential quantifier on previously bound or free variables. To obtain only free variables, bound variables are removed when their quantifier is processed, at which point the bound variable has index 0.

Just as for Π-extended regular expressions, not all formulas in Φ are meaningful. Consider $0 \in 0$, where 0 is both a first-order and a second-order variable. To exclude such formulas, we define the predicate $\mathsf{wf}^\Phi :: \mathit{nat} \to \Phi \to \mathit{bool}$ as $\mathsf{wf}^\Phi\ n\ \varphi = (\mathcal{V}_1\ \varphi \cap \mathcal{V}_2\ \varphi = \{\}) \wedge \mathsf{wf}_0^\Phi\ n\ \varphi$ and call a formula φ *n-wellformed* if $\mathsf{wf}^\Phi\ n\ \varphi$ holds. The recursively defined predicate wf_0^Φ is used to impose further assumptions on the structure of n-wellformed formulas, which will simplify our proofs.

$$
\begin{array}{l}
\texttt{primrec}\ \mathsf{wf}_0^\Phi :: \mathit{nat} \to \Phi \to \mathit{bool}\ \texttt{where} \\
\quad \mathsf{wf}_0^\Phi\ n\ (m_1 < m_2) = m_1 < n \wedge m_2 < n \\
\quad \mathsf{wf}_0^\Phi\ n\ (m \in M) = m < n \wedge M < n \\
\quad \mathsf{wf}_0^\Phi\ n\ (\varphi \vee \psi) = \mathsf{wf}_0^\Phi\ n\ \varphi \wedge \mathsf{wf}_0^\Phi\ n\ \psi \\
\quad \mathsf{wf}_0^\Phi\ n\ (\neg\ \varphi) = \mathsf{wf}_0^\Phi\ n\ \varphi \\
\quad \mathsf{wf}_0^\Phi\ n\ (\exists_1\ \varphi) = \mathsf{wf}_0^\Phi\ (n+1)\ \varphi \wedge 0 \in \mathcal{V}_1\ \varphi \wedge 0 \notin \mathcal{V}_2\ \varphi \\
\quad \mathsf{wf}_0^\Phi\ n\ (\exists_2\ \varphi) = \mathsf{wf}_0^\Phi\ (n+1)\ \varphi \wedge 0 \notin \mathcal{V}_1\ \varphi \wedge 0 \in \mathcal{V}_2\ \varphi
\end{array}
$$

The predicate $\mathsf{wf}_0^\Phi\ n\ \varphi$ ensures that the index of every free variable in φ is below n, thereby guaranteeing that no array-out-of-bound accesses will happen in $\models$ given an n-wellformed formula and an interpretation of length n. Moreover, wf_0^Φ checks that bound variables are correctly used as first-order or second-order with respect to their binders and excludes formulas with unused binders; unused binders are obviously superfluous.

Next, we turn our attention to the alternative existing semantics for WMSO formulas: M2L [7, 68]. Early versions of MONA implemented the M2L semantics. M2L semantics fits more naturally into the realm of automata. The difference between the two semantics lies only in the treatment of quantifiers. In WS1S, a quantified variable may be assigned arbitrarily large sets. In M2L, allowed assignments are bounded by the some number $\#I$ greater than all numbers occurring in the interpretation I, formally $\forall x \in \bigcup_{i=0}^{|I|} I[i].\ x < \#I$. The number $\#I$ is intrinsic to an interpretation—that is the type *interp* is not just *nat fset list* but a subtype of *nat* × *nat fset list* additionally constrained by the above boundedness assumption. Here, for simplicity (mainly in order to be able to use the standard list notation on *interp*) we pretend that this is not the case and $\#I$ is computable from I. For WS1S the value of $\#I$ does not matter for satisfiability. Also note that $\#I \neq |I|$ in general.

The decision procedure for M2L will arise as a natural by-product on the way to a decision procedure for WS1S . We introduce its semantics formally, written $I \models_< \varphi$. Only the quantifier cases differ from the definition of $\models$.

$$
\begin{array}{l}
I \models_< (\exists_1\ \varphi) = \exists p.\ (p < \#I) \wedge (\{p\}\ \#\ I) \models_< \varphi \\
I \models_< (\exists_2\ \varphi) = \exists P.\ (\forall p \in P.\ p < \#I) \wedge (P\ \#\ I) \models_< \varphi
\end{array}
$$

The subtle difference between M2L and WS1S is best illustrated by the formula $\exists_2\ (\forall_1\ 0 \in 1)$ (with names: $\exists X.\ \forall x.\ x \in X$). In the M2L semantics $\exists_2\ (\forall_1\ 0 \in 1)$ is satisfied by all wellformed interpretations—the witness set for the outer existential

quantifier is for a wellformed interpretation I just the set $\{0, \ldots, \#I - 1\}$. In contrast, in WS1S, there is no finite set which contains all arbitrarily large positions, thus $\exists_2\ (\forall_1\ 0 \in 1)$ is unsatisfiable.

Although equally expressive as WS1S and somehow unusual in its treatment of quantifiers, the logic M2L has certain advantages, for example it yields a better complexity for the bounded model construction problem [7]. Therefore, M2L is not just an ad hoc intermediate construct, and obtaining a decision procedure for it as well (basically for free) is of some value.

Two formulas are equivalent if they have the same models. Thus, we consider the language of a formula to be the set of its models encoded as words. The encoding of models (or interpretations in general) is standard: a finite set of natural numbers X is transformed into a list of Booleans, where $\top$ in the n-th positions means that $n \in X$. For interpretations the encoding function enc applies this transformation pointwise and pads the lists with $\bot$s at the end to have length $\#I$; we omit its obvious definition. For example for $I = [\{1, 2, 3\}, \{0, 2\}]$ with $\#I = 4$, we obtain enc $I = [[\bot, \top, \top, \top], [\top, \bot, \top, \bot]]$, which we transpose to obtain the formal word w_I over the alphabet $bool^{|I|}$ (each letter is a vector, that is a list of fixed length).

$$\mathsf{w}_I = \left[\begin{pmatrix}\bot\\\top\end{pmatrix}, \begin{pmatrix}\top\\\bot\end{pmatrix}, \begin{pmatrix}\top\\\top\end{pmatrix}, \begin{pmatrix}\top\\\bot\end{pmatrix}\right]$$

In w_I a row corresponds to an assignment to a variable. For first-order variables a row must contain exactly one $\top$. Finally, we can defined the *WS1S-language* of an n-wellformed formula as $\mathsf{L}^\Phi\ n\ \varphi = \{\mathsf{w}_I \mid I \vDash \varphi \wedge |I| = n \wedge (\forall m \in \mathcal{V}_1\ \varphi.\ \mathsf{sing}\ (I[m]))\}$. Similarly, the *M2L-language* of an n-wellformed formula uses the bounded semantics $\vDash_<$ as $\mathsf{L}^\Phi_<\ n\ \varphi = \{\mathsf{w}_I \mid I \vDash_< \varphi \wedge |I| = n \wedge (\forall m \in \mathcal{V}_1\ \varphi.\ \mathsf{sing}\ (I[m])) \wedge \#I \neq 0\}$, where for technical reasons we exclude interpretations with $\#I = 0$, thereby ensuring that $[]$ is not contained in the M2L-language of a formula.

Note that we consider every column of w_I as a letter of a new alphabet, which we instantiate for the underlying alphabet $\Sigma_n = bool^n$ of Π-extended regular expressions of Chapter 6. Furthermore, the second parameter $\pi :: \Sigma_{n+1} \to \Sigma_n$ of our decision procedure for those regular expressions can now also be instantiated with tl where tl $(x \# xs) = xs$. Thus, the projection Π operates on words by removing the first row from words in the language of the body expression, reflecting the semantics of an existential quantifier.

7.2 From WMSO Formulas to Regular Expressions

We have fixed the underlying alphabet type *bool list* of the language of a formula. In principle, we could start translating formulas of type Φ into regular expressions of type *bool list RE*. However, the abstraction for atoms introduced in Section 6.3 caters for a more efficient encoding of formulas. We define the datatype *atom* as

$\texttt{datatype}\ \mathit{atom} = \mathsf{ASing}\ \mathit{bool\ list} \mid \mathsf{ANth}\ \mathit{var\ bool} \mid \mathsf{ANth_2}\ \mathit{var\ var}$

Each constructor of *atom* represents a set of elements of type *bool list*. The constructor ASing represents the singleton set containing the constructor's argument. The atom ANth $m\ b$ represents the set of boolean vectors which have the value b in their m-th entry. The remaining constructor $\mathsf{ANth_2}$ has a similar purpose. Let us make this precise by instantiating the two parameters $\mathsf{mem^A}$ and $\mathsf{wf^A}$ from Section 6.3.

$$\begin{array}{l}
\texttt{primrec}\ \mathsf{mem^A} :: \mathit{atom} \to \mathit{bool\ list} \to \mathit{bool}\ \texttt{where} \\
\quad \mathsf{mem^A}\ (\mathsf{ASing}\ bs)\ bs' = (bs = bs') \\
\quad \mathsf{mem^A}\ (\mathsf{ANth}\ m\ b)\ bs = (bs[m] \leftrightarrow b) \\
\quad \mathsf{mem^A}\ (\mathsf{ANth_2}\ m\ M)\ bs = bs[m] \wedge bs[M]
\end{array}$$

$$\begin{array}{l}
\texttt{primrec}\ \mathsf{wf^A} :: \mathit{nat} \to \mathit{atom} \to \mathit{bool}\ \texttt{where} \\
\quad \mathsf{wf^A}\ n\ (\mathsf{ASing}\ bs) = |bs| = n \\
\quad \mathsf{wf^A}\ n\ (\mathsf{ANth}\ m\ _) = m < n \\
\quad \mathsf{wf^A}\ n\ (\mathsf{ANth_2}\ m\ M) = m < n \wedge M < n
\end{array}$$

Now, we are set to tackle the translations of formulas into regular expressions. WMSO formulas interpreted in M2L are translated by means of the following function.

$$\begin{array}{l}
\texttt{primrec}\ \mathsf{mkRE_<} :: \Phi \to \mathit{atom\ RE}\ \texttt{where} \\
\quad \mathsf{mkRE_<}\ (m_1 < m_2) = \neg\, \varnothing \cdot \mathsf{ANth}\ m_1\ \top \cdot \neg\, \varnothing \cdot \mathsf{ANth}\ m_2\ \top \cdot \neg\, \varnothing \\
\quad \mathsf{mkRE_<}\ (m \in M) = \neg\, \varnothing \cdot \mathsf{ANth_2}\ m\ M \cdot \neg\, \varnothing \\
\quad \mathsf{mkRE_<}\ (\varphi \vee \psi) = \mathsf{mkRE_<}\ \varphi + \mathsf{mkRE_<}\ \psi \\
\quad \mathsf{mkRE_<}\ (\neg\, \varphi) = \neg\, \mathsf{mkRE_<}\ \varphi \\
\quad \mathsf{mkRE_<}\ (\exists_1\ \varphi) = \Pi\ (\mathsf{mkRE_<}\ \varphi \cap \mathsf{WF}\ \{0\}) \\
\quad \mathsf{mkRE_<}\ (\exists_2\ \varphi) = \Pi\ (\mathsf{mkRE_<}\ \varphi)
\end{array}$$

At first, we ignore the function WF that is used in the case of the first-order quantifier. The natural number parameter of $\mathsf{mkRE_<}$ indicates the number for free variables for the processed formula. The parameter is increased when entering recursively the scope of an existential quantifier.

Let us explain the intuition behind the translation exemplified by the $m_1 < m_2$ case. We fix a wellformed model I of $m \in M$. This model must satisfy $\mathsf{sing}\ (I[m])$ and $\mathsf{it}\ (I[m]) \in I[M]$, or equivalently there must exist a Boolean vector bs of length n and a position p such that $I[m] = \{p\}$, $\mathsf{w}_I[p] = bs$, and $bs[m] = bs[M] = \top$. Therefore, the letter at position p of w_I is matched by the "middle" part $\mathsf{ANth_2}\ m\ M$ of $\mathsf{mkRE_<}\ (m \in M)$, while the subexpressions $\neg\, \varnothing$ (whose language is Σ_n^*) match the first p and the last $\#I - p - 1$ letters of w_I.

Conversely, if we fix a word from $\mathsf{mkRE}_<\ (m \in M)$, it will be equal to an encoding of an interpretation that satisfies $m \in M$ by a similar argument. However, the interpretation might be not wellformed for $m \in M$. This happens because the regular expression $\mathsf{mkRE}_<\ (m \in M)$ does not capture the distinction between first-order and second-order variables: it accepts encodings of interpretations that have the value $\top$ more than once at different positions representing the same first-order variable. This indicates that the subexpressions $\neg\, \varnothing$ in the base cases are not precise enough, but also in the case of Boolean operators similar issues arise. So instead of tinkering with the base cases, it is better to separate the generation a regular expression that encodes models from the one that encodes wellformed interpretations. To rule out not wellformed interpretations is exactly the purpose of the WF function.

$$\texttt{definition}\ \mathsf{WF} :: \mathit{var\,set} \to \mathit{atom\,RE}\ \texttt{where}$$
$$\mathsf{WF}\ X = \bigcap_{m \in X} \left((\mathsf{ANth}\ m\ \bot)^* \cdot \mathsf{ANth}\ m\ \top \cdot (\mathsf{ANth}\ m\ \bot)^*\right)$$

The regular expression $\mathsf{WF}\ (\mathcal{V}_1\ \varphi)$ accepts exactly the encodings of wellformed interpretations (both models and non-models, therefore semantics-independent) for φ by ensuring that first-order variables are encoded correctly (by forcing the encoding of an interpretation to contain exactly one $\top$ in rows belonging to a first-order variable).

$$\texttt{theorem}\ \forall m \in X.\ m < n \longrightarrow$$
$$\mathsf{L}\ n\ (\mathsf{WF}\ X) = \{\mathsf{w}_I \mid |I| = n \wedge (\forall m \in X.\ \mathsf{sing}\ (I[m]))\}$$

Using WF in every case of the recursive definition of $\mathsf{mkRE}_<$ is sound but very redundant—instead it is enough to perform the intersection once globally for the entire formula and additionally for every variable introduced by the first-order existential quantifier.

WMSO formulas interpreted in WS1S are translated into regular expressions by means of the function mkRE. The definition of mkRE is basically the same as the one of $\mathsf{mkRE}_<$ except for the existential quantifier cases and the natural number parameter n that indicates the number of free variables and needs to be threaded through the recursion here (because of the change in the quantifier cases).

$$\texttt{primrec}\ \mathsf{mkRE} :: \mathit{nat} \to \Phi \to \mathit{atom\,RE}\ \texttt{where}$$
$$\mathsf{mkRE}\ n\ (m_1 < m_2) = \neg\, \varnothing \cdot \mathsf{ANth}\ m_1\ \top \cdot \neg\, \varnothing \cdot \mathsf{ANth}\ m_2\ \top \cdot \neg\, \varnothing$$
$$\mathsf{mkRE}\ n\ (m \in M) = \neg\, \varnothing \cdot \mathsf{ANth}_2\ m\ M \cdot \neg\, \varnothing$$
$$\mathsf{mkRE}\ n\ (\varphi \vee \psi) = \mathsf{mkRE}\ n\ \varphi + \mathsf{mkRE}\ n\ \psi$$
$$\mathsf{mkRE}\ n\ (\neg\, \varphi) = \neg\, \mathsf{mkRE}\ n\ \varphi$$
$$\mathsf{mkRE}\ n\ (\exists_1\, \varphi) = \mathcal{Q}\ (\bot^n)\ (\Pi\ (\mathsf{mkRE}\ (n+1)\ \varphi \cap \mathsf{WF}\ \{0\}))$$
$$\mathsf{mkRE}\ n\ (\exists_2\, \varphi) = \mathcal{Q}\ (\bot^n)\ (\Pi\ (\mathsf{mkRE}\ (n+1)\ \varphi))$$

A first step to understand the required change is to note that interpretations can be arbitrarily padded or shortened with the letter $\bot^n$ from the right without affecting their status of being models or non-models of a given formula. Thus, a language of a formula must be closed under all such paddings/shortening. Now, after projection, the language of a regular expression might not fulfill this invariant. For example the language of the 1-wellformed expression $r = \Pi\left(\left(\binom{\bot}{\top}\cdot\binom{\top}{\bot}\cdot\binom{\bot}{\bot}\right)^*\right)$ does contain the word $[\top^1, \bot^1]$ but not the word $[\top^1]$ and is therefore not closed under all shortenings by $\bot^1$. In contrast, the 2-wellformed argument expression to Π in r does fulfill the invariant.

The regular operation $\mathcal{Q} :: \mathit{bool\ list} \to \mathit{atom\ RE} \to \mathit{atom\ RE}$ implements this padding/shortening on the regular expression level and reestablishes thereby the required invariant. (In the literature, this process of reestablishing the invariant after a projection is called *futurization* [68].) In more detail, the following language identity holds for an n-wellformed Π-extended regular expression r.

$$\texttt{theorem}\ \mathsf{wf}\ n\ r \longrightarrow \mathsf{L}\ n\ (\mathcal{Q}\ a\ r) = \left\{x\ @\ a^m \mid \exists l.\ x\ @\ a^l \in \mathsf{L}\ n\ r\right\}$$

The concrete executable definition of $\mathcal{Q}$ is more involved. On a high-level, $\mathcal{Q}$ is computed by repeatedly deriving from the right by a via the function $\eth\ a$ (followed by ACI-normalization) until a repetition is encountered. The definition of $\eth$ is identical to the familiar Brzozowski derivative δ that derives from the left except for the concatenation and iteration cases (in which $\eth$ is dual).

```
primrec ծ :: α → α RE → α RE where
  ծ a ∅ = ∅
  ծ a ε = ∅
  ծ a (A b) = if a = b then ε else ∅
  ծ a (r + s) = ծ a r + ծ a s
  ծ a (r · s) = r · ծ a s + if o s then ծ a r else ∅
  ծ a (r*) = r* · ծ a r
  ծ a (r ∩ s) = ծ a r ∩ ծ a s
  ծ a (¬ r) = ¬ ծ a r
  ծ a (Π r) = Π (⊕_{b∈π⁻a} ծ r b)
```

Unsurprisingly, due to their similarity to ordinary Brzozowski derivatives, right derivatives preserve wellformedness, compute right quotients, and have finitely many equivalence classes modulo ACI.

```
theorem wf n r ⟶ wf n (ծ a r)
theorem wf n r ⟶ L n (ծ a r) = {w | w @ [a] ∈ L n r}
theorem finite {⟨fold ծ w r⟩ | w :: α list}
```

Repeatedly deriving from the right is implemented using the while combinator. The state over which the combinator iterates has type *bool* × *α RE list*. The Boolean component indicates whether the loop should be executed once more, while the list contains all the derivatives computed so far in reversed order (the last element of the list is the initial regular expression). The loop is exited on the first déjà vu. The termination of this procedure is established by the finiteness theorem of ACI-equivalent right derivatives. After exiting the while loop, the operation $\mathcal{Q}$ unions all the expressions computed so far, yielding an operation whose language is $\{x \mid \exists l.\ x\ @\ a^l \in \mathsf{L}\ n\ r\}$. To obtain the desired semantics the iteration of a (lifted to the *atom* type by ASing) is concatenated to the union.

```
definition b :: bool × α RE list → bool where
  b (b, _) = b

definition c :: α → bool × α RE list → bool × α RE list where
  c a (_, rs) =
    let s = ⟨ჵ a (hd rs)⟩
    in if s ∈ set rs then (⊥, rs) else (⊤, s # rs)

definition Q :: α → α RE → α RE where
  Q a r =
    let R = case while b (c a) (⊤, ⟨r⟩) of Some (_, rs) ⇒ set rs
    in (⊕_{r∈R} r) · (ASing a)*
```

Finally, we can establish the language correspondence between formulas and generated regular expressions.

$$\text{theorem } \mathsf{wf}^{\Phi}\ n\ \varphi \longrightarrow \mathsf{L}^{\Phi}\ n\ \varphi = \mathsf{L}\ n\ (\mathsf{mkRE}\ n\ \varphi \cap \mathsf{WF}\ (\mathcal{V}_1\ \varphi))$$
$$\text{theorem } \mathsf{wf}^{\Phi}\ n\ \varphi \longrightarrow \mathsf{L}^{\Phi}_{<}\ n\ \varphi = \mathsf{L}\ n\ (\mathsf{mkRE}_{<}\ \varphi \cap \mathsf{WF}\ (\mathcal{V}_1\ \varphi)) \setminus \{[]\}$$

The proof is by structural induction on φ. Above we have seen the argument for the base case $m \in M$, the other base case follows similarly. The cases $\exists_1\ \varphi$ and $\exists_2\ \varphi$ follow easily from the semantics of Π given by our concrete instantiation for π and Σ_n and the induction hypothesis. The most interesting cases are, somehow unexpectedly, those for Boolean operators. Although the definitions are purely structural, sets of encodings of models must be composed or, even worse, complemented in the inductive steps. The key property required here is that the encoding of interpretations as formal words does not identify models and non-models: two different wellformed interpretations for a formula—one being a model, the other being a non-model—are encoded into different words. This is again established by structural induction on formulas for both semantics.

$$\text{theorem } \mathsf{w}_I = \mathsf{w}_J \wedge \mathsf{wf}\ n\ \varphi \wedge |I| = |J| = n \wedge$$
$$(\forall x \in \mathcal{V}_1\ \varphi.\ \mathsf{sing}\ (I[x]) \wedge \mathsf{sing}\ (J[x])) \longrightarrow I \models \varphi \leftrightarrow J \models \varphi$$

$\texttt{theorem}\ \mathsf{w}_I = \mathsf{w}_J \wedge \mathsf{wf}\ n\ \varphi \wedge |I| = |J| = n \wedge$
$(\forall x \in \mathcal{V}_1\ \varphi.\ \mathsf{sing}\ (I[x]) \wedge \mathsf{sing}\ (J[x])) \longrightarrow I \models_< \varphi \leftrightarrow J \models_< \varphi$

7.3 Decision Procedure

To decide language equivalence of WMSO formulas we now can simply translate them to Π-extended regular expressions and use then the decision procedures on those expressions developed in Chapter 6. We obtain the following four interpretations of our locales: for each of the two semantics a totally correct one using only ACI normalization and a partially correct one using the stronger normalization function. Note that we add the empty word into both regular expression when working with the M2L semantics. This is allowed, since [] is not a valid encoding of an interpretation, and necessary because the characteristic theorem of $\mathsf{mkRE}_<$ does not give us any information whether the empty word is contained in the output of $\mathsf{mkRE}_<$ or not.

$\texttt{interpretation}\ \mathsf{M2L}^{\mathsf{D}}_{\Pi} : \mathsf{fin_coalg}_D\ \texttt{where}$
$\Sigma = \Sigma_n$
$\mathsf{i}\ \varphi = \langle \mathsf{mkRE}_<\ \varphi \rangle$
$\mathsf{o}\ r = o\ r$
$\mathsf{d}\ a\ r = \langle \delta\ a\ r \rangle$
$\mathsf{L}^{\sigma}\ r = \mathsf{L}\ n\ r$
$\mathsf{L}^{\tau}\ \varphi = \mathsf{L}^{\Phi}_<\ n\ \varphi$
$\mathsf{wf}^{\sigma}\ r = \mathsf{wf}\ n\ r$
$\mathsf{wf}^{\tau}\ \varphi = \mathsf{wf}^{\Phi}\ n\ \varphi$

$\texttt{interpretation}\ \mathsf{M2L}^{\mathsf{P}}_{\Pi} : \mathsf{coalg}_D\ \texttt{where}$
$\Sigma = \Sigma_n$
$\mathsf{i}\ \varphi = \langle\!\langle \mathsf{mkRE}_<\ \varphi \rangle\!\rangle$
$\mathsf{o}\ r = o\ r$
$\mathsf{d}\ a\ r = \langle\!\langle \delta\ a\ r \rangle\!\rangle$
$\mathsf{L}^{\sigma}\ r = \mathsf{L}\ n\ r$
$\mathsf{L}^{\tau}\ \varphi = \mathsf{L}^{\Phi}_<\ n\ \varphi$
$\mathsf{wf}^{\sigma}\ r = \mathsf{wf}\ n\ r$
$\mathsf{wf}^{\tau}\ \varphi = \mathsf{wf}^{\Phi}\ n\ \varphi$

$\texttt{interpretation}\ \mathsf{WS1S}^{\mathsf{D}}_{\Pi} : \mathsf{fin_coalg}_D\ \texttt{where}$
$\Sigma = \Sigma_n$
$\mathsf{i}\ \varphi = \langle \mathsf{mkRE}\ n\ \varphi \rangle$
$\mathsf{o}\ r = o\ r$
$\mathsf{d}\ a\ r = \langle \delta\ a\ r \rangle$
$\mathsf{L}^{\sigma}\ r = \mathsf{L}\ n\ r$
$\mathsf{L}^{\tau}\ \varphi = \mathsf{L}^{\Phi}\ n\ \varphi$
$\mathsf{wf}^{\sigma}\ r = \mathsf{wf}\ n\ r$
$\mathsf{wf}^{\tau}\ \varphi = \mathsf{wf}^{\Phi}\ n\ \varphi$

$\texttt{interpretation}\ \mathsf{WS1S}^{\mathsf{P}}_{\Pi} : \mathsf{coalg}_D\ \texttt{where}$
$\Sigma = \Sigma_n$
$\mathsf{i}\ \varphi = \langle\!\langle \mathsf{mkRE}\ n\ \varphi \rangle\!\rangle$
$\mathsf{o}\ r = o\ r$
$\mathsf{d}\ a\ r = \langle\!\langle \delta\ a\ r \rangle\!\rangle$
$\mathsf{L}^{\sigma}\ r = \mathsf{L}\ n\ r$
$\mathsf{L}^{\tau}\ \varphi = \mathsf{L}^{\Phi}\ n\ \varphi$
$\mathsf{wf}^{\sigma}\ r = \mathsf{wf}\ n\ r$
$\mathsf{wf}^{\tau}\ \varphi = \mathsf{wf}^{\Phi}\ n\ \varphi$

As a first sanity check, we apply our translation for M2L to the simple formula $\varphi = \exists_2\ (\forall_1\ 0 \in 1)$ (with names: $\exists X.\ \forall x.\ x \in X$), that is valid under the M2L semantics (but unsatisfiable under the WS1S semantics as discussed earlier). Since φ is closed, it is 0-wellformed and our underlying alphabet is $\Sigma_0 = \mathit{bool}^0 = \{()\}$ for some base alphabet Σ. The function $\langle\!\langle \mathsf{mkRE}_<\ \varphi \rangle\!\rangle$ translates φ to the accepting Π-extended regular expression $\Pi\ \mathsf{r}$ over Σ_0 where r is an abbreviation:

$$\mathsf{r} = \neg\ \Pi\ (((\mathsf{ANth}\ 0\ \bot)^* \cdot \mathsf{ANth}\ 0\ \top \cdot (\mathsf{ANth}\ 0\ \bot)^*) \cap \neg\ (\neg\ \varnothing \cdot \mathsf{ANth}_2\ 0\ 1 \cdot \neg\ \varnothing))$$

Derivatives of Π r by words of the form $()^n$ for $n > 0$ are all equal modulo $⟪_⟫$ to a single (also accepting) expression. More precisely, for all $w \in \Sigma_0^* \setminus \{[]\}$ we have:

$$\mathsf{fold}\ \delta\ w\ (\Pi\ \mathsf{r}) = \Pi\ \mathsf{r} + \Pi\ (\mathsf{r} \cap \neg\ \Pi\ ((\mathsf{ANth}\ 0\ \bot^*) \cap \neg\ (\neg\ \varnothing \cdot \mathsf{ANth}_2\ 0\ 1 \cdot \neg\ \varnothing)))$$

Because all derivatives of its translation are accepting, the formula φ must be valid. We would have loved to include the same example using the WS1S semantics as well, but unfortunately the output of the translation (and normalization) is a regular expression with more than 2000 constructors (which the decision procedure still can handle).

Finally, we want to remark that the usage of de Bruijn indices, which is often perceived as an intrusion of syntax into the semantics, could be in principle easily hidden from the user of our procedure by means of a simple transformation. Standard nameful WMSO formulas are given by the following datatype.

$$\texttt{datatype}\ \Psi = \mathit{var} < \mathit{var}\ \mid\ \mathit{var} \in \mathit{var}\ \mid\ \Psi \vee \Psi\ \mid\ \neg\ \Psi\ \mid\ \exists_1 \mathit{var}.\ \Psi\ \mid\ \exists_2 \mathit{var}.\ \Psi$$

In Ψ the type *var* denotes some arbitrary type of variable names and not just natural numbers as the de Bruijn indices in Φ. Formulas of type Ψ can be equipped with the standard M2L or WS1S semantics (which we omit here). Then, the recursive translation ι gets rid of the names in a semantics preserving fashion.

$$\begin{aligned}
&\texttt{primrec}\ \iota :: \mathit{var\,list} \to \Psi \to \Phi\ \texttt{where}\\
&\quad \iota\ xs\ (m_1 < m_2) = \mathsf{idx}\ xs\ m_1 < \mathsf{idx}\ xs\ m_2\\
&\quad \iota\ xs\ (m \in M) = \mathsf{idx}\ xs\ m \in \mathsf{idx}\ xs\ M\\
&\quad \iota\ xs\ (\varphi \vee \psi) = \iota\ xs\ \varphi \vee \iota\ xs\ \psi\\
&\quad \iota\ xs\ (\neg\ \varphi) = \neg\ \iota\ xs\ \varphi\\
&\quad \iota\ xs\ (\exists_1 x.\ \varphi) = \exists_1\ (\iota\ (x\ \#\ xs)\ \varphi)\\
&\quad \iota\ xs\ (\exists_2 x.\ \varphi) = \exists_2\ (\iota\ (x\ \#\ xs)\ \varphi)
\end{aligned}$$

The function $\mathsf{idx} :: \alpha\,\mathit{list} \to \alpha \to \mathit{nat}$ computes the index of the first occurrence of a given element in a given list and some arbitrary value, if there is no occurrence. Given a nameful formula $\psi :: \Psi$, the function ι must then be called with an arbitrarily ordered list xs of free variables of ψ, to obtain the corresponding nameless formula $\iota\ xs\ \psi$ that denotes the same regular language as ψ.

7.4 Case Study: Finite-Word LTL

We want to execute the code generated by Isabelle/HOL for our decision procedures on some larger examples. For simplicity, we first focus on M2L.

To create larger formulas, it is helpful to introduce some syntactic abbreviations. Checking that a formula is valid amounts to checking its equivalence to T. We call

the function that performs this check Thm. Implication $\varphi \to \psi$ is defined as $(\neg\,\varphi) \vee \psi$ and universal quantification $\forall_i\ \varphi$ as before as $\neg\ \exists_i\ \neg\ \varphi$. Next, we introduce temporal logical operators *always* $\Box P :: nat \to \Phi$ and *eventually* $\Diamond P :: nat \to \Phi$ depending on $P :: nat \to \Phi$—a formula parameterized by a single variable t indicating the time. The operators have their usual meaning except that with the given M2L semantics the time variable ranges over a fixed set determined by the interpretation. Additionally, we lift the disjunction and implication to time-parameterized formulas.

$$\begin{array}{lcl} \Box P\ t & = & \forall_1\ (\neg\ t+1 < 0 \to P\ 0) \\ \Diamond P\ t & = & \exists_1\ (\neg\ t+1 < 0 \wedge P\ 0) \\ (P \boxed{\to} Q)\ t & = & P\ t \to Q\ t \\ (P \boxed{\vee} Q)\ t & = & P\ t \vee Q\ t \end{array}$$

Note that $t+1$ has nothing to do with the next time step. It is just the lifting of the de Bruijn index under a single quantifier. The same applies to the de Bruijn index 0. Above, 0 represents all/some time steps *after* the time step represented by t.

Formulas of linear temporal logic usually reason about atomic predicates for which the interpretation must specify at which points in time they are true. This information can be encoded by second-order variables.

A free second-order variable can be namely used to encode an atomic predicate. The variable denotes the set of points in time for which the atomic predicate holds. Thus, the LTL formulas $\Box a \to \Diamond a$ and $\Box a \to \Box \Diamond a$ (with a single atomic predicate a) can be modeled by the following two formulas.

$$\begin{array}{l} \forall_1\ \big(\Box\,(\lambda t.\, t \in 2) \boxed{\to} \Diamond\,(\lambda t.\, t \in 2)\big)\ 0 \\ \forall_1\ \big(\Box\,(\lambda t.\, t \in 2) \boxed{\to} \Box\,\Diamond\,(\lambda t.\, t \in 3)\big)\ 0 \end{array}$$

Both formulas have one free second-order variable 0 (modeling a) that is lifted when passing two or three quantifiers. The generated algorithm shows the equivalence of both formulas to T within milliseconds under both semantics.

In order to explore the limits of our decision procedure, formulas over more atomic predicates are required. Therefore, we consider the distributivity theorems of $\Box$ over n-ary implications as shown in Figure 7.1. The size of φ_n grows linearly with n. So does the number of free variables in φ_n. For the underlying alphabet Σ_n, however, this implies exponential growth.

$$\begin{array}{l} \varphi_1 = \forall_1\ \big(\Box\,(\lambda t.\, t \in 2) \boxed{\to} \Box\,(\lambda t.\, t \in 2)\big)\ 0 \\ \varphi_2 = \forall_1\ \big(\Box\,(\lambda t.\, t \in 2 \to t \in 3) \boxed{\to} \Box\,(\lambda t.\, t \in 2) \boxed{\to} \Box\,(\lambda t.\, t \in 3)\big)\ 0 \\ \varphi_3 = \forall_1\ \big(\Box\,(\lambda t.\, t \in 2 \to t \in 3 \to t \in 4) \boxed{\to} \Box\,(\lambda t.\, t \in 2) \boxed{\to} \Box\,(\lambda t.\, t \in 3) \boxed{\to} \Box\,(\lambda t.\, t \in 4)\big)\ 0 \end{array}$$

Figure 7.1: Definition of φ_n

n	size	ICFP 2013	Thm	Thm_P	$\mathsf{Thm}_\mathsf{dual}$
1	54/31907	0/–	0/224.4	0/ 22.6	0/ 0.4
2	81/48797	2/–	0.3/ –	0/434.1	0/14.4
3	108/65687	2640/–	64.4/ –	3.9/ –	17.8/ –

Figure 7.2: Benchmarks for φ_n (under M2L/WS1S semantics)

Formulas φ_i are theorems under both semantics. The running times of the decision procedure Thm (using ACI normalization) in seconds are summarized in Figure 7.2 (column Thm, first number refers to the M2L semantics, the second to WS1S). The column "ICFP 2013" recapitulates the running times from the earlier unoptimized version[1] of this procedure [113].

Figures 7.2 also shows the sizes (column size counting the number of constructors) of the regular expressions generated from the input formulas. These numbers show a huge gap between WS1S and M2L that also shows up in the runtime results. Our implementation of $\mathcal{Q}$ is very inefficient. As future work, we plan to investigate the addition of this regular operator as a constructor to the data type of regular expressions, similarly to our addition of the projection operator.

The last two columns show the running times of two variations of Thm: Thm_P using the $\langle\!\langle\!\langle - \rangle\!\rangle\!\rangle$ normalization instead of $\langle - \rangle$ and $\mathsf{Thm}_\mathsf{dual}$ working with dual regular expressions introduced at the end of Chapter 6. Both guarantee only partial correctness. The procedure $\mathsf{Thm}_\mathsf{dual}$ seems to be the better choice for the WS1S semantics.

The attentive reader will have noticed that we have said nothing about how sets are represented in the code generated from our mathematical definitions. We use the default implementation as lists (with a linear membership test) from Isabelle's library for our measurements. We have also experimented with an existing verified red–black tree implementation. Isabelle's code generator supports the transparent replacement of sets by some verified implementation [53]. Unfortunately, the overhead incurred by the trees outweighed the gain of a logarithmic membership test instead of a linear one.

The performance of our automatically generated code may appear disappointing but that would be a misunderstanding of our overall intentions. We see our work primarily as a succinct and elegant functional program that may pave the way towards verified and efficient decision procedures. As a bonus, the generated code is applicable to small examples. In the context of interactive theorem proving, this is primarily what one encounters: small formulas. Any automation is welcome here because it saves the user time and effort. Automatic verification of larger systems is the domain of highly tuned implementations such as MONA.

[1] This early version, for example, does not employ the *atom* abstraction, but works instead with big unions of singleton atoms.

As the historians have it all mixed up, I repeat:
Whether you like it or not, the automata are
there, in the inside of MT1[1] formulas.

— Julius R. Büchi (1983)

Do you think I am an automaton?
—a machine without feelings?

— Charlotte Brontë, *Jane Eyre* (1847)

Chapter 8

Derivatives of Formulas

The automata that are "in the inside" of WMSO formulas are traditionally[2] exposed by a recursive translation of formulas into automata—the classical logic-automaton connection [26, 58]. A subsequent (or eager intermediate as in case of MONA) minimization of the resulting automata completes the traditional decision procedures. However, formulas are more than just automata: the may have no feelings but they do carry a rich algebraic structure including binders and high-level constructs. During the translation all the rich structure is lost. On the other hand, the subsequent minimization might have benefited from some simplifications on the formula level. As an example, given a formula $\psi = \exists x.\ \varphi$ where φ does not contain x it is immediately clear that ψ is equivalent to φ. When translated to automata, however this equivalence only becomes visible after performing the minimization, the running time of which necessarily will depend on the size of φ's translation.

Concerning the algebraic structure, regular expressions are situated somewhere in between WMSO formulas and automata. In the previous chapter, we have presented a semantics-preserving translation of formulas into Π-extended regular expressions. Thereby, equivalence of formulas was reduced to equivalence of regular expressions. Still, the above example $\psi = \exists x.\ \varphi$ suffers from the same problem: it is hard (if not impossible) to tell whether $\Pi\ (\mathsf{mkRE}_{<}\ \varphi)$ is equivalent to $\mathsf{mkRE}_{<}\ \varphi$ without running a general decision procedure.

Here, we avoid translating formulas altogether. Instead we define a coalgebraic structure—notions of derivatives (Section 8.1) and empty word acceptance (Section 8.2)—directly on WMSO formulas. From this coalgebraic structure we obtain decision procedures for the two semantics WS1S and M2L by instantiating our framework (Section 8.3). Then we show how altering the modeling of interpretations slightly (which also induces a change in the coalgebraic structure) leads to

[1]MT1 is just another name for WS1S.

[2]The only notable exception, we are aware of, is the procedure implemented in the Toss tool [46].

an empirically more efficient algorithm (Section 8.4). We conclude the chapter with two case studies: one investigating the proposed procedure in a more realistic albeit unverified setting (Section 8.5); the other stepping out of the realm of WMSO and devising a derivative operation for formulas of Presburger arithmetic (Section 8.6). This chapter includes the material presented at CSL 2015 [111].

8.1 Formula Derivatives

The syntax Φ and the two semantics $\models$ and $\models_<$ of our input formulas stay the same as in Chapter 7. However, we will supply the coalgebraic structure for a slightly modified syntax Ψ and simplified semantics $\Vdash$ and $\Vdash_<$, which we define next. In Section 8.3 a simple translation will connect the two variants by instantiating the parameter i of our locale. The syntax, which still uses de Bruijn indices, is extended with few additional base cases. Moreover, we want to factor out at first the native support for first-order quantification. We can then think of all variables as being second-order, although we keep the convention of writing lower-case variables where the intuition demands a first-order variable. Therefore in Ψ there is only one existential quantifier corresponding to the second-order quantifier of Φ.

$$\texttt{datatype}\ \Psi = \mathsf{T} \mid \mathsf{F} \mid \mathsf{FO}\ \mathit{var} \mid \mathsf{Z}\ \mathit{var} \mid \mathit{var} < \mathit{var} \mid \mathit{var} \in \mathit{var} \mid \Psi \vee \Psi \mid \neg\, \Psi \mid \exists\, \Psi$$

The additional base formulas T and F represent truth and falsity, respectively. The formula FO x asserts that x is a first-order variable: it is satisfied by all interpretations that assign a singleton finite set to x. The formula Z x is satisfied by all interpretations that assign the empty set to x. We slightly postpone the explanation of its usefulness. We define the *simplified WS1S semantics* formally as follows.

$$
\begin{aligned}
&\texttt{primrec}\ \Vdash\ ::\ \mathit{interp} \to \Psi \to \mathit{bool}\ \texttt{where}\\
&\quad I \Vdash \mathsf{T} = \top\\
&\quad I \Vdash \mathsf{F} = \bot\\
&\quad I \Vdash (\mathsf{FO}\ x) = \mathsf{sing}\ (I[x])\\
&\quad I \Vdash (\mathsf{Z}\ x) = (I[x] = \{\})\\
&\quad I \Vdash (x < y) = \mathsf{sing}\ (I[x]) \wedge \mathsf{sing}\ (I[y]) \wedge \mathsf{it}\ (I[x]) < \mathsf{it}\ (I[y])\\
&\quad I \Vdash (x \in X) = \mathsf{sing}\ (I[x]) \wedge \mathsf{it}\ (I[x]) \in I[X]\\
&\quad I \Vdash (\varphi \vee \psi) = I \Vdash \varphi \vee I \Vdash \psi\\
&\quad I \Vdash (\neg\, \varphi) = \neg (I \Vdash \varphi)\\
&\quad I \Vdash (\exists\, \varphi) = \exists P.\ |P| < \infty \wedge (P \# I) \Vdash \varphi
\end{aligned}
$$

As before the *simplified M2L semantics* $\Vdash_<$ only differs in the quantifier case: it restricts the quantified values to be bounded by $\#I$.

$$I \Vdash_< (\exists\, \varphi) = \exists P.\ (\forall p \in P.\ p < \#I) \wedge (P \# I) \Vdash_< \varphi$$

The *simplified WS1S language* of a formula is then defined as $\mathsf{L}^{\Psi}\ n\ \varphi = \{\mathsf{w}_I \mid I \Vdash \varphi \wedge |I| = n\}$ and the *simplified M2L language* similarly as $\mathsf{L}^{\Psi}_{<}\ n\ \varphi = \{\mathsf{w}_I \mid I \Vdash_{<} \varphi \wedge |I| = n\}$. Since there are only second-order variables, no wellformedness conditions on the interpretation are necessary except for the requirement to assign a value to all the free variables of an n-wellformed formulas. The *n-wellformedness* of formulas wf^{Ψ} merely checks that all free variables are bounded by n.

$$
\begin{aligned}
&\texttt{primrec}\ \mathsf{wf}^{\Psi} :: \mathit{nat} \to \Psi \to \mathit{bool}\ \texttt{where}\\
&\quad \mathsf{wf}^{\Psi}\ n\ \mathsf{T} = \top\\
&\quad \mathsf{wf}^{\Psi}\ n\ \mathsf{F} = \top\\
&\quad \mathsf{wf}^{\Psi}\ n\ (\mathsf{FO}\ m) = m < n\\
&\quad \mathsf{wf}^{\Psi}\ n\ (\mathsf{Z}\ m) = m < n\\
&\quad \mathsf{wf}^{\Psi}\ n\ (m_1 < m_2) = m_1 < n \wedge m_2 < n\\
&\quad \mathsf{wf}^{\Psi}\ n\ (m \in M) = m < n \wedge M < n\\
&\quad \mathsf{wf}^{\Psi}\ n\ (\varphi \vee \psi) = \mathsf{wf}^{\Psi}\ n\ \varphi \wedge \mathsf{wf}^{\Psi}\ n\ \psi\\
&\quad \mathsf{wf}^{\Psi}\ n\ (\neg\ \varphi) = \mathsf{wf}^{\Psi}\ n\ \varphi\\
&\quad \mathsf{wf}^{\Psi}\ n\ (\exists\ \varphi) = \mathsf{wf}^{\Psi}\ (n+1)\ \varphi
\end{aligned}
$$

Finally, we are ready to introduce the central notion of this chapter: the derivative of a formula. Brzozowski derivatives compute symbolically for a regular expression r the regular expression $\delta\ a\ r$ whose language is the left quotient of r's language by a. A *formula derivative* is the analogous operation on a formula. Given a formula φ, its derivative $\delta\ v\ \varphi$ is a formula whose language is the left quotient of φ's language by the letter $v : \mathit{bool}^n$, where n is the number of free variables of φ. Formula derivatives are defined by primitive recursion as follows.

$$
\begin{aligned}
&\texttt{primrec}\ \delta :: \mathit{bool\ list} \to \Psi \to \Psi\ \texttt{where}\\
&\quad \delta\ v\ \mathsf{T} = \mathsf{T}\\
&\quad \delta\ v\ \mathsf{F} = \mathsf{F}\\
&\quad \delta\ v\ (\mathsf{Z}\ x) = \begin{cases} \mathsf{F} & \text{if } v[x]\\ \mathsf{Z}\ x & \text{otherwise}\end{cases}\\
&\quad \delta\ v\ (\mathsf{FO}\ x) = \begin{cases} \mathsf{Z}\ x & \text{if } v[x]\\ \mathsf{FO}\ x & \text{otherwise}\end{cases}\\
&\quad \delta\ v\ (x < y) = \begin{cases} x < y & \text{if } \neg v[x] \wedge \neg v[y]\\ \mathsf{Z}\ x \wedge \mathsf{FO}\ y & \text{if } v[x] \wedge \neg v[y]\\ \mathsf{F} & \text{otherwise}\end{cases}\\
&\quad \delta\ v\ (x \in X) = \begin{cases} x \in X & \text{if } \neg v[x]\\ \mathsf{Z}\ x & \text{if } v[x] \wedge v[X]\\ \mathsf{F} & \text{otherwise}\end{cases}\\
&\quad \delta\ v\ (\varphi \vee \psi) = \delta\ v\ \varphi \vee \delta\ v\ \psi\\
&\quad \delta\ v\ (\neg\ \varphi) = \neg\ \delta\ v\ \varphi\\
&\quad \delta\ v\ (\exists\ \varphi) = \exists\ (\delta\ (\top \mathbin{\#} v)\ \varphi \vee \delta\ (\bot \mathbin{\#} v)\ \varphi)
\end{aligned}
$$

Here we see the first important usage of the FO x and Z x: to establish the closure of Ψ under δ.

Let us try to intuitively understand some of the base cases of the definition. The language of F is the empty set. The quotient of the empty set is again empty. Thus, the derivative of F should be again F. For FO x, the language will contain all words which have exactly one $\top$ in the x-th row in some letter. Thus, if we quotient this language by a letter that has a $\bot$ in the x-th row, we should obtain the same language represented by formula FO x. However, once the first $\top$ in the x-th row has been discovered by quotienting by a letter v that has a $\top$ in the x-th row (written $v[x]$), the remaining words to be accepted must have only $\bot$'s in the x-th row—thus the quotient is the language represented by formula Z x.

Other base cases follow similarly. The most interesting recursive case is that of the existential quantifier. The first observation that should be made is that if φ is a formula with $n+1$ free variables, then $\exists\ \varphi$ is a formula with n free variables (assuming that the bound variable 0 is used in φ). This implies that when we derive $\exists\ \varphi$ by v, we cannot derive φ by v, but only by some vector $b \# v$ because all variables (de Bruijn indices) are shifted. Since the Boolean b is unknown, we consider both possibilities: $b = \top$ and $b = \bot$. The existential quantifier requires just one witness assignment for the quantified variable. This means only one of the possibilities has to hold—that is why the recursive calls $\delta\ (\top \# v)\ \varphi$ and $\delta\ (\bot \# v)\ \varphi$ are connected by disjunction.

To reason more directly about interpretations and derivatives, we lift the list constructor # on words to interpretations. This yields the operator CONS, with the characteristic property $w_{(\mathsf{CONS}\ v\ I)} = v \# \mathrm{w}_I$ assuming $|v| = |I|$, defined as follows.

```
fun CONS :: bool list → interp → interp where
  CONS [] [] = []
  CONS (⊤ # v) (X # I) = ({0} ∪ (+1) • X) # CONS v I
  CONS (⊥ # v) (X # I) = ((+1) • X) # CONS v I
```

Additionally, we require $\#(\mathsf{CONS}\ v\ I) = \#I + 1$. We obtain the following key characterization of the derivative by two straightforward inductions on φ.

```
theorem wf^Ψ n φ ∧ |I| = |v| = n ⟶ I ⊩ δ v φ ↔ CONS v I ⊩ φ
theorem wf^Ψ n φ ∧ |I| = |v| = n ⟶ I ⊩< δ v φ ↔ CONS v I ⊩< φ
```

In general, the set of all word derivatives $\{\mathsf{fold}\ \delta\ w\ \varphi \mid w :: \mathit{bool\ list\ list}\}$ can be infinite. However, Brzozowski's fundamental result about dissimilar derivatives or regular expressions carries over: there are finitely many distinct word derivatives modulo associativity, commutativity and idempotence of $\vee$ (ACI, written $\sim$). As with regular expressions, we implement an ACI normalization function $\langle_\rangle$ via smart

constructors (the formal definition is very similar to the one on regular expressions an therefore omitted).

theorem finite $\{\langle \mathsf{fold}\ \delta\ w\ \varphi \rangle \mid w :: \mathit{bool\ list\ list}\}$

This theorem is proved by induction on φ. All cases except for $\exists\ \varphi$ are either obvious or carry over smoothly from Brzozowski's proof. The proof of the existential quantifier case resembles the projection case in the finiteness proof for Π-extended regular expressions. By induction hypothesis we know that φ has a finite set D of distinct derivatives modulo ACI. Some of the formulas in D can be disjunctions. If we repeatedly split outermost disjunctions in D until none are left, we obtain a finite set X of disjuncts. Each word derivative fold $\delta\ w\ (\exists\ \varphi)$ is ACI equivalent to some $\exists\ (\bigvee Y)$ for some $Y \subseteq X$. Since X is finite, its powerset is also finite. Hence, there are finitely many distinct fold $\delta\ w\ (\exists\ \varphi)$ modulo ACI.

Note that the above theorem (also the finiteness theorems for different structures) is purely syntactic: no semantic (or language) equivalence is involved. In particular, this theorem does not follow from the Myhill-Nerode theorem nor from the fact that there are only finitely many semantically inequivalent formulas of bounded quantifier rank. Although not all language equivalent formulas are also ACI equivalent, ACI suffices for gaining finiteness.

Example 8.1. *The formula $\varphi = \exists\ 1 \in 0$ (or $\exists X.\ x \in X$ in conventional notation) has three distinct derivatives modulo ACI over the alphabet $\Sigma_1 = \mathit{bool}^1 = [(\bot), (\top)]$ (we abbreviate* fold $\delta\ w$ *by δ_w). Note that the subformula $1 \in 0$ is derived by letters of the alphabet $\Sigma_2 = \mathit{bool}^2$ that consists of all Boolean vectors of length two.*

- $\delta_{[]}\ \varphi = \varphi \sim \exists\ (1 \in 0 \vee 1 \in 0) = \delta_{[(\bot)]}\ \varphi \sim \delta_{(\bot)^n}\ \varphi$
  for all n
- $\delta_{[(\top)]}\ \varphi = \exists\ (\mathsf{Z}\ 1 \vee \mathsf{F}) \sim \exists\ ((\mathsf{Z}\ 1 \vee \mathsf{F}) \vee (\mathsf{Z}\ 1 \vee \mathsf{F})) \sim \delta_{((\bot)^n @ (\top) \# (\bot)^m)}\ \varphi$
  for all m, n
- $\delta_{[(\top),(\top)]}\ \varphi = \exists\ (\mathsf{F} \vee \mathsf{F}) \vee (\mathsf{F} \vee \mathsf{F}) \sim \exists\ \mathsf{F} \sim \delta_{((\bot)^n @ (\top) \# (\bot)^m @ (\top) \# w)}\ \varphi$
  for all m, n, w

Figure 8.1 shows the bisimulation induced by formula derivatives (as our forthcoming decision procedure will produce it) for the above formula $\varphi = \exists\ 1 \in 0$ and its equivalent $\psi = \mathsf{FO}\ 0$ over the alphabet $\Sigma_1 = \mathit{bool}^1 = [(\top), (\bot)]$. The bisimulation is depicted as the product automaton of the automata whose states are derivatives of the formulas with transitions given by δ. Accepting states are pairs (φ, ψ) for which $o\ \varphi$ and $o\ \psi$ holds (the definition of o is the topic of the next section). They are marked with a double margin. Previously seen ACI equivalent states, detected using the ACI normalization $\langle _ \rangle$ and marked by a dashed backedge, are not explored further (and not even checked for being both accepting or both not accepting). Self-loops annotated with $\langle _ \rangle$ are omitted.

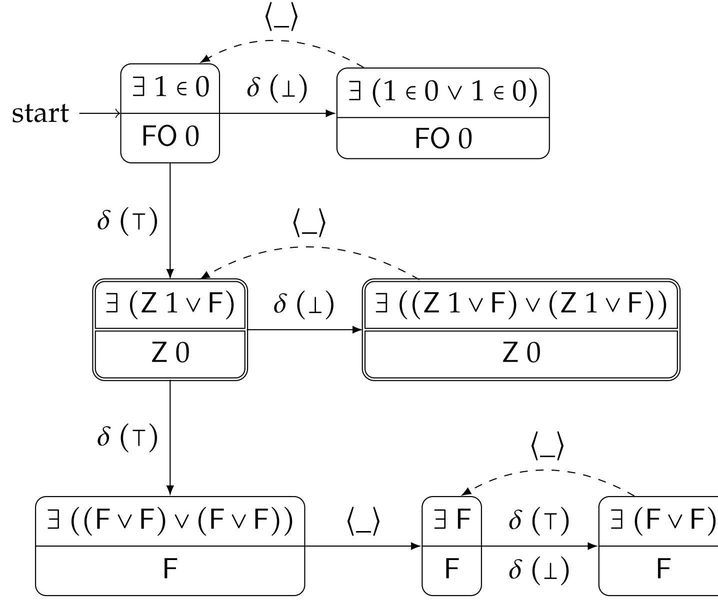


Figure 8.1: Bisimilarity of $\exists\ 1 \in 0$ and $\mathsf{FO}\ 0$ via formula derivatives

8.2 Accepting Formulas

Defining a function that checks whether a regular expression accepts the empty word is straightforward. A priori, the task seems barely harder for formulas—check whether the empty interpretation $\{\}^n$ (that is encoded as the empty word) satisfies a formula with n free variables. We start with the following attempt.

$$
\begin{aligned}
&o_< \ \mathsf{T} = \top \\
&o_< \ \mathsf{F} = \bot \\
&o_< \ (\mathsf{FO}\ x) = \bot \\
&o_< \ (\mathsf{Z}\ x) = \top \\
&o_< \ (x < y) = \bot \\
&o_< \ (x \in X) = \bot \\
&o_< \ (\varphi \vee \psi) = o_< \ \varphi \vee o_< \ \psi \\
&o_< \ (\neg\ \varphi) = \neg o_< \ \varphi \\
&o_< \ (\exists\ \varphi) = o_< \ \varphi
\end{aligned}
$$

As the name of the function indicates this acceptance test will work only for the M2L semantics, justifying M2L's automata theoretic reputation. As we will see the acceptance test for the number theoretic WS1S is by far more involved.

First, we should understand why $o_<$ works well for M2L but fails for WS1S. Therefore, we consider the formula $\varphi = \exists\ (1 < 0)$ ($= \exists y.\ x < y$ in conventional notation) that behaves differently under the two semantics. Interpreted in WS1S, φ is true:

for each natural number x there exists a bigger one. Interpreted in M2L, the fact whether φ is true depends on the interpretation I: for example if $\#I = 4$ then the quantifier is allowed to assign to y only values smaller than 4. If additionally $I[x] = \{3\}$, the formula becomes false. When checking acceptance, we only consider interpretations I with $\#I = 0$. Therefore, $o_<$ rightly returns $\bot$ in the M2L setting for our particular formula φ.

Asserting $o_< (\exists\, \varphi) = o_< \varphi$ effectively corresponds to the restriction of M2L on the quantified values. Fortunately, unrestricted WS1S quantifiers can be related to M2L quantifiers by repeatedly padding the interpretations I with the letter $0^{|I|}$. This means that we can reuse $o_<$ for WS1S if we can get rid of the padding. In the context of automata this step is called futurization and is implemented by traversing the automaton backwards using only $0^{|I|}$-labeled transitions [68]. Here, we work with formulas. Thus traversing means deriving and traversing backwards means deriving from the right. We therefore introduce a second derivative operation ƍ (a mirrored δ) that computes the right quotient and is symmetric to the derivative δ. The symmetry is caused by the fact that all encodings of models of our formulas are nicely symmetric. In Section 8.4 we will modify the encodings with the goal of obtaining a more efficient procedure, at the expense of giving up the symmetry.

$$
\begin{array}{l}
\texttt{primrec}\ ƍ :: \mathit{bool\ list} \to \Psi \to \Psi\ \texttt{where} \\
\quad ƍ\ v\ \mathsf{T} = \mathsf{T} \\
\quad ƍ\ v\ \mathsf{F} = \mathsf{F} \\
\quad ƍ\ v\ (\mathsf{FO}\ x) = \begin{cases} \mathsf{Z}\ x & \text{if } v[x] \\ \mathsf{FO}\ x & \text{otherwise} \end{cases} \\
\quad ƍ\ v\ (\mathsf{Z}\ x) = \begin{cases} \mathsf{F} & \text{if } v[x] \\ \mathsf{Z}\ x & \text{otherwise} \end{cases} \\
\quad ƍ\ v\ (x < y) = \begin{cases} x < y & \text{if } \neg v[x] \wedge \neg v[y] \\ \mathsf{FO}\ x \wedge \mathsf{Z}\ y & \text{if } \neg v[x] \wedge v[y] \\ \mathsf{F} & \text{otherwise} \end{cases} \\
\quad ƍ\ v\ (x \in X) = \begin{cases} x \in X & \text{if } \neg v[x] \\ \mathsf{Z}\ x & \text{if } v[x] \wedge v[X] \\ \mathsf{F} & \text{otherwise} \end{cases} \\
\quad ƍ\ v\ (\varphi \vee \psi) = ƍ\ v\ \varphi \vee ƍ\ v\ \psi \\
\quad ƍ\ v\ (\neg\, \varphi) = \neg\, ƍ\ v\ \varphi \\
\quad ƍ\ v\ (\exists\, \varphi) = \exists\ (ƍ\ (\top \# v)\ \varphi \vee ƍ\ (\bot \# v)\ \varphi)
\end{array}
$$

A first thing worth noticing is that again the number of right derivatives ƍ modulo ACI of $\vee$ is finite. The proof is analogous to the one for left derivatives.

$$\texttt{theorem}\ \mathsf{finite}\ \{\langle \mathsf{fold}\ ƍ\ w\ \varphi \rangle \mid w :: \mathit{bool\ list\ list}\}$$

Next, we establish a correspondence between the M2L semantics and right derivatives. Therefore, we introduce the operation SNOC, symmetric to CONS that was used for left derivatives, on interpretations.

```
fun SNOC :: bool list → interp → interp where
    SNOC [] [] = []
    SNOC (⊤ # v) (X # I) = ({#I} ∪ X) # SNOC v I
    SNOC (⊥ # v) (X # I) = X # SNOC v I
```

Just as for CONS, we require $\#(\mathsf{SNOC}\ v\ I) = \#I + 1$. Another routine induction yields the fundamental property of right derivatives. Note that there is no equivalent theorem for the WS1S semantics.

```
theorem wf^Ψ n φ ∧ |I| = |v| = n ⟶ I ⊩< ☡ v φ ↔ SNOC v I ⊩< φ
```

Finally, we define a function futurize to repeatedly apply $☡\ 0^n$ for a given n (using once again the while combinator), an operation $\lfloor_\rfloor_$ to recursively apply futurize to all quantifiers in a formula, and the acceptance test o for WS1S formulas. The equations of $\lfloor_\rfloor_$ are matched sequentially.

```
definition futurize :: nat → Ψ → Ψ where
    futurize n φ =
        let (_, X) = the
            (while (λ(φ, X). φ ∉ X) (λ(φ, X). (☡ 0^n φ, {φ} ∪ X)) (φ, {}))
        in ⋁X

fun ⌊_⌋_ :: Ψ → nat → Ψ where
    ⌊φ ∨ ψ⌋_n = ⌊φ⌋_n ∨ ⌊ψ⌋_n
    ⌊¬ φ⌋_n = ¬ ⌊φ⌋_n
    ⌊∃ φ⌋_n = futurize n (∃ ⌊φ⌋_(n+1))
    ⌊φ⌋_n = φ

definition o :: nat → Ψ → bool where
    o n φ = o< ⌊φ⌋_n
```

Because the set of right derivatives modulo ACI is finite the function futurize does always terminate. Moreover, we can prove the expected properties of the acceptance tests. We perform the proof in four steps. Below, $(\mathsf{SNOC}\ (0^n))^k$ denotes the k-fold repeated application of $\mathsf{SNOC}\ (0^n)$. Moreover, note that in case $\#I = 0$ and $|I| = n$ holds, I necessarily must have the shape $\{\}^n$ (also we have $\mathsf{w}_I = []$).

1. $\mathsf{wf}^{\Psi}\ n\ \varphi \wedge |I| = n \wedge \#I = 0 \longrightarrow o_<\ \varphi \leftrightarrow I \Vdash_< \varphi$,
2. $\mathsf{wf}^{\Psi}\ n\ \varphi \wedge |I| = n \longrightarrow I \Vdash_< \mathsf{futurize}\ n\ \varphi \leftrightarrow \exists k.\ ((\mathsf{SNOC}\ (0^n))^k\ I) \Vdash_< \varphi$,

3. $\mathsf{wf}^{\Psi}\ n\ \varphi \wedge |I| = n \longrightarrow I \Vdash_{<} \lfloor\varphi\rfloor_n \leftrightarrow I \Vdash \varphi$, and
4. $\mathsf{wf}^{\Psi}\ n\ \varphi \wedge |I| = n \wedge \#I = 0 \longrightarrow o\ n\ \varphi \leftrightarrow I \Vdash \varphi$,

Prop. 1 follows by a routine induction on φ. Prop. 2 follows from the fact that futurize $n\ \varphi$ is the disjunction of all right derivatives of φ by words of the form $(0^n)^k$ for some k and the characteristic property of right derivatives. Prop. 3 is again a routine induction using Prop. 2 in the existential quantifier cases. Finally, for Prop. 4 we calculate: $o\ n\ \varphi \leftrightarrow o_{<}\ \lfloor\varphi\rfloor_n \overset{\text{Prop. 1}}{\leftrightarrow} I \Vdash_{<} \lfloor\varphi\rfloor_n \overset{\text{Prop. 3}}{\leftrightarrow} I \Vdash \varphi$.

8.3 Decision Procedure

We are close to being able to instantiate the generic bisimulation computation with our notions to obtain decision procedures for WS1S and M2L. The only missing jigsaw piece is the treatment of first-order quantification that was swept under the carpet so far. The solution is rather straightforward: in our new syntax Ψ there are no first-order quantifiers, but there is a base formula that asserts that a variable is first order. Therefore, when translating formulas in the standard syntax Φ introduced in Chapter 7 into our new syntax we insert additional *restrictions* FO x in the appropriate places into the formula. This approach is inspired by the treatment of the issue of first-order variables in MONA (although recent versions of MONA use a more efficient solution than the outlined simple approach [68]). Restrictions are inserted under each first-order quantifier by the function restr, but also globally to annotate free first-order variables by the function restrict.

```
primrec restr :: Φ → Ψ where
    restr (φ ∨ ψ) = restr φ ∨ restr ψ
    restr (¬ φ) = ¬ restr φ
    restr (∃₁ φ) = ∃ (restr φ ∧ FO 0)
    restr (∃₂ φ) = ∃ (restr φ)
    restr φ = φ
definition restrict :: Φ → Ψ where
    restrict φ = fold (λi φ. φ ∧ FO i) (𝒱₁ φ) (restr φ)
```

Note that the last equation of restr uses some notation overloading for constructors since the type of the formula φ does change. The key property of restrict is that it translate between the semantics $\vDash$ and $\Vdash$.

```
theorem L^Ψ n (restrict φ) = L^Φ n φ
theorem L^Ψ_< n (restrict φ) = L^Φ_< n φ
```

Finally, we are ready to instantiate our locale to obtain decision procedures for the M2L and the WS1S semantics. As in the previous chapter the alphabet Σ_n denotes the list of Boolean vectors of length n.

`interpretation` WS1S : $\mathsf{fin_coalg}_D$ `where`	`interpretation` M2L : $\mathsf{fin_coalg}_D$ `where`
$\Sigma = \Sigma_n$	$\Sigma = \Sigma_{n+1}$
$\mathsf{i}\ \varphi = \langle \mathsf{restrict}\ \varphi \rangle$	$\mathsf{i}\ \varphi = \langle \mathsf{restrict}\ \varphi \wedge \mathsf{FO}\ n \rangle$
$\mathsf{o}\ \psi = o\ n\ \psi$	$\mathsf{o}\ \psi = o_<\ \psi$
$\mathsf{d}\ v\ \psi = \langle \delta\ v\ \psi \rangle$	$\mathsf{d}\ v\ \psi = \langle \delta\ v\ \psi \rangle$
$\mathsf{L}^\sigma\ \psi = \mathsf{L}^\Psi\ n\ \psi$	$\mathsf{L}^\sigma\ \psi = \mathsf{L}^\Psi_<\ (n+1)\ \psi$
$\mathsf{L}^\tau\ \varphi = \mathsf{L}^\Phi\ n\ \varphi$	$\mathsf{L}^\tau\ \varphi = \mathsf{L}^\Phi_<\ n\ \varphi$
$\mathsf{wf}^\sigma\ \psi = \mathsf{wf}^\Psi\ n\ \psi$	$\mathsf{wf}^\sigma\ \psi = \mathsf{wf}^\Psi\ (n+1)\ \psi$
$\mathsf{wf}^\tau\ \varphi = \mathsf{wf}^\Phi\ n\ \varphi$	$\mathsf{wf}^\tau\ \varphi = \mathsf{wf}^\Phi\ n\ \varphi$

The last point deserving an explanation is the conjunction of the fresh variable with the index n (not occurring in n-wellformed φ) in the decision procedure for M2L. This is required to circumvent the situation that a M2L formula can only be true in the empty model if it does not quantify over first-order variables and has no free first-order variables. Because of this $\forall_1$ FO 0 would be different from T but equivalent to FO 42. Conjoining a fresh first-order variable restriction essentially means, that the first pair of a bisimulation must not be checked for acceptance. With other words, the empty word is explicitly excluded as in Chapter 7. This admittedly surprising workaround is required due to the unusual quantification rules of M2L. In WS1S there is nothing special about empty models.

Example 8.1 suggests that a stronger normalization function than $\langle_\rangle$ might be useful. For example, trivial identities involving T or F (such as $\mathsf{T} \vee \varphi \equiv \varphi$) should be also simplified. Indeed in our tests we employ a much stronger normalization function, that additionally pushes negations inside (which requires adding universal quantification and conjunction as primitives) and eliminates needless quantifiers $\exists\ \varphi \equiv \varphi$ provided that 0 is not free in φ. MONA performs a similar optimization, which according to its user manual [69] "can cause tremendous improvements". However, MONA can perform this optimization only on the initially entered formula, while for us it is available for every pair of formulas in the bisimulation because we keep the rich formula structure.

This is only one of a wealth of optimizations performed in MONA. Competing with this state-of-the-art tool is not our main objective yet. Our procedure does outperform MONA on specific examples, for example on formulas of the form $\varphi \wedge \psi$ where the minimal automata for φ is small, the one for ψ is huge, and for almost all w either fold $\delta\ w\ \varphi = \mathsf{F}$ or fold $\delta\ w\ \psi = \mathsf{F}$. When extending the syntax with formulas of the form $x = n$ for constants n (and defining the left and right derivatives for them), such examples can be immediately constructed: $x = 10 \wedge x = 10000$ takes MONA roughly 40 minutes to disproof, while our procedure requires only a few seconds. MONA computes eagerly both minimal automata for φ and ψ, whereas our procedures is

	regular expression-based		formula derivative-based	
n	M2L	WS1S	M2L	WS1S
0	0	0.4	0	0
1	0	14.4	0	0.5
2	3.9	–	0.8	0.9
3	–	–	1.5	1.8
4	–	–	3.1	4.5

Figure 8.2: Running times (in sec) for proving $\psi_n \equiv \top$

lazier, which pays off here. More generally, we can compete with and outperform by far our own procedure based on Π-extended regular expressions.

We have evaluated the performance of our new procedure on the family of formulas ψ_n defined in Chapter 7 against the best timings obtained for M2L and WS1S by the procedure based on Π-extended regular expressions. The encouraging results are shown in Figure 8.2. A "–" means that the computation did not terminate within five minutes. MONA solves all those examples instantly. We should note that the optimization $\exists\, \varphi \equiv \varphi$ provided that 0 is not free in φ is indeed a tremendous improvement—without its usage our decision procedure scales only slightly better than the regular expression-based one. The possibility to inspect the binder structure after several derivative steps, which is not available when translating to regular expressions, is the main advantage of the proposed decision procedure.

8.4 Minimum Semantics Optimization

MONA is an excellent source for ideas on how to optimize any algorithm dealing with WMSO. Here, we consider a particular optimization which we call the *minimum semantics* of first-order variables. MONA's manual [69] gives the following description for this optimization.

> First-order values are not encoded as singleton sets as suggested, but as non-empty sets. The value denoted by such a set is interpreted as its smallest number. (This encoding turns out to be slightly more efficient than the singleton method.)

Although this is not the most effective optimization that MONA has to offer, we chose to study it in our setting since it directly affects the coalgebraic structure that we have defined for formulas. Indeed, implementing the optimization will make the derivative operation simpler, at the expense of a complication of right derivatives and therefore of the empty word (model) acceptance test. Empirically, the gains seem to outweigh the losses.

Formally, we define the minimum semantics as follows on formulas of type Ψ. We omit the atomic formula $\mathsf{Z}\ x$—it will not be necessary to establish the closure under derivatives for the minimum semantics and can therefore be safely removed from Ψ (also since the translation restrict does not generate it as a subformula).

$$
\begin{aligned}
&\texttt{primrec}\ \Vdash\ ::\ \mathit{interp} \to \Psi \to \mathit{bool}\ \texttt{where}\\
&\quad I \Vdash \mathsf{T} = \top\\
&\quad I \Vdash \mathsf{F} = \bot\\
&\quad I \Vdash (\mathsf{FO}\ x) = I[x] \neq \{\}\\
&\quad I \Vdash (x < y) = I[x] \neq \{\} \wedge I[y] \neq \{\} \wedge \mathsf{min}\ (I[x]) < \mathsf{min}\ (I[y])\\
&\quad I \Vdash (x \in X) = I[x] \neq \{\} \wedge \mathsf{min}\ (I[x]) \in I[X]\\
&\quad I \Vdash (\varphi \vee \psi) = I \Vdash \varphi \vee I \Vdash \psi\\
&\quad I \Vdash (\neg\, \varphi) = \neg (I \Vdash \varphi)\\
&\quad I \Vdash (\exists\, \varphi) = \exists P.\ |P| < \infty \wedge (P \# I) \Vdash \varphi
\end{aligned}
$$

The function min denotes the minimal element of a non-empty finite set of numbers.

The semantics of input formulas Φ as well as the actual language definition are also adjusted accordingly (omitted here). Clearly, this change does not influence satisfiability in a pure number-theoretic sense: first-order variables must be assigned a single natural number and the change affects only the formal modeling of those values by finite sets. (The actual word encodings of models do change.)

The effect of the minimum semantics on the derivative operation is best seen on the formula $\mathsf{FO}\ x$. In the standard semantics $\mathsf{FO}\ x$ is satisfied by all interpretations that assign x a singleton set. Thus, after deriving by a letter v that has $\top$ in the row belonging to x, we have to accept only interpretations that assign the empty set to x, represented by the helper formula $\mathsf{Z}\ x$. With the minimum semantics, it does not matter what comes after the first $\top$. This is represented by the formula T. The advantage of the minimum semantics lies in the fact that then arising subformulas T can be eliminated, thereby simplifying the whole formula, while $\mathsf{Z}\ x$ can not be eliminated as easily. We obtain the derivative δ for the minimum semantics simply by replacing all occurrences of the formula $\mathsf{Z}\ x$ in the original definition with T (we only show the constructors for which the definition of δ actually does change).

$$
\delta\ v\ (\mathsf{FO}\ x) = \begin{cases} \mathsf{T} & \text{if } v[x] \\ \mathsf{FO}\ x & \text{otherwise} \end{cases}
$$

$$
\delta\ v\ (x < y) = \begin{cases} x < y & \text{if } \neg v[x] \wedge \neg v[y] \\ \mathsf{FO}\ y & \text{if } v[x] \wedge \neg v[y] \\ \mathsf{F} & \text{otherwise} \end{cases}
$$

$$
\delta\ v\ (x \in X) = \begin{cases} x \in X & \text{if } \neg v[x] \\ \mathsf{T} & \text{if } v[x] \wedge v[X] \\ \mathsf{F} & \text{otherwise} \end{cases}
$$

For right derivatives, the situation is more complex, since the interpretations do not have the property anymore of being symmetric. In particular Ψ as defined so far is not closed under $\overleftarrow{\delta}$—when deriving a first-order formula from the right by a $\top$, it is not clear anymore that the derivative will cancel out the actual assigned value (that is the first $\top$ in a row). To capture this uncertainty, we extend it with three further base formulas: $x <_{\mathsf{F}} y$, $x <_{\mathsf{T}} y$, and $x \in_{\mathsf{T}} y$ with the following WS1S semantics (the M2L semantics is the same, but not really relevant).

$$\begin{aligned}
&I \Vdash (x <_{\mathsf{F}} y) = I[x] \neq \{\} \wedge I[y] = \{\} \vee I \Vdash x < y \\
&I \Vdash (x <_{\mathsf{T}} y) = I[y] = \{\} \vee I \Vdash x < y \\
&I \Vdash (x \in_{\mathsf{T}} X) = I[x] = \{\} \vee I \Vdash x \in X
\end{aligned}$$

In principle, $x <_{\mathsf{F}} y$, $x <_{\mathsf{T}} y$, and $x \in_{\mathsf{T}} X$ are random names for formulas denoting "what is left" after deriving from the right. These remainders are closely related to their non-subscripted cousins, behaving differently only if one of the sets assigned to a first-order variable is empty. The subscript T indicates the acceptance of the empty model.

Next, we adjust the definition of the right derivative $\overleftarrow{\delta}$. We omit the constructors for which $\overleftarrow{\delta}$ is unchanged (Boolean connectives and the existential quantifier). Moreover, the M2L acceptance test $o_<$ must be lifted to the additional formulas. Although we will not use the derivative δ on the new formulas, we want our previously stated theorems still to hold universally. Thus, we also extend δ accordingly.

$$\overleftarrow{\delta}\ v\ (\mathsf{FO}\ x) = \begin{cases} \mathsf{T} & \text{if } v[x] \\ \mathsf{FO}\ x & \text{otherwise} \end{cases}$$

$$\overleftarrow{\delta}\ v\ (x < y) = \begin{cases} x < y & \text{if } \neg v[y] \\ x <_{\mathsf{F}} y & \text{otherwise} \end{cases}$$

$$\overleftarrow{\delta}\ v\ (x <_{\mathsf{F}} y) = \begin{cases} x <_{\mathsf{T}} y & \text{if } v[x] \wedge \neg v[y] \\ x <_{\mathsf{F}} y & \text{otherwise} \end{cases}$$

$$\overleftarrow{\delta}\ v\ (x <_{\mathsf{T}} y) = \begin{cases} x <_{\mathsf{T}} y & \text{if } \neg v[y] \\ x <_{\mathsf{F}} y & \text{otherwise} \end{cases}$$

$$\overleftarrow{\delta}\ v\ (x \in X) = \begin{cases} x \in_{\mathsf{T}} X & \text{if } v[x] \wedge v[X] \\ x \in X & \text{otherwise} \end{cases}$$

$$\overleftarrow{\delta}\ v\ (x \in_{\mathsf{T}} X) = \begin{cases} x \in X & \text{if } v[x] \wedge \neg v[X] \\ x \in_{\mathsf{T}} X & \text{otherwise} \end{cases}$$

$$o_<\ (x <_{\mathsf{F}} y) = \bot$$
$$o_<\ (x <_{\mathsf{T}} y) = \top$$
$$o_<\ (x \in_{\mathsf{T}} X) = \top$$

$$\delta\ v\ (x <_{\mathsf{F}} y) = \begin{cases} x <_{\mathsf{F}} y & \text{if } \neg v[x] \wedge \neg v[y] \\ \mathsf{T} & \text{if } v[x] \wedge \neg v[y] \\ \mathsf{F} & \text{otherwise} \end{cases}$$

$$\delta\ v\ (x <_{\mathsf{T}} y) = \begin{cases} x <_{\mathsf{T}} y & \text{if } \neg v[x] \wedge \neg v[y] \\ \mathsf{T} & \text{if } v[x] \wedge \neg v[y] \\ \mathsf{F} & \text{otherwise} \end{cases}$$

$$\delta\ v\ (x \in_{\mathsf{T}} X) = \begin{cases} x \in_{\mathsf{T}} X & \text{if } \neg v[x] \\ \mathsf{T} & \text{if } v[x] \wedge v[X] \\ \mathsf{F} & \text{otherwise} \end{cases}$$

All other notions required for the decision procedure remain unchanged and we obtain two further interpretations M2L′ and WS1S′ that instantiate the parameters in exactly the same way as M2L and WS1S (however including the above changes to the derivative operations hidden through overloading).

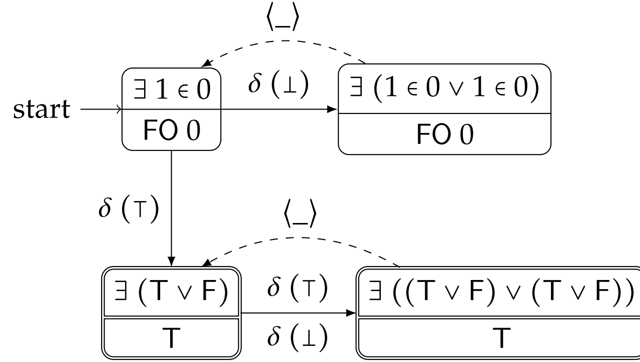


Figure 8.3: Bisimulation for $\exists\ 1 \in 0 \equiv \mathsf{FO}\ 0$

The modified derivatives enjoy all the previously stated properties (with respect to the minimum semantics). Since the modifications affect only base formulas we have invested some effort into structuring our formal proofs in order not to repeat large parts such as the formalization of the futurization step required by o. In the end, we have completely separated the base formulas from the type of formulas that then have a single constructor to capture all base formulas. The communication between those two parts is realized via a locale that abstracts over the type of base formulas α and the implementations of all involved functions, such as $\models$, $\Vdash$, δ, $\eth$, and $o_<$, on α. The decision procedure is then defined abstractly and proved correct once and for all, without knowing how the concrete base formulas look like. The resulting locale is even more complicated, since we also want to reuse it for deciding different logics (Section 8.6)—it has over 20 parameters interconnected via 50 assumptions. Due to this complexity, in this thesis we rather present the different procedures with some redundancy. In formal proofs abstraction helps a lot to focus on exactly what is necessary and in the end is a clear time saver.

Our new procedure for the minimum semantics outperforms the one for the standard semantics. Indication of this are already apparent when viewing Example 8.1 through the lens of the minimal semantics.

Example 8.2. *The formula* $\varphi = \exists\ 1 \in 0$ *(or* $\exists X.\ x \in X$ *in conventional notation) has two distinct derivatives modulo ACI over the alphabet* $\Sigma_1 = bool^1 = [(\bot), (\top)]$ *(we again abbreviate* $\mathsf{fold}\ \delta\ w$ *by* δ_w*).*

- $\delta_{[]}\ \varphi = \varphi \sim \exists\ (1 \in 0 \vee 1 \in 0) = \delta_{[(\bot)]}\ \varphi \sim \delta_{(\bot)^n}\ \varphi$ *for all* n
- $\delta_{[(\top)]}\ \varphi = \exists\ (\mathsf{T} \vee \mathsf{F}) \sim \exists\ ((\mathsf{T} \vee \mathsf{F}) \vee (\mathsf{T} \vee \mathsf{F})) = \delta_{[(\top),(\bot)]}\ \varphi \sim \delta_{((\bot)^n @ (\top) \# w)}\ \varphi$ *for all* w, n

Figure 8.3 shows the bisimulation induced by formula derivatives for the above formula $\varphi = \exists\ 1 \in 0$ *and its equivalent* $\psi = \mathsf{FO}\ 0$ *under the minimum semantics.*

	standard semantics		minimum semantics	
n	M2L	WS1S	M2L	WS1S
0	0	0	0	0
1	0	0.5	0	0
2	0.8	0.9	0	0
3	1.5	1.8	0	0
4	3.1	4.5	0	0
10	–	–	0.1	0.1
15	–	–	3.4	3.5

Figure 8.4: Running times (in sec) for proving $\psi_n \equiv \top$

Similarly, for our running example ψ_n from earlier the minimum semantics constitutes an even further improvement. The results are shown in Figure 8.4. As before, a "–" means that the computation did not terminate within five minutes.

8.5 Case Study: MonaCo

MONA benefits a lot from the BDD representation of automata [70]. Pous [96] presents a fast symbolic decision procedure for Kleene algebra with tests (KAT) employing BDDs whose leafs are KAT expressions and lifting derivative operations of KAT expressions to those BDDs. The coalgebraic core of this decision procedure is identical to the one we formalized.

A rather natural experiment is to combine our derivative operations on formulas with the generic infrastructure, in form of a Ocaml library called safa[1] [95], developed by Pous for his procedure. Together with Pous we are in the process of conducting this experiment in a prototype tool that we have called MonaCo[2] [97]. Unlike anything else presented in this thesis MonaCo is unverified and implemented in Ocaml. We report on the initial progress of this project.

Safa and BDDs Safa is a generic Ocaml library. At its core there is the module `SDFA` that captures a symbolic coalgebra by two functions δ and o. From this module safa derives an equivalence checker that constructs a bisimulation. All of this is very similar to our locales, with the exception that Ocaml code of course does not require the user to define a semantics and prove semantic properties about the coalgebraic structure.

The main difference with our algorithm lies in the type of the derivative. Instantiated to formulas safa expects the function $\delta :: \Phi \to \Phi\, bdd$ to be supplied. A binary decision diagram (BDD) of type $\alpha\, bdd$ is a standard compact representation

[1] *S*ymbolic *a*lgorithms for *f*inite *a*utomata
[2] Which is like *MONA* but *co*algebraic.

of a function of type *bool list* $\to \alpha$ as a decision tree with maximal sharing of subtrees and redundancy elimination.[3] BDDs are implemented in Ocaml using hash-consing [42] to guarantee maximal sharing. The safa library conveniently provides all the necessary infrastructure for the manipulation of BDDs including an efficient lifting of binary operators $(\alpha \to \alpha \to \alpha) \to \alpha\, bdd \to \alpha\, bdd \to \alpha\, bdd$ that is required for computing derivatives recursively. This makes it easy to translate our explicit derivatives into their counterpart that employ BDDs. Empirically, using BDDs for left derivatives pays off more than using them for right derivatives. Indeed for right derivatives the explicit version is on par with the BDD version. Our tentative explanation for this phenomenon is that right derivatives are computed only for letters v that have $\bot$ in most of their entries, while for left derivatives always all $2^{|v|}$ possibilities are explored (such that sharing is potentially more often applicable).

Memoization Using BDDs helps, especially with respect to space requirements which are dramatically reduced due to maximal sharing. However, an at least equally important time saver (even more so in conjunction with BDDs) is memoization. A function f is memoizing if it never recomputes a value $f\, x$ twice for the same x. Memoization again relies on hash-consing and is implemented by a (per function global) hash map that caches performed computations for already encountered arguments. Thus, the actual computation is only performed for newly encountered arguments and the computation's result is stored in the hash map.

In our experiments, we found it useful to employ memoization in almost all functions that perform primitive recursion on formulas: left and right derivatives δ and $\eth$, the M2L acceptance test $o_<$, the insertion of restrictions for first-order variables restr and the computation of free variables $\mathcal{V}_1$.

Richer Formula Language Our procedure does not guarantee to produce minimal automata. Instead, we aim for a formula normalization function that is easy to compute (complexity-wise) and produces small automata in practice. Those two aims are contradictory—the smallest automata are produced by normalizing with respect to language equivalence and this is the non-elementary problem we are solving. Quite certainly, there is no optimal resolution to this discrepancy, but there are heuristics that help. One such heuristic is to include conjunction and universal quantification into formulas. Then, negation can be pushed inwards towards the base formulas, leading to a negation normal form.

Another heuristic is to extend base formulas beyond the two we have considered so far. This pays off because for base formulas, in contrast to composite formulas, we are in control of defining minimal (derivative) automata. Indeed, MONA incorporates a wealth of basic formulas. Figure 8.5 shows a large subset of those, their (minimum) semantics and derivatives. Since those syntactic derivative definitions are rather delicate, we preferred to formally verify their characteristic properties.

[3] The more precise term is reduced order binary decision diagrams (ROBDD).

$\varphi :: \Psi$	$I \models \varphi$	$\delta\ v\ \varphi$
$x = n$	$I[x] \neq \{\} \wedge \min(I[x]) = n$	$\begin{cases} \mathsf{T} & \text{if } n = 0 \wedge v[x] \\ x = n - 1 & \text{if } n \neq 0 \wedge \neg v[x] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$x = y + n$	$I[x] \neq \{\} \wedge I[y] \neq \{\} \wedge \min(I[x]) = \min(I[y]) + n$	$\begin{cases} \mathsf{T} & \text{if } n = 0 \wedge v[x] \wedge v[y] \\ x = y + n & \text{if } \neg v[x] \wedge \neg v[y] \\ x = n - 1 & \text{if } n \neq 0 \wedge \neg v[x] \wedge v[y] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X = Y$	$I[X] = I[Y]$	$\begin{cases} X = Y & \text{if } v[X] \leftrightarrow v[Y] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X =_b Y + 1$	$I[X] = (+1) \bullet I[Y] \cup$ `if` b `then` $\{0\}$ `else` $\{\}$	$\begin{cases} X =_{v[Y]} Y + 1 & \text{if } b = v[X] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X = \{\}$	$I[X] = \{\}$	$\begin{cases} X = \{\} & \text{if } \neg v[X] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$\mathsf{sing}\ X$	$\mathsf{sing}\ (I[X])$	$\begin{cases} X = \{\} & \text{if } v[X] \\ \mathsf{sing}\ X & \text{otherwise} \end{cases}$
$X \subseteq Y$	$I[X] \subseteq I[Y]$	$\begin{cases} X \subseteq Y & \text{if } v[X] \longrightarrow v[Y] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$x = \min X$	$I[x] \neq \{\} \wedge I[X] \neq \{\} \wedge \min(I[x]) = \min(I[X])$	$\begin{cases} \mathsf{T} & \text{if } v[x] \wedge v[X] \\ x = \min X & \text{if } \neg v[x] \wedge \neg v[X] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$x = \max X$	$I[x] \neq \{\} \wedge I[X] \neq \{\} \wedge \min(I[x]) = \max(I[X])$	$\begin{cases} X = \{\} & \text{if } v[x] \wedge v[X] \\ x = \max X & \text{if } \neg v[x] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X = Y \cup Z$	$I[X] = I[Y] \cup I[Z]$	$\begin{cases} X = Y \cup Z & \text{if } v[X] \leftrightarrow v[Y] \vee v[Z] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X = Y \cap Z$	$I[X] = I[Y] \cap I[Z]$	$\begin{cases} X = Y \cap Z & \text{if } v[X] \leftrightarrow v[Y] \wedge v[Z] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X = Y \setminus Z$	$I[X] = I[Y] \setminus I[Z]$	$\begin{cases} X = Y \setminus Z & \text{if } v[X] \leftrightarrow v[Y] \wedge \neg v[Z] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X \cap Y = \{\}$	$I[X] \cap I[Y] = \{\}$	$\begin{cases} \mathsf{F} & \text{if } v[X] \wedge v[Y] \\ X \cap Y = \{\} & \text{otherwise} \end{cases}$
$\sum 2^X = n$	$\sum_{i \in I[X]} 2^i = n$	$\begin{cases} \mathsf{F} & \text{if } v[X] \leftrightarrow (n \bmod 2 = 0) \\ \sum 2^X = n/2 & \text{otherwise} \end{cases}$

Figure 8.5: Semantics and derivatives of extended base formulas

Things get even more delicate when defining right derivatives for the minimum semantics as we have learned already on the two earlier base formulas $x < y$ and $x \in X$. There, we have introduced additional formulas to capture the remainders after deriving. Here, we enrich the base formulas from Figure 8.5 with certain flags that encode these additional formulas. Only the six atomic formulas below will need such an adjustment. The added flags are superscripted after the equality signs of the affected formulas. The flags are either of type *nat option* (named i) or of type *bool* (named b or b'). We also redefine (or rather extend) the semantics—the changes affect mostly cases where a first-order variable has been assigned an empty set. The original formulas are obtained by setting $i = \mathsf{None}$ or $b = \bot$. Note that we do not extend the left derivative for formulas with different flag values (such as $i = \mathsf{Some}\ n$ or $b = \top$), since those formulas will never be derived from the left.

$$I \models (x =^i n) = \begin{cases} \min(I[x]) = n & \text{if } I[x] \neq \{\} \\ i = \mathsf{Some}\ 0 & \text{otherwise} \end{cases}$$

$$I \models (x =^i y + n) = \begin{cases} \min(I[x]) = \min(I[y]) + n & \text{if } I[x] \neq \{\} \wedge I[y] \neq \{\} \\ i = \mathsf{Some}\ 0 & \text{if } I[x] = \{\} \wedge I[y] = \{\} \\ \#I = \min(I[y]) + \mathsf{the}\ i & \text{if } I[x] = \{\} \wedge I[y] \neq \{\} \wedge i \neq \mathsf{None} \wedge i \neq \mathsf{Some}\ 0 \\ \bot & \text{otherwise} \end{cases}$$

$$I \models (X =^{b'}_{b} Y + 1) = I[X] \cup \mathtt{if}\ b'\ \mathtt{then}\ \{\#I\}\ \mathtt{else}\ \{\} = (+1) \bullet I[Y] \cup \mathtt{if}\ b\ \mathtt{then}\ \{0\}\ \mathtt{else}\ \{\}$$

$$I \models (x =^b \min X) = \begin{cases} \min(I[x]) = \min(I[X]) & \text{if } I[x] \neq \{\} \wedge I[X] \neq \{\} \\ b & \text{if } I[x] = \{\} \wedge I[X] = \{\} \\ \bot & \text{otherwise} \end{cases}$$

$$I \models (x =^b \max X) = \begin{cases} \min(I[x]) = \max(I[X]) \wedge \neg b & \text{if } I[x] \neq \{\} \wedge I[X] \neq \{\} \\ b & \text{if } I[x] = \{\} \\ \bot & \text{otherwise} \end{cases}$$

$$I \models (\textstyle\sum 2^X =^i n) = \sum_{i \in I[X]} 2^i = n \wedge (\mathtt{case}\ i\ \mathtt{of}\ \mathsf{Some}\ k \Rightarrow \#I = k \mid _ \Rightarrow \top)$$

Figure 8.6 defines the M2L empty model acceptance test $o_<$ and the right derivatives for the extended formulas. We have also verified the expected properties of those operations. The function $\mathsf{ld}\ x$ computes the binary logarithm of $x :: nat$, that is the largest $m :: nat$ such that $2^m \leq x$. In the figure, we use the following abbreviations.

$i \mathbin{\hat{-}} 1 = \mathtt{case}\ i\ \mathtt{of}\ \mathsf{Some}\ (n+1) \Rightarrow \mathsf{Some}\ n \mid _ \Rightarrow \mathsf{None}$
$\vartriangle = \mathsf{None}$
$\lceil x \rceil = \mathsf{Some}\ x$
$\blacktriangle$ = one of $\{\cup, \cap, \setminus\}$
$\triangle$ = one of $\{\wedge, \vee, \lambda b\, b'.\ b \wedge \neg b'\}$ fitting the choice of $\blacktriangle$

$\varphi :: \Psi$	$o_< \varphi$	$\eth\ v\ \varphi$
$x =^i n$	$i = \lceil 0 \rceil$	$\begin{cases} x =^{\lceil n \rceil} n & \text{if } v[x] \\ x =^{i \hat{-} 1} n & \text{otherwise} \end{cases}$
$x =^i y + n$	$i = \lceil 0 \rceil$	$\begin{cases} x =^i y + n & \text{if } n = 0 \wedge \neg v[x] \wedge \neg v[y] \\ x =^{\vartriangle} y + n & \text{if } n = 0 \wedge (v[x] \leftrightarrow v[y]) \vee \\ & \quad n \neq 0 \wedge \neg v[x] \wedge \neg v[y] \wedge i = \lceil 1 \rceil \\ x =^{\lceil 0 \rceil} y + n & \text{if } n = 0 \wedge v[x] \wedge v[y] \vee \\ & \quad n \neq 0 \wedge \neg v[x] \wedge \neg v[y] \wedge i = \lceil 0 \rceil \\ x =^{\lceil n \rceil} y + n & \text{if } n \neq 0 \wedge v[x] \\ x =^{i \hat{-} 1} y + n & \text{otherwise} \end{cases}$
$X = Y$	$\top$	$\begin{cases} X = Y & \text{if } v[X] \leftrightarrow v[Y] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X =_b^{b'} Y + 1$	$b = b'$	$\begin{cases} X =_b^{v[X]} Y + 1 & \text{if } b' = v[Y] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X = \{\}$	$\top$	$\begin{cases} X = \{\} & \text{if } \neg v[X] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$\mathsf{sing}\ X$	$\bot$	$\begin{cases} X = \{\} & \text{if } v[X] \\ \mathsf{sing}\ X & \text{otherwise} \end{cases}$
$X \subseteq Y$	$\top$	$\begin{cases} X \subseteq Y & \text{if } v[X] \longrightarrow v[Y] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$x =^b \min X$	b	$\begin{cases} x =^{\top} \min X & \text{if } v[x] \wedge v[X] \\ x =^b \min X & \text{if } \neg v[x] \wedge \neg v[X] \\ x =^{\bot} \min X & \text{otherwise} \end{cases}$
$x =^b \max X$	b	$\begin{cases} \bot & \text{if } (b \wedge v[x]) \vee (\neg b \wedge \neg v[x] \wedge v[X]) \\ x =^b \max X & \text{if } \neg v[x] \wedge \neg v[X] \\ x =^{v[X]} \max X & \text{otherwise} \end{cases}$
$X = Y \blacktriangle Z$	$\top$	$\begin{cases} X = Y \blacktriangle Z & \text{if } v[X] \leftrightarrow v[Y] \vartriangle v[Z] \\ \mathsf{F} & \text{otherwise} \end{cases}$
$X \cap Y = \{\}$	$\top$	$\begin{cases} \mathsf{F} & \text{if } v[X] \wedge v[Y] \\ X \cap Y = \{\} & \text{otherwise} \end{cases}$
$\sum 2^X =^i n$	$n = 0 \wedge (i = \vartriangle \vee i = \lceil 0 \rceil)$	$\begin{cases} \sum 2^X =^{\lceil \mathsf{ld}\ n \rceil} n - 2^{\mathsf{ld}\ n} & \text{if } i = \vartriangle \wedge v[X] \wedge n \neq 0 \\ \sum 2^X =^i n & \text{if } i = \vartriangle \wedge \neg v[X] \\ \sum 2^X =^{i \hat{-} 1} n - 2^{\mathsf{the}\ i - 1} & \text{if } i \neq \vartriangle \wedge i \neq \lceil 0 \rceil \wedge v[X] \wedge n \geq 2^{\mathsf{the}\ i-1} \\ \sum 2^X =^{i \hat{-} 1} n & \text{if } i \neq \vartriangle \wedge i \neq \lceil 0 \rceil \wedge \neg v[X] \wedge n < 2^{\mathsf{the}\ i-1} \\ \bot & \text{otherwise} \end{cases}$

Figure 8.6: Acceptance test and right derivatives of extended base formulas

The functions δ, $\eth$, and $o_<$ are the core ingredients that are needed for using additional formulas in MonaCo (also in our verified procedure). Therefore, both MonaCo and the formalization, are easily extensible with additional base formulas.

Evaluation MonaCo is not a finished project. Nevertheless, from day one efficiency considerations were guiding its design and were the main reason for giving up the cosy verified environment of a proof assistant.

MonaCo can show that previously considered formulas ψ_n are equivalent to T within less than one second for $n \leq 200$. The fastest verified decision procedure can only achieve this for $n \leq 10$. With MONA it is possible to get even further.

Another benchmark is proving commutativity of the specification of an adder [10]. While the verified procedures have never managed to prove this one within hours, MonaCo requires 0.05 seconds, which is about the same as MONA needs.

We have already mentioned that the derivative based approach can outperform MONA by far on formulas such as $x = 10 \wedge x = 10000$. Indeed, for MONA it takes roughly 40 minutes for proving this example being equivalent to F, while MonaCo gives an instant answer.

Finally, we consider four families of benchmarks (a)–(d) proposed by D'Antoni and Veanes [37]. On the benchmark (a), which states that there exist n different (ordered by $<$) natural numbers for increasing n, MonaCo outperforms MONA and even the customized tool presented by D'Antoni and Veanes. On the other three benchmarks MonaCo's performance is less convincing.

Future Work We outline the next steps for further improvements of MonaCo.

Our algorithm constructs a bisimulation. It is well known that a bisimulation up to congruence and context is often much smaller. Bonchi and Pous [21] successfully employ this technique in practice, outperforming state-of-the-art solvers for NFA equivalence. Bonchi and Pous [21] showed that bisimulation up to congruence and context is an even further improvement over the antichain technique for non-deterministic automata. Either of these techniques will improve the performance of our algorithm (extended to work non-deterministically using partial derivatives) at least for some classes of formulas. Further refinements of the congruence closure, such as congruence modulo integer offsets [86] used in SMT solvers, also bear some potential for improvement. (A more radical measure would be trying to connect MonaCo with an SMT solver directly.)

An alternative to conjoining formulas $\mathsf{FO}\ 0$ to every first-order quantifier, which turned out to be more efficient in MONA, is to work with a three-values semantics [68] of acceptance (where the third value indicates that no $\mathtt{T}$ for a first-order variable has been read yet). In our setting, this would require a change of the underlying final coalgebra (essentially working with Moore machines rather than

with DFAs), and therefore a complication of the syntactic coalgebra. It is not clear whether this complication will pay off in our setting.

Finally, as most research tools, MonaCo desperately needs a proper, carefully designed benchmark suite that ideally will include easy problems, difficult problems, random problems, and real-world problems. For WS1S such a benchmark suite does not exist today to our knowledge. The design of such a suite is time-consuming effort but should prove rewarding for the whole community.

8.6 Case Study: Presburger Arithmetic

To conclude our study of formula derivatives in this thesis we want to explore their applicability to different logics than WMSO. A natural candidate for this endeavor is Presburger arithmetic (PA)—a decidable first-order logic with built-in addition but without multiplication. Below we define this logic formally. For a deeper introduction we refer to the presentation of PA by Boudet and Comon [23], which explores the logic-automaton connection for PA and has also served as a role model for us when devising the derivatives of PA formulas presented below. There also exists an automaton-based formalization of Boudet and Comon's decision procedure for PA in Isabelle/HOL [13].

Note that, as usual, our primary goal is not efficiency. As for WMSO, the derivatives of PA formulas do not yield minimal automata. Hence, we do not expect out procedure to have the optimal doubly exponential complexity [66].

PA formulas are defined inductively by the following datatype.

$$\texttt{datatype}\ \Phi = \mathsf{T} \mid \mathsf{F} \mid \mathsf{Eq}^{int}\ (int\,list)\ int \mid \Phi \vee \Phi \mid \neg\,\Phi \mid \exists\,\Phi$$

Only the base formula $\mathsf{Eq}^{\Delta}\ cs\ x$ deserves an explanation. It represents a Diophantine equation[1] with integer coefficients cs over $|cs|$ natural variables $x_0, \ldots, x_{|cs|}$. The list of values $xs :: nat\,list$ assigned to those variables constitutes an interpretation. The formula's semantics $\models$ requires the scalar product $xs \cdot cs = \sum_i^{|v|} xs[i] \cdot cs[i]$ of the interpretation xs and the coefficients cs to be equal to the given integer x. A further parameter Δ is an additional flag (in the spirit of the various flags from the previous section) that will be useful for right derivatives. In input formulas—the only ones that will be derived by δ—we require $\Delta = 0$. We omit the equations for formulas different from $\mathsf{Eq}^{\Delta}\ cs\ x$ in the definition of $\models$ (and other forthcoming operations), since they are defined exactly in the same way as for WMSO.

$$xs \models \mathsf{Eq}^{\Delta}\ cs\ x = (xs \cdot cs = x - \Delta \cdot 2^{\#xs})$$

[1] Boudet and Comon also include inequalities which we for simplicity leave out.

Formally, Δ connects the Diophantine equation with the intrinsic value $\#xs$ of the interpretation xs, that as for WMSO denotes the length of the interpretation's encoding as a formal word. This standard encoding translates the natural numbers in the interpretation xs into their binary representation of type *bool list* (the least significant bit comes first). Similarly to WMSO, the resulting list of lists of Booleans is then transposed and padded with $\perp$ to have the length $\#xs$ to obtain a formal word over the alphabet of Boolean vectors of length $|xs|$—each row corresponding to one variable. For example, for $xs = [4, 7, 1]$ with $\#cs = 4$ we obtain the following formal word in which the original values can be reconstructed by reading rows in binary.

$$\mathsf{w}_{xs} = \left[\begin{pmatrix}\perp\\\top\\\top\end{pmatrix}, \begin{pmatrix}\perp\\\top\\\perp\end{pmatrix}, \begin{pmatrix}\top\\\top\\\perp\end{pmatrix}, \begin{pmatrix}\perp\\\perp\\\perp\end{pmatrix}\right]$$

The PA language is then unsurprisingly defined as $\mathsf{L}^{\Phi}\ n\ \varphi = \{\mathsf{w}_{xs} \mid xs \models \varphi \wedge |xs| = n\}$. Note that PA also admits a bounded semantics (corresponding to M2L) which appears not to have been studied in the literature. Our formalization proves that it is decidable. As future work, it would be interesting to investigate its complexity properties and usefulness.

For input formulas we define the derivative of the base formula (with Δ restricted to 0) as follows. The definition is rather intuitive—it generates the Diophantine equation whose solutions (that is satisfying interpretations) are the solutions xs of the input equation after losing their least significant bit. The construction of the base automaton by Boudet and Comon is very similar.

$$\delta\ v\ (\mathsf{Eq}^0\ cs\ x) = \mathtt{let}\ y = x - v \cdot cs\ \mathtt{in}\ \mathtt{if}\ y\ \mathsf{mod}\ 2 = 0\ \mathtt{then}\ \mathsf{Eq}^0\ cs\ (y/2)\ \mathtt{else}\ \mathsf{F}$$

Boudet and Comon prove that the values that the last argument y of all word derivatives of $\mathsf{Eq}^0\ cs\ x$ that are different from F can take are bounded as follows: $|y| \leq \max x\ (\sum[|c| \mid c \leftarrow cs])$. This immediately implies that the number of word derivatives of $\mathsf{Eq}^0\ cs\ x$ is finite. For the whole type Φ the number of derivatives is finite only when considering derivatives modulo ACI, as always. In our formal proof we tighten the bound to $\min x\ (-\sum[c \mid c \leftarrow cs,\ c > 0]) \leq y \leq \max x\ (-\sum[c \mid c \leftarrow cs,\ c < 0])$.

As for WMSO, the characteristic property of derivatives is best expressed via the operator CONS, for which we have $w_{(\mathsf{CONS}\ v\ xs)} = v \# \mathsf{w}_{xs}$ assuming $|v| = |xs|$.

```
fun CONS :: bool list → nat list → nat list where
  CONS [] [] = []
  CONS (⊤ # v) (x # xs) = (2 · x + 1) # CONS v I
  CONS (⊥ # v) (x # xs) = (2 · x) # CONS v I
```

Additionally, we require $\#(\mathsf{CONS}\ v\ I) = \#I + 1$. The derivatives of n-wellformed formulas (defined for the new base case by $\mathsf{wf}^{\Phi}\ n\ (\mathsf{Eq}^{\Delta}\ cs\ x) = (|cs| = n)$) are then characterized by the following property which follows by induction.

$$\texttt{theorem}\ \mathsf{wf}^{\Phi}\ n\ \varphi \wedge |xs| = |v| = n \longrightarrow xs \models \delta\ v\ \varphi \leftrightarrow \mathsf{CONS}\ v\ xs \models \varphi$$

As for right derivatives, we need to generate solutions that are the solutions of the input equations after losing their most significant bit. A sound way for doing so is given by the following formula. Here, we see how the parameter Δ is used to accumulate what we have already derived (from the right).

$$\eth\ v\ (\mathsf{Eq}^{\Delta}\ cs\ x) = \mathsf{Eq}^{v \cdot cs + 2 * \Delta}\ cs\ x$$

Unfortunately, when using this definition the parameter Δ can grow arbitrarily large (for example by repeatedly deriving by a vector $v = 1\ \#\ 0^{|vcs|-1}$, assuming that $cs = x\ \#\ xs$ with $x > 0$). However, to obtain a finite number of right derivatives, it is sound to restrict Δ's growth by the same bounds that we have used to prove finiteness of left derivatives. For values of Δ outside of these bounds there is no interpretation xs that would satisfy the formula, hence the latter is equivalent to F in that case.

$$\begin{aligned}
&\eth\ v\ (\mathsf{Eq}^{\Delta}\ cs\ x) = \texttt{let}\ \Gamma = v \cdot cs + 2 * \Delta\ \texttt{in} \\
&\quad \texttt{if}\ \min x\ (-\textstyle\sum[c \mid c \leftarrow cs,\ c > 0]) \le \Gamma \le \max x\ (-\textstyle\sum[c \mid c \leftarrow cs,\ c < 0]) \\
&\quad \texttt{then}\ \mathsf{Eq}^{\Gamma}\ cs\ x \\
&\quad \texttt{else}\ \mathsf{F}
\end{aligned}$$

The right derivatives of PA formulas also satisfy a characteristic property in terms of a SNOC operation on interpretations (for the bounded semantics only) which we omit here. The empty model acceptance test o combines the right derivatives with the empty model acceptance of the bounded semantics $o_<$ in the same way as for WMSO. The predicate $o_<$ is defined as follows.

$$o_<\ (\mathsf{Eq}^{\Delta}\ cs\ x) = (x = \Delta)$$

This definition merely asserts that for $\#xs = 0$ the solutions of the Diophantine equation $xs \cdot cs = x - \Delta \cdot 2^{\#xs}$ only include the solution $xs = 0^{|cs|}$ if the right hand side of the equation is equal to 0.

We obtain the interpretation of our locale for Presburger arithmetic where Σ_n is the list of Boolean vectors of length n and $\langle_\rangle$ is the usual ACI normalization function.

```
interpretation PA : fin_coalg_D where
  Σ = Σ_n
  i φ = ⟨φ⟩
  o φ = o n φ
  d v φ = ⟨δ v φ⟩
  L^σ φ = L^τ φ = L^Φ n φ
  wf^σ φ = wf^τ φ = wf^Φ n φ
```

Using the decision procedure that is generated by this interpretation we can decide the famous *coin problem*: all amounts greater than 8 can be obtained using only coins of value 3 and 5. Formally, this corresponds to the formula $\forall x.\ \exists y\, z.\ 5 \cdot z + 3 \cdot y - x = 8$, which we can write in our syntax as $\forall \exists\ \exists\ \mathsf{Eq}^0\ [5, 3, -1]\ 8$ and prove equivalent to T.

Finally, let us for a change instantiate our $\mathsf{fin_coalgs}_D$ locale with two different coalgebras. Here, we chose the coalgebras for WS1S and PA and thereby compare formulas of WS1S with formulas of PA.

```
interpretation WS1S_PA : fin_coalgs_D where
  A = WS1S
  B = PA
  v ▽ v' = (v = v')
```

Note that second-order variables in WS1S can be basically seen as binary encodings of natural numbers and thus correspond to Presburger variables. The interpretation WS1S_PA defines a procedure that for example can prove that the formulas $\varphi = \sum 2^0 = 42 \wedge \sum 2^1 = 43$ (in common syntax $\sum_{i \in X} 2^i = 42 \wedge \sum_{i \in Y} 2^i = 43$) and $\psi = \mathsf{Eq}^0\ [1, 0]\ 42 \wedge \mathsf{Eq}^0\ [0, 1]\ 43$ (in common syntax $1 \cdot x + 0 \cdot y = 42 \wedge 0 \cdot x + 1 \cdot y = 43$) denote the same regular language, namely the following set.

$$\left\{ \begin{pmatrix} \bot \\ \top \end{pmatrix} \# \begin{pmatrix} \top \\ \top \end{pmatrix} \# \begin{pmatrix} \bot \\ \bot \end{pmatrix} \# \begin{pmatrix} \top \\ \top \end{pmatrix} \# \begin{pmatrix} \bot \\ \bot \end{pmatrix} \# \begin{pmatrix} \top \\ \top \end{pmatrix} \# \begin{pmatrix} \bot \\ \bot \end{pmatrix}^n \mid n :: nat \right\}$$

The rows of all words in this set are binary encodings of 42 and 43, respectively. Note that the exponents in the WS1S formula φ are actually de Bruijn indices.

By the way, in case you'd like to know: yes, it can be proved that if it can be proved that it can't be proved that two plus two is five, then it can be proved that two plus two is five.

— George Boolos (1994)

Chapter 9

Conclusion

We hope to have convinced the reader that Conway was right: being able to compute derivatives is "a skill worth acquiring" [34, p. 43].

9.1 Results

We have presented a framework for deciding equivalence of regular languages, which we have formalized in Isabelle/HOL. The framework is based on the coalgebraic view on formal languages; essentially it computes a bisimulation relation by moving forward via an abstract derivative operation and checks each pair in the bisimulation for consistent acceptance of the empty word.

The framework may be simple, but it constitutes a powerful abstraction layer: we have presented twenty instantiations of the framework summarized in Figure 9.1. Each instantiation yields an executable decision procedure of the symbolic representations of regular languages listed in the second column. All procedures instantiate the abstract derivative operation of the framework with a combination of a concrete derivative operation and a normalization function that ensures that the number of all word derivatives for an input is finite. These choices uniquely characterize the instantiation, which we note down in the third and fourth column. The procedures that use $\langle\!\langle _ \rangle\!\rangle$ or $\langle\!\langle\!\langle _ \rangle\!\rangle\!\rangle$ as the normalization are proved to be partially correct, all others—totally correct.

Concerning the equivalence of regular expressions, we have shown that all the previously published verified decision procedures that work directly with expressions can be obtained as instances of our framework. The unified presentation caters for a meaningful comparison of the performance of the various instances. Marked regular expressions are superior on average but partial derivatives can outperform them in specific cases.

For each of the two semantics of weak monadic-second order logic (WMSO), WS1S and M2L, we have presented two kinds of procedures. The first kind translates

	Equivalence of	Derivative	Normalization	Chapter
$D_\sim$	regular expressions	Brzozowski derivatives	$[_]_\sim$	5.1.1
D	regular expressions	Brzozowski derivatives	$\langle_\rangle$	5.1.1
N	regular expressions	Brzozowski derivatives	$\langle\langle_\rangle\rangle$	5.1.1
P	regular expressions	Partial derivatives	–	5.1.2
PD	regular expressions	Brzozowski derivatives	$\langle\langle\langle_\rangle\rangle\rangle$	5.1.2
A	regular expressions	mark after atom	–	5.2.1
A_2	regular expressions	mark after atom (caching)	–	5.2.1
B	regular expressions	mark before atom	–	5.2.2
B_2	regular expressions	mark before atom (caching)	–	5.2.2
D_Π	Π-extended regular expressions	Π-extended Brzozowski derivatives	$\langle_\rangle$	6.2
P_Π	Π-extended regular expressions	Π-extended Brzozowski derivatives	$\langle\langle\langle_\rangle\rangle\rangle$	6.4
$M2L^D_\Pi$	M2L formulas	Π-extended Brzozowski derivatives	$\langle_\rangle$	7.3
$M2L^P_\Pi$	M2L formulas	Π-extended Brzozowski derivatives	$\langle\langle\langle_\rangle\rangle\rangle$	7.3
$WS1S^D_\Pi$	WS1S formulas	Π-extended Brzozowski derivatives	$\langle_\rangle$	7.3
$WS1S^P_\Pi$	WS1S formulas	Π-extended Brzozowski derivatives	$\langle\langle\langle_\rangle\rangle\rangle$	7.3
M2L	M2L formulas	formula derivatives	$\langle_\rangle$	8.3
WS1S	WS1S formulas	formula derivatives	$\langle_\rangle$	8.3
$M2L'$	M2L formulas (extended base)	formula derivatives (minimum semantics)	$\langle_\rangle$	8.4 & 8.5
$WS1S'$	WS1S formulas (extended base)	formula derivatives (minimum semantics)	$\langle_\rangle$	8.4 & 8.5
PA	PA formulas	PA formula derivatives	$\langle_\rangle$	8.6

Figure 9.1: Formalized decision procedures

logical formulas into regular expressions extended with an additional constructor that models the existential quantifier; the second derives formulas directly. While for M2L the empty word acceptance check is as easy to define as for regular expressions, WS1S requires more work to cancel trailing zeros, which do not affect satisfiability, via right derivatives. Furthermore, we have considered a variation of the encoding of first-order variables that improves on efficiency and explored the usage of derivatives for different logics on the example of Presburger arithmetic.

For empirical experiments with formulas we have used a stronger normalization function than the one given in the figure. The used normalization additionally simplifies Boolean subexpressions and removes needless quantifiers. (In principle this yields even further instantiations of our framework that we have not shown because they do not differ fundamentally from M2L, WS1S, and PA.)

On the one hand, the obtained verified decision procedures for WMSO can be considered elegant toys—implementable only with a few hundred lines of Standard ML (and thus well-suited for formalization) and teachable in class. At this stage, our intention is not to compete with MONA—the state-of-the-art tool for WS1S—which employs thousands of lines of tricky performance optimizations (but still can be outperformed by our procedures on well selected formulas).

On the other hand, being able to safely decide small formulas is already of some value. As an experiment, we randomly generated small formulas and compared the outcome of our verified algorithm with the results produced by MONA and two other (less mature) tools: Toss [46] and dWiNA [41]. As the result, we were able to point the developers of the latter two tools to corner cases where their tools gave a wrong answer [108, 109]. Admittedly, one could have performed the same test without a formalized decision procedure, but then in case of a discrepancy, determining who is right might be difficult with random formulas.

Our unverified but optimized MonaCo tool has stronger ambitions with respect to a performance comparison with MONA. Despite promising initial benchmarks, the tool is in its early days and will take some time to mature.

9.2 Future Work

Here we suggest follow-up projects that are interesting to work on in the future.

Normalization It is slightly disappointing that using a stronger normalization function $\langle\!\langle _ \rangle\!\rangle$, that identifies strictly more expressions than the ACI normalization $\langle _ \rangle$, might lead to non-termination of the algorithm when the normalization is interspersed with derivatives rather than performed only at the comparison time. On the other hand interspersing the derivatives with the normalization is reasonable in practice. It is interesting to investigate what general (and easy to prove) criteria a normalization function must satisfy in order to guarantee termination.

A natural criterion would be to require that $\langle\!\langle \delta\ a\ \langle\!\langle r \rangle\!\rangle \rangle\!\rangle = \langle\!\langle \delta\ a\ r \rangle\!\rangle$ holds. Then it does not matter whether the normalization is performed interspersed with derivatives or at the end (additionally assuming that $\langle\!\langle _ \rangle\!\rangle$ is idempotent). However, it is not clear to us whether the normalizations employed in our procedures satisfy this rather strong property. Indeed, we can relax it by requiring a quasi-order $\leq$, that is a reflexive and transitive relation, on the structures under consideration (for example regular expressions), that fulfills the following properties.

$$\forall r.\ \mathsf{finite}\ \{s \mid s \leq r\}$$
$$\forall r.\ \langle\!\langle r \rangle\!\rangle \leq \langle r \rangle$$
$$\forall a\ r\ s.\ r \leq s \longrightarrow \langle \delta\ a\ r \rangle \leq \langle \delta\ a\ s \rangle$$

Although the existence of such a relation $\leq$ guarantees termination when interspersing the normalization $\langle\!\langle _ \rangle\!\rangle$ with derivatives, it is still not clear to us what relations satisfy the required properties for our different normalizations.

ω-regular languages Since S1S, in which second-order variables quantify over potentially infinite sets of numbers, is also decidable, the question naturally arises whether our ideas are applicable there as well. Some hope for the affirmative answer can be drawn from the work of Esparza and Křetínský [40, 78], who employ something similar to "derivatives of LTL formulas" (but in conjunction with a nonstandard automaton model) for model checking. Furthermore, Ciancia and Venema [32] recast a more standard automaton model on infinite words into the coalgebraic setting.

The same question applies to ω-regular expressions. Here the situation is almost clear thanks to Ciancia and Venema's exposition. To decide equivalence of ω-regular expressions we would proceed to construct a bisimulation via derivatives (which easily generalize to ω-regular expressions). However for each pair of expressions in the bisimulation, instead of checking the consistent acceptance of the empty word, we would actually check equivalence of two regular languages (potentially reusing procedures developed in this thesis) that must be computed from the expressions. This computation is the only missing piece: given an ω-regular expression r, find a regular expression $\mathsf{Loop}\ r$ that denotes the *loop*-language, that is satisfies $\mathsf{L}\ (\mathsf{Loop}\ r) = \{v \mid v^\omega \in \mathsf{L}\ r\}$. Ciancia and Venema show that the loop-language of an ω-regular expression is indeed regular and how to compute a finite automaton accepting it from a given Muller automaton. However, starting from an ω-regular expressions the computation seems to be more contrived.

Regular tree languages Besides WS1S, MONA, as well as dWiNA and Toss, implements a decision procedure for WS2S—a logic over regular trees. An interesting project would be devising derivatives of WS2S formulas.

Again the starting point of this endeavor would be to understand the derivative operations on regular tree expressions [33] first. In joint (unpublished) work with Peter Lammich, we have formally defined derivatives that decompose such expressions in a top-down fashion. While this is enough for matching, equivalence of regular tree expressions can not be decided via top-down derivatives—the core of the problem is that non-deterministic top-down tree automata are strictly more expressive than the deterministic ones. However, for bottom-up automata determinism is not a limitation. Therefore, we are currently investigating a derivative operation that works in a bottom-up fashion.

More speculatively, it would be interesting to study ω-regular tree languages, the associated logic S2S, which is also decidable [98], and its derivatives.

Bibliography

[1] M. Adams. Introducing HOL Zero (extended abstract). In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *ICMS '10*, volume 6327 of *LNCS*, pages 142–143. Springer, 2010.

[2] V. Antimirov. Partial derivatives of regular expressions and finite automata constructions. In E. W. Mayr and C. Puech, editors, *STACS 95*, volume 900 of *LNCS*, pages 455–466. Springer, 1995.

[3] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[4] R. D. Arthan. Some mathematical case studies in ProofPower–HOL. In K. Slind, editor, *TPHOLs 2004 (Emerging Trends)*, pages 1–16, 2004.

[5] A. Asperti. A compact proof of decidability for regular expression equivalence. In L. Beringer and A. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 283–298. Springer, 2012.

[6] A. Asperti, C. S. Coen, and E. Tassi. Regular expressions, au point. *CoRR*, abs/1010.2604, 2010.

[7] A. Ayari and D. Basin. Bounded model construction for monadic second-order logics. In E. A. Emerson and A. P. Sistla, editors, *CAV 2000*, volume 1855 of *LNCS*, pages 99–112. Springer, 2000.

[8] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *MKM 2006*, volume 4108 of *LNCS*, pages 31–43. Springer, 2006.

[9] J. Barwise and L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. Cambridge University Press, 1996.

[10] D. Basin and S. Friedrich. Combining WS1S and HOL. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, 2000.

[11] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *Formal Methods In System Design*, 13:255–288, 1998. Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939.

[12] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In S. Graf and M. I. Schwartzbach, editors, *TACAS 2000*, volume 1785 of *LNCS*, pages 188–203. Springer, 2000.

[13] S. Berghofer and M. Reiter. Formalizing the logic-automaton connection. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs 2009*, volume 5674 of *LNCS*, pages 147–163. Springer, 2009.

[14] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014.

[15] J. C. Blanchette and A. Popescu. Mechanizing the metatheory of sledgehammer. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *FroCoS 2013*, volume 8152 of *LNCS*, pages 245–260. Springer, 2013.

[16] J. C. Blanchette, A. Popescu, and D. Traytel. Cardinals in Isabelle/HOL. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 111–127. Springer, 2014.

[17] J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion. In J. Reppy, editor, *ICFP 2015*, pages 192–204. ACM, 2015.

[18] J. C. Blanchette, A. Popescu, and D. Traytel. Witnessing (co)datatypes. In J. Vitek, editor, *ESOP 2015*, volume 9032 of *LNCS*, pages 359–382. Springer, 2015.

[19] F. Bonchi, M. M. Bonsangue, J. J. M. M. Rutten, and A. Silva. Brzozowski's algorithm (co)algebraically. In R. L. Constable and A. Silva, editors, *Logic and Program Semantics - Essays Dedicated to Dexter Kozen on the Occasion of His 60th Birthday*, volume 7230 of *LNCS*, pages 12–23. Springer, 2012.

[20] F. Bonchi, G. Caltais, D. Pous, and A. Silva. Brzozowski's and up-to algorithms for must testing. In C. Shan, editor, *APLAS 2013*, volume 8301 of *LNCS*, pages 1–16. Springer, 2013.

[21] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In R. Giacobazzi and R. Cousot, editors, *POPL 2013*, pages 457–468. ACM, 2013.

[22] M. M. Bonsangue, J. J. M. M. Rutten, and A. Silva. A Kleene theorem for polynomial coalgebras. In L. de Alfaro, editor, *FoSSaCS 2009*, volume 5504 of *LNCS*, pages 122–136. Springer, 2009.

[23] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *CAAP 1996*, volume 1059 of *LNCS*, pages 30–43. Springer, 1996.

[24] T. Braibant and D. Pous. An efficient Coq tactic for deciding Kleene algebras. In M. Kaufmann and L. Paulson, editors, *ITP 2010*, volume 6172 of *LNCS*, pages 163–178. Springer, 2010.

[25] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[26] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik und Grundl. Math.*, 6:66–92, 1960.

[27] L. Bulwahn. The new Quickcheck for Isabelle: Random, exhaustive and symbolic testing under one roof. In C. Hawblitzel and D. Miller, editors, *CPP 2012*, volume 7679 of *LNCS*, pages 92–108. Springer, 2012.

[28] P. Caron, J. Champarnaud, and L. Mignot. A general framework for the derivation of regular expressions. *RAIRO - Theor. Inf. and Applic.*, 48(3):281–305, 2014.

[29] 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 2011*, volume 6638 of *LNCS*, pages 179–191. Springer, 2011.

[30] A. Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[31] A. Church. A formulation of the simple theory of types. *J. Symb. Logic*, 5(2):56–68, 1940.

[32] V. Ciancia and Y. Venema. Stream automata are coalgebras. In D. Pattinson and L. Schröder, editors, *CMCS 2012*, volume 7399 of *LNCS*, pages 90–108. Springer, 2012.

[33] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[34] J. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall mathematics series. Dover Publications, Incorporated, 2012.

[35] 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*, volume 7086 of *LNCS*, pages 119–134. Springer, 2011.

[36] N. A. Danielsson. Total parser combinators. In P. Hudak and S. Weirich, editors, *ICFP 2010*, pages 285–296. ACM, 2010.

[37] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In S. Jagannathan and P. Sewell, editors, *POPL 2014*, pages 541–554. ACM, 2014.

[38] J. Elgaard, N. Klarlund, and A. Møller. MONA 1.x: new techniques for WS1S and WS2S. In A. J. Hu and M. Y. Vardi, editors, *CAV 1998*, volume 1427 of *LNCS*, pages 516–520. Springer, 1998.

[39] C. C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–51, 1961.

[40] J. Esparza and J. Křetínský. From LTL to deterministic automata: A Safraless compositional approach. In A. Biere and R. Bloem, editors, *CAV 2014*, volume 8559 of *LNCS*, pages 192–208. Springer, 2014.

[41] T. Fiedor, L. Holík, O. Lengál, and T. Vojnar. Nested antichains for WS1S. In C. Baier and C. Tinelli, editors, *TACAS 2015*, LNCS. Springer, 2015. to appear.

[42] J. Filliâtre and S. Conchon. Type-safe modular hash-consing. In A. Kennedy and F. Pottier, editors, *ACM Workshop on ML*, pages 12–19. ACM, 2006.

[43] S. Fischer, F. Huch, and T. Wilke. A play on regular expressions: functional pearl. In P. Hudak and S. Weirich, editors, *ICFP 2010*, pages 357–368. ACM, 2010.

[44] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for NetKAT. In D. Walker, editor, *POPL 2015*, pages 343–355. ACM, 2015.

[45] T. Ganzow. *Definability and Model Checking: The Role of Orders and Compositionality*. PhD thesis, RWTH Aachen University, 2012.

[46] T. Ganzow and L. Kaiser. New algorithm for weak monadic second-order logic on inductive structures. In A. Dawar and H. Veith, editors, *CSL 2010*, volume 6247 of *LNCS*, pages 366–380. Springer, 2010.

[47] W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. *ACM Trans. Comput. Log.*, 13(1):4:1–19, 2012.

[48] E. Giménez and P. Castéran. A tutorial on [co-]inductive types in Coq. `http://www.labri.fr/perso/casteran/RecTutorial.pdf`, 1998.

[49] A. Ginzburg. A procedure for checking equality of regular expressions. *J. ACM*, 14(2):355–362, 1967.

[50] V. M. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16:1–53, 1961.

[51] M. Gordon. From LCF to HOL: a short history. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

[52] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[53] F. Haftmann, A. Krauss, O. Kunčar, and T. Nipkow. Data refinement in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 100–115. Springer, 2013.

[54] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *FLOPS 2010*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

[55] J. Hamza, B. Jobstmann, and V. Kuncak. Synthesis for regular specifications over unbounded domains. In R. Bloem and N. Sharygina, editors, *FMCAD 2010*, pages 101–109. IEEE, 2010.

[56] J. Harrison. HOL Light: A tutorial introduction. In *FMCAD '96*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996.

[57] M. Haslbeck. Verified decision procedures for the equivalence of regular expressions. B.Sc. thesis, Department of Informatics, Technische Universität München, 2013.

[58] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. MONA: Monadic second-order logic in practice. In E. Brinksma, R. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, editors, *TACAS 1995*, volume 1019 of *LNCS*, pages 89–110. Springer, 1995.

[59] R. Hinze. Concrete stream calculus: An extended study. *J. Funct. Program.*, 20(5-6):463–535, 2010.

[60] B. Huffman and O. Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In G. Gonthier and M. Norrish, editors, *CPP 2013*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.

[61] B. Jacobs. A bialgebraic review of deterministic automata, regular expressions and languages. In K. Futatsugi, J. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning, and Computation*, volume 4060 of *LNCS*, pages 375–404. Springer, 2006.

[62] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[63] J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In M. C. Chen, R. K. Cytron, and A. M. Berman, editors, *PLDI 1997*, pages 226–236. ACM, 1997.

[64] C. Kaliszyk and C. Urban. Quotients revisited for Isabelle/HOL. In W. C. Chu, W. E. Wong, M. J. Palakal, and C.-C. Hung, editors, *SAC 2011*, pages 1639–1644. ACM, 2011.

[65] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. Mosel: A sound and efficient tool for M2L(Str). In O. Grumberg, editor, *CAV 1997*, volume 1254 of *LNCS*, pages 448–451. Springer, 1997.

[66] F. Klaedtke. Bounds on the automata size for Presburger arithmetic. *ACM Trans. Comput. Log.*, 9(2), 2008.

[67] N. Klarlund. A theory of restrictions for logics and automata. In N. Halbwachs and D. Peled, editors, *CAV 1999*, volume 1633 of *LNCS*, pages 406–417. Springer, 1999.

[68] N. Klarlund. Relativizations for the logic-automata connection. *Higher-Order and Symbolic Computation*, 18(1-2):79–120, 2005.

[69] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[70] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586, 2002.

[71] S. E. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34, pages 3–42. Princeton University Press, 1956.

[72] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Inf. Comput.*, 110(2):366–390, 1994.

[73] D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Technical Report `http://hdl.handle.net/1813/10173`, Computing and Information Science, Cornell University, March 2008.

[74] D. Kozen and A. Silva. Practical coinduction. Technical report, Cornell University, 2012. `http://hdl.handle.net/1813/30510`.

[75] A. Krauss. Recursive definitions of monadic functions. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *PAR 2010*, volume 43 of *EPTCS*, pages 1–13, 2010.

[76] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, 49:95–106, 2012. published online March 2011.

[77] A. Krauss, C. Sternagel, R. Thiemann, C. Fuhs, and J. Giesl. Termination of isabelle functions via termination of rewriting. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *ITP 2011*, volume 6898 of *LNCS*, pages 152–167. Springer, 2011.

[78] J. Křetínský and J. Esparza. Deterministic automata for the (f, g)-fragment of LTL. In A. Biere and R. Bloem, editors, *CAV 2012*, volume 7358 of *LNCS*, pages 7–22. Springer, 2012.

[79] O. Kunčar. Correctness of Isabelle's cyclicity checker—Implementability of overloading in proof assistants. In X. Leroy and A. Tiu, editors, *CPP 2015*, pages 85–94. ACM, 2015.

[80] O. Kunčar and A. Popescu. A consistent foundation for Isabelle/HOL. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 234–252. Springer, 2015.

[81] A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 341–357. Springer, 2014.

[82] R. McNaughton and H. Yamada. Regular expressions and finite state graphs for automata. *IRE Trans. on Electronic Comput*, EC-9:38–47, 1960.

[83] A. R. Meyer. Weak monadic second order theory of succesor is not elementary-recursive. In R. Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154. Springer, 1975.

[84] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: A functional pearl. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP 2011*, pages 189–195. ACM, 2011.

[85] N. Moreira, D. Pereira, and S. M. de Sousa. Deciding regular expressions (in-)equivalence in Coq. In W. Kahl and T. Griffin, editors, *RAMiCS 2012*, volume 7560 of *LNCS*, pages 98–113. Springer, 2012.

[86] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.

[87] T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014. `http://www.in.tum.de/~nipkow/Concrete-Semantics`.

[88] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[89] T. Nipkow and D. Traytel. Unified decision procedures for regular expression equivalence. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 450–466. Springer, 2014.

[90] T. Nipkow and D. Traytel. Unified decision procedures for regular expression equivalence. In *Archive of Formal Proofs*. 2014. `http://afp.sf.net/entries/Regex_Equivalence.shtml`.

[91] A. Okhotin. The dual of concatenation. *Theor. Comput. Sci.*, 345(2-3):425–447, 2005.

[92] S. Owens, J. H. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, 2009.

[93] S. Owre and H. Rueß. Integrating WS1S with PVS. In E. A. Emerson and A. P. Sistla, editors, *CAV 2000*, volume 1855 of *LNCS*, pages 548–551. Springer, 2000.

[94] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Logic Comp.*, 7(2):175–204, 1997.

[95] D. Pous. Symbolic algorithms for finite automata (safa). `https://opam.ocaml.org/packages/safa/safa.1.3/`, 2014.

[96] D. Pous. Symbolic algorithms for language equivalence and Kleene algebra with test. In D. Walker, editor, *POPL 2015*, pages 357–368. ACM, 2015.

[97] D. Pous and D. Traytel. MonaCo: Symbolic/coalgebraic algorithms for WS1S. `https://bitbucket.org/dpous/monaco`, 2015.

[98] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Tran. of the AMS*, 141:1–35, 1969.

[99] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In D. Sangiorgi and R. de Simone, editors, *CONCUR 1998*, volume 1466 of *LNCS*, pages 194–218. Springer, 1998.

[100] J. J. M. M. Rutten. Universal coalgebra: A theory of systems. *Theor. Comput. Sci.*, 249:3–80, 2000.

[101] N. Schaffroth. A specification-based testing tool for Isabelle's ML environment. B.Sc. thesis, Department of Informatics, Technische Universität München, 2013.

[102] S. Shelah. The monadic theory of order. *Ann. Math.*, 102(3):379–419, 1975.

[103] A. Silva. *Kleene Coalgebra*. PhD thesis, Radboud University Nijmegen, 2010.

[104] A. Silva. A short introduction to the coalgebraic method. *ACM SIGLOG News*, 2(2):16–27, 2015.

[105] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In A. V. Aho, A. Borodin, R. L. Constable, R. W. Floyd, M. A. Harrison, R. M. Karp, and H. R. Strong, editors, *STOC 1973*, pages 1–9. ACM, 1973.

[106] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 389–455. Springer, 1997.

[107] B. A. Trakhtenbrot. Finite automata and monadic second order logic. *Sib. Math. J.*, 3(1):103–131, 1962.

[108] D. Traytel. [dWiNA Issue #1] Wrong results for simple formulas, 14 Jan 2015. Archived at `https://github.com/Raph-Stash/dWiNA/issues/1`.

[109] D. Traytel. [Toss-devel] Toss deciding MSO, 14 Jan 2015. Archived at `http://sourceforge.net/p/toss/mailman/message/33232473/`.

[110] D. Traytel. A codatatype of formal languages. In *Archive of Formal Proofs*. 2013. `http://afp.sf.net/entries/Coinductive_Languages.shtml`.

[111] D. Traytel. A coalgebraic decision procedure for WS1S. In S. Kreutzer, editor, *CSL 2015*, volume 41 of *LIPIcs*, pages 487–503. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 2015.

[112] D. Traytel. Derivatives of logical formulas. In *Archive of Formal Proofs*. 2015. `http://afp.sf.net/entries/Formula_Derivatives.shtml`.

[113] D. Traytel and T. Nipkow. Verified decision procedures for MSO on words based on derivatives of regular expressions. In G. Morrisett and T. Uustalu, editors, *ICFP 2013*, pages 3–12. ACM, 2013.

[114] D. Traytel and T. Nipkow. Decision procedures for MSO on words based on derivatives of regular expressions. In *Archive of Formal Proofs*. 2014. `http://afp.sf.net/entries/MSO_Regex_Equivalence.shtml`.

[115] D. Traytel and T. Nipkow. Verified decision procedures for MSO on words based on derivatives of regular expressions. *J. Funct. Program.*, 25, 2015. Extended version of [113].

[116] D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic—Category theory applied to theorem proving. In *LICS 2012*, pages 596–605. IEEE Computer Society, 2012.

[117] M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs 1999*, volume 1690 of *LNCS*, pages 167–184. Springer, 1999.

[118] C. Wu, X. Zhang, and C. Urban. A formalisation of the Myhill–Nerode theorem based on regular expressions (Proof pearl). In M. Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *ITP 2011*, volume 6898 of *LNCS*, pages 341–356. Springer, 2011.

[119] C. Wu, X. Zhang, and C. Urban. A formalisation of the Myhill-Nerode theorem based on regular expressions. *J. Automated Reasoning*, 52:451–480, 2014.